Джей Макгаврен

Head First Изучаем Ruby

Короткий код может быть эффективным

Даже тяжелую работу проще делать с блоками

Освой стандартную библиотеку Ruby

Избегай глупых ОО-ошибок

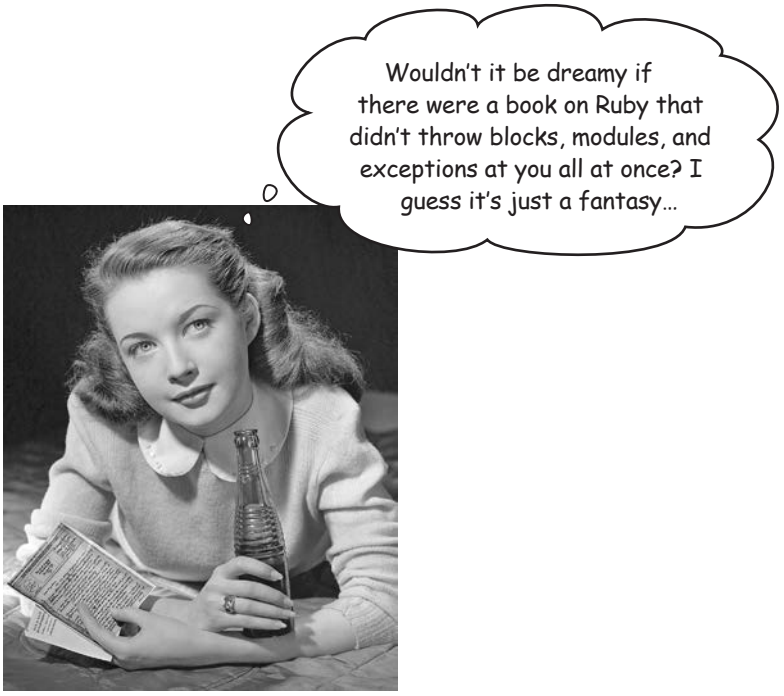
Поломай мозг над 40 упражнениями

Познакомь мир со своими веб-приложениями

O'REILLY®

ПИТЕР®

Head First Ruby



Jay McGavren

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Head First
Изучаем Ruby

Джей Макгаврен

25 лет ПИТЕР®
Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск
2016

Дж. Макгаврен

Head First. Изучаем Ruby

Серия «Head First O'Reilly»

Перевел с английского *Е. Матвеев*

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*Н. Римицан*
Художник	*С. Заматевская*
Корректор	*С. Беляева*
Верстка	*Н. Лукьянова*

ББК 32.988.0-018
УДК 004.738.5

Макгаврен Дж.

М15 Head First. Изучаем Ruby. — СПб.: Питер, 2016. — 528 с.: ил. — (Серия «Head First O'Reilly»).

ISBN 978-5-496-02278-1

Вам интересно, почему буквально все вокруг заговорили о языке Ruby? Спросите себя прямо: вам нравится работать эффективно? Неужели многочисленные компиляторы, библиотеки, классы, которыми грузят вас другие языки программирования, приближают вас к решению конкретной задачи, восхищению коллег и толпе счастливых заказчиков? Вы хотите, чтобы язык программирования занимался техническими подробностями вместо вас? Тогда бросайте рутинную работу и приступайте к решению конкретных задач, а язык Ruby сделает за вас все остальное.

Как и все книги серии Head First, книга «Изучаем Ruby» использует активный подход к обучению, выходя за рамки сухих, абстрактных объяснений и справочников. Вас не только научат языку Ruby, но и помогут вашей программистской звезде ярко воссиять на небосклоне. Вы освоите основы языка и продвинутые возможности Ruby, такие как блоки, объекты, методы, классы и регулярные выражения. С улучшением ваших навыков задачи будут усложняться, и вы перейдете к таким темам, как обработка исключений, модули, подмешанные классы и метапрограммирование.

12+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-1449372651 англ.
ISBN 978-5-496-02278-1

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), 3, литер А, пом. 7Н.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12.000 — Книги печатные профессиональные, технические и научные.

Подписано в печать 20.05.16. Формат 84×108/16. Бумага писчая. Усл. п. л. 55,440. Тираж 1000. Заказ 0000.

Отпечатано в соответствии с предоставленными материалами в ООО «ИПК Парето-Принт».
170546, Тверская область, Промышленная зона Боровлево-1, комплекс № 3А, www.pareto-print.

Автор

Джей Макгаврен

Джей Макгаврен занимался автоматизацией деятельности компании, работающей в области гостиничного обслуживания, когда коллега показал ему книгу *Programming Perl* (так называемая «книга с верблюдом»). Джей мгновенно стал фанатом Perl, потому что ему понравилось писать код, не дожидаясь, пока группа разработчиков из 10 человек настроит систему сборки. Заодно у него родилась безумная идея когда-нибудь самому написать техническую книгу.

В 2007 году, когда развитие Perl зашло в тупик, Джей стал искать новый интерпретируемый язык. Ruby победил — благодаря своей сильной объектной ориентации, превосходной поддержке и невероятной гибкости. С тех пор он использовал Ruby в работе над двумя игровыми библиотеками, в проекте в области искусства, а также занимался независимой разработкой с использованием Ruby on Rails. С 2011 года он работал в области интернет-обучения разработчиков.

Вы можете читать Джея в Твиттере по адресу *https://twitter.com/jaymcgavren* или посетить его персональный сайт *http://jay.mcgavren.com*.

Содержание (сводка)

Содержание (настоящее)

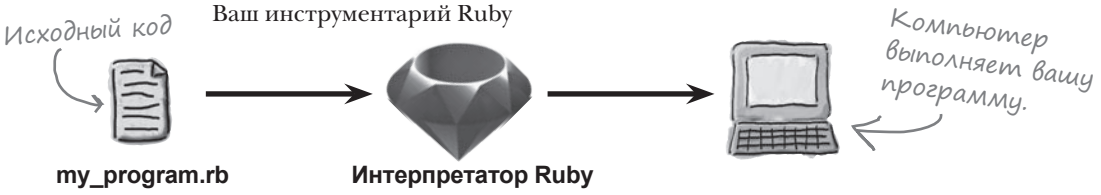
Введение

Ваш мозг думает о Ruby. Вы сидите за книгой и пытаетесь что-нибудь выучить, но ваш мозг считает, что вся эта писанина не нужна. Ваш мозг говорит: «Выгляни в окно! На свете есть более важные вещи, например сноуборд». Как заставить мозг думать, что ваша жизнь действительно зависит от Ruby?

1 Как сделать больше меньшими усилиями

Программируйте так, как вам удобно

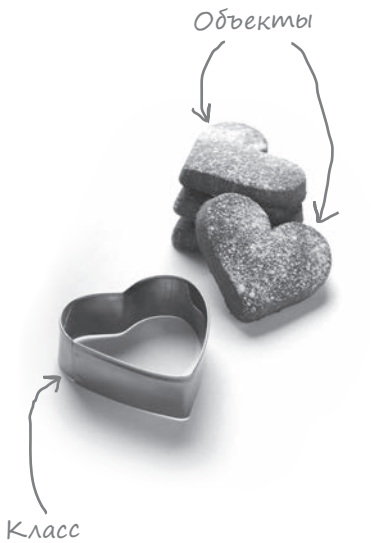
Вас интересует, почему вокруг столько говорят о языке Ruby и подойдет ли он вам? Ответьте на простой вопрос: ***вам нравится работать эффективно?*** Кажется ли вам, что с многочисленными компиляторами, библиотеками, файлами классов и комбинациями клавиш других языков вы действительно становитесь ближе к **готовому продукту**, **восхищению коллег** и **толпе счастливых заказчиков**? Вы хотите, чтобы язык программирования **занимался техническими подробностями** за вас? Если вам хочется перестать писать рутинный код и *работать над задачей*, то язык Ruby — для вас. Ruby позволит вам **сделать больше с меньшим объемом кода.**



2 Методы и классы

Наводим порядок

В вашей работе кое-чего не хватало. Да, вы вызывали методы и создавали объекты, как настоящий профессионал. Но при этом вы могли вызывать только те методы и создавать только те виды объектов, которые были определены за вас в Ruby. Теперь ваша очередь. В этой главе вы научитесь создавать *свои* методы, а также свои **классы** — своего рода «шаблоны» для создания новых объектов. *Вы сами решаете*, как будут выглядеть объекты, созданные на базе вашего класса. **Переменные экземпляра** определяют, какая информация *хранится* в ваших объектах, а **методы экземпляра** определяют, что эти объекты *делают*. А самое главное — вы узнаете, как определение собственных классов *упрощает чтение и сопровождение вашего кода*.

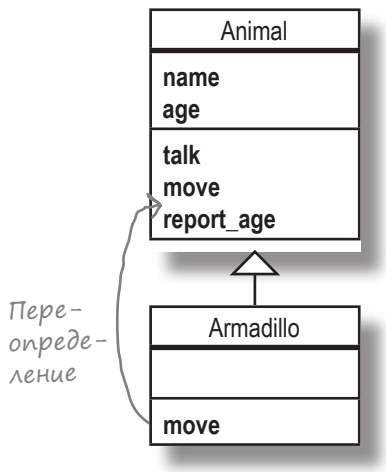


3 Наследование

С родительской помощью

Столько повторений! Новые классы, представляющие разные типы машин или животных, удобны — это правда. Но ведь вам придется *копировать методы экземпляра из класса в класс*. И эти копии будут вести самостоятельную жизнь — одни будут работать нормально, в других могут появиться ошибки. Разве классы создавались не для того, чтобы *упростить* сопровождение кода?

В этой главе вы научитесь применять **наследование**, чтобы ваши классы могли использовать *одни и те же* методы. Меньше дубликатов — меньше проблем с сопровождением!



4 Инициализация экземпляров

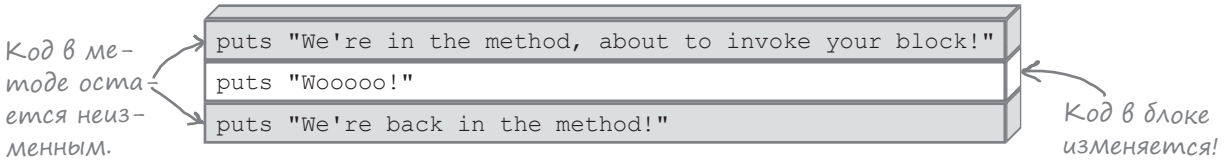
Хороший старт — половина дела

Пока что ваш класс напоминает бомбу с часовым механизмом. Каждый экземпляр, созданный вами, начинается «с пустого места». Если вы вызовете методы экземпляра до того, как в нем появятся данные, то может произойти ошибка, приводящая к аварийному завершению программы. В этой главе мы представим пару способов создания объектов, которые можно сразу безопасно использовать. Начнем с метода `initialize`, который позволяет передать набор аргументов для заполнения данных объекта *на момент его создания*. Затем мы покажем, как написать **методы класса**, которые позволяют **еще** проще создавать и инициализировать объекты.

5 Массивы и блоки

Лучше, чем цикл

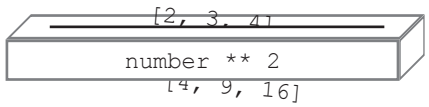
Очень многие задачи из области программирования связаны с обработкой списков. Списки адресов. Списки телефонных номеров. Списки продуктов. Мац, создатель языка Ruby, знал об этом. Поэтому он *основательно* потрудился над тем, чтобы работать со списками в Ruby было *действительно просто*. Сначала он поработал над тем, чтобы **массивы**, используемые для работы со списками в Ruby, обладали множеством *мощных методов* для выполнения практически любых операций. Затем он осознал, что написание кода для *перебора всех элементов списка* для выполнения некоторой операции с каждым элементом — утомительная рутинная работа, которую программистам приходится выполнять *очень часто*. Поэтому он добавил в язык **блоки**, благодаря которым код перебора стал лишним. Что же это такое — блок? Сейчас узнаете...

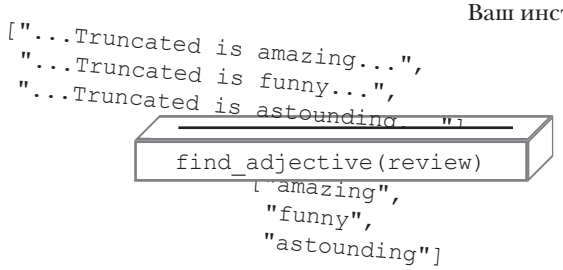


6 Возвращаемые значения блоков

Как будем обрабатывать?

Вы видели лишь малую часть возможностей блоков. До настоящего момента *методы* передавали данные *блоку* и ожидали, что блок сделает все необходимое. Однако *блок* также может возвращать данные *методу*. Эта возможность позволяет методу получать *команды* от блока, позволяя ему выполнять более содержательную работу. Методы, рассмотренные в этой главе, получают *большие и сложные* коллекции и используют **возвращаемые значения блоков** для сокращения их размера.





7 Хеши

Пометка данных

Хранить данные в одной большой куче удобно... До тех пор, пока вам не потребуется в них что-нибудь найти. Вы уже видели, как создать коллекцию объектов с использованием *массива*. Вы уже видели, как обработать *каждый элемент* массива и как *найти нужные* элементы. В обоих случаях мы начинаем от начала массива и *проверяем каждый отдельный объект.* Вы уже видели методы, получающие большие коллекции в параметрах. Вам уже известно, какие проблемы это создаст: вызовы методов требуют передачи большой, *запутанной коллекции аргументов,* для которой вам нужно помнить точный порядок. А не существует ли коллекции, у которой *все данные* уже снабжены *метками*? Тогда вы могли бы *быстро найти* нужные элементы! В этой главе рассматриваются **хеши** Ruby, предназначенные именно для этого.

Массив

Хеш

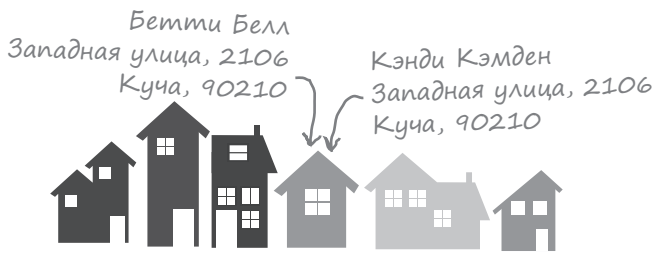
8 Ссылки

Путаница с сообщениями

Вы когда-нибудь отправляли сообщения не тому контакту?

Наверное, вам пришлось изрядно потрудиться, чтобы разобраться с возникшей путаницей. *Объекты Ruby* очень похожи на *контакты* в адресной книге, и *вызов методов* для них напоминает *отправку сообщений*. Если содержимое вашей адресной книги будет *перепутано*, вы рискуете отправить сообщение *не тому объекту*. В этой главе вы научитесь *распознавать такие ситуации* и узнаете, как снова *наладить работу ваших программ*.

Западная улица в реальности



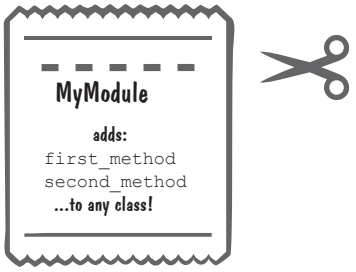
Западная улица по адресной книге Артура

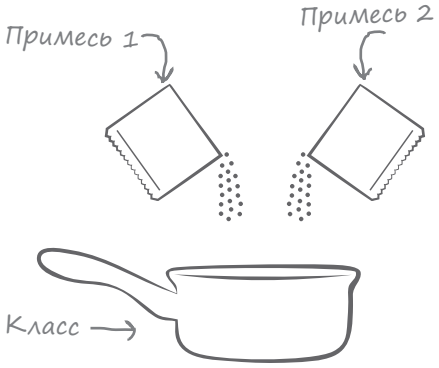
9 Примеси

Аккуратный выбор

У наследования есть свои ограничения. Наследовать методы можно только от одного класса. Но что, если вам понадобилось использовать *некоторое поведение* в совершенно разных классах? Представьте методы для запуска цикла зарядки и получения информации о текущем уровне заряда — такие методы могут использоваться телефонами, электродрелями и электромобилями. И вы готовы создать *один* суперкласс для всех *этих* устройств? (Не пытайтесь, ничем хорошим это не кончится.) Или методы запуска и остановки двигателя. Конечно, эти методы могут пригодиться дрели и автомобилю, но не телефону!

В этой главе рассматриваются **модули** и **примеси** — мощный механизм *группировки методов* и их последующего включения *только в те классы, для которых они актуальны*.



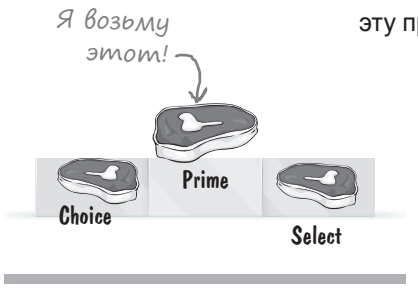


10

Comparable и enumerable

Готовые примеси

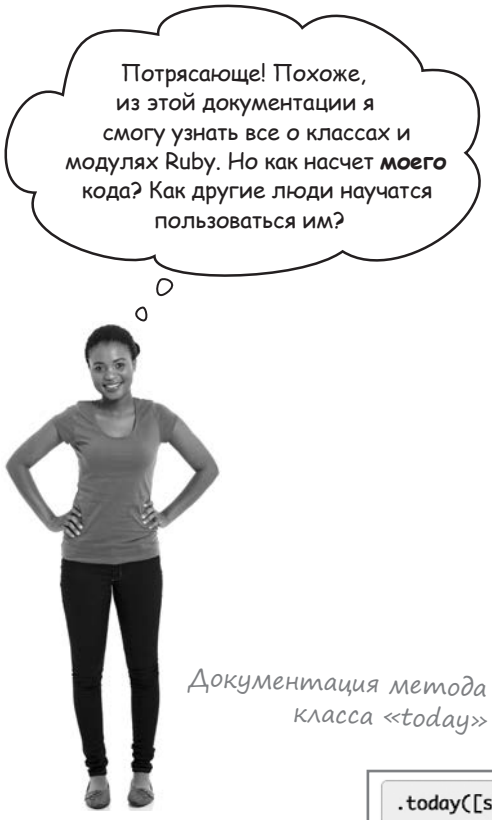
Из предыдущей главы вы уже знаете, что примеси полезны. Но вы еще не видели их истинную мощь. В стандартную библиотеку Ruby входят две совершенно *потрясающие* примеси. Первая, `Comparable`, используется для сравнения объектов. Мы уже использовали такие операторы, как <, > и ==, с числами и строками, а с `Comparable` сможете использовать их с *вашими* классами. Вторая примесь, `Enumerable`, применяется при работе с коллекциями. Помните замечательные методы `find_all`, `reject` и `map`, которые мы использовали с массивами? Так вот, они были предоставлены `Enumerable`. Но это лишь малая часть возможностей `Enumerable`! И разумеется, вы можете включать эту примесь в *свои* классы. Хотите узнать, как это делается? Читайте дальше!



11 Документация

Читайте документацию

В книге не хватит места, чтобы рассказать все о Ruby. Есть старая поговорка: «Дайте человеку рыбу, и он будет сыт один день. Научите его ловить рыбу, и он будет сыт всю жизнь». До сих пор мы *давали вам рыбу*: показали, как использовать некоторые классы и модули Ruby. Однако существуют десятки других классов и модулей, которые могут пригодиться в вашей работе, но нам не хватит места, чтобы описать их в книге. Пришло время *научить вас ловить рыбу*. Существует превосходная бесплатная **документация** по всем классам, модулям и методам Ruby. Вам просто нужно знать, где ее найти и как ее интерпретировать. Именно об этом пойдет речь в этой главе.



Документация метода класса «today»

```
.today([start = Date::ITALY]) ⇒ Object

Date.today #=> #<Date: 2011-06-11 ..>

Creates a date object denoting the present day.
```

```
#year ⇒ Integer

Returns the year.
```

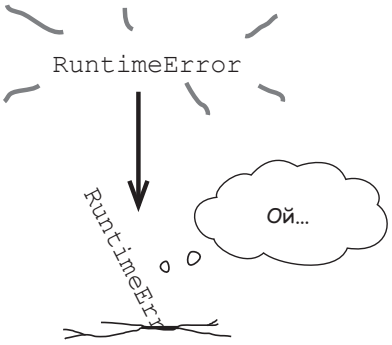
Документация метода экземпляра «year».

12 Исключения

Непредвиденное препятствие

В реальном мире порой происходит нечто неожиданное. Кто-то удаляет файл, который пытается загрузить ваша программа, или сервер, с которым программа пытается связаться, выходит из строя. Вы можете проверять такие исключительные ситуации в своем коде, но такие проверки перемешиваются с кодом, выполняемым в ходе нормальной работы. (И в результате возникает жуткая мешанина, в которой невозможно разобраться.)

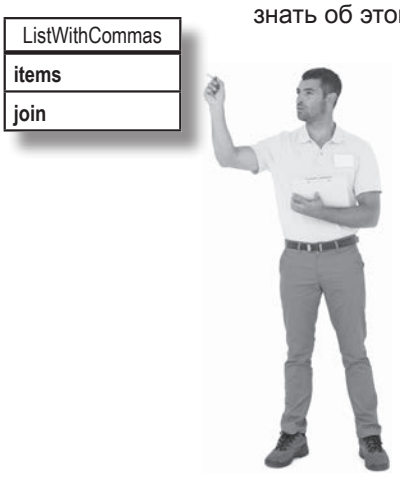
Эта глава посвящена средствам обработки исключений в Ruby, которые позволяют писать код для обработки аварийных ситуаций и размещать его отдельно от нормального кода.



13 Модульное тестирование

Контроль качества кода

А вы уверены, что ваша программа работает правильно? Действительно уверены? Прежде чем передавать новую версию пользователям, вы, вероятно, опробовали новые функции и убедились в том, что они работают. Но проверили ли вы *старые* функции, чтобы убедиться, что они не перестали работать? *Все* старые функции? Если этот вопрос не дает вам покоя, значит, вашей программе необходимо автоматизированное тестирование. Автоматизированные тесты гарантируют, что основные компоненты программы продолжают правильно работать даже после изменения кода. **Модульные тесты** — самая распространенная, самая важная разновидность автоматизированных тестов. В поставку Ruby входит библиотека **MiniTest**, предназначенная для модульного тестирования. В этой главе вы узнаете все, что действительно необходимо знать об этой библиотеке!



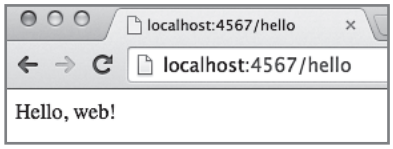
14

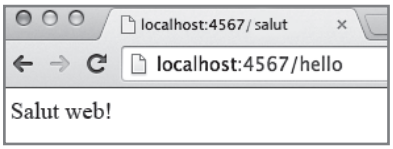
Веб-приложения

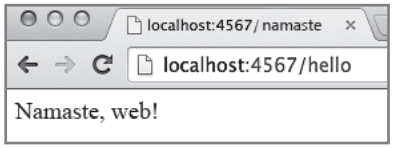
На раздаче HTML

На дворе XXI век. Пользователи требуют веб-приложения. И Ruby поможет вам в этом! Существуют библиотеки, которые помогут вам развернуть собственные веб-приложения и обеспечить доступ к ним из любого браузера. Итак, в двух последних главах этой книги мы покажем вам, как построить полноценное веб-приложение.

Прежде всего вам понадобится Sinatra — независимая библиотека для создания веб-приложений. Но не беспокойтесь, мы научим вас использовать программу RubyGems (включенную в поставку Ruby) для автоматизации загрузки и установки библиотек! Затем будет рассмотрена разметка HTML — ровно столько, сколько необходимо для создания собственных веб-страниц. И конечно, мы покажем, как предоставить эти страницы браузеру!



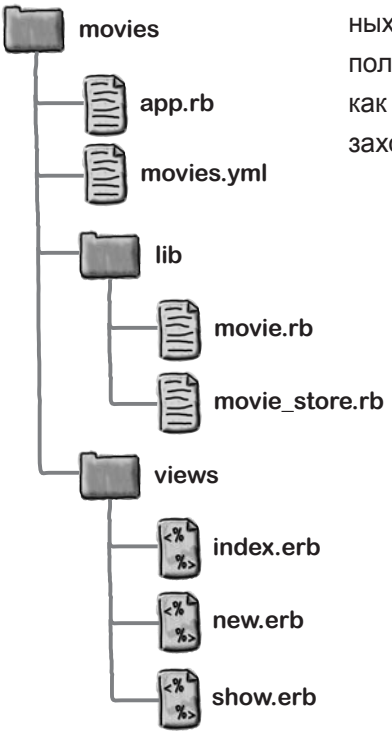




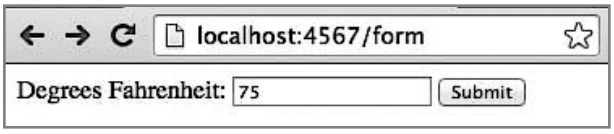
15 Сохранение и загрузка данных

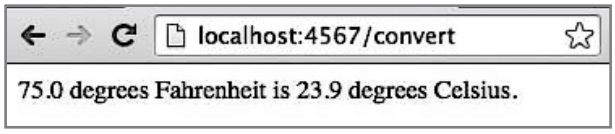
Сбережения пользователя

Сейчас наше веб-приожение просто выбрасывает данные, введенные пользователем. Мы создали форму, на которой пользователь *вводит* данные. Пользователь ожидает. что данные будут *сохранены*, чтобы их можно было *прочитать* и *вывести* позднее. Но сейчас ничего подобного не происходит! Все введенные данные просто *пропадают*.



В последней главе книги приложение будет подготовлено к сохранению данных, введенных пользователем. Мы покажем, как настроить приложение для получения данных формы, как преобразовать эти данные в объекты Ruby, как сохранить их в файле и как загрузить нужный объект, когда пользователь захочет увидеть его. Готовы? Так доделаем наше приложение!



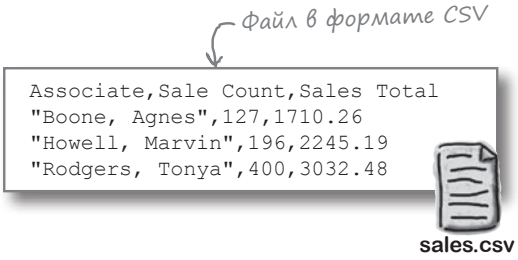


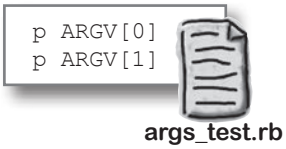
Приложение. Оставшиеся темы

Десять основных тем (не рассмотренных в книге)

Мы прошли долгий путь, и книга почти закончена. Мы будем скучать по вам, но было бы неправильно расставаться и отпускать вас в самостоятельное путешествие без еще *нескольких* напутственных слов. Нам при всем желании не удалось бы уместить все, что вам еще нужно знать о Ruby, на этих страницах… (Вообще-то сначала мы *уместили* все необходимое, уменьшив размер шрифта в 25 000 раз. Все поместилось, но текст было не прочитать без микроскопа, поэтому материал пришлось изрядно сократить.) Но мы оставили все самое лучшее для этого приложения.

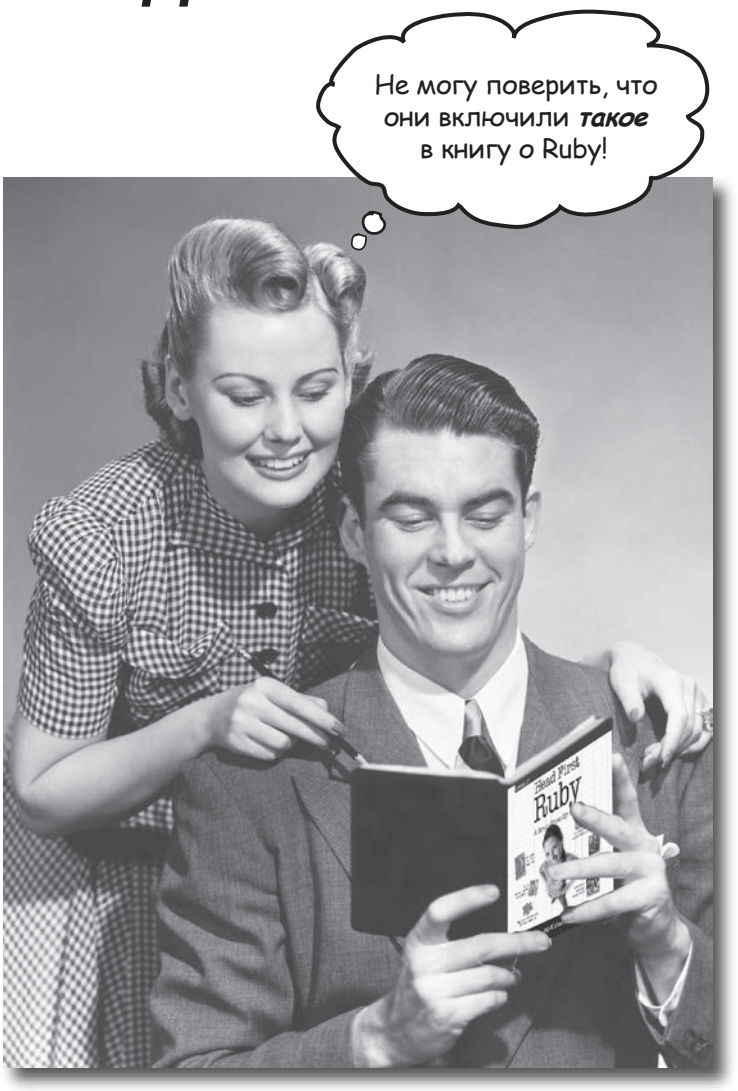
И это уже *действительно* конец книги!






```
File Edit Window Help
$ ruby args_test.rb hello terminal
"hello"
"terminal"
```

Введение



В этом разделе мы ответим на важный вопрос:
«Зачем они включили ТАКОЕ в книгу о Ruby?»

Для кого написана эта книга?

Если вы ответите «да» на ***все*** следующие вопросы...

1. В вашем распоряжении имеется компьютер с текстовым редактором?

2. Вы хотите изучить язык программирования, с которым процесс разработки становится **простым** и **производительным**?

3. Вы предпочитаете **оживленную беседу сухим, скучным академическим лекциям**?

...то эта книга для вас.

Кому эта книга не подойдет?

Если вы ответите «да» на ***любой*** из следующих вопросов:

1. Вы **абсолютно** не разбираетесь в компьютерах?

   (Быть специалистом не обязательно, но вы должны понимать, что такое файлы и папки, уметь запустить терминальное приложение и пользоваться простым текстовым редактором.)

2. Вы супер-пупер-разработчик, которому нужен ***справочник***?

3. Вы **боитесь попробовать что-нибудь новое**? Скорее пойдете к зубному врачу, чем наденете полосатое с клетчатым? Считаете, что техническая книга, в которой наследование объясняется на примере броненосцев, серьезной быть не может?

...эта книга *не* для вас.

[Заметка от отдела продаж: вообще-то эта книга для любого, у кого есть деньги.]

Мы знаем, о чем вы думаете

«Разве серьезная книга по программированию на Ruby может быть *такой?*»

«И почему здесь столько рисунков?»

«Можно ли так чему-нибудь *научиться?*»

И мы знаем, о чем думает ваш мозг

Мозг жаждет новых впечатлений. Он постоянно ищет, анализирует, *ожидает* чего-то необычного. Он так устроен, и это помогает нам выжить.

Как наш мозг поступает со всеми обычными, повседневными вещами? Он всеми силами пытается оградиться от них, чтобы они не мешали его *настоящей* работе — сохранению того, что действительно *важно*. Мозг не считает нужным сохранять скучную информацию. Она не проходит фильтр, отсекающий «очевидно несущественное».

Но как же мозг *узнает*, что важно? Представьте, что вы выехали на прогулку и вдруг прямо перед вами появляется тигр. Что происходит в вашей голове и теле?

Активизируются нейроны. Вспыхивают эмоции. Происходят химические реакции. И тогда ваш мозг понимает...

Конечно, это важно! Не забывать!

А теперь представьте, что вы находитесь дома или в библиотеке в теплом, уютном месте, где тигры не водятся. Вы учитесь — готовитесь к экзамену. Или пытаетесь освоить сложную техническую тему, на которую вам выделили неделю... максимум десять дней.

И тут возникает проблема: ваш мозг пытается оказать вам услугу. Он старается сделать так, чтобы на эту *очевидно* несущественную информацию не тратились драгоценные ресурсы. Их лучше потратить на что-нибудь важное. На тигров, например. Или на то, что к огню лучше не прикасаться. Или что на лыжах не стоит кататься в футболке и шортах.

Нет простого способа сказать своему мозгу: «Послушай, мозг, я тебе, конечно, благодарен, но какой бы скучной ни была эта книга и пусть мой датчик эмоций сейчас на нуле, я *хочу* запомнить то, что здесь написано».

Эта книга для тех, кто хочет учиться

Как мы что-то узнаем? Сначала нужно это «что-то» *понять*, а потом *не забыть*. Затолкать в голову побольше фактов недостаточно. Согласно новейшим исследованиям в области когнитивистики, нейробиологии и психологии обучения, для *усвоения материала* требуется что-то большее, чем простой текст на странице. Мы знаем, как заставить ваш мозг работать.

Основные принципы серии Head First:

Наглядность. Графика запоминается лучше, чем обычный текст, и значительно повышает эффективность восприятия информации (до 89 % по данным исследований). Кроме того, материал становится более понятным. **Текст размещается на рисунках**, к которым он относится, а не под ними или на соседней странице — и вероятность успешного решения задач, связанных с текущим материалом, повышается *вдвое*.

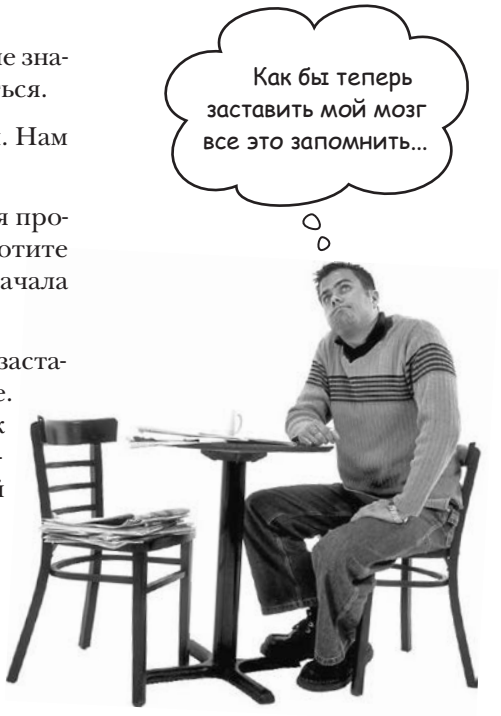
Разговорный стиль изложения. Недавние исследования показали, что при разговорном стиле изложения материала (вместо формальных лекций) улучшение результатов на итоговом тестировании достигает 40 %. Рассказывайте историю, вместо того чтобы читать лекцию. Не относитесь к себе слишком серьезно. Что скорее привлечет *ваше* внимание: занимательная беседа за столом или лекция?

Активное участие читателя. Пока вы не начнете напрягать извилины, в вашей голове ничего не произойдет. Читатель должен быть заинтересован в результате; он должен решать задачи, формулировать выводы и овладевать новыми знаниями. А для этого необходимы упражнения и каверзные вопросы, в решении которых задействованы оба полушария мозга и разные чувства.

Привлечение (и сохранение) внимания читателя. Ситуация, знакомая каждому: «Я очень хочу изучить это, но засыпаю на первой странице». Мозг обращает внимание на интересное, странное, притягательное, неожиданное. Изучение сложной технической темы не обязано быть скучным. Интересное узнается намного быстрее.

Обращение к эмоциям. Известно, что наша способность запоминать в значительной мере зависит от эмоционального сопереживания. Мы запоминаем то, что нам небезразлично. Мы запоминаем, когда что-то *чувствуем*. Нет, сантименты здесь ни при чем: речь идет о таких эмоциях, как удивление, любопытство, интерес и чувство «Да я крут!» при решении задачи, которую окружающие считают сложной — или когда вы понимаете, что разбираетесь в теме *лучше*, чем всезнайка Боб из технического отдела.

Метапознание: наука о мышлении



Если вы действительно хотите быстрее и глубже усваивать новые знания — задумайтесь над тем, как вы задумываетесь. Учитесь учиться.

Мало кто из нас изучает теорию метапознания во время учебы. Нам *положено* учиться, но нас редко этому *учат*.

Но раз вы читаете эту книгу, то, вероятно, вы хотите научиться программировать на языке Ruby, и по возможности быстрее. Вы хотите *запомнить* прочитанное, а для этого абсолютно необходимо сначала *понять* прочитанное.

Чтобы извлечь максимум пользы из учебного процесса, нужно заставить ваш мозг воспринимать новый материал как Нечто Важное. Критичное для вашего существования. Такое же важное, как тигр. Иначе вам предостоит бесконечная борьба с вашим мозгом, который всеми силами уклоняется от запоминания новой информации.

Как же УБЕДИТЬ мозг, что программирование на языке Ruby не менее важно, чем тигр?

Есть способ медленный и скучный, а есть быстрый и эффективный. Первый основан на тупом повторении. Всем известно, что даже самую скучную информацию *можно* запомнить, если повторять ее снова и снова. При достаточном количестве повторений ваш мозг прикидывает: «*Вроде бы* несущественно, но раз одно и то же повторяется *столько* раз... Ладно, уговорил».

Быстрый способ основан на ***повышении активности мозга*** и особенно на сочетании разных ее *видов*. Доказано, что все факторы, перечисленные на предыдущей странице, помогают вашему мозгу работать на вас. Например, исследования показали, что размещение слов *внутри* рисунков (а не в подписях, в основном тексте и т. д.) заставляет мозг анализировать связи между текстом и графикой, а это приводит к активизации большего количества нейронов. Больше нейронов — выше вероятность того, что информация будет сочтена важной и достойной запоминания.

Разговорный стиль тоже важен: обычно люди проявляют больше внимания, когда они участвуют в разговоре, так как им приходится следить за ходом беседы и высказывать свое мнение. Причем мозг совершенно не интересует, что вы «разговариваете» с книгой! С другой стороны, если текст сух и формален, то мозг чувствует то же, что чувствуете вы на скучной лекции в роли пассивного участника. Его клонит в сон.

Но рисунки и разговорный стиль — это только начало.

Вот что сделали МЫ:

Мы использовали ***рисунки***, потому что мозг лучше приспособлен для восприятия графики, чем текста. С точки зрения мозга рисунок стоит тысячи слов. А когда текст комбинируется с графикой, мы внедряем текст прямо *в* рисунки, потому что мозг при этом работает эффективнее.

Мы используем ***избыточность***: повторяем одно и то же несколько раз, применяя *разные* средства передачи информации, обращаемся к *разным* чувствам — и все для повышения вероятности того, что материал будет закодирован в нескольких областях вашего мозга.

Мы используем концепции и рисунки несколько ***неожиданным*** образом, потому что мозг лучше воспринимает новую информацию. Кроме того, рисунки и идеи обычно имеют ***эмоциональное*** *содержание*, потому что мозг обращает внимание на биохимию эмоций. То, что заставляет нас *чувствовать*, лучше запоминается — будь то ***шутка***, ***удивление*** или ***интерес***.

Мы используем ***разговорный стиль***, потому что мозг лучше воспринимает информацию, когда вы участвуете в разговоре, а не пассивно слушаете лекцию. Это происходит и при *чтении*.

В книгу включены многочисленные ***упражнения***, потому что мозг лучше запоминает, когда вы что-то *делаете*. Мы постарались сделать их непростыми, но интересными — то, что предпочитает большинство читателей.

Мы совместили ***несколько стилей*** обучения, потому что *одни* читатели предпочитают пошаговые описания, *другие* стремятся сначала представить «общую картину», а *третьим* хватает фрагмента кода. Независимо от ваших личных предпочтений *полезно* видеть несколько вариантов представления одного материала.

Мы постарались задействовать ***оба полушария вашего мозга***; это повышает вероятность усвоения материала. Пока одна сторона мозга работает, другая часто имеет возможность отдохнуть; это повышает эффективность обучения в течение продолжительного времени.

А еще в книгу включены ***истории*** и упражнения, отражающие ***другие точки зрения***. Мозг глубже усваивает информацию, когда ему приходится оценивать и выносить суждения.

В книге часто встречаются ***вопросы***, на которые не всегда можно дать простой ответ, потому что мозг быстрее учится и запоминает, когда ему приходится что-то делать. Невозможно накачать *мышцы*, наблюдая за тем, как занимаются *другие*. Однако мы позаботились о том, чтобы усилия читателей были приложены в *верном* направлении. Вам не придется ломать голову над невразумительными примерами или разбираться в сложном, перенасыщенном техническим жаргоном или слишком лаконичном тексте.

В историях, примерах, на картинках используются ***люди*** — потому что вы тоже ***человек***. И ваш мозг обращает на людей больше внимания, чем на неодушевленные ***предметы***.

Вырежьте и прикрепите на холодильник.

Что можете сделать ВЫ, чтобы заставить свой мозг повиноваться

Мы свое дело сделали. Остальное за вами. Эти советы станут отправной точкой; прислушайтесь к своему мозгу и определите, что вам подходит, а что не подходит. Пробуйте новое.

1. **Не торопитесь. Чем больше вы поймете, тем меньше придется запоминать.**

   Просто *читать* недостаточно. Когда в книге задается вопрос, не переходите сразу к ответу. Представьте, что кто-то действительно *задает* вам вопрос. Чем глубже ваш мозг будет мыслить, тем скорее вы поймете и запомните материал.

2. **Выполняйте упражнения, делайте заметки.**

   Мы включили упражнения в книгу, но выполнять их за вас не собираемся. И не *разглядывайте упражнения. **Берите карандаш и пишите.*** Физические действия *во время* учения повышают его эффективность.

3. **Читайте врезки.**

   Это значит: читайте всё. ***Врезки — часть основного материала!*** Не пропускайте их.

4. **Не читайте другие книги перед сном.**

   Часть обучения (особенно перенос информации в долгосрочную память) происходит после того, как вы откладываете книгу. Ваш мозг не сразу усваивает информацию. Если во время обработки поступит новая информация, часть того, что вы узнали ранее, может быть потеряна.

5. **Говорите вслух.**

   Речь активизирует другие участки мозга. Если вы пытаетесь что-то понять или получше запомнить, произнесите вслух. А еще лучше, попробуйте объяснить кому-нибудь другому. Вы будете быстрее усваивать материал и, возможно, откроете для себя что-то новое.

6. **Пейте воду. И побольше.**

   Мозг лучше всего работает в условиях высокой влажности. Дегидратация (которая может наступить еще до того, как вы почувствуете жажду) снижает когнитивные функции.

7. **Прислушивайтесь к своему мозгу.**

   Следите за тем, когда ваш мозг начинает уставать. Если вы начинаете поверхностно воспринимать материал или забываете только что прочитанное, пора сделать перерыв. С определенного момента попытки «загрузить» побольше материала становятся бесполезными и даже могут повредить процессу.

8. **Чувствуйте!**

   Ваш мозг должен знать, что материал книги действительно *важен*. Переживайте за героев наших историй. Придумывайте собственные подписи к фотографиям. Поморщиться над неудачной шуткой все равно лучше, чем не почувствовать ничего.

9. **Пишите побольше кода!**

   Освоить разработку приложений Ruby можно только одним способом: **писать побольше кода**. Собственно, именно этим вы будете заниматься по мере прочтения книги. Программирование — это квалифицированная работа, и тренировка — единственный путь к мастерству. И мы дадим вам много возможностей для тренировки: в каждой главе содержатся упражнения. Не пропускайте их; работа над упражнениями является важной частью процесса обучения. Мы приводим решения всех упражнений; не бойтесь **заглянуть в решение**, если окажетесь в тупике! И все же сначала постарайтесь решить задачу самостоятельно. И обязательно добейтесь того, чтобы ваше решение заработало, прежде чем переходить к следующей части книги.

Примите к сведению

Это учебник, а не справочник. Мы намеренно убрали из книги все, что могло бы помешать изучению материала, над которым вы работаете. И при первом чтении книги начинать следует с самого начала, потому что книга предполагает наличие у читателя определенных знаний и опыта.

Небольшой опыт программирования на другом языке не повредит.

Многие разработчики переходят на Ruby *после* другого языка программирования (часто пытаясь держаться подальше от этого языка). Материал излагается на уровне, понятном даже для новичка, но мы не рассказываем подробно о том, что такое переменная и как работает команда `if`. Вам будет проще, если вы хотя бы *в общих чертах* представляете эти темы.

Мы не пытаемся подробно описывать каждый класс, библиотеку и метод.

В Ruby существует *множество* встроенных классов и методов. Конечно, все они представляют интерес, но нам бы не удалось их рассмотреть даже в книге *вдвое большего* объема. Наше внимание будет сосредоточено на основных классах и методах, которые *важны* для вас — начинающего разработчика. Мы позаботимся о том, чтобы вы хорошо понимали их суть и достаточно уверенно чувствовали себя в отношении того, когда и как их следует использовать. В любом случае после прочтения книги вы сможете взять любой справочник и быстро найти информацию обо всех классах и методах, которые в книге не рассматриваются.

Упражнения ОБЯЗАТЕЛЬНЫ.

Упражнения являются частью основного материала книги. Одни упражнения способствуют запоминанию материала, другие помогают лучше понять его, третьи ориентированы на его практическое применение. ***Не пропускайте упражнения.***

Повторение применяется намеренно.

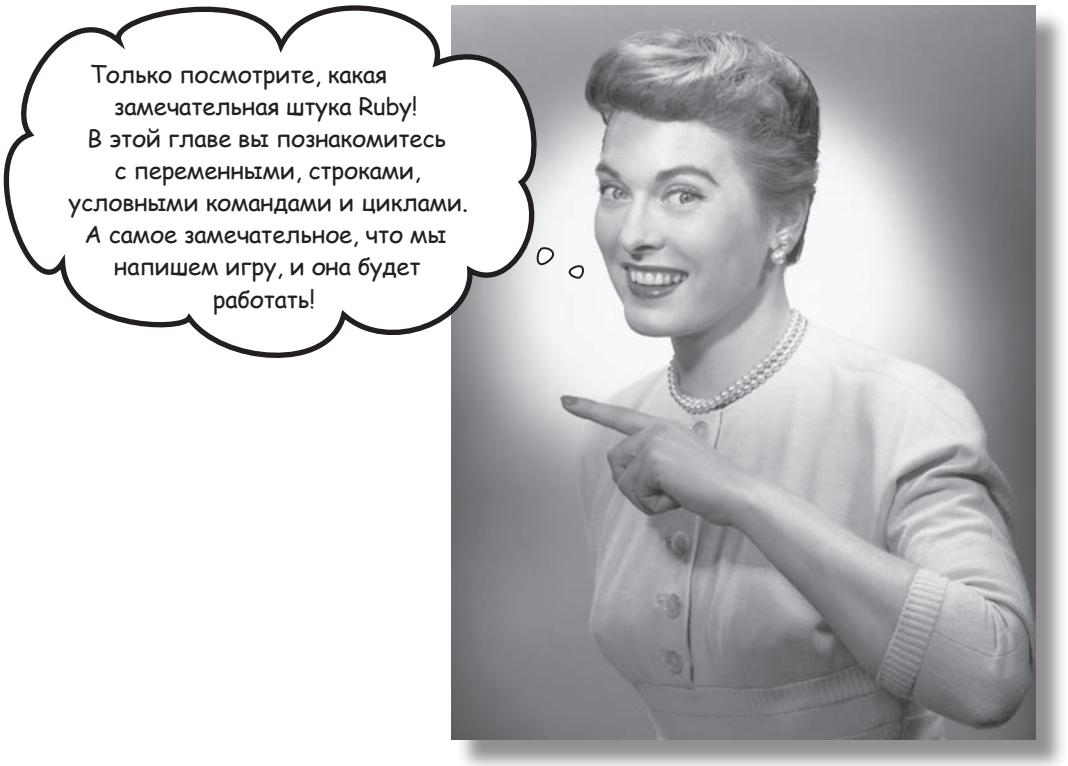
У книг этой серии есть одна принципиальная особенность: мы хотим, чтобы вы *действительно хорошо* усвоили материал. И чтобы вы запомнили все, что узнали. Большинство справочников не ставит своей целью успешное запоминание, но это не справочник, а *учебник*, поэтому некоторые концепции излагаются в книге по нескольку раз.

Примеры были сделаны по возможности компактными.

Нашим читателям не нравится просматривать 200 строк кода в примерах, чтобы найти две действительно важные строки. Большинство примеров книги приводится в минмально возможном контексте, чтобы та часть, которую вы изучаете, была простой и наглядной. Не ждите, что все примеры будут защищенными от ошибок, или хотя бы полными — они написаны *в учебных целях* и не всегда обладают полноценной функциональностью.

Все файлы примеров доступны для загрузки в Интернете. Вы найдете их по адресу *http://headfirstruby.com/*.

Программируйте так, как вам удобно



Вас интересует, почему вокруг столько говорят о языке Ruby и подойдет ли он вам? Ответьте на простой вопрос: ***вам нравится работать эффективно?*** Кажется ли вам, что с многочисленными компиляторами, библиотеками, файлами классов и комбинациями клавиш других языков вы действительно становитесь ближе к **готовому продукту**, **восхищению коллег** и **толпе счастливых заказчиков**? Вы хотите, чтобы язык программирования **занимался техническими подробностями** за вас? Если вам хочется перестать писать рутинный код и *работать над задачей*, то язык Ruby — для вас. Ruby позволит вам **сделать больше с меньшим объемом кода.**

Философия Ruby

В 1990-е годы японский программист Юкихиро Мацумото (сокращенно Мац) мечтал об идеальном языке программирования. Он хотел, чтобы его язык:

- Был простым в изучении и использовании.
- Был достаточно гибким для решения любых задач из области программирования.
- Позволял программисту сосредоточиться на решаемой задаче.
- Создавал меньше проблем у программиста.
- Был объектно-ориентированным.

Он присмотрелся ко всем языкам того времени, но ни один из этих языков не соответствовал его требованиям в полной мере. И тогда Мац решил создать собственный язык, который получил название Ruby.

Немного повозившись с Ruby в своей работе, Мац сделал его достоянием широкой публики в 1995-м. С тех пор в сообществе Ruby произошло немало интересного:

- Была построена обширная подборка библиотек Ruby для решения самых разнообразных задач, от чтения файлов с данными CSV (значения, разделенные запятыми) до работы с объектами по сети.
- Были написаны альтернативные интерпретаторы, способные ускорить выполнение кода Ruby или интегрировать его с другими языками.
- Был создан Ruby on Rails — чрезвычайно популярная инфраструктура для веб-приложений.

Причины столь стремительного роста творческого потенциала и продуктивности следует искать в самом языке Ruby. Гибкость и простота использования — основополагающие принципы языка, а это означает, что Ruby может использоваться для решения любых задач программирования с меньшим количеством строк программного кода, чем в других языках.

Изучив основы, вы согласитесь: с Ruby приятно иметь дело!

Гибкость и простота использования — основополагающие принципы языка.

Где взять Ruby

Сначала о главном: вы можете *писать* Ruby весь день напролет, только от этого не будет никакого толку, если вы не сможете этот код *выполнить*. Прежде всего потребуется работающий **интерпретатор** Ruby. Вам нужна версия 2.0 или выше. Откройте новое терминальное окно (также называемое *окном командной строки*) и введите следующую команду:

```
ruby -v
```

С ключом «-v» Ruby выводит номер версии.

Команда «ruby» сама по себе запускает интерпретатор Ruby.

Нажмите Ctrl-C, чтобы завершить работу интерпретатора и вернуться к подсказке операционной системы.

```
File Edit Window Help
$ ruby -v
ruby 2.0.0p0 (2013-02-24 revision 39474) [x86_64-darwin11.4.2]
$ ruby
^Cruby: Interrupt

$
```

Если вы ввели в приглашении команду **`ruby -v`** и получили ответ следующего вида, значит, все хорошо:

```
ruby 2.0.0p0 (2013-02-24 revision 39474) [x86_64-darwin11.4.2]
```

На остальное не обращайте внимания. Важно, чтобы здесь была указана версия «ruby 2.0» или выше.

Задание!

Если в вашей системе нет версии Ruby 2.0 или выше, зайдите на сайт ***www.ruby-lang.org*** и загрузите пакет для своей ОС.

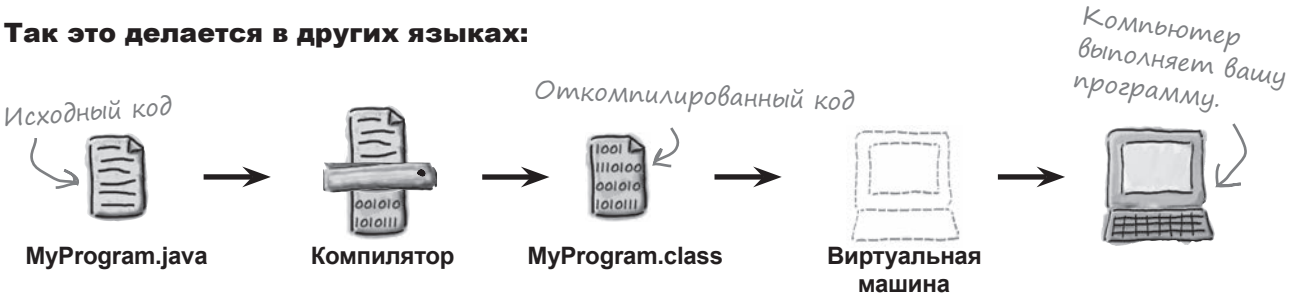
Кстати говоря, если вы случайно введете команду `ruby` (без ключа `-v`), Ruby будет ждать, пока вы введете какой-нибудь код. Чтобы выйти из этого режима, просто нажмите клавишу Control одновременно с клавишей C. Это можно сделать в любой момент, если вам захочется немедленно завершить работу с Ruby.

Использование Ruby

Файлы с исходным кодом Ruby называются **сценариями** (scripts), но на самом деле это обычные текстовые файлы. Чтобы запустить сценарий Ruby на выполнение, просто сохраните код Ruby в файле и запустите этот файл в интерпретаторе Ruby.

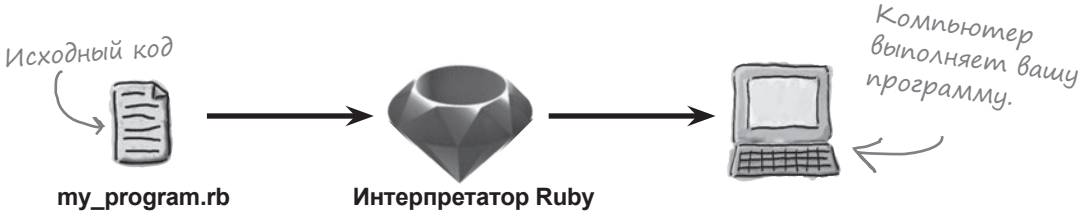
Возможно, вы привыкли к другим языкам (таким, как C++, C# или Java), в которых код компилируется в двоичный формат, «понятный» для процессора или виртуальной машины. В таких языках код не может выполняться без предварительной обработки компилятором.

Так это делается в других языках:



В Ruby *этот шаг можно пропустить*. Ruby автоматически, практически мгновенно компилирует исходный код сценария. А это означает, что вы можете *быстрее* опробовать только что написанный код в деле!

Так это делается в Ruby:




```
puts "hello world"
```

Введите исходный код.
Сохраните в файле: ***hello.rb***

```
File Edit Window Help
$ ruby hello.rb
hello world
```

А вот и результат!

Запустите исходный код в интерпретаторе Ruby.

Интерактивное использование Ruby

У таких языков, как Ruby, есть еще одно большое преимущество. Вам не только не обязательно запускать компилятор каждый раз, когда потребуется опробовать свой код, — его даже не обязательно сохранять в файле.

В поставку Ruby входит отдельная программа, которая называется **irb** (**I**nteractive **R**ub**y**). Оболочка `irb` берет любое выражение Ruby, введенное пользователем, немедленно вычисляет его и выводит результат. Эта возможность особенно полезна при изучении языка, потому что новичок немедленно получает обратную связь. Впрочем, даже профессионалы Ruby используют `irb` для проверки новых идей.

В этой книге мы напишем много сценариев, предназначенных для запуска в интерпретаторе Ruby. Но каждый раз, когда вы осваиваете новую концепцию, будет полезно запустить `irb` и немного поэкспериментировать.

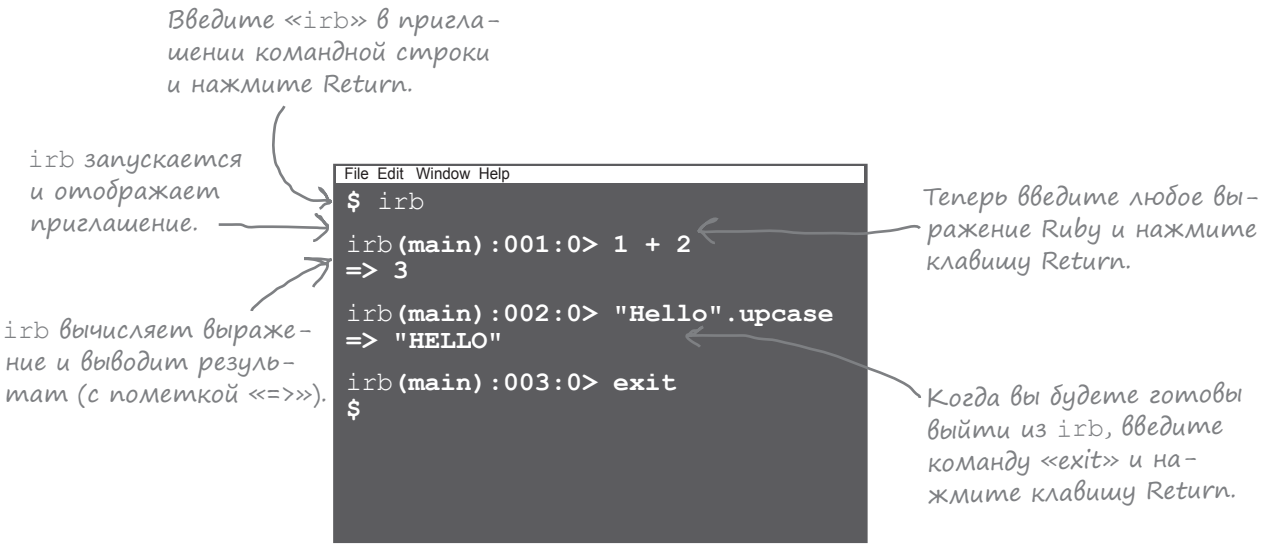
Так чего же мы ждем? Давайте запустим `irb` и посмотрим, как работают выражения Ruby.

Использование интерактивной оболочки irb

Откройте терминальное окно и введите команду **`irb`**. Команда запускает интерактивный интерпретатор Ruby. (О том, что интерпретатор запустился, вы узнаете по изменению внешнего вида подсказки, хотя в вашей системе она может выглядеть не так, как на иллюстрации.)

После этого введите любое выражение и нажмите клавишу Enter/Return. Ruby мгновенно вычисляет его и выводит результат.

Завершив работу с `irb`, введите в приглашении команду **`exit`**. На экране снова появляется приглашение командной строки ОС.



Ваши первые выражения Ruby

Итак, теперь вы знаете, как запустить `irb`. Введем несколько выражений и посмотрим, какие результаты будут получены.

Введите следующее выражение в приглашении и нажмите клавишу Return: `1 + 2`

Вы увидите следующий результат: `=> 3`

Математические операторы и сравнения

Основные математические операторы Ruby работают так же, как в большинстве других языков. Оператор «+» выполняет сложение, «—» выполняет вычитание, «*» — умножение, «/» — деление и «**» — возведение в степень.

Если ввести:	irb выведет следующий результат:
`5.4 - 2.2`	`=> 3.2`
`3 * 4`	`=> 12`
`7 / 3.5`	`=> 2.0`
`2 ** 3`	`=> 8`

Операторы «<» и «>» сравнивают два значения и проверяют, что одно из них больше (или меньше) другого. Оператор «==» (здесь *два* знака равенства) проверяет, равны ли два значения.

`4 < 6`	`=> true`
`4 > 6`	`=> false`
`2 + 2 == 5`	`=> false`

Строки

Строка (string) представляет собой цепочку символов. Строки могут использоваться для хранения имен, адресов электронной почты, телефонных номеров... в общем, чего угодно. У строк языка Ruby есть одна особенность: даже очень большие строки в Ruby обрабатываются очень эффективно (в отличие от некоторых других языков).

Самый простой способ определить строку — заключить составляющие ее символы в одинарные (') или двойные кавычки("). Эти две разновидности ограничителей слегка отличаются друг от друга; вскоре мы вернемся к этой теме.

`"Hello"`	`=> "Hello"`
`'world'`	`=> "world"`

Переменные

В языке Ruby можно создавать **переменные** — имена, ссылающиеся на значения.

Объявлять переменные заранее в Ruby не обязательно; они автоматически создаются в момент присваивания. Присваивание выполняется оператором «=» (*один* знак равенства).

Если ввести:	*irb выведет следующий результат:*
`small = 8`	`=> 8`
`medium = 12`	`=> 12`

Имена переменных начинаются со строчной буквы и могут содержать буквы, цифры и символы подчеркивания.

После того как переменной будет присвоено значение, вы сможете в любой момент использовать имя переменной в любом контексте, в котором может использоваться исходное значение.

```
small + medium    => 20
```

У переменных в Ruby нет типа; в них могут храниться любые значения. Переменной можно присвоить строку, потом тут же присвоить ей вещественное число — это абсолютно нормально.

```
pie = "Lemon"     => "Lemon"
pie = 3.14        => 3.14
```

Оператор «+=» увеличивает текущее значение переменной.

```
number = 3        => 3
number += 1       => 4
number            => 4

string = "ab"     => "ab"
string += "cd"    => "abcd"
string            => "abcd"
```

Записывайте имена переменных в нижнем регистре. Постарайтесь не включать в имена переменных цифры; обычно без них можно обойтись. Разделяйте слова символами подчеркивания.

```
my_rank = 1
```

Этот стиль записи иногда называется «змеиным»: с символами подчеркивания имя напоминает змею, ползущую по земле.

Вокруг одни объекты!

Ruby относится к семейству *объектно-ориентированных* языков. Это означает, что прямо к данным присоединяются полезные **методы** — фрагменты кода, выполняемые по мере надобности.

В современных языках такие данные, как строки, часто представляют собой полноценные объекты. И конечно, у строк есть свои методы, которые могут вызываться в программах:

Если ввести: *irb выведет следующий результат:*

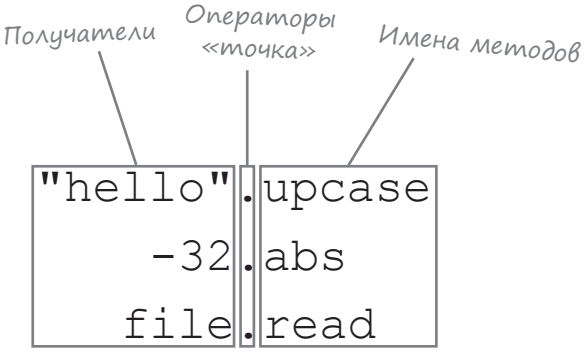
```
"Hello".upcase      => HELLO
"Hello".reverse     => olleH
```

Язык Ruby идет еще дальше: в нем *все* данные представляют собой объекты: даже обычное число является объектом. А это означает, что у чисел тоже могут быть полезные методы.

```
42.even?            => true
-32.abs             => 32
```

Вызов метода для объекта

При таком вызове объект, для которого вызывается метод, называется **получателем** метода. К этой категории относится все, что находится слева от оператора «точка». Представьте, что вызов метода для объекта напоминает *передачу сообщения* — что-то вроде записки «Привет, вы не могли бы передать мне свою версию, записанную в верхнем регистре?» или «Нельзя ли получить ваше абсолютное значение (модуль)?»



Упражнение

Откройте терминальное окно, введите команду **`irb`** и нажмите клавишу Enter/Return. Под каждым из приведенных ниже выражений Ruby запишите, какой результат, по вашему мнению, будет получен при выполнении его в интерпретаторе. Потом введите выражение в `irb` и нажмите Enter. Совпадет ли ваше предположение с тем, что выдаст `irb`?

```
42 / 6
```

.....

```
name = "Zaphod"
```

.....

```
name.upcase
```

.....

```
"Zaphod".upcase
```

.....

```
name.reverse
```

.....

```
name.upcase.reverse
```

.....

```
name.class
```

.....

```
name * 3
```

.....

```
5 > 4
```

.....

```
number = -32
```

.....

```
number.abs
```

.....

```
-32.abs
```

.....

```
number += 10
```

.....

```
rand(25)
```

.....

```
number.class
```

.....

Упражнение
Решение

Откройте терминальное окно, введите команду **`irb`** и нажмите клавишу Enter/Return. Под каждым из приведенных ниже выражений Ruby запишите, какой результат, по вашему мнению, будет получен при выполнении его в интерпретаторе. Потом введите выражение в `irb` и нажмите Enter. Совпадет ли ваше предположение с тем, что выдаст `irb`?

```
42 / 6
```

7

Операция присваивания переменной возвращает присвоенное значение.

```
name = "Zaphod"
```

«Zaphod"

```
name.upcase
```

«ZAPHOD"

Вы можете вызывать методы объекта, хранящегося в переменной...

```
"Zaphod".upcase
```

«ZAPHOD"

Причем вам даже не обязательно сначала сохранять его в переменной!

Для значения, возвращаемого методом, можно вызвать другой метод.

```
name.reverse
```

«dohpaZ"

```
name.upcase.reverse
```

«DOHPAZ"

Класс объекта определяет, что собой представляет этот объект.

```
name.class
```

String

```
name * 3
```

Оказывается, строки можно «умножать»!

«ZaphodZaphodZaphod"

```
5 > 4
```

true

```
number = -32
```

-32

```
number.abs
```

32

```
-32.abs
```

32

Текущее значение переменной увеличивается на 10, а результат снова присваивается переменной.

```
number += 10
```

-22

```
rand(25)
```

Да, это ТОЖЕ вызов метода; просто мы не указали получателя. Скоро расскажем подробнее!

Случайное число

Результат неизвестен заранее (это случайное число).

```
number.class
```

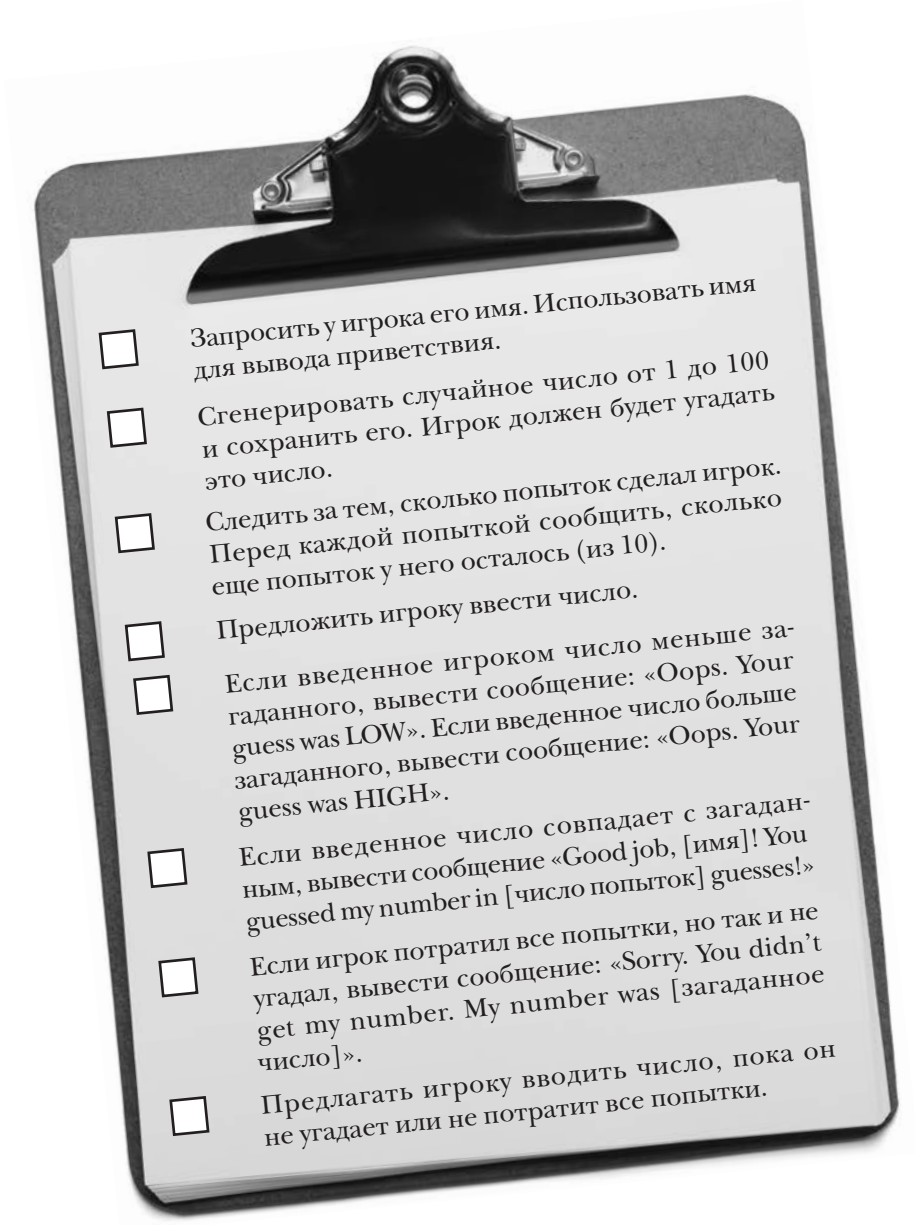
Fixnum

Fixnum — разновидность целых чисел.

Построим игру

В первой главе книги мы построим простую игру. Если задача покажется слишком сложной, не беспокойтесь: когда вы работаете на Ruby, все не так страшно!

Для начала разберемся, что же нужно сделать:

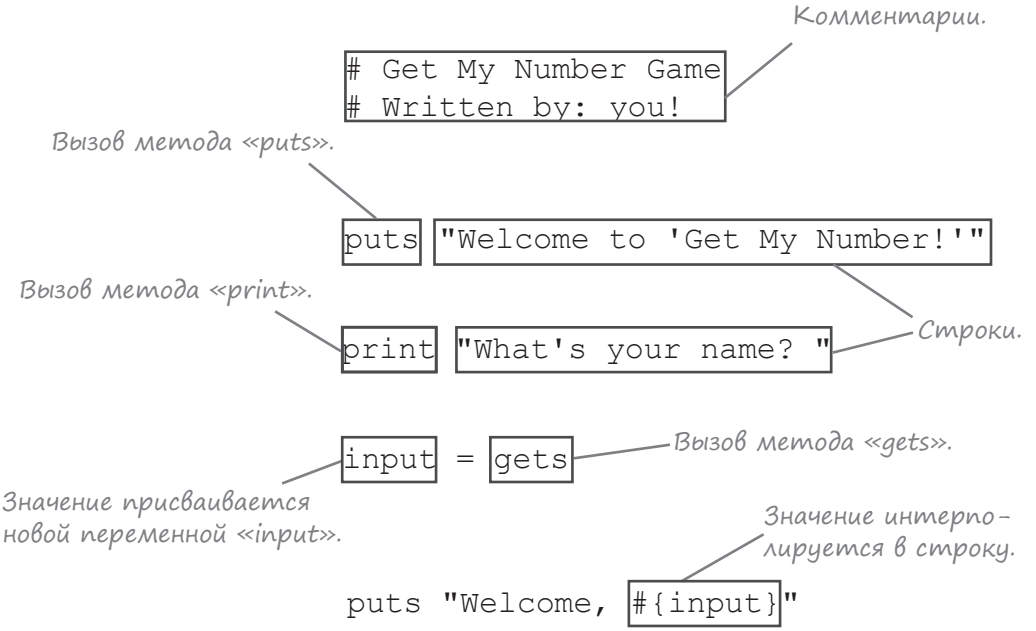


Гэри, разработчик игр

Ввод, сохранение и вывод

Итак, первое требование — поприветствовать пользователя по имени. Для этого необходимо написать сценарий, который запрашивает *ввод* (входные данные) у пользователя, *сохраняет* этот ввод, а затем использует сохраненное значение для *создания вывода*.

Все это можно сделать буквально в нескольких строках кода Ruby:



Все составляющие этого сценария будут более подробно рассмотрены на нескольких ближайших страницах. Но для начала опробуем его в деле!

Запуск сценариев

Мы написали простой сценарий, который выполняет первое требование: поприветствовать игрока по имени. А теперь вы узнаете, как выполнить этот сценарий, чтобы вы могли воочию увидеть результаты своей работы.

Шаг 1:

Откройте новый документ в своем любимом текстовом редакторе и введите следующий фрагмент.

```
# Get My Number Game
# Written by: you!

puts "Welcome to 'Get My Number!'"
print "What's your name? "

input = gets

puts "Welcome, #{input}"
```

get_number.rb

Шаг 2:

Сохраните файл под именем *get_number.rb*.

Шаг 3:

Откройте новое терминальное окно и перейдите в каталог, в котором была сохранена программа.

Шаг 4:

Запустите программу командой **`ruby get_number.rb`**.

Шаг 5:

На экране появляется приветствие и приглашение. Введите свое имя и нажмите клавишу Enter/Return. Вы увидите сообщение, в котором программа приветствует вас по имени.

```
File Edit Window Help
$ ruby get_number.rb
Welcome to 'Get My Number!'
What's your name? Jay
Welcome, Jay
```

Рассмотрим каждую из составляющих этого кода более подробно.

Комментарии

Файл с исходным кодом начинается с пары комментариев. Ruby игнорирует все символы от решетки (#) до конца строки, чтобы вы могли оставлять в программах инструкции или заметки для себя и своих коллег-разработчиков.

Если вы включили знак # в свой код, то все символы до конца этой строки будут рассматриваться как комментарий. Этот способ комментирования работает точно так же, как комментарии // в Java или JavaScript.

```
i_am = "executed" # I'm not.
# Me neither.
```

Комментарии.

```
# Get My Number Game
# Written by: you!
```

«puts» и «print»

«Настоящий» код начинается с вызова метода `puts` («puts» — сокращение от «**put s**tring»), который выводит текст на стандартном устройстве вывода (обычно на терминале). При вызове метода передается строка с выводимым текстом.

Вызов метода «puts».

Вызов метода «print».

```
puts "Welcome to 'Get My Number!'"
print "What's your name? "
```

Строки.

Другая строка — с предложением ввести имя пользователя — передается ниже методу `print`. Метод `print` работает точно так же, как `puts`, за одним исключением: `puts` добавляет в конец выводимого текста символ новой строки (если там еще нет такого символа), а метод `print` этого не делает. По эстетическим соображениям мы завершаем строку, передаваемую `print`, пробелом, чтобы выведенное сообщение немного отстояло от области, в которой пользователь будет вводить свое имя.

Погодите — вы назвали `print` и `puts` методами... Разве не нужно добавить оператор «точка» для указания объекта, для которого они вызываются?

Иногда получателя при вызовах методов указывать не обязательно.

Методы `puts` и `print` настолько важны и так часто используются, что в Ruby они были включены в **среду выполнения верхнего уровня**. Методы, определяемые в среде верхнего уровня, могут вызываться *где угодно* в коде Ruby *без* указания получателя. О том, как определяются подобные методы, вы узнаете в начале главы 2.

Аргументы методов

Метод `puts` получает строку и выводит ее на стандартное устройство вывода (в терминальное окно).

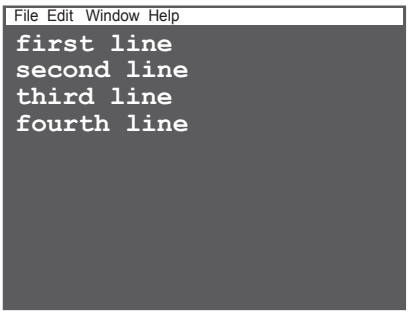
```
puts "first line"
```

Строка, передаваемая методу `puts`, называется **аргументом** метода.

Метод `puts` может получать несколько аргументов: просто разделите их запятыми. Каждый аргумент выводится в отдельной строке.

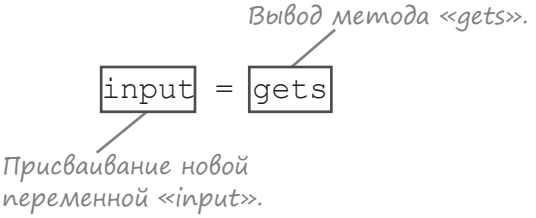
```
puts "second line", "third line", "fourth line"
```

Так это выглядит в терминальном окне.



«gets»

Метод `gets` (сокращение от «**get s**tring») читает строку из стандартного ввода (символы, вводимые в терминальном окне). При вызове `gets` программа приостанавливается, пока пользователь не введет свое имя и не нажмет клавишу Enter. Программа получает введенный пользователем текст как отдельную строку.



Как и в случае с `puts` и `print`, метод `gets` может вызываться в любом месте кода без указания получателя.

Круглые скобки не обязательны

В языке Ruby аргументы методов *могут* заключаться в круглые скобки:

```
puts("one", "two")
```

Однако круглые скобки не обязательны, и при вызове `puts` многие разработчики предпочитают их опускать.

```
puts "one", "two"
```

Как упоминалось выше, метод `gets` читает строку из стандартного ввода. Аргументы ему (обычно) не нужны:

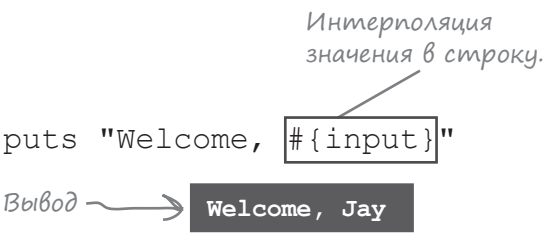
```
gets
```

Знатоки Ruby *непреклонно* считают, что если метод вызывается *без* аргументов, то и круглых скобок быть *не должно*. Пожалуйста, не используйте такие команды (хотя формально они ничего не нарушают):

```
gets()
```
No!

Если метод не получает аргументов, не включайте круглые скобки при вызове. Круглые скобки можно опустить и в том случае, если аргументы *есть*, но это может немного усложнить чтение кода. Если не уверены — лучше поставьте круглые скобки!

Интерполяция строк



В последней строке нашего сценария `puts` вызывается еще с одной строкой. Этот вызов отличается от других тем, что в нем в строку **интерполируется** (подставляется) значение переменной. Каждый раз, когда *внутри* строки встречается запись `#{...}`, Ruby использует значение в фигурных скобках для «заполнения пробелов». Маркеры `#{...}` могут располагаться в любом месте строки: в начале, в конце или где-то в середине.

Маркеры `#{...}` не ограничиваются использованием переменных — в них могут содержаться любые выражения Ruby.

```
puts "The answer is #{6 * 7}."
```

Вывод → `The answer is 42.`

Обратите внимание: Ruby применяет интерполяцию только к строкам в *двойных* кавычках. Если включить маркер `#{...}` в строку, заключенную в *одинарные* кавычки, он будет интерпретирован буквально.

```
puts 'Welcome, #{input}'
```

Вывод → `Welcome, #{input}`

Часто Задаваемые Вопросы

В: А где завершители «;»?

О: В Ruby символ **«;»** *может* использоваться для разделения команд, но обычно так делать *не рекомендуется* (потому что код сложнее читается).

```
puts "Hello";  <-- Нет!
puts "World";
```

В Ruby отдельные строки считаются отдельными командами, поэтому символы **«;»** становятся лишними.

```
puts "Hello"
puts "World"
```

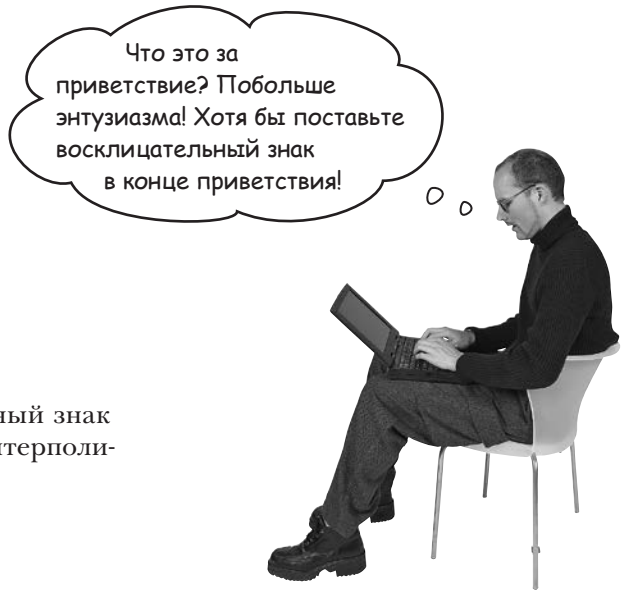
В: Мой другой язык потребовал бы, чтобы сценарий был помещен в класс с методом «main». В Ruby этого нет?

О: Нет! И это одна из замечательных особенностей Ruby — простые программы пишутся без лишних церемоний. Напишите несколько команд, и работа завершена!

В Ruby простые программы пишутся без лишних церемоний.

Что там в строке?

```
File Edit Window Help
$ ruby get_number.rb
Welcome to 'Get My Number!'
What's your name? Jay
Welcome, Jay
```



К счастью, это сделать несложно. Восклицательный знак добавляется в конец строки приветствия, после интерполируемого значения.

```
puts "Welcome to 'Get My Number!'"
print "What's your name? "

input = gets

puts "Welcome, #{input}!"
```

Всего один символ!

Но попытавшись запустить программу, вы увидите, что восклицательный знак не выводится после имени пользователя, а опускается на следующую строку!

Стоп! Почему он перешел на следующую строку?

```
File Edit Window Help
$ ruby get_number.rb
Welcome to 'Get My Number!'
What's your name? Jay
Welcome, Jay
!
```

Почему это происходит? Может, что-то не так с переменной `input`...

Но в результате вызова `puts` ничего особенного не видно. Если присоединить следующую строку к приведенному выше коду, будет получен следующий результат:

```
puts input
```

```
Jay
```

Анализ объектов методами «inspect» и «p»

Попробуем еще, на этот раз с использованием методов, предназначенных специально для отладки программ Ruby. Метод `inspect` поддерживается всеми объектами Ruby. Он преобразует объект в строковое представление, удобное для отладки. Иначе говоря, метод раскрывает те стороны объекта, которые обычно не видны в выходных данных программы.

Результат вызова `inspect` для нашей строки выглядит так:

```
puts input.inspect      "Jay\n"  <-- A-HA!
```

Погодите, что это за `\n` в конце строки? Мы раскроем эту тайну на следующей странице...

Вывод результата `inspect` является настолько частой операцией, что в Ruby для него существует специальная сокращенная запись: метод `p`. Этот метод работает точно так же, как `puts`, не считая того, что он вызывает `inspect` для каждого аргумента перед его выводом.

Этот вызов `p` почти идентичен предшествующему коду:

```
p input                 "Jay\n"
```

Запомните метод `p`; в последующих главах он будет использоваться для отладки кода Ruby.

Метод «inspect» раскрывает те стороны объекта, которые обычно не видны в выходных данных программы.

Служебные последовательности в строках

Наш вызов метода `p` показал, что в конце пользовательского ввода располагаются какие-то «лишние» данные:

```
p input
```

```
"Jay\n"
```

Эти два символа, обратная косая черта (`\`) и `n` сразу же после нее, в действительности представляют один символ: символ новой строки. (Такое название происходит из-за того, что при выводе этого символа курсор в терминальном окне переходит на *новую строку*.) Данные, введенные пользователем, завершаются символом новой строки, потому что пользователь нажимает клавишу Return, чтобы сообщить о завершении ввода, и это нажатие клавиши сохраняется в виде дополнительного символа. Этот символ включается в возвращаемое значение метода `gets`.

Символ косой черты (`\`) и следующий за ним символ `n` образует **служебную последовательность** — часть строки, представляющую символы, которые не имеют обычного представления в исходном коде.

На практике чаще всего встречаются служебные последовательности `\n` (новая строка, вы уже видели этот символ) и `\t` (символ табуляции для создания отступов).

```
puts "First line\nSecond line\nThird line"
puts "\tIndented line"
```

```
First line
Second line
Third line
      Indented line
```

Часто используемые служебные последовательности

Если эта последовательность входит в строку в двойных кавычках...	*...вы получите следующий символ...*
`\n`	новая строка
`\t`	табуляция
`\"`	двойные кавычки
`\'`	одиночная кавычка
`\\`	обратная косая черта

Обычно попытка включения двойной кавычки (`"`) в строку, заключенную в двойные кавычки, приводит к тому, что внутренняя кавычка интерпретируется как признак завершения строки, что приводит к ошибке:

```
puts ""It's okay," he said."
```

Ошибка →

```
syntax error, unexpected
tCONSTANT
```

Если «экранировать» внутреннюю двойную кавычку, поставив перед ней обратную косую черту, то ее можно будет включить в строку, заключенную в двойные кавычки, и это не приведет к ошибке.

```
puts "\"It's okay,\" he said."
```

```
"It's okay," he said.
```

Наконец, раз символ `\` отмечает начало служебной последовательности, нам также понадобится способ представления символа обратной косой черты, который *не является* частью служебной последовательности. Комбинация `\\` обозначает «фактический» символ обратной косой черты.

```
puts "One backslash: \\"
```

```
One backslash: \
```

Учтите, что большинство служебных последовательностей действует только в строках, заключенных в *двойные* кавычки. В строках, заключенных в *одинарные* кавычки, служебные последовательности обычно интерпретируются буквально.

```
puts '\n\t\"'
```

```
\n\t\"
```

Вызов «chomp» для объекта строки

```
File Edit Window Help
$ ruby get_number.rb
Welcome to 'Get My Number!'
What's your name? Jay
Welcome, Jay
!
```

Нужно удалить символ новой строки вызовом метода `chomp`.

Если текстовые данные завершаются символом новой строки, то вызов метода `chomp` удалит его. Этот метод очень удобен для очистки строк, полученных при вызове `gets`.

Метод `chomp` обладает более узкой специализацией, чем методы `print`, `puts` и `gets`: он доступен только для объектов строк. Это означает, что строка, на которую ссылается переменная `input`, должна быть указана как *получатель* метода `chomp`. К переменной `input` необходимо применить оператор «точка».

```
# Get My Number Game
# Written by: you!

puts "Welcome to 'Get My Number!'"
print "What's your name? "

input = gets

name = input.chomp
puts "Welcome, #{name}!"
```

Возвращаемое значение «chomp» сохраняется в новой переменной «name».

Вызов метода «chomp».

Строка в «input» является получателем метода «chomp».

Оператор «точка».

В приветствии используется «name» вместо «input».

Метод `chomp` возвращает ту же строку, но без завершающего символа новой строки. Результат сохраняется в новой переменной `name`, которая затем выводится в составе приветственного сообщения.

Если снова запустить программу, вы увидите, что наше эмоциональное приветствие теперь работает правильно.

```
File Edit Window Help
$ ruby get_number.rb
Welcome to 'Get My Number!'
What's your name? Jay
Welcome, Jay!
```

Какие методы доступны для объекта?

Не стоит полагать, что вы можете вызвать любой метод для любого объекта. Например, следующая попытка приведет к ошибке:

```
puts 42.upcase
```
Ошибка →
```
undefined method `upcase' for 42:Fixnum (NoMethodError)
```

Хотя если подумать, это логично. Попытка перевести число в верхний регистр не имеет особого смысла, верно?

Но тогда какие методы *можно* вызывать для числа? Чтобы получить ответ на этот вопрос, можно воспользоваться еще одним методом с именем `methods`:

```
puts 42.methods
```

```
to_s
abs
odd?
...
```

И еще очень много!

Вызвав метод `methods` для строки, вы получите другой список:

```
puts "hello".methods
```

```
to_i
length
upcase
...
```

И еще намного больше — здесь не поместятся!

Почему списки различаются? Это связано с классом объекта. **Класс** представляет собой «шаблон» для создания новых объектов. Среди прочего он определяет, какие методы могут вызываться для создаваемого объекта.

Также существует другой метод, который сообщает, к какому классу относится объект. Он называется `class` (кто бы мог подумать!). Давайте опробуем его на нескольких объектах.

```
puts 42.class
puts "hello".class
puts true.class
```

```
Fixnum
String
TrueClass
```

В следующей главе мы поговорим о классах более подробно, так что следите за новостями.

Часто Задаваемые Вопросы

В: Как узнать, что делают все эти методы?

О: О том, как найти документацию по методам класса, вы узнаете в главе 11. А пока многие из этих методов вам попросту не понадобятся (а возможно, и *никогда* не понадобятся). Не беспокойтесь: если метод действительно важен, мы непременно расскажем, как его использовать!

Собственно, это весь код первого требования. Его можно вычеркнуть из списка!

Генерирование случайного числа

С приветствием мы разобрались, можно переходить к следующему требованию.

Метод `rand` генерирует случайное число в заданном диапазоне. Пожалуй, мы сможем воспользоваться им для генерирования загаданного числа.

При вызове `rand` передается аргумент — число, определяющее верхнюю границу диапазона (100). Посмотрим, как он работает:

```
puts rand(100)
puts rand(100)
```

```
67
25
```

Вроде бы неплохо, но есть одна проблема: `rand` генерирует числа в диапазоне от *нуля* до числа, *предшествующего* заданному максимуму. А это означает, что случайные числа будут генерироваться в диапазоне 0–99, а не 1–100, как нам нужно.

Впрочем, проблема легко решается — достаточно прибавить 1 к значению, полученному при вызове `rand`. И мы снова возвращаемся к диапазону 1–100!

```
rand(100) + 1
```

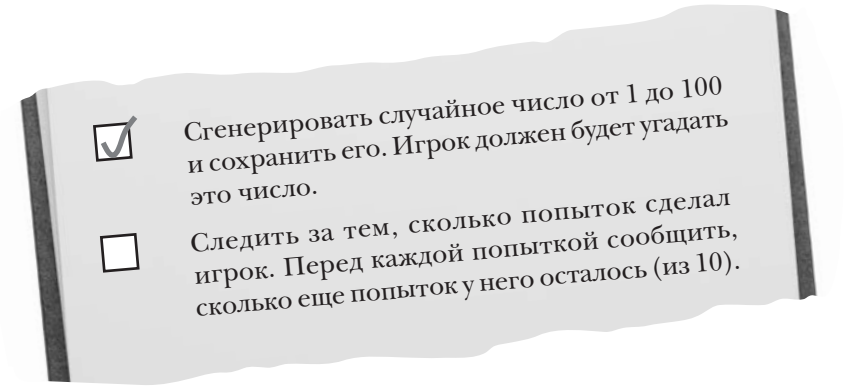
Результат сохраняется в новой переменной с именем `target`.

Наш новый код!

```
# Get My Number Game
# Written by: you!

puts "Welcome to 'Get My Number!'"

# Получение имени игрока и вывод приветствия.
print "What's your name? "
input = gets
name = input.chomp
puts "Welcome, #{name}!"

# Сохранение случайного числа.
puts "I've got a random number between 1 and 100."
puts "Can you guess it?"
target = rand(100) + 1
```

Преобразование числа в строку

Еще одно требование выполнено! Переходим к следующему...



«Следить за тем, сколько попыток сделал игрок...» Похоже, нам понадобится переменная для хранения количества попыток. Разумеется, в начале игры ни одной попытки еще не сделано, поэтому при создании переменной `num_guesses` должно присваиваться значение `0`.

```
num_guesses = 0
```

Первое, что приходит в голову для вывода количества оставшихся попыток, — конкатенация (сцепление) строк со знаком «плюс» (+), как это делается во многих других языках. Однако такое решение, как показано ниже, работать *не будет*:

```
remaining_guesses = 10 - num_guesses
puts remaining_guesses + " guesses left."
```

Ошибка!

Оператор + используется не только для *сложения чисел*, но и для *конкатенации строк*, а поскольку `remaining_guesses` содержит число, знак «плюс» воспринимается как попытка суммирования чисел.

Что делать? Нужно преобразовать число в строку. Почти у всех объектов Ruby есть метод `to_s` для выполнения такого преобразования; попробуем воспользоваться этим методом.

```
remaining_guesses = 10 - num_guesses
puts remaining_guesses.to_s + " guesses left."
```

```
10 guesses left.
```

Работает! Преобразование числа в строку сначала четко показывает Ruby, что выполняется конкатенация, а не сложение.

Впрочем, в Ruby существует и более простой способ...

Упрощенная работа со строками в Ruby

Вместо того чтобы вызывать `to_s`, мы можем избавиться от лишних хлопот с явным преобразованием числа в строку, воспользовавшись интерполяцией. Как вы видели в коде приветствия пользователя, при включении `#{...}` в строку, заключенную в двойные кавычки, Ruby вычисляет результат кода в фигурных скобках, преобразует его в строку в случае необходимости, после чего подставляет его в более длинную строку.

Автоматическое преобразование в строку означает, что мы можем обойтись без вызова `to_s`.

```
remaining_guesses = 10 - num_guesses
puts "#{remaining_guesses} guesses left."
```

```
10 guesses left.
```

Ruby позволяет размещать операторы в фигурных скобках, так что от переменной `remaining_guesses` тоже можно избавиться:

```
puts "#{10 - num_guesses} guesses left."
```

```
10 guesses left.
```

Так как конструкция `#{...}` может располагаться в любом месте строки, мы можем переместить ее в середину, чтобы вывод стал чуть более понятным для пользователя.

```
puts "You've got #{10 - num_guesses} guesses left."
```

```
You've got 10 guesses left.
```

Теперь игрок знает, сколько попыток у него осталось. А значит, из списка можно вычеркнуть еще один пункт!

```
# Get My Number Game
# Written by: you!

puts "Welcome to 'Get My Number!'"

# Получение имени игрока и вывод приветствия.
print "What's your name? "
input = gets
name = input.chomp
puts "Welcome, #{name}!"

# Сохранение случайного числа
puts "I've got a random number between 1 and 100."
puts "Can you guess it?"
target = rand(100) + 1
```

Наш новый код!

```
# Отслеживание количества попыток.
num_guesses = 0

puts "You've got #{10 - num_guesses} guesses left."
```

Преобразование строк в числа

- [x] Следить за тем, сколько попыток сделал игрок. Перед каждой попыткой сообщить, сколько еще попыток у него осталось (из 10).
- [ ] Предложить игроку ввести число.

Наше следующее требование — предложить игроку ввести число. Итак, игрок вводит число, а затем сохраняет введенное число в переменной. Метод `gets`, как вы помните, получает данные у пользователя. (Мы уже использовали его для получения имени игрока.) К сожалению, метод `gets` возвращает строку, поэтому сразу получить число не удастся. Проблема возникает позднее, когда мы попытаемся сравнить введенное число с загаданным при помощи операторов > и <.

```
print "Make a guess: "
guess = gets
guess < target
guess > target
```

В каждом из этих сравнений происходит ошибка!

Строку, полученную от метода `gets`, необходимо преобразовать в число, чтобы сравнить его с загаданным числом. Проще простого! У строк существует метод `to_i`, который выполнит преобразование за нас.

Приведенный ниже код вызывает `to_i` для строки, полученной `gets`. Строку даже не нужно сохранять в переменной; мы просто воспользуемся оператором «точка», чтобы вызвать метод прямо для возвращаемого значения.

```
guess = gets.to_i
```

При вызове метода `to_i` для строки игнорируются все нецифровые символы, следующие за числом. А это означает, что нам даже не придется удалять символ новой строки, оставшийся от `gets`.

преобразования

Вызывая этот метод для объекта...	*...вы получаете объект этого типа.*
`to_s`	строка
`to_i`	целое число
`to_f`	вещественное число

Чтобы проверить внесенные изменения, можно вывести результат сравнения.

```
puts guess < target
```

true

Так гораздо лучше — число, введенное пользователем, сравнивается с загаданным. Еще одно требование выполнено!

Наш новый код!

```
...

# Store a random number for the player to guess.
puts "I've got a random number between 1 and 100."
puts "Can you guess it?"
target = rand(100) + 1

# Track how many guesses the player has made.
num_guesses = 0

puts "You've got #{10 - num_guesses} guesses left."
print "Make a guess: "
guess = gets.to_i
```

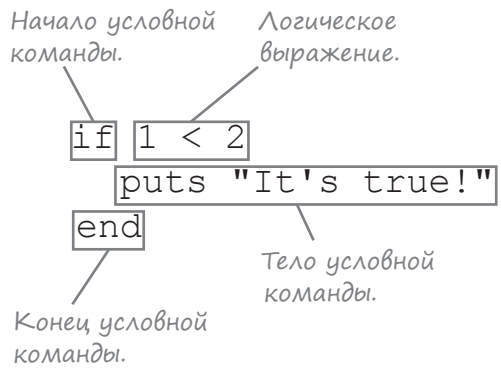
Условные команды

Еще два требования закрыты, осталось четыре! Переходим к следующей группе.

- [x] Предложить игроку ввести число.
- [ ] Если введенное игроком число меньше загаданного, вывести сообщение: «Oops. Your guess was LOW». Если введенное число больше загаданного, вывести сообщение: «Oops. Your guess was HIGH».
- [ ] Если введенное число совпадает с загаданным, вывести сообщение: «Good job, [имя]! You guessed my number in [число попыток] guesses!»
- [ ] Если игрок потратил все попытки, но так и не угадал, вывести сообщение: «Sorry. You didn't get my number. My number was [загаданное число]».

Теперь число, введенное пользователем, необходимо сравнить с загаданным. *Если* оно слишком велико, выводится соответствующее сообщение. *Иначе, если* число слишком мало, выводится другое сообщение, и так далее... Похоже, нам нужна возможность выполнения отдельных фрагментов кода только при выполнении определенных *условий*.

Как и во многих языках, в Ruby существуют **условные** команды, выполняемые только при наличии определенных условий. Программа проверяет выражение и, если его результат оказывается истинным, выполняет код тела условной команды. В противном случае тело команды пропускается.



Как и многие другие языки, Ruby позволяет определить несколько ветвей в условной команде. Такие команды записываются в форме `if/elsif/else`.

Обратите внимание: в середине «elsif» нет второй буквы «e»!

```
if score == 100
  puts "Perfect!"
elsif score >= 70
  puts "You pass!"
else
  puts "Summer school time!"
end
```

Чтобы определить, должен ли выполняться код, условные команды используют **логические** (булевские) выражения. В Ruby существуют константы, представляющие два логических значения: `true` и `false`.

```
if true
  puts "I'll be printed!"
end
```

```
if false
  puts "I won't!"
end
```

Условные команды (продолжение)

В Ruby также поддерживаются все операторы сравнения, знакомые вам по другим языкам.

```
if 1 == 1
  puts "I'll be printed!"
end

if 1 > 2
  puts "I won't!"
end

if 1 < 2
  puts "I'll be printed!"
end
```

```
if 1 >= 2
  puts "I won't!"
end

if 2 <= 2
  puts "I'll be printed!"
end

if 2 != 2
  puts "I won't!"
end
```

Означает «не равно».

В Ruby поддерживается оператор логического отрицания «!», который превращает значение true в false (или наоборот.) Также существует ключевое слово not, которое лучше читается, но делает практически то же самое.

```
if ! true
  puts "I won't be printed!"
end

if ! false
  puts "I will!"
end
```

```
if not true
  puts "I won't be printed!"
end

if not false
  puts "I will!"
end
```

Если вам нужно проверить, что истинны *оба* условия, используйте оператор «&&» («И»). Если вы хотите проверить, что истинно *хотя бы одно* из двух условий, используйте оператор «||» («ИЛИ»).

```
if true && true
  puts "I'll be printed!"
end

if true && false
  puts "I won't!"
end
```

```
if false || true
  puts "I'll be printed!"
end

if false || false
  puts "I won't!"
end
```

Я заметил отступы в коде между if и end. Они обязательны?

Отступ на два пробела!

```
if true
  puts "I'll be printed!"
end
```

Нет, в Ruby отступы не имеют специального смысла (в отличие от некоторых языков — например Python).

Однако наличие отступов в командах if, циклах, методах, классах и т. д. — всего лишь признак хорошего стиля программирования. Отступы помогут разобраться в структуре кода другим разработчикам (и даже вам).

Число, введенное пользователем, нужно сравнить с загаданным случайным числом. Воспользуемся всем, что вы узнали об условных командах, для реализации этой группы требований.

get_number.rb

```
# Get My Number Game
# Written by: you!

puts "Welcome to 'Get My Number!'"

# Получение имени игрока и вывод приветствия.
print "What's your name? "
input = gets
name = input.chomp
puts "Welcome, #{name}!"

# Сохранение случайного числа.
puts "I've got a random number between 1 and 100."
puts "Can you guess it?"
target = rand(100) + 1

# Отслеживание количества попыток.
num_guesses = 0

# Признак продолжения игры.
guessed_it = false

puts "You've got #{10 — num_guesses} guesses left."
print "Make a guess: "
guess = gets.to_i

# Сравнение введенного числа с загаданным
# и вывод соответствующего сообщения.
if guess < target
  puts "Oops. Your guess was LOW."
elsif guess > target
  puts "Oops. Your guess was HIGH."
elsif guess == target
  puts "Good job, #{name}!"
  puts "You guessed my number in #{num_guesses} guesses!"
  guessed_it = true
end

# Если попыток не осталось, сообщить загаданное число.
if not guessed_it
  puts "Sorry. You didn't get my number. (It was #{target}.)"
end
```

Эта переменная добавлена для проверки того, нужно ли выводить сообщение о проигрыше. Также она будет использоваться позднее для остановки игры в том случае, если число угадано верно.

А вот и условные команды!

Позднее будет описан менее громоздкий способ записи этой логики.

«unless» как противоположность «if»

Эта команда работает, но читать ее неудобно:

```
if not guessed_it
  puts "Sorry. You didn't get my number. (It was #{target}.)"
end
```

Во многих отношениях условные команды Ruby похожи на условные команды других языков. Однако в Ruby также существует дополнительное ключевое слово `unless`.

Код команды `if` выполняется только в том случае, если условие *истинно*. С другой стороны, код команды `unless` выполняется только в том случае, если условие *ложно*.

```
unless true
  puts "I won't be printed!"
end
```

```
unless false
  puts "I will!"
end
```

Ключевое слово `unless` — один из примеров того, как язык Ruby упрощает чтение кода. Его можно использовать тогда, когда условие с оператором отрицания выглядит громоздко. Конструкция:

```
if ! (light == "red")
  puts "Go!"
end
```

записывается в следующем виде:

```
unless light == "red"
  puts "Go!"
end
```

Ключевое слово `unless` поможет упростить последнюю условную команду.

```
unless guessed_it
  puts "Sorry. You didn't get my number. (It was #{target}.)"
end
```

Получилось намного понятнее! При этом условные команды работают так же, как прежде!

Если запустить сценарий get_number.rb сейчас, результат будет выглядеть примерно так...

```
File Edit Window Help
$ ruby get_number.rb
Welcome to 'Get My Number!'
What's your name? Jay
Welcome, Jay!
I've got a random number between 1 and 100.
Can you guess it?
You've got 10 guesses left.
Make a guess: 50
Oops. Your guess was HIGH.
Sorry. You didn't get my number. (It was 34.)
```

В текущем состоянии программа предоставляет игроку всего одну попытку, а должно быть 10. Сейчас мы исправим этот недостаток...

Житейская мудрость

Ключевые слова `else` и `elsif` могут использоваться с `unless` в языке Ruby:

```
unless light == "red"
  puts "Go!"
else
  puts "Stop!"
end
```

Странно!

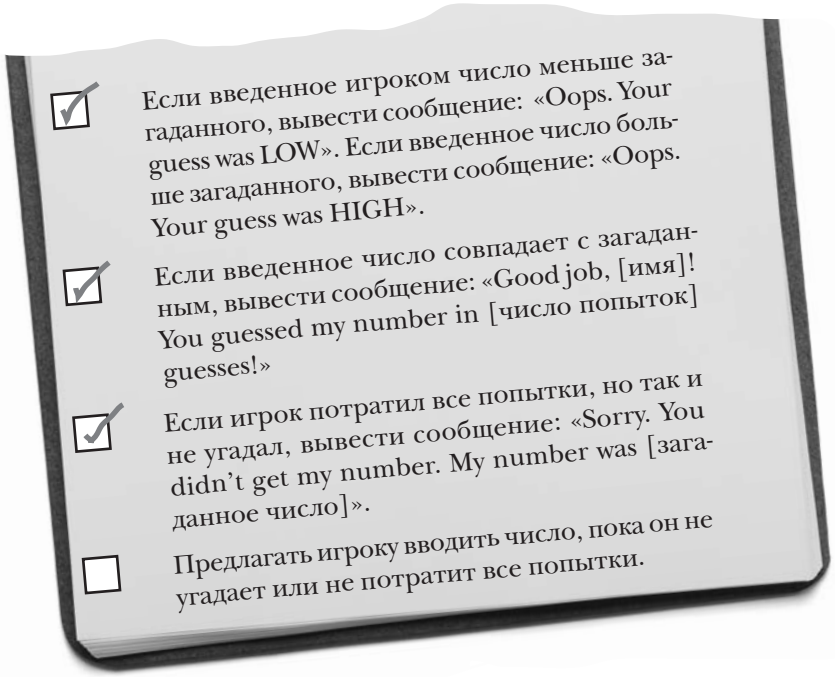
Но такой код читается очень плохо. Если вам нужна секция `else`, лучше используйте `if` в основной ветви!

```
if light == "red"
  puts "Stop!"
else
  puts "Go!"
end
```

Переместилось в основную ветвь.

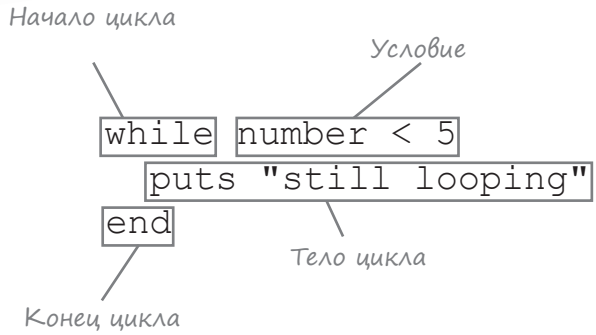
Циклы

Пока все замечательно! В нашей игре осталось реализовать всего одно требование!



В настоящее время игроку дается всего одна попытка. Угадать одно число из ста возможных непросто, так что такую игру честной не назовешь. Игрок должен угадывать 10 раз — или до тех пор, пока не получит правильный ответ (в зависимости от того, что произойдет раньше).

Код, запрашивающий у пользователя число, уже готов. Просто нам нужно выполнить его *многократно*. Для повторного выполнения сегмента кода можно использовать **цикл**. Возможно, вы уже встречали циклы в других языках программирования. Если программист хочет, чтобы одна или несколько команд выполнялись снова и снова, он помещает их в цикл.



Цикл `while` состоит из ключевого слова `while`, логического выражения (как и в командах `if` и `unless`), выполняемого кода и ключевого слова `end`. Код в теле цикла продолжает выполняться, *пока* условие остается истинным.

В следующем простом примере цикл используется для вывода последовательных значений.

```
number = 1
while number <= 5
  puts number
  number += 1
end
```

```
1
2
3
4
5
```

Как вы уже знаете, у `if` есть «двойник» `unless`. Точно так же у `while` есть свой «двойник» — цикл `until`. Цикл `until` повторяется до того момента, *когда* условие станет истинным (то есть цикл продолжает выполняться, пока его условие остается ложным).

Ниже приведен аналогичный пример с использованием `until`.

```
number = 1
until number > 5
  puts number
  number += 1
end
```

```
1
2
3
4
5
```

Перед вами тот же условный код, измененный для выполнения в цикле `while`:

Цикл останавливается после 10-й попытки или после того, как число будет угадано правильно (в зависимости от того, что произойдет раньше).

```
# Отслеживание количества попыток
num_guesses = 0

# Признак продолжения игры.
guessed_it = false

while num_guesses < 10 && guessed_it == false
```

Этот код совершенно не изменился; просто мы поместили его в цикл.

```
  puts "You've got #{10 — num_guesses} guesses left."
  print "Make a guess: "
  guess = gets.to_i
```

При каждом выполнении цикла счетчик увеличивается на 1, чтобы цикл не выполнялся бесконечно.

```
  num_guesses += 1
```

Здесь тоже ничего не изменилось.

```
  # Сравнение введенного числа с загаданным
  # и вывод соответствующего сообщения.
  if guess < target
    puts "Oops. Your guess was LOW."
  elsif guess > target
    puts "Oops. Your guess was HIGH."
  elsif guess == target
    puts "Good job, #{name}!"
    puts "You guessed my number in #{num_guesses} guesses!"
    guessed_it = true
  end
```

Это ключевое слово обозначает конец кода, который будет выполняться в цикле.

```
end

unless guessed_it
  puts "Sorry. You didn't get my number. (It was #{target}.)"
end
```

Осталось еще одно улучшение, которое упростит чтение кода. Как и в случае с командой `if`, которая была заменена командой `unless`, этот цикл `while` можно сделать более понятным. Для этого его следует преобразовать в цикл `until`.

До:

```
while num_guesses < 10 && guessed_it == false
  ...
end
```

После:

```
until num_guesses == 10 || guessed_it
  ...
end
```

Полный код нашей игры.

get_number.rb

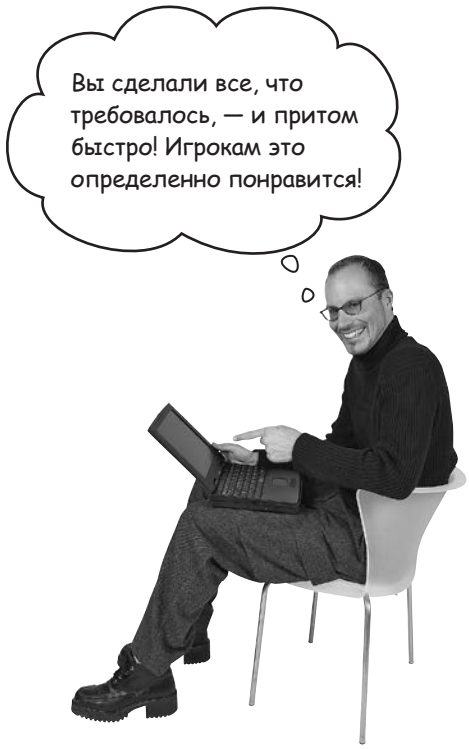
```
# Get My Number Game
# Written by: you!

puts "Welcome to 'Get My Number!'"

# Получение имени игрока и вывод приветствия.
print "What's your name? "
input = gets
name = input.chomp
puts "Welcome, #{name}!"

# Сохранение случайного числа.
puts "I've got a random number between 1 and 100."
puts "Can you guess it?"
target = rand(100) + 1

# Отслеживание количества попыток.
num_guesses = 0

# Признак продолжения игры.
guessed_it = false

until num_guesses == 10 || guessed_it

  puts "You've got #{10 — num_guesses} guesses left."
  print "Make a guess: "
  guess = gets.to_i

  num_guesses += 1

  # Сравнение введенного числа с загаданным
  # и вывод соответствующего сообщения.
  if guess < target
    puts "Oops. Your guess was LOW."
  elsif guess > target
    puts "Oops. Your guess was HIGH."
  elsif guess == target
    puts "Good job, #{name}!"
    puts "You guessed my number in #{num_guesses} guesses!"
    guessed_it = true
  end

end

# Если попыток не осталось, сообщить загаданное число.
unless guessed_it
  puts "Sorry. You didn't get my number. (It was #{target}.)"
end
```

Попробуем запустить игру!

Цикл готов, а последнее требование выполнено! Давайте откроем терминальное окно и попробуем запустить программу.

- ☑ Предлагать игроку вводить число, пока он не угадает или не потратит все попытки.

```
File Edit Window Help Cheats
$ ruby get_number.rb
Welcome to 'Get My Number!'
What's your name? Gary
Welcome, Gary!
I've got a random number between 1 and 100.
Can you guess it?
You've got 10 guesses left.
Make a guess: 50
Oops. Your guess was LOW.
You've got 9 guesses left.
Make a guess: 75
Oops. Your guess was HIGH.
You've got 8 guesses left.
Make a guess: 62
Oops. Your guess was HIGH.
You've got 7 guesses left.
Make a guess: 56
Oops. Your guess was HIGH.
You've got 6 guesses left.
Make a guess: 53
Good job, Gary!
You guessed my number in 5 guesses!
$
```



Итак, мы написали полноценную игру на Ruby с использованием переменных, строк, вызовов методов, условных команд и циклов! А самое замечательное, что программа заняла менее 30 строк кода! Налейте себе чего-нибудь прохладительного, вы это заслужили.

Ваш инструментарий Ruby

Глава 1 осталась позади. В ней ваш инструментарий Ruby пополнился вызовами методов, условными командами и циклами.

Команды

Условные команды выполняют содержащийся в них код в зависимости от некоторого условия.

Циклы выполняют содержащийся в них код многократно. Цикл прерывается по некоторому условию.

Далее в программе…

Весь код сейчас свален в одну большую кучу. В следующей главе вы узнаете, как разбить код на простые, удобные блоки при помощи классов и методов.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Ruby — интерпретируемый язык. Код Ruby не нужно компилировать перед выполнением.
- Переменные не нужно объявлять перед присваиванием им значений. Также не нужно указывать тип переменной.
- Ruby рассматривает все символы от # до конца строки как комментарий и игнорирует их.
- Текст, заключенный в кавычки, рассматривается как строка, то есть последовательность символов.
- Если в строку Ruby входит конструкция `#{...}`, то выражение в фигурных скобках интерполируется (подставляется) в строку.
- При вызове методов *могут* передаваться аргументы, разделенные запятыми.
- Заключать список аргументов в круглые скобки не обязательно. Если аргументы отсутствуют, не ставьте пустые круглые скобки.
- Используйте методы `inspect` и `p` для просмотра отладочной информации по объектам Ruby.
- Для включения специальных символов в строки, заключенные в двойные кавычки, используются служебные последовательности (такие, как `\n` и `\t`).
- Интерактивный интерпретатор Ruby (или `irb`) позволяет быстро проверить результаты выражений Ruby.
- Вызов метода `to_s` для (почти) любого объекта возвращает строковое представление объекта. Вызов `to_i` для строки преобразует ее в целое число.
- `unless` — противоположность `if`; код выполняется в том случае, *если* условие *ложно*.
- `until` — противоположность `while`; цикл выполняется многократно, *пока* условие не станет истинным.

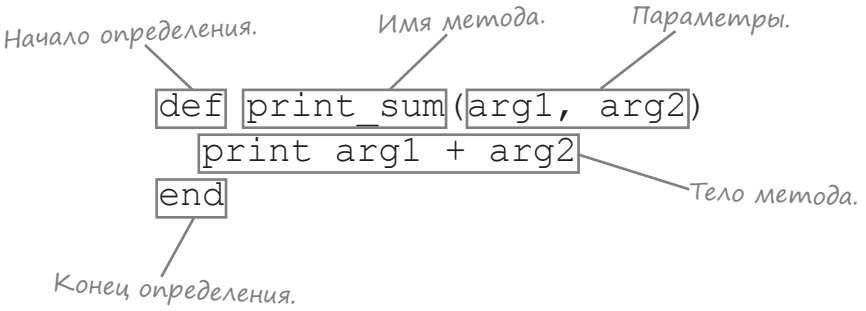
Наводим порядок

В вашей работе кое-чего не хватало. Да, вы вызывали методы и создавали объекты, как настоящий профессионал. Но при этом вы могли вызывать только те методы и создавать только те виды объектов, которые были определены за вас в Ruby. Теперь ваша очередь. В этой главе вы научитесь создавать *свои* методы, а также свои **классы** — своего рода «шаблоны» для создания новых объектов. *Вы сами решаете*, как будут выглядеть объекты, созданные на базе вашего класса. **Переменные экземпляра** определяют, какая информация *хранится* в ваших объектах, а **методы экземпляра** определяют, что эти объекты *делают*. А самое главное — вы узнаете, как определение собственных классов *упрощает чтение и сопровождение вашего кода*.

Определение методов

Компания Got-A-Motor, Inc. работает над приложением виртуального тест-драйва: клиент опробует новую машину прямо на своем компьютере, без похода в автосалон. В первой версии приложения компания хочет видеть методы, с помощью которых пользователь сможет нажать виртуальную педаль газа, подать виртуальный сигнал и включить виртуальные фары в режиме ближнего или дальнего света.

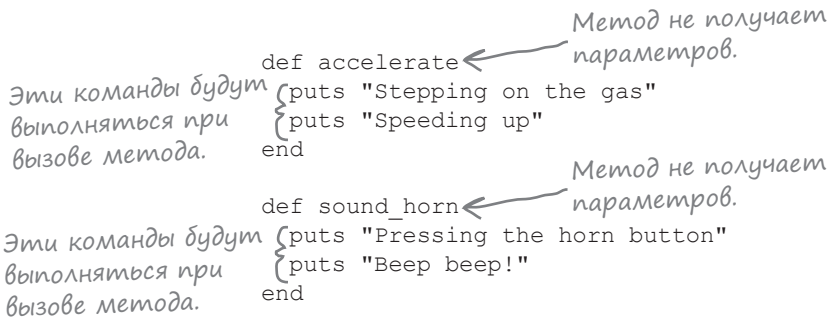
Определения методов в языке Ruby выглядят примерно так:



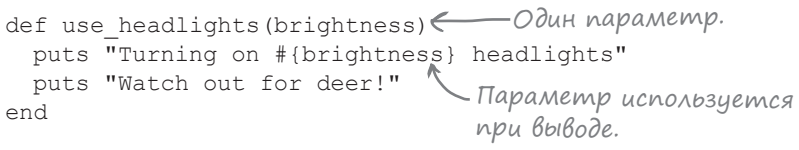
Если вы хотите, чтобы при вызове вашего метода передавались аргументы, в определение метода необходимо добавить **параметры**. Параметры следуют после имени метода и заключаются в круглые скобки. (При отсутствии параметров круглые скобки можно не указывать.) Каждый аргумент в вызове метода сохраняется в одном из параметров внутри метода.

Тело метода состоит из одной или нескольких команд Ruby, которые выполняются при вызове метода. Давайте создадим методы, представляющие действия пользователя в приложении виртуального тест-драйва.

Перед вами два метода для наращивания скорости и подачи сигнала. Пожалуй, это едва ли не самые простые методы, которые можно написать на Ruby: тело каждого метода состоит из пары команд, которые выводят строки.



Метод `use_headlights` ненамного сложнее; он получает один параметр, который интерполируется в одну из выходных строк.



Вот и всё! Определения готовы, теперь методы можно вызывать в программе.

Вызов методов

Методы, определенные вами, вызываются точно так же, как и любые другие методы. Давайте попробуем вызвать новые методы в автомобильном приложении.

Ruby позволяет размещать вызовы методов где угодно — даже в том исходном файле, в котором эти методы определяются. Пока наша программа очень проста, мы именно так и поступим — просто для удобства. Вызовы методов будут размещены сразу же после их объявлений.

```
def accelerate
  puts "Stepping on the gas"
  puts "Speeding up"
end

def sound_horn
  puts "Pressing the horn button"
  puts "Beep beep!"
end

def use_headlights(brightness)
  puts "Turning on #{brightness} headlights"
  puts "Watch out for deer!"
end

sound_horn
accelerate
use_headlights("high-beam")
```

Вызовы без аргументов.

Передается как значение аргумента «brightness».

vehicle_methods.rb

Запустите исходный файл в терминальном окне — вы увидите результат вызова новых методов!

```
File Edit Window Help
$ ruby vehicle_methods.rb
Pressing the horn button
Beep beep!
Stepping on the gas
Speeding up
Turning on high-beam headlights
Watch out for deer!
$
```

Я заметил, что мы не использовали оператор «точка» для определения получателя при этих вызовах, как и в случае с методами `puts` и `print`.

Верно. Как и методы `puts` и `print`, эти методы включаются в среду выполнения верхнего уровня.

Методы, определяемые вне любых классов (как в нашем примере), включаются в среду выполнения верхнего уровня. Как было показано в главе 1, их можно вызывать в любом месте кода *без* указания получателя с использованием оператора «точка».

Имена методов

Имя метода может состоять из нескольких слов, записанных символами нижнего регистра и разделенных символами подчеркивания (по тем же правилам, что и имена переменных). Цифры в именах методов допустимы, но используются редко.

Также имя метода может завершаться вопросительным (?) или восклицательным знаком (!). Такие суффиксы не имеют специального смысла в языке Ruby, но, по общепринятым соглашениям, именам, возвращающим логическое значение (true/false), присваиваются имена, завершающиеся знаком «?», а методам с возможными неожиданными побочными эффектами присваиваются имена, завершающиеся знаком «!».

Наконец, имя метода может завершаться знаком равенства (=). Методы, имена которых завершаются этим символом, используются для назначения атрибутов (мы рассмотрим их позднее, когда будем рассматривать классы). В языке Ruby этот суффикс *имеет* специальный смысл, поэтому не применяйте его в обычных методах — или вы увидите, что ваш метод ведет себя странно!

Параметры

Если методу должны передаваться данные, укажите после имени метода один или несколько параметров, разделенных запятыми. В теле метода к параметрам можно обращаться точно так же, как к любым другим переменным.

```
def print_area(length, width)
  puts length * width
end
```

Необязательные параметры

Разработчики из Got-A-Motor довольны вашей работой над системой виртуального тест-драйва... в основном.

А указывать аргумент при вызове метода `use_headlights` **обязательно**? Мы почти всегда используем значение `"low-beam"`, и эта строка повторяется в нескольких местах кода!

```
use_headlights("low-beam")
stop_engine
buy_coffee
start_engine
use_headlights("low-beam")
accelerate
create_obstacle("deer")
use_headlights("high-beam")
```

Житейская мудрость

Имена методов следует записывать в «змеином» стиле: одно или несколько слов в нижнем регистре, разделенных подчеркиваниями (как в именах переменных).

```
def bark
end

def wag_tail
end
```

Как и при вызове методов, не ставьте круглые скобки в определении метода при отсутствии параметров. Пожалуйста, не делайте так (притом что эта запись формально допустима):

```
def no_args()
  puts "Bad Rubyist!"
end
```

Но если параметры передаются, круглые скобки ставятся всегда. (В главе 1 упоминались исключения, относящиеся к вызову методов, но при объявлении методов исключений не бывает.) Формально круглые скобки можно опустить, но мы еще раз говорим: не надо так делать.

```
def with_args first, second
  puts "No! Bad!"
end
```

Необязательные параметры (продолжение)

Ситуация достаточно распространенная — один конкретный аргумент используется в 90% случаев, и вам надоело его повторять. Но и убрать его тоже нельзя, потому что в 10% случаев используется другое значение.

У проблемы есть простое решение: *сделайте параметр необязательным*. В объявлении метода можно задать для него значение по умолчанию.

Пример метода, использующего значения по умолчанию для некоторых параметров:

```
def order_soda(flavor, size = "medium", quantity = 1)
  if quantity == 1
    plural = "soda"
  else
    plural = "sodas"
  end
  puts "#{quantity} #{size} #{flavor} #{plural}, coming right up!"
end
```

Значение по умолчанию для size.

Значение по умолчанию для quantity.

Если вам потребуется заменить значение по умолчанию, просто передайте аргумент с нужным значением. А если вас устраивает значение по умолчанию, аргумент просто не указывается.

Задается параметр flavor; для size и quantity используются значения по умолчанию.

```
order_soda("orange")
order_soda("lemon-lime", "small", 2)
order_soda("grape", "large")
```

Задаются все значения.

Задаются параметры flavor и size; для quantity используется значение по умолчанию.

```
1 medium orange soda, coming right up!
2 small lemon-lime sodas, coming right up!
1 large grape soda, coming right up!
```

При использовании необязательных параметров необходимо помнить об одном правиле: они должны следовать *после* всех остальных параметров. Если обязательный параметр следует в списке за необязательным, то опустить необязательный параметр не удастся:

```
def order_soda(flavor, size = "medium", quantity)
  ...
end

order_soda("grape")
```

Необязательный параметр не должен находиться перед обязательным!

Ошибка →

```
wrong number of
arguments (1 for 2..3)
```

Часто Задаваемые Вопросы

В: Чем аргументы отличаются от параметров?

О: *Параметры* определяются и используются в *определении* метода. *Аргументы* передаются при *вызове* метода.

Параметр

```
def say_hello(name)
  puts "Hello, #{name}!"
end
```

Параметр

```
say_hello("Marcy")
```

Аргумент

Каждый аргумент, передаваемый при вызове метода, сохраняется на месте соответствующего параметра.

Эти два термина в основном используются для того, чтобы понять, о чем идет речь: об *определении* или о *вызове* метода.

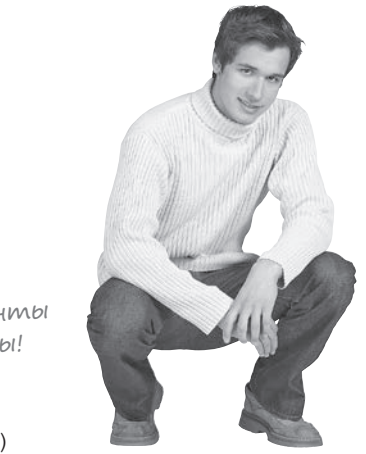
Необязательные параметры (продолжение)

Так давайте сделаем параметр `use_headlights` необязательным, чтобы упростить жизнь разработчикам, использующим наши методы.

```
def use_headlights(brightness = "low-beam")
  puts "Turning on #{brightness} headlights"
  puts "Watch out for deer!"
end
```

Теперь указывать значение не обязательно, если только не понадобится режим high-beam.

```
use_headlights                      ← Использовать значение по умолчанию «low-beam».
use_headlights("high-beam")         ← Переопределить значение по умолчанию.
```




```
Turning on low-beam headlights
Watch out for deer!
Turning on high-beam headlights
Watch out for deer!
```

```
use_headlights      ←
stop_engine           Аргументы
start_engine          не нужны!
use_headlights      ←
accelerate
use_headlights("high-beam")
```

Упражнение

Наши методы для приложения виртуального тест-драйва готовы. Попробуем загрузить их в irb и опробовать в деле.

Шаг 1: Сохраните определения методов в файле с именем *vehicle_methods.rb*.

Шаг 2: Откройте терминальное окно и перейдите в каталог, в котором был сохранен файл.

```
def accelerate
  puts "Stepping on the gas"
  puts "Speeding up"
end

def sound_horn
  puts "Pressing the horn button"
  puts "Beep beep!"
end

def use_headlights(brightness = "low-beam")
  puts "Turning on #{brightness} headlights"
  puts "Watch out for deer!"
end
```

vehicle_methods.rb

Упражнение (продолжение)

Шаг 3: Так как мы собираемся загрузить в irb код из файла, интерпретатор должен иметь возможность загружать файлы Ruby из текущего каталога. По этой причине команда запуска irb на этот раз будет выглядеть немного иначе.

Введите следующую команду в терминальном окне и нажмите Enter:

```
irb -I .
```

Это означает: «искать файлы для загрузки в текущем каталоге».

`-I` — *параметр командной строки*, который добавляется в команду для изменения режима ее выполнения. В данном случае `-I` изменяет набор каталогов, в которых Ruby ищет загружаемые файлы. Точка (`.`) обозначает текущий каталог.

Шаг 4: К этому моменту интерпретатор irb запущен, а мы можем загрузить файл с нашими методами. Введите следующую строку:

```
require "vehicle_methods"
```

Ruby знает, что по умолчанию следует искать файлы с расширением *.rb*, поэтому расширение в команде не указано. Если будет выведен результат `true`, значит, файл загрузился успешно.

Теперь вы сможете ввести команду вызова любого из наших методов, и этот метод будет выполнен!

Пример:

```
File Edit Window Help
$ irb -I .
irb(main):001:0> require "vehicle_methods"
 => true
irb(main):002:0> sound_horn
Pressing the horn button
Beep beep!
 => nil
irb(main):003:0> use_headlights
Turning on low-beam headlights
Watch out for deer!
 => nil
irb(main):004:0> use_headlights("high-beam")
Turning on high-beam headlights
Watch out for deer!
 => nil
irb(main):005:0> exit
$
```

Возвращаемые значения

Теперь компания Got-A-Motor хочет, чтобы в приложении выводилась информация об эффективности использования топлива. Приложение должно выводить расход топлива за последнюю поездку, а также средний расход топлива за весь срок эксплуатации.

В первом случае расстояние с путевого одометра делится на количество галлонов с последней заправки, а во втором — значение с основного одометра делится на количество галлонов за весь срок эксплуатации. Но в обоих случаях мы берем расстояние в милях и делим его на количество галлонов. Так нужно ли писать два разных метода?

Нет! Как и в большинстве языков, методы Ruby имеют **возвращаемое значение**, которое передается вызвавшему их коду. Метод Ruby возвращает значение вызывающей стороне с помощью ключевого слова `return`.

Методы возвращают значение в код, из которого они были вызваны.

Напишем один метод `mileage` и используем его возвращаемое значение в выводе.

```
def mileage(miles_driven, gas_used)
  return miles_driven / gas_used
end
```

И тогда один метод можно будет использовать для вычисления обоих видов расхода топлива.

```
trip_mileage = mileage(400, 12)
puts "You got #{trip_mileage} MPG on this trip."

lifetime_mileage = mileage(11432, 366)
puts "This car averages #{lifetime_mileage} MPG."
```

```
You got 33 MPG on this trip.
This car averages 31 MPG.
```

Неявные возвращаемые значения

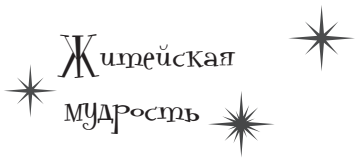
Ключевое слово `return` в этом методе не обязательно. Значение последнего выражения, вычисленного в методе, автоматически становится возвращаемым значением этого метода. Таким образом, метод `mileage` можно переписать без явного включения команды `return`:

```
def mileage(miles_driven, gas_used)
  miles_driven / gas_used
end
```

Работать он будет точно так же.

```
puts mileage(400, 12)
```

```
33
```



Программисты, работающие на Ruby, обычно предпочитают неявные возвращаемые значения. В коротком методе нет смысла использовать запись:

```
def area(length, width)
  return length * width
end
```

...когда можно просто написать:

```
def area(length, width)
  length * width
end
```

Раннее возвращение из метода

Существуют обстоятельства, в которых ключевое слово `return` может оказаться полезным.

Ключевое слово `return` приводит к немедленному выходу из метода без выполнения оставшегося кода. Это бывает полезно в ситуациях, когда выполнение этого кода будет бессмысленно или даже вредно.

Например, представьте, что машина совсем новая и пробега у нее еще нет. И расстояние, и затраты топлива могут быть равны нулю. Что произойдет, если вызвать метод `mileage` для такой машины?

Как вы помните, метод `mileage` делит `miles_driven` на `gas_used`... И как вас учили в других языках программирования, деление чего-либо на нуль является ошибкой!

```
puts mileage(0, 0)
```

Ошибка →

```
in `/': divided by 0 (ZeroDivisionError)
```

Как решить эту проблему? Нужно проверить, равно ли значение `gas_used` нулю, и если равно — преждевременно вернуть управление из метода.

```
def mileage(miles_driven, gas_used)
  if gas_used == 0
    return 0.0
  end
  miles_driven / gas_used
end
```

Если бензин еще не расходовался...

...вернуть 0.

Для нулевого значения «gas_used» этот код выполняться не будет.

На этот раз код просто возвращает `0.0`, не пытаясь делить на нуль. Проблема решена!

```
puts mileage(0, 0)
```

```
0.0
```

Методы сильно помогают в упорядочении кода и устранении повторов. Однако самих методов иногда оказывается недостаточно. Оставим на время друзей из Got-A-Motor и займемся вместо машин живыми существами...

Беспорядок в методах

Сотрудники экологической организации Fuzzy Friends Animal Rescue в ходе кампании по привлечению средств решили создать интерактивное приложение. Они обратились в вашу компанию за помощью. В их приложении есть много видов животных, каждое из которых издает некоторые звуки и выполняет некоторые действия.

Они создали методы для моделирования перемещений и звуков. При вызове этих методов в первом аргументе передается тип животного, а за ним следуют любые необходимые дополнительные аргументы.

На данный момент их код выглядит примерно так:

```
def talk(animal_type, name)
  if animal_type == "bird"
    puts "#{name} says Chirp! Chirp!"
  elsif animal_type == "dog"
    puts "#{name} says Bark!"
  elsif animal_type == "cat"
    puts "#{name} says Meow!"
  end
end

def move(animal_type, name, destination)
  if animal_type == "bird"
    puts "#{name} flies to the #{destination}."
  elsif animal_type == "dog"
    puts "#{name} runs to the #{destination}."
  elsif animal_type == "cat"
    puts "#{name} runs to the #{destination}."
  end
end

def report_age(name, age)
  puts "#{name} is #{age} years old."
end
```

Выводимая строка выбирается в зависимости от параметра animal_type.

Этот метод одинаков для всех типов животных, поэтому параметр animal_type не используется.

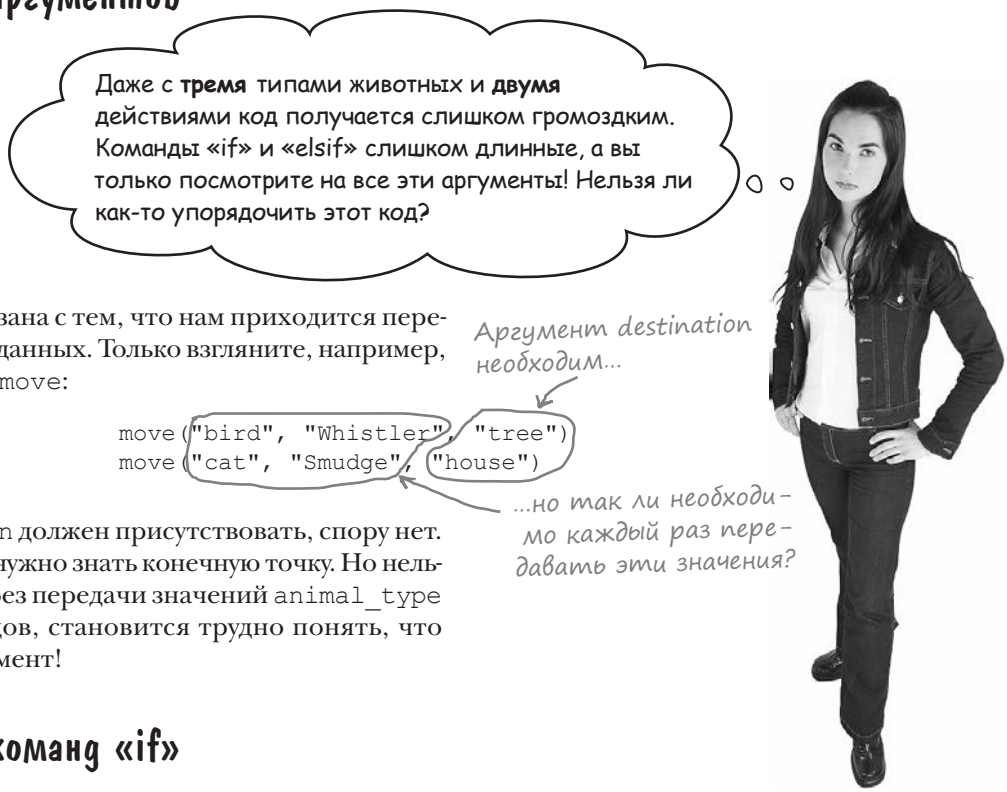
Несколько типичных примеров вызовов этих методов:

```
move("bird", "Whistler", "tree")
talk("dog", "Sadie")
talk("bird", "Whistler")
move("cat", "Smudge", "house")
report_age("Smudge", 6)
```

```
Whistler flies to the tree.
Sadie says Bark!
Whistler says Chirp! Chirp!
Smudge runs to the house.
Smudge is 6 years old.
```

Ребята из Fuzzy Friends хотят совсем немного: добавьте всего 10 типов животных, еще 30 новых действий — и версия 1.0 будет готова!

Слишком много аргументов



Проблема отчасти связана с тем, что нам приходится передавать слишком много данных. Только взгляните, например, на эти вызовы метода `move`:

```
move("bird", "Whistler", "tree")
move("cat", "Smudge", "house")
```

Аргумент destination необходим...

...но так ли необходимо каждый раз передавать эти значения?

Аргумент `destination` должен присутствовать, спору нет. Чтобы перемещаться, нужно знать конечную точку. Но нельзя ли как-то обойтись без передачи значений `animal_type` и `name`? В конце концов, становится трудно понять, что означает каждый аргумент!

Слишком много команд «if»

Кроме того, проблема не ограничивается аргументами методов — *внутри* самих методов все тоже неблагополучно. Только представьте, как будет выглядеть метод `talk`, если добавить еще 10 типов животных...

Каждый раз, когда от вас потребуют звуки, издаваемые животным (а от вас это *потребуют*, можете не сомневаться), вам придется копаться во всех условиях `elsif` в поисках нужного типа животного... А что будет, если код `talk` усложнится — например, в него добавятся анимации и воспроизведение звуковых файлов? Что произойдет, если *все* методы станут такими?

Нам нужен более удобный способ представления типа животного, с которым работает код. Необходимо как-то разбить этот код по типу животного, чтобы упростить его сопровождение. И еще нужен более удобный способ хранения атрибутов каждого отдельного животного (таких, как имя и возраст), чтобы нам не приходилось передавать столько аргументов..

Данные животных и весь код, работающий с этими данными, должны храниться в одном месте. Короче, нам нужны *классы* и *объекты*.

```
def talk(animal_type, name)
  if animal_type == "bird"
    puts "#{name} says Chirp! Chirp!"
  elsif animal_type == "dog"
    puts "#{name} says Bark!"
  elsif animal_type == "cat"
    puts "#{name} says Meow!"
  elsif animal_type == "lion"
    puts "#{name} says Roar!"
  elsif animal_type == "cow"
    puts "#{name} says Moo."
  elsif animal_type == "bob"
    puts "#{name} says Hello."
  elsif animal_type == "duck"
    puts "#{name} says Quack."
  ...
  end
end
```

Для полного списка не хватит места на странице...

Проектирование класса

Основное преимущество объектов заключается в том, что они позволяют хранить набор данных и методы, работающие с этими данными, в одном месте. И эта возможность очень пригодится в приложении Fuzzy Friends.

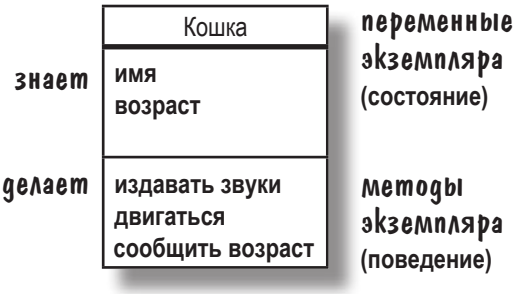
Но чтобы начать создавать собственные объекты, их необходимо сначала определить. **Класс** представляет собой «шаблон» для создания объектов. Когда вы используете класс для создания объекта, класс описывает, что этот объект *знает* о себе и что этот объект *делает*.

Переменные экземпляра:

то, что объект *знает* о себе

Методы экземпляра:

то, что объект *делает*



Экземпляром класса называется объект, созданный на основе класса. Достаточно написать всего *один* класс, а потом создать *много* экземпляров этого класса.

Считайте, что «экземпляр» и «объект» — одно и то же.

Переменными экземпляра называются переменные, принадлежащие одному объекту. Они составляют всю информацию, которая **известна** объекту о нем самом. Переменные экземпляра представляют состояние объекта (его данные) и могут иметь разные значения для разных экземпляров этого класса.

Методами экземпляра называются методы, вызываемые непосредственно для этого объекта. Они составляют все то, что объект **делает**. Методы экземпляра могут обращаться к переменным экземпляра объекта и могут изменять свое поведение в зависимости от значений этих переменных.

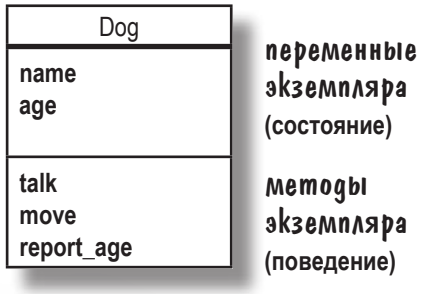
Чем класс отличается от объекта

Класс — шаблон для создания объекта. Класс сообщает Ruby, как следует создать объект этого конкретного типа. У объектов есть переменные экземпляра и методы экземпляра, но эти переменные и методы включаются в класс в процессе его *проектирования*.

Если классы — это формы для печенья, то объекты — это печенье, которое делается по форме.

Каждый экземпляр класса может содержать собственный набор значений для переменных, используемых в методах этого класса. Например, класс `Dog`, представляющий собаку, определяется всего один раз. В методах этого класса `Dog` достаточно указать всего один раз, что каждый экземпляр `Dog` должен содержать переменные для имени (`name`) и возраста (`age`). При этом каждый *объект* `Dog` содержит собственные значения `name` и `age`, отличающиеся от значений остальных экземпляров `Dog`.

Класс для собаки:



Экземпляры Dog:

Первый класс

Перед вами пример класса, который вполне может использоваться в нашем интерактивном приложении: класс `Dog`.

Определение класса начинается с ключевого слова `class`, за которым следует имя нового класса.

В определение класса можно включить определения методов. Любой метод, определяемый здесь, будет доступен в методах экземпляра у экземпляров этого класса.

Конец определения класса обозначается ключевым словом `end`.

```
class Dog

  def talk
    puts "Bark!"
  end

  def move(destination)
    puts "Running to the #{destination}."
  end

end
```

Объявление нового класса.

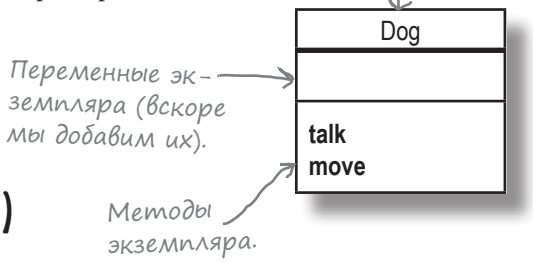
Имя класса.

Метод экземпляра.

Другой метод экземпляра.

Конец объявления класса.

Диаграмма этого класса может выглядеть примерно так...



Создание новых экземпляров (объектов)

Если вызвать для класса метод `new`, он вернет новый экземпляр этого класса. Полученный экземпляр можно присвоить переменной или сделать с ним что-нибудь другое.

```
fido = Dog.new
rex = Dog.new
```

После создания одного или нескольких экземпляров класса можно переходить к вызову их методов экземпляра. Эти методы вызываются точно так же, как и все остальные методы объектов, встречавшиеся до настоящего момента: оператор «точка» определяет, какой экземпляр является получателем данного метода.

```
fido.talk
rex.move("food bowl")
```

```
Bark!
Running to the food bowl.
```

Разбиение больших методов на классы

Для отслеживания типа животного, с которым должен работать метод, в текущей версии приложения используются строки. Кроме того, вся информация о разных вариантах реакции животных встроена в нагромождения команд `if/else`. Такой подход в лучшем случае неудобен и громоздок.

```
def talk(animal_type, name)
  if animal_type == "bird"
    puts "#{name} says Chirp! Chirp!"
  elsif animal_type == "dog"
    puts "#{name} says Bark!"
  elsif animal_type == "cat"
    puts "#{name} says Meow!"
  end
end
```

Объектно-ориентированное решение

Теперь, когда вы научились создавать классы, мы можем применить *объектно-ориентированный* подход к решению задачи. Для каждого *типа* животного создается *класс*. Тогда вместо одного *большого* метода, содержащего описание поведения для *всех* типов животных, мы включим *маленький* метод в *каждый* класс — метод с определением поведения, относящегося к данному типу животного.

Вы можете вызвать «talk» или «move» для любого созданного вами экземпляра Bird!

То же для экземпляров Dog...

То же для экземпляров Cat!

```
class Bird
  def talk
    puts "Chirp! Chirp!"
  end
  def move(destination)
    puts "Flying to the #{destination}."
  end
end

class Dog
  def talk
    puts "Bark!"
  end
  def move(destination)
    puts "Running to the #{destination}."
  end
end

class Cat
  def talk
    puts "Meow!"
  end
  def move(destination)
    puts "Running to the #{destination}."
  end
end
```

Команды if/elsif не нужны!

Примечание: имена животных пока не поддерживаются. Скоро мы этим займемся!

Китейская мудрость

Имена классов Ruby должны начинаться с буквы верхнего регистра. Все символы после первого записываются в нижнем регистре.

```
class Appointment
  ...
end
```

Если имя состоит из нескольких слов, каждое слово тоже должно начинаться с буквы верхнего регистра.

```
class AddressBook
  ...
end

class PhoneNumber
  ...
end
```

Помните схему записи имен переменных (с разделением слов символами подчеркивания), которая называлась «змеиной» записью? Стиль записи имен классов называется «верблюжьим», потому что буквы верхнего регистра напоминают горбы верблюда.

Создание экземпляров новых классов животных

После того как мы определили эти классы, мы сможем создать новые экземпляры этих классов (новые объекты, созданные по шаблону класса) и вызывать для них методы.

Как и в случае с методами, Ruby позволяет создавать экземпляры классов прямо в том файле, в котором они были объявлены. Вероятно, для более крупных приложений такая организация кода не подойдет, но в таком простом приложении, как у нас, можно просто создать несколько экземпляров сразу же после объявлений этих классов.


```
class Bird
  def talk
    puts "Chirp! Chirp!"
  end
  def move(destination)
    puts "Flying to the #{destination}."
  end
end

class Dog
  def talk
    puts "Bark!"
  end
  def move(destination)
    puts "Running to the #{destination}."
  end
end

class Cat
  def talk
    puts "Meow!"
  end
  def move(destination)
    puts "Running to the #{destination}."
  end
end

bird = Bird.new
dog = Dog.new
cat = Cat.new

bird.move("tree")
dog.talk
bird.talk
cat.move("house")
```

Создание новых экземпляров наших классов.

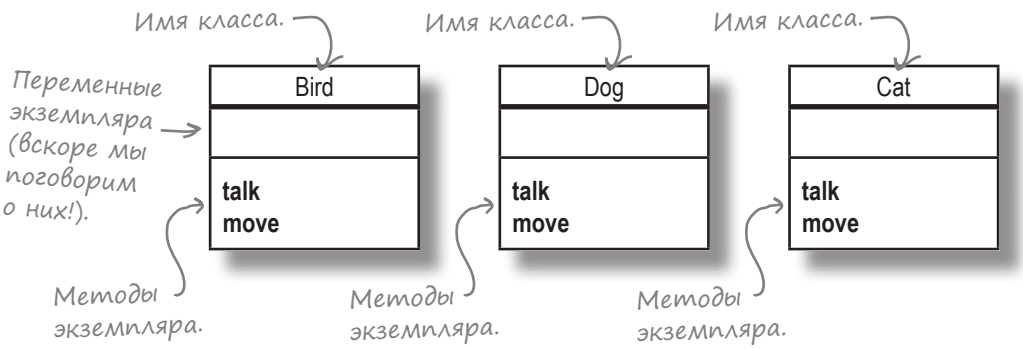
Вызов методов для экземпляров.

```
File Edit Window Help
$ ruby animals.rb
Flying to the tree.
Bark!
Chirp! Chirp!
Running to the house.
$
```

Сохраните весь этот код в файле с именем *animals.rb*, выполните команду `ruby animals.rb` в терминальном окне — и вы увидите результаты выполнения наших методов экземпляра!

Диаграмма классов с методами экземпляра

Если бы нас попросили нарисовать диаграмму классов, входящих в систему, то диаграмма выглядела бы примерно так:



На данный момент экземпляры наших классов содержат два метода экземпляра (то, что они могут *делать*): `talk` и `move`. Впрочем, в них пока нет переменных экземпляра (то, что они *знают*). Сейчас мы займемся этой темой.

Развлечения с магнитами

На холодильнике разложена программа на языке Ruby. Некоторые фрагменты находятся на своих местах, другие были перемешаны случайным образом. Сможете ли вы переставить фрагменты так, чтобы получить работоспособную программу, которая бы выдавала приведенный ниже результат?

```
class Blender
  def close_lid
  end
```

```
(speed)   .blend   ("high")
blender   blender   .close_lid
puts "Sealed tight!"   def   blend
puts "Spinning on #{speed} setting."   end
```

```
end

blender = Blender.new
```

Вывод

```
File Edit Window Help
Sealed tight!
Spinning on high setting.
```

Развлечения с магнитами. Решение

На холодильнике разложена программа на языке Ruby. Некоторые фрагменты находятся на своих местах, другие были перемешаны случайным образом. Сможете ли вы переставить фрагменты так, чтобы получить работоспособную программу, которая бы выдавала приведенный ниже результат?

```
class Blender
    def close_lid
        puts "Sealed tight!"
    end
    def blend (speed)
        puts "Spinning on #{speed} setting."
    end
end

blender = Blender.new
blender.close_lid
blender.blend ("high")
```

Output

```
File Edit Window Help
Sealed tight!
Spinning on high setting.
```

Часто Задаваемые Вопросы

В: Смогу ли я вызывать новые методы `move` и `talk` сами по себе (без объекта)?

О: За пределами класса — нет. Вспомните, что получатель в языке Ruby указывает, для какого объекта вызывается метод. Методы `move` и `talk` являются *методами экземпляра*; вызывать их без указания экземпляра, к которому они должны применяться, попросту бессмысленно. Если вы попробуете это сделать, то получите ошибку:

```
move("food bowl")

undefined method `move' for
main:Object (NoMethodError)
```

В: Вы говорите, что для создания объекта нужно вызвать метод `new` для класса. В главе 1 вы также говорили, что числа и строки в Ruby являются объектами. Почему мы не вызываем `new` для получения нового числа или строки?

О: Разработчикам приходится так часто создавать новые числа и строки, что для этих операций в языке была предусмотрена специальная сокращенная запись: строковые и числовые *литералы*.

```
new_string = "Hello!"
new_float = 4.2
```

Чтобы сделать то же самое для других классов, придется вносить изменения на уровне самого языка Ruby, поэтому для создания новых экземпляров большинство классов использует new. (Впрочем, есть и исключения; вы еще узнаете о них.)

Наши объекты «не знают» свое имя и возраст!

Руководитель проекта из экологической организации напоминает о паре мелочей, о которых мы забыли в своем решении с классами:

При вызове этих методов должно выводиться имя животного! И кстати, где метод `report_age`?

```
Flying to the tree.
Bark!
Chirp! Chirp!
Running to the house.
```

Верно подмечено: мы забыли реализовать некоторые возможности исходной программы.

Начнем с добавления параметра `name` в методы `talk` и `move`:

Имя должно передаваться при вызове методов, как и прежде.

```
class Bird
  def talk(name)
    puts "#{name} says Chirp! Chirp!"
  end
  def move(name, destination)
    puts "#{name} flies to the #{destination}."
  end
end

class Dog
  def talk(name)
    puts "#{name} says Bark!"
  end
  def move(name, destination)
    puts "#{name} runs to the #{destination}."
  end
end

class Cat
  def talk(name)
    puts "#{name} says Meow!"
  end
  def move(name, destination)
    puts "#{name} runs to the #{destination}."
  end
end
```

Имена используются в выводе, здесь тоже ничего не изменилось.

Слишком много аргументов (снова)

После того как мы снова добавили параметр `name` к методам `talk` и `move`, при вызове метода можно снова передавать имя животного для вывода сообщения.

```
dog = Dog.new
dog_name = "Lucy"
dog.talk(dog_name)
dog.move(dog_name, "fence")

cat = Cat.new
cat_name = "Fluffy"
cat.talk(cat_name)
cat.move(cat_name, "litter box")
```

```
Lucy says Bark!
Lucy runs to the fence.
Fluffy says Meow!
Fluffy runs to the litter box.
```

Да ладно. У нас уже есть переменная для хранения объекта animal. И вы еще хотите передавать **вторую** переменную с именем животного? Так неудобно!

```
dog = Dog.new
dog_name = "Lucy"
cat = Cat.new
cat_name = "Fluffy"
```

Вообще-то более удобное решение существует. Можно воспользоваться переменными экземпляра для хранения данных в объекте.

Одно из важнейших преимуществ объектно-ориентированного программирования заключается в том, что данные и методы, которые с этими данными работают, хранятся в одном месте. Попробуем сохранить имена *прямо в объектах* animal, чтобы при вызове методов экземпляров не приходилось передавать столько аргументов.

Локальные переменные существуют до завершения метода

До сих пор мы работали с **локальными переменными** — такие переменные *локальны* по отношению к текущей области видимости (обычно в границах текущего метода). После выхода из текущей области видимости локальные переменные перестают существовать, поэтому они *не могут* использоваться для хранения имен животных.

Перед вами новая версия класса `Dog` с дополнительным методом `make_up_name`. Этот метод сохраняет имя собаки для последующего обращения из метода `talk`.

```
class Dog

  def make_up_name
    name = "Sandy"
  end

  def talk
    puts "#{name} says Bark!"
  end

end
```

Сохраняем имя.

Пытаемся обратиться к сохраненному имени.

Однако при вызове метода `talk` появляется сообщение об ошибке: переменная `name` не существует.

```
dog = Dog.new
dog.make_up_name
dog.talk
```

Ошибка

```
in `talk': undefined local variable or method `name' for #<Dog:0x007fa3188ae428>
```

Что произошло? Ведь мы *определили* переменную `name` в методе `make_up_name`!

Дело в том, что мы использовали локальную переменную. Локальные переменные существуют только до завершения того метода, в котором они были созданы. В данном случае переменная `name` перестает существовать сразу же после завершения `make_up_name`.

```
class Dog

  def make_up_name
    name = "Sandy"
  end

  def talk
    puts "#{name} says Bark!"
  end

end
```

При завершении метода «name» выходит из области видимости.

В этой точке переменная уже не существует!

Поверьте, недолговечность локальных переменных — это *хорошо*. Если бы *любая* переменная была доступна в *любой* точке программы, вы бы *постоянно* путались и обращались не к тем переменным! Как и большинство языков, Ruby ограничивает область видимости переменных для предотвращения подобных ошибок.

Только представьте, что это локальная переменная...

```
def alert_ceo
  message = "Sell your stock."
  email(ceo, message)
end

email(shareholders, message)
```

...будет доступна здесь...

Уф! Пронесло.

Ошибка

```
undefined local variable or method `message'
```

Срок жизни переменной экземпляра

Любая локальная переменная, созданная нами, исчезает сразу же при выходе из области видимости. Но в таком случае как хранить имя собаки вместе с объектом? Для этого понадобится новая разновидность переменных.

Объект позволяет хранить данные в **переменных экземпляра** — то есть переменных, связанных с конкретным экземпляром. Данные, сохраненные в переменных экземпляра, продолжают существовать вместе с объектом и удаляются из памяти только при удалении объекта.

Переменные экземпляра в объекте существуют, пока существует сам объект.

Переменная экземпляра выглядит как обычная переменная, а при выборе ее имени применяются те же соглашения. Единственное отличие: имя переменной экземпляра должно начинаться с символа @ («собака»).

```
my_variable    @my_variable
```

Локальная переменная | **Переменная экземпляра**

Вернемся к классу `Dog`. Он почти не отличается от предыдущего, если не считать двух символов `@`, превращающих *две* локальные переменные в *одну* переменную экземпляра.

Сохраняем значение в переменной экземпляра.

```
class Dog

  def make_up_name
    @name = "Sandy"
  end

  def talk
    puts "#{@name} says Bark!"
  end

end
```

Обращаемся к переменной экземпляра.

Теперь точно такой же вызов `talk`, как в предыдущем случае, прекрасно работает! Переменная экземпляра `@name`, созданная в методе `make_up_name`, остается доступной в методе `talk`.

```
dog = Dog.new
dog.make_up_name
dog.talk
```

```
Sandy says Bark!
```

Срок жизни переменной экземпляра (продолжение)

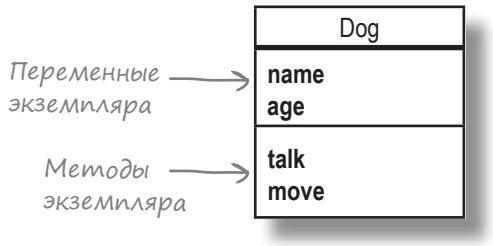
С переменными экземпляра мы также сможем легко добавить методы `move` и `report_age`...

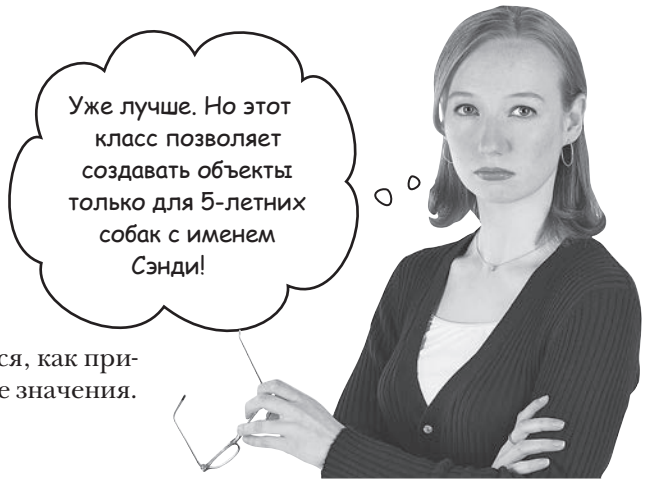
```
class Dog

  def make_up_name
    @name = "Sandy"
  end

  def talk
    puts "#{@name} says Bark!"
  end

  def move(destination)
    puts "#{@name} runs to the #{destination}."
  end

  def make_up_age
    @age = 5
  end

  def report_age
    puts "#{@name} is #{@age} years old."
  end

end

dog = Dog.new
dog.make_up_name
dog.move("yard")
dog.make_up_age
dog.report_age
```

Новый код!

```
Sandy runs to the yard.
Sandy is 5 years old.
```

И теперь мы наконец-то сможем заполнить пустые места на диаграмме класса `Dog`!





Что верно, то верно. Пора разобраться, как присвоить переменным `name` и `age` другие значения.

Инкапсуляция

Благодаря переменным экземпляра мы теперь можем сохранять имена и возраст животных. Однако методы `make_up_name` и `make_up_age` methods допускают использование только фиксированных значений (которые не могут изменяться во время выполнения программы). Нужно сделать так, чтобы программа могла присвоить переменным любые необходимые значения.

```
class Dog

  def make_up_name
    @name = "Sandy"
  end

  def make_up_age
    @age = 5
  end
...
end
```

Впрочем, *такой* код работать не будет:

```
fido = Dog.new
fido.@age = 3
```

Ошибка

```
syntax error, unexpected tIVAR
```

Ruby не позволяет напрямую обращаться к переменным экземпляра за пределами класса. И это вовсе не прихоть; такие ограничения существуют для того, чтобы другие программы и классы не изменяли ваши переменные экземпляра как попало.

Представьте, что переменные экземпляра *возможно* было бы изменять напрямую. Что помешает другим частям программы присвоить им некорректные значения?

Этот код некорректен!

```
fido = Dog.new
fido.@name = ""
fido.@age = -1
fido.report_age
```

Если бы это МОЖНО было сделать, то результат выглядел бы так...

```
is -1 years old.
```

Что? Простите, *кому* и *сколько* лет? Данные объекта явно недействительны, как хорошо видно из вывода программы!

Пустые имена и отрицательный возраст — это лишь начало. Представьте, что кто-то по ошибке заменил значение переменной экземпляра `@date` в объекте встречи `Appointment` телефонным номером. Или установил ставку налога `@sales_tax` для всех своих объектов счетов `Invoice` равной нулю. Да здесь возможен целый миллион разных ошибок!

Чтобы перекрыть доступ к данным объекта злоумышленнику (или просто некомпетентному пользователю), во многих объектно-ориентированных языках используется концепция **инкапсуляции**: запрета на прямое обращение или изменение переменных экземпляра из других частей программы.

Методы доступа

Чтобы обеспечить инкапсуляцию данных и защитить экземпляры от некорректных изменений, Ruby не позволяет обращаться к переменным экземпляра или изменять их за пределами класса. Вместо этого можно создать **методы доступа** (accessor methods), которые записывают значения в переменные экземпляра и читают их за вас. После того как работа с данными будет вестить исключительно через методы доступа, вы сможете легко расширить эти методы для *проверки* данных — с отклонением любых некорректных значений, переданных при вызове.

В Ruby существуют два вида методов доступа: *методы записи атрибутов* и *методы чтения атрибутов*. (*Атрибут* — другое название для блока данных, относящегося к объекту.) Методы записи атрибутов *присваивают* значение переменным экземпляра, а методы чтения атрибутов *получают* сохраненное ранее значение.

Перед вами простой класс с методами чтения и записи атрибута с именем `my_attribute`:

```
class MyClass

  def my_attribute=(new_value)
    @my_attribute = new_value
  end

  def my_attribute
    @my_attribute
  end

end
```

Метод записи атрибута.

Методы доступа.

Метод чтения атрибута.

Если создать новый экземпляр этого класса...
```
my_instance = MyClass.new
```

...мы сможем присвоить значение атрибута...
```
my_instance.my_attribute = "a value"
```

...и прочитать его.
```
puts my_instance.my_attribute
```

Методы доступа представляют собой обычные методы экземпляра; особое название «методы доступа» связано только с тем, что их основной целью является обращение к переменной экземпляра.

Для примера рассмотрим метод чтения атрибута: это совершенно обычный метод, который просто возвращает текущее значение `@my_attribute`.

```
def my_attribute
  @my_attribute
end
```

Никакого волшебства! Метод просто возвращает текущее значение.

Методы доступа (продолжение)

Как и методы *чтения* атрибута, метод *записи* атрибута представляет собой совершенно обычный метод экземпляра. Мы называем его «методом записи» только потому, что он предназначен для обновления переменной экземпляра.

```
class MyClass

  def my_attribute=(new_value)
    @my_attribute = new_value
  end

  ...

end
```

Метод записи атрибута.

Возможно, метод самый обычный, но его *вызовы* обрабатываются не совсем обычно.

Помните, ранее в этой главе мы упоминали о том, что имена методов Ruby могут завершаться знаком равенства (=)? Эта возможность существует в Ruby именно для методов записи атрибута.

```
def my_attribute=(new_value)
  ...
end
```

Часть имени метода!

Когда Ruby встречает в вашем коде конструкцию вида:

```
my_instance.my_attribute = "a value"
```

...он преобразует ее в вызов метода экземпляра `my_attribute=`. Значение справа от = передается методу как аргумент:

```
my_instance.my_attribute=("a value")
```

Вызов метода.

Аргумент метода.

Приведенный выше фрагмент является действительным кодом Ruby. Если хотите, попробуйте его выполнить самостоятельно:

```
class MyClass
  def my_attribute=(new_value)
    @my_attribute = new_value
  end
  def my_attribute
    @my_attribute
  end
end

my_instance = MyClass.new
my_instance.my_attribute = "assigned via method call"
puts my_instance.my_attribute
my_instance.my_attribute=("same here")
puts my_instance.my_attribute
```

Вызов метода «my_attribute=», замаскированный под присваивание.

Вызов «my_attribute=» действительно выглядит так!

```
assigned via method call
same here
```

Житейская мудрость

Альтернативный способ вызова методов записи атрибута приводится только для того, чтобы вы поняли, что происходит «за кулисами». В настоящих программах на языке Ruby следует использовать только синтаксис с присваиванием!

Использование методов доступа

Итак, вы готовы использовать новые знания в приложении Fuzzy Friends. Для начала дополним класс `Dog` методами для чтения переменных экземпляра `@name` и `@age`. Также переменные `@name` и `@age` будут использоваться в методе `report_age`. Проверка данных будет рассмотрена позднее.

```
class Dog

  def name=(new_value)
    @name = new_value
  end

  def name
    @name
  end

  def age=(new_value)
    @age = new_value
  end

  def age
    @age
  end

  def report_age
    puts "#{@name} is #{@age} years old."
  end

end
```

Записываем новое значение в @name.

Читаем значение из @name.

Записываем новое значение в @age.

Читаем значение из @age.

После определения методов доступа мы можем (косвенно) записывать и использовать переменные экземпляра `@name` и `@age` за пределами класса `Dog`!

```
fido = Dog.new
fido.name = "Fido"
fido.age = 2
rex = Dog.new
rex.name = "Rex"
rex.age = 3
fido.report_age
rex.report_age
```

Присвоить @name значение Fido.

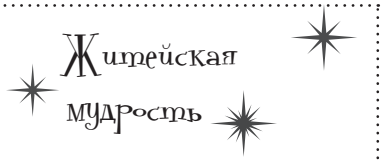
Задать переменную @age для объекта с именем Fido.

Присвоить @name значение Rex.

Задать @age для объекта с именем Rex.

```
Fido is 2 years old.
Rex is 3 years old.
```

Впрочем, вручную определять методы чтения и записи для каждого атрибута довольно утомительно. Сейчас мы рассмотрим более простой способ...



Имя метода чтения атрибута обычно совпадает с именем переменной экземпляра, из которой читается значение (конечно, без символа `@`).

```
def tail_length
  @tail_length
end
```

То же относится к именам методов записи атрибута, но в конец имени добавляется символ `=`.

```
def tail_length=(value)
  @tail_length = value
end
```

Методы чтения и записи атрибутов

Создание пары методов доступа для атрибута является задачей настолько распространенной, что в Ruby для нее были определены сокращения — методы с именами `attr_writer`, `attr_reader` и `attr_accessor`. Вызов этих трех методов в определении класса автоматически определяет эти методы доступа за вас:

Включите в определение класса...	*...и Ruby автоматически определит эти методы:*	

`attr_writer :name`	`def name=(new_value)`
	`  @name = new_value`
	`end`
		Совсем как наше старое определение!

`attr_reader :name`	`def name`
	`  @name`
	`end`
		Совсем как наше старое определение!

`attr_accessor :name`	`def name=(new_value)`
	`  @name = new_value`
	`end`
	
	`def name`
	`  @name`
	`end`
		Определяет два метода сразу!

Все три метода могут получать несколько аргументов с атрибутами, для которых должны определяться методы доступа.

Символические имена

```
attr_accessor :name, :age
```

Определяет ЧЕТЫРЕ метода сразу!

Если вас заинтересовали `:name` и `:age`, объясняем: это символические имена. **Символическое имя** Ruby представляет собой последовательность символов (как и строки). С другой стороны, в отличие от строки значение символического имени не может быть изменено позднее. По этой причине символические имена идеально подходят для ссылок на любые сущности, имена которых (обычно) не изменяются, — скажем, методы. Например, если вы вызовете в irb метод с именем `methods` для объекта, то увидите, что он возвращает список символических имен.

```
:hello
```

Символические имена Ruby

```
:over_easy
```

```
:east
```

```
> Object.new.methods
=> [:class, :singleton_class, :clone, ...]
```

Ссылки на символические имена в коде Ruby всегда начинаются с двоеточия (`:`). Символические имена должны записываться в нижнем регистре с разделением слов подчеркиваниями, как и в именах переменных.

Методы чтения и записи атрибутов в действии

В классе `Dog` в настоящее время методы доступа занимают 12 строк кода. С методом `attr_accessor` можно обойтись всего... одной строкой!

В результате размер класса `Dog` сокращается...

с такого...

```
class Dog

  def name=(new_value)
    @name = new_value
  end

  def name
    @name
  end

  def age=(new_value)
    @age = new_value
  end

  def age
    @age
  end

  def report_age
    puts "#{@name} is #{@age} years old."
  end

  def talk
    puts "#{@name} says Bark!"
  end

  def move(destination)
    puts "#{@name} runs to the #{destination}."
  end

end
```

Эквивалентно!

Эквивалентно!

...до такого!

```
class Dog

  attr_accessor :name, :age

  def report_age
    puts "#{@name} is #{@age} years old."
  end

  def talk
    puts "#{@name} says Bark!"
  end

  def move(destination)
    puts "#{@name} runs to the #{destination}."
  end

end
```

Что *теперь* скажете об эффективности? Не говоря уже о том, что этот код гораздо проще читается!

Но не будем забывать, для чего вообще создавались методы доступа. Мы хотели *защитить* переменные экземпляров от некорректных данных. В настоящее время методы никакой защиты не обеспечивают... Но через несколько страниц проблема будет решена!

Упражнение

До настоящего момента у нас еще не было возможности нормально поэкспериментировать с классами и объектами. Откройте новый сеанс irb. Мы загрузим простой класс и попробуем создать его экземпляры в интерактивном режиме.

Шаг 1:

Сохраните это определение класса в файле с именем *mage.rb*.

```
class Mage

  attr_accessor :name, :spell

  def enchant(target)
    puts "#{@name} casts #{@spell} on #{target.name}!"
  end

end
```

mage.rb

Шаг 2:

В терминальном окне перейдите в каталог, в котором был сохранен файл.

Шаг 3:

В этом упражнении, как и в предыдущем, Ruby будет загружать файлы из текущего каталога, поэтому для запуска irb должна использоваться команда

```
irb -I .
```

Шаг 4:

Как и прежде, необходимо загрузить файл с сохраненным кодом Ruby. Введите следующую команду:

```
require "mage"
```

Упражнение (продолжение)

После загрузки кода класса `Mage` вы можете свободно экспериментировать — создайте столько экземпляров, сколько вам захочется, задайте их атрибуты и вызывайте методы! Попробуйте для начала выполнить следующую цепочку команд:

```
merlin = Mage.new
merlin.name = "Merlin"
morgana = Mage.new
morgana.name = "Morgana"
morgana.spell = "Shrink"
morgana.enchant(merlin)
```

Пример сеанса:

```
File  Edit  Window  Help
$ irb -I .
irb(main):001:0> require 'mage'
 => true
irb(main):002:0> merlin = Mage.new
 => #<Mage:0x007fd432082308>
irb(main):003:0> merlin.name = "Merlin"
 => "Merlin"
irb(main):004:0> morgana = Mage.new
 => #<Mage:0x007fd43206b310>
irb(main):005:0> morgana.name = "Morgana"
 => "Morgana"
irb(main):006:0> morgana.spell = "Shrink"
 => "Shrink"
irb(main):007:0> morgana.enchant(merlin)
Morgana casts Shrink on Merlin!
 => nil
irb(main):008:0>
```

Кто я?

Компания концепций Ruby, облаченных в маскарадные костюмы, развлекается игрой «Кто я?». Игрок дает подсказку, а остальные на основании сказанного им пытаются угадать, кого он изображает. Будем считать, что игроки всегда говорят правду о себе. Заполните пропуски справа именами участников. (Мы уже заполнили одно поле за вас.)

Сегодняшние участники: среди них могут быть любые термины, относящиеся к хранению данных в объекте!

	Имя
Я существую в экземпляре объекта и храню данные, связанные с этим объектом.	переменная экземпляра
Я — другое название элемента данных, связанного с объектом. И еще я живу в переменной экземпляра.	
Я храню данные внутри метода. Как только метод завершается, я перестаю существовать.	
Я — разновидность метода экземпляра. Смысл моего существования — чтение или запись переменной экземпляра.	
Меня используют в программах для ссылок на сущности, имена которых не изменяются (например, методы).	

Кто я? Решение

Имя

Я существую в экземпляре объекта и храню данные, связанные с этим объектом. *переменная экземпляра*

Я — другое название элемента данных, связанного с объектом. И еще я живу в переменной экземпляра. *атрибут*

Я храню данные внутри метода. Как только метод завершается, я перестаю существовать. *локальная переменная*

Я — разновидность метода экземпляра. Смысл моего существования — чтение или запись переменной экземпляра. *метод доступа*

Меня используют в программах для ссылок на сущности, имена которых не изменяются (например, методы). *символическое имя*

Часто Задаваемые Вопросы

В: Чем методы доступа отличаются от методов экземпляра?

О: «Метод доступа» — всего лишь термин, обозначающий одну конкретную *разновидность* методов экземпляра. Методы этой группы предназначены для чтения или записи значений переменных экземпляра. Во всех остальных отношениях методы доступа ничем не отличаются от «обычных» методов экзмпляра.

В: Я создал переменную экземпляра за пределами метода экземпляра, но когда я пытаюсь обратиться к ней, у меня ничего не получается. Почему?

```
class Widget
  @size = 'large'
  def show_size
    puts "Size: #{@size}"
  end
end

widget = Widget.new
widget.show_size
```

Пусто!

```
Size:
```

О: Когда вы используете переменную экземпляра вне метода экземпляра, вы в действительности создаете переменную экземпляра *для объекта класса*. (Да, все верно — даже сами классы в Ruby являются объектами.)

И хотя такие переменные тоже могут принести пользу, мы не сможем рассмотреть их в этой книге. На данный момент результат почти наверняка будет не тем, на который вы рассчитывали. Вместо этого создайте переменную экземпляра в методе экземпляра:

```
class Widget
  def set_size
    @size = 'large'
  end
  ...
end
```

У бассейна

Выловите из бассейна фрагменты кода и расставьте их в пустых местах в коде. Каждый фрагмент может использоваться **только один** раз, причем использовать все фрагменты не обязательно.

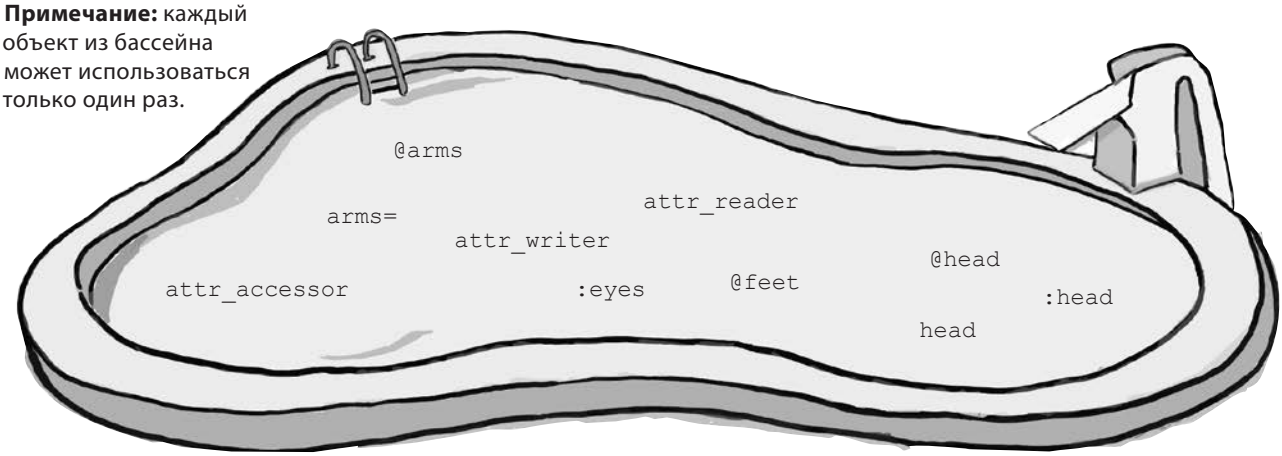
Ваша **задача** — составить код, который будет нормально выполняться и выдавать приведенный ниже результат.

```
class Robot

  def ____
    @head
  end

  def _____(value)
    @arms = value
  end

  __________ :legs, :body

  attr_writer _____

  _____________ :feet

  def assemble
    @legs = "RubyTek Walkers"
    @body = "BurlyBot Frame"
    _____ = "SuperAI 9000"
  end

  def diagnostic
    puts _____
    puts @eyes
  end

end

robot = Robot.new

robot.assemble

robot.arms = "MagGrip Claws"
robot.eyes = "X-Ray Scopes"
robot.feet = "MagGrip Boots"

puts robot.head
puts robot.legs
puts robot.body
puts robot.feet
robot.diagnostic
```

Результат

```
File Edit Window Help
SuperAI 9000
RubyTek Walkers
BurlyBot Frame
MagGrip Boots
MagGrip Claws
X-Ray Scopes
```

Примечание: каждый объект из бассейна может использоваться только один раз.



У бассейна. Решение

Выловите из бассейна фрагменты кода и расставьте их в пустых местах в коде. Каждый фрагмент может использоваться **только один** раз, причем использовать все фрагменты не обязательно. Ваша **задача** — составить код, который будет нормально выполняться и выдавать приведенный ниже результат.

```
class Robot

  def head
    @head
  end

  def arms=(value)
    @arms = value
  end

  attr_reader :legs, :body

  attr_writer :eyes

  attr_accessor :feet

  def assemble
    @legs = "RubyTek Walkers"
    @body = "BurlyBot Frame"
    @head = "SuperAI 9000"
  end

  def diagnostic
    puts @arms
    puts @eyes
  end

end
```

```
robot = Robot.new

robot.assemble

robot.arms = "MagGrip Claws"
robot.eyes = "X-Ray Scopes"
robot.feet = "MagGrip Boots"

puts robot.head
puts robot.legs
puts robot.body
puts robot.feet
robot.diagnostic
```

Результат

```
File Edit Window Help Lasers
SuperAI 9000
RubyTek Walkers
BurlyBot Frame
MagGrip Boots
MagGrip Claws
X-Ray Scopes
```

Проверка данных при вызове методов доступа

Помните наш кошмарный сценарий, в котором Ruby позволял программам напрямую обращаться к переменным экземпляра и вашим экземплярам `Dog` были назначены *пустые* имена с *отрицательным* возрастом? Плохие новости: теперь, после добавления методов записи атрибутов в класс `Dog`, это стало *возможно*!

```
joey = Dog.new
joey.name = ""
joey.age = -1
joey.report_age
```

```
 is -1 years old.
```

Без паники! Те же самые методы доступа помогут избежать подобных неприятностей в будущем. Мы добавим в методы простой механизм *проверки* данных, который будет выдавать ошибку при любой попытке передачи некорректного значения.

Так как `name=` и `age=` — самые обычные методы Ruby, реализовать проверку будет несложно; мы просто используем команды `if` для выявления пустых строк (для `name=`) или отрицательных чисел (для `age=`). При обнаружении недопустимого значения выводится сообщение об ошибке. И только если значение успешно прошло проверку, программа действительно присваивает его переменным экземпляра `@name` и `@age`.

```
class Dog

  attr_reader :name, :age

  def name=(value)
    if value == ""
      puts "Name can't be blank!"
    else
      @name = value
    end
  end

  def age=(value)
    if value < 0
      puts "An age of #{value} isn't valid!"
    else
      @age = value
    end
  end

  def report_age
    puts "#{@name} is #{@age} years old."
  end

end
```

Автоматически определяются только методы чтения, потому что методы записи мы будем определять самостоятельно.

Если имя пустое, вывести сообщение об ошибке.

Значение переменной экземпляра присваивается только в том случае, если имя прошло проверку.

Если возраст отрицателен, вывести сообщение об ошибке.

Значение переменной экземпляра присваивается только в том случае, если возраст прошел проверку.

Ошибки и «аварийная остановка»

Значит, при присваивании недопустимого имени или возраста выводится предупреждение. Прекрасно. Но ведь потом программа все равно вызывает `report_age`, и переменные name и age оказываются пустыми!

```
glitch = Dog.new
glitch.name = ""
glitch.age = -256
glitch.report_age
```

```
Name can't be blank!
An age of -256 isn't valid!
 is  years old.
```

Пусто!

Просто *вывести* сообщение недостаточно. Необходимо сделать с недействительными параметрами методов доступа `name=` и `age=` что-то более осмысленное. Изменим код проверки в методах `name=` и `age=` так, чтобы вызов встроенного метода Ruby `raise` сообщал пользователю о возникшей ошибке.

```
raise "Something bad happened!"
```

Вызов `raise` привлекает внимание пользователя к проблеме.

Методу `raise` может передаваться строка с описанием проблемы. Когда в ходе выполнения встречается этот вызов, Ruby прекращает заниматься текущим делом и выводит сообщение об ошибке. Так как наша программа не пытается как-то обработать ошибку, она просто немедленно завершается.

Использование «raise» в методах записи атрибутов

Перед вами обновленный код класса `Dog`...

Если мы используем `raise` в методах записи, включать секцию `else` в команды `if` не обязательно. Если новое значение недействительно, то после вызова `raise` выполнение программы прерывается. Программа просто не дойдет до команды, присваивающей значение переменной экземпляра.

```
class Dog

  attr_reader :name, :age

  def name=(value)
    if value == ""
      raise "Name can't be blank!"
    end
    @name = value
  end

  def age=(value)
    if value < 0
      raise "An age of #{value} isn't valid!"
    end
    @age = value
  end

  def report_age
    puts "#{@name} is #{@age} years old."
  end

end
```

Если значение «value» недействительно...

...выполнение прерывается в этой точке.

Эта команда не будет выполнена, если был вызван метод «raise».

Если значение «value» недействительно...

...выполнение прерывается в этой точке.

Эта команда не будет выполнена, если был вызван метод «raise».

Если теперь передать методу `name=` пустое имя, Ruby выдаст сообщение об ошибке и выполнение программы прерывается.

```
anonymous = Dog.new
anonymous.name = ""
```

Ошибка →

```
in `name=': Name can't be blank! (RuntimeError)
```

При попытке присваивания отрицательного возраста будет выведено другое сообщение об ошибке.

```
joey = Dog.new
joey.age = -1
```

Ошибка →

```
in `age=': An age of -1 isn't valid! (RuntimeError)
```

Как будет показано в главе 12, ошибки также могут обрабатываться в других частях вашей программы, чтобы она могла продолжить работу. А пока несознательный разработчик, который попытается передать экземпляру вашего класса `Dog` пустое имя или отрицательный возраст, немедленно узнает о том, что он должен переделать свой код.

Потрясающе! Если в коде разработчика допущена ошибка, он узнает об этом еще до того, как эту ошибку увидит пользователь. Отличная работа!

Код класса Dog

Ниже приведен полный код класса `Dog`, а также дополнительный код создания экземпляра `Dog`.

```
class Dog

  attr_reader :name, :age

  def name=(value)
    if value == ""
      raise "Name can't be blank!"
    end
    @name = value
  end

  def age=(value)
    if value < 0
      raise "An age of #{value} isn't valid!"
    end
    @age = value
  end

  def move(destination)
    puts "#{@name} runs to the #{destination}."
  end

  def talk
    puts "#{@name} says Bark!"
  end

  def report_age
    puts "#{@name} is #{@age} years old."
  end

end

dog = Dog.new
dog.name = "Daisy"
dog.age = 3
dog.report_age
dog.talk
dog.move("bed")
```

Определение методов чтения атрибутов «name» и «age».

Метод записи атрибута для «@name».

Проверка данных.

Метод записи атрибута для «@age».

Проверка данных.

Метод экземпляра.

Использование переменной экземпляра.

Метод экземпляра.

Использование переменной экземпляра.

Метод экземпляра.

Использование переменных экземпляра.

Создание нового экземпляра Dog.

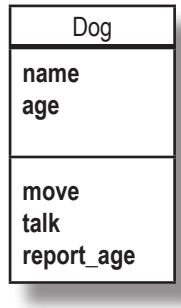
Инициализация атрибутов.

Вызов методов экземпляра.

dog.rb

Задание!

Сохраните приведенный выше код в файле с именем *dog.rb*. Попробуйте создать новые экземпляры `Dog`, затем выполните команду **`ruby dog.rb`** в терминале.



Мы определили методы экземпляра, которые работают как *методы доступа к атрибутам* и используются для чтения и записи значений переменных экземпляра.

```
puts dog.name
dog.age = 3
puts dog.age
```

```
Daisy
3
```

Методы экземпляра позволяют объекту Dog выполнять разные действия: перемещаться, издавать звуки и сообщать свой возраст. Методы экземпляра могут использовать данные, хранящиеся в переменных экземпляра.

```
dog.report_age
dog.talk
dog.move("bed")
```

```
Daisy is 3 years old.
Daisy says Bark!
Daisy runs to the bed.
```

Также мы изменили методы записи атрибутов, чтобы они *проверяли* передаваемые данные и выдавали сообщение об ошибке в случае их недействительности.

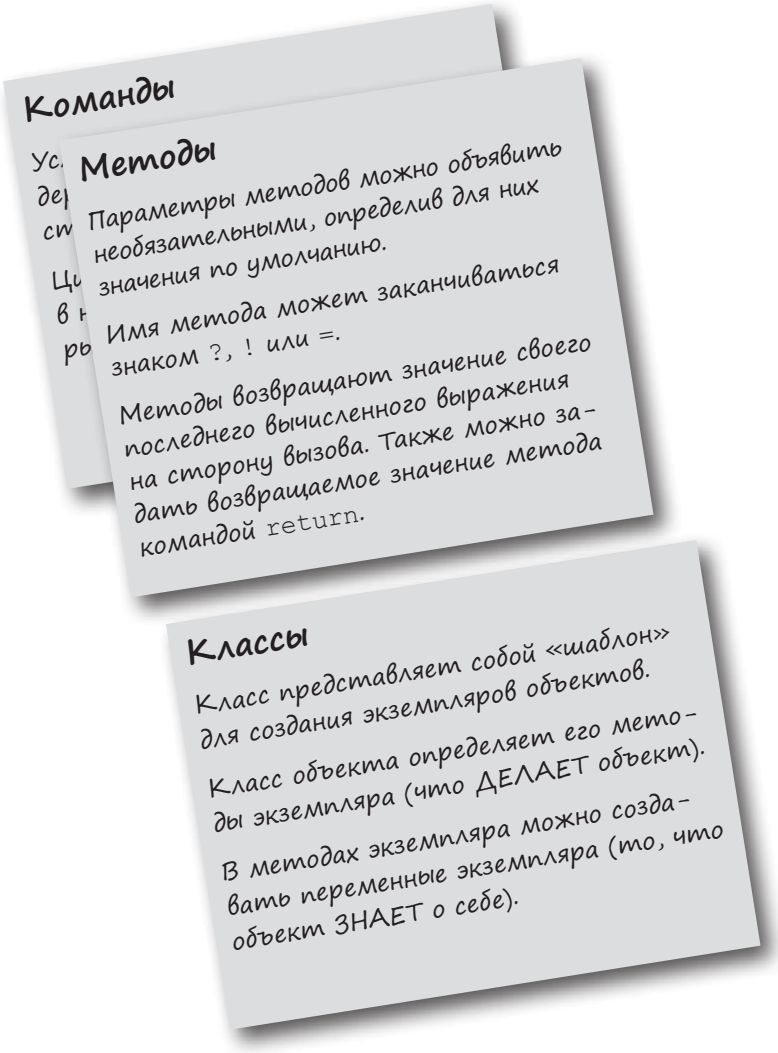
```
dog.name = ""
```

Ошибка →

```
in `name=': Name
can't be blank!
(RuntimeError)
```

Ваш инструментарий Ruby

Глава 2 подошла к концу. В ней ваш инструментарий Ruby пополнился методами и классами.



КЛЮЧЕВЫЕ МОМЕНТЫ

- Тело метода состоит из одной или нескольких команд Ruby, выполняемых при вызове метода.
- Круглые скобки не указываются в определении метода в том (и только в том) случае, если в методе не определяется ни один параметр.
- Если возвращаемое значение не указано явно, то метод возвращает значение последнего вычисленного выражения.
- Определения методов в определении класса рассматриваются как методы экземпляра этого класса.
- За пределами определения класса к переменным экземпляра можно обращаться только через методы доступа.
- Вызовы методов `attr_writer`, `attr_reader` и `attr_accessor` в определении класса обеспечивают сокращенную запись для определения методов доступа.
- Методы доступа могут использоваться для проверки действительности данных перед их сохранением в переменных экземпляра.
- Метод `raise` предназначен для вывода сообщений об ошибках в программе.

Далее в программе...

Мы создали полноценный класс `Dog`. Остается лишь добавить те же возможности в классы `Cat` и `Bird`!

Перспектива копирования кода вас не радует? Не беспокойтесь! В следующей главе мы займемся наследованием, и этот механизм упростит задачу!

С родительской помощью

Столько повторений! Новые классы, представляющие разные типы машин или животных, удобны — это правда. Но ведь вам придется *копировать методы экземпляра из класса в класс*. И эти копии будут вести самостоятельную жизнь — одни будут работать нормально, в других могут появиться ошибки. Разве классы создавались не для того, чтобы *упростить* сопровождение кода?

В этой главе вы научитесь применять **наследование**, чтобы ваши классы могли использовать *одни и те же* методы. Меньше дубликатов — меньше проблем с сопровождением!

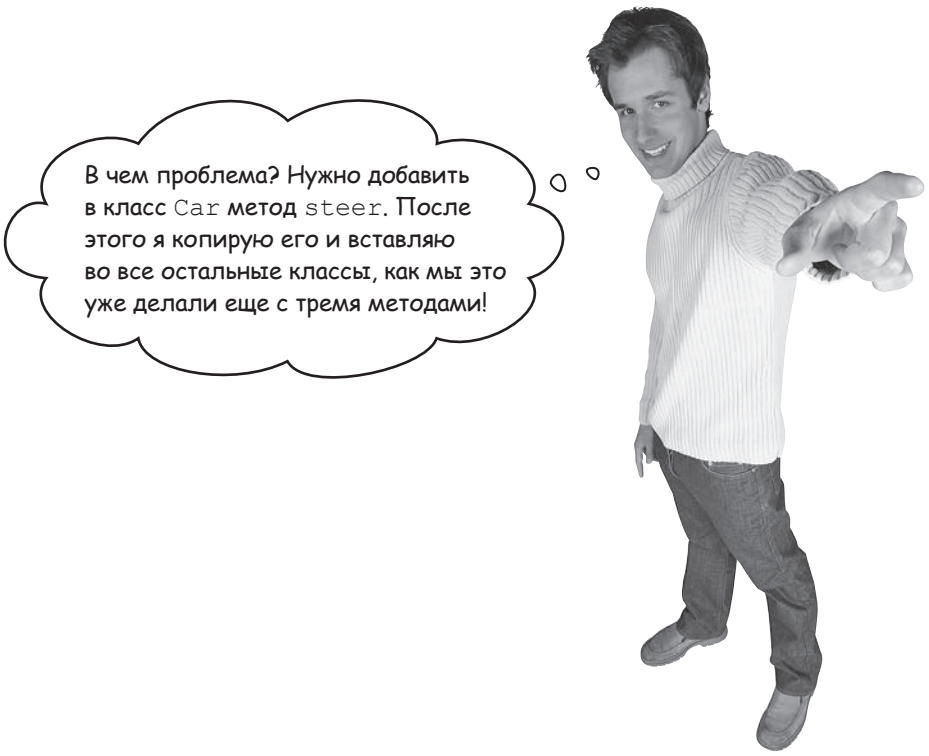
Копировать, вставлять... Столько проблем!

Разработчики из Got-A-Motor, Inc. решили внедрить принципы объектно-ориентированного программирования в своей работе. Старое приложение виртуального тест-драйва было переработано так, чтобы каждый тип машины был представлен отдельным классом. У них есть классы, представляющие легковые машины (Car), грузовики (Truck) и мотоциклы (Motorcycle).

На данный момент структура классов выглядит так:

	Car	Truck	Motorcycle
переменные экземпляра	odometer gas_used	odometer gas_used	odometer gas_used
методы экземпляра	mileage accelerate sound_horn	mileage accelerate sound_horn	mileage accelerate sound_horn

Прислушавшись к пожеланиям клиентов, руководство захотело включить во все типы машин метод для управления рулем. Майк, неопытный разработчик из Got-A-Motor, считает, что с этим требованием справиться несложно.



Код Майка для приложения виртуального тест-драйва

```
class Car

  attr_accessor :odometer
  attr_accessor :gas_used

  def mileage
    @odometer / @gas_used
  end

  def accelerate
    puts "Floor it!"
  end

  def sound_horn
    puts "Beep! Beep!"
  end

  def steer                          Копируем!
    puts "Turn front 2 wheels."
  end

end
```

```
class Truck

  attr_accessor :odometer
  attr_accessor :gas_used

  def mileage
    @odometer / @gas_used
  end

  def accelerate
    puts "Floor it!"
  end

  def sound_horn
    puts "Beep! Beep!"
  end

  def steer                          Вставляем!
    puts "Turn front 2 wheels."
  end

end
```

```
class Motorcycle

  attr_accessor :odometer
  attr_accessor :gas_used

  def mileage
    @odometer / @gas_used
  end

  def accelerate
    puts "Floor it!"
  end

  def sound_horn
    puts "Beep! Beep!"
  end

  def steer                          Вставляем!
    puts "Turn front 2 wheels."
  end

end
```

Но у Марси, опытного объектно-ориентированного разработчика из этой группы, такая идея вызывает сомнения.

Марси права; сопровождение кода вскоре превратится в сущий кошмар. Для начала разберемся, как решить проблему с дублированием, а затем исправим метод `steer` для объектов `Motorcycle`.

На помощь приходит наследование!

К счастью, в Ruby, как и в большинстве объектно-ориентированных языков, поддерживается концепция **наследования**, позволяющая классам наследовать методы друг от друга. Если один класс поддерживает некоторую функциональность, другие классы, наследующие от него, наделяются этой функциональностью *автоматически*.

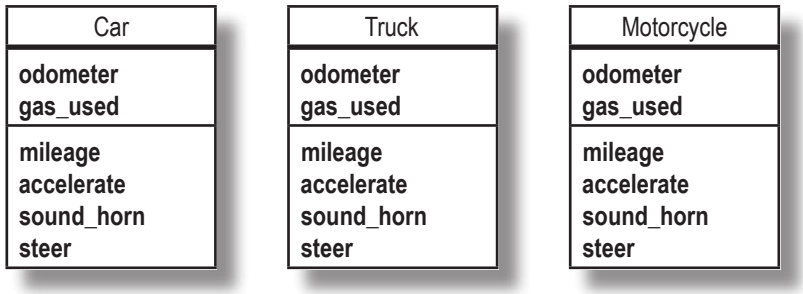
Вместо того чтобы повторять определения метода во многих похожих классах, вы выделяете общие методы в один класс, а затем указываете, что другие классы наследуют от него. Класс, содержащий общие методы, называется **суперклассом**, а классы, наследующие методы, называются **субклассами**.

Если суперкласс содержит методы экземпляра, то эти методы автоматически наследуются его субклассами. Вы можете получить доступ ко всем нужным методам суперкласса без дублирования их кода в каждом субклассе.

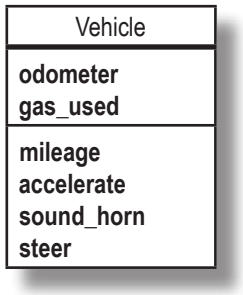
А теперь посмотрим, как наследование поможет исключить дублирование кода в приложении виртуального тест-драйва...

Наследование позволяет нескольким субклассам наследовать методы от одного суперкласса.

1. Мы видим, что классы `Car`, `Truck` и `Motorcycle` содержат несколько общих методов экземпляра и атрибутов.

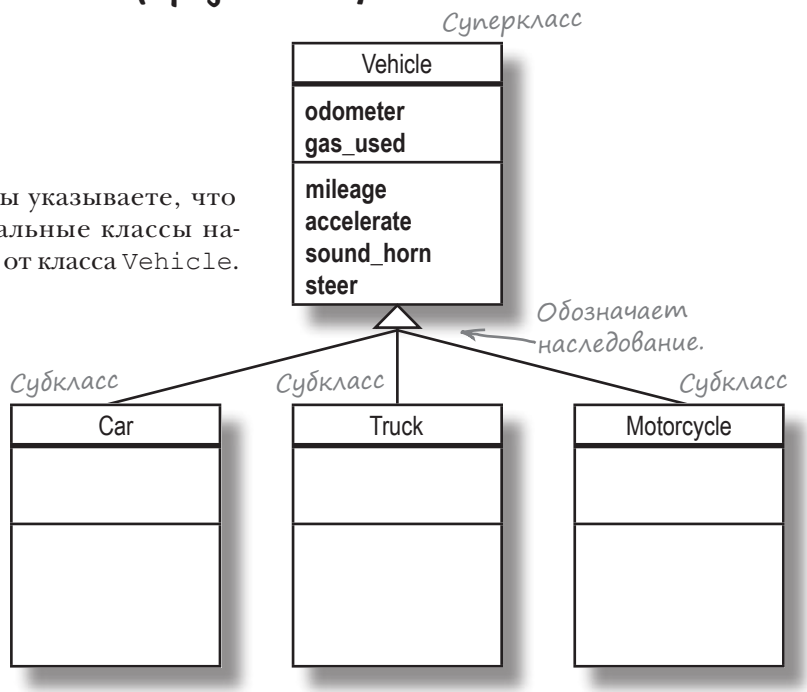


2. Каждый из этих классов является разновидностью машины. Мы можем создать новый класс (допустим, он будет называться `Vehicle`) и переместить в него общие методы и атрибуты.



На помощь приходит наследование! (продолжение)

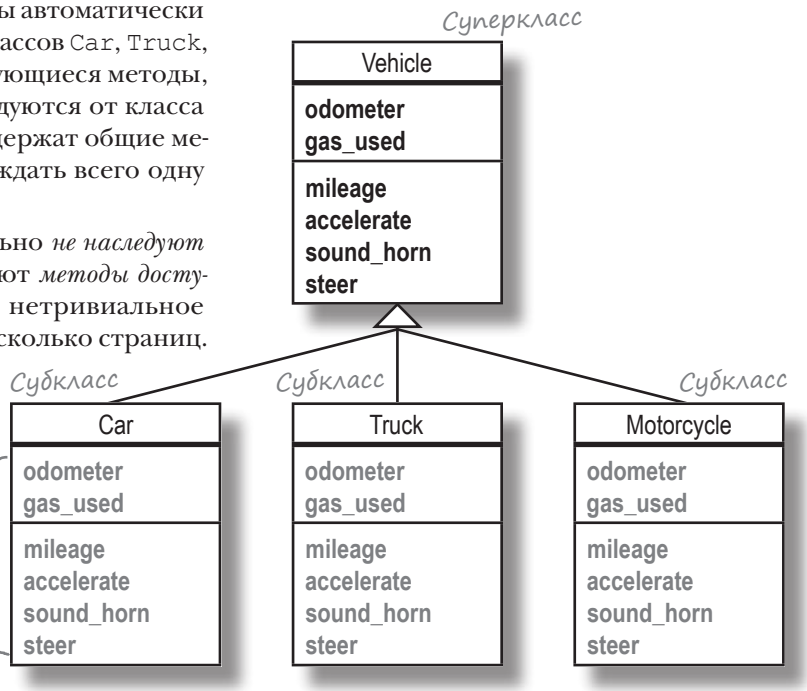
❸ Затем вы указываете, что все остальные классы наследуют от класса `Vehicle`.



Класс `Vehicle` называется *суперклассом* для трех других классов. `Car`, `Truck` и `Motorcycle` называются *субклассами* по отношению к `Vehicle`.

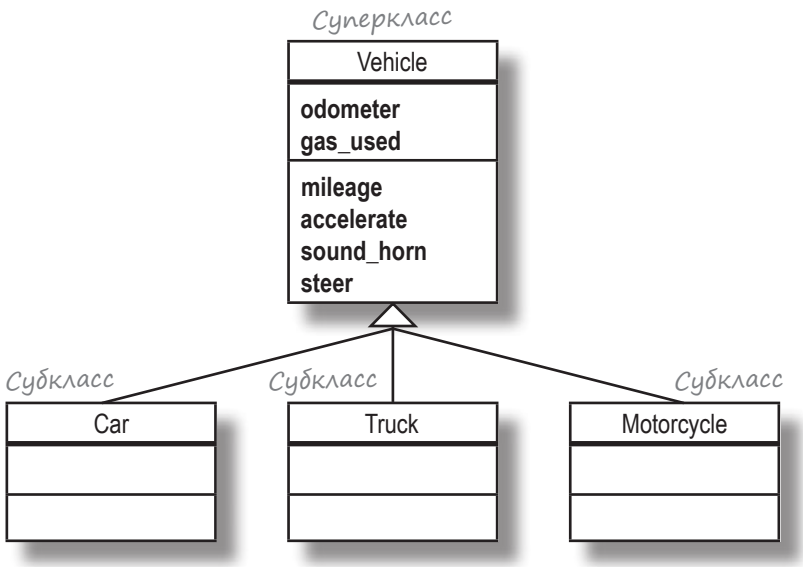
Субклассы *наследуют* все методы и атрибуты суперкласса. Иначе говоря, если суперкласс обладает некоторой функциональностью, то и его субклассы автоматически получают эту функциональность. Из классов `Car`, `Truck`, и `Motorcycle` можно удалить дублирующиеся методы, потому что они автоматически наследуются от класса `Vehicle`. Все классы по-прежнему содержат общие методы, но теперь достаточно сопровождать всего одну копию каждого метода!

Учтите, что в Ruby субклассы формально *не наследуют переменные* экземпляров; они наследуют *методы доступа*, создающие эти переменные. Это нетривиальное различие будет рассмотрено через несколько страниц.



Определение суперкласса (в общем-то ничего особенного!)

Чтобы исключить повторяющиеся методы и атрибуты в классах `Car`, `Truck` и `Motorcycle`, Марси разработала эту иерархию классов. Общие методы и атрибуты были вынесены в *суперкласс* `Vehicle`. `Car`, `Truck` и `Motorcycle` являются *субклассами* `Vehicle`, и они *наследуют* все методы `Vehicle`.



В Ruby не существует никакого специального синтаксиса определения суперклассов; это самый обычный класс. (Это относится к большинству объектно-ориентированных языков.)

```
class Vehicle
  attr_accessor :odometer
  attr_accessor :gas_used

  def accelerate
    puts "Floor it!"
  end

  def sound_horn
    puts "Beep! Beep!"
  end

  def steer
    puts "Turn front 2 wheels."
  end

  def mileage
    return @odometer / @gas_used
  end

end
```

Все атрибуты будут унаследованы при объявлении субкласса.

Как и все методы экземпляра.

Определение субкласса (совсем просто)

Синтаксис субклассов тоже не отличается сложностью. Определение субкласса выглядит как определение обычного класса, не считая того, что вы указываете суперкласс, от которого он наследует.

В Ruby используется знак «меньше» (<), потому что субкласс является *подмножеством* суперкласса. (Все грузовики — машины, но не все машины — грузовики.) Можно считать, что субкласс *меньше* суперкласса.

Итак, чтобы указать, что `Car`, `Truck` и `Motorcycle` являются субклассами `Vehicle`, достаточно использовать следующую запись:

```
class Car < Vehicle
end

class Truck < Vehicle
end

class Motorcycle < Vehicle
end
```

Как только `Car`, `Truck` и `Motorcycle` определяются как субклассы, они немедленно наследуют все атрибуты и методы экземпляра `Vehicle`. И хотя субклассы не содержат собственного кода, для всех созданных нами экземпляров будет доступна вся функциональность суперкласса!

```
truck = Truck.new
truck.accelerate
truck.steer

car = Car.new
car.odometer = 11432
car.gas_used = 366

puts "Lifetime MPG:"
puts car.mileage
```

```
Floor it!
Turn front 2 wheels.
Lifetime MPG:
31
```

Классы `Car`, `Truck` и `Motorcycle` обладают той же функциональностью, что и прежде, но без дублирования кода. Наследование избавило нас от серьезных хлопот с сопровождением кода!

Добавление методов в субклассы

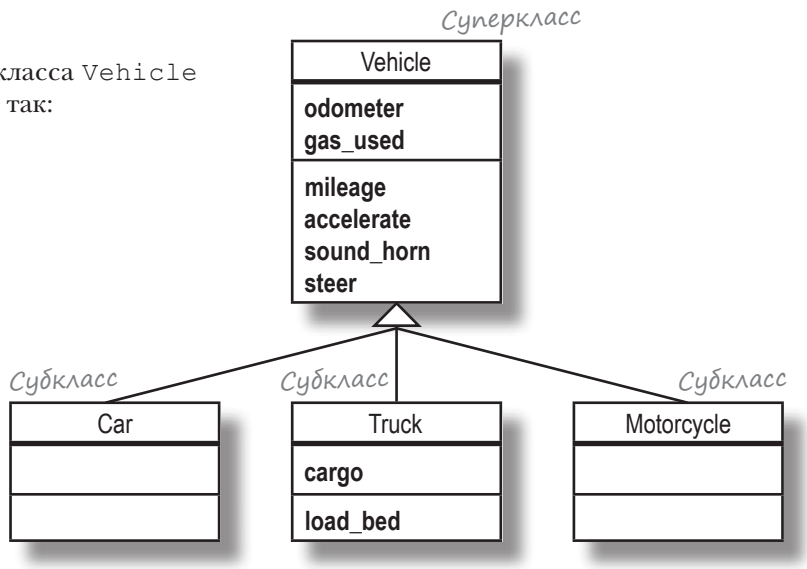
На данный момент классы `Truck` , `Car` или `Motorcycle` совершенно не отличаются друг от друга. Но какой прок от грузовика, если он не может перевозить грузы? Компания Got-A-Motor хочет добавить в экземпляры `Truck` метод `load_bed`, а также атрибут `cargo` для хранения информации о грузе.

Однако добавлять `cargo` и `load_bed` в класс `Vehicle` было бы неверно. Конечно, класс `Truck` унаследует их, но они также будут унаследованы классами `Car` и `Motorcycle`. У легковых машин и мотоциклов *нет* грузовой платформы!

Вместо этого мы определим атрибут `cargo` и метод `load_bed` *прямо в классе* `Truck`.

```
class Truck < Vehicle

  attr_accessor :cargo

  def load_bed(contents)
    puts "Securing #{contents} in the truck bed."
    @cargo = contents
  end

end
```

Если сейчас нарисовать диаграмму класса `Vehicle` и его субклассов, она будет выглядеть так:



С такими изменениями мы можем создать новый экземпляр `Truck`, загрузить машину и получить доступ к данным груза.

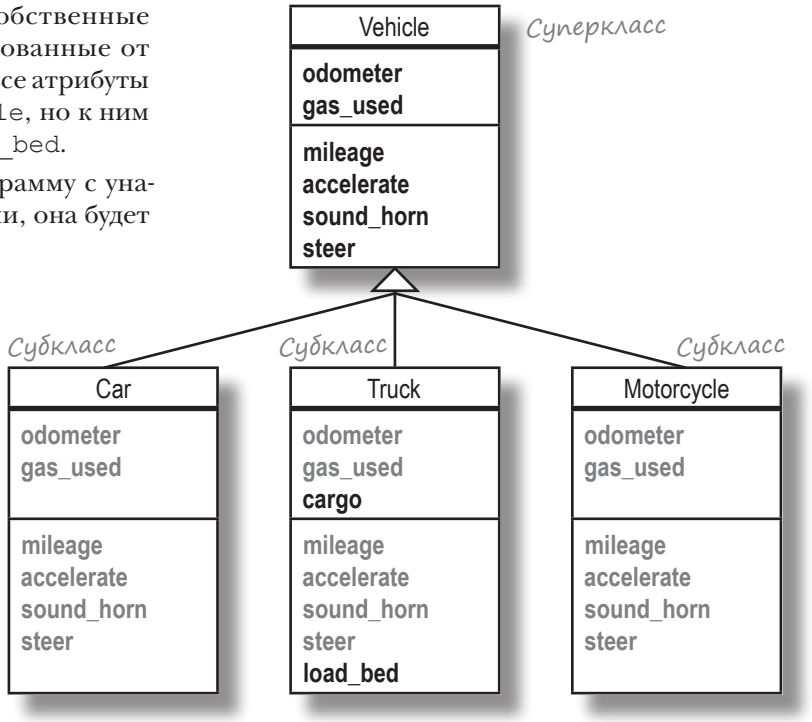
```
truck = Truck.new
truck.load_bed("259 bouncy balls")
puts "The truck is carrying #{truck.cargo}."
```

```
Securing 259 bouncy balls in the truck bed.
The truck is carrying 259 bouncy balls.
```

Субклассы содержат как новые, так и унаследованные методы

Однако субкласс, определяющий собственные методы, не теряет методы, унаследованные от суперкласса. Класс `Truck` сохраняет все атрибуты и методы, унаследованные от `Vehicle`, но к ним добавляются атрибуты `cargo` и `load_bed`.

Если теперь снова нарисовать диаграмму с унаследованными атрибутами и методами, она будет выглядеть так:



Итак, кроме атрибута `cargo` и метода `load_bed`, наш экземпляр `Truck` также может обращаться к старым унаследованным атрибутам и методам.

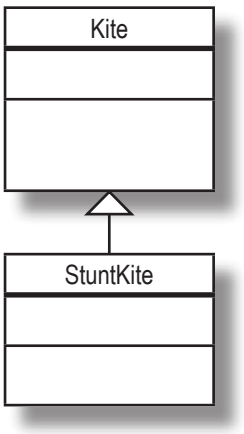
```
truck.odometer = 11432
truck.gas_used = 366
puts "Average MPG:"
puts truck.mileage
```

```
Average MPG:
31
```

Итак, субкласс наследует методы экземпляра от своего суперкласса. А как насчет наследования переменных экземпляра?

Как ни странно, нет! Потерпите минутку; чтобы прояснить этот вопрос, нам придется сделать «крюк» на пару страниц...

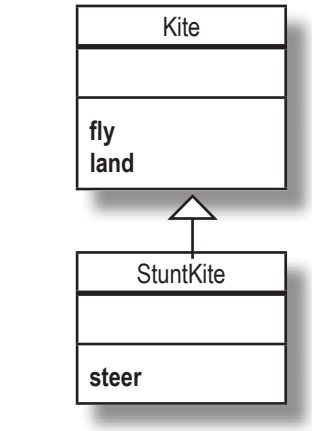
Возьми в руку карандаш



Имеются два класса, `Kite` и `StuntKite`. Экземпляры обоих классов, `Kite` и `StuntKite`, должны содержать методы `fly` и `land`. С другой стороны, метод `steer` должен присуствовать только в классе `StuntKite`. Разместите имена и определения классов в соответствующих местах диаграммы классов.

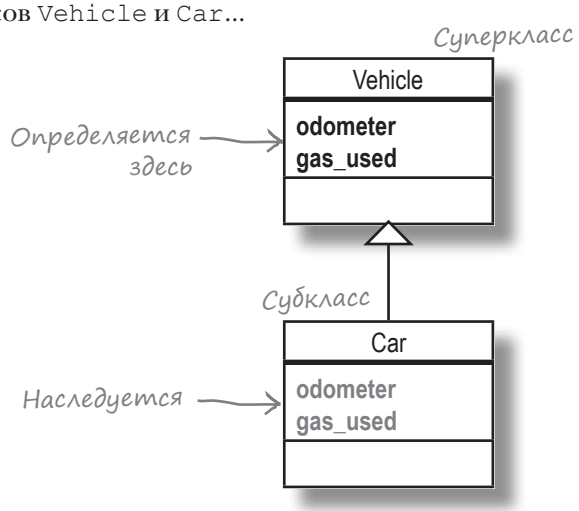
fly **steer** **land**

Возьми в руку карандаш
Решение



Переменные экземпляра принадлежат объекту, а не классу!

Возникает впечатление, что переменные экземпляров, определенные в суперклассе, наследуются субклассами, но в Ruby механизм наследования работает не так. Давайте еще раз взглянем на диаграмму классов, уделяя особое внимание методам доступа классов `Vehicle` и `Car`...

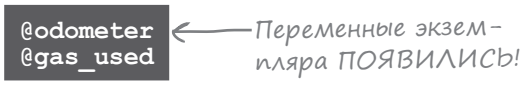


Вроде бы разумно предположить, что `Car` унаследует переменные экземпляра `@odometer` и `@gas_used` от `Vehicle`. Давайте проверим... У каждого объекта Ruby имеется метод `instance_variables`, который можно вызвать для просмотра списка переменных экземпляра, определенных для этого объекта. Давайте попробуем создать новый объект `Car` и посмотрим, какие переменные экземпляра он содержит.

```
car = Car.new
puts car.instance_variables
```

Вывод пуст, потому что `car` *не содержит* никаких переменных экземпляра! У объекта нет никаких переменных экземпляра, пока для него не будут вызваны методы экземпляра; в этот момент *метод* создает переменные в объекте. Давайте вызовем методы записи атрибута `odometer` и `gas_used`, а затем снова проверим список переменных экземпляра.

```
car.odometer = 22914
car.gas_used = 728
puts car.instance_variables
```



Итак, класс `Car` не наследует *переменные* экземпляра `@odometer` и `@gas_used`... Он наследует *методы* экземпляра `odometer=` и `gas_used=`, а *методы* создали переменные экземпляра!

Во многих других объектно-ориентированных языках переменные экземпляра объявляются на уровне *класса*, так что язык Ruby *отличается* от них в этом отношении. Различие достаточно тонкое, и все же о нем следует знать...

Переменные экземпляра принадлежат объекту, а не классу! (продолжение)

Почему важно знать, что переменные экземпляра принадлежат объекту, а не классу? Если вы соблюдаете правила и следите за тем, чтобы имена переменных экземпляра совпадали с именами методов доступа, беспокоиться об этом не обязательно. Но если вы отклоняетесь от этой схемы, берегитесь! Может оказаться, что субкласс нарушает функциональность суперкласса, *перезаписывая* его переменные экземпляра.

Допустим, имеется суперкласс, который нарушает правило и использует переменную экземпляра `@storage` для хранения значения методов доступа `name=` и `name`. Затем предположим, что субкласс использует то же имя переменной `@storage` для хранения значения, используемого методами доступа `salary=` и `salary`.

НЕПРАВИЛЬНЫЙ выбор имен переменных...

```
class Person
  def name=(new_value)
    @storage = new_value
  end
  def name
    @storage
  end
end

class Employee < Person
  def salary=(new_value)
    @storage = new_value
  end
  def salary
    @storage
  end
end
```

...то же имя используется здесь. (А почему бы и нет?)

Если вы попытаетесь работать с субклассом `Employee`, то обнаружите, что каждое присваивание атрибуту `salary` приводит к перезаписи атрибута `name`, потому что оба атрибута используют *одну* переменную экземпляра.

```
employee = Employee.new
employee.name = "John Smith"
employee.salary = 80000
puts employee.name
```

Это имя? Выглядит странно.

```
80000
```

Следите за тем, чтобы имена переменных в вашей программе соответствовали именам методов доступа к атрибутам. Соблюдайте это простое правило — оно позволит вам избежать многих проблем!

Переопределение методов

Марси, опытный специалист по объектно-ориентированной разработке, переписала наши классы `Car`, `Truck` и `Motorcycle` как субклассы `Vehicle`. Никакие собственные методы или атрибуты им не нужны — вся функциональность наследуется от суперкласса! Но Майк указывает на недостаток такой архитектуры...

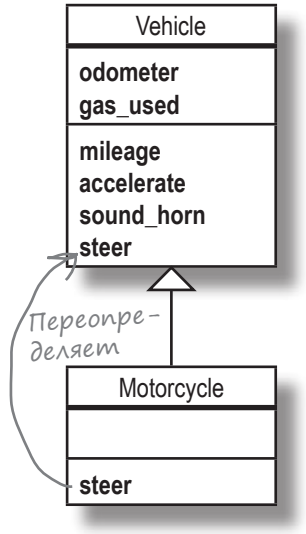

```
motorcycle = Motorcycle.new
motorcycle.steer
```

```
Turn front 2 wheels.
```

Многовато для мотоцикла!

Если поведение суперкласса не соответствует потребностям субкласса, наследование предоставляет еще один механизм: *переопределение* методов. При **переопределении (overriding)** одного или нескольких методов в субклассе происходит замена унаследованных методов суперкласса методами, специализированными для субкласса.

```
class Motorcycle < Vehicle
  def steer
    puts "Turn front wheel."
  end
end
```



Если теперь вызвать метод `steer` для экземпляра `Motorcycle`, будет вызван переопределяющий метод — иначе говоря, будет вызвана версия `steer`, определенная в классе `Motorcycle`, а не версия из класса `Vehicle`.

```
motorcycle.steer
```

```
Turn front wheel.
```

Переопределение методов (продолжение)

Однако при вызове любых других методов для экземпляра `Motorcycle` будет вызван унаследованный метод.

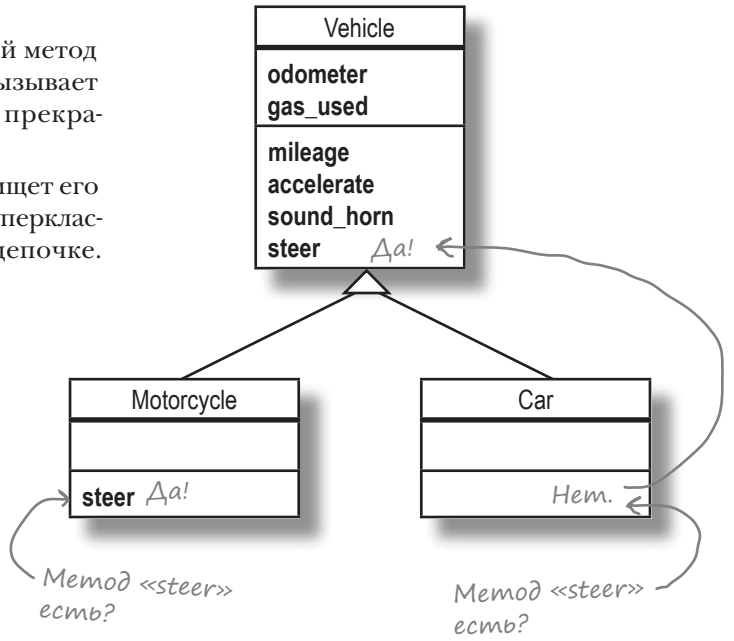
```
motorcycle.accelerate
```

```
Floor it!
```

Как работает этот механизм?

Если Ruby видит, что вызванный метод определен в субклассе, то он вызывает этот метод, и на этом поиски прекращаются.

Но если метод не найден, Ruby ищет его сначала в суперклассе, потом в суперклассе суперкласса и так далее по цепочке.



И снова все работает! Если потребуется внести изменения, их можно внести в классе `Vehicle`, и они автоматически распространяются по субклассам; это означает, что все классы быстрее воспользуются преимуществами изменений. Если субклассу потребуется специализированное поведение, то он просто переопределяет метод, унаследованный от суперкласса.

А вы неплохо справились с переработкой классов Got-A-Motor!

Пора снова обратиться к коду Fuzzy Friends. Классы этого приложения все еще содержат большое количество избыточных методов. Нельзя ли избавиться от них за счет применения наследования и переопределения методов?

Часто Задаваемые Вопросы

В: Может ли наследование проходить через несколько уровней? Иначе говоря, может ли субкласс иметь собственные субклассы?

О: Да! Если вам потребуется переопределить методы некоторых экземпляров вашего субкласса, оставив остальные без изменений, возможно, вам стоит создать субкласс для этого субкласса.

```
class Car < Vehicle
end
class DragRacer < Car
  def accelerate
    puts "Inject nitrous!"
  end
end
```

Только не увлекайтесь! Сложность многоуровневых иерархий быстро растет. Формально количество уровней наследования в Ruby может быть произвольным, но большинство разработчиков Ruby ограничивается одним или двумя уровнями.

В: Вы сказали, что если для экземпляра вызывается метод и Ruby этот метод не находит, то поиск продолжается в суперклассе, потом в суперклассе этого суперкласса... А если суперклассы кончатся, а метод так и не будет найден?

О: После поиска в последнем суперклассе Ruby отказывается от дальнейших попыток. Именно в этот момент выдаются сообщения об ошибках неопределенных методов, которые нам уже встречались.

```
Car.new.fly
```

```
undefined method
`fly' for
#<Car:0x007ffec48c>
```

В: С чего следует начинать проектирование иерархии наследования — с субклассов или суперклассов?

О: Возможны оба варианта! Не исключено, что сама мысль о применении наследования придет уже после того, как вы начнете писать код приложения.

Допустим, вы обнаружили, что два взаимосвязанных класса должны содержать похожие или одинаковые методы, и просто преобразовали их в субклассы нового суперкласса. Затем общие методы были выделены в суперкласс. Получается, что вы начали с проектирования субклассов.

Аналогичным образом, когда вы обнаруживаете, что метод используется не всеми экземплярами класса, создайте новый субкласс для существующего класса и переместите метод в него. В этом случае проектирование началось с суперкласса!

Развлечения с магнитами

В программе на языке Ruby, выложенной на холодильнике, часть магнитов перепуталась. Сможете ли вы расставить фрагменты кода по местам и получить полноценный суперкласс и субкласс, с которыми для приведенного примера выводился бы заданный результат?

```
class   Camera   DigitalCamera   <   Camera

class   def   def   def   load

end   end   end   end   load

end   take_picture

puts "Triggering shutter."

puts "Inserting memory card."

puts "Winding film."
```

Пример кода:

```
camera = Camera.new
camera.load
camera.take_picture

camera2 = DigitalCamera.new
camera2.load
camera2.take_picture
```

Вывод:

```
File Edit Window Help
Winding film.
Triggering shutter.
Inserting memory card.
Triggering shutter.
```

Кто и что делает?

Соедините каждую из концепций в левом столбце с определением справа.

Субкласс	Заменяет метод, унаследованный от суперкласса, новой функциональностью.
Переопределение	Обеспечивает возможность использования одного метода или атрибута в нескольких классах.
Наследование	Класс с кодом методов, совместно используемых одним или несколькими классами.
Суперкласс	Класс, наследующий методы и/или атрибуты от суперкласса.

Развлечения с магнитами. Решение

В программе на языке Ruby, выложенной на холодильнике, часть магнитов перепуталась. Сможете ли вы расставить фрагменты кода по местам и получить полноценный суперкласс и субкласс, с которыми для приведенного примера выводился бы заданный результат?

```
class Camera
  def take_picture
    puts "Triggering shutter."
  end
  def load
    puts "Winding film."
  end
end
```

```
class DigitalCamera < Camera
  def load
    puts "Inserting memory card."
  end
end
```

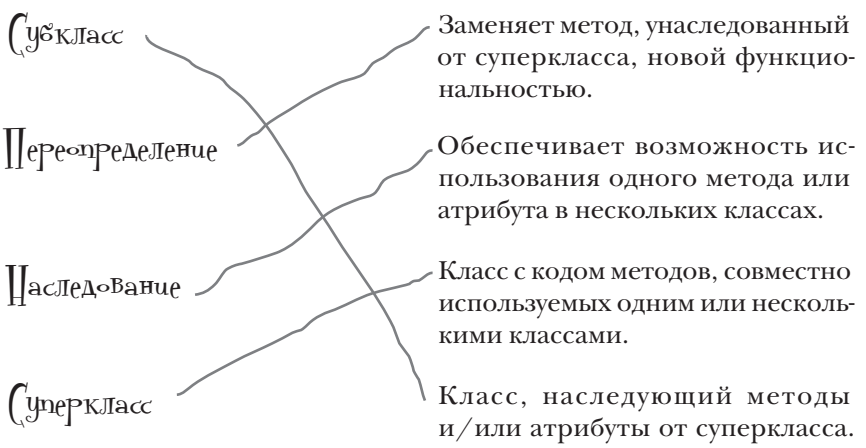
Пример кода:

```
camera = Camera.new
camera.load
camera.take_picture

camera2 = DigitalCamera.new
camera2.load
camera2.take_picture
```

Вывод:

```
File Edit Window Help
Winding film.
Triggering shutter.
Inserting memory card.
Triggering shutter.
```

Соедините каждую из концепций в левом столбце с определением справа.



Включение наследования в иерархию классов животных

Помните приложение Fuzzy Friends из предыдущей главы? Мы тогда неплохо потрудились над классом `Dog`: добавили методы доступа к атрибутам `name` и `age` (с проверкой данных), а также обновили методы `talk`, `move` и `report_age`, чтобы в них использовались переменные экземпляра `@name` и `@age`.

Кратко напомним, что же у нас в итоге получилось:

Создает методы для получения текущих значений @name и @age.

Мы создаем собственные методы записи атрибутов, чтобы проверить новые значения на действительность.

Другие методы экземпляра для объектов Dog.

```
class Dog

  attr_reader :name, :age

  def name=(value)
    if value == ""
      raise "Name can't be blank!"
    end
    @name = value
  end

  def age=(value)
    if value < 0
      raise "An age of #{value} isn't valid!"
    end
    @age = value
  end

  def talk
    puts "#{@name} says Bark!"
  end

  def move(destination)
    puts "#{@name} runs to the #{destination}."
  end

  def report_age
    puts "#{@name} is #{@age} years old."
  end

end
```

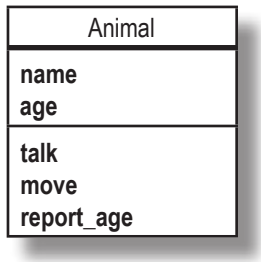
Однако классы `Bird` и `Cat` были полностью забыты, хотя они обладают почти идентичной функциональностью.

Применим новую концепцию наследования для создания иерархии, которая обновляет все классы сразу (и обеспечивает их быстрое обновление в будущем).

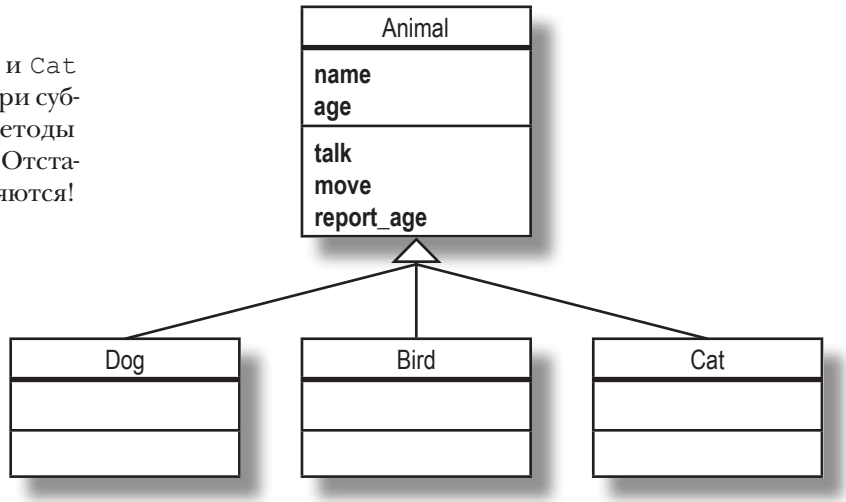
Проектирование иерархий классов животных

Мы добавили много новой функциональности в класс `Dog`, а теперь эта функциональность также должна появиться в классах `Cat` и `Bird`...

Все классы должны обладать атрибутами `name` и `age`, а также методами `talk`, `move` и `report_age`. Давайте переместим все эти атрибуты и методы в новый класс, который мы назовем `Animal`.



Затем мы объявим, что `Dog`, `Bird` и `Cat` являются *субклассами* `Animal`. Все три субкласса наследуют все атрибуты и методы экземпляра от своего суперкласса. «Отстающие» классы моментально обновляются!



Код класса Animal и его субклассов

Перед вами код суперкласса `Animal`, в который были перенесены все старые методы из `Dog`...

```
class Animal

  attr_reader :name, :age

  def name=(value)
    if value == ""
      raise "Name can't be blank!"
    end
    @name = value
  end

  def age=(value)
    if value < 0
      raise "An age of #{value} isn't valid!"
    end
    @age = value
  end

  def talk
    puts "#{@name} says Bark!"
  end

  def move(destination)
    puts "#{@name} runs to the #{destination}."
  end

  def report_age
    puts "#{@name} is #{@age} years old."
  end

end
```

Точно такой же код содержался в классе Dog!

А вот и остальные классы, переписанные в виде субклассов `Animal`.

```
class Dog < Animal
end

class Bird < Animal
end

class Cat < Animal
end
```

Записывать здесь методы не нужно; эти классы наследуют все методы от приведенного выше класса Animal!

Переопределение метода в субклассах Animal

Классам `Dog`, `Bird` и `Cat`, переписанным в виде субклассов `Animal`, не нужны собственные методы или атрибуты — они наследуют все необходимое от суперкласса.

```
whiskers = Cat.new
whiskers.name = "Whiskers"
fido = Dog.new
fido.name = "Fido"
polly = Bird.new
polly.name = "Polly"

polly.age = 2
polly.report_age
fido.move("yard")
whiskers.talk
```

```
Polly is 2 years old.
Fido runs to the yard.
Whiskers says Bark!
```

Погодите... Но ведь Whiskers — объект Cat...

Выглядит неплохо, если не считать одного... Наша кошка (объект `Cat`) лает!

Субклассы унаследовали этот метод от Animal:

```
def talk
  puts "#{@name} says Bark!"
end
```

Такое поведение нормально для `Dog`, но не для `Cat` или `Bird`.

```
whiskers = Cat.new
whiskers.name = "Whiskers"
polly = Bird.new
polly.name = "Polly"

whiskers.talk
polly.talk
```

```
Whiskers says Bark!
Polly says Bark!
```

Следующий код переопределяет метод `talk`, унаследованный от `Animal`:

```
class Cat < Animal
  def talk                                  # Переопределяет унаследованный метод.
    puts "#{@name} says Meow!"
  end
end

class Bird < Animal
  def talk                                  # Переопределяет унаследованный метод.
    puts "#{@name} says Chirp! Chirp!"
  end
end
```

Теперь при вызове метода `talk` для экземпляров `Cat` или `Bird` будет вызвана переопределенная версия.

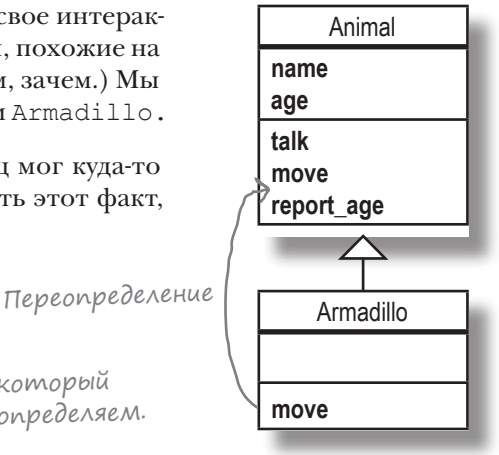
```
whiskers.talk
polly.talk
```

```
Whiskers says Meow!
Polly says Chirp! Chirp!
```

Как добраться до переопределяемого метода?

На следующем шаге компания Friends хочет добавить в свое интерактивное приложение броненосцев. (Да, это такие зверьки, похожие на муравьеда, которые могут сворачиваться в шар. Не знаем, зачем.) Мы можем просто добавить новый субкласс `Animal` с именем `Armadillo`.

Но здесь возникает одна проблема: чтобы броненосец мог куда-то пойти, он должен сначала развернуться. Чтобы отразить этот факт, мы переопределим метод `move`.




```
class Animal
  ...
  def move(destination)
    puts "#{@name} runs to the #{destination}."
  end
  ...
end
```

Метод, который мы переопределяем.

Наш субкласс.

Переопределяет метод «move» из суперкласса

```
class Armadillo < Animal

  def move(destination)
    puts "#{@name} unrolls!"
    puts "#{@name} runs to the #{destination}."
  end

end
```

Новая функциональность.

Этот код дублируется из метода суперкласса. (Здесь всего одна строка, но в реальном приложении их будет намного больше.)

Работает, но дублирование кода метода `move` из класса `Animal` нежелательно.

А если бы мы могли переопределить метод `move` в новом коде, и *при этом* использовать код из суперкласса? В Ruby для этого существует специальный механизм...

Ключевое слово «super»

При включении ключевого слова `super` в метод оно вызывает версию одноименного метода из суперкласса.

```
class Person

  def greeting
    puts "Hello!"
  end

end


class Friend < Person

  def greeting
    super
    puts "Glad to see you!"
  end

end
```

«super» вызывает этот код.

При вызове *переопределяющего* метода в *субклассе* ключевое слово `super` вызывает *переопределяемый* метод в *суперклассе*:

```
Friend.new.greeting
```

```
Hello!
Glad to see you!
```

Ключевое слово `super` почти во всех отношениях не отличается от обычного вызова метода.

Например, возвращаемое значение метода из суперкласса становится значением выражения `super`:

```
class Person

  def greeting
    "Hello!"
  end

end

class Friend < Person

  def greeting
    basic_greeting = super
    "#{basic_greeting} Glad to see you!"
  end

end

puts Friend.new.greeting
```

Возвращаемое значение метода.

Переменной basic_greeting присваивается «Hello!»

```
Hello! Glad to see you!
```

Ключевое слово «super» (продолжение)

Другой способ использования ключевого слова `super` напоминает обычный вызов метода: с ним указываются аргументы, которые будут переданы методу из суперкласса.

```
class Person

  def greet_by_name(name)
    "Hello, #{name}!"
  end

end

class Friend < Person

  def greet_by_name(name)
    basic_greeting = super(name)
    "#{basic_greeting} Glad to see you!"
  end

end

puts Friend.new.greet_by_name("Meghan")
```

При вызове метода передается аргумент.

```
Hello, Meghan! Glad to see you!
```

Но в одном отношении ключевое слово `super` *отличается* от обычного вызова метода: если аргументы *не указаны*, то метод из суперкласса автоматически вызывается с теми же аргументами, которые были переданы методу из субкласса.

```
class Friend < Person

  def greet_by_name(name)
    basic_greeting = super
    "#{basic_greeting} Glad to see you!"
  end

end

puts Friend.new.greet_by_name("Bert")
```

Метод greet_by_name класса Friend должен вызываться с аргументом «name»...

...поэтому аргумент «name» также будет передан методу greet_by_name класса Person.

```
Hello, Bert! Glad to see you!
```

Будьте осторожны!

Вызовы `super` и `super()` — не одно и то же.

Ключевое слово `super` без скобок вызывает переопределенный метод с теми же аргументами, которые были переданы переопределяющему методу. С другой стороны, вызов `super()` вызывает переопределенный метод без аргументов, даже если переопределяющий метод получает аргументы.

Субкласс обретает суперсилу

А теперь воспользуемся тем, что мы узнали о ключевом слове `super`, и устраним дублирующийся код из метода `move` класса `Armadillo`.

Метод, унаследованный от суперкласса `Animal`:

```
class Animal
  ...
  def move(destination)
    puts "#{@name} runs to the #{destination}."
  end
  ...
end
```

Повторяющаяся строка.

А это переопределенная версия из субкласса `Armadillo`:

```
class Armadillo < Animal

  def move(destination)
    puts "#{@name} unrolls!"
    puts "#{@name} runs to the #{destination}."
  end

end
```

Мы можем заменить повторяющийся код в методе `move` субкласса вызовом `super` и положиться на то, что метод `move` суперкласса предоставит нужную функциональность.

Здесь мы явно передаем параметр `destination`, который должен использоваться методом `move` класса `Animal`:

```
class Armadillo < Animal

  def move(destination)
    puts "#{@name} unrolls!"
    super(destination)
  end

end
```

Аргумент передается явно...

...ИЛИ...

А можно поступить иначе — не указывать аргументы при вызове `super`, чтобы параметр `destination` был передан методу `move` суперкласса автоматически:

```
class Armadillo < Animal

  def move(destination)
    puts "#{@name} unrolls!"
    super
  end

end
```

Автоматически передается тот же аргумент (или аргументы), с которым был вызван метод «move».

В любом случае код прекрасно работает!

```
dillon = Armadillo.new
dillon.name = "Dillon"
dillon.move("burrow")
```

```
Dillon unrolls!
Dillon runs to the burrow.
```

Ваше мастерство в применении наследования привело к тому, что все дублирование кода исчезло, словно вода из отжатой губки. Ваши коллеги благодарны: меньше кода — меньше ошибок! Отличная работа!

Упражнение

Ниже приведен код трех классов Ruby. Фрагменты кода справа используют эти классы напрямую или через механизм наследования. Заполните пропуски под каждым фрагментом и запишите, как, по вашему мнению, будет выглядеть вывод. И не забудьте учесть переопределение методов и ключевое слово `super`!

(Мы заполнили первый ответ за вас.)

```
class Robot

  attr_accessor :name

  def activate
    puts "#{@name} is powering up"
  end

  def move(destination)
    puts "#{@name} walks to #{destination}"
  end

end

class TankBot < Robot

  attr_accessor :weapon

  def attack
    puts "#{@name} fires #{@weapon}"
  end

  def move(destination)
    puts "#{@name} rolls to #{destination}"
  end

end

class SolarBot < Robot

  def activate
    puts "#{@name} deploys solar panel"
    super
  end

end
```

Ваши ответы:

```
tank = TankBot.new
tank.name = "Hugo"
tank.weapon = "laser"
tank.activate
tank.move("test dummy")
tank.attack
```

Hugo is powering up

.....

.....

.....

```
sunny = SolarBot.new
sunny.name = "Sunny"
sunny.activate
sunny.move("tanning bed")
```

.....

.....

.....

Упражнение
Решение

Ниже приведен код трех классов Ruby. Фрагменты кода справа используют эти классы напрямую или через механизм наследования. Заполните пропуски под каждым фрагментом и запишите, как, по вашему мнению, будет выглядеть вывод. И не забудьте учесть переопределение методов и ключевое слово `super`!

```
class Robot

  attr_accessor :name

  def activate
    puts "#{@name} is powering up"
  end

  def move(destination)
    puts "#{@name} walks to #{destination}"
  end

end

class TankBot < Robot

  attr_accessor :weapon

  def attack
    puts "#{@name} fires #{@weapon}"
  end

  def move(destination)
    puts "#{@name} rolls to #{destination}"
  end

end

class SolarBot < Robot

  def activate
    puts "#{@name} deploys solar panel"
    super
  end

end
```

```
tank = TankBot.new
tank.name = "Hugo"
tank.weapon = "laser"
tank.activate
tank.move("test dummy")
tank.attack
```

Hugo is powering up

Hugo rolls to test dummy

Hugo fires laser

```
sunny = SolarBot.new
sunny.name = "Sunny"
sunny.activate
sunny.move("tanning bed")
```

Sunny deploys solar panel

Sunny is powering up

Sunny walks to tanning bed

Трудности с выводом Dog

Прежде чем заявить, что класс `Dog` готов, стоит внести еще одно усовершенствование. В текущей версии при передаче экземпляра `Dog` методам `print` или `puts` пользы от вывода будет немного:

```
lucy = Dog.new
lucy.name = "Lucy"
lucy.age = 4

rex = Dog.new
rex.name = "Rex"
rex.age = 2

puts lucy, rex
```

Полученный результат:

```
#<Dog:0x007fb2b50c4468>
#<Dog:0x007fb2b3902000>
```

Из вывода можно понять, что это объекты `Dog`, но в остальном отличить один объект `Dog` от другого будет непросто. Было бы намного удобнее, если бы выходные данные объектов выглядели бы так:

```
Lucy the dog, age 4
Rex the dog, age 2
```

А такой результат нам ХОТЕЛОСЬ бы получить...

Когда объект передается методу `puts`, Ruby вызывает для него метод `to_s`, чтобы преобразовать этот объект в выводимую строку. Если вызвать `to_s` напрямую, вы получите тот же результат:

```
puts lucy.to_s, rex.to_s
```

```
#<Dog:0x007fb2b50c4468>
#<Dog:0x007fb2b3902000>
```

А теперь вопрос: откуда взялся метод экземпляра `to_s`?

В самом деле, откуда берется *большинство* методов экземпляра объектов `Dog`? Если вызвать метод с именем `methods` для экземпляра `Dog`, то только несколько первых методов экземпляра в списке выглядят знакомо...

```
puts rex.methods
```

```
name
age
name=
age=
talk
move
report_age
eql?
hash
class
clone
to_s
inspect
methods
object_id
...
```

Эти методы унаследованы от Animal...

...но откуда взялись эти методы?

Методов так много, что все они не поместятся на странице!

Методы экземпляра с именами `clone`, `hash`, `inspect`... В классе `Dog` мы их не определяли. И от суперкласса `Animal` они тоже не наследовались.

Но — и это кому-то может показаться удивительным — они все же *были* унаследованы от *чего-то*.

Класс Object

Откуда наши экземпляры Dog могли унаследовать все эти методы экземпляра? В суперклассе Animal они не определялись. А суперкласс для Animal мы не указывали...

```
class Dog < Animal
end
```

Суперклассом для Dog является класс Animal.

```
class Animal
  ...
end
```

Суперкласс не указан!

У классов Ruby существует метод superclass, который можно вызвать для получения их суперкласса. Результат вызова этого метода для Dog вполне предсказуем:

```
puts Dog.superclass
```

```
Animal
```

Но что произойдет, если вызвать метод superclass для Animal?

```
puts Animal.superclass
```

```
Object
```

Вот это да! Откуда *это* здесь взялось?

При определении нового класса Ruby неявно назначает класс с именем Object его суперклассом (если только вы не указали суперкласс сами).

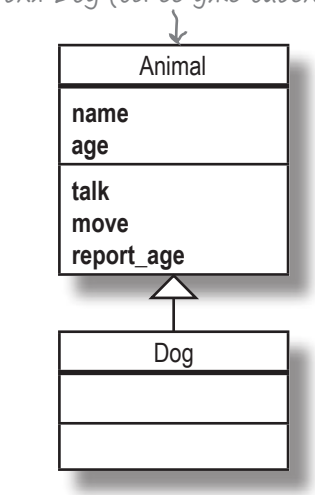
Итак, запись вида:

```
class Animal
  ...
end
```

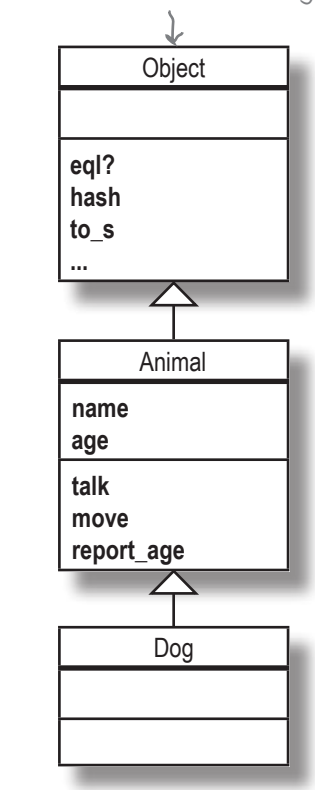
...эквивалентна следующей:

```
class Animal < Object
  ...
end
```

Диаграмма наследования для Dog (вы ее уже видели):



Реальная диаграмма наследования для Dog:



Почему все классы наследуют от Object

Если вы явно не укажете суперкласс для определяемого вами класса, то Ruby неявно назначает его суперклассом класс с именем `Object`.

```
class Animal < Object
  ...
end

class Dog < Animal
end
```

Неявно подставляется Ruby.

Наследует от Animal — а значит, в конечном итоге от Object!

Впрочем, даже если вы *укажете* суперкласс для своего класса, то, скорее всего, этот суперкласс наследует от `Object`. А это означает, что почти любой объект Ruby явно или косвенно происходит от суперкласса `Object`!

Ruby действует так из-за того, что класс `Object` определяет десятки полезных методов, которые нужны почти всем объектам Ruby. В их число входят многие методы, которые мы уже вызывали для объектов:

- Метод `to_s` преобразует объект в строку для вывода.
- Метод `inspect` преобразует объект в строку для отладки.
- Метод `class` сообщает, экземпляром какого класса является объект.
- Метод `methods` выводит список методов экземпляра, поддерживаемых объектом.
- Метод `instance_variables` возвращает список переменных экземпляра, содержащихся в объекте.

Кроме этих методов, есть еще много других. Методы, унаследованные от класса `Object`, играют ключевую роль в работе с объектами в языке Ruby.

Надеемся, это небольшое отступление показалось вам полезным, но оно не помогает нам в решении исходной задачи: вместо объектов `Dog` выводится какая-то белиберда.

Хотя... *так ли это?*

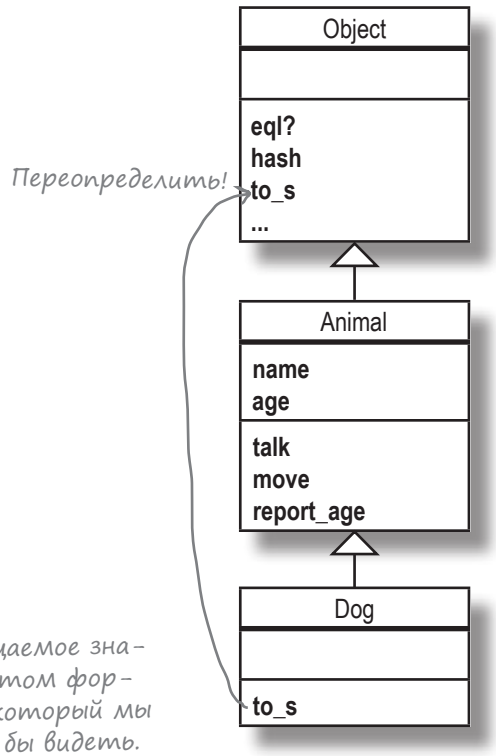
Объекты Ruby наследуют десятки полезных методов от класса Object.

Переопределение унаследованного метода

Мы указали, что суперклассом класса `Dog` является класс `Animal`. И еще, как было сказано в тексте, поскольку суперкласс для `Animal` *не был* задан, Ruby автоматически назначает его суперклассом класс `Object`.

Это означает, что экземпляры `Animal` наследуют метод `to_s` от `Object`. Экземпляры `Dog`, в свою очередь, наследуют `to_s` от `Animal`. Когда объект `Dog` передается методу `puts` или `print`, вызывается его метод `to_s` для преобразования его в строку.

Видите, к чему мы клоним? Если невразумительные строки, выводимые для экземпляров `Dog`, генерируются методом `to_s` и метод `to_s` является *унаследованным*, то нам остается лишь *переопределить* `to_s` *для класса* `Dog`!




```
class Dog < Animal

  def to_s
    "#{@name} the dog, age #{age}"
  end

end
```

Возвращаемое значение в том формате, который мы хотели бы видеть.

Готовы? Давайте попробуем.

```
lucy = Dog.new
lucy.name = "Lucy"
lucy.age = 4

rex = Dog.new
rex.name = "Rex"
rex.age = 2

puts lucy.to_s, rex.to_s
```

```
Lucy the dog, age 4
Rex the dog, age 2
```

Работает! Больше никаких `#<Dog:0x007fb2b50c4468>`. Теперь результат нормально читается!

И последнее усовершенствование: метод `to_s` автоматически вызывается при выводе объектов, поэтому его вызов можно опустить:

```
puts lucy, rex
```

```
Lucy the dog, age 4
Rex the dog, age 2
```

С новым форматом выходных данных отладка приложения заметно упростилась. А вы узнали один из ключевых моментов в работе объектов Ruby: наследование играет в нем очень важную роль!

Часто Задаваемые Вопросы

В: Я попробовал выполнить этот код в irb вместо использования команды `ruby`. Если после переопределения `to_s` я ввожу команду `lucy = Dog.new` в irb, я по-прежнему получаю строку вида `#<Dog:0x007fb2b50c4468>`. Почему я не вижу имя и возраст собаки?

О: Значения, выводимые irb, получены в результате вызова для объекта метода `inspect`, а не `to_s`. Чтобы увидеть результат `to_s`, необходимо задать name и age и передать объект `puts`.

Ваш инструментарий Ruby

Глава 3 осталась позади, а в вашем инструментарии появилось наследование.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Любой класс Ruby может использоваться в качестве суперкласса.
- Чтобы определить субкласс, просто укажите суперкласс в определении класса.
- Переменные экземпляра *не* наследуются от суперкласса — но *наследуются* методы, которые создают переменные экземпляра и обращаются к ним.
- Ключевое слово `super` может использоваться в методах субклассов для вызова одноименного переопределенного метода из суперкласса.
- Если ключевому слову `super` аргументы не передаются, то Ruby берет все аргументы, с которыми был вызван метод субкласса, и передает их методу суперкласса.
- Значением выражения ключевого слова `super` является возвращаемое значение вызываемого метода суперкласса.
- Если при определении класса его суперкласс не задается явно, то Ruby неявно назначает им класс `Object`.
- Почти каждый объект Ruby содержит методы экземпляра из класса `Object`, унаследованные напрямую или через другой суперкласс.
- Методы `to_s`, `methods`, `instance_variables` и `class` унаследованы от класса `Object`.

Далее в программе...

А что произойдет, если создать новый экземпляр `Dog`, но вызвать `move` для него *до того*, как будет задан атрибут `name`? (Попробуйте, если интересно; результат вас вряд ли обрадует.) В следующей главе рассматривается метод `initialize`, который помогает предотвратить неприятности такого рода.

Хороший старт — половина дела

Пока что ваш класс напоминает бомбу с часовым механизмом. Каждый экземпляр, созданный вами, начинается «с пустого места». Если вы вызовете методы экземпляра до того, как в нем появятся данные, то может произойти ошибка, приводящая к аварийному завершению программы.

В этой главе мы представим пару способов создания объектов, которые можно сразу безопасно использовать. Начнем с метода `initialize`, который позволяет передать набор аргументов для заполнения данных объекта *на момент его создания*. Затем мы покажем, как написать **методы класса**, которые позволяют **еще** проще создавать и инициализировать объекты.

Зарплата в Chargemore

Вам поручено создать систему начисления зарплаты для Chargemore — новой сети универсальных магазинов. Система должна печатать платежные квитанции для работников.

Зарплата в Chargemore выплачивается раз в две недели. Одни работники получают двухнедельную долю своего годового оклада, другим деньги начисляются за фактическое количество проработанных часов за двухнедельный период. Впрочем, начнем мы со штатных работников с фиксированным окладом.

Платежная квитанция должна включать следующую информацию:

- Имя работника.
- Сумму, начисленную работнику за двухнедельный платежный период.

Итак... Что же должна *знать* такая система о каждом работнике?

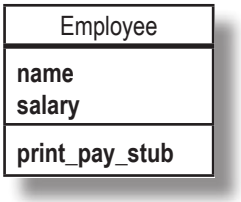
- Имя работника.
- Зарплату работника.

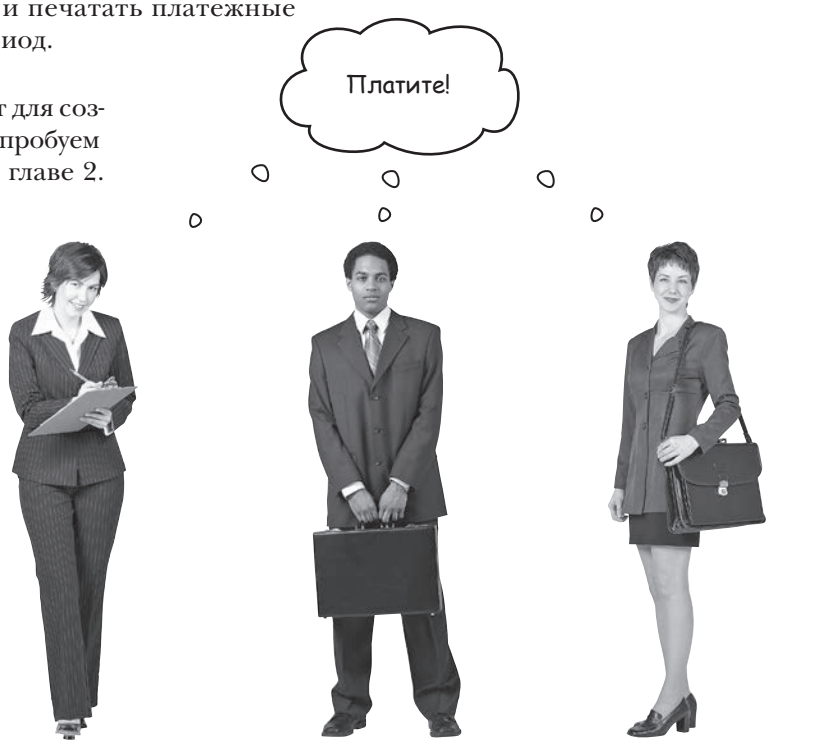
А вот что она должна *делать*:

- Вычислять начисленную сумму и печатать платежные квитанции за двухнедельный период.

Похоже, ситуация идеально подходит для создания класса работника `Employee`! Попробуем сделать это так же, как мы делали в главе 2.

Начнем с создания методов чтения атрибутов для переменных `@name` и `@salary`, а затем напишем методы записи (с проверкой). Далее мы напишем метод экземпляра `print_pay_stub`, который выводит имя работника и его оплату за период.





@name = "Kara Byrd"
@salary = 45000

@name = "Ben Weber"
@salary = 50000

@name = "Amy Blake"
@salary = 50000

Класс Employee

Ниже приведен код, использованный для реализации класса `Employee`...

```
class Employee

  attr_reader :name, :salary

  def name=(name)
    if name == ""
      raise "Name can't be blank!"
    end
    @name = name
  end

  def salary=(salary)
    if salary < 0
      raise "A salary of #{salary} isn't valid!"
    end
    @salary = salary
  end

  def print_pay_stub
    puts "Name: #{@name}"
    pay_for_period = (@salary / 365) * 14
    puts "Pay This Period: $#{pay_for_period}"
  end

end
```

Методы записи атрибутов необходимо создавать вручную, чтобы организовать проверку данных. Впрочем, методы чтения атрибутов можно сгенерировать автоматически.

Если имя остается пустым, вывести сообщение об ошибке.

Сохранить имя в переменной экземпляра.

Вывести сообщение об ошибке, если годовой оклад отрицателен.

Сохранить оклад в переменной экземпляра salary.

Вывести имя работника.

вычислить 14-дневную часть оклада работника.

Вывести начисленную сумму.

(Да, мы понимаем, что здесь не учитываются високосные годы, праздники и другие моменты, которые должны учитываться в настоящем приложении. Но мы хотели, чтобы код метода `print_pay_stub` помещался на одной странице.)

Создание новых экземпляров Employee

После того как класс `Employee` будет определен, мы сможем создавать новые экземпляры и присваивать значения их атрибутам `name` и `salary`.

```
amy = Employee.new
amy.name = "Amy Blake"
amy.salary = 50000
```

Код проверки в методе `name=` защищает от случайного присваивания пустых имен.

```
kara = Employee.new
kara.name = ""
```

Ошибка →

```
in `name=': Name can't be blank! (RuntimeError)
```

Метод `salary=` проверяет, что присваиваемое salary значение не отрицательно.

```
ben = Employee.new
ben.salary = -246
```

Ошибка →

```
in `salary=': A salary of -246 isn't valid! (RuntimeError)
```

После того как экземпляр `Employee` будет правильно инициализирован, мы можем использовать сохраненные значения имени и оклада для вывода информации за платежный период.

```
amy.print_pay_stub
```

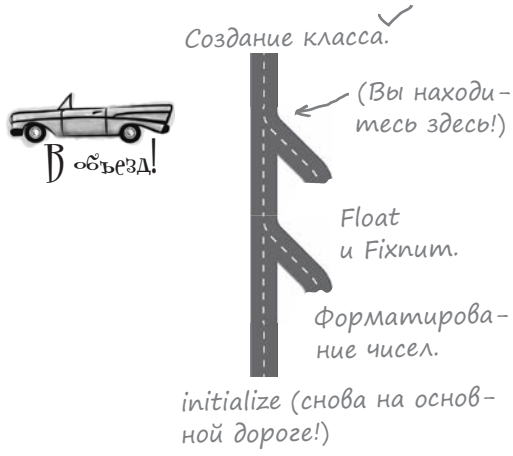
```
Name: Amy Blake
Pay This Period: $1904
```

← Неплохо, но куда делись центы?

Хмм... Вообще-то при выводе денежных сумм обычно выводятся две цифры в дробной части. Могло ли быть так, что в результате вычислений была получена круглая сумма в долларах?

Прежде чем переходить к доработке класса `Employee`, нужно разобраться с одной ошибкой. А для этого нам придется пару раз отклониться от основного пути. (Но вы освоите навыки форматирования чисел, которые пригодятся вам позднее — обещаем!)

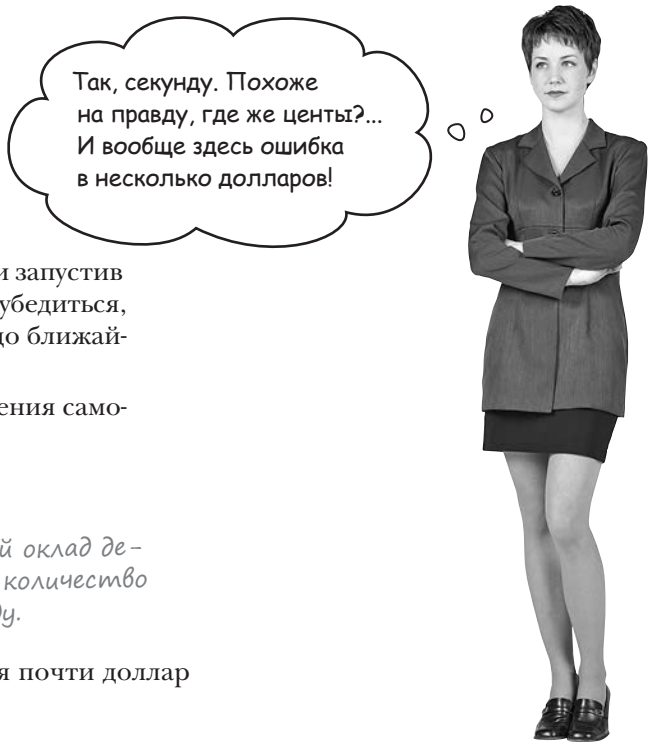
1. В нашей платежной системе дробная часть отсекается. Чтобы решить проблему, необходимо разобраться в различиях между числовыми классами Ruby `Float` и `Fixnum`.
2. Впрочем, выводить *слишком много* цифр в дробной части тоже нежелательно, поэтому нужно поближе познакомиться с методом `format` для форматирования чисел.



Трудности с делением

Мы продолжаем работу над идеальным классом `Employee` для начисления зарплаты в магазине Chargemore. Но сначала нужно разобраться с одной маленькой подробностью...

```
Name: Amy Blake
Pay This Period: $1904
```



А ведь и верно. Проведя вычисления на бумаге (или запустив приложение-калькулятор, если угодно), вы сможете убедиться, что Эми должна получить $1917.81 с округлением до ближайшего цента. Куда же делись $13.81?

Чтобы узнать это, запустим irb и проведем вычисления самостоятельно, шаг за шагом.

Начнем с вычисления дневной оплаты.

В irb:

```
>> 50000 / 365
=> 136
```

Ежегодный оклад делится на количество дней в году.

По сравнению с ручными вычислениями теряется почти доллар в день:

50,000 ÷ 365 = 136.9863...

Затем при вычислении оплаты за *четырнадцать* дней ошибка накапливается:

```
>> 136 * 14
=> 1904
```

Сравните с ответом, который мы получили бы при умножении на *полную* ставку за день:

136.9863 x 14 = 1917.8082...

Потеряно почти $14. Умножьте *это* на количество работников — и попробуйте выпутаться из неожиданных неприятностей. Проблему нужно решить, притом как можно скорее...

Деление с использованием класса Ruby Fixnum

Результат выражения Ruby для вычисления двухнедельной зарплаты не совпадает с результатом, полученным при ручных вычислениях...

```
>> 50000 / 365 * 14
=> 1904
```

50,000 ÷ 365 x 14 = 1917.8082...

Дело в том, что при делении экземпляров класса `Fixnum` (класс Ruby для представления целых чисел) Ruby округляет дробные числа *в меньшую сторону* до ближайшего целого числа.

```
>> 1 / 2
=> 0
```

Результат округляется в меньшую сторону!

Результат округляется, потому что экземпляры `Fixnum` *не предназначены* для хранения чисел с дробной частью. Они предназначены для ситуаций, в которых имеют смысл только целые величины — например, для подсчета работников в отделе или количества товаров в корзине покупателя. Создавая экземпляр `Fixnum`, вы фактически говорите Ruby: «Я собираюсь работать только с целыми числами. Если кто-то выполнит вычисления, которые приведут к дробному результату, эту лишнюю дробную часть следует просто отбросить».

Как узнать, работаем ли мы с экземплярами `Fixnum`? Можно вызвать для данных метод экземпляра `class`. (Помните класс `Object`, который упоминался в главе 3? Метод `class` — один из методов экземпляра, унаследованных от `Object`.)

```
>> salary = 50000
=> 50000
>> salary.class
=> Fixnum
```

Или если вы предпочитаете избавиться от лишних хлопот, просто запомните: любое число, *не содержащее* точки, интерпретируется Ruby как экземпляр `Fixnum`.

Любое число в вашем коде, *содержащее* точку, интерпретируется как экземпляр `Float` (класс Ruby, представляющий дробные числа с плавающей точкой):

```
>> salary = 50000.0
=> 50000.0
>> salary.class
=> Float
```

Если в числе есть точка — значит, это `Float`.
Если точки нет — значит, это `Fixnum`.

273 — **Fixnum**

273.4 — **Float**

Деление с классом Ruby Float

Итак, мы загрузили irb и увидели, что при делении одного экземпляра `Fixnum` (целого числа) на другой экземпляр `Fixnum` Ruby округляет результат *в меньшую сторону*.

Должно быть 136.9863...

```
>> 50000 / 365
=> 136
```

Что делать? Используйте в операциях экземпляры `Float`. Для этого достаточно включить в число точку. В этом случае Ruby вернет результат операции как экземпляр `Float`:

```
>> 50000.0 / 365.0
=> 136.986301369863
>> (50000.0 / 365.0).class
=> Float
```

При этом даже не обязательно, чтобы и делимое, и делитель были экземплярами `Float`; Ruby вернет `Float` в том случае, если *хотя бы один* операнд является экземпляром `Float`.

```
>> 50000.0 / 365
=> 136.986301369863
```

Это правило истинно также и для сложения, вычитания и умножения: Ruby возвращает `Float`, если *хотя бы один* операнд является экземпляром `Float`:

```
>> 50000 + 1.5
=> 50001.5
>> 50000 - 1.5
=> 49998.5
>> 50000 * 1.5
=> 75000.0
```

Если первый операнд...	А второй операнд...	То результат...
Fixnum	Fixnum	Fixnum
Fixnum	Float	Float
Float	Fixnum	Float
Float	Float	Float

И конечно, для сложения, вычитания и умножения, даже если оба операнда являются экземплярами `Fixnum`, это не создаст проблем, потому что отсутствует дробная часть, которая могла бы быть потеряна. Единственная операция, для которой это важно, — деление. Запомните следующее правило:

При выполнении деления проследите за тем, чтобы хотя бы один операнд был экзепляром `Float`.

А теперь посмотрим, удастся ли применить новые драгоценные познания для решения проблем с классом `Employee`.

Исправление ошибки округления в Employee

Если хотя бы один операнд является экземпляром `Float`, Ruby не будет отсекать дробную часть при делении.

```
>> 50000 / 365.0
=> 136.986301369863
```

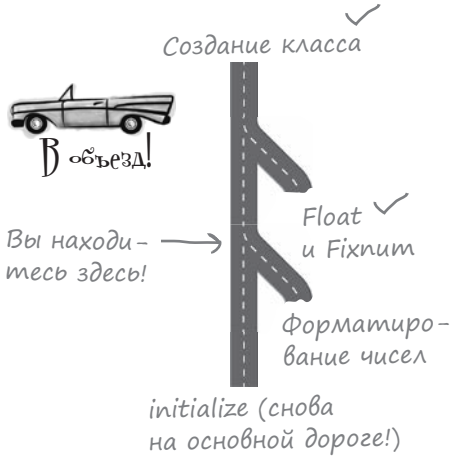
С учетом этого правила мы можем переработать класс `Employee`, чтобы предотвратить потерю дробной части при расчете зарплаты работника:

```
class Employee
  ...                                  # Код методов чтения/записи пропущен для краткости.
  def print_pay_stub
    puts "Name: #{@name}"
    pay_for_period = (@salary / 365.0) * 14   # Теперь независимо от того, является ли @salary экземпляром Float, будет получен результат Float.
    puts "Pay This Period: $#{pay_for_period}"   # Вывести сумму.
  end

end

employee = Employee.new
employee.name = "Jane Doe"
employee.salary = 50000                # Здесь использование Fixnum совершенно нормально!
employee.print_pay_stub
```

Но тут возникает новая проблема: только взгляните, что произошло с выводом!

```
Name: Jane Doe
Pay This Period: $1917.8082191780823
```

Теперь данные выводятся с *избыточной* точностью! В конце концов, денежные суммы обычно содержат всего два знака в дробной части. Итак, прежде чем возвращаться к построению идеального класса `Employee`, необходимо сделать еще одно лирическое отступление...



Форматирование чисел для вывода

Наш метод `print_pay_stub` выводит слишком много знаков в дробной части. Остается выяснить, как вывести сумму с округлением до ближайшего цента (два знака в дробной части).

```
Name: Jane Doe
Pay This Period: $1917.8082191780823
```

Для решения подобных проблем с форматированием Ruby предоставляет метод `format`.

Рассмотрим пример того, что можно сделать с помощью этого метода. На первый взгляд пример кажется довольно запутанным, но мы все объясним на нескольких ближайших страницах!

```
result = format("Rounded to two decimal places: %0.2f", 3.14159265)
puts result
```

Округляет число до двух знаков в дробной части и выводит его.

```
Rounded to two decimal places: 3.14
```

Похоже, метод `format` *может* ограничить выводимую сумму до нужного количества цифр в дробной части. Вопрос в том, *как* это сделать? Чтобы эффективно использовать этот метод, необходимо освоить две возможности `format`:

1. Форматные последовательности (странное `%0.2f` в приведенном примере — форматная последовательность).
2. Ширина форматной последовательности (`0.2` в середине формальной последовательности).

Сейчас мы объясним, что означают аргументы `format`.

Конечно, все эти вызовы методов выглядят непонятно. Многочисленные примеры помогут прояснить ситуацию. Основное внимание будет уделяться форматированию дробных чисел, потому что, скорее всего, в своей карьере Ruby-программиста вы будете использовать `format` в основном для этой цели.

Форматные последовательности

В первом аргументе `format` передается строка, используемая для форматирования выходных данных. Большая часть данных форматируется точно в таком виде, в котором они встречаются в строке. При этом знаки процента (`%`) интерпретируются как начало **форматной последовательности** — части строки, которая будет заменяться значением в определенном формате. Остальные аргументы содержат значения для форматных последовательностей.

Форматная последовательность

Форматная последовательность

```
puts format("The %s cost %i cents each.", "gumballs", 23)
puts format("That will be $%f please.", 0.23 * 5)
```

Форматная последовательность

```
The gumballs cost 23 cents each.
That will be $1.150000 please.
```

Вскоре мы покажем, как исправить этот недостаток.

Типы форматных последовательностей

Буква после знака процента обозначает тип предполагаемого значения. Самые распространенные типы:

- `%s` строка
- `%i` целое число
- `%f` дробное число с плавающей точкой

```
puts format("A string: %s", "hello")
puts format("An integer: %i", 15)
puts format("A float: %f", 3.1415)
```

```
A string: hello
An integer: 15
A float: 3.141500
```

Итак, тип `%f` предназначен для дробных чисел с плавающей точкой... Мы можем воспользоваться им для форматирования денежных сумм в платежных квитанциях.

Однако сам по себе тип последовательности `%f` особой пользы не принесет. Результат по-прежнему содержит слишком много цифр в дробной части.

```
puts format("$%f", 1917.8082191780823)
```

```
$1917.808219
```

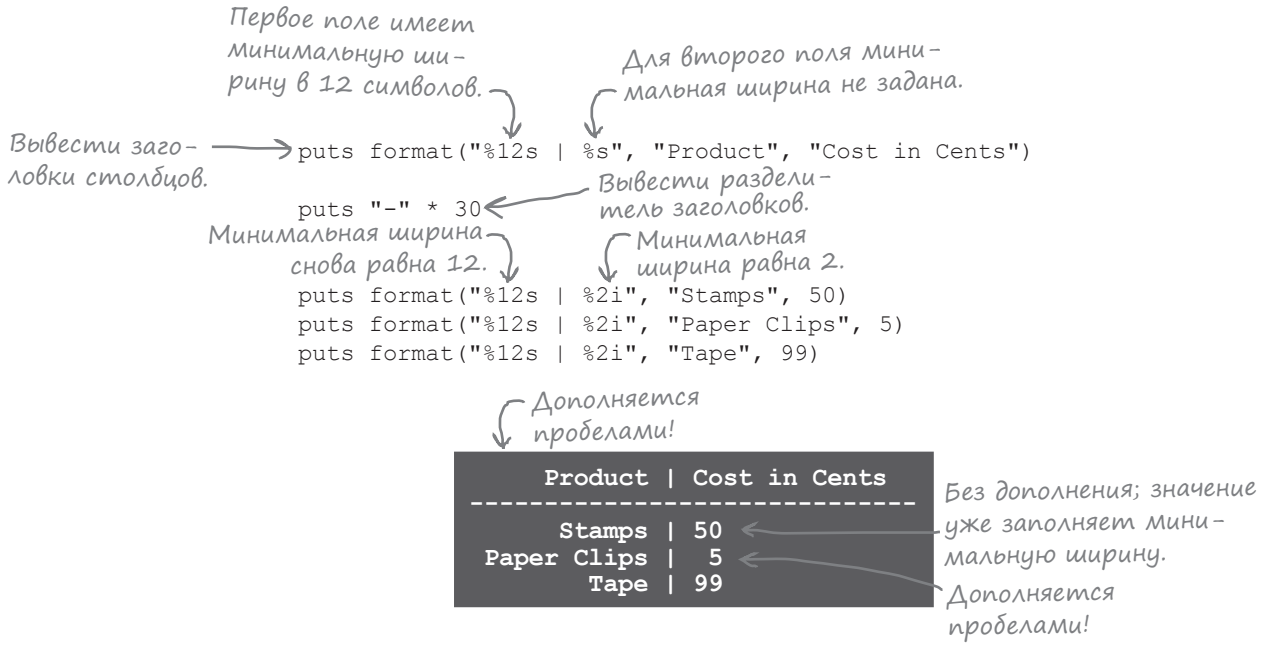
Сейчас мы покажем, как исправить ситуацию: в этом нам поможет *ширина* форматной последовательности.

Ширина форматной последовательности

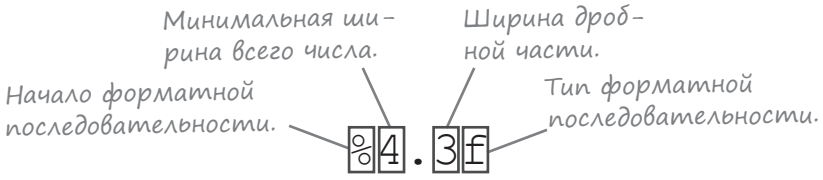
Это самая полезная часть форматной последовательности: она задает *ширину* поля при выводе.

Допустим, мы хотим отформатировать данные в виде текстовой таблицы. Нужно позаботиться о том, чтобы отформатированное значение заполняло минимальное количество позиций, а столбцы правильно выравнивались.

Минимальная ширина может задаваться после знака % в форматной последовательности. Если аргумент форматной последовательности короче минимума, то он дополняется пробелами до минимальной ширины.



А теперь мы подходим к тому, что действительно важно для текущей задачи: ширина форматной последовательности может использоваться для определения точности (количества выводимых цифр) чисел с плавающей точкой. Формат выглядит так:



Минимальная ширина всего числа включает разряды дробной части. Если она включена, то более короткие числа будут дополняться пробелами в начале до достижения минимальной ширины. Если минимальная ширина не указана, то пробелы не добавляются.

Ширина после точки определяет максимальное количество выводимых цифр. Число с большей точностью округляется (в большую или меньшую сторону) до заданного количества цифр в дробной части.

Ширина форматной последовательности для дробных чисел

Итак, при работе с числами с плавающей точкой спецификатор ширины форматной последовательности позволяет задать количество цифр, выводимых до *и* после точки. Нельзя ли воспользоваться этим обстоятельством для решения проблемы с выводом сумм?

Ниже кратко продемонстрированы примеры различных значений ширины:

```
def test_format(format_string)
  print "Testing '#{format_string}': "
  puts format(format_string, 12.3456)
end

test_format "%7.3f"
test_format "%7.2f"
test_format "%7.1f"
test_format "%.1f"
test_format "%.2f"
```

```
Testing '%7.3f':  12.346
Testing '%7.2f':   12.35
Testing '%7.1f':    12.3
Testing '%.1f': 12.3
Testing '%.2f': 12.35
```

Округляется до трех цифр.
Округляется до двух цифр.
Округляется до одной цифры.
Округляется до одной цифры без дополнения пробелами.
Округляется до двух цифр без дополнения пробелами.

Последний формат `"%.2f"` берет дробное число произвольной точности и округляет его до двух цифр в дробной части. (Лишнее дополнение пробелами при этом не производится.) Этот формат, идеально подходящий для вывода денежных сумм, — именно то, что нужно для нашего метода `print_pay_stub`!

```
puts format("$%.2f", 2514.2727367874069)
puts format("$%.2f", 1150.6849315068494)
puts format("$%.2f", 3068.4931506849316)
```

```
$2514.27
$1150.68
$3068.49
```

Во всех случаях округляется до двух цифр!

Ранее вычисленная зарплата в методе `print_pay_stub` класса `Employee` выводилась с лишними цифрами в дробной части:

```
salary = 50000
puts "$#{(salary / 365.0) * 14}"
```

```
$1917.8082191780823
```

Но теперь у нас есть форматная последовательность, которая округляет дробное число до двух цифр в дробной части:

```
puts format("$%.2f", (salary / 365.0) * 14 )
```

```
$1917.81
```

Попробуем воспользоваться методом `format` в методе `print_pay_stub`.

```
class Employee
  ...
  def print_pay_stub
    puts "Name: #{@name}"
    pay_for_period = (@salary / 365.0) * 14
    formatted_pay = format("%.2f", pay_for_period)
    puts "Pay This Period: $#{formatted_pay}"
  end
end
```

Получение строки с денежной суммой, округленной до двух цифр в дробной части.

Вывод строки с отформатированной суммой.

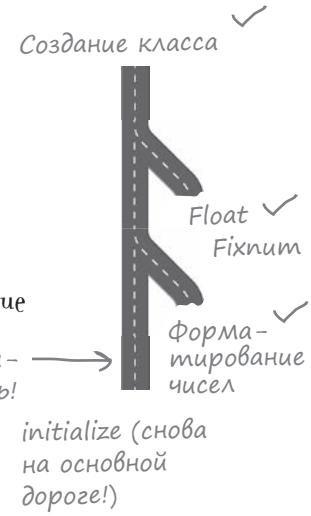
Применение метода «format» для исправления вывода

Для тестирования переработанного метода `print_pay_stub` мы воспользуемся теми же значениями, что и прежде:

```
amy = Employee.new
amy.name = "Amy Blake"
amy.salary = 50000
amy.print_pay_stub
```

```
Name: Amy Blake
Pay This Period: $1917.81
```

Нам пришлось пару раз отклониться от основного пути, но в итоге класс `Employee` печатает платежные квитанции так, как положено! А теперь вернемся к задаче по доработке класса...



Присмотритесь к каждой из этих команд Ruby и запишите, как по вашему мнению будет выглядеть результат. Учтите как результат операции деления, так и форматирование, применяемое к ее результату. Первое решение мы уже записали за вас.

```
format "%.2f", 3 / 4.0
```

0.75

```
format "$%.2f", 3 / 4.0
```

.....

```
format "%.2f", 3 / 4
```

.....

```
format "%.1f", 3 / 4.0
```

.....

```
format "%i", 3 / 4.0
```

.....

Упражнение
Решение

Присмотритесь к каждой из этих команд Ruby и запишите, как по вашему мнению будет выглядеть результат. Учтите как результат операции деления, так и форматирование, применяемое к ее результату. Первое решение мы уже записали за вас.

```
format "%.2f", 3 / 4.0
```
0.75 ← Форматная последовательность указывает, что должны выводиться две цифры в дробной части.

```
format "%.1f", 3 / 4.0
```
0.8 ← Значение не поместится в заданное количество позиций в дробной части, поэтому оно округляется.

```
format "$%.2f", 3 / 4.0
```
$0.75 ← Части строки, не входящие в форматную последовательность, выводятся без изменений.

```
format "%i", 3 / 4.0
```
0 ← Форматная последовательность %i выводит целое число, поэтому аргумент округляется в меньшую сторону.

```
format "%.2f", 3 / 4
```
0.00 ← Оба операнда при делении являются целыми числами. Результат округляется в МЕНЬШУЮ сторону до целого числа (0).

Если атрибуты объекта не заданы

Итак, теперь зарплата выводится в правильном формате, и вы радостно поручаете новому классу `Employee` обработку платежной ведомости. Вернее, это длится до того момента, когда вы создаете новый экземпляр `Employee` и забываете задать атрибуты `name` и `salary` перед вызовом `print_pay_stub`:

```
employee = Employee.new
employee.print_pay_stub
```

Ошибкой не является, но выводится пустое значение!

```
Name:
in `print_pay_stub': undefined method
`/' for nil:NilClass
```
← Ошибка!

Что произошло? Пустое имя вполне естественно; мы забыли его задать. Но что это за ошибка «неопределенного метода для nil»? Что это вообще такое — `nil`?

Подобные ошибки встречаются в Ruby довольно часто, поэтому мы посвятим несколько страниц тому, чтобы разобраться с ней.

Давайте изменим метод `print_pay_stub` так, чтобы он выводил значения `@name` и `@salary` и мы могли разобраться в происходящем.

```
class Employee

  ...

  def print_pay_stub
    puts @name, @salary
  end

end
```
Вывод значений.

Позднее мы вернем код на место.

«nil» означает «ничто»

Теперь создадим новый экземпляр `Employee` и вызовем переработанный метод:

Должно выводить @name и @salary.

```
employee = Employee.new
employee.print_pay_stub
```

Две пустые строки!

Да, от *этого* пользы немного. Похоже, мы что-то упустили.

В главе 1 мы узнали, что методы `inspect` и `p` могут предоставить информацию, которая не видна в обычном выводе. Проверим вывод, полученный при вызове `p`:

```
class Employee
  ...
  def print_pay_stub
    p @name, @salary
  end
end
```

Вывод значений в отладочном формате.

Мы создаем другой экземпляр, снова вызываем метод экземпляра, и...

```
employee = Employee.new
employee.print_pay_stub
```

```
nil
nil
```

АГА!

В Ruby существует специальное значение `nil`, представляющее *ничто*. Другими словами, оно представляет *отсутствие* значения.

То, что `nil` *представляет* ничто, не означает, что это *действительно* ничто. Как и все остальные сущности в Ruby, это объект, и у него есть свой класс:

```
puts nil.class
```

```
NilClass
```

Но если там что-то есть, то почему мы ничего не увидели при выводе?

Потому что метод экземпляра `to_s` из класса `NilClass` всегда возвращает пустую строку.

```
puts nil.to_s
```

Пустая строка!

Методы `puts` и `print` автоматически вызывают `to_s` для объекта, чтобы преобразовать его в строку для вывода. Именно поэтому при попытке вывода значений `@name` и `@salary` методом `puts` выводятся две пустые строки; обоим присвоены `nil`.

В отличие от `to_s`, метод экземпляра `inspect` из `NilClass` всегда возвращает строку `"nil"`.

```
puts nil.inspect
```

```
nil
```

Ранее мы уже упоминали о том, что метод `p` вызывает `inspect` для каждого объекта перед выводом. Вот почему значения `nil` в `@name` и `@salary` появились в выходных данных, когда мы вызвали для них метод `p`.

«/» — это метод

Итак, при создании экземпляра класса `Employee` его переменные экземпляра `@name` и `@salary` содержат значение `nil`. Проблемы создает переменная `@salary`, если вы вызовете метод `print_pay_stub` без предварительного присваивания значения:

Ошибка →
```
in `print_pay_stub': undefined method `/' for nil:NilClass
```
Значение «nil»!

Из описания ошибки очевидно следует, что проблема возникла из-за `nil`. Но в нем говорится о неопределенном методе `'/'`... Выходит, операция деления является методом?

В языке Ruby ответ на этот вопрос положителен; многие математические операторы реализованы в виде методов. Но когда Ruby встречает в коде конструкцию следующего вида:

```
6 + 2
```

...эта конструкция преобразуется в вызов метода с именем `+` для объекта `Fixnum` со значением `6`; при этом объект справа от `+` (то есть `2`) передается как аргумент:

Вызов метода! Другой операнд передается в аргументе.
```
6.+(2)
```

Обе формы абсолютно законны в Ruby. При желании можете убедиться в этом:

```
puts 6 + 2
puts 6.+(2)
```
```
8
8
```

Это относится к большинству других математических операторов:

```
puts 7 - 3
puts 7.-(3)
puts 3.0 * 2
puts 3.0.*(2)
puts 8.0 / 4.0
puts 8.0./(4.0)
```
```
4
4
6.0
6.0
2.0
2.0
```

Даже операторы сравнения реализованы в виде методов:

```
puts 9 < 7
puts 9.<(7)
puts 9 > 7
puts 9.>(7)
```
```
false
false
true
true
```

Но если классы `Fixnum` и `Float` определяют эти методы операторов, `NilClass` их *не определяет*.

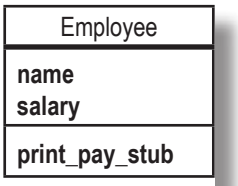
```
puts nil./(365.0)
```
Ошибка →
```
undefined method `/' for nil:NilClass
```

Более того, `nil` не определяет *большинство* методов экземпляра, поддерживаемых другими объектами Ruby.

Да и зачем их определять? Если с `nil` выполняются математические операции, то это почти наверняка произошло из-за того, что вы забыли присвоить значение одному из операндов. В такой ситуации *должно* быть выведено сообщение об ошибке, которое привлечет ваше внимание к проблеме.

Метод «initialize»

Мы попытались вызвать `print_pay_stub` для экземпляра класса `Employee`, но получили `nil` при обращении к переменным экземпляра `@name` и `@salary`.




```
employee = Employee.new
employee.print_pay_stub
```

И тут разражается хаос.

Не ошибка, а пустое значение!

```
Name:
in `print_pay_stub': undefined method
`/' for nil:NilClass
```

Ошибка!

А вот и метод, в котором значения `nil` породили столько проблем:

```
def print_pay_stub
  puts "Name: #{@name}"
  pay_for_period = (@salary / 365.0) * 14
  formatted_pay = format("$%.2f", pay_for_period)
  puts "Pay This Period: #{formatted_pay}"
end
```

Приводит к вызову to_s для @name. Так как переменная содержит nil, выводится пустая строка.

Приводит к вызову «/» (а это метод экземпляра!) для @salary. Так как значение равно nil, выдается сообщение об ошибке.

Здесь-то и проявляется суть проблемы: в момент создания экземпляра `Employee` он находится в некорректном состоянии. Вызов `print_pay_stub` для него небезопасен до тех пор, пока не будут заданы переменные экземпляра `@name` и `@salary`.

Если бы мы могли задавать значения `@name` и `@salary` *одновременно* с созданием экземпляра `Employee`, то это привело бы к снижению риска ошибок.

Ruby предоставляет механизм, упрощающий решение этой задачи: метод `initialize`. Метод `initialize` позволяет подключиться к процессу создания и сделать объект безопасным для использования, прежде чем кто-либо попробует вызывать для него методы.

```
class MyClass
  def initialize
    puts "Setting up new instance!"
  end
end
```

При вызове `MyClass.new` Ruby выделяет память для хранения нового объекта `MyClass`, после чего вызывает метод `initialize` для нового объекта.

```
MyClass.new
```

```
Setting up new instance!
```

Ruby вызывает метод `initialize` для новых объектов после их создания.

Безопасность использования Employee и «initialize»

Добавим метод `initialize`, который будет присваивать значения `@name` и `@salary` новых экземпляров `Employee` до того, как для них будут вызваны другие методы экземпляра.

```
class Employee

  attr_reader :name, :salary

  def name=(name)
    if name == ""
      raise "Name can't be blank!"
    end
    @name = name
  end

  def salary=(salary)
    if salary < 0
      raise "A salary of #{salary} isn't valid!"
    end
    @salary = salary
  end

  def initialize
    @name = "Anonymous"
    @salary = 0.0
  end

  def print_pay_stub
    puts "Name: #{@name}"
    pay_for_period = (@salary / 365.0) * 14
    formatted_pay = format("$%.2f", pay_for_period)
    puts "Pay This Period: #{formatted_pay}"
  end

end
```

Новый метод

Присваивает значение переменной экземпляра @name.

Присваивает значение переменной экземпляра @salary.

Теперь, когда мы создали метод `initialize`, значения `@name` и `@salary` будут автоматически присваиваться для всех новых экземпляров `Employee`. И вы можете спокойно вызвать для них метод `print_pay_stub` сразу же после создания.

```
employee = Employee.new
employee.print_pay_stub
```

Переменным @name и @salary присваиваются значения.

Выводится успешно.

```
Name: Anonymous
Pay This Period: $0.00
```

Аргументы «initialize»

Наш метод `initialize` теперь назначает по умолчанию `@name` имя `"Anonymous"`, а `@salary` — значение `0.0`. Впрочем, было бы лучше, если бы при инициализации объекта можно было предоставить другие значения, отличные от этих.

Именно для таких ситуаций любые аргументы, переданные при вызове метода `new`, передаются `initialize`.

```
class MyClass
  def initialize(my_param)
    puts "Got a parameter from 'new': #{my_param}"
  end
end

MyClass.new("hello")
```

Передается «initialize»!

```
Got a parameter from 'new': hello
```

Это обстоятельство позволяет указать при вызове исходные значения имени и оклада. Для этого необходимо лишь добавить параметры `name` и `salary` в определение `initialize` и использовать их для инициализации переменных экземпляра `name` и `@salary`.

```
class Employee

  ...

  def initialize(name, salary)
    @name = name
    @salary = salary
  end

  ...

end
```

Параметр «name» используется для инициализации переменной экземпляра «@name».

Параметр «salary» используется для инициализации переменной экземпляра «@salary».

Новые значения `@name` и `@salary` передаются в аргументах `Employee.new`!

```
employee = Employee.new("Amy Blake", 50000)
employee.print_pay_stub
```

Передаются «initialize»!

```
Name: Amy Blake
Pay This Period: $1917.81
```

Конечно, при таком подходе требуется осторожность. Если вы забудете передать `new` аргументы, то не будет аргументов, которые можно было бы передать `initialize`. И тогда произойдет то же, что происходит всегда при вызове метода Ruby с неправильным количеством аргументов: ошибка.

```
employee = Employee.new
```

Ошибка

```
in `initialize': wrong number
of arguments (0 for 2)
```

Вскоре вы узнаете, как можно решить эту проблему.

Необязательные параметры и «initialize»

Мы начали с метода `initialize`, который задает переменным экземпляра значения по умолчанию, но не позволяет задать собственные значения...

```
class Employee
  ...
  def initialize
    @name = "Anonymous"
    @salary = 0.0
  end
  ...
end
```

Инициализируется переменная экземпляра @name.

Инициализируется переменная экземпляра @salary.

Затем мы добавили в `initialize` параметры — это означало, что вы *должны* задать собственные значения имени и оклада и *не можете* полагаться на значения по умолчанию...

```
class Employee
  ...
  def initialize(name, salary)
    @name = name
    @salary = salary
  end
  ...
end
```

Параметр «name» используется для задания переменной экземпляра «@name».

Параметр «salary» используется для задания переменной экземпляра «@salary».

Нельзя ли совместить преимущества обоих решений?

Можно! Метод `initialize` ничем не отличается от других методов, поэтому он может пользоваться всеми возможностями обычных методов, в том числе и необязательными параметрами. (Еще не забыли — глава 2?)

При объявлении параметров можно задать значения по умолчанию. Если аргумент пропущен, то ему присваивается значение по умолчанию. Далее параметры, как обычно, присваиваются переменным экземпляра.

```
class Employee
  ...
  def initialize(name = "Anonymous", salary = 0.0)
    @name = name
    @salary = salary
  end
  ...
end
```

Параметрам задаются значения по умолчанию.

С таким изменением можно опустить один или оба аргумента — и все равно получить нормальные значения по умолчанию!

```
Employee.new("Jane Doe", 50000).print_pay_stub
Employee.new("Jane Doe").print_pay_stub
Employee.new.print_pay_stub
```

```
Name: Jane Doe
Pay This Period: $1917.81
Name: Jane Doe
Pay This Period: $0.00
Name: Anonymous
Pay This Period: $0.00
```

У бассейна

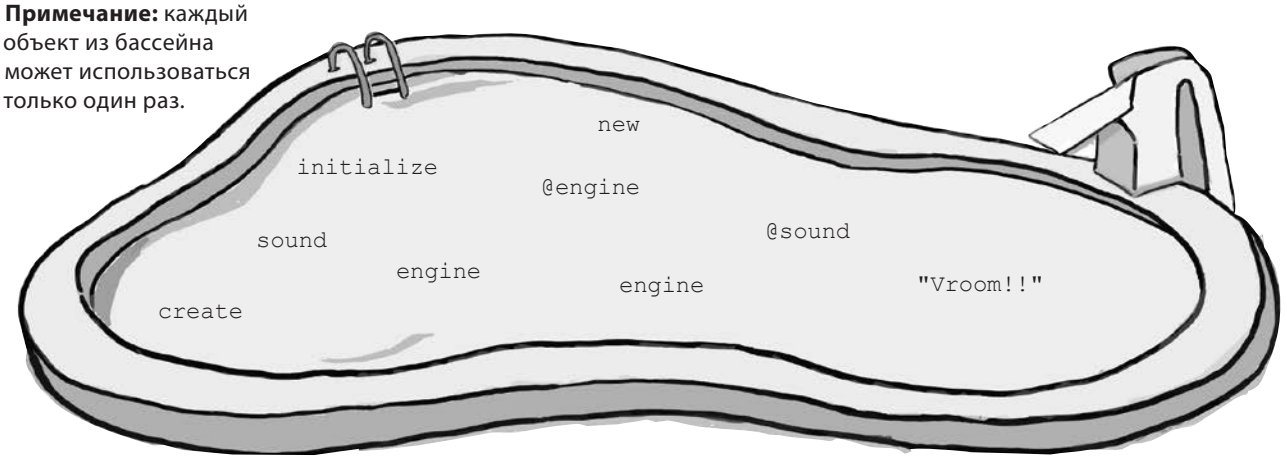
Выловите из бассейна фрагменты кода и расставьте их в пустых местах в коде. Каждый фрагмент может использоваться **только один** раз, причем использовать все фрагменты не обязательно. Ваша **задача** — составить код, который будет нормально выполняться и выдавать приведенный ниже результат.

```
class Car

  def __________(_______)
    _______ = engine
  end

  def rev_engine
    @engine.make_sound
  end

end

class Engine

  def initialize(_____ = __________)
    @sound = sound
  end

  def make_sound
    puts @sound
  end

end

engine = Engine.___
car = Car.new(_______)
car.rev_engine
```

Результат:

```
File Edit Window Help
Vroom!!
```

Примечание: каждый объект из бассейна может использоваться только один раз.



У бассейна. Решение

```
class Car

  def initialize(engine)
    @engine = engine
  end

  def rev_engine
    @engine.make_sound
  end

end

engine = Engine.new
car = Car.new(engine)
car.rev_engine
```

```
class Engine

  def initialize(sound = <<Vroom!!>>)
    @sound = sound
  end

  def make_sound
    puts @sound
  end

end
```

Результат:

```
File Edit Window Help
Vroom!!
```

Часто Задаваемые Вопросы

В: Чем методы `initialize` в языке Ruby отличаются от конструкторов в других объектно-ориентированных языках?

О: Они выполняют одну основную функцию: помогают классам подготовить новые экземпляры для использования. Но если в большинстве языков конструкторы относятся к специальным синтаксическим структурам, метод `initialize` в Ruby является обычным методом экземпляра.

В: Зачем вызывать `MyClass.new`? Нельзя ли сразу вызвать `initialize`?

О: Метод `new` необходим для фактического *создания* объекта; `initialize` задает значения переменных экземпляра. Без вызова `new` не будет объекта, который нужно инициализировать! По этой причине Ruby не позволяет вызывать метод `initialize` напрямую вне экземпляра. (Таким образом, мы немного упростили ситуацию — `initialize` все же *отличается* от обычных методов экземпляра.)

В: При вызове `MyClass.new` метод `initialize` всегда вызывается для нового объекта?

О: Да, всегда.

В: Тогда как мы вызывали `new` для классов, создававшихся до сих пор? В них не было методов `initialize`!

О: Вообще-то *были*... Все классы Ruby наследуют метод `initialize` от суперкласса `Object`.

В: Но если `Employee` наследует метод `initialize`, зачем мы пишем собственную версию?

О: Метод `initialize` из `Object` не получает аргументов и по сути ничего не делает. Он не задает переменные экземпляра за вас; для этого нам пришлось переопределить его собственной версией.

В: Можно ли вернуть значение из метода `initialize`?

О: Можно, но Ruby его проигнорирует. Метод `initialize` предназначен исключительно для создания новых экземпляров класса, так что если вам нужно вернуть значение — сделайте это в другом месте своего кода.

Метод `new` непосредственно создает объект; метод `initialize` только задает значения переменных экземпляра созданного объекта.

«initialize» обходит проверку данных

Новый метод `initialize` замечателен. Он гарантирует, что полям имени и оклада в объекте работника будут присвоены **какие-то** значения. Но вы помните проверку данных в методах доступа? Метод `initialize` полностью обходит ее, и в объект могут попасть плохие данные!

@name = "Steve Wilson (HR Manager)"
@salary = 80000

Помните наш метод записи атрибута `name=`, который предотвращает присваивание пустой строки имени работника?

```
ben = Employee.new
ben.name = ""
```

Ошибка →
```
in `name=': Name can't be blank! (RuntimeError)
```

Также существует метод записи атрибута `salary=`, который следит за тем, чтобы отрицательные числа не присваивались переменной оклада:

```
kara = Employee.new
kara.salary = -246
```

Ошибка →
```
in `salary=': A salary of -246 isn't valid! (RuntimeError)
```

А теперь плохая новость: так как метод `initialize` напрямую присваивает значения переменных `@name` и `@salary`, у плохих данных появляется лазейка для проникновения в объект!

```
employee = Employee.new("", -246)
employee.print_pay_stub
```

Пустое имя в выводе! →
```
Name:
Pay This Period: $-9.44
```
← *Отрицательная сумма!*

«initialize» и проверка данных

Проверку данных в методе `initialize` *можно* организовать, добавив в метод `initialize` тот же код проверки...

```
class Employee
  ...
  def name=(name)
    if name == ""
      raise "Name can't be blank!"
    end
    @name = name
  end

  def salary=(salary)
    if salary < 0
      raise "A salary of #{salary} isn't valid!"
    end
    @salary = salary
  end

  def initialize(name = "Anonymous", salary = 0.0)
    if name == ""
      raise "Name can't be blank!"
    end
    @name = name
    if salary < 0
      raise "A salary of #{salary} isn't valid!"
    end
    @salary = salary
  end
  ...
end
```

Дублирование кода!

Дублирование кода!

Однако подобное дублирование кода создает проблемы. Что, если позднее мы изменим код проверки при инициализации, но забудем обновить метод `name=`? В классе будут действовать разные правила присваивания имени в зависимости от того, каким способом оно присваивается!

Программисты Ruby стараются соблюдать принцип *DRY* (сокращение DRY означает «Don't Repeat Yourself», то есть «не повторяйтесь»). Это означает, что программист должен по возможности избегать дублирования кода, так как оно с большой вероятностью приводит к ошибкам.

Что, если вызвать методы `name=` и `salary=` *из* метода `initialize`? Это позволит нам задать переменные экземпляра `@name` и `@salary`. При этом код проверки будет выполнен *без* дублирования!

«self» и вызов других методов для того же экземпляра

Нам необходимо вызвать методы записи атрибутов `name=` и `salary=` из метода `initialize` *того же объекта*. Это позволит выполнить код проверки методов записи до присваивания значений переменных экземпляра `@name` и `@salary`.

К сожалению, такой код работать *не будет*...

```
class Employee
  ...
  def initialize(name = "Anonymous", salary = 0.0)
    name = name
    salary = salary
  end
  ...
end

amy = Employee.new("Amy Blake", 50000)
amy.print_pay_stub
```

Не работает — Ruby считает, что происходит присваивание переменной!

@name и @salary снова равны nil!

```
Name:
in `print_pay_stub': undefined method
`/' for nil:NilClass (NoMethodError)
```

Код метода `initialize` рассматривает `name=` и `salary=` *не* как вызовы методов записи атрибутов, а как присваивание локальным переменным `name` и `salary` значений, которые в них уже хранятся! (Если вам покажется, что это бесполезно и бессмысленно, — так оно и есть.)

Необходимо однозначно сообщить Ruby, что мы хотим вызвать методы экземпляра `name=` и `salary=`. А для вызова метода экземпляра обычно применяется оператор «точка».

Но в данный момент выполняется метод экземпляра `initialize`... Что поставить слева от оператора «точка»?

Использовать переменную `amy` нельзя; глупо ссылаться на конкретный экземпляр класса из кода самого класса. Кроме того, имя `amy` внутри метода `initialize` находится за пределами области видимости.

Вне области видимости!

```
class Employee
  ...
  def initialize(name = "Anonymous", salary = 0.0)
    amy.name = name
    amy.salary = salary
  end
  ...
end

amy = Employee.new("Amy Blake", 50000)
```

Ошибка

```
in `initialize': undefined local variable or method `amy'
```

«self» и вызов других методов для того же экземпляра (продолжение)

Необходимо что-то поставить слева от оператора «точка», чтобы мы могли вызвать методы доступа к атрибутам `name=` и `salary=` класса `Employee` из метода `initialize`. Вопрос в том, что же здесь поставить? Как получить ссылку на текущий экземпляр *из* метода экземпляра?

Вне области видимости!

```
class Employee
  ...
  def initialize(name = "Anonymous", salary = 0.0)
    amy.name = name
    amy.salary = salary
  end
  ...
end

amy = Employee.new("Amy Blake", 50000)
```

У Ruby есть ответ на этот вопрос: ключевое слово `self`. В методах экземпляра `self` всегда обозначает текущий объект.

В методах экземпляра ключевое слово `self` всегда обозначает текущий объект.

Следующий простой пример демонстрирует использование `self`:

```
class MyClass
  def first_method
    puts "Current instance within first_method: #{self}"
  end
end
```

Если создать экземпляр и вызвать для него `first_method`, становится очевидно, что внутри метода экземпляра ключевое слово `self` обозначает объект, для которого вызывается метод.

```
my_object = MyClass.new
puts "my_object refers to this object: #{my_object}"
my_object.first_method
```

```
my_object refers to this object: #<MyClass:0x007f91fb0ae508>
Current instance within first_method: #<MyClass:0x007f91fb0ae508>
```

Один объект!

Строковые представления `my_object` и `self` включают уникальный идентификатор объекта. (Эта тема более подробно рассматривается в главе 8.) Идентификаторы совпадают, а значит, это один и тот же объект!

«self» и вызов других методов для того же экземпляра (продолжение)

Ключевое слово `self` также может использоваться с оператором «точка» для вызова второго метода экземпляра из первого.

```
class MyClass
  def first_method
    puts "Current instance within first_method: #{self}"
    self.second_method
  end

  def second_method
    puts "Current instance within second_method: #{self}"
  end
end

my_object = MyClass.new
my_object.first_method
```

Вызывает этот метод!

```
Current instance within first_method: #<MyClass:0x007ffd4b077510>
Current instance within second_method: #<MyClass:0x007ffd4b077510>
```

Один и тот же объект!

Научившись использовать оператор «точка» с ключевым словом `self`, мы можем четко сообщить Ruby, что код должен вызывать методы экземпляра `name=` и `salary=`, а не присваивать значения переменным `name` и `salary`...

```
class Employee
  ...
  def initialize(name = "Anonymous", salary = 0.0)
    self.name = name
    self.salary = salary
  end
  ...
end
```

ОДНОЗНАЧНО вызывается метод «name=».

ОДНОЗНАЧНО вызывается метод «salary=».

Попробуем вызвать новый конструктор и посмотрим, что из этого получится.

```
amy = Employee.new("Amy Blake", 50000)
amy.print_pay_stub
```

```
Name: Amy Blake
Pay This Period: $1917.81
```

«self» и вызов других методов для того же экземпляра (продолжение)

Полный успех! Благодаря `self` и оператору «точка» Ruby (и всем остальным) теперь абсолютно ясно, что в коде вызываются методы записи атрибутов, а не выполняется присваивание значения переменным.

А раз выполнение идет через методы доступа, это означает, что проверка данных работает без дублирования кода.

```
employee = Employee.new("", 50000)
```

Ошибка → `in `name=': Name can't be blank!`

```
employee = Employee.new("Jane Doe", -99999)
```

Ошибка → `in `salary=': A salary of -99999 isn't valid!`

Когда ключевое слово «self» не обязательно

В настоящий момент наш метод `print_pay_stub` обращается к переменным экземпляра `@name` и `@salary` напрямую:

```
class Employee

  def print_pay_stub
    puts "Name: #{@name}"
    pay_for_period = (@salary / 365.0) * 14
    formatted_pay = format("$%.2f", pay_for_period)
    puts "Pay This Period: #{formatted_pay}"
  end

end
```

Но ведь мы определили методы чтения атрибутов `name` и `salary` в классе `Employee`; вместо того чтобы обращаться к переменным экземпляра напрямую, мы могли воспользоваться ими. (Причем если в будущем метод `name` будет изменен так, чтобы сначала выводилась фамилия, или метод `salary` будет изменен для вычисления оклада по некоторому алгоритму, то код `print_pay_stub` изменять не придется.)

При вызове `name` и `salary` *можно* использовать ключевое слово `self` с оператором «точка», и такое решение будет нормально работать:

```
class Employee

  attr_reader :name, :salary

  ...

  def print_pay_stub
    puts "Name: #{self.name}"
    pay_for_period = (self.salary / 365.0) * 14
    formatted_pay = format("$%.2f", pay_for_period)
    puts "Pay This Period: #{formatted_pay}"
  end

end

Employee.new("Amy Blake", 50000).print_pay_stub
```

```
Name: Amy Blake
Pay This Period: $1917.81
```

Когда ключевое слово «self» не обязательно (продолжение)

Но в Ruby существует правило, которое позволяет немного упростить вызовы одного метода экземпляра из другого... Если вы не укажете получателя с использованием оператора «точка», то по умолчанию получателем считается текущий объект `self`.

```
class Employee
  ...
  def print_pay_stub
    puts "Name: #{name}"
    pay_for_period = (salary / 365.0) * 14
    formatted_pay = format("$%.2f", pay_for_period)
    puts "Pay This Period: #{formatted_pay}"
  end
  ...
end

Employee.new("Amy Blake", 50000).print_pay_stub
```

Ключевое слово «self» опущено; код работает!

Ключевое слово «self» опущено; код работает!

Все равно работает!

```
Name: Amy Blake
Pay This Period: $1917.81
```

Как было показано в предыдущем разделе, при вызове методов записи атрибутов включать ключевое слово `self` *обязательно, иначе* Ruby ошибочно примет = за присваивание. Но для любых других вызовов методов экземпляра `self` при желании можно не указывать.

Если вы не указали получателя при помощи оператора «точка», то по умолчанию получателем считается текущий объект `self`.

Реализация почасовой оплаты на основе наследования

Класс `Employee`, который мы создали для компании Chargemore, отлично работает! Он выводит аккуратные, правильно отформатированные платежные квитанции, а благодаря написанному нами методу `initialize` новые экземпляры `Employee` создаются проще простого.

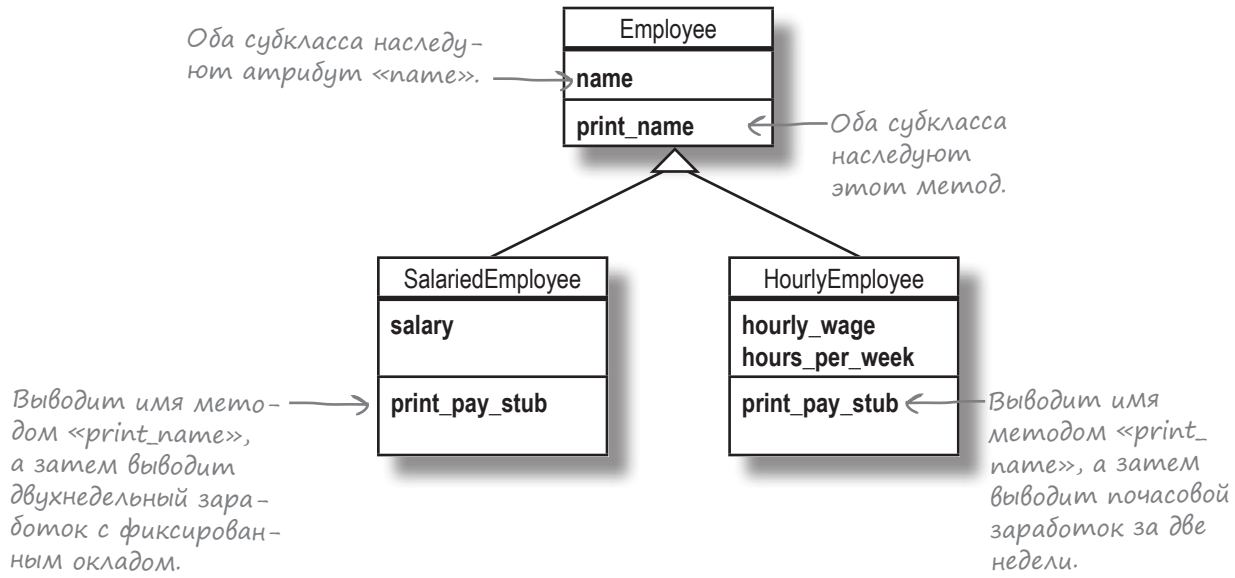
Но на данный момент класс поддерживает только штатных работников с фиксированным окладом. Пришло время добавить поддержку работников, получающих почасовую оплату.

Требования для работников с почасовой оплатой выглядят практически так же, как и для работников на окладе: требуется выводить платежные квитанции, в которых указано имя и начисленная сумма. Отличается только способ вычисления суммы. Для работников с почасовой оплатой необходимо умножить часовую ставку на количество часов, проработанных за неделю, и удвоить результат для получения двухнедельной зарплаты.

*(salary / 365.0) * 14*
Заработок штатного работника вычисляется по формуле

*часовая ставка * часов_в_неделю * 2*
Заработок работника с почасовой оплатой вычисляется по формуле

Так как функциональность работников с фиксированной и почасовой оплатой очень похожа, эту общую функциональность стоит выделить в суперкласс. Далее остается создать два субкласса с разной логикой вычисления оплаты.



Реализация почасовой оплаты на основе наследования (продолжение)

Начнем с выделения общей логики классов `SalariedEmployee` и `HourlyEmployee` в суперкласс `Employee`.

Так как платежные квитанции для работников на окладе и с почасовой оплатой должны включать их имена, атрибут `name` остается в суперклассе для наследования всеми субклассами. Код вывода имени перемещается в метод `print_name` в суперклассе.

```
class Employee

  attr_reader :name

  def name=(name)
    # Код проверки и присваивания @
name ←— Весь код методов доступа к атри-
  end     бутам опущен для краткости.

  def print_name
    puts "Name: #{name}"
  end
                    ↑
                    Помните: это то же,
                    что и вызов self.name.
end
```

Логика вычисления зарплаты для работников с фиксированным окладом перемещается в класс `SalariedEmployee`, но для вывода имени работника будет вызываться унаследованный метод `print_name`.

```
class SalariedEmployee < Employee

  attr_reader :salary

  def salary=(salary)
    # Код проверки и присваивания @salary
  end

  def print_pay_stub
    print_name ←— Вызывает метод print_name,
                 унаследованный от суперкласса.
    pay_for_period = (salary / 365.0) * 14
    formatted_pay = format("$%.2f", pay_for_period)
    puts "Pay This Period: #{formatted_pay}"
  end

end
```

Этот код не отличается от того, который использовался в старом методе print_pay_stub класса Employee.

После реализации таких изменений мы можем создать экземпляр нового класса `SalariedEmployee`, задать имя и оклад работника и вывести платежную квитанцию, как и прежде:

```
salaried_employee = SalariedEmployee.new
salaried_employee.name = "Jane Doe"
salaried_employee.salary = 50000
salaried_employee.print_pay_stub
```

```
Name: Jane Doe
Pay This Period: $1917.81
```

Реализация почасовой оплаты на основе наследования (продолжение)

А теперь мы построим новый класс `HourlyEmployee`. Он почти не отличается от `SalariedEmployee`, только в нем хранится почасовая ставка и количество рабочих часов в неделю. Эти данные используются для вычисления зарплаты за двухнедельный период. Как и в случае с классом `SalariedEmployee`, функции хранения и вывода имени работника обеспечиваются суперклассом `Employee`.

```
class HourlyEmployee < Employee

  attr_reader :hourly_wage, :hours_per_week

  def hourly_wage=(hourly_wage)
    # Код проверки и присваивания @hourly_wage
  end

  def hours_per_week=(hours_per_week)
    # Код проверки и присваивания @hours_per_week
  end

  def print_pay_stub
    print_name
    pay_for_period = hourly_wage * hours_per_week * 2
    formatted_pay = format("$%.2f", pay_for_period)
    puts "Pay This Period: #{formatted_pay}"
  end

end
```

А теперь мы можем создать экземпляр `HourlyEmployee`. Вместо оклада сохраняется почасовая ставка и количество рабочих часов в неделю, после чего на основании этих значений вычисляется заработанная сумма.

```
hourly_employee = HourlyEmployee.new
hourly_employee.name = "John Smith"
hourly_employee.hourly_wage = 14.97
hourly_employee.hours_per_week = 30
hourly_employee.print_pay_stub
```

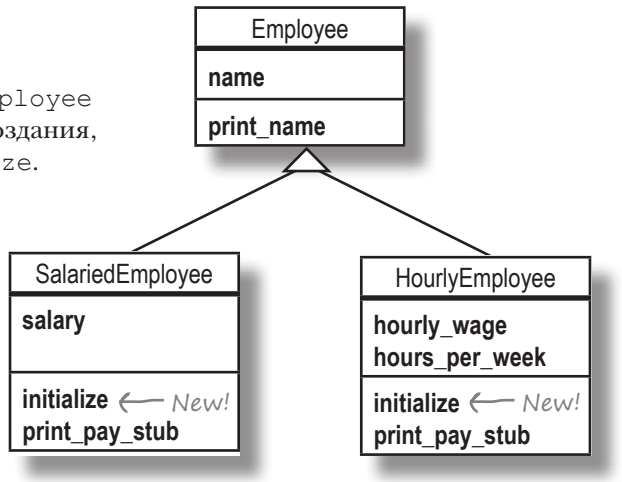
```
Name: John Smith
Pay This Period: $898.20
```

Все прошло как по маслу! Применяя наследование, мы реализовали начисление зарплаты для работников с почасовой оплатой, сохранили действующую систему расчета зарплаты для работников на окладе и свели к минимуму дублирование кода между двумя классами.

Впрочем, кое-что все же потерялось в суматохе — речь идет о методе `initialize`. Прежде данные объекта `Employee` инициализировались в момент создания, а новые классы такой возможности не дают. Метод `initialize` придется добавить повторно.

Восстановление методов «initialize»

Чтобы с объектами `SalariedEmployee` и `HourlyEmployee` можно было безопасно работать сразу же после их создания, необходимо добавить в эти классы методы `initialize`.



Как и у рассмотренного ранее класса `Employee`, метод `initialize` должен получать параметр для каждого задаваемого атрибута. Метод `initialize` класса `SalariedEmployee` выглядит так же, как в старом классе `Employee` (поскольку атрибуты не изменились), но метод `initialize` для `HourlyEmployee` получает другой набор параметров (и устанавливает другие атрибуты).

```
class SalariedEmployee < Employee
  ...
  def initialize(name = "Anonymous", salary = 0.0)
    self.name = name
    self.salary = salary
  end
  ...
end

class HourlyEmployee < Employee
  ...
  def initialize(name = "Anonymous", hourly_wage = 0.0, hours_per_week = 0.0)
    self.name = name
    self.hourly_wage = hourly_wage
    self.hours_per_week = hours_per_week
  end
  ...
end
```

Не отличается от метода initialize старого класса Employee.

И снова параметры назначаются необязательными, для чего определяются значения по умолчанию.

Метод должен получать три параметра и задавать три атрибута.

С добавлением методов `initialize` мы снова можем передавать аргументы методу `new` каждого класса. Наши объекты будут готовы к использованию сразу же после создания.

```
salaried_employee = SalariedEmployee.new("Jane Doe", 50000)
salaried_employee.print_pay_stub

hourly_employee = HourlyEmployee.new("John Smith", 14.97, 30)
hourly_employee.print_pay_stub
```

```
Name: Jane Doe
Pay This Period: $1917.81
Name: John Smith
Pay This Period: $898.20
```

Наследование и метод «initialize»

Впрочем, у новых методов `initialize` есть один недостаток: код присваивания имени работника дублируется в двух субклассах.

```
class SalariedEmployee < Employee
  ...
  def initialize(name = "Anonymous", salary = 0.0)
    self.name = name
    self.salary = salary
  end
  ...
end

class HourlyEmployee < Employee
  ...
  def initialize(name = "Anonymous", hourly_wage = 0.0, hours_per_week = 0.0)
    self.name = name <-- Дублируется в SalariedEmployee!*
    self.hourly_wage = hourly_wage
    self.hours_per_week = hours_per_week
  end
  ...
end
```

Во всех остальных аспектах наших субклассов обработка атрибута `name` поручается суперклассу `Employee`. Здесь мы определяем методы чтения и записи. Даже для вывода имени используется метод `print_name`, который вызывается субклассами из соответствующих методов `print_pay_stub`.

```
class Employee

  attr_reader :name          <-- Суперкласс содержит атрибут «name».

  def name=(name)
    # Code to validate and set @name
  end

  def print_name
    puts "Name: #{name}"     <-- Суперкласс содержит общий код для вывода имени.
  end

end
```

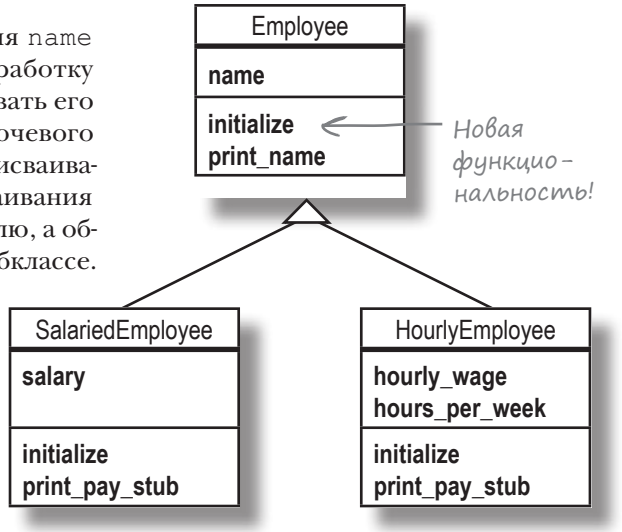
Но с `initialize` мы так не поступаем. А это возможно?

Да! Мы уже говорили это прежде и повторим еще раз: `initialize` — это *обычный метод экземпляра*. Это означает, что он наследуется, как любой другой метод, а переопределяющие методы могут вызывать его при помощи ключевого слова `super`, как любой другой метод. Эта возможность будет продемонстрирована на следующей странице.

*Да, мы понимаем, что здесь повторяется всего одна строка кода. Но прием, который мы вам покажем, позволит избежать дублирования много большего объема кода.

«super» и «initialize»

Чтобы исключить повторяющийся код присваивания `name` в субклассах `Employee`, мы можем переместить обработку имени в метод `initialize` суперкласса, а затем вызвать его из методов `initialize` в субклассах при помощи ключевого слова `super`. `SalariedEmployee` сохранит логику присваивания оклада, `HourlyEmployee` сохранит логику присваивания почасовой ставки и количества рабочих часов в неделю, а общая логика работы с именем размещается в общем субклассе.



Сначала попробуем вынести обработку `name` из метода `initialize` из `SalariedEmployee` в класс `Employee`.

```
class Employee
  ...
  def initialize(name = "Anonymous")
    self.name = name
  end
  ...
end

class SalariedEmployee < Employee
  ...
  def initialize(name = "Anonymous", salary = 0.0)
    super
    self.salary = salary
  end
  ...
end
```

Новый метод «initialize», который работает только с name!

Попытка вызова «initialize» в Employee для присваивания имени name.

Однако при попытке использования измененного метода `initialize` обнаруживается проблема...

```
salaried_employee = SalariedEmployee.new("Jane Doe", 50000)
salaried_employee.print_pay_stub
```

Ошибка

```
in `initialize': wrong number of arguments (2 for 0..1)
```

«super» и «initialize» (продолжение)

Ой! Мы забыли важную особенность метода `super`, о которой упоминалось ранее, — если ему не передается набор аргументов, то он вызывает метод суперкласса с набором аргументов, полученным методом субкласса. (Это относится и к использованию `super` в других методах экземпляров, и к использованию `super` в `initialize`.) Метод `initialize` в `SalariedEmployee` получает *два* параметра, и `super` передает *оба* методу `initialize` из класса `Employee`. (Хотя метод получает только *один* аргумент.)

Таким образом, проблема легко решается: достаточно указать, какой параметр должен передаваться при вызове — параметр `name`.

```
class SalariedEmployee < Employee
  ...
  def initialize(name = "Anonymous", salary = 0.0)
    super(name)
    self.salary = salary
  end
  ...
end
```

Вызываем «initialize» из Employee, передавая только name.

Пробуем снова инициализировать только что созданный объект `SalariedEmployee`...

```
salaried_employee = SalariedEmployee.new("Jane Doe", 50000)
salaried_employee.print_pay_stub
```

```
Name: Jane Doe
Pay This Period: $1917.81
```

Получилось! Внесем те же изменения в классе `HourlyEmployee`...

```
class HourlyEmployee < Employee
  ...
  def initialize(name = "Anonymous", hourly_wage = 0.0, hours_per_week = 0.0)
    super(name)
    self.hourly_wage = hourly_wage
    self.hours_per_week = hours_per_week
  end
  ...
end

hourly_employee = HourlyEmployee.new("John Smith", 14.97, 30)
hourly_employee.print_pay_stub
```

Вызываем «initialize» из класса Employee, передавая только name.

```
Name: John Smith
Pay This Period: $898.20
```

Ранее мы использовали `super` в методах `print_pay_stub`, `SalariedEmployee` и `HourlyEmployee` для того, чтобы поручить вывод имени работника суперклассу `Employee`. Сейчас мы просто проделали то же самое с методом `initialize`, поручив суперклассу присваивание атрибута `name`.

Почему такое решение работает? Потому что `initialize` — такой же метод экземпляра, как и любой другой. Любая возможность Ruby, которая может использоваться с обычным методом экземпляра, может использоваться и с `initialize`.

Часто Задаваемые Вопросы

В: Если я переопределяю `initialize` в субклассе, то будет ли выполняться метод `initialize` суперкласса при выполнении переопределенного метода `initialize`?

О: Нет, если вы не вызовете его явно при помощи ключевого слова `super`. Помните: в Ruby `initialize` — всего лишь обычный метод, как и любой другой. Если вы вызываете метод `move` для экземпляра `Dog`, будет ли автоматически вызываться метод `move` из класса `Animal`? Нет, если вы не используете `super`. С методом `initialize` дело обстоит точно так же.

Ruby в этом отношении отличается от многих других объектно-ориентированных языков, в которых при вызове конструктора субкласса автоматически вызывается конструктор суперкласса.

В: Если я явно использую `super` для вызова метода `initialize` суперкласса, должно ли это быть первой командой в методе `initialize` субкласса?

О: Если ваш субкласс зависит от переменных экземпляра, которые задаются методом `initialize` суперкласса, `super` следует вызывать до выполнения любых других операций. Но Ruby этого не требует. Как и других методах, вы можете вызывать `super` в любой точке `initialize` по своему желанию.

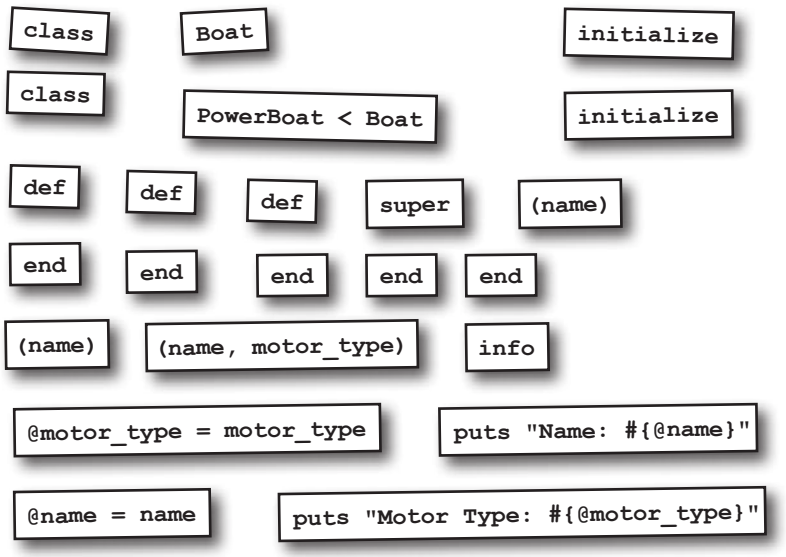
В: Вы *утверждаете*, что метод `initialize` суперкласса не выполняется до вызова `super`... Если это правда, то почему задается значение `@last_name` в этом примере?

```
class Parent
  attr_accessor :last_name
  def initialize(last_name)
    @last_name = last_name
  end
end

class Child < Parent
end

child = Child.new("Smith")
puts child.last_name
```

О: Потому что `initialize` *наследуется* от класса `Parent`. В методах экземпляра Ruby использовать `super` для вызова метода родительского класса обязательно только в том случае, *если* вы хотите выполнить этот метод *и* он был переопределен в субклассе. Если метод не переопределялся, то унаследованный метод выполняется напрямую. К `initialize` это относится в той же мере, что и к любому другому методу.

Развлечения с магнитами

В программе на языке Ruby, выложенной на холодильнике, часть магнитов перепуталась. Сможете ли вы расставить фрагменты кода по местам и получить полноценный суперкласс и субкласс, с которыми для приведенного примера выводился бы заданный результат?



Пример кода:

```
boat = PowerBoat.new("Guppy", "outboard")
boat.info
```

Результат:

```
File Edit Window Help
Name: Guppy
Motor Type: outboard
```

Развлечения с магнитами. Решение

В программе на языке Ruby, выложенной на холодильнике, часть магнитов перепуталась. Сможете ли вы расставить фрагменты кода по местам и получить полноценный суперкласс и субкласс, с которыми для приведенного примера выводился бы заданный результат?

```
class Boat

  def initialize(name)
    @name = name
  end

end
```

```
class PowerBoat < Boat

  def initialize(name, motor_type)
    super(name)
    @motor_type = motor_type
  end

  def info
    puts "Name: #{@name}"
    puts "Motor Type: #{@motor_type}"
  end

end
```

Пример кода:

```
boat = PowerBoat.new("Guppy", "outboard")
boat.info
```

Результат:

```
File Edit Window Help
Name: Guppy
Motor Type: outboard
```

Тот же класс, те же значения атрибутов

Класс `HourlyEmployee` готов, и компания Chargemore приступает к срочному найму персонала для расширения сети. Ниже перечислены вакансии работников для первого магазина:

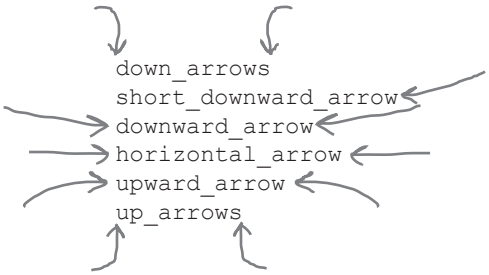
Пробелы в коде Ruby игнорируются, поэтому мы выровняли код для удобства чтения.

```
ivan    = HourlyEmployee.new("Ivan Stokes",      12.75, 25)
harold  = HourlyEmployee.new("Harold Nguyen",    12.75, 25)
tamara  = HourlyEmployee.new("Tamara Wells",     12.75, 25)
susie   = HourlyEmployee.new("Susie Powell",     12.75, 25)

edwin   = HourlyEmployee.new("Edwin Burgess",    10.50, 20)
ethel   = HourlyEmployee.new("Ethel Harris",     10.50, 20)

angela  = HourlyEmployee.new("Angela Matthews", 19.25, 30)
stewart = HourlyEmployee.new("Stewart Sanchez", 19.25, 30)
```

hourly_wage *hours_per_week*

Взглянув на приведенный выше код, вероятно, вы заметите большие группы объектов, у которых методу `new` передаются похожие аргументы. И на это есть веская причина: первую группу составляют кассиры, вторую — уборщики, а третью — охранники.

В компании Chargemore все новые кассиры начинают работать на одной почасовой ставке и с одинаковым количеством рабочих часов в неделю. Уборщикам назначается другая ставка и другое количество часов в неделю, но они совпадают для всех уборщиков. То же относится и к охранникам. (Позднее каждый отдельный работник может получить прибавку, но начинают все в одинаковых условиях.)



В результате в вызовах `new` мы видим значительные повторения аргументов, а это означает повышенную вероятность опечатки. И это всего лишь первая волна найма для первого магазина Chargemore, так что в будущем ситуация только ухудшится. Нельзя ли упростить запись?

Неэффективный фабричный метод

Если вам потребовалось создать много экземпляров класса с однотипными данными, для борьбы с дублированием кода можно написать *фабричный метод* для создания объектов, заранее заполненных нужными значениями атрибутов. (Фабричные методы — паттерн программирования, который может применяться в любом объектно-ориентированном языке, а не только в Ruby.)

Но если мы будем пользоваться только теми инструментами, которыми сейчас располагаем, любой фабричный метод будет в лучшем случае неэффективным.

Чтобы продемонстрировать сказанное, попробуем написать метод для создания новых объектов `HourlyEmployee` с заданными по умолчанию атрибутами почасовой ставки и количества рабочих часов для кассиров.

```
class HourlyEmployee
  ...
  def turn_into_cashier
    self.hourly_wage = 12.75
    self.hours_per_week = 25
  end
  ...
end

ivan = HourlyEmployee.new("Ivan Stokes")
ivan.turn_into_cashier
ivan.print_pay_stub
```

Назначение почасовой ставки.

Назначение количества часов в неделю.

```
Name: Ivan Stokes
Pay This Period: $637.50
```

Да, такое решение работает. Чем же оно так неэффективно? Присмотритесь повнимательнее к методу `initialize` (который, конечно, должен выполняться при создании нового объекта `HourlyEmployee`).

```
class HourlyEmployee
  ...
  def initialize(name = "Anonymous", hourly_wage = 0, hours_per_week = 0)
    super(name)
    self.hourly_wage = hourly_wage
    self.hours_per_week = hours_per_week
  end
  ...
end
```

Назначение почасовой ставки.

Назначение количества часов в неделю.

Мы присваиваем атрибуты `hourly_wage` и `hours_per_week` в методе `initialize`, а затем немедленно *снова* присваиваем их в `turn_into_cashier`!

Повторное присваивание неэффективно с точки зрения Ruby, но оно создает риск и для нас, программистов. Что, если для `hourly_wage` и `hours_per_week` в `initialize` параметры по умолчанию не определяются? Тогда *нам* придется задавать аргументы, которые будут попросту отброшены.

```
ivan = HourlyEmployee.new("Ivan Stokes", 0, 0)
ivan.turn_into_cashier
```

Ни одно из этих значений не используется!

И в этом проявляется проблема с записью фабричных методов в форме методов экземпляра: мы пытаемся *создать* новый экземпляр класса, но для выполнения методов уже *должен* существовать экземпляр! Не может быть, чтобы не существовало более эффективного решения...

К счастью, оно существует! Пора познакомиться с *методами класса*.

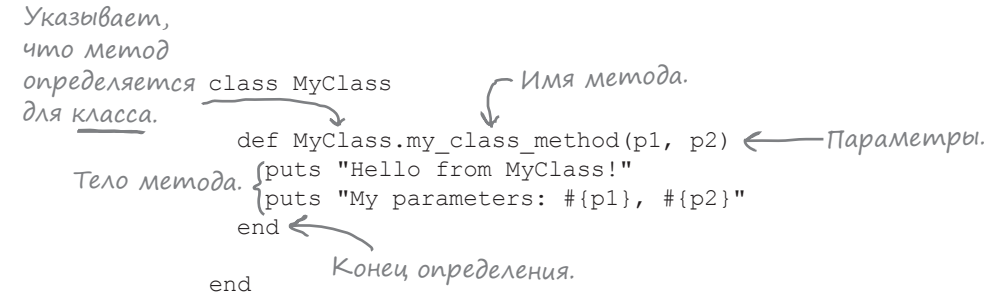
Методы класса

У нас еще *нет* экземпляра класса, но мы *хотим* его создать. И нам нужен метод, который создаст его за нас. Где разместить этот метод?

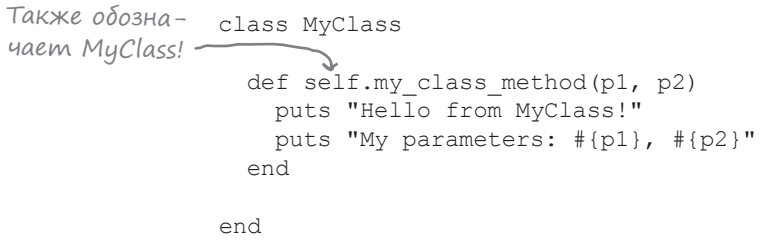
Можно выделить его в маленький файл с исходным кодом Ruby, но лучше было бы хранить его вместе с классом, экземпляры которого он создает. С другой стороны, оформить его в виде метода экземпляра этого класса не удастся. Ведь если у вас уже *есть* экземпляр класса, то вам не нужно будет его *создавать*, верно?

Для таких ситуаций в Ruby существуют **методы класса**: методы, которые могут вызываться на уровне класса, а не для какого-то отдельного экземпляра этого класса. Оформлять фабричный метод в виде метода класса *не обязательно*, но такой вариант *идеально* подходит для решаемой задачи.

Определение метода класса очень похоже на определение любого другого метода в Ruby. Различие состоит в том, что этот метод определяется *на уровне самого класса*.



В определении класса (но вне определений методов экземпляра) Ruby считает, что ключевое слово `self` относится к определяемому классу. По этой причине многие Ruby-программисты предпочитают заменять имя класса ключевым словом `self`:



Во многих отношениях определения методов класса похожи на те, которые вам уже хорошо знакомы:

- В тело метода можно включить столько команд Ruby, сколько вам захочется.
- Метод класса может возвращать значение ключевым словом `return`. Если это не сделано, то в качестве возвращаемого используется значение последнего выражения в теле метода.
- При желании вы можете определить один или несколько параметров, получаемых методом. Эти параметры можно сделать необязательными, определив для них значения по умолчанию.

Методы класса (продолжение)

Мы определили новый класс `MyClass` с одним методом класса:

```
class MyClass

  def self.my_class_method(p1, p2)
    puts "Hello from MyClass!"
    puts "My parameters: #{p1}, #{p2}"
  end

end
```

После того как метод класса будет определен, его можно напрямую вызвать для класса:

```
MyClass.my_class_method(1, 2)
```

```
Hello from MyClass!
My parameters: 1, 2
```

А может быть, этот синтаксис вызова метода класса вам покажется знакомым...

```
MyClass.new
```

Все верно, `new` является методом класса! Если задуматься, это вполне разумно; метод `new` не может быть методом *экземпляра*, потому что он вызывается для *получения* метода экземпляра. Вместо этого нужно запросить у *класса* его новый экземпляр.

Теперь, когда вы умеете создавать методы класса, попробуем написать фабричные методы для создания новых объектов `HourlyEmployee` с уже заполненными значениями почасовой ставки и количества рабочих часов в неделю. Нам понадобятся методы для присваивания заранее определенных значений ставки и рабочих часов для трех должностей: кассир, уборщик и охранник.

```
class HourlyEmployee < Employee
  ...
  def self.security_guard(name)
    HourlyEmployee.new(name, 19.25, 30)
  end

  def self.cashier(name)
    HourlyEmployee.new(name, 12.75, 25)
  end

  def self.janitor(name)
    HourlyEmployee.new(name, 10.50, 20)
  end
  ...
end
```

Имя работника передается в параметре.

Заранее определенные значения hourly_wage и hours_per_week используются для каждого типа работника.

Полученное имя используется для создания объекта работника.

То же для кассиров.

То же для уборщиков.

Имя работника заранее неизвестно, поэтому оно передается в параметре каждого из методов класса. С другой стороны, значения `hourly_wage` и `hours_per_week` для каждой должности нам *известны*. Мы передаем эти три аргумента методу `new` класса и получаем обратно новый объект `HourlyEmployee`. Затем этот новый объект возвращается методом класса.

Методы класса (продолжение)

Теперь мы можем вызывать фабричные методы напрямую для класса, передавая только имя работника.

```
angela = HourlyEmployee.security_guard("Angela Matthews")
edwin = HourlyEmployee.janitor("Edwin Burgess")
ivan = HourlyEmployee.cashier("Ivan Stokes")
```

Экземпляры `HourlyEmployee` возвращаются полностью инициализированными переданным именем и соответствующими значениями `hourly_wage` и `hours_per_week`. И мы можем немедленно перейти к выводу платежных квитанций для этих работников!

```
angela.print_pay_stub
edwin.print_pay_stub
ivan.print_pay_stub
```

```
Name: Angela Matthews
Pay This Period: $1155.00
Name: Edwin Burgess
Pay This Period: $420.00
Name: Ivan Stokes
Pay This Period: $637.50
```

В этой главе вы узнали о некоторых ловушках, подстерегающих вас на пути создания новых объектов. Но вы также освоили полезные приемы, обеспечивающие безопасность использования *ваших* объектов сразу же после их создания. При наличии хорошо спроектированных методов `initialize` и фабричных методов задачи создания и настройки новых объектов решаются проще простого!

При наличии хорошо спроектированных методов `initialize` и фабричных методов задачи создания и настройки новых объектов решаются проще простого!

Полный исходный код класса


employees.rb

```
class Employee

  attr_reader :name

  def name=(name)
    if name == ""
      raise "Name can't be blank!"
    end
    @name = name
  end

  def initialize(name = "Anonymous")
    self.name = name
  end

  def print_name
    puts "Name: #{name}"
  end

end

class SalariedEmployee < Employee

  attr_reader :salary

  def salary=(salary)
    if salary < 0
      raise "A salary of #{salary} isn't valid!"
    end
    @salary = salary
  end

  def initialize(name = "Anonymous", salary = 0.0)
    super(name)
    self.salary = salary
  end

  def print_pay_stub
    print_name
    pay_for_period = (salary / 365.0) * 14
    formatted_pay = format("$%.2f", pay_for_period)
    puts "Pay This Period: #{formatted_pay}"
  end

end
```

Атрибут «name» наследуется классами SalariedEmployee и HourlyEmployee.

Методы «initialize» классов SalariedEmployee и HourlyEmployee вызывают этот метод при помощи ключевого слова «super».

Методы «print_pay_stub» классов SalariedEmployee и HourlyEmployee вызывают этот метод.

Этим атрибутом обладают только штатные работники с фиксированным окладом.

Вызывается при вызове «SalariedEmployee.new».

Вызывает метод «initialize» суперкласса, передавая ему только имя.

Задает оклад самостоятельно, так как этим атрибутом обладает только этот класс.

Вывод имени поручается суперклассу.

Вычисление заработка за две недели.

Заработок форматируется до двух цифр в дробной части.

Продолжение на следующей странице!

employees.rb (continued)

```
class HourlyEmployee < Employee

  def self.security_guard(name)
    HourlyEmployee.new(name, 19.25, 30)
  end
  def self.cashier(name)
    HourlyEmployee.new(name, 12.75, 25)
  end
  def self.janitor(name)
    HourlyEmployee.new(name, 10.50, 20)
  end

  attr_reader :hourly_wage, :hours_per_week

  def hourly_wage=(hourly_wage)
    if hourly_wage < 0
      raise "An hourly wage of #{hourly_wage} isn't valid!"
    end
    @hourly_wage = hourly_wage
  end

  def hours_per_week=(hours_per_week)
    if hours_per_week < 0
      raise "#{hours_per_week} hours per week isn't valid!"
    end
    @hours_per_week = hours_per_week
  end

  def initialize(name = "Anonymous", hourly_wage = 0.0, hours_per_week = 0.0)
    super(name)
    self.hourly_wage = hourly_wage
    self.hours_per_week = hours_per_week
  end

  def print_pay_stub
    print_name
    pay_for_period = hourly_wage * hours_per_week * 2
    formatted_pay = format("$%.2f", pay_for_period)
    puts "Pay This Period: #{formatted_pay}"
  end

end

jane = SalariedEmployee.new("Jane Doe", 50000)
jane.print_pay_stub

angela = HourlyEmployee.security_guard("Angela Matthews")
ivan = HourlyEmployee.cashier("Ivan Stokes")
angela.print_pay_stub
ivan.print_pay_stub
```

Определение нового метода класса.

Создание нового экземпляра с заданным именем, а также заранее определенными значениями почасовой ставки и количества рабочих часов в неделю.

То же самое для других типов работников с почасовой оплатой.

Этими атрибутами обладают только работники с почасовой оплатой.

Вызывается при вызове «HourlyEmployee.new».

Вызываем метод «initialize» суперкласса, передавая ему только имя.

Задаем самостоятельно, потому что эти значения принадлежат только нашему классу.

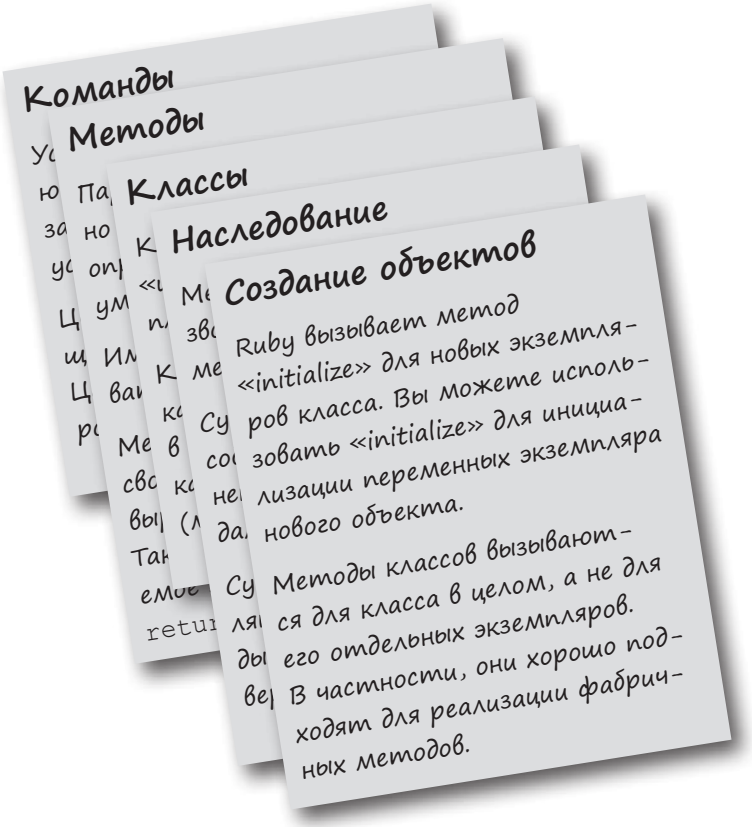
Вывод имени поручается суперклассу.

Вычисление заработка за две недели.

Заработок форматируется до двух цифр в дробной части.

Ваш инструментарий Ruby

Глава 4 осталась позади, а в вашем инструментарии появились метод `initialize` и методы класса.



Далее в программе...

До сих пор мы работали с объектами по одному. Но на практике чаще приходится работать с *группами* объектов. В следующей главе мы покажем, как создавать группы объектов в *массивах*. Также вы научитесь последовательно обрабатывать каждый объект в массиве с помощью *блоков*.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Числовые литералы, *содержащие* точку, интерпретируются как экземпляры `Float`. *Без* точки они интерпретируются как экземпляры `Fixnum`.
- Если хотя бы один из операндов математической операции является экземпляром `Float`, то результат тоже является экземпляром `Float`.
- Метод `format` использует форматные последовательности для вставки отформатированных значений в строку.
- Тип форматной последовательности определяет тип вставляемого значения: число с плавающей точкой, целое число, строка и т. д.
- Ширина форматной последовательности определяет количество символов, которые отформатированное значение должно занимать в строке.
- Значение `nil` представляет *ничто — отсутствие* значения.
- Такие операторы, как `+`, `-`, `*` и `/`, реализуются в Ruby в виде методов. Когда такой оператор встречается в коде, он преобразуется в вызов метода.
- В методах экземпляров ключевое слово `self` ссылается на экземпляр, для которого вызывается метод.
- Если при вызове метода экземпляра не указан получатель, то по умолчанию в качестве получателя используется `self`.
- В теле класса для определения метода класса используются конструкции `def ClassName.имя_метода` и `def self.имя_метода`.

Лучше, чем цикл

Очень многие задачи из области программирования связаны с обработкой списков. Списки адресов. Списки телефонных номеров. Списки продуктов. Мац, создатель языка Ruby, знал об этом. Поэтому он *основательно* потрудился над тем, чтобы работать со списками в Ruby было *действительно просто*. Сначала он поработал над тем, чтобы **массивы**, используемые для работы со списками в Ruby, обладали множеством *мощных методов* для выполнения практически любых операций. Затем он осознал, что написание кода для *перебора всех элементов списка* для выполнения некоторой операции с каждым элементом — утомительная рутинная работа, которую программистам приходится выполнять *очень часто*. Поэтому он добавил в язык **блоки**, благодаря которым код перебора стал лишним. Что же это такое — блок? Сейчас узнаете…

Массивы

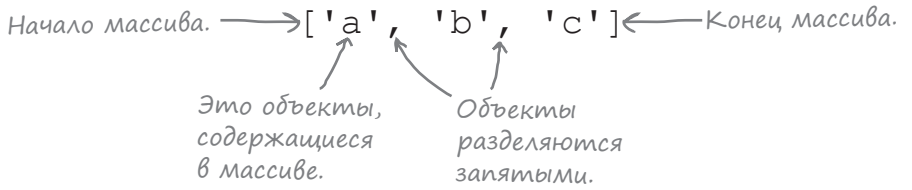
Ваш новый клиент работает над программой управления заказами для интернет-магазина. Ему нужны три разных метода, каждый из которых работает с ценами товаров, входящих в заказ. Первый метод должен просуммировать все цены для вычисления общей суммы заказа. Второй метод должен обработать снятие средств со счета клиента. Наконец, третий метод уменьшает каждую цену на 1/3 и выводит скидку.

Хмм, имеется список цен (*коллекция*, если выражаться точнее), и вы заранее не знаете, сколько их будет... Это означает, что хранить данные в переменных не удастся — ведь количество переменных неизвестно. Для хранения цен потребуется *массив*.

Массивы используются для хранения коллекций объектов. Коллекция может иметь произвольный размер. В массивах могут храниться объекты любого типа (даже другие массивы); даже допускается хранение разнотипных объектов в массиве.

В массиве хранится коллекция объектов.

Мы можем создать объект массива и инициализировать его данными с использованием *литерала* массива: заключенного в квадратные скобки (`[]`) списка значений, разделенных запятыми.



Давайте создадим массив для хранения цен из первого заказа.

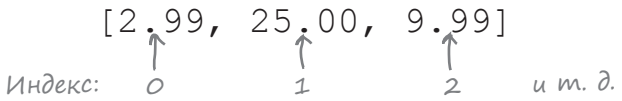
```
prices = [2.99, 25.00, 9.99]
```

Впрочем, знать все содержимое массива на момент его создания не обязательно. С массивами также можно работать и *после* их создания.

Работа с массивами

Итак, у нас появилось место для хранения цен всех товаров, входящих в заказ. Чтобы прочитать цены, хранящиеся в массиве, сначала необходимо указать, какие из них нам нужны.

Элементы массива нумеруются слева направо, начиная с `0`. Эти последовательные номера называются **индексами**.



Чтобы получить значение из массива, вы указываете целочисленный индекс нужного элемента в квадратных скобках:

```
prices[0]   # Первый элемент.
prices[1]   # Второй элемент.
prices[2]   # Третий элемент.
```

Таким образом, код вывода всех элементов массива может выглядеть примерно так:

```
puts prices[0]
puts prices[2]
puts prices[1]
```

```
3.99
25.0
8.99
```

Присваивание значения элементу с заданным индексом выполняется оператором =, как и присваивание переменной.

(Методы «p» и «inspect» пригодятся и при работе с массивами!)

```
prices[0] = 0.99
prices[1] = 1.99
prices[2] = 2.99
p prices
```

```
[0.99, 1.99, 2.99]
```

Если присвоить значение с индексом, выходящим за текущую границу массива, то массив увеличится по мере необходимости.

```
prices[3] = 3.99
p prices
```

Новый элемент.

```
[0.99, 1.99, 2.99, 3.99]
```

Если присвоить значение с индексом, выходящим *очень далеко* за границу массива, то массив все равно увеличится в соответствии с выполненным присваиванием. Все промежуточные индексы при этом просто остаются пустыми.

```
prices[6] = 6.99
p prices
```

«nil» означает «здесь ничего нет»!

Элемент, которому было присвоено значение.

```
[0.99, 1.99, 2.99, 3.99, nil, nil, 6.99]
```

Элементы с индексами, по которым еще не выполнялось присваивание, содержат значение `nil` (которое, как вы помните, обозначает отсутствие значения).

При попытке обращения к элементу, выходящему за границу массива, вы также получите `nil`.

```
p prices[7]
```

Массив расширяется только до индекса 6!

```
nil
```

Массивы тоже являются объектами!

Массивы, как и все остальное в Ruby, тоже являются объектами:

```
prices = [7.99, 25.00, 3.99, 9.99]
puts prices.class
```

```
Array
```

А это означает, что прямо с объектом массива связано множество полезных методов. Приведем несколько примеров.

Для обращения к первому и последнему элементу массива вместо индексов вида `prices[0]` можно использовать понятные и удобные методы:

```
puts prices.first
```

```
7.99
```

```
puts prices.last
```

```
9.99
```

Существуют методы для определения размера массива:

```
puts prices.length
```

```
4
```

И методы для поиска значений в массиве:

```
puts prices.include?(25.00)
```

```
true
```

```
puts prices.find_index(9.99)
```

```
3
```

Есть методы для вставки и удаления элементов, вызов которых приводит к увеличению или уменьшению массива:

```
prices.push(0.99)
p prices
```

```
[7.99, 25.0, 3.99, 9.99, 0.99]
```

```
prices.pop
p prices
```

```
[7.99, 25.0, 3.99, 9.99]
```

```
prices.shift
p prices
```

```
[25.0, 3.99, 9.99]
```

Оператор `<<` (который, как и большинство операторов, в действительности представляет собой метод) также добавляет элементы:

```
prices << 5.99
prices << 8.99
p prices
```

```
[25.0, 3.99, 9.99, 5.99, 8.99]
```

У массивов есть методы, преобразующие их в строки:

```
puts ["d", "o", "g"].join
puts ["d", "o", "g"].join("-")
```

```
dog
d-o-g
```

А у строк есть методы, преобразующие их в массивы:

```
p "d-o-g".chars
```

```
["d", "-", "o", "-", "g"]
```

```
p "d-o-g".split("-")
```

```
["d", "o", "g"]
```

Откройте терминальное окно, введите команду **`irb`** и нажмите клавишу Enter/Return. Под каждым из приведенных ниже выражений Ruby запишите, какой результат, по вашему мнению, будет получен при его выполнении. Потом введите выражение в irb и нажмите Enter. Совпадет ли ваше предположение с тем, что выдаст irb?

```
mix = ["one", 2, "three", Time.new]
```

.....

```
mix.length
```

.....

```
mix[0]
```

.....

```
mix[1]
```

.....

```
mix[0].capitalize
```

.....

```
mix[1].capitalize
```

.....

```
letters = ["b", "c", "b", "a"]
```

.....

```
letters.shift
```

.....

```
letters
```

.....

```
letters.join("/")
```

.....

```
letters.pop
```

.....

```
letters
```

.....

Упражнение
Решение

Откройте терминальное окно, введите команду **`irb`** и нажмите клавишу Enter/Return. Под каждым из приведенных ниже выражений Ruby запишите, какой результат, по вашему мнению, будет получен при его выполнении. Потом введите выражение в irb и нажмите Enter. Совпадет ли ваше предположение с тем, что выдаст irb?

```
mix = ["one", 2, "three", Time.new]
```

["one", 2, "three", 2014-01-01 11:11:11]

В одном массиве могут храниться экземпляры разных классов!

```
mix.length
```

4

```
mix[0]
```

"one"

```
mix[1]
```

2

```
mix[0].capitalize
```

Методы можно вызывать непосредственно для объектов, прочитанных из массива.

"One"

```
mix[1].capitalize
```

Ошибка: метод «capitalize» не определен для 2: FixNum

Если в массиве хранятся объекты разных классов, будьте внимательны с тем, какие методы вы для них вызываете!

```
letters = ["b", "c", "b", "a"]
```

["b", "c", "b", "a"]

```
letters.shift
```

"b" ← *«shift» извлекает первый элемент из массива и возвращает его.*

```
letters
```

«shift» изменяет содержимое массива.

["c", "b", "a"]

```
letters.join("/")
```

"c/b/a"

```
letters.pop
```

"a" ← *«pop» извлекает последний элемент из массива и возвращает его.*

```
letters
```

«pop» также изменяет содержимое массива.

["c", "b"]

Перебор элементов массива

Пока мы умеем только обращаться к элементам с конкретными индексами, заданными в коде. Чтобы просто вывести все цены из массива, вам пришлось бы использовать следующую запись:

```
prices = [3.99, 25.00, 8.99]

puts prices[0]
puts prices[1]
puts prices[2]
```

Первый элемент.
Второй элемент.
Третий элемент.

Такой способ не подходит для очень больших массивов, а также в том случае, если размер массива неизвестен заранее.

Однако вы можете воспользоваться циклом `while` для того, чтобы последовательно обработать *все* элементы в массиве.

```
index = 0
while index < prices.length
  puts prices[index]
  index += 1
end
```

Начинаем с индекса 0.
Перебираем элементы, пока не будет достигнут конец массива.
Обращение к элементу с текущим индексом.
Переход к следующему элементу массива.

```
3.99
25.0
8.99
```

Будьте осторожны!

Вызывая для массива метод экземпляра `length`, вы получаете количество элементов, хранящихся в массиве, а не индекс последнего элемента.

Следовательно, этот вызов не вернет последний элемент массива:

```
p prices[prices.length]
```

```
nil
```

Выражение должно выглядеть так:

```
p prices[prices.length - 1]
```

```
8.99
```

Аналогичным образом следующий цикл выйдет за границу массива:

```
index = 0
while index <= prices.length
  puts prices[index]
  index += 1
end
```

Индекс не должен быть равен length!

*Так как **индексы начинаются с нуля**, цикл должен работать с индексами, **меньшими** `prices.length`:*

```
index = 0
while index < prices.length
  puts prices[index]
  index += 1
end
```

Индексы должны быть МЕНЬШЕ length.

Снова и снова

Итак, теперь вы знаете, как сохранить цены из заказа в массиве и как использовать цикл `while` для обработки каждой из цен, входящих в заказ, и мы можем перейти к работе над тремя методами, необходимыми вашему клиенту:

В первом пункте необходимо взять элементы списка и просуммировать их. Мы создадим метод, который ведет накапливаемую сумму элементов в массиве. Цикл последовательно берет каждый элемент массива и прибавляет его к сумме (которая будет храниться в переменной). После обработки всех элементов метод возвращает сумму.

```
def total(prices)
  amount = 0
  index = 0
  while index < prices.length
    amount += prices[index]
    index += 1
  end
  amount
end

prices = [3.99, 25.00, 8.99]

puts format("%.2f", total(prices))
```

```
37.98
```

Сумма начинается с 0.

Перебор начинается с первого индекса в массиве.

Пока цикл не вышел за границу массива...

Текущая цена прибавляется к накапливаемой сумме.

Переход к следующей цене.

Возвращение суммы.

Создание массива с ценами, входящими в заказ.

Передача массива цен методу и форматирование результата.

Обеспечивает вывод правильного количества знаков в дробной части.

Снова и снова (продолжение)

Второй метод должен осуществлять списание средств. Он перебирает все элементы массива и вычисляет сумму для списания с текущего баланса на счету клиента.

```
def refund(prices)
  amount = 0
  index = 0
  while index < prices.length
    amount -= prices[index]
    index += 1
  end
  amount
end

puts format("%.2f", refund(prices))
```

Начинаем с 0.
Начинаем с первого индекса в массиве.
Пока цикл не вышел за границу массива...
Вычесть текущую цену.
Переход к следующей цене.
Вернуть общую сумму списания.
Передать массив цен методу и отформатировать результат.

```
-37.98
```

Наконец, третий метод должен уменьшать цену каждого элемента на 1/3 и выводить экономию.

```
def show_discounts(prices)
  index = 0
  while index < prices.length
    amount_off = prices[index] / 3.0
    puts format("Your discount: $%.2f", amount_off)
    index += 1
  end
end

show_discounts(prices)
```

Начинаем с первого индекса в массиве.
Пока цикл не вышел за границу массива...
Определение скидки для текущей позиции.
Форматирование скидки.
Переход к следующей цене.
Передача массива цен методу.

```
Your discount: $1.33
Your discount: $8.33
Your discount: $3.00
```

Все не так страшно! Перебор элементов в массиве позволяет реализовать все три метода, необходимых вашему клиенту!

- [x] Задан массив цен. Просуммировать все элементы и вернуть сумму.
- [x] Задан массив цен. Просуммировать все элементы и вернуть сумму.
- [x] Задан массив цен. Уменьшить каждую цену на 1/3 и вывести размер скидки.

Снова и снова (продолжение)

Но если внимательно присмотреться к коду всех трех методов, вы заметите, что методы содержат *большое количество* дублирующегося кода. И похоже, это как-то связано с задачей перебора в массиве цен. В следующем листинге повторяющиеся строки выделены темным фоном.

Выделенные строки повторяются в трех методах.

Впрочем, строка в середине отличается...

Отличается...

Отличается...

```
def total(prices)
  amount = 0
  index = 0
  while index < prices.length
    amount += prices[index]
    index += 1
  end
  amount
end

def refund(prices)
  amount = 0
  index = 0
  while index < prices.length
    amount -= prices[index]
    index += 1
  end
  amount
end

def show_discounts(prices)
  index = 0
  while index < prices.length
    amount_off = prices[index] / 3.0
    puts format("Your discount: $%.2f", amount_off)
    index += 1
  end
end
```

Перед нами явное нарушение правила DRY (Don't Repeat Yourself, то есть «Не повторяйтесь»). Необходимо снова заняться проектированием и переработать эти методы.

Переработка

- ☐ Задан массив цен. Просуммировать все элементы и вернуть сумму.
- ☐ Задан массив цен. Вычесть каждую цену из баланса на счету клиента.
- ☐ Задан массив цен. Уменьшить каждую цену на 1/3 и вывести размер скидки.

Устранение дубликатов — НЕПРАВИЛЬНЫЙ способ

Методы `total`, `refund` и `show_discounts` содержат довольно большой объем дублирующегося кода, связанного с перебором элементов массива. Было бы неплохо вынести повторяющийся код в отдельный метод и вызывать его из `total`, `refund` и `show_discounts`.

Но метод, объединяющий *всю* логику из `total`, `refund` и `show_variables`, получится *слишком* громоздким... Конечно, код *самого* цикла повторяется, но код в *середине* цикла различается. Кроме того, методам `total` и `refund` необходима переменная для хранения накопленной суммы, а методу `show_discounts` она не нужна.

Давайте посмотрим, *насколько* жутко может выглядеть такой метод. (Мы хотим, чтобы вы в полной мере оценили его, прежде чем приводить более эффективное решение.) Попробуем написать метод с дополнительным параметром `operation`. В зависимости от значения из `operation` будут выбираться используемые переменные и код, выполняемый в середине цикла.

```
def do_something_with_every_item(array, operation)

  if operation == "total" or operation == "refund"
    amount = 0
  end
  index = 0
  while index < array.length

    if operation == "total"
      amount += array[index]
    elsif operation == "refund"
      amount -= array[index]
    elsif operation == "show discounts"
      amount_off = array[index] / 3.0
      puts format("Your discount: $%.2f", amount_off)
    end

    index += 1
  end

  if operation == "total" or operation == "refund"
    return amount
  end

end
```

Переменной «operation» присваивается одно из значений: «total», «refund» или «show discounts». Смотрите, не ошибитесь!

Эта переменная не понадобится для операции «show discounts».

Начало цикла — теперь никакого дублирования!

Выбор правильной логики для текущей операции.

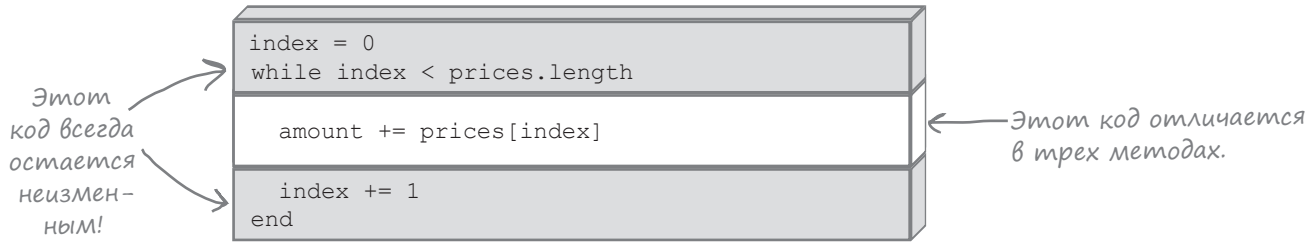
Для варианта «show discounts» это значение возвращаться не должно.

Мы же предупреждали, что все будет плохо... Повсюду разбросаны `if`, проверяющие значение параметра `operation`. Есть переменная `amount`, которая в одних случаях используется, а в других — нет. В одних случаях значение возвращается, а в других — нет. Код получается уродливым, и при его вызове слишком легко ошибиться.

Но если так писать код *не стоит*, как присвоить значения необходимым переменным перед началом цикла? И как выполнить код, необходимый *в середине* цикла?

Фрагменты кода?

Проблема в том, что повторяющийся код в начале и в конце каждого метода *окружает* код, который должен изменяться.

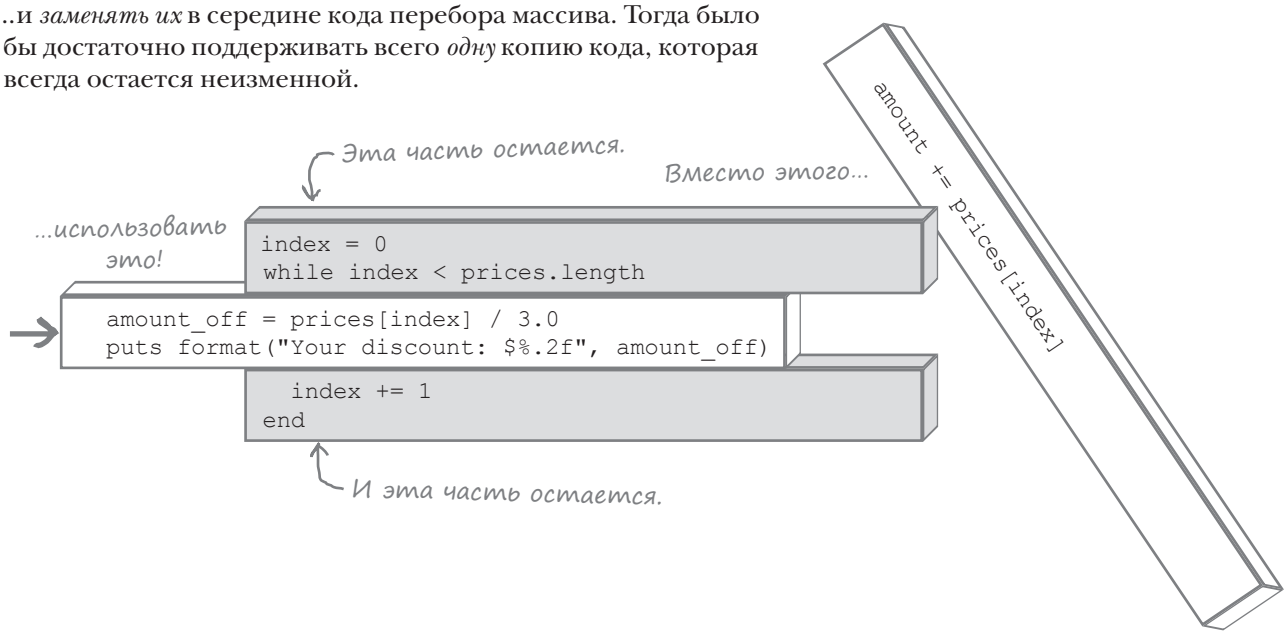


Было бы удобно, если бы мы могли взять фрагменты кода, которые могут *изменяться*...

```
amount -= prices[index]
```

```
amount_off = prices[index] / 3.0
puts format("Your discount: $%.2f", amount_off)
```

...и *заменять их* в середине кода перебора массива. Тогда было бы достаточно поддерживать всего *одну* копию кода, которая всегда остается неизменной.



Блоки

А нельзя ли передать методу фрагмент **кода**, словно это обычный аргумент? Тогда в начале и в конце метода выполняется стандартный код перебора элементов, а в середине выполняется переданный код!

Оказывается, такая возможность существует. Для этого нужно воспользоваться блоками Ruby.

Блок представляет собой фрагмент кода, который связывается с вызовом метода. Во время выполнения метод может *выполнить* блок один или несколько раз. *Методы и блоки совместными усилиями обеспечивают обработку ваших данных.*

Блоки — непростая тема. Будьте начеку и не расслабляйтесь!

Давайте начистоту: блоки — самая сложная часть этой книги. Даже если вы программировали на других языках, вполне возможно, что вы не видели ничего похожего на блоки. *Не отступайте*, потому что ваши труды окупятся *многократно*.

Представьте, что во всех методах, которые вам предстоит написать за вашу карьеру программиста, кто-то уже *написал половину кода за вас*. Бесплатно. *За вас* уже написали весь рутинный код в начале и в конце, а вам осталось заполнить лишь небольшой пропуск в середине. Вы должны вставить *свой* код — умный, содержательный код, который решает вашу задачу.

И если мы скажем вам, что блоки предоставят вам такую возможность, то вы будете готовы на все ради того, чтобы изучить их, верно?

Что ж, от вас кое-что потребуется: будьте терпеливы и настойчивы. Мы постараемся вам помочь. Мы по несколько раз рассмотрим каждую концепцию под разными углами. Мы приведем упражнения для закрепления полученных навыков. Обязательно *выполняйте* их, потому что упражнения помогут вам понять и запомнить, как работают блоки.

Всего несколько часов усердной работы — и вы будете получать дивиденды на протяжении всей своей карьеры программиста на Ruby, мы обещаем. Так за дело!

При вызове метода можно передать блок. Тогда метод сможет выполнять код, содержащийся в этом блоке.

Определение метода, получающего блок

Блоки и методы работают на пару. Собственно, вы *не сможете* создать блок без метода, которому он будет передаваться. Итак, начнем с определения метода, который работает с блоками.

(На этой странице вы узнаете, как использовать символ `&` для получения блока и метод `call` для выполнения этого блока. Пусть это не самый быстрый способ работы с блоками, зато он *более* наглядно показывает, что при этом происходит. А еще через несколько страниц мы представим метод `yield`, чаще применяемый на практике!)

Знакомство с темой только начинается, так что не будем усложнять. Метод будет выводить сообщение, выполнять полученный блок и выводить другое сообщение.

```
В параметре этого метода передается блок!
                ↓
def my_method(&my_block)
  puts "We're in the method, about to invoke your block!"
  my_block.call ←— Метод «call» вызывает переданный блок.
  puts "We're back in the method!"
end
```

Если перед последним параметром в определении метода стоит символ &, Ruby ожидает, что при любом вызове этого метода будет передаваться блок. Ruby получает блок, преобразует его в объект и сохраняет в этом параметре.

```
def my_method(&my_block)
  ...                    ↑
end                      При вызове этого метода передается
                         блок, который сохраняется в «my_block».
```

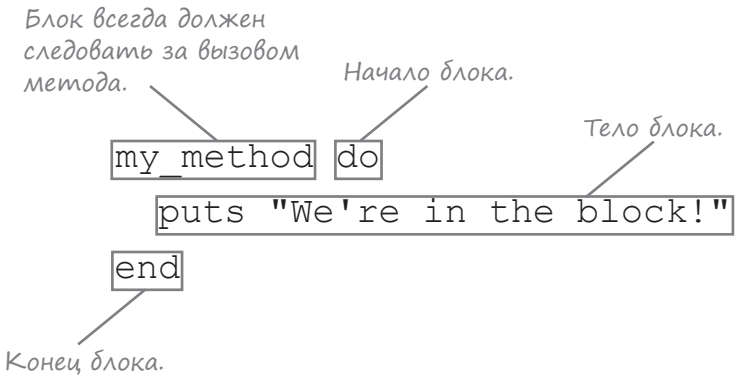
Помните: блок — всего лишь фрагмент кода, который передается методу. Для выполнения этого кода существует метод экземпляра `call`, который необходимо для него вызвать. Метод `call` обеспечивает выполнение кода блока, содержащегося в блоке.

```
                    def my_method(&my_block)
                      ...
Символа & здесь —→    my_block.call ←— Выполняет код в блоке.
нет; он используется  ...
только при определе- end
нии параметра.
```

Да, мы отлично понимаем: вы еще не *видели* ни одного блока, и вам не терпится узнать, как они выглядят. С подготовкой мы разобрались и теперь можем показать...

Ваш первый блок

Готовы? Торжественный момент наступил: вы видите свой первый блок Ruby.



Вот так! Как мы уже сказали, блок представляет собой *фрагмент кода*, который передается методу. Мы вызываем только что определенный метод `my_method` и передаем ему блок, определенный сразу же после него. Метод получит блок в параметре `my_block`.

- Начало блока обозначается ключевым словом `do`, а конец — ключевым словом `end`.
- *Тело* блока состоит из одной или нескольких строк кода Ruby, заключенных между `do` и `end`. Вы можете разместить здесь любой код по своему желанию.
- При вызове блока из метода выполняется код, содержащийся в теле блока.
- После выполнения блока управление передается методу, из которого он был вызван.

Часто Задаваемые Вопросы

В: Могу ли я использовать блок сам по себе?

О: Нет, это приведет к синтаксической ошибке. Блоки должны использоваться в сочетании с методами.

```
do
  puts "Woooo!"
end
```

```
syntax error,
unexpected
keyword_do_block
```

Проблем с этим быть не должно; если вы пишете блок, не связанный с вызовом метода, то скорее всего, то же самое можно сделать при помощи обычных команд Ruby.

Теперь мы можем вызвать метод `my_method` и передать ему приведенный выше блок:

```
def my_method(&my_block)
  puts "We're in the method, about to invoke your block!"
  my_block.call
  puts "We're back in the method!"
end

my_method do
  puts "We're in the block!"
end
```

Вызов my_method

Блок. Он будет сохранен в параметре «my_block».

...а вот результат, который вы увидите:

```
We're in the method, about to invoke your block!
We're in the block!
We're back in the method!
```

Передача управления между методом и блоком

Мы объявили метод с именем `my_method`, вызвали его с передачей блока и получили следующий результат:

```
my_method do
  puts "We're in the block!"
end
```

```
We're in the method, about to invoke your block!
We're in the block!
We're back in the method!
```

А сейчас мы подробно, шаг за шагом разберем, что же происходит между методом и блоком.

1 Выполняется первая команда `puts` в теле `my_method`.

Метод:

```
def my_method(&my_block)
  puts "We're in the method, about to invoke your block!"
  my_block.call
  puts "We're back in the method!"
end
```

Блок:

```
do
  puts "We're in the block!"
end
```

```
We're in the method, about to invoke your block!
```

2 Выполняется выражение `my_block.call`, и управление передается блоку. Выполняется выражение `puts` в теле блока.

```
def my_method(&my_block)
  puts "We're in the method, about to invoke your block!"
  my_block.call
  puts "We're back in the method!"
end
```

```
do
  puts "We're in the block!"
end
```

```
We're in the block!
```

3 Когда все команды в теле блока будут выполнены, управление возвращается в тело метода. Выполняется второй вызов `puts` в теле `my_method`, после чего метод возвращает управление.

```
def my_method(&my_block)
  puts "We're in the method, about to invoke your block!"
  my_block.call
  puts "We're back in the method!"
end
```

```
do
  puts "We're in the block!"
end
```

```
We're back in the method!
```

Вызов одного метода с разными блоками

Одному *методу* можно передать *много разных блоков*.

Передавая только что определенному методу разные блоки, мы заставляем его работать по-разному:

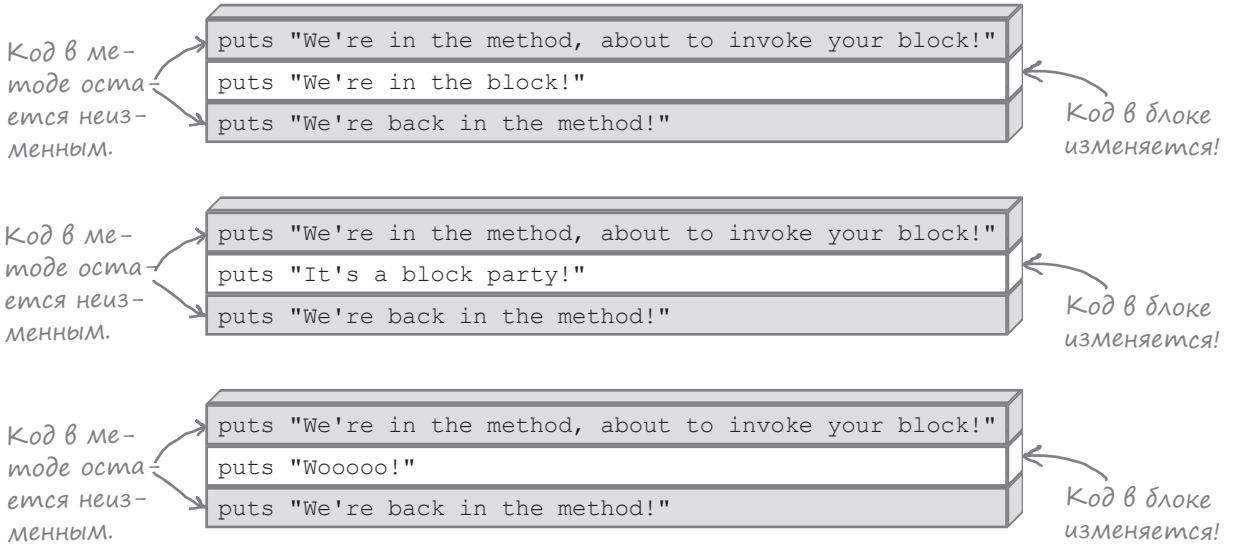
```
my_method do
  puts "It's a block party!"
end
```

```
We're in the method, about to invoke your block!
It's a block party!
We're back in the method!
```

```
my_method do
  puts "Wooooo!"
end
```

```
We're in the method, about to invoke your block!
Wooooo!
We're back in the method!
```

Код в методе всегда остается *постоянным*, однако вы можете *изменять* код, передаваемый в блоке.



Многократный вызов блока

Метод может вызвать блок столько раз, сколько посчитает нужным.

Следующий метод очень похож на предыдущий, но он содержит *два* выражения `my_block.call`:

Объявление другого метода, получающего блок.

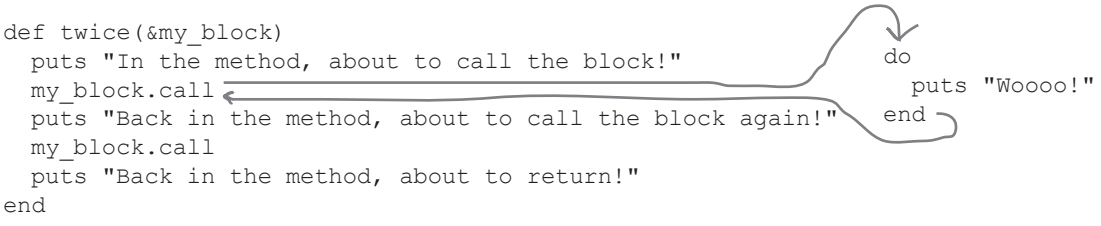
Вызов метода с передачей блока.

```
def twice(&my_block)
  puts "In the method, about to call the block!"
  my_block.call          # Вызывается блок.
  puts "Back in the method, about to call the block again!"
  my_block.call          # СНОВА вызывается блок.
  puts "Back in the method, about to return!"
end

twice do
  puts "Woooo!"
end
```

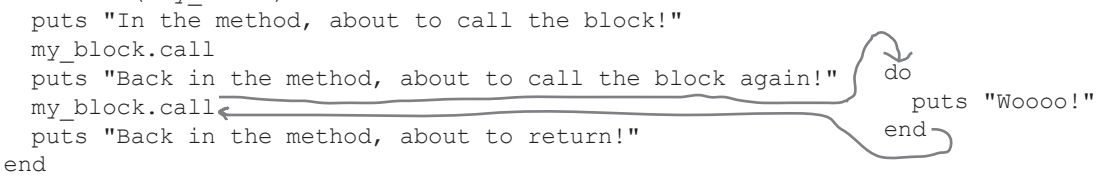
Как видно из результата, метод действительно вызывает блок дважды!

```
In the method, about to call the block!
Woooo!
Back in the method, about to call the block again!
Woooo!
Back in the method, about to return!
```

1 Команды в теле метода выполняются до того, как будет обнаружено первое выражение `my_block.call`. Выполняется блок, а после его завершения управление снова возвращается методу.



2 Тело метода продолжает выполняться. Когда будет обнаружено второе выражение `my_block.call`, блок выполняется снова. После его завершения управление возвращается методу для выполнения оставшихся команд.



Параметры блоков

В главе 2 вы узнали, что при определении метода в языке Ruby можно указать, что метод получает один или несколько параметров:

```
def print_parameters(p1, p2)
  puts p1, p2
end
```

Вероятно, вы также помните, что при вызове таких методов передаются аргументы, определяющие значение этих параметров.

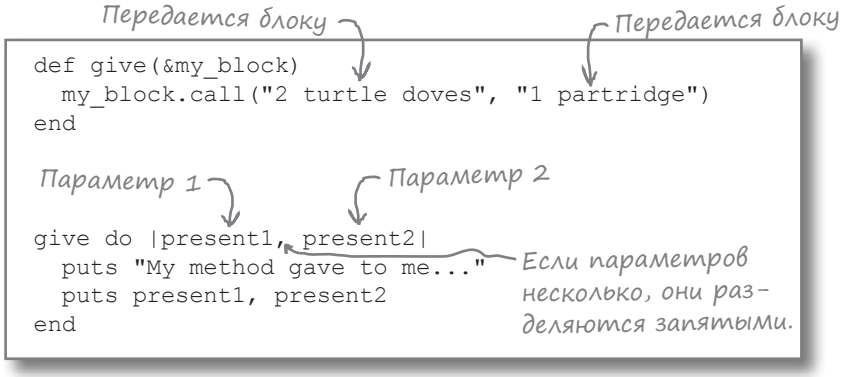
```
print_parameters("one", "two")
```

```
one
two
```

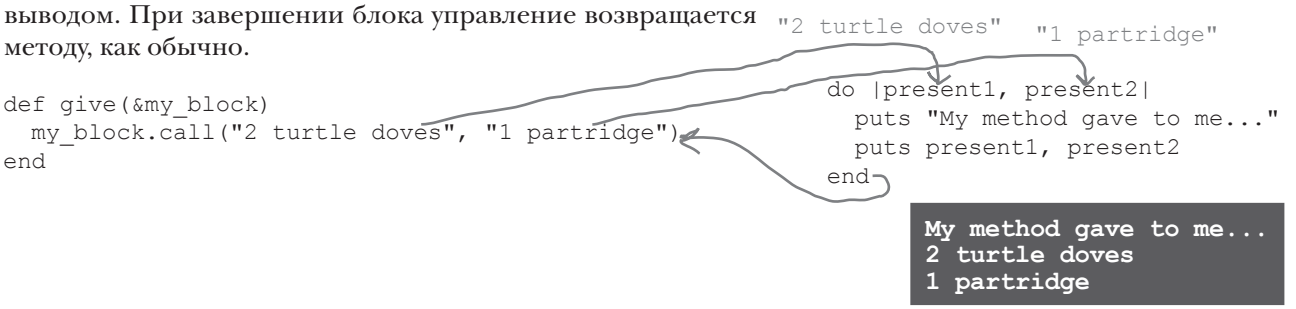
Точно так же метод может передать один или несколько аргументов блоку. Параметры блоков похожи на параметры методов; эти значения передаются при выполнении блока, и блок может обращаться к ним в своем коде.

Аргументы `call` передаются в блок:

Чтобы определить параметры, передаваемые блоку, заключите их между символами вертикальной черты (`|`) в начале блока:



Таким образом, при вызове метода с передачей блока аргументы `call` передаются блоку в параметрах с последующим выводом. При завершении блока управление возвращается методу, как обычно.



Часто Задаваемые Вопросы

В: Можно ли определить блок один раз, а затем использовать его с несколькими методами?

О: Нечто подобное в Ruby можно сделать с помощью *процедур* (эта тема выходит за рамки книги). Тем не менее, на практике так лучше не делать. Блок тесно связан с вызовом конкретного метода, так что конкретный блок обычно работает только с одним методом.

В: Может ли метод получать сразу несколько блоков?

О: Нет. Ситуация с одним блоком встречается настолько часто, что возможность передачи нескольких блоков просто не оправдает той синтаксической неразберихи, которая необходима в Ruby для ее поддержки. Если вам когда-либо потребуется сделать нечто подобное, вы также можете воспользоваться процедурами Ruby (но как уже было сказано, эта тема выходит за рамки книги).

Ключевое слово «yield»

До сих пор мы рассматривали блоки как аргументы методов. Фактически мы объявляли дополнительный параметр метода, в котором блок передавался в виде объекта, а затем вызывали метод `call` для этого объекта.

```
def twice(&my_block)
  my_block.call
  my_block.call
end
```

Впрочем, как упоминалось ранее, это не самый удобный способ передачи блоков. Сейчас мы покажем менее очевидную, но более компактную запись: ключевое слово `yield`.

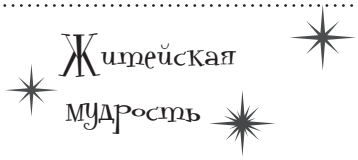
Ключевое слово `yield` находит и выполняет блок, для которого был вызван метод, — без необходимости объявлять специальный параметр для передачи блока.

Следующий метод функционально эквивалентен приведенному выше:

```
def twice
  yield
  yield
end
```

Как и в случае с `call`, при вызове `yield` также можно указать один или несколько аргументов, которые будут переданы блоку в параметрах. И снова эти два метода являются функционально эквивалентными:

```
def give(&my_block)
  my_block.call("2 turtle doves", "1 partridge")
end

def give
  yield "2 turtle doves", "1 partridge"
end
```



Объявление параметра `&block` может пригодиться в нескольких относительно редких ситуациях (выходящих за рамки темы книги.) Если вы знаете, что делает ключевое слово `yield`, в большинстве случаев следует применять именно его. С `yield` код получается более элегантным и проще читается.

Форматы блоков

Ранее для определения блоков использовался формат `do...end`. В Ruby также существует второй формат блоков: формат с «фигурными скобками». Оба формата встречаются в реальных программах, поэтому вы должны уметь распознавать оба формата.

```
def run_block
  yield
end

run_block do
  puts "do/end"
end
```

Формат do...end, использовавшийся до настоящего момента.

Формат с «фигурными скобками» →

```
run_block { puts "braces" }
```

Начало блока. *Конец блока.*

Тело блока, как и в синтаксисе с «do...end».

```
do/end
braces
```

Кроме замены `do` и `end` фигурными скобками, синтаксис и функциональность остаются неизменными.

Блоки в фигурных скобках, как и блоки в синтаксисе `do...end`, могут получать параметры:

```
def take_this
  yield "present"
end

take_this do |thing|
  puts "do/end block got #{thing}"
end

take_this { |thing| puts "braces block got #{thing}" }
```

```
do/end block got present
braces block got present
```

Кстати, а вы заметили, что наши блоки `do...end` занимают несколько строк, а блоки в фигурных скобках выводятся в одну строку? В этом проявляется еще одно общепринятое соглашение, действующее в сообществе Ruby. В принципе *синтаксис* может выглядеть и наоборот:

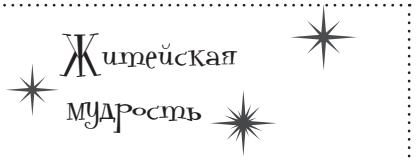
Нарушает соглашение!

```
take_this { |thing|
  puts "braces: got #{thing}"
}
take_this do |thing| puts "do/end: got #{thing}" end
```

Нарушает соглашение (и ужасно выглядит!)

```
braces: got present
do/end: got present
```

Дело даже не только в соглашениях — выглядит действительно ужасно.



Блоки Ruby, умещающиеся в одну строку, должны быть заключены в фигурные скобки. Блоки, занимающие несколько строк, должны заключаться в do...end.

Такой способ форматирования блоков не является единственно возможным, но именно он чаще всего применяется на практике.

Беседа у камина

Сегодня в студии: Метод и Блок рассуждают о том, почему они так тесно связаны друг с другом.

Метод:

Спасибо, что зашел, Блок! Я пригласил тебя сегодня, чтобы мы могли рассказать людям, как блоки сотрудничают с методами. Люди спрашивают меня, какую роль ты играешь в этих отношениях, и я думаю, что нам стоит прояснить эту тему.

Итак, основные обязанности метода вполне четко определены. Например, моя задача — перебор всех элементов в массиве.

Безусловно! Эта работа нужна очень *многим* разработчикам; мои услуги пользуются большим спросом. Но тут я сталкиваюсь с проблемой: *что делать* с каждым из этих элементов массива? Каждому разработчику нужно что-то свое! И здесь на помощь приходят блоки...

Я знаком с другим методом, который не делает ничего, кроме открытия или закрытия файла. Он *отлично* справляется с этой частью задачи. Но он *понятия не имеет*, что делать с *содержимым* файла...

Я беру на себя общую работу, которая необходима *в широком спектре* задач...

Блок:

Конечно, Метод! Ты же знаешь — я всегда откликаюсь, когда тебе понадобится моя помощь.

Верно. Может, работа не самая впечатляющая, но зато важная.

Точно. Каждый разработчик может написать *собственный* блок, который точно описывает, что именно нужно сделать с каждым элементом массива.

...и тогда он обращается за помощью к блоку, верно? И блок выводит содержимое файла, или обновляет его, или делает еще что-нибудь, что нужно разработчику. Прекрасный пример сотрудничества!

А я занимаюсь логикой, относящейся к *конкретной* задаче.

Упражнение

Перед вами определения трех методов Ruby, каждый из которых получает блок:

```
def call_block(&block)
  puts 1
  block.call
  puts 3
end
```

```
def call_twice
  puts 1
  yield
  yield
  puts 3
end
```

```
def pass_parameters_to_block
  puts 1
  yield 9, 3
  puts 3
end
```

А вот несколько примеров вызовов этих методов. Соедините каждый вызов метода с результатом, который он выведет.

(Мы решили первый пример за вас.)

B

```
call_block do
  puts 2
end
```

```
call_block { puts "two" }
```

```
call_twice { puts 2 }
```

```
call_twice do
  puts "two"
end
```

```
pass_parameters_to_block do |param1, param2|
  puts param1 + param2
end
```

```
pass_parameters_to_block do |param1, param2|
  puts param1 / param2
end
```

A
```
1
2
2
3
```

B
```
1
2
3
```

C
```
1
3
3
```

D
```
1
12
3
```

E
```
1
two
3
```

F
```
1
two
two
3
```

Упражнение
Решение

Перед вами определения трех методов Ruby, каждый из которых получает блок:

```
def call_block(&block)
  puts 1
  block.call
  puts 3
end
```

```
def call_twice
  puts 1
  yield
  yield
  puts 3
end
```

```
def pass_parameters_to_block
  puts 1
  yield 9, 3
  puts 3
end
```

А вот несколько примеров вызовов этих методов. Соедините каждый вызов метода с результатом, который он выведет.

B
```
call_block do
  puts 2
end
```

E
```
call_block { puts "two" }
```

A
```
call_twice { puts 2 }
```

F
```
call_twice do
  puts "two"
end
```

D
```
pass_parameters_to_block do |param1, param2|
  puts param1 + param2
end
```

C
```
pass_parameters_to_block do |param1, param2|
  puts param1 / param2
end
```

A
```
1
2
2
3
```

B
```
1
2
3
```

C
```
1
3
3
```

D
```
1
12
3
```

E
```
1
two
3
```

F
```
1
two
two
3
```

Метод «each»

Чтобы добраться до этого места, нам пришлось много всего узнать: как написать блок, как метод вызывает блоки, как метод передает параметры блокам. А теперь наконец-то пришло время основательно, без спешки рассмотреть метод, который помогает избавиться от повторения кода в методах `total`, `refund` и `show_discounts`. Этот метод экземпляра — он называется `each` — поддерживается всеми объектами `Array`.

Ранее уже было показано, что метод может содержать несколько вызовов `yield` с разными значениями:

```
def my_method
  yield 1
  yield 2
  yield 3
end

my_method { |param| puts param }
```

```
1
2
3
```

Метод `each` использует эту возможность для последовательного перебора всех элементов массива с передачей их блоку через `yield`.

```
["a", "b", "c"].each { |param| puts param }
```

```
a
b
c
```

Если бы нам потребовалось написать собственный метод, который работает как `each`, он получился бы очень похожим на код, который мы уже неоднократно писали:

```
class Array

  def each
    index = 0
    while index < self.length
      yield self[index]
      index += 1
    end
  end

end
```

Почти не отличается от циклов в наших методах «total», «refund» и «show_discounts»!

Вспомните: «self» обозначает текущий объект — в данном случае текущий массив.

Главное отличие: текущий элемент передается блоку!

Затем происходит переход к следующему элементу, как и прежде.

Мы последовательно перебираем все элементы массива, как и в методах `total`, `refund` и `show_discounts`. Основное отличие заключается в том, что вместо включения кода обработки текущего элемента массива *в середину цикла* ключевое слово `yield` используется для *передачи элемента блоку*.

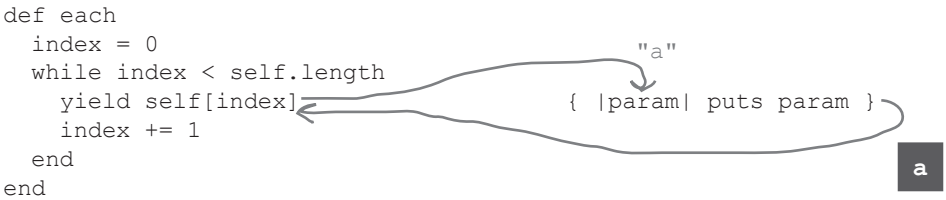
Метод «each» шаг за шагом

Метод `each` используется в сочетании с блоком для обработки каждого элемента в массиве: `["a", "b", "c"].each { |param| puts param }`

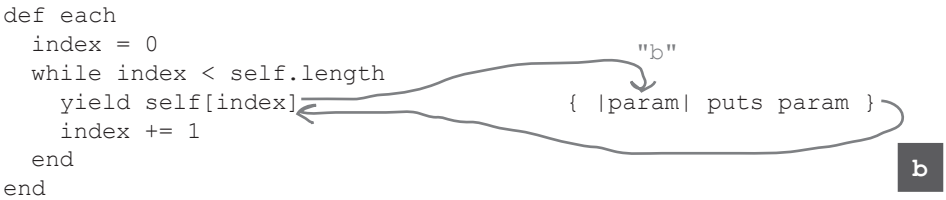
```
a
b
c
```

Давайте шаг за шагом разберем каждое из обращений к блоку и посмотрим, что происходит.

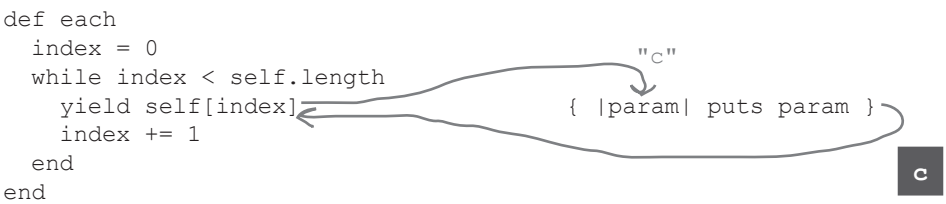
1 При первом проходе цикла `while` переменной `index` присваивается `0`, поэтому первый элемент массива передается блоку в параметре. В теле блока параметр выводится, после чего управление возвращается методу, значение `index` увеличивается, а цикл `while` продолжается.



2 Затем при втором проходе цикла `while` переменной `index` присваивается `1`, поэтому блоку в параметре будет передан *второй* элемент массива. Как и прежде, тело блока выводит параметр, после чего управление возвращается методу, и цикл продолжается.



3 После того как третий элемент массива будет передан блоку для вывода, а управление вернется методу, цикл `while` завершается, потому что мы достигли конца массива. Дальнейших итераций не будет, а значит, не будет и дальнейших обращений к блоку; работа завершена!



Вот так! Мы нашли метод, который может взять на себя функции повторяющегося кода и при этом позволяет выполнить нужный нам код в середине цикла (при помощи блока). Применим его на практике!

Устранение повторений из кода при помощи «each» и блоков

Для нашей системы ведения счетов необходимо реализовать эти три метода. Все три метода содержат почти идентичный код для перебора содержимого массива.

Впрочем, избавиться от повторения кода было достаточно сложно, потому что все три метода содержат *разный* код в *середине* цикла.

Выделенные строки повторяются во всех трех методах.

Но строка в середине не отличается...

Отличается...

Отличается...

```
def total(prices)
  amount = 0
  index = 0
  while index < prices.length
    amount += prices[index]
    index += 1
  end
  amount
end

def refund(prices)
  amount = 0
  index = 0
  while index < prices.length
    amount -= prices[index]
    index += 1
  end
  amount
end

def show_discounts(prices)
  index = 0
  while index < prices.length
    amount_off = prices[index] / 3.0
    puts format("Your discount: $%.2f", amount_off)
    index += 1
  end
end
```

Но теперь вы освоили метод `each`, который перебирает элементы массива и передает их блоку для обработки.

```
["a", "b", "c"].each { |param| puts param }
```

```
a
b
c
```

Посмотрим, нельзя ли воспользоваться `each` для переработки всех трех методов и устранения повторений.

Переработано

- ☐ Задан массив цен. Просуммировать все элементы и вернуть сумму.
- ☐ Задан массив цен. Вычесть каждую цену из баланса на счету клиента.
- ☐ Задан массив цен. Уменьшить каждую цену на 1/3 и вывести размер скидки.

Устранение повторений из кода при помощи «each» и блоков (продолжение)

Первым в очереди на переработку стоит метод `total`. Как и другие методы, он содержит код перебора цен, хранящихся в массиве. В середине этого кода `total` прибавляет текущую цену к накапливаемой сумме.

Похоже, метод `each` идеально подходит для исключения повторяющегося кода перебора! Мы можем просто взять код из середины, в котором увеличивается накапливаемая сумма, и разместить его в блоке, передаваемом `each`.

```
index = 0
while index < prices.length
  amount += prices[index]
  index += 1
end
```

Берем отсюда...

...Размещаем здесь!

```
prices.each { |price| amount += price }
```

Извлекать текущий элемент из массива больше не нужно; «each» делает это за нас!

Переопределим метод `total` с использованием `each`, а затем опробуем его.

```
def total(prices)
  amount = 0
  prices.each do |price|
    amount += price
  end
  amount
end

prices = [3.99, 25.00, 8.99]

puts format("%.2f", total(prices))
```

Суммирование начинается с 0.

Обработать каждую цену в массиве.

Прибавить текущую цену к накапливаемой сумме.

Вернуть итоговое накопленное значение.

```
37.98
```

Прекрасно! Сумма успешно вычислена. Метод `each` сработал!

Устранение повторений из кода при помощи «each» и блоков (продолжение)

Каждый элемента в массиве передается методом `each` как параметр блока. Код в блоке прибавляет текущий элемент массива к переменной `amount`, после чего управление возвращается `each`.

```
prices = [3.99, 25.00, 8.99]
puts format("%.2f", total(prices))
```

```
37.98
```

1

```
def each
  index = 0
  while index < self.length
    yield self[index]
    index += 1
  end
end
```

```
3.99
do |price|
  amount += price
end
```

2

```
def each
  index = 0
  while index < self.length
    yield self[index]
    index += 1
  end
end
```

```
25.00
do |price|
  amount += price
end
```

3

```
def each
  index = 0
  while index < self.length
    yield self[index]
    index += 1
  end
end
```

```
8.99
do |price|
  amount += price
end
```

Метод `total` успешно переработан!

Но прежде чем переходить к двум другим методам, давайте повнимательнее разберемся в том, как переменная `amount` взаимодействует с блоком.

Переработано

- [x] Задан массив цен. Просуммировать все элементы и вернуть сумму.
- [ ] Задан массив цен. Вычесть каждую цену из баланса на счету клиента.
- [ ] Задан массив цен. Уменьшить каждую цену на 1/3 и вывести размер скидки.

Блоки и область видимости переменных

Обратите внимание на один нюанс, относящийся к новому методу `total`. Вы заметили, что переменная `amount` используется как *внутри, так и снаружи* блока?

```
def total(prices)
  amount = 0
  prices.each do |price|
    amount += price
  end
  amount
end
```

Как упоминалось в главе 2, *область видимости* локальных переменных, определяемых в методе, ограничивается телом этого метода. Вы не сможете обратиться к переменным, локальным по отношению к методу, *за пределами* этого метода.

```
def my_method
  greeting = "hello"
end

my_method

puts greeting
```

Переменная определяется в методе.

Вызов метода.

Пытаемся вывести значение переменной.

Ошибка →

```
undefined local variable
or method `greeting'
```

Это правило относится и к блокам, *если* переменная впервые определяется *внутри* блока.

```
def run_block
  yield
end

run_block do
  greeting = "hello"
end

puts greeting
```

Переменная определяется внутри блока.

Пытаемся вывести значение переменной.

Ошибка →

```
undefined local variable
or method `greeting'
```

Но если переменная определяется *перед* блоком, к ней можно обращаться *внутри* тела блока. И *кроме того,* вы сможете обращаться к ней *после* завершения блока!

```
greeting = nil

run_block do
  greeting = "hello"
end

puts greeting
```

Переменная определяется ПЕРЕД блоком.

Внутри блока присваивается новое значение.

Выводим значение переменной.

```
hello
```

Блоки и область видимости переменных (продолжение)

Так как блоки Ruby могут обращаться к переменным, объявленным вне тела цикла, наш метод `total` сможет использовать `each` в блоке для обновления переменной `amount`.

Вызов `total` может выглядеть примерно так:

```
total([3.99, 25.00, 8.99])
```

Переменной `amount` присваивается `0`, после чего для массива вызывается `each`. Каждое из значений, содержащихся в массиве, передается блоку. При каждом выполнении блока происходит обновление `amount`:

```
def total(prices)
  amount = 0
  prices.each do |price|
    amount += price
  end
  amount
end
```

1

```
def each
  index = 0
  while index < self.length
    yield self[index]
    index += 1
  end
end
```

```
3.99
do |price|
  amount += price
end
```

Изменяется с 0 до 3.99.

2

```
def each
  index = 0
  while index < self.length
    yield self[index]
    index += 1
  end
end
```

```
25.00
do |price|
  amount += price
end
```

Изменяется с 3.99 на 28.99.

3

```
def each
  index = 0
  while index < self.length
    yield self[index]
    index += 1
  end
end
```

```
8.99
do |price|
  amount += price
end
```

Изменяется с 28.99 на 37.98.

При завершении метода `each` переменная `amount` сохраняет свое конечное значение, `37.98`. Именно это значение будет возвращено методом.

Часто Задаваемые Вопросы

В: Почему блоки могут обращаться к переменным, объявленным вне их тел, а методы не могут? Не создает ли это проблем с безопасностью данных?

О: Метод может вызываться в других местах программы — далеко от того места, в котором он был объявлен (возможно, даже в другом файле с исходным кодом). Напротив, блок обычно доступен только во время вызова метода, с которым он связан. Блок и доступные для него переменные и хранятся ***в одном месте*** в вашем коде. Это означает, что все переменные, с которыми работает блок, можно легко найти, а значит, обращения к ним реже приводят к неприятным сюрпризам.

Использование «each» с методом «refund»

В результате переработки из метода `total` был исключен повторяющийся код перебора элементов. Теперь то же самое нужно сделать с методами `refund` и `show_discounts`, и работа будет завершена!

Процесс обновления метода `refund` очень напоминает тот, который использовался для `total`. Специализированный код извлекается из середины обобщенного кода перебора и выносится в блок, передаваемый `each`.

```
def refund(prices)
  amount = 0
  index = 0
  while index < prices.length
    amount -= prices[index]
    index += 1
  end
  amount
end
```

Берем отсюда...

Размещаем здесь!

```
def refund(prices)
  amount = 0
  prices.each do |price|
    amount -= price
  end
  amount
end
```

Как и в предыдущем случае, извлекать элементы из массива не нужно; «each» сделает все за нас!

Получается намного аккуратнее, а вызовы метода работают точно так же, как прежде!

```
prices = [3.99, 25.00, 8.99]
puts format("%.2f", refund(prices))
```

```
-37.98
```

В вызовах `each` и блоках передача управления происходит практически так же, как в методе `total`:

❶

```
def each
  index = 0
  while index < self.length
    yield self[index]
    index += 1
  end
end
```

```
3.99
do |price|
  amount -= price
end
```

Изменяется с 0 на -3.99.

❷

```
def each
  index = 0
  while index < self.length
    yield self[index]
    index += 1
  end
end
```

```
25.00
do |price|
  amount -= price
end
```

Изменяется с -3.99 на -28.99.

❸

```
def each
  index = 0
  while index < self.length
    yield self[index]
    index += 1
  end
end
```

```
8.99
do |price|
  amount -= price
end
```

Изменяется с -28.99 на -37.98.

Использование «each» в последнем методе

Остался всего один метод! В методе `show_discounts` задача снова сводится к извлечению кода из середины цикла и его перемещению в блок, передаваемый `each`.

```
def show_discounts(prices)
  index = 0
  while index < prices.length
    amount_off = prices[index] / 3.0
    puts format("Your discount: $%.2f", amount_off)
    index += 1
  end
end
```

Берем отсюда...

Перемещаем сюда!

```
def show_discounts(prices)
  prices.each do |price|
    amount_off = price / 3.0
    puts format("Your discount: $%.2f", amount_off)
  end
end
```

И снова с точки зрения пользователя вашего метода все работает точно так же, как и прежде!

```
prices = [3.99, 25.00, 8.99]
show_discounts(prices)
```

```
Your discount: $1.33
Your discount: $8.33
Your discount: $3.00
```

А вот что происходит при передаче управления блоку:

1

```
def each
  index = 0
  while index < self.length
    yield self[index]
    index += 1
  end
end
```

3.99

```
prices.each do |price|
  amount_off = price / 3.0
  puts format("Your discount: $%.2f", amount_off)
end
```

```
Your discount: $1.33
```

2

```
def each
  index = 0
  while index < self.length
    yield self[index]
    index += 1
  end
end
```

25.00

```
prices.each do |price|
  amount_off = price / 3.0
  puts format("Your discount: $%.2f", amount_off)
end
```

```
Your discount: $8.33
```

3

```
def each
  index = 0
  while index < self.length
    yield self[index]
    index += 1
  end
end
```

8.99

```
prices.each do |price|
  amount_off = price / 3.0
  puts format("Your discount: $%.2f", amount_off)
end
```

```
Your discount: $3.00
```

Полный код методов

```
def total(prices)
  amount = 0
  prices.each do |price|
    amount += price
  end
  amount
end

def refund(prices)
  amount = 0
  prices.each do |price|
    amount -= price
  end
  amount
end

def show_discounts(prices)
  prices.each do |price|
    amount_off = price / 3.0
    puts format("Your discount: $%.2f", amount_off)
  end
end

prices = [3.99, 25.00, 8.99]

puts format("%.2f", total(prices))
puts format("%.2f", refund(prices))
show_discounts(prices)
```

Начинаем с 0.
Обрабатываем каждую цену.
Текущая цена прибавляется к сумме.
Возвращаем итоговое значение.
Начинаем с 0.
Обрабатываем каждую цену.
Вычитаем текущую цену.
Возвращаем итоговое значение.
Обрабатываем каждую цену.
Вычисляем скидку.
Форматируем и выводим текущую скидку.

Сохраните этот код в файле с именем *prices.rb*. Затем попробуйте выполнить его из терминального окна!

```
$ ruby prices.rb
37.98
-37.98
Your discount: $1.33
Your discount: $8.33
Your discount: $3.00
```

Мы избавились от повторяющегося кода перебора!

У нас получилось! Мы удалили повторяющийся код перебора элементов из своих методов. *Отличающаяся* часть кода была вынесена в блоки, а метод `each` заменил код, который остается *неизменным*.

Переработано

- ☑ Задан массив цен. Просуммировать все элементы и вернуть сумму.
- ☑ Задан массив цен. Вычесть каждую цену из баланса на счету клиента.
- ☑ Задан массив цен. Уменьшить каждую цену на 1/3 и вывести размер скидки.

У бассейна

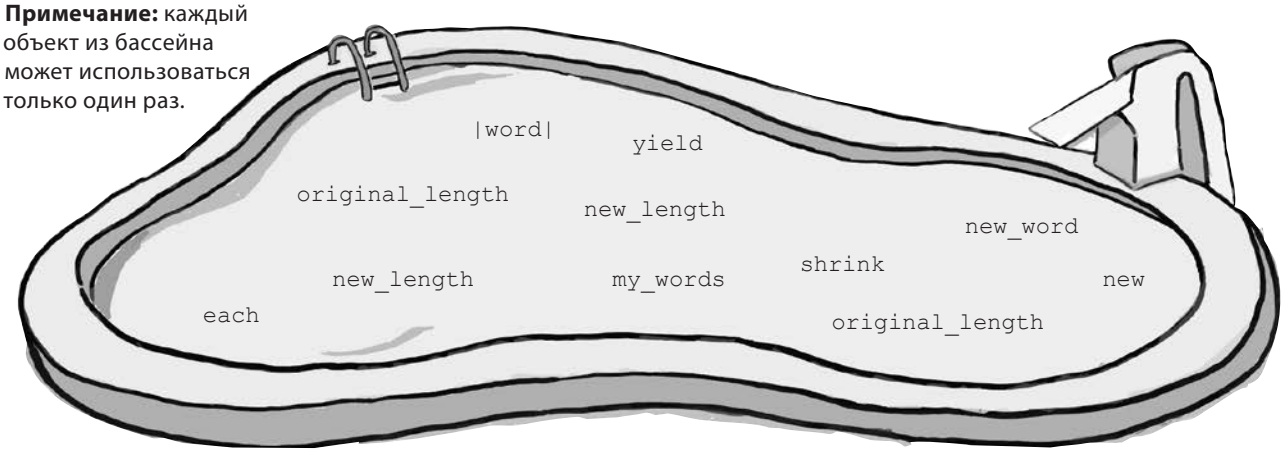
Выловите из бассейна фрагменты кода и расставьте их в пустых местах в коде. Каждый фрагмент может использоваться **только один** раз, причем использовать все фрагменты не обязательно. Ваша **задача** — составить код, который будет нормально выполняться и выдавать приведенный ниже результат.

```
def pig_latin(words)

  original_length = 0
  __________ = 0

  words.____ do _______
    puts "Original word: #{word}"
    _______________ += word.length
    letters = word.chars
    first_letter = letters.shift
    new_word = "#{letters.join}#{first_letter}ay"
    puts "Pig Latin word: #{________}"
    __________ += new_word.length
  end

  puts "Total original length: #{________________}"
  puts "Total Pig Latin length: #{new_length}"

end

my_words = ["blocks", "totally", "rock"]
pig_latin(________)
```

Результат:

```
File Edit Window Help
Original word: blocks
Pig Latin word: locksbay
Original word: totally
Pig Latin word: otallytay
Original word: rock
Pig Latin word: ockray
Original total length: 17
Total Pig Latin length: 23
```

Примечание: каждый объект из бассейна может использоваться только один раз.



У бассейна. Решение

```
def pig_latin(words)

  original_length = 0
  new_length = 0

  words.each do |word|
    puts "Original word: #{word}"
    original_length += word.length
    letters = word.chars
    first_letter = letters.shift
    new_word = "#{letters.join}#{first_letter}ay"
    puts "Pig Latin word: #{new_word}"
    new_length += new_word.length
  end

  puts "Total original length: #{original_length}"
  puts "Total Pig Latin length: #{new_length}"

end

my_words = ["blocks", "totally", "rock"]
pig_latin(my_words)
```

Результат:

```
File Edit Window Help
Original word: blocks
Pig Latin word: locksbay
Original word: totally
Pig Latin word: otallytay
Original word: rock
Pig Latin word: ockray
Original total length: 17
Total Pig Latin length: 23
```

Розетки и устройства, блоки и методы

Возьмем два разных устройства: кухонный миксер и электродрель. Они выполняют разные функции; миксер используется в кулинарии, а дрель — в столярных работах. Тем не менее у обоих устройств есть нечто общее: для работы им необходимо электричество.

Представьте, что каждый раз, когда вам потребуется воспользоваться миксером или дрелью, вам приходилось бы подключать устройство к электрической сети самостоятельно. Пожалуй, это было бы утомительно (и довольно опасно), верно?

Вот почему во время строительства вашего дома пришел электрик, который установил повсюду *розетки*. Розетка предоставляет единую функцию (электричество) через единый интерфейс (вилка электроприбора) очень разным устройствам.

Электрик не знает, как работает ваш миксер или дрель, да ему и не нужно это знать. Он просто использует свои навыки и мастерство для того, чтобы обеспечить удобный и безопасный доступ электроприбора к сети.

С другой стороны, разработчику прибора тоже не нужно знать, как дом подключен к электросети. Ему достаточно знать, как получить напряжение из розетки и использовать его для приведения устройства в действие.

Автора метода, получающего блок, можно сравнить с электриком. Он не знает, как работает блок, и его это не интересует. Он просто использует свое мастерство в конкретной области (скажем, в переборе элеметров массива) для передачи необходимых данных блоку.

```
def wire
  yield "current"
end
```

А вызов метода с блоком можно сравнить с подключением устройства к розетке. Параметры блока предоставляют безопасный, универсальный интерфейс к методу для передачи данных вашему блоку (словно розетка, поставляющая электричество устройствам). Блоку не нужно беспокоиться о том, откуда взялись данные, — он просто занимается обработкой полученных параметров.

Как при подключении к розетке.

```
wire { |power| puts "Using #{power} to turn drill bit" }
wire { |power| puts "Using #{power} to spin mixer" }
```

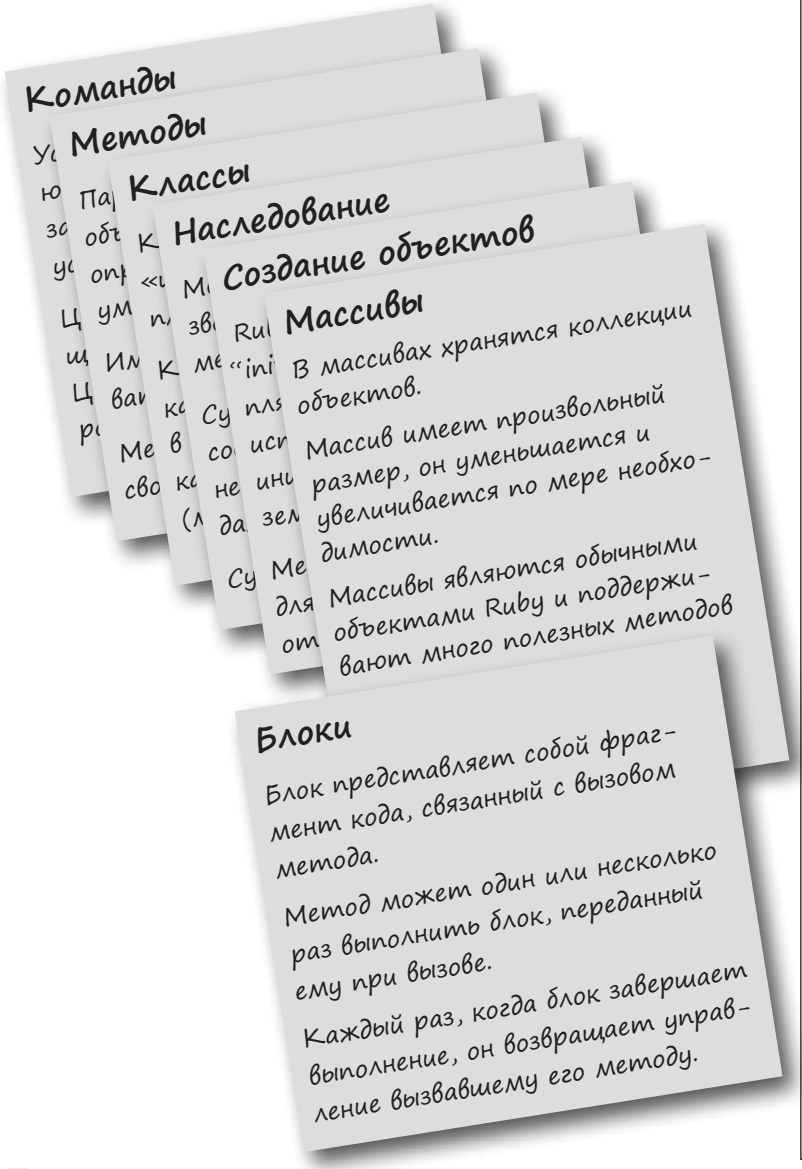
```
Using current to turn drill bit
Using current to spin mixer
```

Конечно, не все домашние устройства потребляют электричество; некоторым нужны виды ресурсов. Например, газовым плитам нужен газ, а системы пожаротушения и полива используют воду.

По аналогии с тем, как существуют разные ресурсы, потребляемые разными устройствами, в Ruby существует много разных методов, поставляющих данные блокам. С метода `each` наше знакомство с ними только началось. В следующей главе будут рассмотрены другие примеры.

Ваш инструментарий Ruby

Глава 5 осталась позади. В ней ваш инструментарий пополнился массивами и блоками.



- Индекс — число, используемое для выборки конкретного элемента из массива. Индексы в массиве начинаются с 0.
- Индекс также может использоваться для сохранения нового значения в конкретной ячейке массива.
- Метод `length` возвращает количество элементов в массиве.
- Блоки Ruby используются только в сочетании с вызовом метода.
- Существуют два варианта синтаксиса блоков: `do...end` и фигурные скобки (`{}`).
- Чтобы указать, что последний параметр метода является блоком, можно поставить перед ним символ `&`.
- На практике чаще применяется ключевое слово `yield`. Указывать параметр метода для передачи блока при этом не нужно — `yield` найдет и выполнит его за вас.
- Блок может получать от метода один или несколько параметров. Параметры блоков имеют много общего с параметрами методов.
- Блок может читать или изменять значения локальных переменных, находящихся в той же области видимости, что и блок.
- Массивы поддерживают метод `each`, который вызывает блок для каждого элемента в массиве.

Далее в программе...

Вы еще не видели всего, на что способны блоки! Блок также может *вернуть* значение методу, а методы могут использовать такие блоки сотнями нестандартных способов. За подробностями обращайтесь к следующей главе.

Как будем обрабатывать?

Вы видели лишь малую часть возможностей блоков. До настоящего момента *методы* передавали данные *блоку* и ожидали, что блок сделает все необходимое. Однако *блок* также может возвращать данные *методу*. Эта возможность позволяет методу получать *команды* от блока, позволяя ему выполнять более содержательную работу. Методы, рассмотренные в этой главе, получают *большие и сложные* коллекции и используют **возвращаемые значения блоков** для сокращения их размера.

Поиск в больших коллекциях слов

Слухи о выдающейся работе, проделанной вами для системы обработки счетов, разлетелись очень быстро, и на пороге уже появился новый клиент: киностудия. Ежегодно выпускается множество фильмов, и публиковать восторженные отзывы по ним становится все сложнее. Киностудия хочет, чтобы вы написали программу, которая проходит по тексту отзывов, находит прилагательные, описывающие фильм, и генерирует подборку таких прилагательных:

Критики единодушны —
Гинденбург:
«Романтичный»
«Волнующий»
«Сенсационный»

Вам предоставили текстовый файл для работы. Удастся ли вам создать подборку для нового фильма *Truncated*?

Впрочем, присмотревшись к файлу, вы видите, что его содержимое «нарезано» по определенному шаблону:

Текст переносится только для того, чтобы он нормально поместился на странице...

```
Строка 1  Normally producers and directors would stop this kind of
          garbage from getting published. Truncated is amazing in that
          it got past those hurdles.
Строка 2       --Joseph Goldstein, "Truncated: Awful," New York Minute
Строка 3  Guppies is destined to be the family film favorite of the
          summer.
Строка 4       --Bill Mosher, "Go see Guppies," Topeka Obscurant
Строка 5  Truncated is funny: it can't be categorized as comedy,
          romance, or horror, because none of those genres would want
          to be associated with it.
Строка 6       --Liz Smith, "Truncated Disappoints," Chicago Some-Times
Строка 7  I'm pretty sure this was shot on a mobile phone. Truncated
          is astounding in its disregard for filmmaking aesthetics.
Строка 8       --Bill Mosher, "Don't See Truncated," Topeka Obscurant
```

Подписи следует пропустить при обработке.

В файле также встречаются рецензии на другие фильмы.

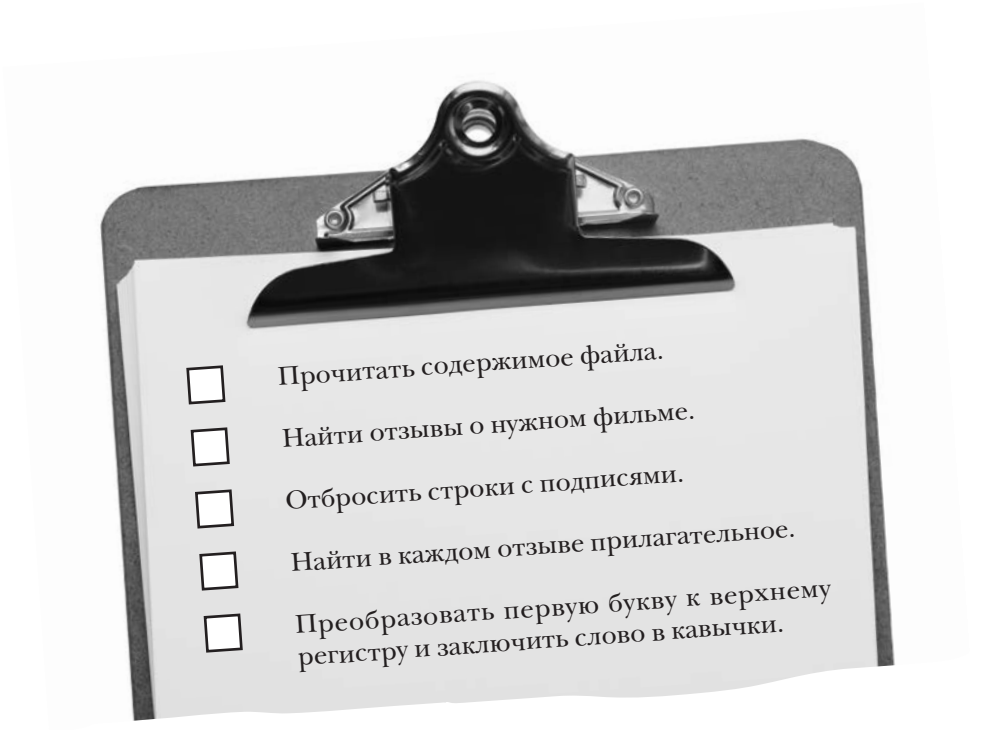
Прилагательные записаны с прописной буквы в подборке, но не в тексте.

reviews.txt

Пожалуй, задача не из простых. Но не беспокойтесь: массивы и блоки нам помогут!

Поиск в больших коллекциях слов (продолжение)

Давайте разобьем задачу по пунктам:



Получается пять задач. Вроде бы не особенно сложных. Тогда за дело!

Открытие файла

Наша первая задача — открыть текстовый файл с отзывами. Сделать это проще, чем может показаться; в Ruby существует встроенный класс с именем `File`, представляющий файл на диске. Чтобы открыть файл с именем *reviews.txt* в текущем каталоге (папке) для чтения данных, вызовите метод `open` для класса `File`:

```
review_file = File.open("reviews.txt")
```

Метод `open` возвращает новый объект `File`. (На самом деле он вызывает `File.new` за вас и возвращает полученный результат.)

```
puts review_file.class
```

```
File
```

Существует много разных методов, которые могут вызываться для этого экземпляра `File`. Для наших целей наибольший интерес представляет метод `readlines`, который возвращает все строки файла в виде массива.

(Текст переносится для размещения на странице.)

```
lines = review_file.readlines
puts "Line 4: #{lines[3]}"
puts "Line 1: #{lines[0]}"
```

```
Line 4:     --Bill Mosher, "Go see Guppies",
Topeka Obscurant
Line 1: Normally producers and directors would
stop this kind of garbage from getting published.
Truncated is amazing in that it got past those
hurdles.
```

Безопасное закрытие файла

Итак, мы открыли файл и прочитали его содержимое. Следующим шагом должно быть *закрытие* файла. Закрывая файл, вы сообщаете операционной системе: «Работа с файлом завершена, теперь он может использоваться другими».

```
review_file.close
```

Почему мы обращаем на это ваше внимание? Потому что если вы не будете закрывать свои файлы, возможны *большие неприятности*.

Вы можете получить сообщения об ошибках, если операционная система обнаружит, что у вас одновременно открыто слишком много файлов. Если вы попытаетесь прочитать все содержимое одного файла несколько раз, не закрыв его, при последующих попытках файл будет казаться пустым (потому что он уже прочитан до конца, и после текущей позиции данные отсутствуют). Если вы записываете данные в файл, никакая другая программа не увидит внесенные изменения, пока *файл не будет закрыт*. Не забывайте закрывать файлы, это *очень важно*.

Занервничали? Не стоит. Как обычно, у Ruby для таких случаев существует решение, удобное для разработчика.

Безопасное закрытие файла в блоке

Ruby предоставляет возможность открыть файл, сделать с ним все, что нужно, а потом *автоматически* закрыть его после завершения работы. Для этого следует вызвать `File.open...` с *блоком*!

Просто замените этот код:

```
review_file = File.open("reviews.txt")
lines = review_file.readlines
review_file.close
```

Вызов возвращает объект File, который необходимо сохранить в переменной.

После завершения работы с файлом необходимо вызвать «close».

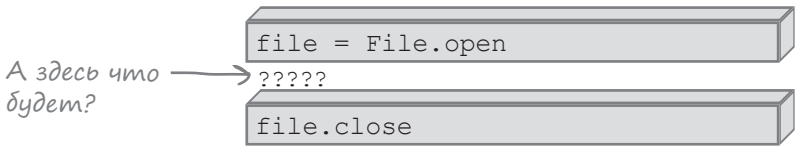
...этим кодом!

```
File.open("reviews.txt") do |review_file|
  lines = review_file.readlines
end
```

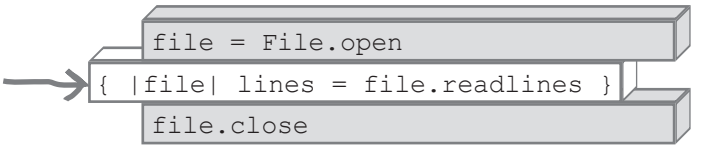
Объект File передается блоку в параметре.

При завершении блока Ruby автоматически закроет файл за вас!

Почему `File.open` использует блок для этой цели? Если подумать, первый и последний шаги процесса достаточно четко определены...



...но создатели `File.open` *понятия не имеют, что вы будете делать с файлом*, пока он остается открытым. Прочитаете по одной строке? Весь файл сразу? Вот почему они предлагают *вам* решить, что *вы* будете делать; для этого необходимо передать блок.



Не забывайте про область видимости!

Когда блок *не* используется, к массиву строк из объекта `File` можно обращаться вполне нормально.

```
review_file = File.open("reviews.txt")
lines = review_file.readlines
review_file.close

puts lines.length
```

```
8
```

Однако переход на форму `File.open` с блоком создает проблему. Массив, возвращаемый `readlines`, сохраняется в переменной *внутри* блока, но обратиться к нему *после* блока не получится.

```
File.open("reviews.txt") do |review_file|
  lines = review_file.readlines
end

puts lines.length
```

```
undefined local variable
or method `lines'
```

Проблема возникает из-за того, что переменная `lines` *создается внутри* блока. Как вы узнали в главе 5, область видимости любой переменной, созданной в блоке, ограничивается телом этого блока. Такие переменные «не видны» за пределами блока.

С другой стороны — и об этом также упоминалось в главе 5 — локальные переменные, объявленные *перед* блоком, *видны* в теле блока (и конечно, остаются видимыми после блока). Итак, проблема проще всего решается созданием переменной `lines` *до* объявления блока.

```
lines = []

File.open("reviews.txt") do |review_file|
  lines = review_file.readlines
end

puts lines.length
```

Находится в области видимости!

```
8
```

Все еще в пределах области видимости!

Хорошо, файл успешно закрыт, а мы получили содержимое файла. Что теперь с ним делать? Этой частью задачи мы займемся на следующем шаге.

Часто Задаваемые Вопросы

В: Как получается, что метод `File.open` может работать с блоком *и* без блока?

О: В методе Ruby можно вызвать метод `block_given?`, чтобы узнать, был ли использован блок при вызове, и изменить поведение метода соответствующим образом.

Если бы мы писали собственную (упрощенную) версию File.open, она могла бы выглядеть так:

```
def File.open(name, mode)
  file = File.new(name, mode)
  if block_given?
    yield(file)
  else
    return file
  end
end
```

Если блок задан, то файл передается ему для использования внутри блока. Если нет, то файл возвращается на сторону вызова.

Ниже приведены три сценария Ruby. Заполните пропуски в сценариях, чтобы они работали без ошибок и выдавали указанный результат.

1

```
def yield_number
  yield 4
end

_____________

yield_number { |number| array << number }

p array
```

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

2

```
_____

[1, 2, 3].each { |number| sum += number }

puts sum
```

```
6
```

3

```
_____________

File.open("sample.txt") do |file|
  contents = file.readlines
end

puts contents
```

```
This is the first line in the file.
This is the second.
This is the last line.
```

Упражнение
Решение

Ниже приведены три сценария Ruby. Заполните пропуски в сценариях, чтобы они работали без ошибок и выдавали указанный результат.

1

```
def yield_number
  yield 4
end

array = [1, 2, 3]

yield_number { |number| array << number }

p array
```

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

2

```
sum = 0

[1, 2, 3].each { |number| sum += number }

puts sum
```

```
6
```

3

```
contents = []
```

Здесь подойдет любое значение, потому что в блоке переменной присваивается новое значение.

```
File.open("sample.txt") do |file|
  contents = file.readlines
end

puts contents
```

```
This is the first line in the file.
This is the second.
This is the last line.
```

Поиск элементов в блоке

Мы открыли файл и воспользовались методом `readlines` для получения массива, в каждом элементе которого содержится отдельная строка из файла. Первый пункт списка выполнен!

Посмотрим, что осталось:

- [x] Прочитать содержимое файла.
- [ ] Найти отзывы о нужном фильме.
- [ ] Отбросить строки с подписями.
- [ ] Найти в каждом отзыве прилагательное.
- [ ] Преобразовать первую букву к верхнему регистру и заключить слово в кавычки.

Похоже, текстовый файл содержит отзывы не только на тот фильм, который нам нужен. К нему примешались отзывы и на другие фильмы:

```
Строка 1  Normally producers and directors would stop this kind of
          garbage from getting published. Truncated is amazing in that
          it got past those hurdles.
Строка 2      --Joseph Goldstein, "Truncated: Awful," New York Minute
Строка 3  Guppies is destined to be the family film favorite of the
          summer.
Строка 4      --Bill Mosher, "Go see Guppies," Topeka Obscurant
Строка 5  Truncated is funny: it can't be categorized as comedy,
          romance, or horror, because none of those genres would want
          to be associated with it.
Строка 6      --Liz Smith, "Truncated Disappoints," Chicago Some-Times
          ...
```

Отзыв на совершенно другой фильм!

reviews.txt

К счастью, в каждом отзыве название фильма упоминается хотя бы один раз. Мы можем воспользоваться этим фактом для поиска отзывов, относящихся к интересующему нас фильму.

```
Normally producers and directors would stop this kind of
garbage from getting published. Truncated is amazing in that it
got past those hurdles.
```

Отберем строки, содержащие название фильма.

Долгий способ поиска с использованием «each»

Чтобы узнать, содержит ли экземпляр класса `String` заданную подстроку, можно вызвать для него метод `include?` (искомая подстрока передается в аргументе). Помните, что по общепринятым соглашениям методы, имена которых завершаются символом `?`, возвращают логическое значение. Метод `include?` возвращает `true`, если строка содержит заданную подстроку, или `false` в противном случае.

```
my_string = "I like apples, bananas, and oranges"
puts my_string.include?("bananas")
puts my_string.include?("elephants")
```

```
true
false
```

Неважно, где находится искомая подстрока — в начале строки, в конце или где-то в середине; вызов `include?` найдет ее.

Итак, первый способ выбора отзывов к конкретному фильму, использующий метод `include?` и другие приемы, уже знакомые вам, выглядит так...

Наш старый код чтения содержимого файла.

```
lines = []

File.open("reviews.txt") do |review_file|
  lines = review_file.readlines
end

relevant_lines = []

lines.each do |line|
  if line.include?("Truncated")
    relevant_lines << line
  end
end

puts relevant_lines
```

Помните, что переменная должна создаваться за пределами блока!

Обработка каждой строки из файла.

Текущая строка передается блоку в параметре.

Текущая строка добавляется в массив отзывов.

Отзыв на другой фильм удален!

```
Normally producers and directors would stop this kind of
garbage from getting published. Truncated is amazing in that
it got past those hurdles.
    --Joseph Goldstein, "Truncated: Awful," New York Minute
Truncated is funny: it can't be categorized as comedy,
romance, or horror, because none of those genres would want
to be associated with it.
    --Liz Smith, "Truncated Disappoints," Chicago Some-Times
I'm pretty sure this was shot on a mobile phone. Truncated
is astounding in its disregard for filmmaking aesthetics.
    --Bill Mosher, "Don't See Truncated," Topeka Obscurant
```

А теперь короткий способ...

Оказывается, в Ruby существует куда более быстрое решение этой задачи. Метод `find_all` использует блок для проверки каждого элемента массива. Он возвращает новый массив только с теми элементами, для которых проверка вернула значение `true`.

Мы можем воспользоваться методом `find_all` для достижения того же результата, включив вызов `include?` в его блок:

```
lines = []

File.open("reviews.txt") do |review_file|
  lines = review_file.readlines
end

relevant_lines = lines.find_all { |line| line.include?("Truncated") }
```

Сокращенная версия кода работает точно так же, как предыдущая: в новый массив копируются только строки, содержащие подстроку `"Truncated"`!

```
puts relevant_lines
```

```
Normally producers and directors would stop this kind of
garbage from getting published. Truncated is amazing in that
it got past those hurdles.
    --Joseph Goldstein, "Truncated: Awful," New York Minute
Truncated is funny: it can't be categorized as comedy,
romance, or horror, because none of those genres would want
to be associated with it.
    --Liz Smith, "Truncated Disappoints," Chicago Some-Times
I'm pretty sure this was shot on a mobile phone. Truncated
is astounding in its disregard for filmmaking aesthetics.
    --Bill Mosher, "Don't See Truncated," Topeka Obscurant
```

Шесть строк кода заменяются одной строкой... Неплохо, верно?

Ой-ой! Кажется, у вас голова пошла кругом?

Мы подробно объясним все, что происходит в этой одной строке.

На нескольких ближайших страницах мы рассмотрим все, что необходимо для полного понимания того, как работает `find_all`. В Ruby есть много других методов, которые работают по тому же принципу, так что поверьте: оно того стоит!

Блоки тоже возвращают значение

Вы только что видели метод `find_all`. Ему передается блок с логикой выбора, а `find_all` находит только те элементы массива, которые соответствуют критерию из блока.

```
lines.find_all { |line| line.include?("Truncated") }
```

Под «элементами, соответствующими критерию блока» понимаются те элементы, для которых блок *возвращает* значение true. Метод `find_all` использует *возвращаемое значение блока* для определения того, какие элементы следует оставить, а какие должны быть отброшены.

Возможно, по ходу дела вы обратили внимание на некоторое сходство между блоками и методами...

Методы:

- Получают параметры
- Содержат тело с выражениями Ruby
- Возвращают значение

Блоки:

- Получают параметры
- Содержат тело с выражениями Ruby
- Возвращают значение

Стоп! Что они делают?

Да, все верно — блоки Ruby, как и методы, возвращают значение последнего вычисленного выражения! Это значение возвращается методу как результат ключевого слова `yield`.

Значение последнего выражения в блоке возвращается методу.

Ниже приведен простой метод, который демонстрирует эту возможность. Далее этот метод вызывается с разными блоками для проверки их возвращаемых значений:

```
def print_block_result
  block_result = yield
  puts block_result
end

print_block_result { 1 + 1 }

print_block_result do
  "I'm not the last expression, so I'm not the return value."
  "I'm the result!"
end

print_block_result { "I hated Truncated".include?("Truncated") }
```

Результат блока присваивается переменной.

Происходит вычисление выражения; блок возвращает 2.

Возвращается значение только последнего выражения.

Выражение вычисляется; блок возвращает true.

```
2
I'm the result!
true
```

Блоки тоже возвращают значение (продолжение)

Конечно, метод не ограничивается *выводом* возвращаемого значения блока. С этим значением можно проделать любые вычисления:

```
def triple_block_result
  puts 3 * yield
end

triple_block_result { 2 }
triple_block_result { 5 }
```

```
6
15
```

Блок возвращает 2.

Блок возвращает 5.

Или использовать его в строке:

```
def greet
  puts "Hello, #{yield}!"
end

greet { "Liz" }
```

```
Hello, Liz!
```

Блок возвращает эту строку.

Или использовать его в условной команде:

```
def alert_if_true
  if yield
    puts "Block returned true!"
  else
    puts "Block returned false."
  end
end

alert_if_true { 2 + 2 == 5 }
alert_if_true { 2 > 1 }
```

```
Block returned false.
Block returned true!
```

Блок возвращает false.

Блок возвращает true.

Далее мы подробно разберемся в том, как `find_all` использует возвращаемое значение блока для получения только тех элементов массива, которые нас интересуют.

Будьте осторожны!

Мы говорим, что блоки «возвращают значение», но это не означает, что в них нужно использовать ключевое слово `return`.

Использование ключевого слова `return` *в блоке не является синтаксической ошибкой, но мы так поступать не рекомендуем. В теле блока ключевое слово* `return` *возвращает управление не из самого блока, а из метода, в котором определяется блок. Маловероятно, что вам нужно именно это.*

```
def print_block_value
  puts yield
end

def other_method
  print_block_value { return 1 + 1 }
end

other_method
```

Этот код ничего не выводит, потому что `other_method` *немедленно завершается при определении блока.*

Если изменить блок так, чтобы последнее выражение просто использовалось как возвращаемое значение, все работает точно так, как задумано:

```
def other_method
  print_block_value { 1 + 1 }
end

other_method
```

2

Часто Задаваемые Вопросы

В: *Все* блоки возвращают значение?

О: Да! Они возвращают результат последнего выражения в теле блока.

В: Тогда почему вы не сказали об этом ранее?

О: Это было не нужно. Блок возвращает значение, но его метод *не обязан* это значение использовать. Скажем, метод `each` **игнорирует значения, возвращаемые из его блока.**

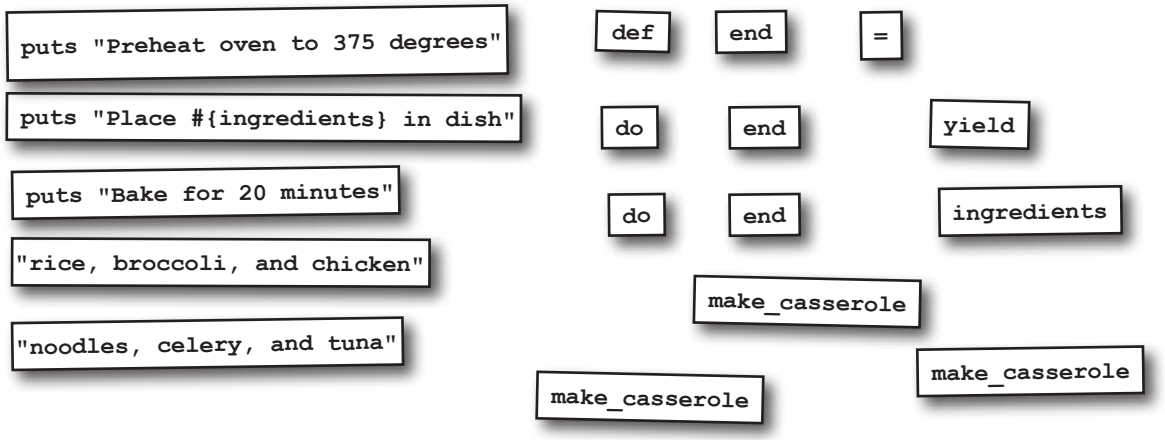
В: Можно ли передать блоку параметры *и* использовать его возвращаемое значение?

О: Да! Вы можете передать параметры, использовать возвращаемое значение, сделать и то и другое или не сделать ни того ни другого; это зависит только от вас.

```
def one_two
  result = yield(1, 2)
  puts result
end

one_two do |param1, param2|
  param1 + param2
end
```

Развлечения с магнитами

В программе на языке Ruby, выложенной на холодильнике, часть магнитов перепуталась. Сможете ли вы расставить фрагменты кода по местам, чтобы программа выводила указанный результат?



Результат:

```
File Edit Window Help
Preheat oven to 375 degrees
Place noodles, celery, and tuna in dish
Bake for 20 minutes
Preheat oven to 375 degrees
Place rice, broccoli, and chicken in dish
Bake for 20 minutes
```

Развлечения с магнитами. Решение

В программе на языке Ruby, выложенной на холодильнике, часть магнитов перепуталась. Сможете ли вы расставить фрагменты кода по местам, чтобы программа выводила указанный результат?

```
def make_casserole
  puts "Preheat oven to 375 degrees"
  ingredients = yield
  puts "Place #{ingredients} in dish"
  puts "Bake for 20 minutes"
end
```

```
make_casserole do
  "noodles, celery, and tuna"
end
```

```
make_casserole do
  "rice, broccoli, and chicken"
end
```

Результат:

```
File Edit Window Help
Preheat oven to 375 degrees
Place noodles, celery, and tuna in dish
Bake for 20 minutes
Preheat oven to 375 degrees
Place rice, broccoli, and chicken in dish
Bake for 20 minutes
```

Как метод использует возвращаемое значение блока

Мы подошли совсем близко к расшифровке этого загадочного фрагмента кода:

```
lines.find_all { |line| line.include?("Truncated") }
```

Осталось разобраться с методом `find_all`. Этот метод передает каждый элемент массива блоку и строит новый массив, содержащий только те элементы, для которых блок вернул `true`.

```
p [1, 2, 3, 4, 5].find_all { |number| number.even? }
p [1, 2, 3, 4, 5].find_all { |number| number.odd? }
```

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

Значения, возвращаемые блоком, представляют собой своего рода *инструкции* для метода. Задача метода `find_all` — оставить одни элементы массива и отбросить другие. Чтобы определить, какие элементы следует оставить, метод полагается на возвращаемое значение блока.

Значения, возвращаемые блоком, представляют собой своего рода инструкции для метода.

В процессе отбора важно только возвращаемое значение блока. Вообще говоря, в теле блока даже не обязательно использовать параметр с текущим элементом массива (хотя в большинстве реальных программ он используется). Если блок возвращает `true` для всех элементов, то включены будут *все* элементы массива...

```
p ['a', 'b', 'c'].find_all { |item| true }
```

```
["a", "b", "c"]
```

...а если он всегда возвращает `false`, то не будет включен *ни один* элемент.

```
p ['a', 'b', 'c'].find_all { |item| false }
```

```
[]
```

Если бы мы захотели написать собственную версию `find_all`, она могла бы выглядеть примерно так:

```
def find_all
  matching_items = []
  self.each do |item|
    if yield(item)
      matching_items << item
    end
  end
  matching_items
end
```

Создание нового массива для хранения элементов, для которых блок возвращает true.

Обработка всех элементов.

Элемент передается блоку. Если блок возвращает true...

...то элемент добавляется в массив подходящих элементов.

Выглядит знакомо? Так оно и должно быть. Это обобщенная версия приведенного выше кода для поиска строк, относящихся к нашему фильму!

Старый код:

```
relevant_lines = []
lines.each do |line|
  if line.include?("Truncated")
    relevant_lines << line
  end
end
puts relevant_lines
```

Все вместе

Теперь, когда вы знаете, как работает метод `find_all`, непонятных мест в коде почти не осталось.

```
lines = []

File.open("reviews.txt") do |review_file|
  lines = review_file.readlines
end

relevant_lines = lines.find_all { |line| line.include?("Truncated") }
```

Уже почти разобрались!

Вот что вы узнали (не обязательно в таком порядке):

- Последнее выражение в блоке становится его возвращаемым значением.

Результат будет использоваться как возвращаемое значение блока.

```
lines.find_all { |line| line.include?("Truncated") }
```

- Метод `include?` возвращает `true`, если строка содержит заданную подстроку, или `false`, если не содержит.

```
lines.find_all { |line| line.include?("Truncated") }
```

Возвращает true, если строка содержит «Truncated».

- Метод `find_all` передает каждый элемент массива блоку и строит новый массив, включающий только те элементы, для которых блок вернул `true`.

```
lines.find_all { |line| line.include?("Truncated") }
```

Результатом будет массив, который содержит все элементы «lines», содержащие строку «Truncated».

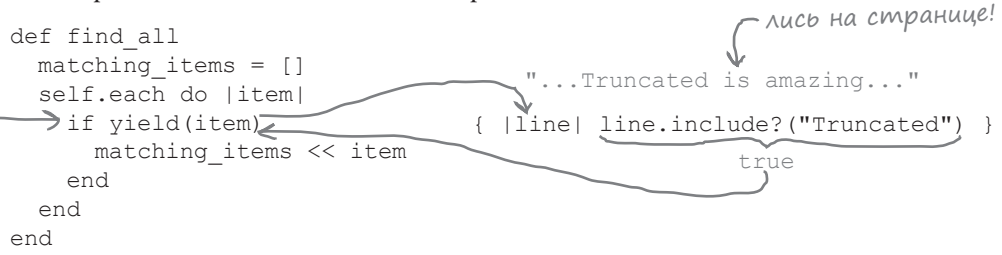
А теперь заглянем вовнутрь метода `find_all` и блока во время обработки нескольких начальных строк файла и посмотрим, что же там происходит...

Возвращаемые значения блоков: присмотримся повнимательнее

1 Метод `find_all` передает первую строку из файла блоку, который получает ее в параметре `line`. Блок проверяет, содержит ли `line` подстроку `"Truncated"`. Если содержит, то возвращаемое значение блока истинно. В методе строка добавляется в массив отобранных элементов.

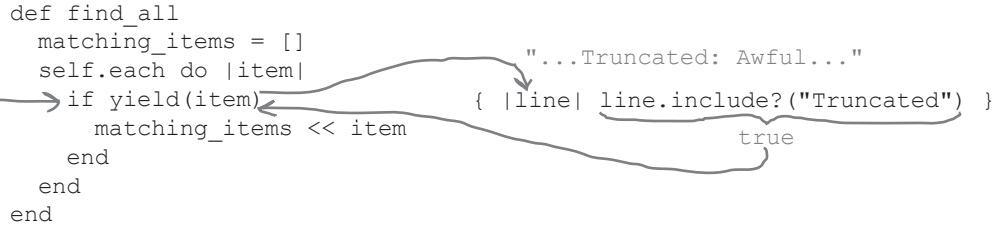
«find_all» передает полные строки; мы просто сократили их, чтобы они поместились на странице!

Блок возвращает true, поэтому текущая строка добавляется в matching_items!



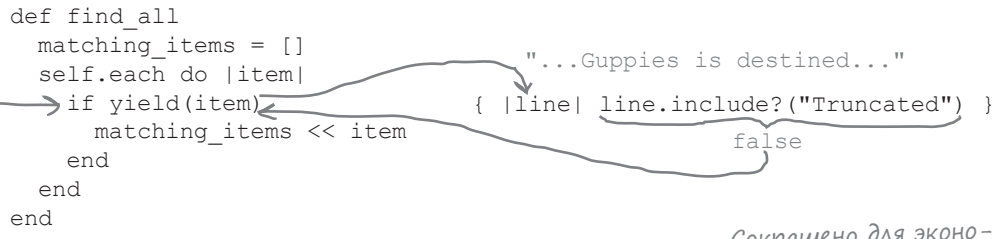
2 Метод `find_all` передает блоку вторую строку из файла. И снова параметр блока `line` включает строку `"Truncated"`, так что возвращаемое значение блока снова равно `true`. В методе эта строка также будет добавлена в массив отобранных элементов.

И снова возвращаемое значение блока равно true, поэтому строка также будет добавлена в массив.



3 *Третья* строка из файла *не содержит* `"Truncated"`, поэтому возвращаемое значение блока равно `false`. Эта строка *не* включается в массив.

Возвращаемое значение блока равно false, поэтому строка НЕ добавляется.



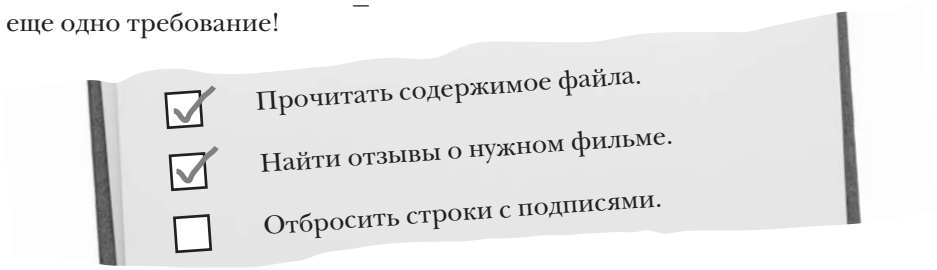
...и так далее для всех остальных строк в файле. Метод `find_all` добавляет текущий элемент в новый массив, если блок возвращает `true`, и пропускает его, если блок возвращает `false`. В результате будет получен массив, содержащий только те строки, в которых упоминается нужный фильм!

Сокращено для экономии места!

```
p relevant_lines
["...Truncated is amazing...",
 "...Truncated: Awful...",
 "...Truncated is funny...",
 "...Truncated Disappoints...",
 "...Truncated is astounding...",
 "...Don't See Truncated..."]
```

Исключение лишних элементов с использованием блока

Используя метод `find_all`, мы успешно нашли все отзывы к нужному фильму и поместили их в массив `relevant_lines`. Из списка можно вычеркнуть еще одно требование!



Следующее требование — отбросить строки с подписями, потому что нас интересуют только прилагательные из основного текста каждого отзыва.

От таких строк надо избавиться:

```
Normally producers and directors would stop this kind of
garbage from getting published. Truncated is amazing in that
it got past those hurdles.
    --Joseph Goldstein, "Truncated: Awful," New York Minute
Truncated is funny: it can't be categorized as comedy,
romance, or horror, because none of those genres would want
to be associated with it.
    --Liz Smith, "Truncated Disappoints," Chicago Some-Times
...
```

К счастью, все эти строки четко помечены. Каждая из них начинается с символов `--`, поэтому метод `include?` сможет легко определить, содержит ли строка подпись.

Ранее мы использовали метод `find_all` для *сохранения* строк, включающих заданную строку. Метод `reject` фактически является противоположностью `find_all` — он передает элементы из массива блоку и *отклоняет* элемент, если блок возвращает значение `true`. Если метод `find_all` полагается на информацию от блока для определения того, какие элементы следует *оставить*, `reject` использует информацию от блока для определения того, какие элементы следует *отбросить*.

Если бы мы решили написать собственную реализацию `reject`, она получилась бы очень похожей на `find_all`:

```
def reject
  kept_items = []
  self.each do |item|
    unless yield(item)
      kept_items << item
    end
  end
  kept_items
end
```

Создание нового массива для элементов, для которых блок возвращает false.

Обработка каждого элемента. Элемент передается блоку. Если возвращаемый результат равен false...

...элемент добавляется в массив сохраняемых элементов.

Возвращаемые значения для «reject»

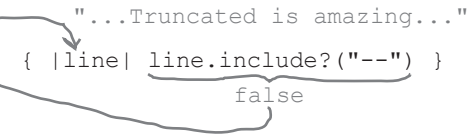
Итак, `reject` работает так же, как `find_all`, если не считать того, что вместо *сохранения* элементов, для которых блок возвращает `true`, он *отклоняет* их. С помощью `reject` мы сможем легко избавиться от подписей!

```
reviews = relevant_lines.reject { |line| line.include?("--") }
```

1 Метод `reject` передает блоку первую строку из файла. Параметр блока `line` *не* включает строку `"--"`, так что возвращаемое значение блока равно `false`. В методе эта строка добавляется в массив сохраняемых элементов.

Блок возвращает false, поэтому текущая строка добавляется в массив сохраняемых элементов.

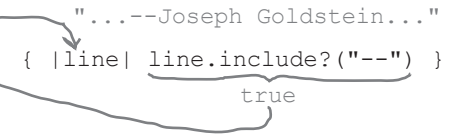
```
def reject
  kept_items = []
  self.each do |item|
    unless yield(item)
      kept_items << item
    end
  end
  kept_items
end
```



2 Метод `reject` передает блоку вторую строку. Параметр `line` *содержит* строку `"--"`, поэтому возвращаемое значение блока равно `true`, и метод отбрасывает эту строку.

Блок возвращает true, поэтому строка НЕ добавляется в массив.

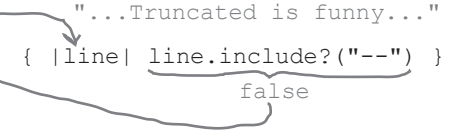
```
def reject
  kept_items = []
  self.each do |item|
    unless yield(item)
      kept_items << item
    end
  end
  kept_items
end
```



3 Третья строка *не содержит* `"--"`, поэтому возвращаемое значение блока равно `false`, и метод сохраняет эту строку.

Блок возвращает false, элемент остается.

```
def reject
  kept_items = []
  self.each do |item|
    unless yield(item)
      kept_items << item
    end
  end
  kept_items
end
```



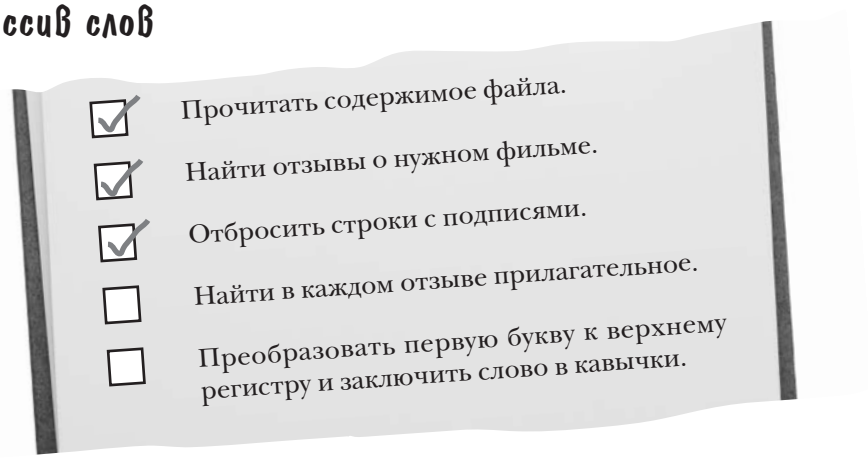
...и так далее для всех остальных строк в файле. Метод `reject` *пропускает* добавление строки в новый массив, если строка содержит `"--"`. В результате создается новый массив, в котором подписей нет, а включены только тексты отзывов!

```
p reviews
```

```
["...Truncated is amazing...",
 "...Truncated is funny...",
 "...Truncated is astounding..."]
```

Преобразование строки в массив слов

Строки с подписями отброшены, и у нас остался массив, содержащий только текст каждого отзыва. Еще одно требование выполнено! Остались еще два...



Для следующего требования нам понадобится пара новых методов. Блоков эти методы вообще не получают, но они *исключительно* полезны.

В каждом отзыве нужно найти прилагательное:

```
p reviews
```

```
["...Truncated is amazing...",
 "...Truncated is funny...",
 "...Truncated is astounding..."]
```

Необходимо отобрать только прилагательные...

Взглянув на этот фрагмент, вы заметите одну закономерность... Похоже, прилагательное всегда следует за словом *is*.

Итак, необходимо найти одно слово, следующее за другим словом... Пока мы работаем со *строками*. Как преобразовать эти строки в *слова*?

У строк имеется метод экземпляра split(), вызов которого разбивает строку на массив подстрок.

```
p "1-800-555-0199".split("-")
p "his/her".split("/")
p "apple, avocado, anvil".split(", ")
```

```
["1", "800", "555", "0199"]
["his", "her"]
["apple", "avocado", "anvil"]
```

В аргументе `split` передается *разделитель*: один или несколько символов, разбивающих строку на части.

Как разделяются слова в английском языке? Пробелами! Передавая методу `split " "` (пробел), мы получим массив. Опробуем его на первом отзыве.

```
string = reviews.first
words = string.split(" ")
p words
```

```
["Normally", "producers", "and", "directors",
"would", "stop", "this", "kind", "of", "garbage",
"from", "getting", "published.", "Truncated", "is",
"amazing", "in", "that", "it", "got", "past",
"those", "hurdles."]
```

Вот он — массив слов!

Определение индекса элемента массива

Метод `split` преобразует строку с отзывом в массив слов. Теперь необходимо найти в массиве слово *is*. И снова в Ruby имеется метод, готовый выполнить работу за вас. Если передать методу `find_index` аргумент, то он найдет первый индекс, с которым заданный элемент встречается в массиве.

```
p ["1", "800", "555", "0199"].find_index("800")
p ["his", "her"].find_index("his")
p ["apple", "avocado", "anvil"].find_index("anvil")
```

```
1
0
2
```

Воспользуемся `find_index` и напишем метод, который преобразует строку в массив слов, находит индекс слова *is* и возвращает слово, следующее *после* него.

```
def find_adjective(string)
  words = string.split(" ")
  index = words.find_index("is")
  words[index + 1]
end
```

Предложения разбиваются на слова.

Определение индекса слова «is» в массиве.

Находим слово ПОСЛЕ «is» и возвращаем его.

Проще всего протестировать метод на одном из отзывов...

```
adjective = find_adjective(reviews.first)
```

```
amazing
```

Да, это наше прилагательное! Впрочем, это только один отзыв. Теперь нужно сделать следующий шаг: обработать *все* отзывы и создать массив найденных прилагательных. С методом `each` это делается совсем несложно.

```
adjectives = []

reviews.each do |review|
  adjectives << find_adjective(review)
end

puts adjectives
```

```
amazing
funny
astounding
```

Создание нового массива для прилагательных.

Для каждого отзыва в массиве...

...вызвать написанный нами метод и добавить прилагательное в список.

В результате мы получили массив прилагательных, по одному для каждого отзыва!

А вы поверите, что существует еще более *простой* способ построения массива прилагательных для массива отзывов?

Создание массива на базе другого массива (сложный способ)

Мы без особых проблем организовали перебор массива отзывов и построили массив прилагательных с использованием `each` и нашего нового метода `find_adjective`.

Однако создание нового массива на базе содержимого другого массива — достаточно распространенная операция, для выполнения которой каждый раз используется примерно одинаковый код. Несколько примеров:

```
numbers = [2, 3, 4]
squares = []          # Создание массива для хранения результатов.
numbers.each do |number|   # Перебор исходного массива.
  squares << number ** 2
end                   # Выполнение операции и копирование результатов в массив.
p squares
```

```
[4, 9, 16]
```

```
numbers = [2, 3, 4]
cubes = []            # Создание массива для хранения результатов.
numbers.each do |number|   # Перебор исходного массива.
  cubes << number ** 3
end                   # Выполнение операции и копирование результата в массив.
p cubes
```

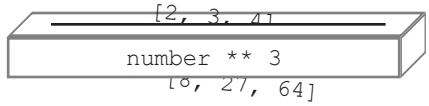
```
[8, 27, 64]
```

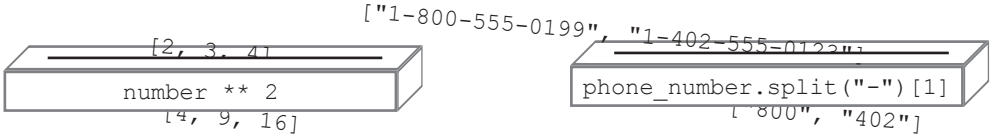
```
phone_numbers = ["1-800-555-0199", "1-402-555-0123"]
area_codes = []       # Создание массива для хранения результатов.
phone_numbers.each do |phone_number|   # Перебор исходного массива.
  area_codes << phone_number.split("-")[1]
end                   # Выполнение операции и копирование результата в массив.
p area_codes
```

```
["800", "402"]
```

В каждом из этих примеров создается новый массив для хранения результатов, осуществляется перебор в исходном массиве, к каждому элементу применяется некоторая логика, после чего результат добавляется в новый массив. (Как и в нашем коде поиска прилагательных.) Все это выглядит довольно однообразно...

Представьте, как было бы удобно иметь волшебный обработчик для массивов... Вы передаете массив, обработчик применяет некую (заменяемую) логику к элементам этого массива, после чего выдает новый массив, содержащий только нужные вам элементы!





Создание массива на базе другого массива с использованием «map»

В Ruby существует как раз такой волшебный обработчик: метод `map`. Метод `map` получает каждый элемент массива, передает его блоку и строит новый массив из значений, возвращаемых блоком.

```
squares = [2, 3, 4].map { |number| number ** 2 }
cubes = [2, 3, 4].map { |number| number ** 3 }
area_codes = ['1-800-555-0199', '1-402-555-0123'].map do |phone|
  phone.split("-")[1]
end
p squares, cubes, area_codes
```

Создавать массивы результатов заранее не нужно — «map» создаст их за нас!

Создание нового массива с квадратами чисел.

Создание нового массива с кубами чисел.

Создание нового массива с кодами зон.

```
[4, 9, 16]
[8, 27, 64]
["800", "402"]
```

Метод `map`, как и методы `find_all` и `reject`, обрабатывает каждый элемент в массиве. Но методы `find_all` и `reject` в зависимости от возвращаемого значения блока принимают решение о том, нужно ли копировать *исходный элемент* из старого массива в новый. Метод `map` добавляет *само возвращаемое значение блока* в новый массив.

Если бы мы захотели написать собственную версию `map`, она выглядела бы примерно так:

```
def map
  results = []
  self.each do |item|
    results << yield(item)
  end
  results
end
```

Создание нового массива для хранения возвращаемых значений блока.

Перебираем все элементы.

Элемент передается блоку, а возвращаемое значение добавляется в новый массив.

Возвращает массив возвращаемых значений блока.

Метод `map` способен сократить код выборки прилагательных до одной строки!

```
adjectives = reviews.map { |review| find_adjective(review) }
```

Массив всех возвращаемых значений find_adjective.

Здесь вызывается наш метод. Его возвращаемое значение определяет возвращаемое значение блока.

Возвращаемое значение `map` представляет собой массив всех значений, возвращенных блоком:

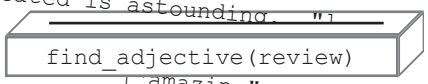
```
p adjectives
```

```
["amazing", "funny", "astounding"]
```

Создание массива на базе другого массива с использованием «map»

Давайте подробно, шаг за шагом разберемся, как метод `map` и наш блок обрабатывают массив с отзывами...

```
["...Truncated is amazing...",
 "...Truncated is funny...",
 "...Truncated is astounding..."]
```




```
["amazing",
 "funny",
 "astounding"]
```

```
adjectives = reviews.map { |review| find_adjective(review) }
```

1 Метод `map` передает первый отзыв блоку. Блок, в свою очередь, передает отзыв методу `find_adjective`, который возвращает строку `"amazing"`. Возвращаемое значение `find_adjective` также становится возвращаемым значением блока. В методе `map` `"amazing"` добавляется в массив `results`.

```
def map
  results = []
  self.each do |item|
    results << yield(item)
  end
  results
end
```

```
"...Truncated is amazing..."
{ |review| find_adjective(review) }
"amazing"
```

2 Блоку передается второй отзыв, `find_adjective` возвращает `"funny"`. В методе новое прилагательное добавляется в массив `results`.

```
def map
  results = []
  self.each do |item|
    results << yield(item)
  end
  results
end
```

```
"...Truncated is funny..."
{ |review| find_adjective(review) }
"funny"
```

3 Для третьего отзыва `find_adjective` возвращает прилагательное `"astounding"`, которое добавляется в массив вместе с другими.

```
def map
  results = []
  self.each do |item|
    results << yield(item)
  end
  results
end
```

```
"...Truncated is astounding..."
{ |review| find_adjective(review) }
"astounding"
```

Еще одно требование выполнено! Остался последний пункт, с которым сложностей не будет.

- [x] Найти в каждом отзыве прилагательное.
- [ ] Преобразовать первую букву к верхнему регистру и заключить слово в кавычки.

Дополнительная логика в теле блока «map»

Мы уже используем map для поиска прилагательного в каждом отзыве:

```
adjectives = reviews.map { |review| find_adjective(review) }
```

Наконец, необходимо преобразовать первую букву прилагательного к верхнему регистру и заключить его в кавычки. Это можно сделать в блоке, прямо после вызова метода find_adjective.

Теперь блок получает более одной строки, поэтому в соответствии с соглашением мы заключаем его в ограничители «do...end».

Мы будем работать с этим значением в будущем, поэтому присваиваем его переменной вместо того, чтобы просто вернуть.

```
adjectives = reviews.map do |review|
  adjective = find_adjective(review)
  "'#{adjective.capitalize}'"
end
```

А это новое возвращаемое значение.

Новые возвращаемые значения, которые генерируются этим кодом, выглядят так:

1

```
def map
  results = []
  self.each do |item|
    results << yield(item)
  end
  results
end
```

"...Truncated is amazing..."

```
do |review|
  adjective = find_adjective(review)
  "'#{adjective.capitalize}'"
end
```

"'Amazing'"

2

```
def map
  results = []
  self.each do |item|
    results << yield(item)
  end
  results
end
```

"...Truncated is funny..."

```
do |review|
  adjective = find_adjective(review)
  "'#{adjective.capitalize}'"
end
```

"'Funny'"

3

```
def map
  results = []
  self.each do |item|
    results << yield(item)
  end
  results
end
```

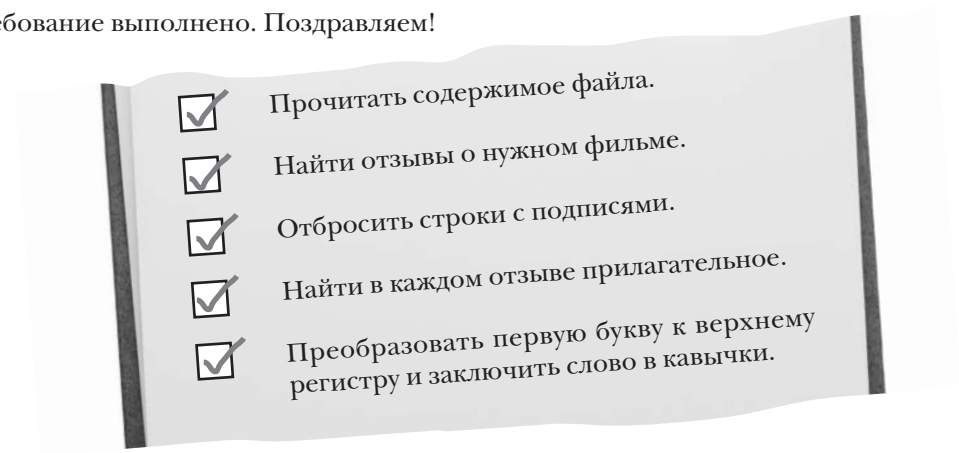
"...Truncated is astounding..."

```
do |review|
  adjective = find_adjective(review)
  "'#{adjective.capitalize}'"
end
```

"'Astounding'"

Работа закончена

Последнее требование выполнено. Поздравляем!



Итак, вы успешно научились использовать возвращаемые значения блоков для нахождения элементов в массиве и исключения лишних элементов и даже применять алгоритмы для построения совершенно новых массивов!

В других языках обработка таких сложных текстовых файлов потребует десятков строк кода с многочисленными повторениями. Методы `find_all`, `reject` и `map` сделают все за вас! Возможно, их изучение потребует некоторых усилий, но когда вы их освоите, в вашем распоряжении появятся невероятно мощные инструменты!

Ниже приведен полный код:

```
def find_adjective(string)
  words = string.split(" ")
  index = words.find_index("is")
  words[index + 1]
end

lines = []
File.open("reviews.txt") do |review_file|
  lines = review_file.readlines
end

relevant_lines = lines.find_all { |line| line.include?("Truncated") }
reviews = relevant_lines.reject { |line| line.include?("--") }

adjectives = reviews.map do |review|
  adjective = find_adjective(review)
  "'#{adjective.capitalize}'"
end

puts "The critics agree, Truncated is:"
puts adjectives
```

Этот метод вызывается ниже для поиска прилагательных в отзывах.

Строка преобразуется в массив слов.

Поиск индекса слова «is».

Возвращает слово, следующее за «is».

Эта переменная должна создаваться за пределами блока.

Файл открывается — и автоматически закрывается после завершения работы с ним.

Каждая строка файла читается в массив.

Находим строки, включающие название фильма.

Исключаем подписи критиков.

Обработка каждого отзыва.

Ищем прилагательное.

Возвращаем прилагательное, записанное с прописной буквы и заключенное в кавычки.

```
The critics agree, Truncated is:
'Amazing'
'Funny'
'Astounding'
```

Откройте терминальное окно, введите команду **`irb`** и нажмите клавишу Enter/Return. Под каждым из приведенных ниже выражений Ruby запишите, какой результат, по вашему мнению, будет получен при его выполнении. Потом введите выражение в irb и нажмите Enter. Совпадет ли ваше предположение с тем, что выдаст irb?

```
[1, 2, 3, 4].find_all { |number| number.odd? }
```
.....

```
[1, 2, 3, 4].find_all { |number| true }
```
.....

```
[1, 2, 3, 4].find_all { |number| false }
```
.....

```
[1, 2, 3, 4].find { |number| number.even? }
```
.....

```
[1, 2, 3, 4].reject { |number| number.odd? }
```
.....

```
[1, 2, 3, 4].all? { |number| number.odd? }
```
.....

```
[1, 2, 3, 4].any? { |number| number.odd? }
```
.....

```
[1, 2, 3, 4].none? { |number| number > 4 }
```
.....

```
[1, 2, 3, 4].count { |number| number.odd? }
```
.....

```
[1, 2, 3, 4].partition { |number| number.odd? }
```
.....

```
['$', '$$', '$$$'].map { |string| string.length }
```
.....

```
['$', '$$', '$$$'].max_by { |string| string.length }
```
.....

```
['$', '$$', '$$$'].min_by { |string| string.length }
```
.....

Откройте терминальное окно, введите команду **`irb`** и нажмите клавишу Enter/Return. Под каждым из приведенных ниже выражений Ruby запишите, какой результат, по вашему мнению, будет получен при его выполнении. Потом введите выражение в irb и нажмите Enter. Совпадет ли ваше предположение с тем, что выдаст irb?

```
[1, 2, 3, 4].find_all { |number| number.odd? }
```

`[1, 3]` ← Массив всех значений, для которых блок возвращает true.

```
[1, 2, 3, 4].find_all { |number| true }
```

`[1, 2, 3, 4]` ← Если он всегда возвращает true, то будут включены все значения.

```
[1, 2, 3, 4].find_all { |number| false }
```

`[]` ← Если он НИКОГДА не возвращает true, то не будет включено НИ ОДНО значение.

```
[1, 2, 3, 4].find { |number| number.even? }
```

`2` ← «find» возвращает ПЕРВОЕ значение, для которого блок возвращает true.

```
[1, 2, 3, 4].reject { |number| number.odd? }
```

`[2, 4]` ← Массив всех значений, для которых блок возвращает false.

```
[1, 2, 3, 4].all? { |number| number.odd? }
```

`false` ← «all?» возвращает true, если блок вернул true для ВСЕХ элементов.

```
[1, 2, 3, 4].any? { |number| number.odd? }
```

`true` ← «any?» возвращает true, если блок вернул true ХОТЯ БЫ ДЛЯ ОДНОГО элемента.

```
[1, 2, 3, 4].none? { |number| number > 4 }
```

`true` ← «none?» возвращает true, если блок вернул FALSE для всех элементов.

```
[1, 2, 3, 4].count { |number| number.odd? }
```

`2` ← Количество элементов, для которых блок вернул true.

```
[1, 2, 3, 4].partition { |number| number.odd? }
```

`[[1, 3], [2, 4]]` ← Два массива: первый содержит все элементы, для которых блок вернул TRUE, а второй — все элементы, для которых он вернул FALSE.

```
['$', '$$', '$$$'].map { |string| string.length }
```

`[1, 2, 3]` ← Массив всех значений, возвращаемых блоком.

```
['$', '$$', '$$$'].max_by { |string| string.length }
```

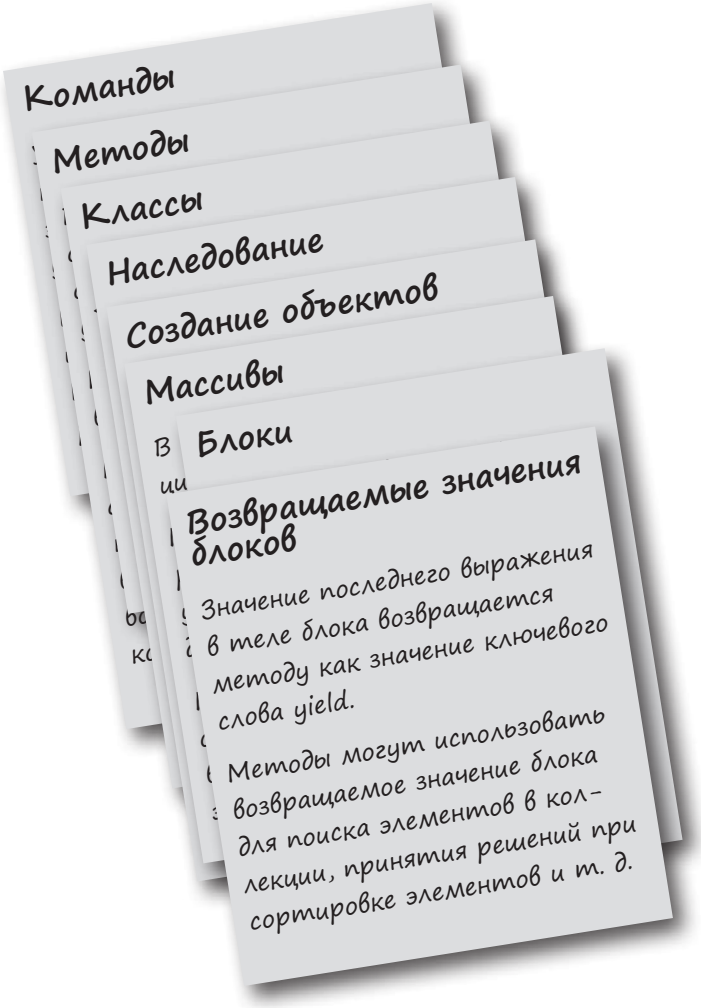
`"$$$"` ← Элемент, для которого блок вернул НАИБОЛЬШЕЕ значение.

```
['$', '$$', '$$$'].min_by { |string| string.length }
```

`"$"` ← Элемент, для которого блок вернул НАИМЕНЬШЕЕ значение.

Ваш инструментарий Ruby

Глава 6 осталась позади, а ваш инструментарий Ruby пополнился возвращаемыми значениями блоков.



КЛЮЧЕВЫЕ МОМЕНТЫ

- Если при вызове `File.open` передается блок, то метод передаст управление блоку для выполнения любых необходимых действий. При завершении блока файл будет автоматически закрыт.
- У строк имеется метод экземпляра `include?`, который получает подстроку в аргументе. Метод возвращает `true`, если строка включает подстроку, и `false` в противном случае.
- Если вам потребуется отобрать все элементы массива, удовлетворяющие некоторому критерию, используйте метод `find_all`. Метод передает каждый элемент массива блоку и возвращает новый массив со всеми элементами, для которых блок вернул `true`.
- Метод `reject` работает так же, как `find_all`, если не считать того, что он отклоняет элементы массива, для которых блок вернул `true`.
- Метод `split` для строк получает аргумент — разделитель. Он находит каждое вхождение разделителя в строки и возвращает массив со всеми подстроками, заключенными между разделителями.
- Метод `find_index` находит первое вхождение элемента в массиве и возвращает его индекс.
- Метод `map` берет каждый элемент массива, передает его блоку и строит новый массив из значений, возвращаемых блоком.

Далее в программе...

У массивов есть свои недостатки. Если вам потребуется найти в массиве конкретное значение, придется начать с начала и последовательно проверять каждый элемент. В следующей главе мы представим *хеши* – другую разновидность коллекций, которая ускоряет поиск элементов.

Пометка данных

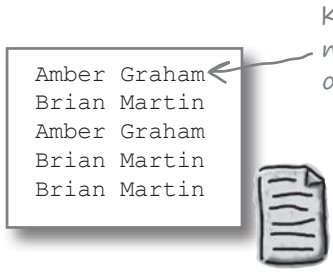
Хранить данные в одной большой куче удобно... До тех пор, пока вам не потребуется в них что-нибудь найти. Вы уже видели, как создать коллекцию объектов с использованием *массива*. Вы уже видели, как обработать *каждый элемент* массива и как *найти нужные* элементы. В обоих случаях мы начинаем от начала массива и *проверяем каждый отдельный объект.* Вы уже видели методы, получающие большие коллекции в параметрах. Вам уже известно, какие проблемы это создаст: вызовы методов требуют передачи большой, *запутанной коллекции аргументов,* для которой вам нужно помнить точный порядок. А не существует ли коллекции, у которой *все данные* уже снабжены *метками*? Тогда вы могли бы *быстро найти* нужные элементы! В этой главе рассматриваются **хеши** Ruby, предназначенные именно для этого.

Подсчет голосов

Место председателя попечительского совета вакантно, и опросы показывают, что голоса избирателей разделились почти поровну. В день выборов кандидаты, затаив дыхание, следят за поступающими результатами.

Электронные устройства для голосования, используемые в этом году, сохраняют результаты голосования в текстовых файлах, по одному голосу в строке. (Бюджет ограничен, так что городской совет выбрал самую дешевую модель.)

Перед вами файл с голосами по Округу А:



Требуется обработать каждую строку файла и подсчитать количество строк с каждым именем. Кандидат с наибольшим количеством голосов объявляется победителем!

Первое требование к группе разработки — прочитать содержимое файла *votes.txt*. Впрочем, здесь все просто; код ничем не отличается от кода, использованного для чтения отзывов о фильмах в главе 6.

```
lines = []
File.open("votes.txt") do |file|
  lines = file.readlines
end
```

Создаем переменную, которая продолжит существовать после блока.

Открываем файл и передаем его блоку.

Все строки файла сохраняются в массиве.

Следующая задача — прочитать имя из каждой строки файла и увеличить счетчик вхождений этого имени.

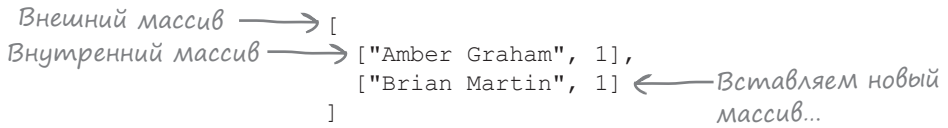
Массив массивов — не идеальное решение

Но как отследить все эти имена и связать с каждым именем счетчик вхождений? Мы покажем два способа. Первое решение использует массивы, уже известные нам по главе 5. Во втором решении используется новая структура данных — *хеши*.

Если бы в нашем распоряжении не было ничего, кроме массивов, то мы могли бы создать *массив массивов* для хранения всей информации. Да, вы поняли правильно: в массивах Ruby могут храниться любые объекты, включая *другие массивы*. Следовательно, мы можем создать массив с именем кандидата и количеством подсчитанных голосов в его пользу:

```
["Brian Martin", 1]
```

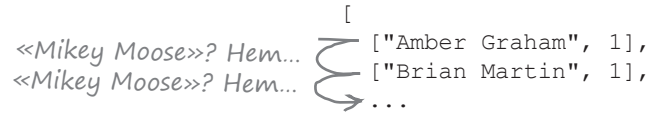
Этот массив можно разместить *внутри* другого массива, в котором хранятся имена *всех* кандидатов и *их* счетчики голосов:



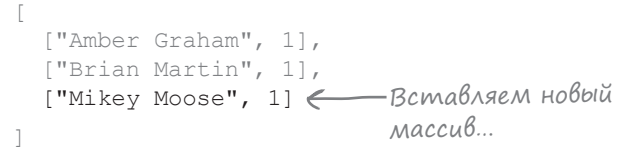
Для каждого имени, обнаруженного в текстовом файле...

```
"Mikey Moose"
```

...необходимо перебрать элементы *внешнего* массива и проверить, совпадает ли с именем первый элемент *внутреннего* массива.



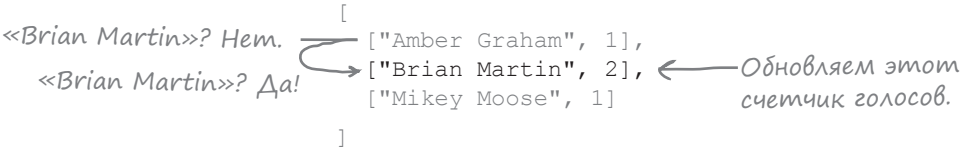
Если совпадений нет, добавляется новый внутренний массив с новым именем.



Но если в текстовом файле встречается имя, *уже существующее* в массиве массивов...

```
"Brian Martin"
```

...то мы обновляем счетчик вхождений для этого имени.



Да, такое решение *возможно*. Но оно потребует лишнего кода, а перебор элементов займет много времени при обработке больших списков. Как обычно, в Ruby существует более удобный способ.

Хеши

Основной недостаток решения с хранением счетчика вхождений для каждого кандидата в массиве — неэффективность его последующей выборки. При поиске любого имени каждый раз приходится просматривать *все остальные* имена.

```
[
  ["Amber Graham", 4],
  ["Brian Martin", 5],
  ["Mikey Moose", 2]
]
```

«Mikey Moose»? Нет...
«Mikey Moose»? Нет...
«Mikey Moose»?

Представьте, что данные, хранящиеся в массиве, свалены в одну кучу: нужный элемент можно найти, но чтобы найти его, придется просмотреть *все остальные* элементы.

Начинаем сверху и просматриваем всю кучу.

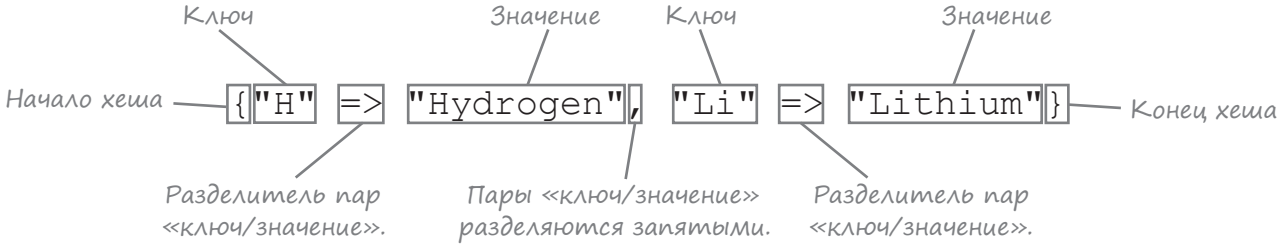
Массив

В Ruby существует другой способ хранения коллекций данных: *хеши*. **Хеш** — разновидность коллекции, в которой для обращения к значениям используются *ключи*. Ключи помогают легко извлечь нужные данные из хеша. Хранение данных в хеше можно сравнить с рядом аккуратно подписанных папок вместо одной неудобной кучи.

Помогает быстро найти нужные данные!

Хеш

Как и в случае с массивами, вы можете создать новый хеш и сразу добавить в него данные, используя литерал хеша. Синтаксис выглядит примерно так:



Символы => указывают, на какое значение указывает тот или иной ключ. Из-за внешнего сходства эту конструкцию иногда называют «ракетой».

Созданный хеш можно присвоить переменной: `elements = {"H" => "Hydrogen", "Li" => "Lithium"}`

Для обращения к значениям из хеша используются ключи, связанные с этими значениями. Если литералы хешей определяются в *фигурных скобках*, для обращения к отдельным значениям используются *квадратные скобки*. Все это напоминает стандартный синтаксис обращения к значениям из массива, если не считать того, что в квадратные скобки заключается ключ хеша, а не числовой индекс.

Укажите здесь ключ хеша, и вы получите соответствующее значение.

```
puts elements["Li"]
puts elements["H"]
```

```
Lithium
Hydrogen
```

Хеши (продолжение)

В существующий хеш тоже можно добавлять новые ключи и значения. В этом случае синтаксис тоже очень похож на синтаксис присваивания элементу массива:

Ключ хеша, с которым связывается значение.

Новое значение.

```
elements["Ne"] = "Neon"
puts elements["Ne"]
```

```
Neon
```

Если индексами массива могут быть только *целые числа*, ключом хеша может быть *любой объект*. В частности, такими объектами могут быть строки, числа и символические имена.

```
mush = {1 => "one", "two" => 2, :three => 3.0}

p mush[:three]
p mush[1]
p mush["two"]
```

```
3.0
"one"
2
```

Индексами массива могут быть только целые числа, а ключом хеша может быть любой объект.

И хотя между массивами и хешами существуют принципиальные отличия, у них также найдется немало общего — достаточно, чтобы сравнить их.

Массивы:

- Массивы увеличиваются и уменьшаются по мере необходимости.
- В массивах могут храниться любые объекты, даже хеши и другие массивы.
- В массивах могут одновременно храниться экземпляры разных классов.
- Литералы заключаются в *квадратные скобки*.
- Для обращения к элементу указывается его индекс в *квадратных скобках*.
- В качестве индексов могут использоваться только целые числа.
- Индекс элемента определяется его позицией в массиве.

```
[2.99, 25.00, 9.99]
 ↑     ↑      ↑
 0     1      2
```

Хеши:

- Хеши увеличиваются и уменьшаются по мере необходимости.
- В хешах могут храниться любые объекты, даже массивы и другие хеши.
- В хешах могут одновременно храниться экземпляры разных классов.
- Литералы заключаются в *фигурные скобки*.
- Для обращения к элементу указывается его индекс в *квадратных скобках*.
- В качестве ключа может использоваться любой объект.
- Ключи не вычисляются, а задаются при добавлении значений.

```
{"M" => "Monday", "T" => "Tuesday"}
 ↑       ↑        ↑       ↑
Ключ  Значение  Ключ  Значение
```

Упражнение

Заполните пробелы в коде, чтобы он выводил показанный результат.

```
my_hash = {"one" => _____, :three => "four", _ => "six"}
puts my_hash[5]
puts my_hash["one"]
puts my_hash[_____]
my_hash[_____] = 8
puts my_hash["seven"]
```

Результат:

```
six
two
four
8
```

Упражнение
Решение

Заполните пробелы в коде, чтобы он выводил показанный результат.

```
my_hash = {"one" => "two", :three => "four", 5 => "six"}
puts my_hash[5]
puts my_hash["one"]
puts my_hash[:three]
my_hash["seven"] = 8
puts my_hash["seven"]
```

Результат:

```
six
two
four
8
```

Хеши — тоже объекты

Вы снова и снова слышите, что в Ruby все сущности являются объектами. Ранее вы уже видели, что массивы являются объектами, и скорее всего, вас не удивит, что и хеши тоже являются объектами.

```
protons = {"H" => 1, "Li" => 3, "Ne" => 10}
puts protons.class
```

```
Hash
```

Как и большинство объектов Ruby, хеши содержат много полезных методов экземпляра. Приведем небольшую подборку...

У них есть методы, которые поддерживаются всеми объектами Ruby, — такие, как `inspect`:

```
puts protons.inspect
```

```
{"H"=>1, "Li"=>3, "Ne"=>10}
```

Метод `length` возвращает количество пар «ключ/значение», хранящихся в хеше:

```
puts protons.length
```

```
3
```

Есть методы для быстрой проверки присутствия конкретных ключей или значений в хеше:

```
puts protons.has_key?("Ne")
```

```
true
```

```
puts protons.has_value?(3)
```

```
true
```

Есть методы, возвращающие массив со всеми ключами или всеми значениями из хеша:

```
p protons.keys
```

```
["H", "Li", "Ne"]
```

```
p protons.values
```

```
[1, 3, 10]
```

Как и у массивов, у хешей существуют методы, которым можно передать блок для перебора содержимого хеша. Например, метод `each` получает блок с *двумя* параметрами: ключом и значением. (Метод `each` будет более подробно рассмотрен через несколько страниц.)

```
protons.each do |element, count|
  puts "#{element}: #{count}"
end
```

```
H: 1
Li: 3
Ne: 10
```

Упражнение

Откройте терминальное окно, введите команду **`irb`** и нажмите клавишу Enter/Return. Под каждым из приведенных ниже выражений Ruby запишите, какой результат, по вашему мнению, будет получен при его выполнении. Потом введите выражение в irb и нажмите Enter. Совпадет ли ваше предположение с тем, что выдаст irb?

```
protons = { "He" => 2 }
```
.....

```
protons["He"]
```
.....

```
protons["C"] = 6
```
.....

```
protons["C"]
```
.....

```
protons.has_key?("C")
```
.....

```
protons.has_value?(119)
```
.....

```
protons.keys
```
.....

```
protons.values
```
.....

```
protons.merge({ "C" => 0, "Uh" => 147.2 })
```
.....

Часто Задаваемые Вопросы

В: Откуда взялось название «хеш»?

О: Откровенно говоря, это не лучшее название. В других языках подобные структуры данных называются «картами», «словарями» и «ассоциативными массивами» (потому что ключи ассоциируются со значениями). В Ruby используется термин «хеш», потому что для быстрого поиска ключей в хеше используется алгоритм *хеширования*. Подробности работы этого алгоритма выходят за рамки темы книги, но вы легко найдете дополнительную информацию в своей любимой поисковой системе.

Откройте терминальное окно, введите команду **`irb`** и нажмите клавишу Enter/Return. Под каждым из приведенных ниже выражений Ruby запишите, какой результат, по вашему мнению, будет получен при его выполнении. Потом введите выражение в irb и нажмите Enter. Совпадет ли ваше предположение с тем, что выдаст irb?

```
protons = { "He" => 2 }
```
{"He"=>2}

Как обычно, результатом команды присваивания является присвоенное значение

```
protons["He"]
```
2

Передаем ключ, получаем соответствующее значение.

```
protons["C"] = 6
```
6

Присвоенное значение.

```
protons["C"]
```
6

Читаем значение, только что записанное в хеш.

```
protons.has_key?("C")
```
true

Возвращает true, потому что указанный ключ присутствует в хеше.

```
protons.has_value?(119)
```
false

Возвращает false, потому что ни с одним ключом в хеше это значение не связано.

```
protons.keys
```
["He", "C"]

Массив, содержащий все ключи из хеша.

```
protons.values
```
[2, 6]

Массив, содержащий все значения из хеша.

```
protons.merge({ "C" => 0, "Uh" => 147.2 })
```
{"He"=>2, "C"=>0, "Uh"=>147.2}

Если ключ из нового хеша уже существует в старом хеше, то старое значение заменяется.

Если ключ еще не существует, он просто добавляется в хеш.

Хеши возвращают «nil» по умолчанию


votes.txt

```
Amber Graham
Brian Martin
Amber Graham
Brian Martin
Brian Martin
```

Вернемся к массиву строк, прочитанному из файла с результатами голосования. Итак, требуется вычислить количество вхождений каждого имени в этом массиве.

```
p lines
```

```
["Amber Graham\n", "Brian Martin\n", "Amber Graham\n",
 "Brian Martin\n", "Brian Martin\n"]
```

Эти символы новой строки были прочитаны из файла.

Вместо упоминавшегося выше массива массивов для хранения счетчиков мы воспользуемся хешем. Когда в массиве `lines` встречается имя, отсутствующее в хеше, оно добавляется в хеш.

Если прочитана эта строка... → `"Amber Graham"`

```
{
  "Amber Graham" => 1,
}
```

...то этот ключ и значение добавляются в хеш.

Для каждого нового имени, обнаруженного в файле, в хеш добавляется отдельная пара из ключа и значения.

Если прочитана эта строка... → `"Brian Martin"`

```
{
  "Amber Graham" => 1,
  "Brian Martin" => 1,
}
```

...то этот ключ и значение добавляются в хеш.

Если обнаруживается имя, которое уже было добавлено в хеш, вместо добавления происходит обновление счетчика.

Если то же имя будет прочитано снова... → `"Amber Graham"`

```
{
  "Amber Graham" => 2,
  "Brian Martin" => 1,
}
```

...мы обновляем соответствующее значение.

...и так далее, пока не будут подсчитаны все голоса.

Во всяком случае, так выглядит исходный план. Но первая версия кода, которая должна обрабатывать данные из файла, выдает сообщение об ошибке...

```
votes = {}                          # Создаем пустой хеш.

lines.each do |line|
  name = line.chomp                 # Удаляем символ новой строки.
  votes[name] += 1                  # Увеличиваем счетчик для текущего имени.
end

p votes
```

Ошибка

```
undefined
method `+' for
nil:NilClass
```

Что произошло? Как было показано в главе 5, при обращении к *элементу массива*, которому еще не было присвоено значение, вы получаете `nil`. При обращении к *ключу хеша*, которому еще не было присвоено значение, по умолчанию *также* используется значение `nil`.

```
array = []
p array[999]                        # Не существует.
hash = {}
p hash["I don't exist"]             # Не существует.
```

```
nil
nil
```

При обращении к счетчику голосов для имени кандидата, которому еще не присваивалось значение, вы также получите `nil`. А при попытке суммирования с `nil` происходит ошибка.

Хеши возвращают «nil» по умолчанию (продолжение)

Когда имя кандидата встречается впервые, вместо счетчика голосов из хеша будет получено значение `nil`. При попытке использовать его для суммирования происходит ошибка.

```
lines.each do |line|
  name = line.chomp
  votes[name] += 1
end
```

```
undefined method `+'
for nil:NilClass
```

Чтобы решить эту проблему, мы проверим, связано ли с текущим ключом хеша значение `nil`. Если нет, значит, значение счетчика может быть безопасно увеличено. Но если оно *равно* `nil`, необходимо задать начальное значение для этого ключа (`1`).

```
lines = []
File.open("votes.txt") do |file|
  lines = file.readlines
end

votes = {}

lines.each do |line|
  name = line.chomp
  if votes[name] != nil ← Если это имя встречалось ранее...
    votes[name] += 1 ← ...увеличиваем значение счетчика.
  else ← Если же это имя встречается впервые...
    votes[name] = 1 ← ...оно добавляется в хеш
                      со значением 1.
  end
end
```

А теперь в выходных данных выводится заполненный хеш. Программа работает!

```
p votes
```

```
{"Amber Graham"=>2, "Brian Martin"=>3}
```

Значение nil (и только nil) интерпретируется как ложное

Впрочем, в программу стоит внести еще одно небольшое улучшение: эта условная конструкция выглядит несколько уродливо.

```
if votes[name] != nil
```

Чтобы немного «подчистить» эту конструкцию, мы можем воспользоваться тем фактом, что в условных командах может использоваться *любое* выражение Ruby. Большинство результатов интерпретируется так, как если бы это была логическая истина, то есть `true`.

```
if "any string" ← Истинно
  puts "I'll be printed!"
end
```

```
if 42 ← Истинно
  puts "I'll be printed!"
end
```

```
if ["any array"] ← Истинно
  puts "I'll be printed!"
end
```

Собственно, кроме логического значения `false`, в Ruby существует только *одно* значение, которое интерпретируется в выражениях как ложное: `nil`.

```
if false ← «Настоящее» ложное значение.
  puts "I won't be printed!"
end
```

```
if nil ← Ложно
  puts "I won't, either!"
end
```

Значение nil (и только nil) интерпретируется как ложное (продолжение)

То обстоятельство, что Ruby интерпретирует `nil` как ложное значение, упрощает проверку того, были ли значения присвоены ранее или нет. Например, при обращении к значению из хеша в команде `if` содержащийся в команде код будет выполнен, если значение существует. Если же значение не существует, то и код не выполняется.

Значение nil интерпретируется как ложное.

Значение 1 интерпретируется как истинное.

```
votes = {}
if votes["Kremit the Toad"]
  puts "I won't be printed!"
end
votes ["Kremit the Toad"] = 1
if votes["Kremit the Toad"]
  puts "I'll be printed!"
end
```

Чтобы условная команда лучше читалась, ее можно сократить с

`if votes[name] != nil` до

`if votes[name]`.

Код работает так же, как в предыдущей версии; просто он стал чуть более компактным. На первый взгляд достижение небольшое, но в типичной программе существование объектов приходится проверять *очень много раз*. В конечном итоге этот прием сэкономит вам множество нажатий клавиш!

```
lines.each do |line|
  name = line.chomp
  if votes[name]
    votes[name] += 1
  else
    votes[name] = 1
  end
end

p votes
```

Громоздкая конструкция «if votes[name] != nil» больше не нужна!

```
{"Amber Graham"=>2, "Brian Martin"=>3}
```

Будьте осторожны!

Когда мы говорим, что только `nil` интерпретируется как логическая ложь, мы не преувеличиваем.

Многие значения, которые интерпретируются как ложные в других языках — пустые строки, пустые массивы, число `0`, — в Ruby интерпретируются как истинные.

Упражнение

Попробуйте предположить, какой результат будет выведен приведенным ниже кодом, и запишите его в пустых полях.

(Мы заполнили первую строку за вас.)

```
school = {
  "Simone" => "here",
  "Jeanie" => "here"
}

names = ["Simone", "Ferriss", "Jeanie", "Cameron"]

names.each do |name|
  if school[name]
    puts "#{name} is present"
  else
    puts "#{name} is absent"
  end
end
```

Simone is present

Попробуйте предположить, какой результат будет выведен приведенным ниже кодом, и запишите его в пустых полях.

```
school = {
  "Simone" => "here",
  "Jeanie" => "here"
}

names = ["Simone", "Ferriss", "Jeanie", "Cameron"]

names.each do |name|
  if school[name]
    puts "#{name} is present"
  else
    puts "#{name} is absent"
  end
end
```

Simone is present

Ferriss is absent

Jeanie is present

Cameron is absent

Возвращение по умолчанию другого значения вместо «nil»

Непропорционально большой объем кода подсчета голосов содержится в команде `if/else`, которая проверяет существование ключа в хеше...

```
votes = {}

lines.each do |line|
  name = line.chomp
  if votes[name]
    votes[name] += 1
  else
    votes[name] = 1
  end
end
```

Если значение votes[name] отлично от nil...

...увеличить существующий счетчик.

Если значение votes[name] РАВНО nil...

...добавить имя в хеш со значением 1.

Но эта команда `if` *нужна*. Обычно при обращении к ключу хеша, с которым еще не связано значение, вы получаете `nil`. Когда мы в первой версии попытались увеличить счетчик для несуществующего ключа, произошла ошибка (потому что `nil` не может использоваться при суммировании).

Для первого имени получает «nil» и пытается увеличить на 1...

```
lines.each do |line|
  name = line.chomp
  votes[name] += 1
end
```

Ошибка

```
undefined method `+'
for nil:NilClass
```

А если для отсутствующих ключей вместо `nil` будет возвращаться *другое* значение? Значение, которое *можно* увеличивать? Давайте посмотрим, как этого добиться...

Возвращение по умолчанию другого значения вместо «nil» (продолжение)

Вместо литералов хешей (`{}`) для создания новых хешей также можно воспользоваться методом `Hash.new`. Без аргументов `Hash.new` работает точно так же, как `{}`: вы получаете хеш, который возвращает `nil` для ключей, с которыми не были ассоциированы значения.

Но если при вызове `Hash.new` в аргументе передается объект, этот аргумент становится объектом хеша по умолчанию. Каждый раз, когда вы обращаетесь к ключу, которому значение еще не присвоено, вместо `nil` возвращается указанный вами объект по умолчанию.

Создаем новый хеш.

```
votes = Hash.new
votes["Amber Graham"] = 1
p votes["Amber Graham"]
p votes["Brian Martin"]
```

При обращении к уже присвоенному значению мы получаем это значение.

При обращении к значению, которое еще НЕ БЫЛО присвоено, мы получаем «nil».

```
1
nil
```

Создаем новый хеш с объектом по умолчанию «0».

```
votes = Hash.new(0)
votes["Amber Graham"] = 1
p votes["Amber Graham"]
p votes["Brian Martin"]
```

При обращении к уже присвоенному значению мы получаем это значение.

При обращении к значению, которое еще НЕ БЫЛО присвоено, мы получаем объект по умолчанию.

```
1
0
```

Воспользуемся объектом хеша по умолчанию, чтобы сократить код подсчета голосов.

Если хеш был создан вызовом `Hash.new(0)`, то при попытке обращения к счетчику для ключа, с которым еще не было ассоциировано значение, возвращается объект по умолчанию (`0`). Значение `0` увеличивается до `1`, потом до `2`, и так далее — каждый раз, когда то же имя будет снова встречаться в массиве.

Назначение любых объектов, кроме чисел, может привести к ошибкам!

Возможности безопасного использования других объектов будут рассмотрены в главе 8. А до тех пор не используйте ничего, кроме чисел, в качестве объектов по умолчанию!

От команды `if` можно полностью избавиться!

И как видно из результата, код по-прежнему нормально работает.

```
lines = []
File.open("votes.txt") do |file|
  lines = file.readlines
end

votes = Hash.new(0)

lines.each do |line|
  name = line.chomp
  votes[name] += 1
end

p votes
```

Создаем новый хеш с объектом по умолчанию «0».

Увеличиваем полученное значение: «0», если счетчик еще не обновлялся, или его текущее значение в противном случае.

```
{"Amber Graham"=>2, "Brian Martin"=>3}
```

Нормализация ключей хеша

Вот что получилось при обработке нового файла существующим кодом:

```
{"Amber Graham"=>2, "Brian Martin"=>3, "amber graham"=>3, "brian martin"=>1}
```

Эти два имени не должны подсчитываться по отдельности!

Нет, так не пойдет... Похоже, в нескольких последних строках имена кандидатов были записаны в нижнем регистре, и программа приняла их за *совершенно новых кандидатов*!

Здесь проявляется одна из проблем, типичных для работы с хешами: если вы хотите обратиться к значению или изменить его, указанный вами ключ должен *точно* совпадать с уже существующим. В противном случае он будет считаться совершенно новым ключом.

```
votes = Hash.new(0)
votes["Amber Graham"] = 1
p votes["Amber Graham"]
p votes["amber graham"]
```

Обращаемся к существующему значению.

Такая пара «ключ/значение» еще не встречалась!

```
1
0
```

Следовательно, строки в файле, записанные в нижнем регистре, должны считаться эквивалентными строкам, записанным в верхнем регистре. Как это сделать? Следует *нормализовать* входные данные: должен существовать один стандартный способ представления имен кандидатов, и именно *этот* способ должен использоваться для записи ключей хеша.

Нормализация ключей хеша (продолжение)

К счастью, в этом конкретном случае нормализация имен кандидатов происходит очень просто. Мы всего лишь добавим одну строку кода, которая устранит расхождения в регистре символов, перед сохранением имени в хеше.

```
lines = []
File.open("votes.txt") do |file|
  lines = file.readlines
end

votes = Hash.new(0)

lines.each do |line|
  name = line.chomp
  name.upcase!
  votes[name] += 1
end

p votes
```

Имя приводится к ВЕРХНЕМУ РЕГИСТРУ перед использованием в качестве ключа хеша.

В выводе программы мы видим обновленное содержимое хеша: голоса в записях с именами в нижнем и верхнем регистре суммируются, как и положено. Проблема решена!

```
{"AMBER GRAHAM"=>5, "BRIAN MARTIN"=>4}
```

Победитель определился!

Будьте осторожны!

Также необходимо нормализовать ключи при обращении к значениям.

Если вы нормализуете ключи при добавлении значений в хеш, также необходимо выполнить нормализацию и при обращении к значениям. В противном случае программа может решить, что значение отсутствует, тогда как на самом деле оно просто хранится под другим ключом.

Такого ключа нет!

```
p votes["Amber Graham"]
p votes["AMBER GRAHAM"]
```

...Зато есть такой!

```
nil
5
```

Хеши и «each»

Мы обработали строки в файле и построили хеш с общим количеством голосов:

```
p votes
```

```
{"AMBER GRAHAM"=>5, "BRIAN MARTIN"=>4}
```

Впрочем, было бы намного удобнее, если бы каждое имя кандидата выводилось в отдельной строке вместе со счетчиком голосов.

Как было показано в главе 5, у массивов есть метод `each`, которому передается блок с одним параметром. Метод `each` последовательно передает каждый элемент массива блоку для обработки. У хешей тоже есть метод `each`, который работает аналогично. Есть всего одно отличие: для хешей `each` передается блок с *двумя* параметрами: ключом и соответствующим значением.

```
hash = { "one" => 1, "two" => 2 }
hash.each do |key, value|
  puts "#{key}: #{value}"
end
```

```
one: 1
two: 2
```

Воспользуемся `each` для вывода каждого имени кандидата из хеша `votes` вместе с соответствующим счетчиком голосов:

```
lines = []
File.open("votes.txt") do |file|
  lines = file.readlines
end

votes = Hash.new(0)

lines.each do |line|
  name = line.chomp
  name.upcase!
  votes[name] += 1
end

votes.each do |name, count|
  puts "#{name}: #{count}"
end
```

Ключ. *Значение.*

Обрабатываем каждую пару «ключ/значение».

```
AMBER GRAHAM: 5
BRIAN MARTIN: 4
```

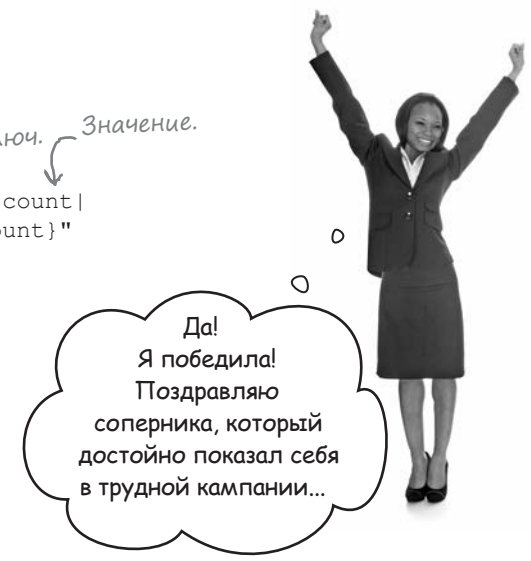
Результаты голосования хорошо видны и аккуратно отформатированы!

Итак, мы рассмотрели одно из классических применений хеша: программу, в которой требуется многократно обращаться к значениям с известным ключом. Сейчас будет представлена еще одна типичная ситуация с использованием хешей: передача аргументов методов.

Часто Задаваемые Вопросы

В: Что произойдет, если вызвать метод `each` для хеша, но передать ему блок с *одним* параметром?

О: Метод `each` для хешей допускает такую возможность; блоку передается массив из двух элементов с ключом и значением для каждой пары «ключ/значение» в хеше. Впрочем, на практике намного чаще применяются блоки с двумя параметрами.



Беседа у камина

Сегодня в студии: Массив и Хеш обсуждают свои различия.

Хеш:

Рад видеть, Массив.

Зря ты так.

Что ж, у меня есть определенное обаяние... Но даже я знаю, что в некоторых ситуациях разработчику *приходится* использовать массив вместо хеша.

Это верно; чтобы элементы можно было быстро извлечь, приходится потрудиться над организацией их хранения! Но зато все усилия окупятся, если вы захотите получить конкретный элемент из середины коллекции. Если вы даете мне правильный ключ, я всегда знаю, где найти значение.

Да, но ведь разработчик должен знать точный индекс, по которому хранятся данные, верно? Конечно, запоминать все эти числа весьма утомительно! Но либо так, либо ждать, пока массив просмотрит все свои элементы, один за одним...

Согласен. Разработчики должны знать о массивах *и* хешах и выбирать подходящую структуру данных для своей текущей задачи.

Массив:

Не могу сказать того же, но будь по-твоему, Хеш.

Да неужели? Я прекрасно справлялся со своим делом — хранением коллекций. А потом явился ты, и разработчики стали говорить: «Ооо! Зачем нужны массивы, когда можно использовать хеш? Хеш — это же так круто!»

Точно! Массивы намного эффективнее хешей! И если вас устраивает выборка элементов в том порядке, в котором они были добавлены (допустим, с использованием `each`), вы выбираете массив, потому что вам не нужно ждать, пока хеш упорядочит данные за вас.

Вообще-то массивы тоже умеют возвращать данные.

Важно другое: я *могу* это сделать. И при построении простой очереди я остаюсь предпочтительным кандидатом.

Резонно, на том и порешим.

Путаница с аргументами

Предположим, мы пишем приложение для хранения основной информации о кандидатах, чтобы пользователи могли побольше узнать о них. Для хранения всей информации о кандидате в одном месте создан класс `Candidate`. Для удобства мы создали метод `initialize`, чтобы все атрибуты экземпляра можно было задать прямо при вызове `Candidate.new`.

Создание методов доступа к атрибутам.

Настройка передачи аргументов Candidate.new.

Параметры используются для назначения атрибутов объекта.

```
class Candidate
  attr_accessor :name, :age, :occupation, :hobby, :birthplace
  def initialize(name, age, occupation, hobby, birthplace)
    self.name = name
    self.age = age
    self.occupation = occupation
    self.hobby = hobby
    self.birthplace = birthplace
  end
end
```

Добавим после определения класса код создания экземпляра `Candidate` и вывода его данных.

```
def print_summary(candidate)
  puts "Candidate: #{candidate.name}"
  puts "Age: #{candidate.age}"
  puts "Occupation: #{candidate.occupation}"
  puts "Hobby: #{candidate.hobby}"
  puts "Birthplace: #{candidate.birthplace}"
end

candidate = Candidate.new("Carl Barnes", 49, "Attorney", nil, "Miami")
print_summary(candidate)
```

Аргумент должен передаваться даже в том случае, если он не используется.

```
Candidate: Carl Barnes
Age: 49
Occupation: Attorney
Hobby:
Birthplace: Miami
```

Первая попытка вызова `Candidate.new` показывает, что пользоваться им довольно неудобно. Нам приходится передавать все аргументы независимо от того, используются они или нет.

Параметр `hobby` можно было бы объявить необязательным, *если бы* за ним не следовал параметр `birthplace`...

```
class Candidate
  attr_accessor :name, :age, :occupation, :hobby, :birthplace
  def initialize(name, age, occupation, hobby = nil, birthplace)
    ...
  end
end
```

Значение по умолчанию, чтобы сделать параметр необязательным...

Но из-за параметра `birthplace` попытка опустить параметр `hobby` приводит к ошибке...

```
Candidate.new("Carl Barnes", 49, "Attorney", , "Miami")
```

Ошибка →

```
syntax error, unexpected ',', expecting ')'
```

Путаница с аргументами (продолжение)

Если вы забудете, в каком порядке должны передаваться аргументы метода, возникает другая проблема...

```
candidate = Candidate.new("Amy Nguyen", 37, "Lacrosse", "Engineer", "Seattle")
print_summary(candidate)
```

Постойте, в каком порядке все это нужно передавать?

Понятно, что с длинными списками параметров методов могут возникнуть проблемы. Их порядок не очевиден, и разработчик легко может что-то пропустить.

Ой! Эти два аргумента должны идти в обратном порядке!

```
Candidate: Amy Nguyen
Age: 37
Occupation: Lacrosse
Hobby: Engineer
Birthplace: Seattle
```

Передача хешей в параметрах методов

Традиционно в Ruby подобные проблемы решались посредством использования хешей как параметров методов. Ниже приведен простой метод `area`, которому вместо раздельных параметров `length` и `width` передается один хеш. (Да, мы понимаем, что запись получается немного громоздкой. На нескольких ближайших страницах мы представим некоторые приемы, существенно упрощающие синтаксис передачи хешей в параметрах!)

Один хеш вместо нескольких параметров.

В Ruby принято использовать символические имена в качестве ключей.

```
def area(options)
  options[:length] * options[:width]
end

puts area({:length => 2, :width => 4})
```

Обращаемся к значениям из хеша вместо отдельных параметров.

Вместо нескольких аргументов передается один хеш с соответствующими ключами и значениями.

```
8
```

По общепринятым соглашениям в Ruby для ключей в хешах параметров используются символические имена, потому что выборка по символическому имени выполняется более эффективно, чем выборка по строковому ключу.

Передача хеша обладает рядом преимуществ перед обычными параметрами методов.

Обычные параметры:

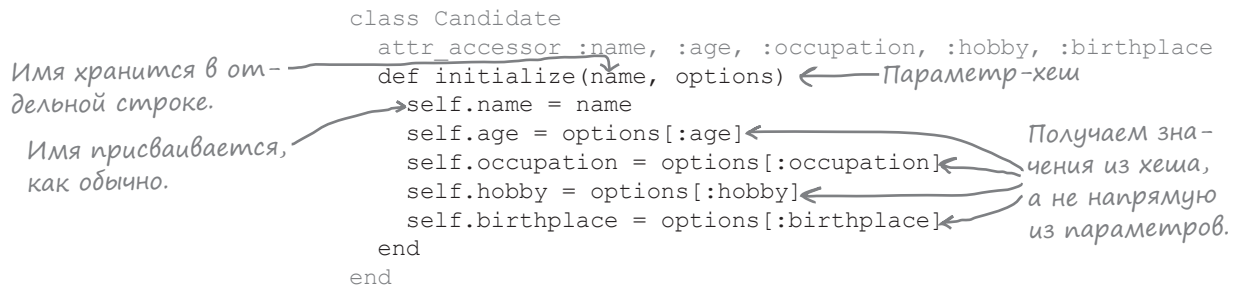
- Аргументы должны следовать в строго заданном порядке.
- Аргументы бывает трудно отличить друг от друга.
- Обязательные параметры должны предшествовать необязательным.

Хеши:

- Ключи могут следовать в любом порядке.
- Ключи могут выполнять функции «меток» для значений.
- Разработчик может опустить значение любого ключа по своему усмотрению.

Параметр-хеш в классе Candidate

Ниже приведена переработанная версия метода `initialize` класса `Candidate` с параметром-хешем.




```
class Candidate
  attr_accessor :name, :age, :occupation, :hobby, :birthplace
  def initialize(name, options)
    self.name = name
    self.age = options[:age]
    self.occupation = options[:occupation]
    self.hobby = options[:hobby]
    self.birthplace = options[:birthplace]
  end
end
```

Теперь при вызове `Candidate.new` сначала передается строка с именем, а затем следует хеш со значениями всех остальных атрибутов `Candidate`:

Теперь понятно, какой атрибут соответствует каждому аргументу!

```
candidate = Candidate.new("Amy Nguyen",
  {:age => 37, :occupation => "Engineer", :hobby => "Lacrosse", :birthplace => "Seattle"})

p candidate
```

```
#<Candidate:0x007fbd7a02e858 @name="Amy Nguyen", @age=37,
 @occupation="Engineer", @hobby="Lacrosse", @birthplace="Seattle">
```

Теперь ошибки с порядком исключены!

Ключи хеша при желании можно опустить. В этом случае атрибуту будет присвоен объект хеша по умолчанию, `nil`.

Атрибут hobby можно не указывать.

```
candidate = Candidate.new("Carl Barnes",
  {:age => 49, :occupation => "Attorney", :birthplace => "Miami"})

p candidate
```

```
#<Candidate:0x007f8aaa042a68 @name="Carl Barnes", @age=49,
 @occupation="Attorney", @hobby=nil, @birthplace="Miami">
```

По умолчанию пропущенным атрибутам присваивается nil.

Наконец, ключи хеша могут следовать в произвольном порядке:

```
candidate = Candidate.new("Amy Nguyen",
  {:birthplace => "Seattle", :hobby => "Lacrosse", :occupation => "Engineer", :age => 37})

p candidate
```

```
#<Candidate:0x007f81a890e8c8 @name="Amy Nguyen", @age=37,
  @occupation="Engineer", @hobby="Lacrosse", @birthplace="Seattle">
```

Фигурные скобки не нужны!

Откровенно говоря, с этими фигурными скобками вызовы методов выглядят хуже обычных вызовов методов с обычными аргументами:

```
candidate = Candidate.new("Carl Barnes",
  {:age => 49, :occupation => "Attorney"})
```

...поэтому Ruby позволяет опустить фигурные скобки при условии, что хеш передается в последнем аргументе:

```
candidate = Candidate.new("Carl Barnes",
  :age => 49, :occupation => "Attorney")
p candidate
```

Без фигурных скобок!

```
#<Candidate:0x007fb412802c30
 @name="Carl Barnes", @age=49,
 @occupation="Attorney",
 @hobby=nil, @birthplace=nil>
```

По этой причине многие методы, определяющие параметр-хеш, определяют его как последний параметр.

И стрелки не нужны!

Ruby предоставляет еще один удобный прием сокращенной записи. Если хеш использует символические имена в качестве ключей, литералы хешей позволяют опускать двоеточия (:) в символических именах и заменять «ракету» (=>) двоеточием.

```
candidate = Candidate.new("Amy Nguyen", age: 37,
  occupation: "Engineer", hobby: "Lacrosse")
p candidate
```

Те же символические имена, но читаются гораздо лучше!

```
#<Candidate:0x007f9dc412aa98
 @name="Amy Nguyen", @age=37,
 @occupation="Engineer",
 @hobby="Lacrosse",
 @birthplace=nil>
```

Не спорим, в исходном варианте аргументы-хеши были довольно-таки уродливыми. Но теперь, когда вы освоили все приемы, упрощающие запись, они выглядят уже вполне прилично, не так ли? Совсем как обычные аргументы методов, но с удобными и понятными метками!

```
Candidate.new("Carl Barnes", age: 49, occupation: "Attorney")
Candidate.new("Amy Nguyen", age: 37, occupation: "Engineer")
```

Часто Задаваемые Вопросы

В: Что такого особенного в параметре-хеше? По мне так самый обычный параметр метода!

О: А это *и есть* самый обычный параметр метода; ничто не мешает вам передать вместо хеша целое число, строку и т. д. Но скорее всего, это приведет к ошибке, когда код вашего метода попытается обратиться к ключам и значениям целого числа или строки!

Житейская мудрость

Когда вы определяете метод, получающий параметр-хеш, проследите за тем, чтобы этот параметр стоял на последнем месте: это позволит опустить фигурные скобки при вызове вашего метода. При вызове метода с передачей хеша в аргументе фигурные скобки следует опускать, если это возможно, — это упрощает чтение кода. И наконец, используйте символические имена в качестве ключей параметра-хеша; они повышают эффективность кода.

Сам хеш тоже может быть необязательным

Осталось еще одно усовершенствование, которое мы можем внести в метод `initialize` класса `Candidate`. В настоящее время при вызове можно передать все ключи хеша:

```
Candidate.new("Amy Nguyen", age: 37, occupation: "Engineer",
  hobby: "Lacrosse", birthplace: "Seattle")
```

А можно опустить *большинство* из них:

```
Candidate.new("Amy Nguyen", age: 37)
```

Но если вы попытаетесь опустить *все* ключи, то получите сообщение об ошибке:

```
p Candidate.new("Amy Nguyen")
```

Ошибка →

```
in `initialize': wrong number of arguments (1 for 2)
```

Если ни один ключ не указан, то с точки зрения Ruby аргумент с хешем вообще не передается.

Чтобы избежать подобной непоследовательности, можно назначить пустой хеш по умолчанию для аргумента `options`:

```
class Candidate
  attr_accessor :name, :age, :occupation, :hobby, :birthplace
  def initialize(name, options = {})
    self.name = name
    self.age = options[:age]
    self.occupation = options[:occupation]
    self.hobby = options[:hobby]
    self.birthplace = options[:birthplace]
  end
end
```

Если хеш не передается, использовать пустой хеш.

Если аргумент-хеш не передается, по умолчанию будет использоваться пустой хеш. Всем атрибутам `Candidate` будет присвоено значение по умолчанию `nil` из пустого хеша.

```
p Candidate.new("Carl Barnes")
```

```
#<Candidate:0x007fbe0981ec18 @name="Carl Barnes", @age=nil,
 @occupation=nil, @hobby=nil, @birthplace=nil>
```

Если же указана хотя бы одна пара «ключ/значение», аргумент с хешем будет интерпретироваться так же, как прежде:

```
p Candidate.new("Carl Barnes", occupation: "Attorney")
```

```
#<Candidate:0x007fbe0981e970 @name="Carl Barnes", @age=nil,
 @occupation="Attorney", @hobby=nil, @birthplace=nil>
```

Развлечения с магнитами

В программе на языке Ruby, выложенной на холодильнике, часть магнитов перепуталась. Сможете ли вы расставить фрагменты кода по местам, чтобы программа выводила указанный результат?

```
volume
result =
options[:depth]
width: 10,
volume
end
options[:height]
)
*
height: 5,
def
(options)
options[:width]
(
*
depth: 2.5
puts "Volume is #{result}"
```

Результат:

```
File Edit Window Help
Volume is 125.0
```

Развлечения с магнитами. Решение

```
def volume (options)
    options[:width] * options[:height] * options[:depth]
end

result = volume ( width: 10, height: 5, depth: 2.5 )

puts "Volume is #{result}"
```

Результат:

```
File Edit Window Help
Volume is 125.0
```

Опасные опечатки в аргументах-хешах

У параметров-хешей есть один недостаток, о котором мы еще не упоминали, и он только и ждет, как бы создать проблемы в самый неподходящий момент... Например, следующий код вроде бы должен присвоить значение атрибута `occupation` нового экземпляра `Candidate`. Как ни странно, он этого не делает:

```
p Candidate.new("Amy Nguyen", occupaiton: "Engineer")
```

```
#<Candidate:0x007f862a022cb0 @name="Amy Nguyen", @age=nil,
 @occupation=nil, @hobby=nil, @birthplace=nil>
```

Почему атрибут содержит nil?

Почему код не работает? Из-за опечатки в символическом имени в ключе хеша!

```
p Candidate.new("Amy Nguyen", occupaiton: "Engineer")
```

Упс!

Этот код даже не выводит сообщение об ошибке. Наш метод `initialize` просто использует значение, связанное с правильно записанным ключом `options[:occupation]`, — которое, разумеется, равно `nil`, потому что это значение не присваивалось.

Не беспокойтесь! В версии 2.0 в Ruby появилась поддержка ключевых слов в аргументах, которая предотвращает подобные проблемы.

Ключевые слова в аргументах

Вместо того чтобы требовать передачи одного параметра-хеша в определениях методов, можно указать отдельные ключи хеша, которые должны передаваться при вызове. Для этого используется следующий синтаксис:

При таком определении метода вам не придется беспокоиться о передаче ключей хеша телу метода. Ruby сохраняет каждое значение в отдельном параметре, к которому можно обращаться напрямую по имени, как к обычному параметру метода.

Метод, определенный подобным образом, можно вызывать с передачей ключей и значений, как это делалось ранее:

```
welcome(greeting: "Hello", name: "Amy")
```

```
Hello, Amy!
```

В действительности при вызове просто передается хеш:

```
my_arguments = {greeting: "Hello", name: "Amy"}
welcome(my_arguments)
```

```
Hello, Amy!
```

Однако хеш обрабатывается внутри метода особым образом. Всем ключам, опущенным при вызове метода, присваиваются указанные значения по умолчанию:

```
welcome(name: "Amy")
```

```
Welcome, Amy!
```

А если при вызове встречаются неизвестные ключевые слова (например, если в имени ключа допущена опечатка), выдается сообщение об ошибке:

```
welcome(greting: "Hello", nme: "Amy")
```

Ошибка →

```
ArgumentError: unknown
keywords: greting, nme
```

Использование ключевых слов в аргументах в классе Candidate

В текущей версии класса `Candidate` метод `initialize` использует параметр-хеш. Код получается излишне громоздким, а вызывающая сторона не узнает о возможных опечатках в ключах хеша.

```
class Candidate
  attr_accessor :name, :age, :occupation, :hobby, :birthplace
  def initialize(name, options = {})
    self.name = name
    self.age = options[:age]
    self.occupation = options[:occupation]
    self.hobby = options[:hobby]
    self.birthplace = options[:birthplace]
  end
end
```

Параметр-хеш

Обращение к значениям из хеша

Переработаем метод `initialize` класса `Candidate` с ключевыми словами в аргументах.

Параметр-хеш заменяется ключевыми словами и значениями по умолчанию.

```
class Candidate
  attr_accessor :name, :age, :occupation, :hobby, :birthplace
  def initialize(name, age: nil, occupation: nil, hobby: nil, birthplace: "Sleepy Creek")
    self.name = name
    self.age = age
    self.occupation = occupation
    self.hobby = hobby
    self.birthplace = birthplace
  end
end
```

Вместо ключей хеша используются имена параметров.

Строка `"Sleepy Creek"` становится значением по умолчанию для ключевого слова `birthplace`, а `nil` — значением по умолчанию для остальных ключевых слов. Также все ссылки на хеш `options` в теле метода заменяются именами параметров. В этой версии метод читается гораздо проще!

При этом он может вызываться в той же форме, что и прежде...

```
p Candidate.new("Amy Nguyen", age: 37, occupation: "Engineer")
```

```
#<Candidate:0x007fbf5b14e520 @name="Amy Nguyen",
 @age=37, @occupation="Engineer", @hobby=nil, @birthplace="Sleepy Creek">
```

Заданные значения. *Значения по умолчанию.*

...и он предупредит об опечатке, допущенной в ключевом слове.

```
p Candidate.new("Amy Nguyen", occupaiton: "Engineer")
```

Ошибка → `ArgumentError: unknown keyword: occupaiton`

Обязательные ключевые слова в аргументах

В текущей версии метод `Candidate.new` можно вызвать даже без указания минимальной информации о кандидате:

```
p Candidate.new("Carl Barnes")
```

Всем атрибутам присваиваются значения по умолчанию!

```
#<Candidate:0x007fe743885d38 @name="Carl Barnes",
 @age=nil, @occupation=nil, @hobby=nil, @birthplace="Sleepy Creek">
```

Такое решение не идеально. Нужно потребовать, чтобы при вызове был указан хотя бы возраст (age) и профессия (occupation) кандидата.

В старой версии, в которой метод `initialize` использовал обычные параметры метода, это не создавало проблем; *все* аргументы были обязательными.

```
class Candidate
  attr_accessor :name, :age, :occupation, :hobby, :birthplace
  def initialize(name, age, occupation, hobby, birthplace)
    ...
  end
end
```

Сделать параметр метода необязательным можно только одним способом: предоставить для него значение по умолчанию.

```
class Candidate
  attr_accessor :name, :age, :occupation, :hobby, :birthplace
  def initialize(name, age = nil, occupation = nil, hobby = nil, birthplace = nil)
    ...
  end
end
```

Но постойте — мы же предоставляем значения по умолчанию для всех своих ключевых слов.

```
class Candidate
  attr_accessor :name, :age, :occupation, :hobby, :birthplace
  def initialize(name, age: nil, occupation: nil, hobby: nil, birthplace: "Sleepy Creek")
    ...
  end
end
```

Если убрать значение по умолчанию для обычного параметра метода, этот параметр становится обязательным; вызвать метод без указания его значения не удастся. А что будет, если убрать значения по умолчанию для *ключевых слов* в аргументах?

Обязательные аргументы в ключевых словах (продолжение)

Попробуем убрать значения по умолчанию для ключевых слов `age` и `occupation` и посмотрим, будут ли они обязательными при вызове `initialize`.

Впрочем, убирать двоеточие после ключевого слова нельзя. В этом случае Ruby не сможет отличить `age` и `occupation` от обычных параметров метода.

Обязательные ключевые слова в аргументах появились только в Ruby 2.1.

Если вы работаете с Ruby 2.0, при попытке использования обязательных ключевых слов в аргументах будет выдано сообщение о синтаксической ошибке. Либо переходите на версию 2.1 (или выше), либо предоставьте значения по умолчанию.

```
class Candidate
  attr_accessor :name, :age, :occupation, :hobby, :birthplace
  def initialize(name, age, occupation, hobby: nil, birthplace: "Sleepy Creek")
    ...
  end
end
```

Обычные параметры, не ключевые слова!

Что же произойдет, если удалить значение по умолчанию, но оставить двоеточие после ключевого слова?

```
class Candidate
  attr_accessor :name, :age, :occupation, :hobby, :birthplace
  def initialize(name, age:, occupation:, hobby: nil, birthplace: "Sleepy Creek")
    self.name = name
    self.age = age
    self.occupation = occupation
    self.hobby = hobby
    self.birthplace = birthplace
  end
end
```

Ключевые слова без значений по умолчанию!

Мы по-прежнему сможем вызывать `Candidate.new`, если при вызове будут указаны все обязательные ключевые слова:

```
p Candidate.new("Carl Barnes", age: 49, occupation: "Attorney")
```

```
#<Candidate:0x007fcec281e5a0 @name="Carl Barnes",
@age=49, @occupation="Attorney", @hobby=nil, @birthplace="Sleepy Creek">
```

...а если обязательные ключевые слова будут пропущены, Ruby предупредит нас об этом!

```
p Candidate.new("Carl Barnes")
```

Ошибка →

```
ArgumentError: missing
keywords: age, occupation
```

Прежде вам приходилось передавать `Candidate.new` длинный список безликих аргументов, причем с *абсолютно* точным соблюдением порядка. Теперь, когда вы научились использовать хеши для передачи аргументов (явно или скрытно при помощи ключевых слов в аргументах), ваш код станет намного элегантнее!

Упражнение

Перед вами определения двух методов Ruby. Свяжите каждый из шести вызовов, приведенных ниже, с тем результатом, который будет получен при его выполнении.

(Первую связь мы уже обозначили за вас.)

```
def create(options = {})
  puts "Creating #{options[:database]} for owner #{options[:user]}..."
end

def connect(database:, host: "localhost", port: 3306, user: "root")
  puts "Connecting to #{database} on #{host} port #{port} as #{user}..."
end
```

A `create(database: "catalog", user: "carl")`

B `create(user: "carl")`

C `create`

D `connect(database: "catalog")`

E `connect(database: "catalog", password: "1234")`

F `connect(user: "carl")`

..... `Creating  for owner carl...`

..... `unknown keyword: password`

..... `Connecting to catalog on localhost port 3306 as root...`

A ..... `Creating catalog for owner carl...`

..... `Creating  for owner ...`

..... `missing keyword: database`

Упражнение
Решение

Перед вами определения двух методов Ruby. Свяжите каждый из шести вызовов, приведенных ниже, с тем результатом, который будет получен при его выполнении.

```
def create(options = {})
  puts "Creating #{options[:database]} for owner #{options[:user]}..."
end

def connect(database:, host: "localhost", port: 3306, user: "root")
  puts "Connecting to #{database} on #{host} port #{port} as #{user}..."
end
```

A `create(database: "catalog", user: "carl")`

B `create(user: "carl")`

C `create`

D `connect(database: "catalog")`

E `connect(database: "catalog", password: "1234")`

F `connect(user: "carl")`

B
```
Creating  for owner carl...
```

E
```
unknown keyword: password
```

D
```
Connecting to catalog on localhost port 3306 as root...
```

A
```
Creating catalog for owner carl...
```

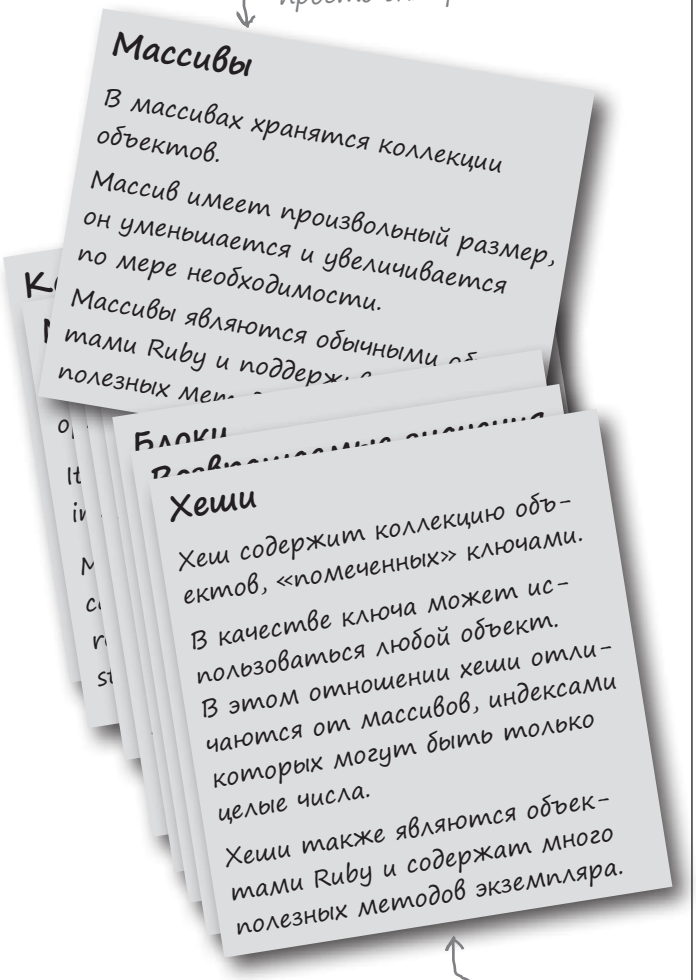
C
```
Creating  for owner ...
```

F
```
missing keyword: database
```

Ваш инструментарий Ruby

Глава 7 осталась позади, а ваш инструментарий пополнился приемами работы с хешами.

Наши заметки о массивах из главы 5 — просто для сравнения...



...а это заметки к этой главе!

Далее в программе...

Мы уже показали, как задать значение по умолчанию для хеша. При неправильном использовании значения по умолчанию могут породить странные ошибки. Проблема связана со *ссылками* на объекты, о которых вы узнаете в следующей главе...

КЛЮЧЕВЫЕ МОМЕНТЫ

- Литералы хешей заключаются в фигурные скобки. Они должны содержать ключ для каждого значения: `{"one" => 1, "two" => 2}`
- При обращении к ключу хеша, с которым ранее не связывалось значение, по умолчанию возвращается `nil`.
- Любое выражение Ruby может использоваться в условных командах. Кроме логического значения `false`, есть только одно значение, которое Ruby интерпретирует как ложное: `nil`.
- Для создания хешей вместо литерала хеша может использоваться метод `Hash.new`. Если передать объект в аргументе `Hash.new`, этот объект становится объектом по умолчанию для хеша. При обращении к любому ключу, с которым ранее не связывалось значение, возвращается объект по умолчанию (вместо `nil`).
- Если ключ, по которому вы обращаетесь, не совпадает полностью с ключом из хеша, он рассматривается как другой ключ.
- У хешей существует метод `each`, очень похожий на метод `each` массивов. Главное отличие заключается в том, что передаваемый блок должен (обычно) получать два параметра (вместо одного): для ключа и для соответствующего значения.
- Если хеш передается в последнем аргументе метода, Ruby позволяет опустить фигурные скобки.
- Если хеш использует символические имена в качестве ключей, то двоеточие в символическом имени можно не указывать и заменить `=>` двоеточием: `{name: "Kim", age: 28}`
- При определении метода можно указать, что вызывающая сторона должна предоставить ключевые слова в аргументах. Ключевые слова и значения в действительности образуют хеш, но значения помещаются в именованные параметры внутри метода.
- Чтобы сделать ключевые слова в аргументах необязательными, определите для них значение по умолчанию.

Путаница с сообщениями

Вы когда-нибудь отправляли сообщения не тому контакту?
Наверное, вам пришлось изрядно потрудиться, чтобы разобраться с возникшей путаницей. *Объекты Ruby* очень похожи на *контакты* в адресной книге, и *вызов методов* для них напоминает *отправку сообщений*. Если содержимое вашей адресной книги будет *перепутано*, вы рискуете отправить сообщение *не тому объекту*. В этой главе вы научитесь *распознавать такие ситуации* и узнаете, как снова *наладить работу ваших программ*.

Загадочные ошибки

Молва продолжает распространяться — если у кого-то возникли проблемы с программой на языке Ruby, ваша компания способна эту проблему решить. И к вашей двери все чаще приходят люди с какими-нибудь необычными просьбами...

Астроном полагает, что он придумал умный прием, который избавит его от написания лишнего кода. Вместо того чтобы вводить команды `my_star = CelestialBody.new` и `my_star.type = 'star'` для каждой звезды, которую он захочет создать, он намерен просто *скопировать* исходную звезду и присвоить ей новое имя.

```
class CelestialBody
  attr_accessor :type, :name
end

altair = CelestialBody.new
altair.name = 'Altair'
altair.type = 'star'
polaris = altair
polaris.name = 'Polaris'
vega = polaris
vega.name = 'Vega'

puts altair.name, polaris.name, vega.name
```

Для экономии времени скопируем предыдущую звезду...

...и просто изменим ее имя.

То же самое здесь.

```
Vega
Vega
Vega
```

Странно, имена всех трех звезд теперь стали одинаковыми!

Но, похоже, план дал осечку. Все три экземпляра `CelestialBody` утверждают, что они имеют *одно и то же* имя!

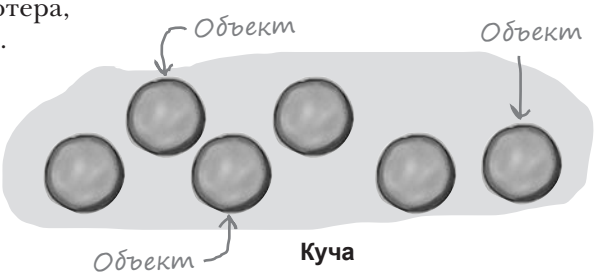
Куча

Ошибка в программе возникает из-за принципиальной проблемы: разработчик *думает*, что он работает с *несколькими* объектами, тогда как в действительности он раз за разом работает с *одним* объектом.

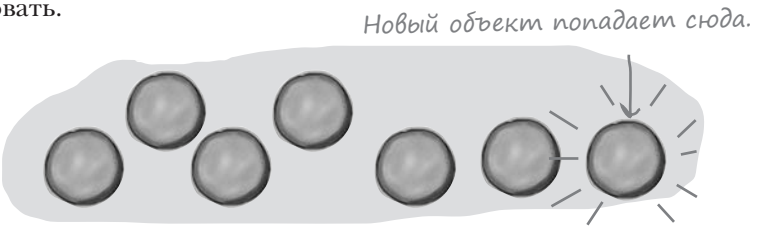
Чтобы понять, как такое возможно, необходимо разобраться в том, где на самом деле «живут» объекты и как ваши программы взаимодействуют с ними.

Программисты, работающие на Ruby, часто говорят о «хранении объектов в переменных», «хранении объектов в массивах», «сохранении объекта в значении хеша» и т. д. Но это лишь упрощенное описание того, что происходит в действительности, потому что объект не может находиться *в* переменной, массиве или хеше.

Вместо этого все объекты Ruby существуют в **куче** — области памяти вашего компьютера, предназначенной для хранения объектов.



При создании нового объекта Ruby выделяет место в куче, где этот объект может существовать.



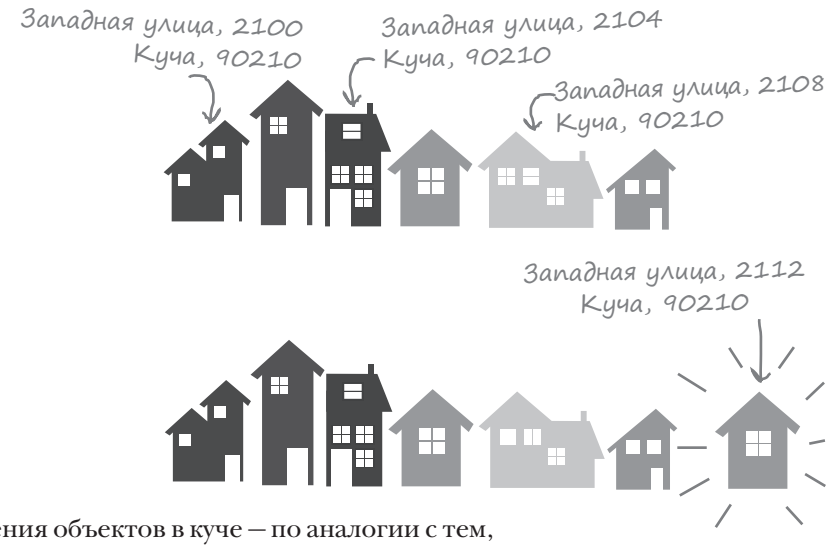
В общем случае разработчику не нужно отвлекаться на технические подробности работы с кучей — Ruby делает это за него. Если потребуется дополнительная память, размер кучи автоматически увеличивается. Неиспользуемые объекты также автоматически удаляются из кучи. Обычно все это происходит само собой, не требуя вашего участия.

Однако программисту *необходим* механизм *чтения* объектов, хранящихся в куче. Для этой цели используются *ссылки*, которые более подробно рассматриваются ниже.

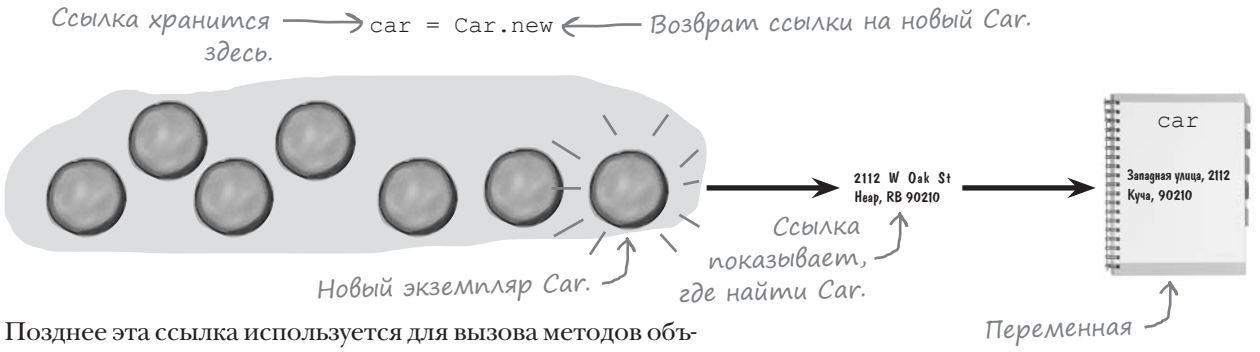
Ссылки

Допустим, вы хотите отправить письмо конкретному человеку. Как почтальон найдет адресата? У каждого дома имеется *адрес*, по которому отправляется почта. Вы просто пишете адрес на конверте, а почтальон использует его для поиска нужного дома и доставки письма.

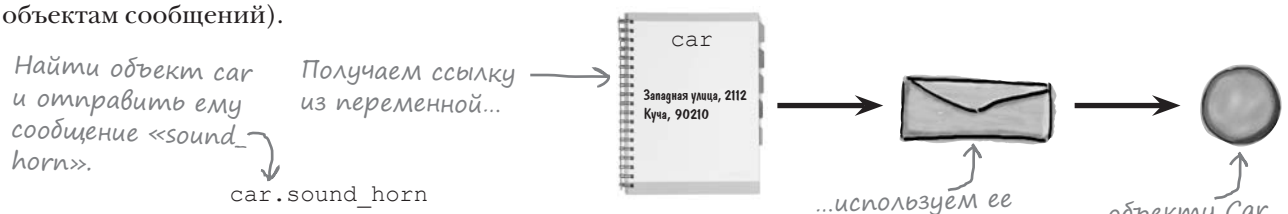
Если ваш друг переезжает в новый дом, он сообщает вам свой адрес, который вы записываете в адресную книгу или в другое удобное место. По этому адресу вы сможете связаться с ним в будущем.



Ruby использует **ссылки** для нахождения объектов в куче — по аналогии с тем, как вы используете адрес для поиска дома. При создании нового объекта Ruby возвращает ссылку на этот объект. Программист сохраняет ссылку в переменной, массиве или другом удобном месте. Ссылка, как и домашний адрес, сообщает Ruby, где объект «живет» в куче.



Позднее эта ссылка используется для вызова методов объекта (что, как вы помните, можно сравнить с отправкой объектам сообщений).



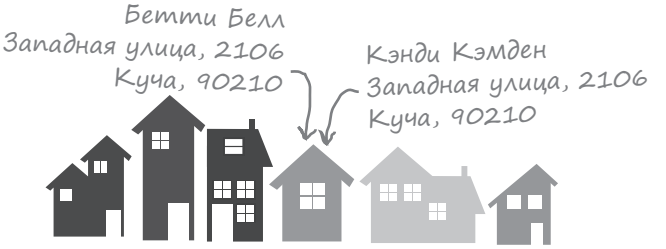
Итак, необходимо твердо понять: в переменных, массивах, хешах и т. д. объекты *никогда* не хранятся. В них хранятся *ссылки* на объекты. Объекты хранятся в куче, а для *обращения* к ним используются ссылки, хранящиеся в переменных.

Путаница со ссылками

На прошлой неделе Артур познакомился не с одной, а сразу с *двумя* красотками: Бетти и Кэнди. А самое замечательное, что обе живут на одной улице с ним.

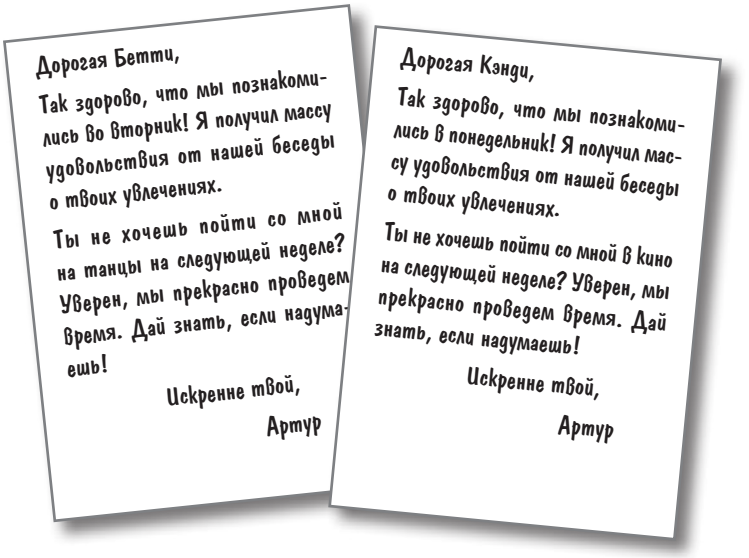
Западная улица в реальности

Артур собирается записать адреса своих новых знакомых в свою адресную книгу. Как ни печально, он случайно записывает *один и тот же* адрес (адрес Бетти) для *обеих* женщин.



Западная улица по адресной книге Артура

Позднее на этой неделе Бетти получает *два* письма от Артура:



Теперь Бетти не желает видеть Артура, а Кэнди (не получившая ни одного письма) считает, что Артур не обратил на нее внимания.

Какое отношение вся эта история имеет к решению проблем в программах Ruby? Сейчас узнаете...

Наложение имен

Для моделирования проблемы Артура на языке Ruby можно воспользоваться простым классом `LoveInterest`. Класс `LoveInterest` содержит метод экземпляра `request_date`, который выводит положительный ответ только один раз. При повторном вызове этого метода экземпляр `LoveInterest` сообщает, что он занят.

```
class LoveInterest

  def request_date
    if @busy
      puts "Sorry, I'm busy."
    else
      puts "Sure, let's go!"
      @busy = true
    end
  end

end
```

@busy содержит nil (и поэтому интерпретируется как ложное значение), пока это значение не будет заменено другим.

Если это не первый запрос...

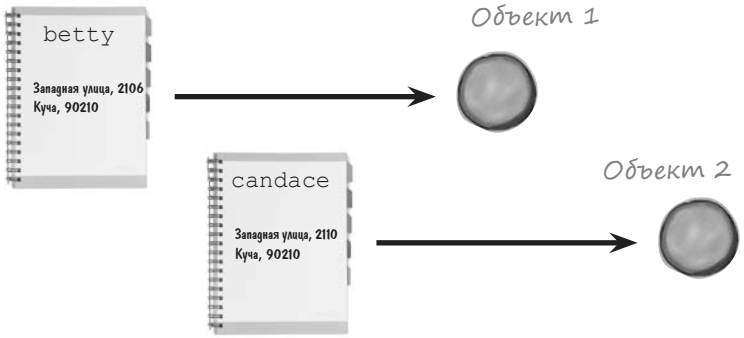
...выдается отрицательный ответ.

Положительный ответ.

Объект помечается для отклонения всех будущих запросов.

Обычно при использовании этого класса мы создаем два разных объекта и сохраняем ссылки на них в двух разных переменных:

```
betty = LoveInterest.new
candace = LoveInterest.new
```



При использовании двух разных ссылок для вызова `request_date` для двух разных объектов мы получаем два положительных ответа, как и ожидалось.

```
betty.request_date
candace.request_date
```

```
Sure, let's go!
Sure, let's go!
```

Чтобы убедиться в том, что мы действительно работаем с двумя разными объектами, можно воспользоваться методом экземпляра `object_id`, который поддерживается практически всеми объектами objects. Метод возвращает уникальный идентификатор объекта.

```
p betty.object_id
p candance.object_id
```

```
70115845133840
70115845133820
```

Два разных объекта

Наложение имен (продолжение)

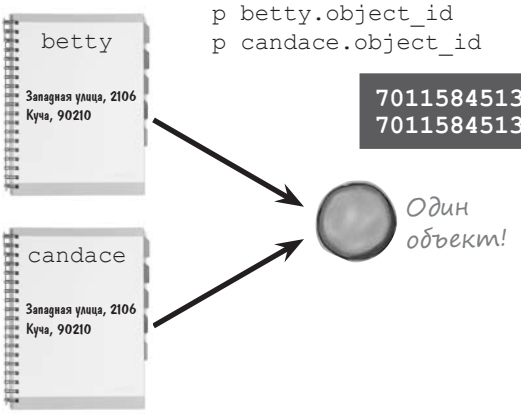
Но если вместо этого *скопировать* ссылку, в итоге мы получим две разные ссылки на *один* объект под двумя разными *именами* (переменные `betty` и `candace`).

Такое явление называется *наложением имен*, потому что разные *имена* обозначают один объект. Наложение имен создает немало опасностей, если вы его не ожидаете!

```
betty = LoveInterest.new
candace = betty

p betty.object_id
p candace.object_id
```

```
70115845133560
70115845133560
```

Один и тот же объект!



Две ссылки...

В данном случае оба вызова `request_date` поступают к одному объекту. В первый раз объект сообщает о своей доступности, но второй запрос отклоняется.

```
betty.request_date
candace.request_date
```

Второй запрос к ТОМУ ЖЕ объекту!

```
Sure, let's go!
Sorry, I'm busy.
```

Поведение программы при наложении имен выглядит *очень* знакомо... Помните программу составления каталога звезд? Давайте вернемся и присмотримся к ней повнимательнее.

Упражнение

Это класс Ruby:

```
class Counter

  def initialize
    @count = 0
  end

  def increment
    @count += 1
    puts @count
  end

end
```

А вот код, в котором этот класс используется:

```
a = Counter.new
b = Counter.new
c = b
d = c

a.increment
b.increment
c.increment
d.increment
```

Попробуйте предположить, что выведет этот код, и запишите свой ответ.

(Мы заполнили первое поле за вас.)

1 .....
.....
.....
.....

Упражнение
Решение

Это класс Ruby:

```
class Counter

  def initialize
    @count = 0
  end

  def increment
    @count += 1
    puts @count
  end

end
```

А вот код, в котором этот класс используется:

```
a = Counter.new
b = Counter.new
c = b
d = c

a.increment
b.increment
c.increment
d.increment
```

Попробуйте предположить, что выведет этот код, и запишите свой ответ.

1
1
2
3

Исправление ошибки в программе астронома

После знакомства с наложением имен мы снова вернемся к неправильно работающему каталогу звезд и посмотрим, удастся ли нам вычислить проблему на этот раз.

```
class CelestialBody
  attr_accessor :type, :name
end

altair = CelestialBody.new
altair.name = 'Altair'
altair.type = 'star'
polaris = altair
polaris.name = 'Polaris'
vega = polaris
vega.name = 'Vega'

puts altair.name, polaris.name, vega.name
```

Для экономии времени скопируем предыдущую звезду...

...и просто изменим ее имя.

То же самое здесь.

```
Vega
Vega
Vega
```

Странно, имена всех трех звезд теперь стали одинаковыми!

Если вызвать `object_id` для объектов в трех переменных, вы увидите, что все три переменные ссылаются на *один и тот же объект*. Один объект под тремя разными именами... типичный случай наложения имен!

```
puts altair.object_id
puts polaris.object_id
puts vega.object_id
```

```
70189936850940
70189936850940
70189936850940
```

Один и тот же объект!

Исправление ошибки в программе астронома (продолжение)

Копируя содержимое переменных, астроном *не получает* три разных экземпляра `CelestialBody`, как было задумано. Вместо этого он стал жертвой непреднамеренного наложения имен — он получил *один экземпляр* `CelestialBody` с *тремя* ссылками на него!

Сохраняет ссылку на новый экземпляр CelestialBody.

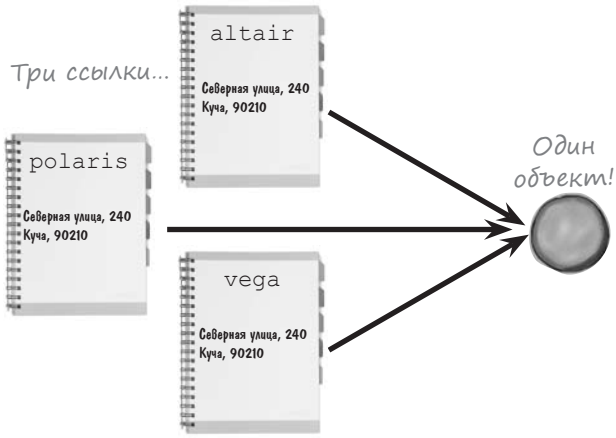
ТА ЖЕ ссылка копируется в новую переменную!

Та же ссылка копируется в ТРЕТЬЮ переменную!

```
altair = CelestialBody.new
altair.name = 'Altair'
altair.type = 'star'
polaris = altair
polaris.name = 'Polaris'
vega = polaris
vega.name = 'Vega'

puts altair.name, polaris.name, vega.name
```

Для несчастного, сбитого с толку объекта последовательность действий выглядит так:

1. «Присвоить атрибуту `name` значение `'Altair'`, а атрибуту `type` присвоить `'star'`».
2. «Теперь присвоить `name` значение `'Polaris'`».
3. «Теперь присвоить `name` значение `'Vega'`».
4. «Сообщи нам значение своего атрибута `name` 3 раза».



Класс `CelestialBody` послушно выполнил приказ и трижды сообщил, что его атрибут `name` содержит строку `Vega`.

```
Vega
Vega
Vega
```

К счастью, проблема решается легко. Нужно лишь отказаться от «оптимизаций» и создать *три* экземпляра `CelestialBody`.

Создаем первый экземпляр CelestialBody.

Вместо копирования ссылки получаем ссылку на второй экземпляр CelestialBody.

Тип должен задаваться для каждого объекта по отдельности.

Получаем ссылку на третий объект.

```
altair = CelestialBody.new
altair.name = 'Altair'
altair.type = 'star'
polaris = CelestialBody.new
polaris.name = 'Polaris'
polaris.type = 'star'
vega = CelestialBody.new
vega.name = 'Vega'
vega.type = 'star'

puts altair.name, polaris.name, vega.name
```

```
Altair
Polaris
Vega
```

Как видно из приведенного результата, проблема решена!

Копирования ссылок из переменной в переменную определенно стоит избегать. Но как вы вскоре увидите, существуют и другие обстоятельства, в которых необходимо понимать суть наложения имен.

Быстрая идентификация объектов методом «inspect»

Прежде чем двигаться дальше, необходимо упомянуть один полезный прием идентификации объектов. Мы уже показали, как использовать метод экземпляра `object_id`. Если он выводит одинаковые значения для объектов в двух переменных, значит, эти переменные указывают на один объект.

```
altair = CelestialBody.new
altair.name = 'Altair'
altair.type = 'star'
polaris = altair
polaris.name = 'Polaris'

puts altair.object_id, polaris.object_id
```

Копирует ТУ ЖЕ ссылку в новую переменную!

```
70350315190400
70350315190400
```

ОДИН И ТОТ ЖЕ объект!

Строка, возвращаемая методом экземпляра `inspect`, также включает представление идентификатора объекта в шестнадцатеричной системе счисления (цифры от 0 до 9 и буквы от *a* до *f*). Понимать шестнадцатеричную запись во всех подробностях не обязательно; просто запомните, что если для объектов, на который ссылаются две переменные, выводятся *одинаковые* значения, то они содержат два альтернативных имени для *одного объекта*. Если *значения разные*, значит, речь идет о *разных объектах*.

```
puts altair.inspect, polaris.inspect

vega = CelestialObject.new
puts vega.inspect
```

Шестнадцатеричное представление идентификатора объекта.

ОДИН И ТОТ ЖЕ объект!

Другой объект.

```
#<CelestialBody:0x007ff76b17f100 @name="Polaris", @type="star">
#<CelestialBody:0x007ff76b17f100 @name="Polaris", @type="star">
#<CelestialBody:0x007ff76b17edb8>
```

Проблемы с объектом по умолчанию для хеша

Астроном вернулся, у него снова возникли проблемы в коде...

Он хочет, чтобы в хеше хранилась информация о планетах и спутниках. Так как большинство объектов составляют планеты, он назначает для `CelestialBody` объект по умолчанию с атрибутом `type`, содержащим `"planet"`. (Объекты по умолчанию для хешей были представлены в предыдущей главе; они позволяют задать объект, который будет возвращаться хешем при обращении к ключам, которым еще не было присвоено значение.)

```
class CelestialBody
  attr_accessor :type, :name
end
```

Создание объекта планеты.

```
default_body = CelestialBody.new
default_body.type = 'planet'
bodies = Hash.new(default_body)
```

Объект создается с присвоением значений по умолчанию для всех неинициализированных ключей хеша.

Он полагает, что это позволит ему добавлять планеты в хеш, просто указывая их имена. И вроде бы на первый взгляд все работает:

```
bodies['Mars'].name = 'Mars'
p bodies['Mars']
```

CelestialBody с правильным атрибутом type...

```
#<CelestialBody:0x007fc60d13e6f8 @type="planet", @name="Mars">
```

Когда астроному потребуется добавить в хеш спутник, он может сделать и это. Для этого ему достаточно задать атрибут `type` наряду с атрибутом `name`.

```
bodies['Europa'].name = 'Europa'
bodies['Europa'].type = 'moon'

p bodies['Europa']
```

CelestialBody с типом «moon»

```
#<CelestialBody:0x007fc60d13e6f8 @type="moon", @name="Europa">
```

Но потом, когда он начинает добавлять в хеш новые объекты `CelestialBody`, начинаются странности...

Проблемы с объектом по умолчанию для хеша (продолжение)

Проблема с использованием `CelestialBody` в качестве объекта по умолчанию для хеша проявляется тогда, когда астроном пытается добавить в хеш новые объекты. Когда после спутника добавляется новая планета, ее атрибуту `type` тоже присваивается значение `"moon"`!

```
bodies['Venus'].name = 'Venus'

p bodies['Venus']
```

А должно быть planet. Откуда взялось «moon»?!

```
#<CelestialBody:0x007fc60d13e6f8 @type="moon", @name="Venus">
```

Но если вернуться и прочитать значения для ключей, добавленных ранее, то *эти* объекты тоже выглядят измененными!

```
p bodies['Mars']
p bodies['Europa']
```

Разве один из этих атрибутов не должен содержать «planet»?

Куда делись имена «Mars» и «Europa»?

```
#<CelestialBody:0x007fc60d13e6f8 @type="moon", @name="Venus">
#<CelestialBody:0x007fc60d13e6f8 @type="moon", @name="Venus">
```

Верно подмечено! Помните, что мы говорили ранее — о том, что результат метода `inspect` включает представление идентификатора объекта? И как вы знаете, метод `p` вызывает `inspect` для каждого объекта перед тем, как выводить его. Вызов метода `p` показывает, что все ключи хеша указывают на *один и тот же* объект!

```
p bodies['Venus']
p bodies['Mars']
p bodies['Europa']
```

Везде ОДИН И ТОТ ЖЕ объект!

```
#<CelestialBody:0x007fc60d13e6f8 @type="moon", @name="Venus">
#<CelestialBody:0x007fc60d13e6f8 @type="moon", @name="Venus">
#<CelestialBody:0x007fc60d13e6f8 @type="moon", @name="Venus">
```

Похоже, мы снова имеем дело с проблемой наложения имен! На нескольких ближайших страницах вы узнаете, как ее исправить.

На самом деле изменяется объект хеша по умолчанию!

Главная проблема с этим кодом заключается в том, что мы не изменяем значения в хеше. Вместо этого изменяется *объект по умолчанию для хеша*.

В этом можно убедиться при помощи метода экземпляра `default`, поддерживаемого любым хешем. Метод выдает информацию и об объекте по умолчанию после создания хеша.

Проверим объект по умолчанию до и после добавления планеты в хеш.

```
class CelestialBody
  attr_accessor :type, :name
end

default_body = CelestialBody.new
default_body.type = 'planet'
bodies = Hash.new(default_body)

p bodies.default ← Проверяем объект по умолчанию.

bodies['Mars'].name = 'Mars' ← Пытаемся добавить значение в хеш.

p bodies.default ← Снова проверяем объект по умолчанию.
```

Объект по умолчанию ПЕРЕД попыткой добавления значения в хеш.

Объект по умолчанию ПОСЛЕ попытки добавления значения в хеш.

```
#<CelestialBody:0x007f868a8274c8 @type="planet">
#<CelestialBody:0x007f868a8274c8 @type="planet", @name="Mars">
```

Имя почему-то добавилось в объект по умолчанию!

Итак, почему же имя было добавлено в объект по умолчанию? Разве оно не должно было стать значением элемента `bodies['Mars']`?

Если проверить идентификаторы объектов для `bodies['Mars']` и объекта по умолчанию для хеша, ответ приходит сам собой:

```
p bodies['Mars']
p bodies.default
```

Одинаковые идентификаторы объекта!

ОДИН И ТОТ ЖЕ объект!

```
#<CelestialBody:0x007f868a8274c8 @type="planet", @name="Mars">
#<CelestialBody:0x007f868a8274c8 @type="planet", @name="Mars">
```

При обращении к `bodies['Mars']` мы по-прежнему получаем ссылку на объект хеша по умолчанию! Но почему?

Объект по умолчанию для хеша: близкое знакомство

Когда в последней главе мы представили объект по умолчанию для хеша, мы сказали, что объект по умолчанию возвращается при каждом обращении к ключу, *которому еще не было присвоено значение*. Последнее обстоятельство заслуживает более внимательного рассмотрения.

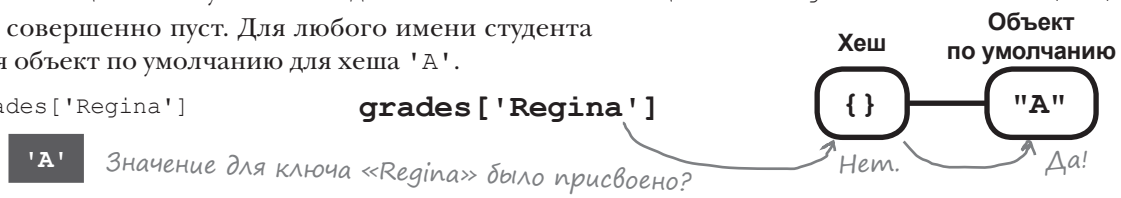
Предположим, мы создали хеш, в котором ключами являются имена студентов, а значениями — их оценки. По умолчанию должна использоваться оценка `'A'`.

Создание нового хеша с объектом по умолчанию.

```
grades = Hash.new('A')
```

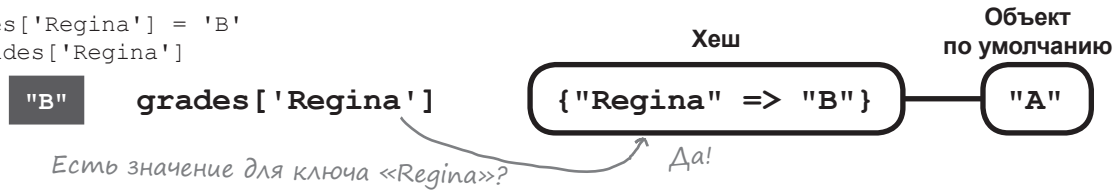
Сначала хеш совершенно пуст. Для любого имени студента возвращается объект по умолчанию для хеша `'A'`.

```
p grades['Regina']
```



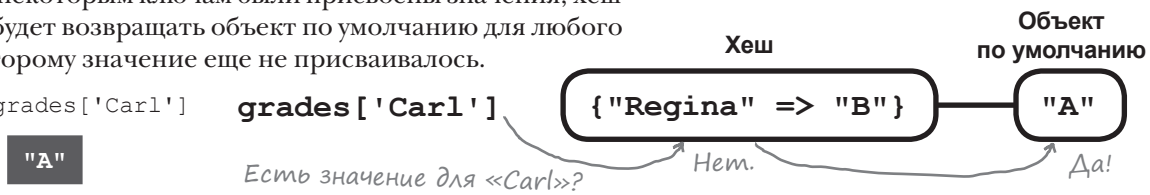
Если для ключа хеша было присвоено значение, при следующей попытке обращения к нему мы получаем это значение (вместо объекта по умолчанию).

```
grades['Regina'] = 'B'
p grades['Regina']
```



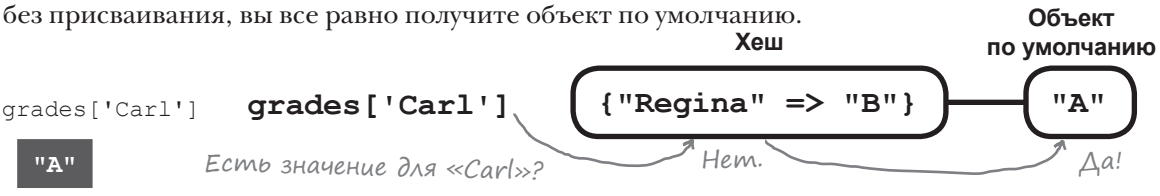
Даже если некоторым ключам были присвоены значения, хеш все равно будет возвращать объект по умолчанию для любого ключа, которому значение еще не присваивалось.

```
p grades['Carl']
```



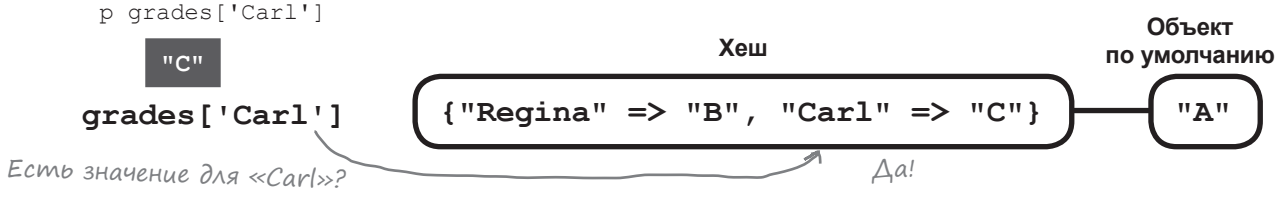
Но *обращение* к значению в хеше — не то же самое, что *присваивание* этого значения. Если вы обратитесь к значению из хеша, а потом обратитесь к нему повторно без присваивания, вы все равно получите объект по умолчанию.

```
p grades['Carl']
```



Только когда значение в хеше *записывается* в хеш (а не просто *читается* из него), возвращается нечто отличное от объекта по умолчанию.

```
grades['Carl'] = 'C'
p grades['Carl']
```



Возвращаемся к хешу с планетами и спутниками

Именно *поэтому* при попытке задать атрибуты `type` и `name` для объектов в хеше с планетами и спутниками мы в конечном итоге изменяем объект по умолчанию. Никакие значения в хеше не присваиваются. Более того, если вы проверите сам хеш, то увидите, что он совершенно пуст!

```
class CelestialBody
  attr_accessor :type, :name
end

default_body = CelestialBody.new
default_body.type = 'planet'
bodies = Hash.new(default_body)

bodies['Mars'].name = 'Mars'
bodies['Europa'].name = 'Europa'
bodies['Europa'].type = 'moon'
bodies['Venus'].name = 'Venus'

p bodies
```

`{}` ← Пустой!

Я думала, что мы **присваиваем** значения в хеше. Ведь это команды присваивания, верно?

Разве это не присваивание в хеше?

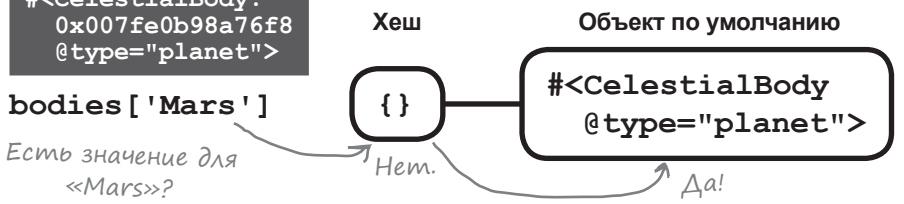
```
bodies['Mars'].name = 'Mars'
bodies['Europa'].name = 'Europa'
bodies['Europa'].type = 'moon'
bodies['Venus'].name = 'Venus'
```

На самом деле это вызовы методов записи атрибутов `name=` и `type=` объекта по умолчанию для хеша. Не путайте их с присваиванием!

При обращении к ключу, с которым не было связано никакое значение, мы получаем объект по умолчанию.

```
default_body = CelestialBody.new
default_body.type = 'planet'
bodies = Hash.new(default_body)

p bodies['Mars']
```

```
#<CelestialBody:
  0x007fe0b98a76f8
  @type="planet">
```

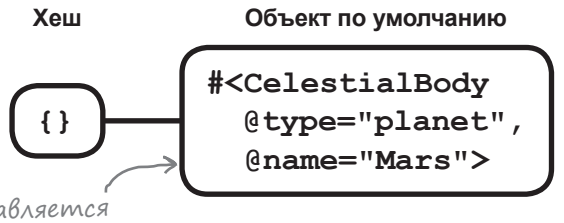


Следующая команда *не является* присваиванием в хеше. Она пытается *обратиться* к значению, связанному с ключом `'Mars'` в хеше (которое до сих пор пусто). Так как значение для ключа `'Mars'` отсутствует, возвращается ссылка на объект по умолчанию, *который затем модифицируется.*

```
bodies['Mars'].name = 'Mars'
```

Обращение к объекту по умолчанию

Изменение объекта по умолчанию



А поскольку значение в хеше *еще* не присвоено, при *следующем* обращении также будет получена ссылка на объект по умолчанию... и так далее.

К счастью, проблему можно решить...

Чего мы хотим от объектов по умолчанию для хеша

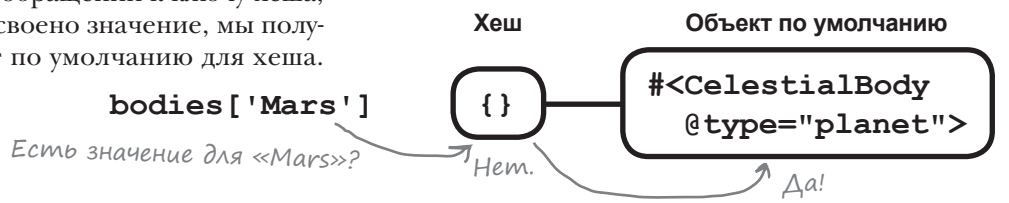
Мы определили, что этот код *не присваивает* значение в хеше, а просто *обращается* к значению. Он получает ссылку на объект по умолчанию, который затем (непреднамеренно) изменяется.

```
default_body = CelestialBody.new
default_body.type = 'planet'
bodies = Hash.new(default_body)

bodies['Mars'].name = 'Mars'
```

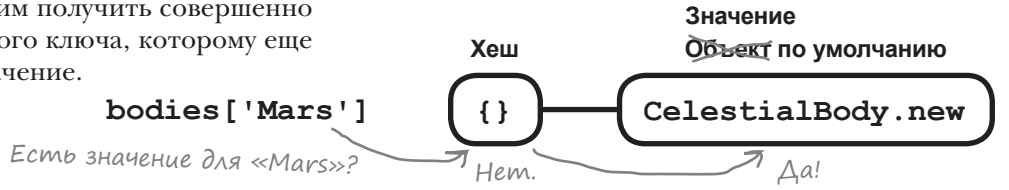
Получаем ссылку на объект по умолчанию.

Изменяет объект по умолчанию!

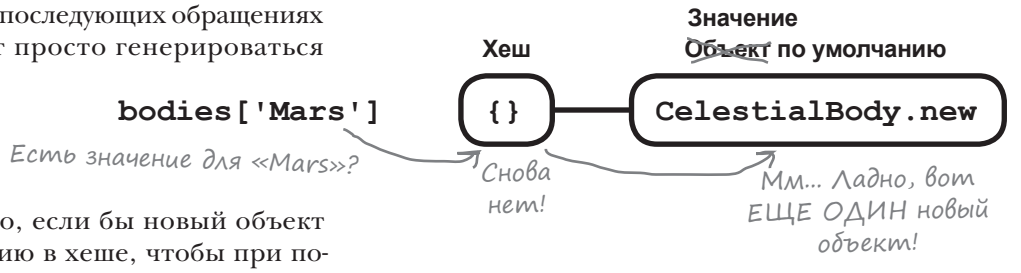
В текущей версии при обращении к ключу хеша, которому не было присвоено значение, мы получаем ссылку на объект по умолчанию для хеша.



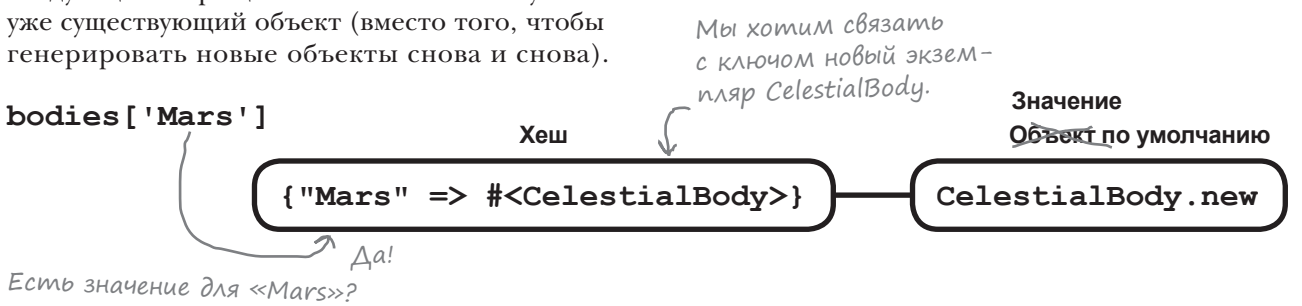
А *на самом деле* мы хотим получить совершенно *новый* объект для каждого ключа, которому еще не было присвоено значение.



Конечно, если это будет происходить *без* присваивания в хеше, то при последующих обращениях новые объекты будут просто генерироваться снова и снова...



Итак, было бы удобно, если бы новый объект присваивался значению в хеше, чтобы при последующих обращениях мы снова получали уже существующий объект (вместо того, чтобы генерировать новые объекты снова и снова).



К счастью, у хешей есть одна возможность, которая способна сделать *все* за нас!

Блоки по умолчанию для хешей

Вместо того чтобы передавать при вызове `Hash.new` *объект* по умолчанию для хеша, можно передать `Hash.new` *блок*, который должен использоваться как *блок* по умолчанию для хеша. При обращении к ключу, которому еще не было присвоено значение:

- Вызывается блок.
- Блок получает ссылки на хеш и текущий ключ в параметрах блока. Эти ссылки могут использоваться для присваивания значения в хеше.
- Возвращаемое значение блока возвращается как текущее значение, присвоенное ключу хеша.

Эти правила могут показаться сложными, поэтому сейчас мы их разберем более подробно. А пока просто взгляните на ваш первый блок по умолчанию для хеша:

При последующем вызове блок получает ссылку на хеш и на ключ, к которому происходит обращение.

Создаем хеш.

```
bodies = Hash.new do |hash, key|
  body = CelestialBody.new
  body.type = "planet"
  hash[key] = body
  body
end
```

Здесь создается объект, который станет значением для этого ключа.

Присваиваем объект текущему ключу хеша.

Возвращаем объект.

Обращаясь к ключам *этого* хеша, мы будем получать разные объекты для разных ключей — как, собственно, и предполагалось.

Этот код идентичен тому, который уже встречался нам пару страниц назад!

```
bodies['Mars'].name = 'Mars'
bodies['Europa'].name = 'Europa'
bodies['Europa'].type = 'moon'
bodies['Venus'].name = 'Venus'

p bodies['Mars']
p bodies['Europa']
p bodies['Venus']
```

Три разных объекта.

bodies['Mars']
bodies['Europa']
bodies['Venus']

```
#<CelestialBody:0x007fe701896580 @type="planet", @name="Mars">
#<CelestialBody:0x007fe7018964b8 @type="moon", @name="Europa">
#<CelestialBody:0x007fe7018963a0 @type="planet", @name="Venus">
```

Что еще лучше — при первом обращении к ключу значение автоматически заносится в хеш!

Значения были присвоены в хеше!

```
p bodies
{"Mars"=>#<CelestialBody:0x007fe701896580 @type="planet", @name="Mars">,
 "Europa"=>#<CelestialBody:0x007fe7018964b8 @type="moon", @name="Europa">,
 "Venus"=>#<CelestialBody:0x007fe7018963a0 @type="planet", @name="Venus">}
```

Теперь, когда мы знаем, что это работает, присмотримся к содержимому блока повнимательнее...

Блоки по умолчанию для хеша: присваивание

В большинстве случаев знание, созданное блоком по умолчанию для хеша, должно присваиваться в хеше. Для этого блоку передается ссылка на хеш и текущий ключ.

При последующем вызове блок получает ссылку на хеш и ключ, по которому происходит обращение.

```
bodies = Hash.new do |hash, key|
  body = CelestialBody.new
  body.type = "planet"
  hash[key] = body
  body
end
```

Объект присваивается текущему ключу хеша.

Когда значения в хеше присваиваются в теле блока, все работает примерно так, как мы и ожидали. Новый объект генерируется для каждого нового ключа, к которому вы обращаетесь. При последующих обращениях вы снова получаете тот же объект, в котором сохранены все внесенные изменения.

Генерирует новый объект.

Возвращает объект из предыдущей строки.

Вносимые изменения сохраняются в хеше..

```
p bodies['Europa']
p bodies['Europa']
bodies['Europa'].type = 'moon'
p bodies['Europa']
```

Один и тот же объект.

Атрибут type не теряется.

```
#<CelestialBody:0x007fb6389eed00 @type="planet">
#<CelestialBody:0x007fb6389eed00 @type="planet">
#<CelestialBody:0x007fb6389eed00 @type="moon">
```

Будьте осторожны!

Не забудьте присвоить значение в хеше!

Если вы забудете это сделать, то сгенерированное значение будет потеряно. С ключом хеша по-прежнему не будет связано значение, а хеш будет вызывать блок снова и снова, чтобы генерировать новые объекты по умолчанию.

Здесь НЕОБХОДИМО выполнить присваивание в хеше. Если этого не сделать...

...вы будете получать новый объект при каждом обращении к этому ключу!

Внесенные изменения будут потеряны!

```
bodies = Hash.new do |hash, key|
  body = CelestialBody.new
  body.type = "planet"
  body
end

p bodies['Europa']
p bodies['Europa']
bodies['Europa'].type = 'moon'
p bodies['Europa']
```

Три разных объекта!

Атрибут type по-прежнему содержит значение по умолчанию!

```
#<CelestialBody:0x007ff95507ee90 @type="planet">
#<CelestialBody:0x007ff95507ecd8 @type="planet">
#<CelestialBody:0x007ff95507eaf8 @type="planet">
```

Блоки по умолчанию для хеша: возвращаемое значение блока

При первом обращении к ключу, которому не было присвоено значение, возвращаемое значение блока по умолчанию возвращается как значение для данного ключа.

```
bodies = Hash.new do |hash, key|
  body = CelestialBody.new
  body.type = "planet"
  hash[key] = body
  body
end

p bodies['Mars']
```

Это возвращаемое значение...

...будет получено здесь!

```
#<CelestialBody:0x007fef7a9132c0 @type="planet">
```

Если в теле блока ключу присваивается значение, то блок по умолчанию не будет вызываться при последующих обращениях к этому ключу; вместо этого вы будете получать присвоенное значение.

Будьте осторожны!

Проследите за тем, чтобы возвращаемое значение блока соответствовало значению, которое присваивается в хеше!

В противном случае при первом обращении к ключу вы получите одно значение, а при последующих обращениях — совершенно иные значения.

```
bodies = Hash.new do |hash, key|
  body = CelestialBody.new
  body.type = "planet"
  hash[key] = body
  "I'm a little teapot"
end

p bodies['Mars']
p bodies['Mars']
```

Значение, возвращаемое из блока.

Значение, присвоенное в хеше.

```
"I'm a little teapot"
#<CelestialBody:0x007fcf830ff000 @type="planet">
```

Честно говоря, вам не придется сильно трудиться над тем, чтобы запомнить это правило. Как вы увидите на следующей странице, назначение правильного возвращаемого значения для блока по умолчанию для хеша происходит вполне естественно.

Блоки по умолчанию для хешей: сокращенная запись

До сих пор мы возвращали значение из блока по умолчанию в отдельной строке:

```
bodies = Hash.new do |hash, key|
  body = CelestialBody.new
  body.type = "planet"
  hash[key] = body
  body                    # Отдельное возвращаемое значение блока.
end

p bodies['Mars']
```

```
#<CelestialBody:0x007fef7a9132c0 @type="planet">
```

Однако в Ruby существует сокращенная запись, которая немного сокращает объем кода в блоке по умолчанию.

Вы уже знаете, что значение последнего выражения в блоке интерпретируется как возвращаемое значение блока... Однако мы *не упоминали* о том, что в Ruby значение выражения присваивания совпадает с присваиваемым значением.

```
p my_hash = {}
p my_array = []
p my_integer = 20
p my_hash['A'] = ['Apple']
p my_array[0] = 245
```

```
{}
[]
20
["Apple"]
245
```

Значения этих выражений совпадают с присваиваемыми значениями.

Следовательно, мы можем разместить отдельную команду присваивания в блоке по умолчанию для хеша, и она вернет присвоенное значение.

```
greetings = Hash.new do |hash, key|
  hash[key] = "Hi, #{key}"
end

p greetings["Kayla"]
```

```
"Hi, Kayla"
```

И конечно, значение при этом также будет добавлено в хеш.

```
p greetings
```

```
{"Kayla"=>"Hi, Kayla"}
```

Итак, в случае с хешем астронома вместо того, чтобы добавлять отдельную строку с возвращаемым значением, можно просто вернуть значение выражения присваивания как возвращаемое значение блока.

```
bodies = Hash.new do |hash, key|
  body = CelestialBody.new
  body.type = "planet"
  hash[key] = body        # Становится возвращаемым значением блока.
end

p bodies['Mars']
```

```
#<CelestialBody:0x007fa769a3f2d8 @type="planet">
```

Все три приведенных ниже фрагмента *должны* создавать хеш с названиями продуктов, сгруппированными по первой букве названия, но работает из них только один. Свяжите каждый фрагмент с тем результатом, который он выдает.

(Мы записали один ответ за вас.)

A

```
foods = Hash.new([])
foods['A'] << "Apple"
foods['A'] << "Avocado"
foods['B'] << "Bacon"
foods['B'] << "Bread"
p foods['A']
p foods['B']
p foods
```

B

```
foods = Hash.new { |hash, key| [] }
foods['A'] << "Apple"
foods['A'] << "Avocado"
foods['B'] << "Bacon"
foods['B'] << "Bread"
p foods['A']
p foods['B']
p foods
```

C

```
foods = Hash.new { |hash, key| hash[key] = [] }
foods['A'] << "Apple"
foods['A'] << "Avocado"
foods['B'] << "Bacon"
foods['B'] << "Bread"
p foods['A']
p foods['B']
p foods
```

.....

```
[]
[]
{}
```

A .....

```
["Apple", "Avocado", "Bacon", "Bread"]
["Apple", "Avocado", "Bacon", "Bread"]
{}
```

.....

```
["Apple", "Avocado"]
["Bacon", "Bread"]
{"A"=>["Apple", "Avocado"], "B"=>["Bacon", "Bread"]}
```

Упражнение
Решение

Все три приведенных ниже фрагмента ***должны*** создавать хеш с названиями продуктов, сгруппированными по первой букве названия, но работает из них только один. Свяжите каждый фрагмент с тем результатом, который он выдает.

A

```
foods = Hash.new([])
foods['A'] << "Apple"
foods['A'] << "Avocado"
foods['B'] << "Bacon"
foods['B'] << "Bread"
p foods['A']
p foods['B']
p foods
```

Этот ЕДИНСТВЕННЫЙ массив будет использоваться как значение по умолчанию для всех ключей хеша!

Все эти значения будут добавлены в ОДИН массив!

B

```
foods = Hash.new { |hash, key| [] }
foods['A'] << "Apple"
foods['A'] << "Avocado"
foods['B'] << "Bacon"
foods['B'] << "Bread"
p foods['A']
p foods['B']
p foods
```

Возвращает новый, пустой массив при каждом вызове блока, но не добавляет его в хеш!

Каждая строка добавляется в новый массив. А потом этот массив пропадает!

C

```
foods = Hash.new { |hash, key| hash[key] = [] }
foods['A'] << "Apple"
foods['A'] << "Avocado"
foods['B'] << "Bacon"
foods['B'] << "Bread"
p foods['A']
p foods['B']
p foods
```

Присваивает новый массив в хеше под текущим ключом.

Добавляется в новый массив.

Добавляется в тот же массив, что и «Apple».

Добавляется в новый массив.

Добавляется в тот же массив, что и «Bacon».

B

```
[]
[]
{}
```

A

```
["Apple", "Avocado", "Bacon", "Bread"]
["Apple", "Avocado", "Bacon", "Bread"]
{}
```

C

```
["Apple", "Avocado"]
["Bacon", "Bread"]
{"A"=>["Apple", "Avocado"], "B"=>["Bacon", "Bread"]}
```

Хеш астронома: окончательная версия кода

Окончательная версия кода с блоком по умолчанию для хеша выглядит так:

```
class CelestialBody
  attr_accessor :type, :name
end

bodies = Hash.new do |hash, key|
  body = CelestialBody.new
  body.type = "planet"
  hash[key] = body
end

bodies['Mars'].name = 'Mars'
bodies['Europa'].name = 'Europa'
bodies['Europa'].type = 'moon'
bodies['Venus'].name = 'Venus'

p bodies
```

Получает ссылку на хеш и текущий ключ.

Создает новый объект для текущего ключа.

Выполняет присваивание в хеше. И возвращает новое значение.

Теперь все эти строки работают так, как было задумано!

По умолчанию type содержит значение «planet», но может переопределяться.

Каждое значение в хеше — отдельный объект.

Все имена сохраняются.

(Результаты выровнены для удобства чтения.)

```
{"Mars"  =>#<CelestialBody:0x007fcde388aaa0 @type="planet", @name="Mars"  >,
 "Europa"=>#<CelestialBody:0x007fcde388a9d8 @type="moon",   @name="Europa">,
 "Venus" =>#<CelestialBody:0x007fcde388a8c0 @type="planet", @name="Venus" >}
```

Основные моменты в работе этой программы:

- Блок по умолчанию для хеша используется для создания *уникального* объекта для каждого ключа хеша. (Этим он отличается от объекта по умолчанию для хеша, который возвращает ссылки на *один* объект для *всех* ключей.)
- В блоке новый объект присваивается текущему ключу хеша.
- Новый объект становится значением выражения присваивания, которое в свою очередь становится возвращаемым значением блока. Итак, при первом обращении к некоторому ключу хеша новый объект возвращается как соответствующее значение.

Безопасное использование объектов по умолчанию

У меня еще один вопрос. Зачем вообще использовать объекты по умолчанию для хешей, если есть блоки по умолчанию?

Объекты по умолчанию для хешей очень хорошо подходят для чисел.

Ограничиться числами? Тогда зачем Ruby позволяет использовать `CelestialBody` как объект по умолчанию, не выдавая даже предупреждения?

Ладно, мы немного упрощаем. Объекты хешей по умолчанию отлично работают, если вы не изменяете объект по умолчанию и если значения присваиваются в хеше. Просто с числами эти правила особенно легко выполняются.

Рассмотрим пример с подсчетом количества вхождений букв в массиве (по тому же принципу, что и в примере с подсчетом голосов в предыдущей главе).

```
letters = ['a', 'c', 'a', 'b', 'c', 'a']

counts = Hash.new(0)

letters.each do |letter|
  counts[letter] += 1
end

p counts
```

Если это значение еще не присвоено, получает объект по умолчанию для хеша, но НЕ изменяет его.

Увеличенное значение снова присваивается в хеше.

```
{"a"=>3, "c"=>2, "b"=>1}
```

Объект по умолчанию для хеша в данном случае работает, потому что выполняются два приведенных выше правила.

Правило №1: Не изменять объект по умолчанию

Если вы собираетесь использовать объект по умолчанию для хеша, очень важно не изменять этот объект в программе. В противном случае при следующем обращении к объекту по умолчанию вы получите неожиданные результаты. Пример такого рода уже встречался нам при использовании объекта по умолчанию (вместо блока по умолчанию) в хеше астронома, когда это породило настоящий хаос:

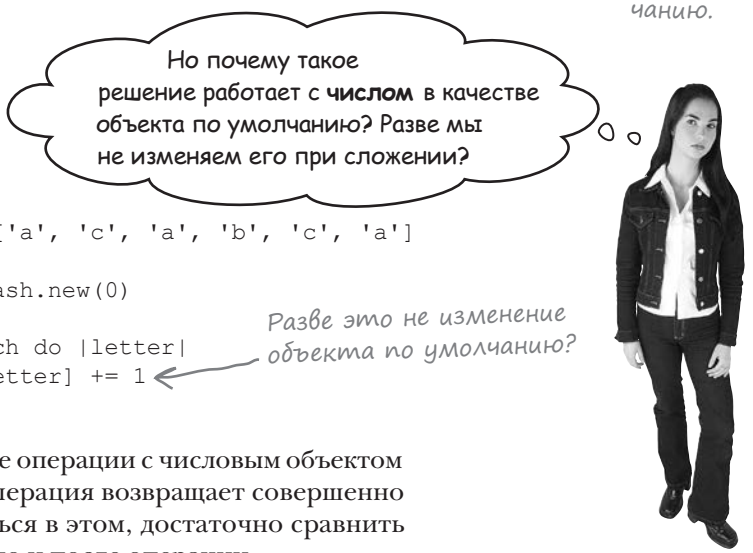
```
default_body = CelestialBody.new
default_body.type = 'planet'
bodies = Hash.new(default_body)
```

Назначаем объект по умолчанию для хеша.

```
bodies['Mars'].name = 'Mars'
```

Получаем ссылку на объект по умолчанию.

Изменяем объект по умолчанию.



В языке Ruby математические операции с числовым объектом не изменяют этот объект; операция возвращает совершенно *новый* объект. Чтобы убедиться в этом, достаточно сравнить идентификаторы объектов до и после операции.

```
number = 0
puts number.object_id
number = number + 1
puts number.object_id
```

```
1
3
```

Два разных объекта! (Идентификаторы объектов для целых чисел намного меньше, чем для других объектов, но это подробность реализации, так что не обращайте внимания. Здесь важно то, что идентификаторы различны.)

Числовые объекты обладают свойством *неизменности*: у них *нет* методов, изменяющих состояние объекта. Любая операция, способная изменить число, возвращает совершенно новый объект.

Именно благодаря этому обстоятельству числа можно безопасно использовать в качестве объектов по умолчанию для хеша; вы можете быть твердо уверены в том, что число по умолчанию не будет случайно изменено.

Числа хорошо подходят на роль объектов по умолчанию для хеша, потому что они неизменны.

Правило №2: Присваивать значения в хеше

Если вы собираетесь использовать объект по умолчанию для хеша, также необходимо проследить за тем, чтобы в хеше действительно присваивались значения. Как вы видели в примере с хешем астронома, иногда создается впечатление, что программа выполняет присваивание в хеше, хотя на самом деле это не так.

```
default_body = CelestialBody.new
default_body.type = 'planet'
bodies = Hash.new(default_body)

bodies['Mars'].name = 'Mars'

p bodies
```

```
{}
```

Вызов метода записи атрибута. НЕ приводит к присваиванию в хеше!

На самом деле хеш все еще остается пустым!

Но когда в качестве объекта по умолчанию используется *число*, присваивание значений в хеше происходит намного естественнее. (Так как числа обладают свойством неизменности, вы *не сможете* сохранить увеличенные значения *без* их присваивания в хеше!)

```
hash = Hash.new(0)

hash['a'] += 1
hash['c'] += 1

p hash.default
p hash
```

```
0
{"a"=>1, "c"=>1}
```

Объект по умолчанию для хеша не изменяется.

Происходит присваивание значений в хеше!

Объекты по умолчанию для хешей: простое правило

Что-то слишком много приходится всего запоминать для простой работы с хешами.

Да, вы правы. Поэтому у нас есть простое правило, которое избавит вас от лишних хлопот...

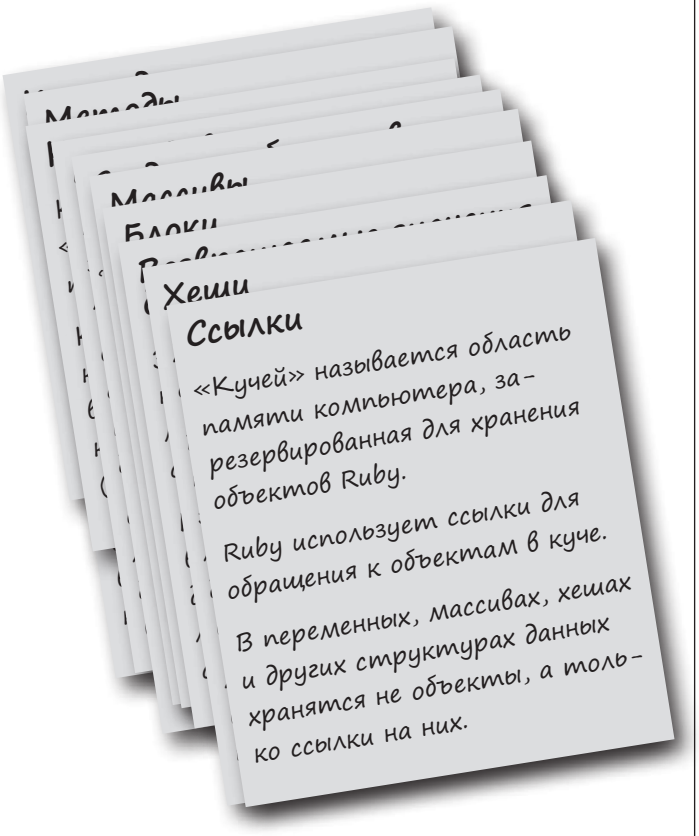
Если по умолчанию используется число, используйте объект по умолчанию для хеша. Если по умолчанию используется что-то другое, используйте блок по умолчанию для хеша.

По мере того как у вас появится практический опыт работы со ссылками, все это войдет у вас в привычку, и вы научитесь нарушать это правило — там, где это уместно. А до тех пор это правило предотвратит большинство проблем, с которыми вы можете столкнуться.

Понимание ссылок Ruby и проблема наложения имен не поможет вам писать более мощные программы на языке Ruby. Однако оно *поможет* быстро находить и решать проблемы в процессе диагностики. Будем надеяться, что с информацией, представленной в этой главе, вы поймете принцип работы ссылок и сможете обойти потенциальные проблемы еще до их возникновения.

Ваш инструментарий Ruby

Глава 8 подошла к концу, а ваш инструментарий пополнился ссылками.



Далее в программе…

В следующей главе мы вернемся к теме структуры и организации кода. Вы уже знаете, как определить общие методы классов посредством наследования. Но даже в тех ситуациях, в которых наследование неуместно, Ruby предоставляет механизм совместного использования поведения между классами: *примеси* (mixin). Эта тема будет представлена в следующей главе.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Если вам потребуется хранить больше объектов, Ruby автоматически увеличит размер кучи. Если объект больше не используется, Ruby удаляет его из хеша.
- Наложение имен возникает в результате копирования ссылки на объект. При непреднамеренном возникновении оно может создать ошибки в программе.
- У большинства объектов Ruby имеется метод экземпляра `object_id`, который возвращает уникальный идентификатор объекта. С его помощью можно выявить ситуацию с возникновением множественных ссылок на один объект.
- В строку, возвращаемую методом `inspect`, также включается представление идентификатора объекта.
- Если назначить для хеша объект по умолчанию, все неприсвоенные ключи хешей будут возвращать ссылки на один объект по умолчанию.
- По этой причине в качестве объекта по умолчанию для хеша лучше использовать только неизменные объекты (объекты, которые не могут изменяться в программе) — например, числа.
- Если вам потребуется использовать в качестве объекта по умолчанию для хеша другой объект, лучше воспользоваться блоком по умолчанию для хеша, чтобы для каждого ключа создавался уникальный объект.
- В параметрах блоки по умолчанию для хешей получают ссылку на хеш и текущий ключ. В большинстве случаев эти параметры будут использоваться для присваивания нового объекта как значения для заданного ключа хеша.
- Возвращаемое значение блока по умолчанию для хеша интерпретируется как исходное значение по умолчанию для заданного ключа.
- Таким образом, если выражение присваивания является последним выражением в блоке, то присвоенное значение становится возвращаемым значением блока.

9 примеси

Аккуратный выбор

У наследования есть свои ограничения. Наследовать методы можно только от одного класса. Но что, если вам понадобилось использовать *некоторое поведение* в совершенно разных классах? Представьте методы для запуска цикла зарядки и получения информации о текущем уровне заряда — такие методы могут использоваться телефонами, электродрелями и электромобилями. И вы готовы создать *один* суперкласс для всех *этих* устройств? (Не пытайтесь, ничем хорошим это не кончится.) Или методы запуска и остановки двигателя. Конечно, эти методы могут пригодиться дрели и автомобилю, но не телефону!

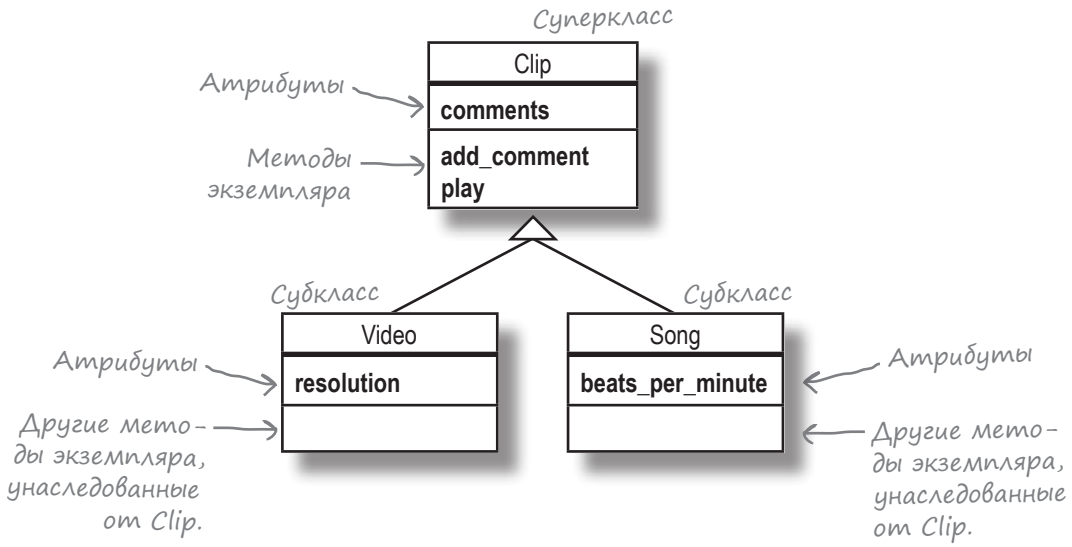
В этой главе рассматриваются **модули** и **примеси** — мощный механизм *группировки методов* и их последующего включения *только в те классы, для которых они актуальны*.

Мультимедийное приложение

К вам обратился очередной клиент. Он занимается разработкой приложения для организации общего доступа к видеороликам, музыке и другим мультимедийным материалам. Как для музыки, так и для видео необходима общая функциональность: пользователи должны иметь возможность воспроизводить песни и видеоролики, а также оставлять комментарии. (Для простоты мы опустили такие функции, как приостановка, перемотка и т. д.)

Впрочем, есть и различия. Для песен необходимо предусмотреть возможность хранения количества ударов в минуту, чтобы пользователь мог мгновенно отличить быструю музыку от медленной. Для видеороликов это не нужно, но *вместо этого* необходимо хранить разрешение (ширина и высота изображения в пикселах).

Так как в этой архитектуре данные (разрешение, количество ударов в минуту и т. д.) смешиваются с поведением (воспроизведение), будет разумно создать классы `Video` и `Song`. С другой стороны, часть поведения (воспроизведение и комментирование) является общей. До сих пор мы видели только один механизм совместного использования поведения между классами: наследование, поэтому можно сделать `Video` и `Song` субклассами суперкласса `Clip`. В классе `Clip` определяется метод чтения атрибута `comments`, а также методы экземпляра `play` и `add_comment`.



Приложение с наследованием

Ниже приведен код классов `Clip`, `Video` и `Song` и их тестирования. Атрибуты `resolution` и `beats_per_minute` достаточно тривиальны, поэтому основное внимание будет уделено методам `add_comment` и `comments`.

Суперкласс

```
class Clip

  attr_reader :comments

  def initialize
    @comments = []
  end

  def add_comment(comment)
    comments << comment
  end

  def play
    puts "Playing #{object_id}..."
  end

end
```

Определяем метод, возвращающий значение переменной экземпляра «@comments».

При создании нового экземпляра создается пустой массив для добавления комментариев.

Вызываем метод «comments» для получения массива из «@comments» для присоединения комментария.

Выводим идентификатор воспроизводимого объекта.

Субкласс

```
class Video < Clip
  attr_accessor :resolution
end
```

Субкласс

```
class Song < Clip
  attr_accessor :beats_per_minute
end
```

```
video = Video.new
video.add_comment("Cool slow motion effect!")
video.add_comment("Weird ending.")
song = Song.new
song.add_comment("Awesome beat.")

p video.comments, song.comments
```

Создаем новый объект Video.

Добавляем комментарии.

Создаем новый объект Song.

Добавляем комментарий.

Просмотр всех комментариев.

```
["Cool slow motion effect!", "Weird ending."]
["Awesome beat."]
```

Вроде бы все работает, притом неплохо! Но тут клиент выступает с неожиданной просьбой...

Один из этих классов не похож на другие

Ваш клиент хочет, чтобы на сайте была возможность добавления фотографий. Для фотографий, как и для видеороликов и музыки, пользователи должны иметь возможность добавлять комментарии. Но конечно, *в отличие* от музыки, фотографиям не нужен метод воспроизведения `play`; вместо него должен использоваться метод `show`.

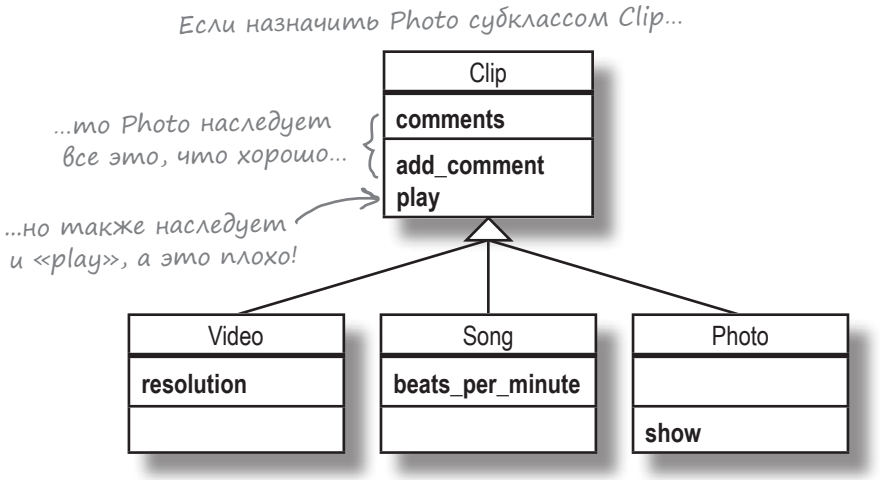
Итак, при создании класса `Photo` его экземпляры должны содержать метод доступа `comments` и метод `add_comment` — как и классы `Video` и `Song`, наследующие от суперкласса `Clip`.

Если ограничиться только тем, что мы узнали о классах Ruby до сих пор, можно рассмотреть пару возможных решений. К сожалению, у каждого из них есть свои недостатки...

Вариант 1: Определение Photo как субкласса Clip

Класс `Photo` *можно* определить как субкласс `Clip` и позволить ему унаследовать `comments` и `add_comment` так же, как во всех остальных субклассах.

Но у такого подхода есть недостаток: он (ошибочно) предполагает, что `Photo` является специализацией `Clip`. Если сделать класс `Photo` субклассом `Clip`, он унаследует методы `comments` и `add_comment`. При этом он *также* унаследует метод `play`.

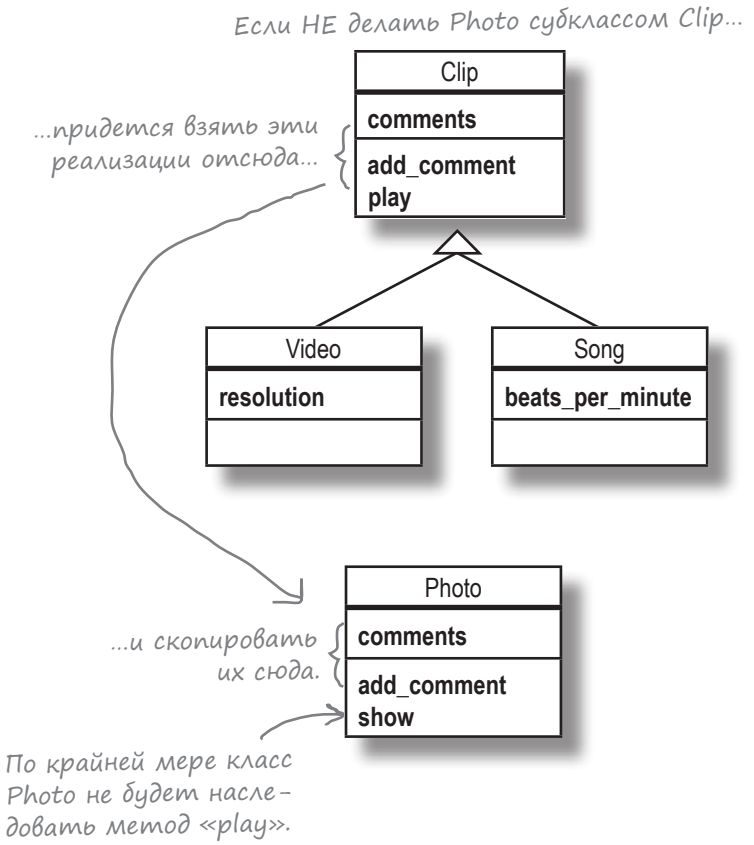


Но фотографию нельзя *воспроизвести* (play). Даже если вы переопределите метод `play` в субклассе `Photo`, чтобы избежать ошибок в программе, каждый разработчик, просматривающий ваш код в будущем, будет недоумевать, почему в классе `Photo` определен метод `play`.

Итак, определять субкласс `Clip` только для получения методов `comments` и `add_comment` было бы нежелательно.

Вариант 2: Копирование нужных методов в класс Photo

Второй вариант ненамного лучше: мы можем отказаться от определения `Photo` как субкласса и *заново* реализовать методы `comments` и `add_comment` в классе `Photo` (иначе говоря, скопировать код).



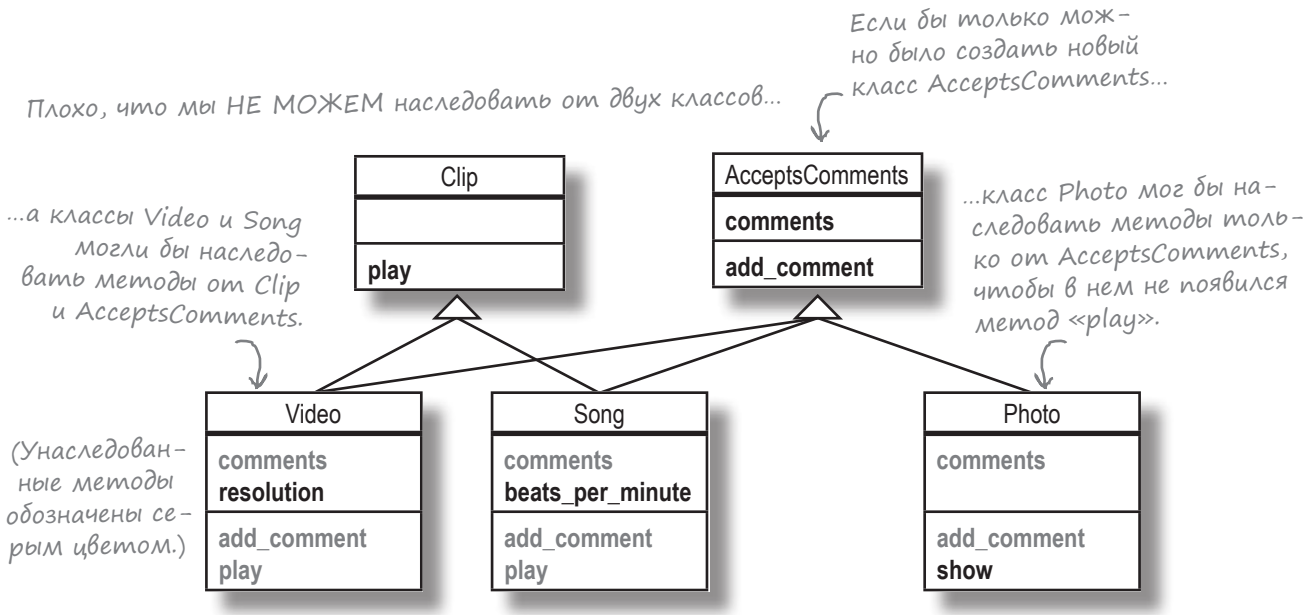
Впрочем, недостатки подобных решений уже были описаны в начале главы 3. Если нам потребуется внести изменения в методы `comments` или `add_comment` в классе `Photo`, нам также придется заняться изменением кода в классе `Clip`. В противном случае эти методы будут работать по-разному, а это может стать неприятным сюрпризом для будущих разработчиков, работающих с этими классами.

Выходит, что *и этот* вариант нельзя признать удовлетворительным.

Не вариант: Множественное наследование

Нам понадобится решение, которое позволит предоставить общий доступ к реализации методов `comments` и `add_comment` между классами `Video`, `Song` и `Photo`, но *без* наследования метода `play` классом `Photo`.

Было бы хорошо вынести методы `comments` и `add_comment` в отдельный суперкласс `AcceptsComments`, оставив метод `play` только в суперклассе `Clip`. В этом случае классы `Video` и `Song` смогут унаследовать метод `play` от класса `Clip` *и* унаследовать методы `comments` и `add_comment` от `AcceptsComments`. У класса `Photo` суперклассом будет только класс `AcceptsComments`, поэтому он наследует методы `comments` и `add_comment` *без* наследования `play`.



Придется вас разочаровать: это невозможно. Концепция наследования от более чем одного класса называется *множественным наследованием* и в Ruby этот механизм не поддерживается. (И не только в Ruby; Java, C#, Objective-C и многие другие объектно-ориентированные языки сознательно не поддерживают множественное наследование.)

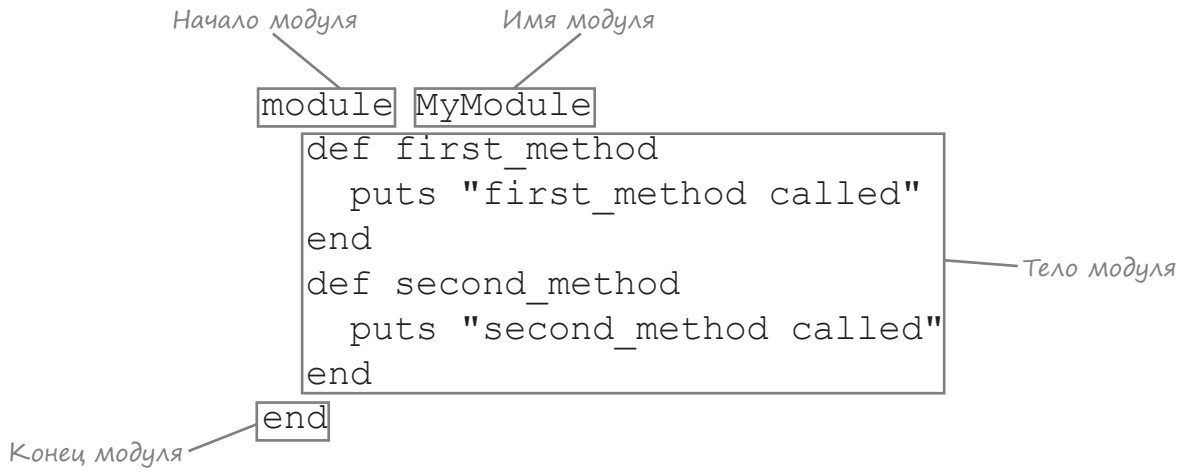
Почему? Слишком много проблем. Множественное наследование порождает неоднозначности, для разрешения которых потребуются сложные правила. Поддержка множественного наследования привела бы к существенному увеличению размера интерпретатора Ruby, а код Ruby стал бы намного более уродливым.

В Ruby используется другое решение...

Модули как примеси

Итак, требуется добавить группу методов в разные классы *без* применения наследования. Как это сделать?

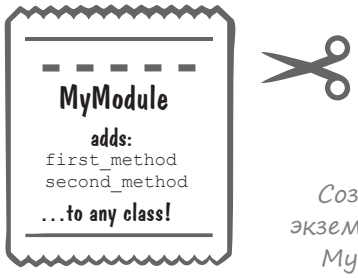
Для группировки взаимосвязанных методов в Ruby используются **модули**. Модуль начинается с ключевого слова `module` и имени модуля (которое должно начинаться с символа верхнего регистра), а заканчивается ключевым словом `end`. Между этими ограничителями — в теле метода — располагаются объявления одного или нескольких методов.



Похоже на класс, верно? Это потому, что класс в действительности является *разновидностью* модуля. Впрочем, существует и ключевое различие: в программе можно создавать экземпляры класса, но не экземпляры модуля:

```
MyModule.new
```
Ошибка →
```
undefined method `new' for MyModule:Module
```

Вместо этого можно объявить класс, а затем включить в него модуль. При этом класс наследует все методы модуля как методы экземпляра.




```
module MyModule
  def first_method
    puts "first_method called"
  end
  def second_method
    puts "second_method called"
  end
end

class MyClass
  include MyModule
end

my_object = MyClass.new
my_object.first_method
my_object.second_method
```

Этот метод станет методом экземпляра любого класса, включающего этот модуль.

И этот тоже.

Модуль MyModule включается в этот класс.

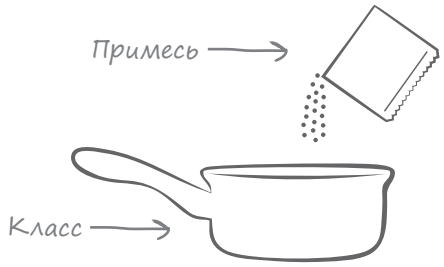
Создаем экземпляр MyClass.

Вызываем методы экземпляра, включенные из MyModule.

```
first_method called
second_method called
```

Модули как примеси (продолжение)

Модули, *предназначенные* для включения в классы, часто называются **примесями** (mixins). По аналогии с тем, как у суперкласса может быть несколько субклассов, примесь может включаться в любое количество классов:



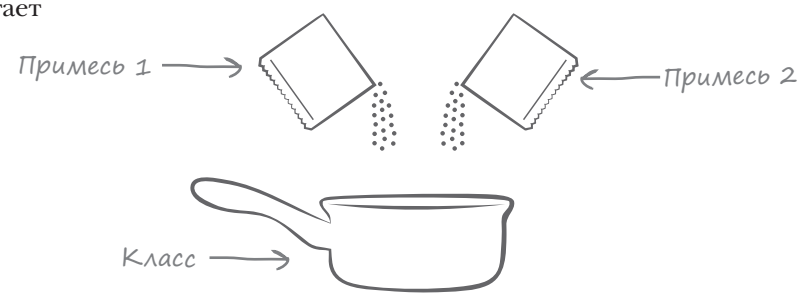

```
module Friendly
  def my_method
    puts "hello from Friendly"
  end
end

class ClassOne
  include Friendly
end

class ClassTwo
  include Friendly
end

ClassOne.new.my_method
ClassTwo.new.my_method
```

Добавляем my_method в ClassOne.

Добавляем my_method в ClassTwo.

```
hello from Friendly
hello from Friendly
```

А теперь самое интересное: в один *класс* может быть включено произвольное количество *модулей*. Такой класс приобретает функциональность *всех* модулей!




```
module Friendly
  def method_one
    puts "hello from Friendly"
  end
end

module Friendlier
  def method_two
    puts "hello from Friendlier!!"
  end
end

class MyClass
  include Friendly
  include Friendlier
end

my_object = MyClass.new
my_object.method_one
my_object.method_two
```

Добавляем method_one.

Добавляем method_two.

```
hello from Friendly
hello from Friendlier!!
```

Модули можно включать в класс независимо от того, есть ли у него суперкласс.

Добавлять примеси все равно возможно, несмотря на наличие суперкласса!

```
class MySuperclass
end

class MySubclass < MySuperclass
  include Friendly
  include Friendlier
end

subclass_instance = MySubclass.new
subclass_instance.method_one
subclass_instance.method_two
```

```
hello from Friendly
hello from Friendlier!!
```

Житейская мудрость

Имена модулей, как и имена классов, должны начинаться с прописной буквы. Обычно для записи имен модулей используется «верблюжья» схема.

```
module Secured
  ...
end
module PasswordProtected
  ...
end
```

Модуль, используемый в качестве примеси, описывает некий аспект поведения объекта, поэтому в именах модулей принято использовать описания. Постарайтесь ограничиться одним прилагательным, если это возможно.

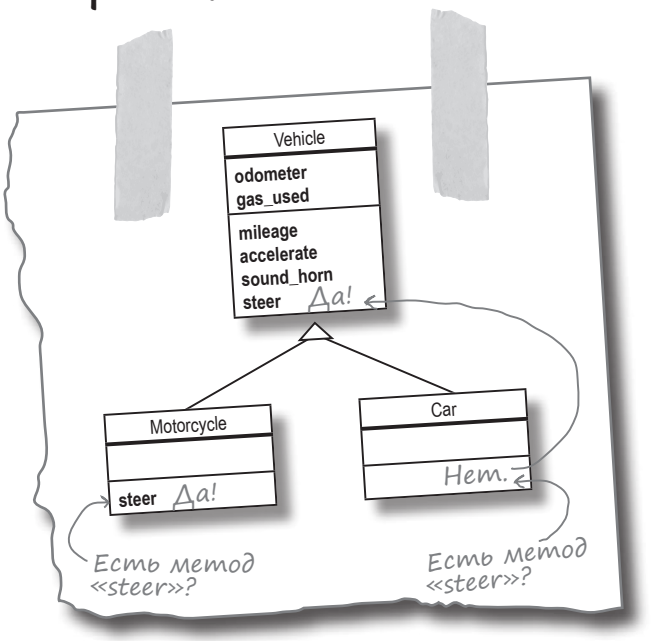
```
module Traceable
  ...
end
```

Если прилагательное оказывается слишком неудобным, описание из нескольких слов тоже подойдет. (Например, имя `Commentable` читается плохо — пожалуй, лучше выбрать имя `AcceptsComments`).

```
module AcceptsComments
  ...
end
```

Примеси: как это работает

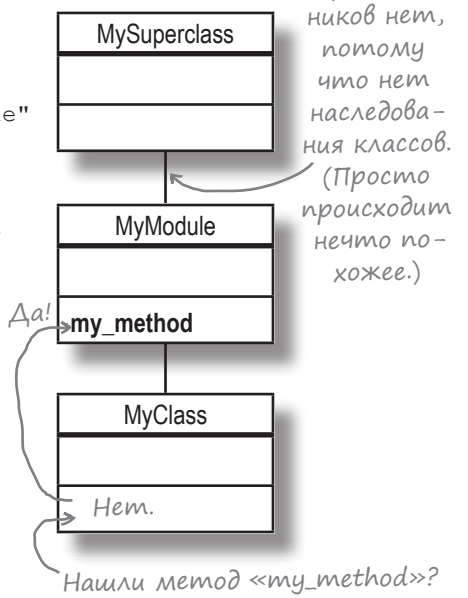
Когда мы рассматривали переопределение методов в главе 3, вы узнали, что Ruby ищет методы экземпляра сначала в классе, потом в суперклассе...



При включении модуля происходит практически то же самое. Ruby добавляет модуль в список мест, в которых производится поиск методов — между классом, в который он включается, и его суперклассом.

```
class MySuperclass
end

module MyModule
  def my_method
    puts "hello from MyModule"
  end
end

class MyClass < MySuperclass
  include MyModule
end

my_object = MyClass.new
my_object.my_method
```

```
hello from MyModule
```



На этой диаграмме треугольников нет, потому что нет наследования классов. (Просто происходит нечто похожее.)

Если после этого будет вызван метод, не найденный в классе, Ruby ищет этот метод в модуле. Так методы из примесей добавляются в классы!

Часто Задаваемые Вопросы

В: Вы сказали, что модули добавляются в список мест, в которых Ruby ищет методы, между классом и его суперклассом. А что произойдет, если суперкласс не определен?

О: Вспомните: у всех классов Ruby есть суперкласс `Object`. Итак, если у вас имеется класс, для которого суперкласс не определен явно, и вы включаете в него модуль:

```
module MyModule
end

class MyClass
  include MyModule
end
```

...то `MyModule` будет добавлен в список поиска между `MyClass` и `Object`.

В: Что произойдет, если модуль включается в суперкласс? Унаследуют ли субклассы добавленные методы?

О: Да! Если Ruby не обнаруживает метод в суперклассе, то поиск продолжается в примесях суперкласса. Таким образом, метод, доступный для суперкласса, будет доступен и для субклассов.

Беседа у камина

Сегодня в студии: Модуль обсуждает с Классом деликатную тему.

Модуль:

Знаешь, Класс, иногда я тебе завидую.

Верно, но я — набор методов, *и ничего более*. А ты умеешь создавать *экземпляры*!

И ты можешь наследовать методы от суперкласса!

Но зачем тебе это нужно?

И тогда методы, актуальные только для *некоторых* классов, перемещаются в модуль?

Да, это выглядит *убедительно*. Пожалуй, я уже не чувствую себя таким неполноценным!

Класс:

С чего бы это, Модуль? Мы оба — всего лишь наборы методов.

Верно, но это далеко не так весело, как тебе кажется.

Я могу наследовать методы от *одного* суперкласса. Ruby большего не позволяет. И правильно делает; иначе ситуация очень сильно... усложнится. Но я могу включить столько модулей, сколько захочу!

Видишь ли, наследование позволяет разместить взаимосвязанные методы разных классов в одном месте. Но тогда *каждый* субкласс должен наследовать *все* эти методы! А может оказаться, что одни методы для него актуальны, а другие нет.

...который затем включается в классы; все верно!

Это действительно полезно. Мы отличная команда!

Развлечения с магнитами

В программе на языке Ruby, выложенной на холодильнике, часть магнитов перепуталась. Программа состоит из двух модулей, класса и кода, который создает экземпляр класса и вызывает его методы. Сможете ли вы расставить фрагменты кода по местам, чтобы программа выводила указанный результат?

```
Monkey
Curious
Clumsy
Curious
Clumsy

end
puts "Looks at #{thing}"

end
end
puts "Knocks over #{thing}"

end
end
bubbles =
Monkey.new

module
module
class

def investigate(thing)
investigate("vase")

def break(thing)
break("vase")

include
include
bubbles.
bubbles.
```

Результат:

```
File Edit Window Help
Looks at vase
Knocks over vase
```

Развлечения с магнитами. Решение

В программе на языке Ruby, выложенной на холодильнике, часть магнитов перепуталась. Программа состоит из двух модулей, класса и кода, который создает экземпляр класса и вызывает его методы. Сможете ли вы расставить фрагменты кода по местам, чтобы программа выводила указанный результат?

```
module Curious
  def investigate(thing)
    puts "Looks at #{thing}"
  end
end

module Clumsy
  def break(thing)
    puts "Knocks over #{thing}"
  end
end

class Monkey
  include Curious
  include Clumsy
end

bubbles = Monkey.new
bubbles.investigate("vase")
bubbles.break("vase")
```

Результат:

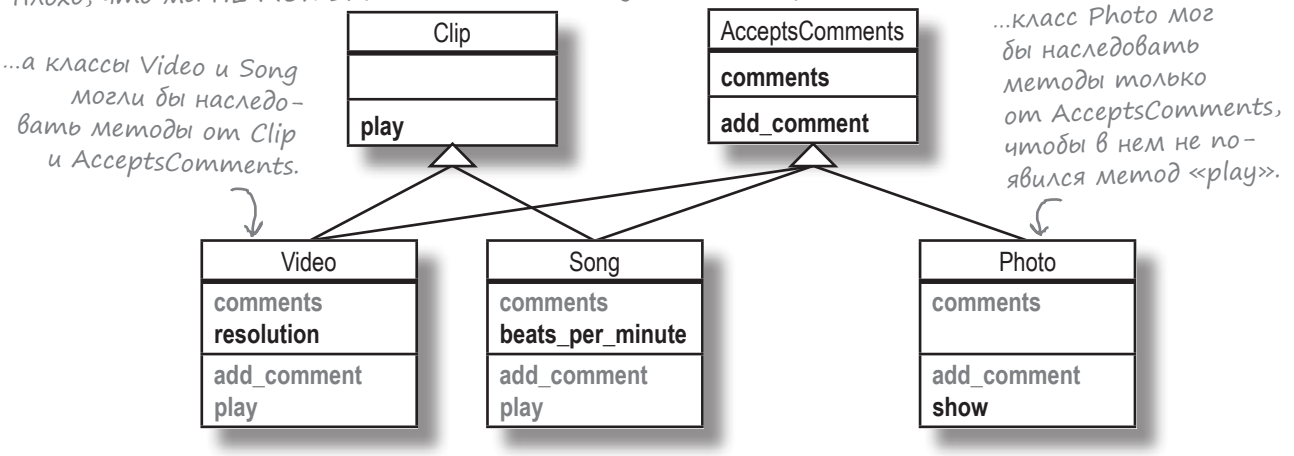
```
File Edit Window Help
Looks at vase
Knocks over vase
```

Создание примеси

Классы `Video`, `Song` и `Photo` должны совместно использовать одну реализацию методов `comments` и `add_comment`, но вы уже знаете, что мы *не можем* применить множественное наследование.

Если бы только можно было создать новый класс AcceptsComments...

Плохо, что мы НЕ МОЖЕМ наследовать от двух классов...



Но как вы уже знаете, организовать совместное использование методов *возможно* – для этого методы следует разместить в модуле и использовать этот модуль как примесь. Попробуем переместить методы `comments` и `add_comment` в модуль. Так как модуль будет содержать методы, при помощи которых объект сможет принимать комментарии, мы назовем его `AcceptsComments`.

Модуль называется AcceptsComments, потому что его методы предназначены для ввода комментариев.

```
class Clip                                  module AcceptsComments
                  Этот метод перемещаем сюда (и вносим некоторые изменения)...
  attr_reader :comments                       def comments
                                                if @comments
Этот ме-                                          @comments
тод удаля-    def initialize                    else
ется...         @comments = []                    @comments = []
              end                               end
                                              end
  def add_comment(comment)                                  Заменяем attr_reader
    comments << comment                                     и метод «initialize».
  end                          ...а этот перемещается сюда.
                                              def add_comment(comment)
А этот                                          comments << comment
метод         def play                        end
остается        puts "Playing #{object_id}..."
на месте.     end                           end

end
```

Кроме перемещения некоторых методов в модуль, мы добавили кое-какую логику в `comments` и избавились от метода `initialize`. Скоро вы узнаете, почему это было сделано, но сначала протестируем внесенные изменения.

Использование примеси

Давайте включим новый модуль `AcceptsComments` в классы* `Video` и `Song` и посмотрим, все ли работает так же, как прежде...

```
module AcceptsComments
  def comments
    if @comments
      @comments
    else
      @comments = []
    end
  end
  def add_comment(comment)
    comments << comment
  end
end

class Clip
  def play
    puts "Playing #{object_id}..."
  end
end

class Video < Clip
  include AcceptsComments
  attr_accessor :resolution
end

class Song < Clip
  include AcceptsComments
  attr_accessor :beats_per_minute
end
```

Вскоре мы объясним этот код, но сначала протестируем этот модуль.

Включаем все методы из модуля AcceptsComments.

Включаем все методы из модуля AcceptsComments.

```
video = Video.new
video.add_comment("Cool slow motion effect!")
video.add_comment("Weird ending.")
song = Song.new
song.add_comment("Awesome beat.")

p video.comments, song.comments
```

Этот код вам уже встречался.

Тот же результат.

```
["Cool slow motion effect!", "Weird ending."]
["Awesome beat."]
```

Примесь работает! Классы `Video` и `Song` содержат методы `comments` и `add_comment`, как и прежде, но теперь эти методы появляются в результате включения `AcceptsComments`, а не наследования от `Clip`. Также можно убедиться в том, что `Video` и `Song` успешно наследуют метод `play` от `Clip`:

```
video.play
song.play
```

```
Playing 70322929946360...
Playing 70322929946280...
```

* *Также можно было включить AcceptsComments в класс Clip и позволить классам Video и Song унаследовать методы, но мы решили, что такое решение чуть проще.*

Часто Задаваемые Вопросы

В: Вы говорили, что экземпляр модуля создать невозможно. Тогда как у модуля могут быть переменные экземпляра?

О: Помните, в главе 3 мы говорили о том, что переменные экземпляра принадлежат *объекту*, а не *классу*? С модулями дело обстоит так же. Методы из модуля, которые ссылаются на переменные экземпляра, включаются в класс как методы экземпляра. Только в тот момент, когда эти методы экземпляра вызываются для объекта, создаются переменные экземпляра *для этого объекта*. Переменные экземпляра вообще не принадлежат модулю; они принадлежат экземплярам классов, которые включают этот модуль.

Использование примеси (продолжение)

Теперь создадим класс `Photo` и посмотрим, удастся ли включить модуль `AcceptsComments` *и в него...*

```
class Photo
  include AcceptsComments
  def show
    puts "Displaying #{object_id}..."
  end
end

photo = Photo.new
photo.add_comment("Beautiful colors.")

p photo.comments
```

Включаем методы «comments» и «add_comment» из модуля AcceptsComments.

```
["Beautiful colors."]
```

Новый класс `Photo` содержит метод `show`, как и положено, но в нем *нет* нежелательного метода `play`. (Этот метод присутствует только в субклассах `Clip`.) Все работает так, как предполагалось!

```
photo.show
```

```
Displaying 70139324385180...
```

Об изменениях в методе «comments»

Итак, вы знаете, как работают примеси, и мы можем повнимательнее присмотреться к методу `comments`. Новая версия метода обращается к переменной экземпляра `@comments` текущего объекта. Если значение `@comments` еще не присваивалось, переменная содержит `nil`. В таком случае `@comments` присваивается пустой массив, который возвращается методом `comments`.

После этого `@comments` уже не содержит `nil`, поэтому последующие вызовы метода `comments` будут возвращать массив, присвоенный `@comments`.

```
module AcceptsComments
  def comments
    if @comments
      @comments
    else
      @comments = []
    end
  end
  def add_comment(comment)
    comments << comment
  end
end
```

Заменяет attr_reader И метод «initialize».

Если @comments уже присвоено значение...

...вернуть это значение.

Если значение еще не задано, присваиваем пустой массив и возвращаем его.

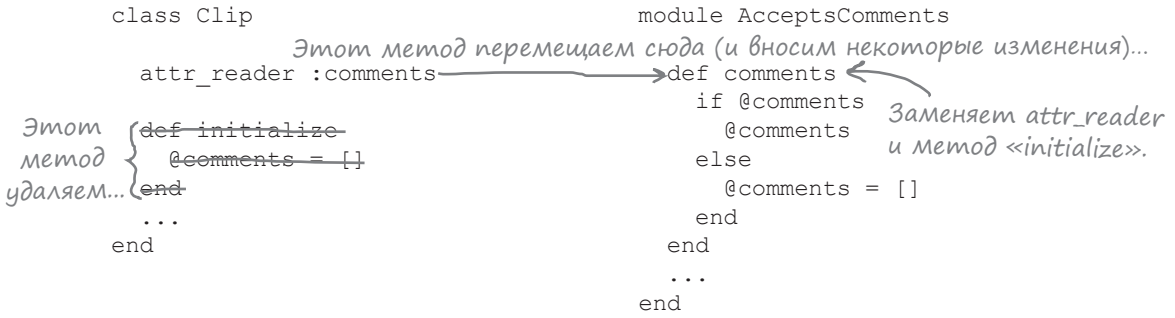
Получает пустой список, если значение еще не было присвоено, или существующий список, если оно БЫЛО присвоено.

Метод `add_comment` не изменился; он по-прежнему использует метод `comments` для получения массива, в который должен добавляться комментарий. Соответственно при первом вызове комментарий будет добавлен в новый, пустой массив. При последующих вызовах комментарии будут добавляться в тот же массив, потому что метод `comments` будет возвращать именно его.

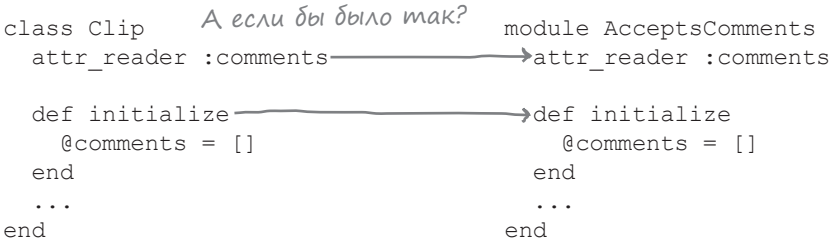
Так как метод экземпляра `comments` гарантирует инициализацию `@comments`, метод `initialize` суперкласса `Clip` уже не нужен. И его можно просто удалить!

Почему не следует добавлять «initialize» в примесь

Переработанный метод `comments` в модуле `AcceptsComments` работает ничуть не хуже метода чтения атрибута в старом суперклассе `Clip`. Метод `initialize` больше не нужен.



Но стоило ли идти на хлопоты с обновлением метода `comments`? Разве мы не могли просто переместить метод `initialize` из `Clip` в `AcceptsComments`?



Если бы мы так поступили, то метод `initialize` работал бы абсолютно нормально... поначалу.

```
module AcceptsComments
  attr_reader :comments
  def initialize
    @comments = []
  end
  def add_comment(comment)
    comments << comment
  end
end

class Photo
  include AcceptsComments
  def show
    puts "Displaying #{object_id}..."
  end
end

photo = Photo.new
photo.add_comment("Beautiful colors.")
p photo.comments
```

При создании объекта @comments присваивается пустой массив.

При первом вызове значение @comments уже задано!

Вызывает примесный метод «initialize».

```
["Beautiful colors."]
```

Метод `initialize` включается в класс `Photo` и вызывается при вызове `Photo.new`. Он присваивает переменной экземпляра `@comments` пустой массив еще до того, как будет вызван метод чтения `comments`.

Почему не следует добавлять «initialize» в примесь (продолжение)

Использование метода `initialize` в нашем модуле работает... *пока* вы не добавите метод `initialize` в один из классов, в который он включен.

Предположим, вы решили добавить метод `initialize` в класс `Photo` для назначения формата по умолчанию:

```
class Photo
  include AcceptsComments
  def initialize
    @format = 'JPEG'
  end
  ...
end
```

Если после добавления метода `initialize` вы создадите новый экземпляр `Photo` и вызовете для него метод `add_comment`, произойдет ошибка!

```
photo = Photo.new
photo.add_comment("Beautiful colors.")
```

Ошибка

```
in `add_comment': undefined
method `<<' for nil:NilClass
```

Если добавить в два метода `initialize` команды `puts` для упрощения отладки, суть проблемы становится очевидной...

```
module AcceptsComments
  ...
  def initialize
    puts "In initialize from AcceptsComments"
    @comments = []
  end
  ...
end

class Photo
  include AcceptsComments
  def initialize
    puts "In initialize from Photo"
    @format = 'JPEG'
  end
  ...
end

photo = Photo.new
```

Нам бы хотелось увидеть эту строку в выводе.

И эта строка тоже должна выводиться.

Из отладочного вывода становится ясно, что вызывается только метод «initialize» класса Photo!

```
In initialize from Photo
```

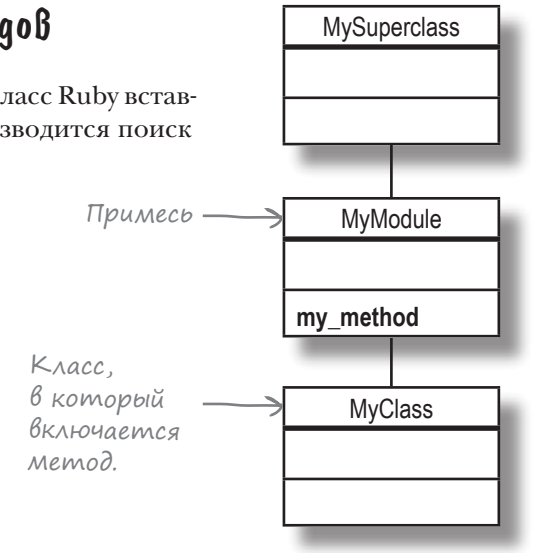
Итак, вызывается метод `initialize`, определенный в классе `Photo`, а метод `initialize` из `AcceptsComments` не вызывается! Поэтому в приведенном выше коде переменная экземпляра `@comments` при вызове `add_comment` все еще содержит `nil`, и происходит ошибка.

Дело в том, что метод `initialize` класса *переопределяет* метод `initialize` примеси.

Примеси и переопределение методов

Как вы уже знаете, при включении модуля в класс Ruby вставляет модуль в цепочку мест, в которых производится поиск методов, между классом и суперклассом.

```
class MySuperclass
end

module MyModule
  def my_method
    puts "hello from MyModule"
  end
end

class MyClass < MySuperclass
  include MyModule
end
```



Метод класса `ancestors` возвращает список всех мест, в которых Ruby ищет методы (как примеси, так и суперклассы). Метод возвращает массив всех примесей и суперклассов в том порядке, в котором будет проводиться поиск.

```
p MyClass.ancestors
```

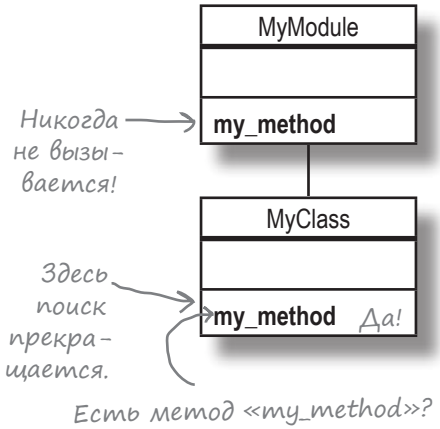
Примесь

```
[MyClass, MyModule, MySuperclass, Object, Kernel, BasicObject]
```

В главе 3 вы видели, что в процессе поиска Ruby ищет метод в классе, вызывает его *и прекращает дальнейший поиск*. Именно это позволяет субклассам переопределять методы из суперклассов.

Этот принцип распространяется и на примеси. Ruby ищет методы экземпляра в модулях и классах, входящих в массив `ancestors` класса, в указанном порядке. Если искомый метод обнаруживается в классе, то Ruby просто вызывает этот метод. Все одноименные методы из примеси игнорируются; фактически они *переопределяются* методом класса.

```
module MyModule
  def my_method
    puts "hello from MyModule"
  end
end

class MyClass
  include MyModule
  def my_method
    puts "hello from MyClass"
  end
end

p MyClass.ancestors

MyClass.new.my_method
```



Переопределенный метод находится здесь.

И до этого места вообще не доходит!

```
[MyClass, MyModule, Object, Kernel, BasicObject]
hello from MyClass
```

Упражнение

Предположите, какой результат выведет этот код, и запишите его в отведенном месте.

```
module JetPropelled
  def move(destination)
    puts "Flying to #{destination}."
  end
end

class Robot
  def move(destination)
    puts "Walking to #{destination}."
  end
end

class TankBot < Robot
  include JetPropelled
  def move(destination)
    puts "Rolling to #{destination}."
  end
end

class HoverBot < Robot
  include JetPropelled
end

class FarmerBot < Robot
end

TankBot.new.move("hangar")
HoverBot.new.move("lab")
FarmerBot.new.move("field")
```

.....

.....

.....

Старайтесь не использовать методы «initialize» в модулях

Вероятно, вы уже поняли, почему присутствие метода `initialize` в примеси `AcceptsComments` может вызвать проблемы в будущем. Если мы добавим метод `initialize` в класс `Photo`, то он переопределит `initialize` из примеси, и переменная `@comments` не будет инициализирована.

```
module AcceptsComments
  attr_reader :comments
  def initialize
    @comments = []
  end
  def add_comment(comment)
    comments << comment
  end
end

class Photo
  include AcceptsComments
  def initialize
    @format = 'JPEG'
  end
end

photo = Photo.new
photo.add_comment("Beautiful colors.")
```

Переменная экземпляра @comments содержит nil, поэтому здесь происходит ошибка.

Переопределяет метод «initialize» в примеси, поэтому тот не будет выполнен.

Ошибка →

```
in `add_comment': undefined
method `<<' for nil:NilClass
```

Старайтесь не использовать методы «initialize» в модулях (продолжение)

Именно поэтому вместо того, чтобы полагаться на метод `initialize` из `AcceptsComments`, мы инициализируем переменную экземпляра `@comments` в методе `comments`.

Заменяет attr_reader и метод «initialize».

Если значение @comments уже задано...

...возвращаем это значение.

Если оно еще не задано, присваиваем пустой массив и возвращаем его.

```
module AcceptsComments
  def comments
    if @comments
      @comments
    else
      @comments = []
    end
  end
  ...
end
```

В этом случае неважно, присутствует ли метод `initialize` в классе `Photo`, потому что в `AcceptsComments` нет метода `initialize`, который бы он переопределил. Пустой массив присваивается переменной экземпляра `@comments` при первом вызове `comments`, и все отлично работает!

```
photo = Photo.new
photo.add_comment("Beautiful colors.")
p photo.comments
```

```
["Beautiful colors."]
```

Вывод: избегайте использования методов `initialize` в модулях. Если вам потребуется инициализировать переменную экземпляра перед использованием, сделайте это в методе доступа в модуле.

Упражнение
Решение

Предположите, какой результат выведет этот код, и запишите его в отведенном месте.

```
module JetPropelled
  def move(destination)
    puts "Flying to #{destination}."
  end
end

class Robot
  def move(destination)
    puts "Walking to #{destination}."
  end
end

class TankBot < Robot
  include JetPropelled
  def move(destination)
    puts "Rolling to #{destination}."
  end
end

class HoverBot < Robot
  include JetPropelled
end

class FarmerBot < Robot
end

TankBot.new.move("hangar")
HoverBot.new.move("lab")
FarmerBot.new.move("field")
```

Переопределяет метод из примеси.

Метод примеси переопределяет метод суперкласса.

Примеси нет, поэтому вызывается метод суперкласса.

Rolling to hangar.
Flying to lab.
Walking to field.

Логический оператор «или» при присваивании

У метода `comments` есть один небольшой недостаток: он занимает несколько строк. Методы доступа к атрибутам желательно сделать более компактными — особенно если вы собираетесь добавить к ним другие.

А теперь представьте, что эти *пять* строк кода в методе `comments` можно сократить до *одной*, полностью сохранив всю функциональность! Один из способов выглядит так:

Присваиваем новое значение @comments, но только в том случае, если @comments содержит nil.

```
module AcceptsComments
  def comments
    if @comments
      @comments
    else
      @comments = []
    end
  end
end
```

Присваиваем новое значение @comments, но только в том случае, если @comments содержит nil.

```
module AcceptsComments
  def comments
    @comments = @comments || []
  end
end
```

Та же функциональность в одной строке!

Как вы узнали в главе 1, оператор `||` (читается «или») проверяет, что хотя бы одно из двух выражений равно `true` (или отлично от `nil`). Если левое выражение истинно, то оно считается значением всего выражения `||`, а если нет, то за значение всего выражения `||` принимается правое выражение. Все это звучит совершенно невразумительно, поэтому проще посмотреть, как работает этот оператор:

`p true || false`	**`true`**
`p false || true`	**`true`**
`p nil || true`	**`true`**
`p true || "righthand value"`	**`true`**
`p "lefthand value" || "righthand value"`	**`"lefthand value"`**
`p nil || []`	**`[]`**
`p [] || "righthand value"`	**`[]`**

Если `@comments` содержит `nil`, то значением выражения `||` в методе `comments` будет пустой массив. Если значение `@comments` *отлично* от `nil`, то значением выражения `||` будет значение из `@comments`.

`@comments = nil` `p @comments || []`	**`[]`**
`@comments = ["Beautiful colors."]` `p @comments || []`	**`["Beautiful colors."]`**

Оператор условного присваивания

Значение выражения `||` снова присваивается `@comments`. Таким образом, `@comments` заново присваивается текущее значение *или*, если текущее значение равно `nil`, присваивается пустой массив.

```
@comments = nil
@comments = @comments || []
p @comments
```

```
[]
```

```
@comments = ["Beautiful colors."]
@comments = @comments || []
p @comments
```

```
["Beautiful colors."]
```

Эта сокращенная запись применяется настолько часто, что в Ruby существует оператор `||=`, также называемый *условным оператором присваивания*, который делает то же самое. Если переменная содержит `nil` (или false), то условный оператор присваивания задает ей новое значение. Но если переменная содержит любое другое значение, то `||=` оставляет его неизменным.

```
@comments = nil
@comments ||= []
p @comments
```

```
[]
```

```
@comments = ["Beautiful colors."]
@comments ||= []
p @comments
```

```
["Beautiful colors."]
```

Условный оператор присваивания идеально подходит для метода `comments` из примеси `AcceptsComments`...

```
module AcceptsComments
  def comments
    @comments ||= []
  end
  def add_comment(comment)
    comments << comment
  end
end

class Photo
  include AcceptsComments
  def initialize
    @format = 'JPEG'
  end
end

photo = Photo.new
photo.add_comment("Beautiful colors.")
p photo.comments
```

Если переменная экземпляра @comments содержит nil, ей присваивается пустой массив. Метод возвращает значение @comments.

```
["Beautiful colors."]
```

Метод `comments` работает так же, но теперь он занимает всего одну строку вместо пяти!

Упражнение

Попробуйте предположить, что выведет каждый из этих фрагментов, и запишите свои ответы.

(Мы вписали первый ответ за вас.)

```
puts true || "my"
```
true

```
puts false || "friendship"
```
.....

```
puts nil || "is"
```
.....

```
puts "not" || "often"
```
.....

```
first = nil
puts first || "easily"
```
.....

```
second = "earned."
puts second || "purchased."
```
.....

```
third = false
third ||= true
puts third
```
.....

```
fourth = "love"
fourth ||= "praise"
puts fourth
```
.....

```
fifth = "takes"
fifth ||= "gives"
puts fifth
```
.....

```
sixth = nil
sixth ||= "work."
puts sixth
```
.....

Попробуйте предположить, что выведет каждый из этих фрагментов, и запишите свои ответы.

```
puts true || "my"
```
true

```
puts false || "friendship"
```
friendship

```
puts nil || "is"
```
is

```
puts "not" || "often"
```
not

```
first = nil
puts first || "easily"
```
easily

```
second = "earned."
puts second || "purchased."
```
earned.

```
third = false
third ||= true
puts third
```
true

```
fourth = "love"
fourth ||= "praise"
puts fourth
```
love

```
fifth = "takes"
fifth ||= "gives"
puts fifth
```
takes

```
sixth = nil
sixth ||= "work."
puts sixth
```
work.

Полный код

Перед вами полный обновленный код приложения общего доступа к мультимедиа. Теперь, когда метод `comments` сам инициализирует массив `@comments`, метод `initialize` из примеси `AcceptsComments` становится лишним. Все прекрасно работает!

```
module AcceptsComments
  def comments
    @comments ||= []
  end
  def add_comment(comment)
    comments << comment
  end
end
```

Возвращает либо значение по умолчанию для @comments, либо текущее значение. Метод «initialize» больше не нужен!

Добавляем новый комментарий в массив.

Используем метод «comments» для получения значения @comments.

Суперкласс

```
class Clip
  def play
    puts "Playing #{object_id}..."
  end
end
```

Наследуется классами Video и Song.

Субкласс

```
class Video < Clip
  include AcceptsComments
  attr_accessor :resolution
end
```

Включаем методы «comments» и «add_comment».

Субкласс

```
class Song < Clip
  include AcceptsComments
  attr_accessor :beats_per_minute
end
```

Включаем методы «comments» и «add_comment».

Автономный класс

```
class Photo
  include AcceptsComments
  def initialize
    @format = 'JPEG'
  end
end
```

Возвращаем методы «comments» и «add_comment».

И не нужно беспокоиться о конфликтах имен с AcceptsComments!

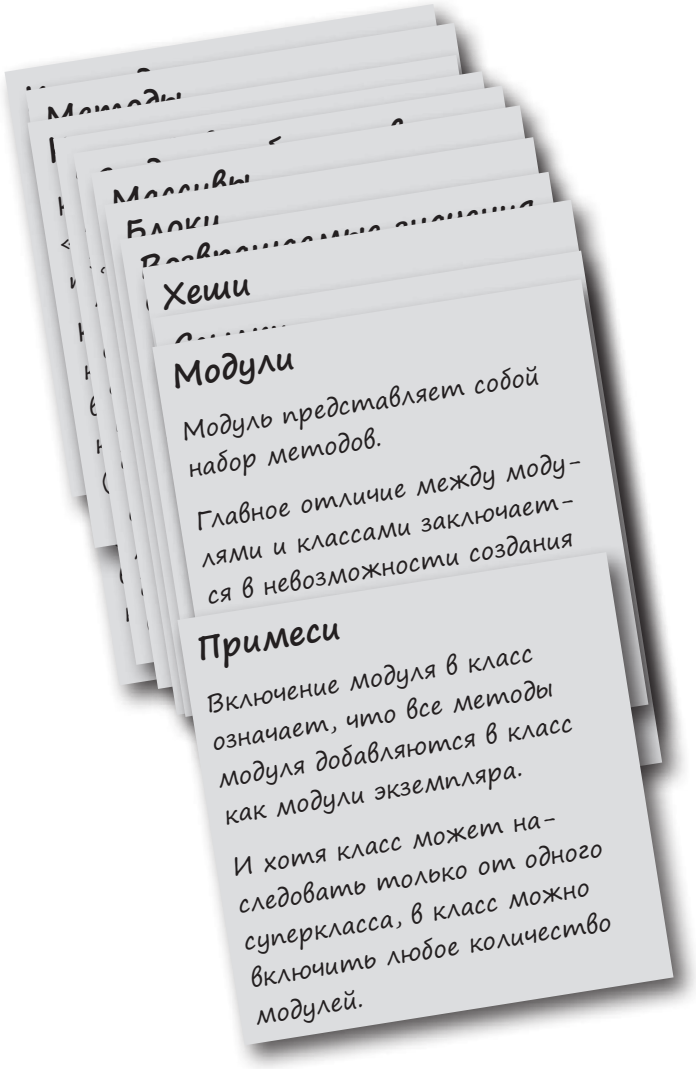
Создаем экземпляры Video, Song и Photo, после чего используем методы из примеси для добавления и вывода комментариев.

```
video = Video.new
video.add_comment("Cool slow motion effect!")
video.add_comment("Weird ending.")
song = Song.new
song.add_comment("Awesome beat.")
photo = Photo.new
photo.add_comment("Beautiful colors.")

p video.comments, song.comments, photo.comments
```

```
["Cool slow motion effect!", "Weird ending."]
["Awesome beat."]
["Beautiful colors."]
```

Ваш инструментарий Ruby

Глава 9 подошла к концу, а ваш инструментарий пополнился модулями и примесями.



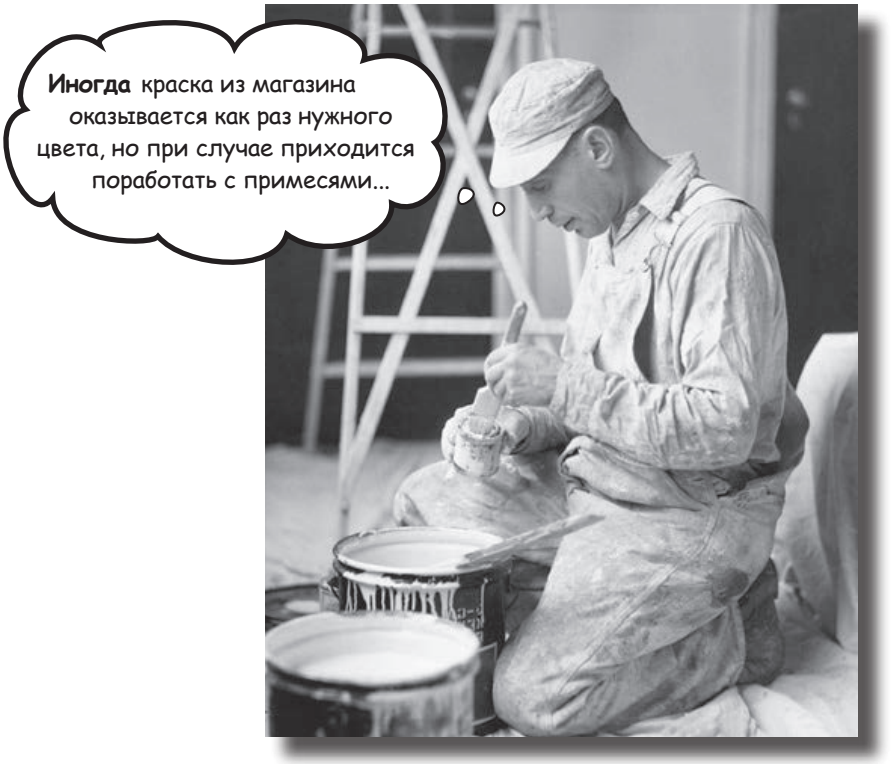
КЛЮЧЕВЫЕ МОМЕНТЫ

- Имена модулей должны начинаться с буквы в верхнем регистре. В остальных символах обычно применяется «верблюжья» схема записи.
- Так как имена модулей обычно описывают некий аспект поведения объекта, в качестве имени модуля также принято выбирать описание (например, `Traceable` или `PasswordProtected`).
- Для каждого класса Ruby поддерживает список мест, в которых будет производиться поиск методов экземпляра. При включении модуля в класс Ruby добавляет модуль в этот список, сразу же после класса.
- Чтобы просмотреть содержимое списка мест для поиска методов, вызовите метод класса `ancestors`.
- Если в классе и в примеси присутствуют одноименные методы, то метод экземпляра класса переопределяет метод экземпляра примеси.
- Методы примесей, как и методы экземпляра в классах, могут создавать переменные экземпляра для текущего объекта.
- Условный оператор присваивания `||=` присваивает новое значение переменной только в том случае, если она содержит `nil` (или `false`). В частности, его удобно использовать для выбора значения по умолчанию для переменной.

Далее в программе…

Итак, вы научились создавать и использовать примеси. Но что *делать* с этим замечательным новым инструментом? У Ruby на этот счет есть свои идеи! Язык включает несколько мощных модулей, готовых к включению в ваши классы. Мы рассмотрим их в следующей главе.

Готовые примеси



Из предыдущей главы вы уже знаете, что примеси полезны. Но вы еще не видели их истинную мощь. В стандартную библиотеку Ruby входят две совершенно *потрясающие* примеси. Первая, `Comparable`, используется для сравнения объектов. Мы уже использовали такие операторы, как <, > и ==, с числами и строками, а с `Comparable` сможете использовать их с *вашими* классами. Вторая примесь, `Enumerable`, применяется при работе с коллекциями. Помните замечательные методы `find_all`, `reject` и `map`, которые мы использовали с массивами? Так вот, они были предоставлены `Enumerable`. Но это лишь малая часть возможностей `Enumerable`! И разумеется, вы можете включать эту примесь в *свои* классы. Хотите узнать, как это делается? Читайте дальше!

Встроенные примеси в Ruby

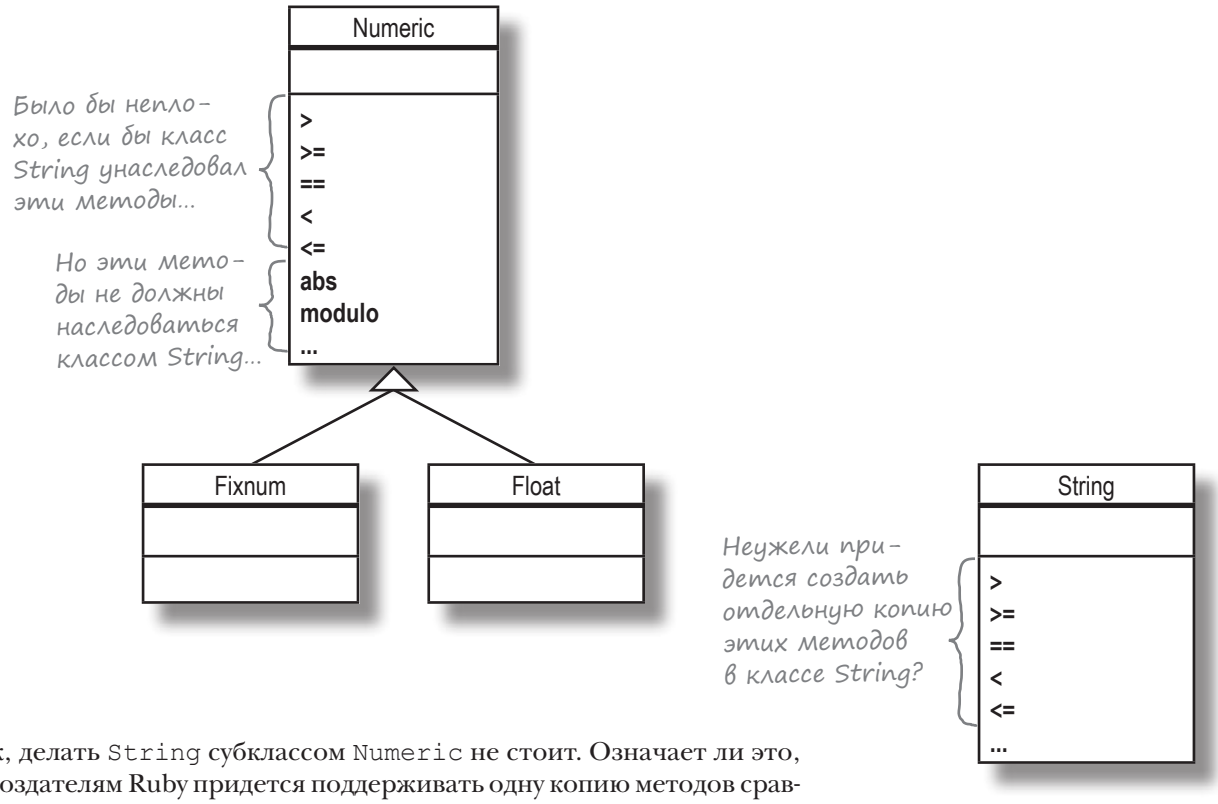
Теперь вы знаете, как работают модули, и мы можем перейти к рассмотрению чрезвычайно полезных модулей, включенных в язык Ruby.

Представьте себе, мы пользовались примесями еще с главы 1. Просто вы об этом не знали.

Помните сравнения чисел операторами <, > и == в игре с угадыванием чисел из главы 1? Стоит напомнить, что операторы сравнения в Ruby реализованы в виде методов.

Операторы сравнения необходимы всем числовым классам, поэтому создатели Ruby *могли* просто добавить их в класс `Numeric`, который является суперклассом `Fixnum`, `Float` и всех остальных числовых классов в Ruby.

Однако операторы сравнения также нужны и классу `String`. А делать `String` субклассом `Numeric` было бы нелогично. Прежде всего `String` унаследует целый набор методов, совершенно неуместных для `String`, — таких, как `abs` (абсолютное значение числа) и `modulo` (остаток от деления).



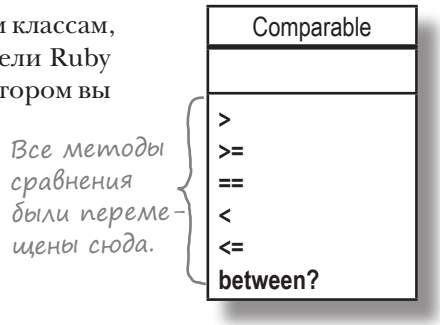
Итак, делать `String` субклассом `Numeric` не стоит. Означает ли это, что создателям Ruby придется поддерживать одну копию методов сравнения в классе `Numeric` и продублировать методы в классе `String`?

Как вы узнали из предыдущей главы — нет! Проблема решается включением соответствующего модуля.

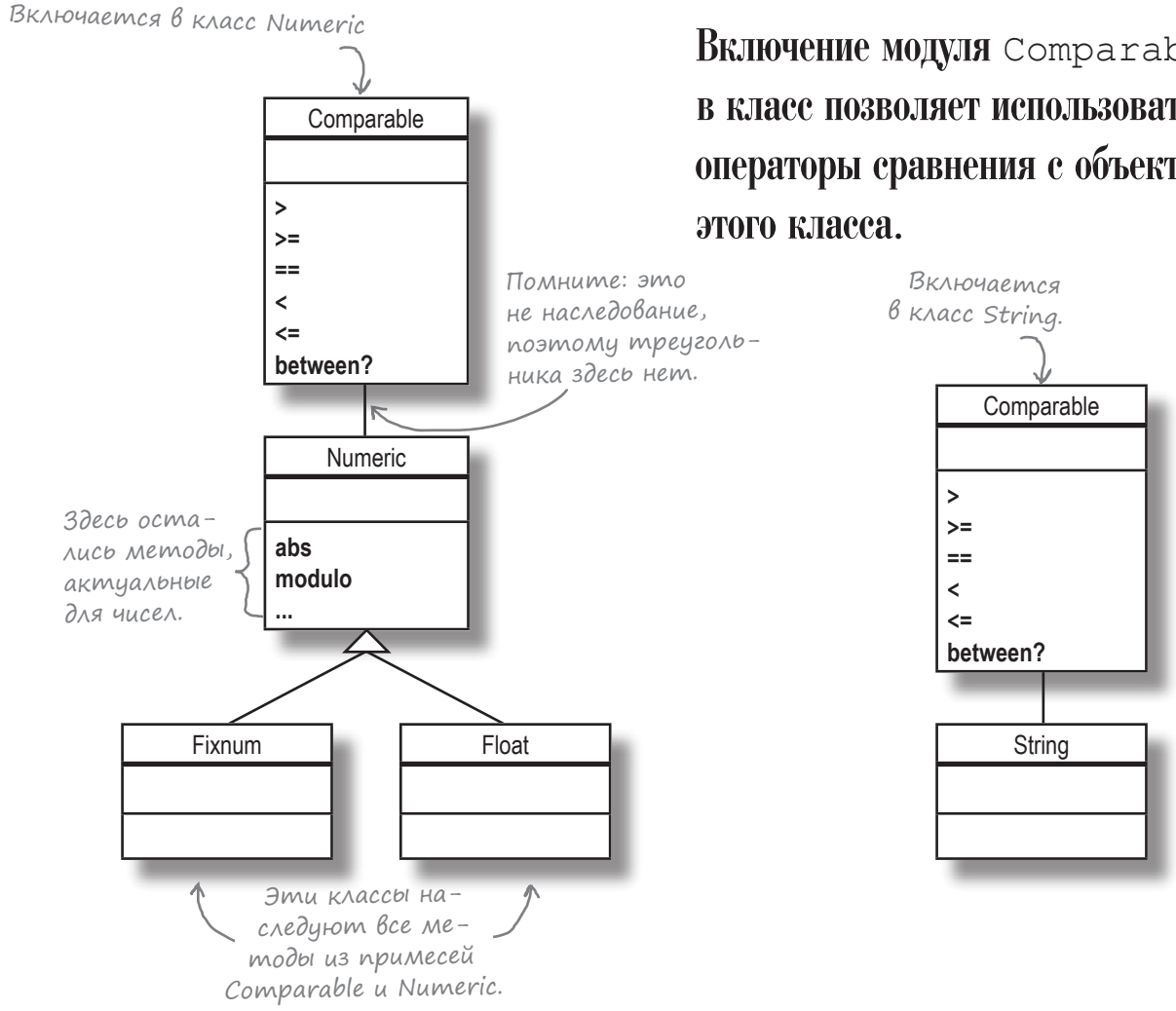
Знакомство с примесью Comparable

Из-за того, что методы <, <=, ==, > и >= нужны многим разнородным классам, применять наследование было бы неразумно. Вместо этого создатели Ruby разместили эти пять методов (а также шестой метод `between?`, о котором вы узнаете позднее) в модуле с именем `Comparable`.

Далее модуль `Comparable` был включен в `Numeric`, `String` и все остальные классы, которым были нужны операторы сравнения. И в результате во всех этих классах появились методы <, <=, ==, >, >= и `between?`.



Включение модуля `Comparable` в класс позволяет использовать операторы сравнения с объектами этого класса.

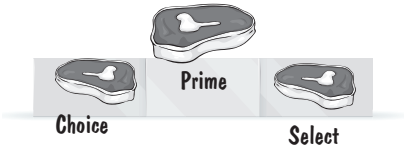


А теперь самая убойная новость: *вы* можете включать `Comparable` и в *свои* классы! Хм... Но в каком классе может пригодиться эта примесь?

Выбор стейка

А сейчас мы поговорим на тему, близкую и дорогую нашему сердцу: о стейках. В США говядина делится на три «категории», описывающие качество мяса. Ниже они перечислены в порядке убывания качества:

- Высшая (Prime)
- Отборная (Choice)
- Средняя (Select)



Категории

Таким образом, стейк из категории «Prime» доставит вам больше удовольствия за обедом, чем стейк из категории «Choice», а стейк из категории «Choice» — больше, чем стейк из категории «Select».

Допустим, имеется простой класс `Steak` с атрибутом `grade`, которому присваивается одно из трех значений: `"Prime"`, `"Choice"` или `"Select"`:

```
class Steak
  attr_accessor :grade
end
```

Мы хотим иметь возможность легко сравнивать объекты `Steak`, чтобы знать, какой из них следует купить. Другими словами, мы хотим использовать код следующего вида:

```
if first_steak > second_steak
  puts "I'll take #{first_steak}"
end
```

Невозможно (пока)!

Как вы узнали в главе 4, операторы сравнения — такие, как >, — в действительности реализуются вызовами метода экземпляра сравниваемого объекта. Следовательно, следующий код:

```
4 > 3
```

на самом деле преобразуется в вызов метода экземпляра с именем > для объекта `4`, с передачей значения `3` как аргумента:

```
4.>(3)
```

Получается, что мы должны определить для класса `Steak` метод экземпляра с именем >, который получает второй экземпляр `Steak` для сравнения.

```
first_steak.>(second_steak)
```

Если такая запись возможна, то в программе также можно будет использовать запись «if first_steak > second_steak».

Реализация метода «больше» в классе Steak

Ниже приведена первая версия метода > для класса `Steak`. Метод сравнивает текущий объект (`self`) со вторым объектом; если `self` «больше», возвращается `true`, а если «больше» другой объект — возвращается `false`. Чтобы определить, какой из двух объектов больше, мы будем сравнивать их атрибуты `grade`.

```
class Steak

  attr_accessor :grade

  def >(other)
    grade_scores = {"Prime" => 3, "Choice" => 2, "Select" => 1}
    grade_scores[grade] > grade_scores[other.grade]
  end

end

first_steak = Steak.new
first_steak.grade = "Prime"
second_steak = Steak.new
second_steak.grade = "Choice"

if first_steak > second_steak
  puts "I'll take #{first_steak.inspect}."
end
```

«Другой» экземпляр Steak, с которым будет сравниваться текущий экземпляр.

Хеш преобразует строку «grade» в число для удобства сравнения.

Возвращает true, если числовой эквивалент атрибута grade текущего экземпляра Steak больше числового эквивалента grade экземпляра other. В противном случае возвращается false.

Новый метод используется для сравнения двух экземпляров Steak.

```
I'll take #<Steak:0x007fc0bc20eae8 @grade="Prime">.
```

В новом методе > важная роль отводится хешу `grade_scores`, который позволяет взять строку категории (`"Prime"`, `"Choice"` или `"Select"`) и преобразовать ее в число. После этого остается только сравнить числа!

```
grade_scores = {"Prime" => 3, "Choice" => 2, "Select" => 1}
puts grade_scores["Prime"]
puts grade_scores["Choice"]
puts grade_scores["Prime"] > grade_scores["Select"]
```

```
3
2
true
```

С готовым методом > мы сможем использовать оператор > для сравнения экземпляров `Steak`. Считается, что экземпляр `Steak` с категорией `"Prime"` больше экземпляра `Steak` с категорией `"Choice"`, а экземпляр `Steak` с оценкой `"Choice"` больше экземпляра `Steak` с оценкой `"Select"`.

У этого кода есть всего один недостаток: он создает новый объект хеша и присваивает его переменной `grade_scores` при каждом выполнении метода >. Получается не очень эффективно. Похоже, нужно ненадолго отвлечься от основной темы и поговорить о *константах*...

Константы

Метод > класса `Steak` при каждом выполнении создает новый объект хеша и присваивает его переменной `grade_scores`. Многократное создание хеша весьма неэффективно, поскольку содержимое хеша *никогда не изменяется*.

```
class Steak

  attr_accessor :grade

  def >(other)
    grade_scores = {"Prime" => 3, "Choice" => 2, "Select" => 1}
    grade_scores[grade] > grade_scores[other.grade]
  end

end
```

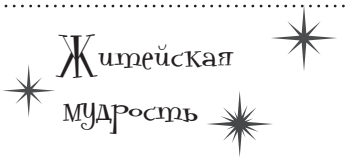
При каждом выполнении метода > один и тот же хеш создается заново!

Для подобных ситуаций в Ruby существуют **константы**: ссылки на объект, который никогда не изменяется.

Если вы присваиваете значение имени, начинающемуся с буквы в верхнем регистре, Ruby будет интерпретировать его как константу, а не как переменную. По общепринятому соглашению имена констант записываются в верхнем регистре (`ALL_CAPS`). Вы можете присвоить значение константе в теле модуля или класса, а затем обращаться к такой константе из любого места этого класса или модуля.

```
class MyClass
  MY_CONSTANT = 42
  def my_method
    puts MY_CONSTANT
  end
end

MyClass.new.my_method
```

```
42
```



Записывайте имена констант ВЕРХНЕМ РЕГИСТРЕ, разделяя слова подчеркиваниями.

```
PHI = 1.618
SPEED_OF_LIGHT = 299792458
```

Вместо того чтобы заново определять хеш числовых эквивалентов при каждом вызове метода >, лучше определить его как константу. Далее остается только обратиться к константе в методе >.

```
class Steak

  GRADE_SCORES = {"Prime" => 3, "Choice" => 2, "Select" => 1}

  attr_accessor :grade

  def >(other)
    GRADE_SCORES[grade] > GRADE_SCORES[other.grade]
  end

end
```

Определяем константу в теле класса.

К константе можно обращаться из этого или любого другого метода в классе Steak.

С константой наш код работает точно так же, как с переменной, но хеш теперь создается всего один раз. Такое решение намного эффективнее!

Работа только начинается...

Переработанный метод > для сравнения объектов Steak отлично работает...

```
class Steak

  GRADE_SCORES = {"Prime" => 3, "Choice" => 2, "Select" => 1}

  attr_accessor :grade

  def >(other)
    GRADE_SCORES[grade] > GRADE_SCORES[other.grade]
  end

end

first_steak = Steak.new
first_steak.grade = "Prime"
second_steak = Steak.new
second_steak.grade = "Choice"

if first_steak > second_steak
  puts "I'll take #{first_steak.inspect}."
end
```

```
I'll take #<Steak:0x007fa5b5816ca0 @grade="Prime">.
```

Но пока он дает только одно: оператор >. Скажем, если вы попытаетесь использовать оператор <, произойдет ошибка:

```
if first_steak < second_steak
  puts "I'll take #{second_steak}."
end
```

```
undefined method `<' for #<Steak:0x007facdb0f2fa0 @grade="Prime">
```

То же самое можно сказать и об операторах <= и >=. Класс Object содержит метод ==, который наследуется классом Steak, так что оператор == не приведет к ошибке, но эта версия == не подходит для наших целей (она возвращает true только при сравнении двух ссылок *на один и тот же объект*).

Итак, прежде чем использовать эти операторы в программе, мы должны реализовать соответствующие методы. Перспектива не самая радостная, верно? К счастью, существует более эффективное решение. Пора взяться за модуль Comparable!

Примесь Comparable

Встроенный модуль `Comparable` позволяет *сравнивать* экземпляры вашего класса. `Comparable` предоставляет методы, в которых вы можете использовать операторы <, <=, ==, > и >= (а также метод `between?`, с помощью которого можно определить, находится ли один экземпляр класса «между» двумя другими экземплярами).

`Comparable` включается в строковые и числовые классы Ruby (и не только) для реализации всех упомянутых операторов — и вы тоже можете использовать этот модуль! Для этого достаточно добавить в ваш класс один конкретный метод, который будет использоваться в работе `Comparable`, а затем включить модуль — и все эти операторы достанутся вам «бесплатно».

Если бы мы решили написать собственную версию `Comparable`, она бы выглядела примерно так:

Как и прежде, «other» — другой экземпляр, с которым сравнивается экземпляр this.

Конечно, «self» обозначает текущий экземпляр.

```
module Comparable
  def <(other)
    (self <=> other) == -1
  end
  def >(other)
    (self <=> other) == 1
  end
  def ==(other)
    (self <=> other) == 0
  end
  def <=(other)
    comparison = (self <=> other)
    comparison == -1 || comparison == 0
  end
  def >=(other)
    comparison = (self <=> other)
    comparison == 1 || comparison == 0
  end
  def between?(first, second)
    (self <=> first) >= 0 && (self <=> second) <= 0
  end
end
```

Все остальные методы используют «other» и «self» аналогичным образом.

Оператор <=>

Этот оператор, называемый на жаргоне «космическим кораблем», представляет собой разновидность оператора сравнения.

Такое название объясняется тем, что <=> напоминает «космический корабль».

Считайте, что <=> представляет собой комбинацию операторов <, == и >. Он возвращает `-1`, если выражение слева *меньше* выражения справа; `0`, если выражения *равны*; и `1`, если выражение слева *больше* выражения справа.

```
puts 3 <=> 4
puts 3 <=> 3
puts 4 <=> 3
```

```
-1
0
1
```

Реализация оператора <=> в классе Steak

Мы уже упоминали о том, что работа `Comparable` зависит от присутствия в классе конкретного метода... И это метод оператора <=>.

Как и <, >, == и т. д., оператор <=> реализуется методом. Когда Ruby встречает в вашем коде конструкцию следующего вида:

```
3 <=> 4
```

он преобразует ее в вызов метода экземпляра:

```
3.<=>(4)
```

А вот что это означает: если добавить в класс `Steak` метод экземпляра <=>, то каждый раз, когда оператор <=> используется для сравнения экземпляров `Steak`, будет вызываться наш метод! Давайте посмотрим, как это делается.

```
class Steak

  GRADE_SCORES = {"Prime" => 3, "Choice" => 2, "Select" => 1}

  attr_accessor :grade

  def <=>(other)
    if GRADE_SCORES[self.grade] < GRADE_SCORES[other.grade]
      return -1
    elsif GRADE_SCORES[self.grade] == GRADE_SCORES[other.grade]
      return 0
    else
      return 1
    end
  end
end

first_steak = Steak.new
first_steak.grade = "Prime"
second_steak = Steak.new
second_steak.grade = "Choice"

puts first_steak <=> second_steak
puts second_steak <=> first_steak
```

```
1
-1
```

Если атрибут grade текущего экземпляра Steak меньше атрибута grade другого экземпляра Steak, вернуть -1.

Если значения равны, вернуть 0.

В противном случае значение атрибута grade текущего экземпляра Steak больше, чем у другого экземпляра, поэтому возвращаем 1.

Если экземпляр Steak слева от оператора <=> «больше» экземпляра справа, будет получен результат 1. Если они равны, результат равен 0. Если же левый экземпляр Steak «меньше» правого, будет получен результат `-1`.

Конечно, код, в котором повсюду используется <=>, плохо читается. Но теперь, когда мы можем использовать <=> с экземплярами `Steak`, можно включить модуль `Comparable` в класс `Steak`, и методы <, >, <=, >=, == и `between?` автоматически заработают!

Включение Comparable в Steak

Оператор <=> работает с экземплярами `Steak`:

```
puts first_steak <=> second_steak
puts second_steak <=> first_steak
```

```
1
-1
```

...и это все, что необходимо примеси `Comparable` для работы с классом `Steak`. Давайте включим `Comparable` в класс. Для этого достаточно добавить всего одну строку кода:

```
class Steak

  include Comparable

  GRADE_SCORES = {"Prime" => 3, "Choice" => 2, "Select" => 1}

  attr_accessor :grade

  def <=>(other)
    if GRADE_SCORES[self.grade] < GRADE_SCORES[other.grade]
      return -1
    elsif GRADE_SCORES[self.grade] == GRADE_SCORES[other.grade]
      return 0
    else
      return 1
    end
  end

end
```

Все экземпляры Steak теперь содержат Comparable!

После включения `Comparable` все операторы сравнения (и метод `between?`) моментально начинают работать с экземплярами `Steak`:

```
prime = Steak.new
prime.grade = "Prime"
choice = Steak.new
choice.grade = "Choice"
select = Steak.new
select.grade = "Select"

puts "prime > choice: #{prime > choice}"
puts "prime < select: #{prime < select}"
puts "select == select: #{select == select}"
puts "select <= select: #{select <= select}"
puts "select >= choice: #{select >= choice}"
print "choice.between?(select, prime): "
puts choice.between?(select, prime)
```

```
prime > choice: true
prime < select: false
select == select: true
select <= select: true
select >= choice: false
choice.between?(select, prime): true
```

Как работают методы Comparable

Когда оператор сравнения > встречается в вашем коде, метод > вызывается для объекта слева от оператора >, а объект справа от > передается в аргументе `other`.

```
prime = Steak.new
prime.grade = "Prime"
choice = Steak.new
choice.grade = "Choice"

puts prime > choice
```

Вызывает метод > для экземпляра Steak из переменной «prime».

Это происходит из-за того, что модуль `Comparable` был включен в ваш класс, поэтому метод > стал доступным как метод экземпляра для всех экземпляров `Steak`.

```
module Comparable
  ...
  def >(other)
    (self <=> other) == 1
  end
  ...
end
```

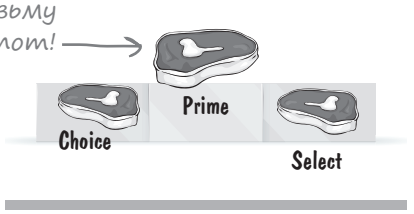
Вызывает метод <=>.

Метод >, в свою очередь, вызывает метод экземпляра <=> (определенный прямо в классе `Steak`) для определения того, какой экземпляр `Steak` имеет «больший» атрибут `grade`. Метод > возвращает `true` или `false` в зависимости от возвращаемого значения <=>.

```
class Steak

  include Comparable

  GRADE_SCORES = {"Prime" => 3, "Choice" => 2, "Select" => 1}

  attr_accessor :grade

  def <=>(other)
    if GRADE_SCORES[self.grade] < GRADE_SCORES[other.grade]
      return -1
    elsif GRADE_SCORES[self.grade] == GRADE_SCORES[other.grade]
      return 0
    else
      return 1
    end
  end

end
```

Возвращает -1, 0 или 1 методу >.

Я возьму этот!



И в конечном итоге будет выбран самый вкусный стейк!

Методы <, <=, ==, >= и `between?` работают аналогично: они используют метод <=> для определения того, нужно ли возвращать `true` или `false`. Реализуйте метод <=>, включите модуль `Comparable` — и вы получите методы <, <=, ==, >, >= и `between?` автоматически! Неплохо, верно?

Что ж, если *это* вам понравилось, то от модуля `Enumerable` вы будете просто *в восторге*.

У бассейна

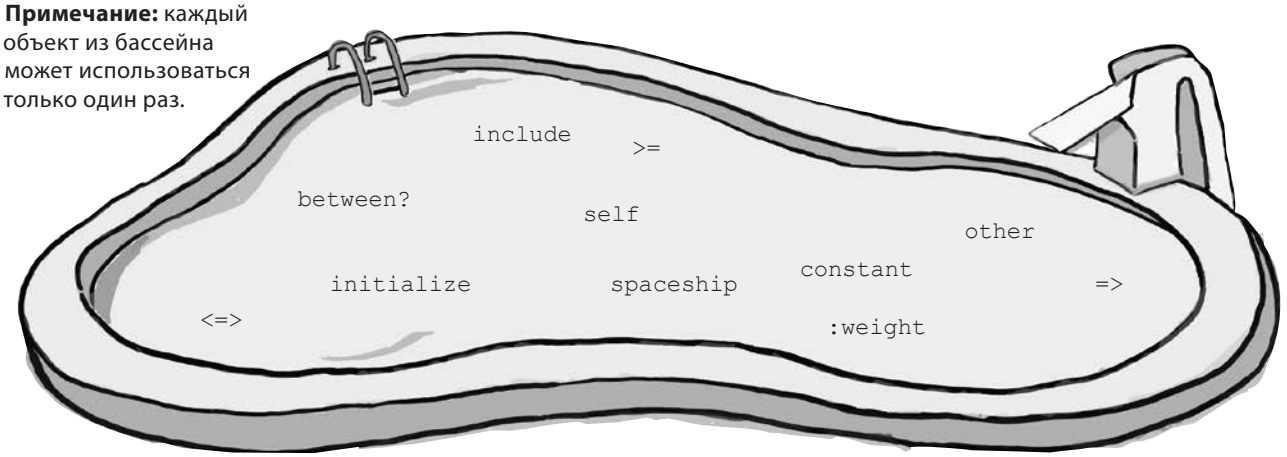
Выловите из бассейна фрагменты кода и расставьте их в пустых местах в коде. Каждый фрагмент может использоваться **только один** раз, причем использовать все фрагменты не обязательно. Ваша **задача** — составить код, который будет нормально выполняться и выдавать приведенный ниже результат.

```
class Apple

  _______ Comparable

  attr_accessor _______

  def ___________(weight)
    _____.weight = weight
  end

  def ___(other)
    self.weight <=> ______.weight
  end

end

small_apple = Apple.new(0.17)
medium_apple = Apple.new(0.22)
big_apple = Apple.new(0.25)

puts "small_apple > medium_apple:"
puts small_apple > medium_apple
puts "medium_apple < big_apple:"
puts medium_apple < big_apple
```

Результат:

```
File Edit Window Help Pie
small_apple > medium_apple:
false
medium_apple < big_apple:
true
```

Примечание: каждый объект из бассейна может использоваться только один раз.



У бассейна. Решение

```
class Apple

  include Comparable

  attr_accessor :weight

  def initialize(weight)
    self.weight = weight
  end

  def <=>(other)
    self.weight <=> other.weight
  end

end

small_apple = Apple.new(0.17)
medium_apple = Apple.new(0.22)
big_apple = Apple.new(0.25)

puts "small_apple > medium_apple:"
puts small_apple > medium_apple
puts "medium_apple < big_apple:"
puts medium_apple < big_apple
```

Результат:

```
File Edit Window Help
small_apple > medium_apple:
false
medium_apple < big_apple:
true
```

Следующая примесь

Помните замечательный метод `find_all` из главы 6? Тот, который легко позволял отобрать элементы из массива по произвольно заданному критерию?

```
relevant_lines = lines.find_all { |line| line.include?("Truncated") }
```

Сокращенная версия кода работает точно так же, как предыдущая: в новый массив копируются только строки, содержащие подстроку «`Truncated`»!

```
puts relevant_lines
```

```
Normally producers and directors would stop this kind of
garbage from getting published. Truncated is amazing in that
it got past those hurdles.
    --Joseph Goldstein, "Truncated: Awful," New York Minute
Truncated is funny: it can't be categorized as comedy,
romance, or horror, because none of those genres would want
to be associated with it.
    --Liz Smith, "Truncated Disappoints," Chicago Some-Times
I'm pretty sure this was shot on a mobile phone. Truncated
is astounding in its disregard for filmmaking aesthetics.
    --Bill Mosher, "Don't See Truncated," Topeka Obscurant
```

А помните невероятно полезные методы `reject` и `map` из той же главы? Все эти методы пришли из одного источника, и это *не* класс `Array`...

Модуль Enumerable

По аналогии с тем, как строковые и числовые классы Ruby включают `Comparable` для реализации своих методов сравнения, многие классы коллекций Ruby (например, `Array` или `Hash`) включают модуль `Enumerable` для реализации своих методов, работающих с коллекциями. К их числу относятся методы `find_all`, `reject` и `map`, использованные в главе 6, а также еще 47 других:

Включение модуля `Enumerable` в класс добавляет методы для работы с коллекциями.

Методы экземпляра из Enumerable:

```
all?              find_all          none?
any?              find_index        one?
chunk             first             partition
collect           flat_map          reduce
collect_concat    grep              reject
count             group_by          reverse_each
cycle             include?          select
detect            inject            slice_before
drop              lazy              sort
drop_while        map               sort_by
each_cons         max               take
each_entry        max_by            take_while
each_slice        member?           to_a
each_with_index   min               to_h
each_with_object  min_by            to_set
entries           minmax            zip
find              minmax_by
```

Как и в случае с `Comparable`, вы можете включить `Enumerable`, чтобы все эти методы стали доступны в *вашем* классе! Нужно лишь предоставить конкретный метод, который необходим для работы `Enumerable`. Речь идет о методе, с которым вы уже работали в других классах: это метод `each`. Методы `Enumerable` вызывают вашу реализацию `each`, чтобы перебирать элементы вашего класса и выполнять с ними необходимые операции.

Comparable:

- Предоставляет <, >, == и еще три метода.
- Включается классами `String`, `Fixnum` и другими числовыми классами.
- Включающий класс должен предоставить метод <=>.

Enumerable:

- Предоставляет методы `find_all`, `reject`, `map` и еще 47 других методов.
- Включается классами `Array`, `Hash` и другими классами коллекций.
- Включающий класс должен предоставить метод `each`.

В книге не хватит места даже для поверхностного рассмотрения *всех* методов `Enumerable`, но на нескольких ближайших страницах мы опробуем некоторые методы в деле. А в следующей главе, посвященной документации Ruby, вы узнаете, где найти информацию об остальных.

Класс для включения Enumerable

Чтобы поэкспериментировать с модулем `Enumerable`, необходимо найти класс для его включения. Причем первый попавшийся класс не подойдет... класс должен содержать метод `each`.

Вот почему мы создали `WordSplitter` — класс для обработки слов в строке. Его метод `each` разделяет строку по пробелам, чтобы выделить отдельные слова, а затем передает каждое слово блоку. Код получается кратким и выразительным:

```
class WordSplitter

  attr_accessor :string

  def each
    string.split(" ").each do |word|
      yield word
    end
  end

end
```

Для хранения строки, разделяемой на слова.

Будет вызываться методами Enumerable.

Разделяем строку на слова (по пробелам) и обрабатываем каждое слово.

Текущее слово передается блоку, который был передан «each».

Чтобы убедиться в том, что новый метод работает, создайте новый экземпляр `WordSplitter`, назначьте строку и вызовите `each` с блоком, который выводит каждое слово:

```
splitter = WordSplitter.new
splitter.string = "one two three four"

splitter.each do |word|
  puts word
end
```

Строка, разделяемая на слова.

Каждое слово передается блоку в параметре.

Выводим текущее слово.

```
one
two
three
four
```

Метод `each` — это, конечно, хорошо... но нам-то нужны 50 методов из `Enumerable`! К счастью, с имеющимся методом `each` для получения остальных методов достаточно просто включить `Enumerable`...

Включение Enumerable в класс

Не будем тратить время на разговоры — попробуем включить `Enumerable` в класс `WordSplitter`:

```
class WordSplitter

  include Enumerable

  attr_accessor :string

  def each
    string.split(" ").each do |word|
      yield word
    end
  end

end
```

Включаем Enumerable.

Создадим другой экземпляр и зададим его атрибут `string`:

```
splitter = WordSplitter.new
splitter.string = "how do you do"
```

А теперь попробуем вызвать новые методы! Методы `find_all`, `reject` и `map` работают точно так же, как в главе 6, если не считать того, что вместо элементов массива блоку передаются слова! (Потому что именно слова они получают от метода `each` класса `WordSplitter`.)

```
p splitter.find_all { |word| word.include?("d") }
p splitter.reject { |word| word.include?("d") }
p splitter.map { |word| word.reverse }
```

Находит все элементы, для которых блок возвращает true.

Отклоняет элементы, для которых блок возвращает true.

Возвращает массив со всеми возвращаемыми значениями блока.

```
["do", "do"]
["how", "you"]
["woh", "od", "uoy", "od"]
```

Также в вашем распоряжении оказывается множество других методов:

```
p splitter.any? { |word| word.include?("e") }
p splitter.count
p splitter.first
p splitter.sort
```

Этим методам блок не нужен.

Метод возвращает true, если блок возвращает true хотя бы для одного элемента.

Подсчет всех элементов.

Первый элемент.

Отсортированный массив элементов.

```
false
4
"how"
["do", "do", "how", "you"]
```

Итого целых 50 методов! Неплохо для добавления всего *одной* строки в класс!

Внутри модуля Enumerable

Если бы мы решили написать собственную версию модуля `Enumerable`, то каждый метод включал бы вызов метода `each` включающего класса. Методы `Enumerable` используют `each` для получения обрабатываемых данных. А вот как могут выглядеть методы `find_all`, `reject` и `map`:

```
module Enumerable

  def find_all
    matching_items = []
    self.each do |item|
      if yield(item)
        matching_items << item
      end
    end
    matching_items
  end

  def reject
    kept_items = []
    self.each do |item|
      unless yield(item)
        kept_items << item
      end
    end
    kept_items
  end

  def map
    results = []
    self.each do |item|
      results << yield(item)
    end
    results
  end

  ...
end
```

Создаем новый массив для хранения элементов, для которых блок возвращает true.

Обрабатываем каждый элемент.

Элемент передается блоку. Если результат равен true...

...он добавляется в массив.

Создаем новый массив для хранения элементов, для которых блок возвращает false.

Обрабатываем каждый элемент.

Элемент передается блоку. Если результат равен false...

...он добавляется в массив.

Создаем новый массив для хранения возвращаемых значений блока.

Обрабатываем все элементы.

Элемент передается блоку, а возвращаемое значение добавляется в новый массив.

Возвращаем массив возвращаемых значений блока.

И еще очень много методов!

Кроме `find_all`, `reject` и `map`, наша самодельная реализация `Enumerable` должна включать еще много методов. *Очень* много. И все они относятся к работе с коллекциями. И каждый из них включает вызов `each`.

Подобно тому как методы `Comparable` зависят от метода `<=>` для сравнения двух строк, методы `Enumerable` зависят от метода `each` для обработки всех элементов коллекции. Модуль `Enumerable` не содержит собственной реализации `each`; вместо этого реализация предоставляется тем классом, в который он включается.

Эта глава дает лишь отдаленное представление о возможностях модулей `Comparable` и `Enumerable`. Попробуйте поэкспериментировать с другими классами. Помните: если для класса можно написать метод `<=>`, то в него можно включить `Comparable`. А если для класса можно написать метод `each`, то в него можно включить `Enumerable`!

Давайте загрузим класс `WordSplitter` с включенным модулем `Enumerable` в irb и опробуем его в деле!

Шаг 1:

Сохраните следующее определение класса в файле с именем *word_splitter.rb*.

```
class WordSplitter

  include Enumerable

  attr_accessor :string

  def each
    string.split(" ").each do |word|
      yield word
    end
  end

end
```

word_splitter.rb

Шаг 2:

В приглашении командной строки перейдите в тот каталог, в котором был сохранен файл.

Шаг 3:

Запустите irb следующей командой:

```
irb -I .
```

Напоминаем: этот ключ позволяет загружать файлы из текущего каталога.

Шаг 4:

Как и прежде, необходимо загрузить файл с сохраненным кодом Ruby. Введите следующую команду:

```
require "word_splitter"
```

Упражнение (Продолжение)

После того как вы загрузите код класса `WordSplitter`, вы сможете создать сколько угодно экземпляров, задать их атрибуты `string` и протестировать все доступные методы `Enumerable`. Для начала опробуйте следующие методы:

```
splitter = WordSplitter.new
splitter.string = "salad beefcake corn beef pasta beefy"

splitter.find_all { |word| word.include?("beef") }

splitter.reject { |word| word.include?("beef") }

splitter.map { |word| word.capitalize }

splitter.count

splitter.find { |word| word.include?("beef") }

splitter.first

splitter.group_by { |word| word.include?("beef") }

splitter.max_by { |word| word.length }

splitter.to_a
```

Найти все слова, включающие «beef».

Отклонить все слова, включающие «beef».

Получить массив, в котором все слова начинаются с буквы верхнего регистра.

Получить количество слов.

Найти первое слово, включающее «beef».

Получить первое слово.

Разделить слова на два массива: включающие и не включающие «beef».

Найти самое длинное слово.

Получить массив всех слов.

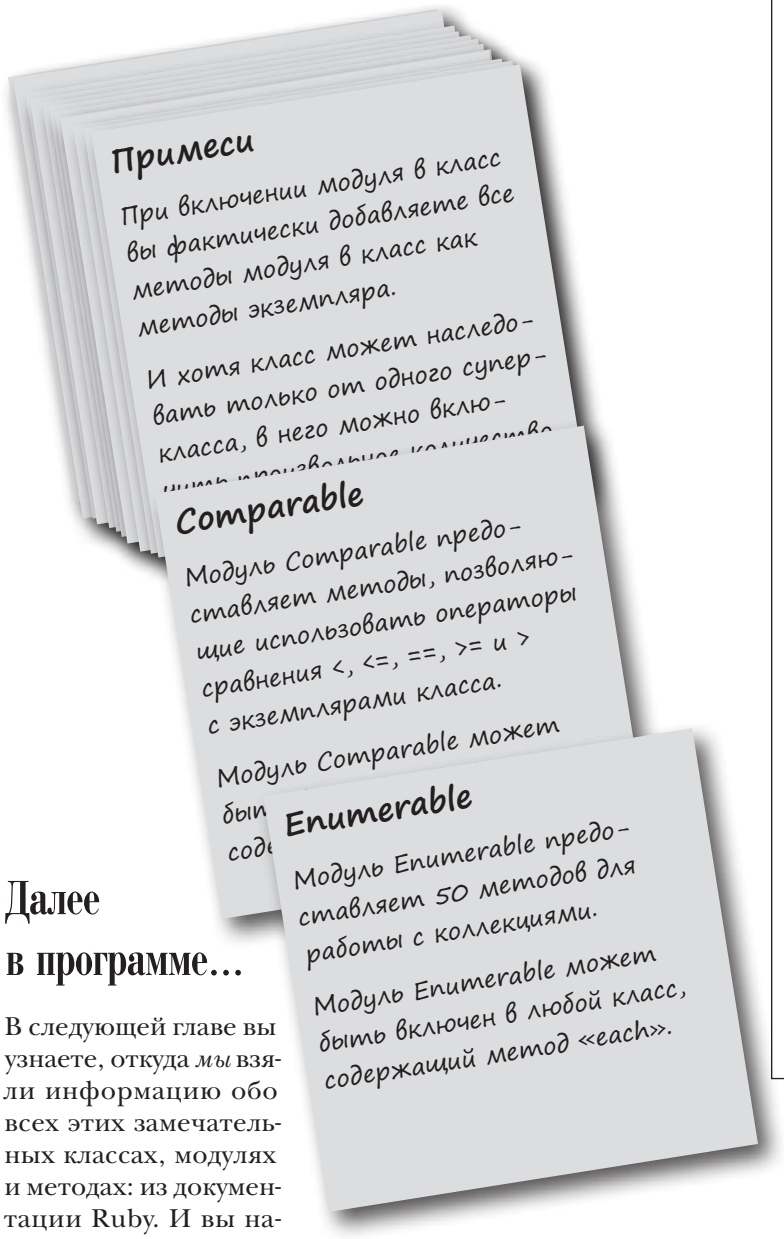
Пример сеанса:

File Edit Window Help Cow

```
$ irb -I .
irb(main):001:0> require "word_splitter"
 => true
irb(main):001:0> splitter = WordSplitter.new
 => #<WordSplitter:0x007fbf6c801eb0>
irb(main):001:0> splitter.string = "salad beefcake corn beef pasta beefy"
 => "salad beefcake corn beef pasta beefy"
irb(main):001:0> splitter.find_all { |word| word.include?("beef") }
 => ["beefcake", "beef", "beefy"]
irb(main):001:0> splitter.reject { |word| word.include?("beef") }
 => ["salad", "corn", "pasta"]
irb(main):001:0> splitter.map { |word| word.capitalize }
 => ["Salad", "Beefcake", "Corn", "Beef", "Pasta", "Beefy"]
irb(main):001:0>
```

Ваш инструментарий Ruby

Глава 10 осталась позади, а ваш инструментарий пополнился модулями `Comparable` и `Enumerable`.



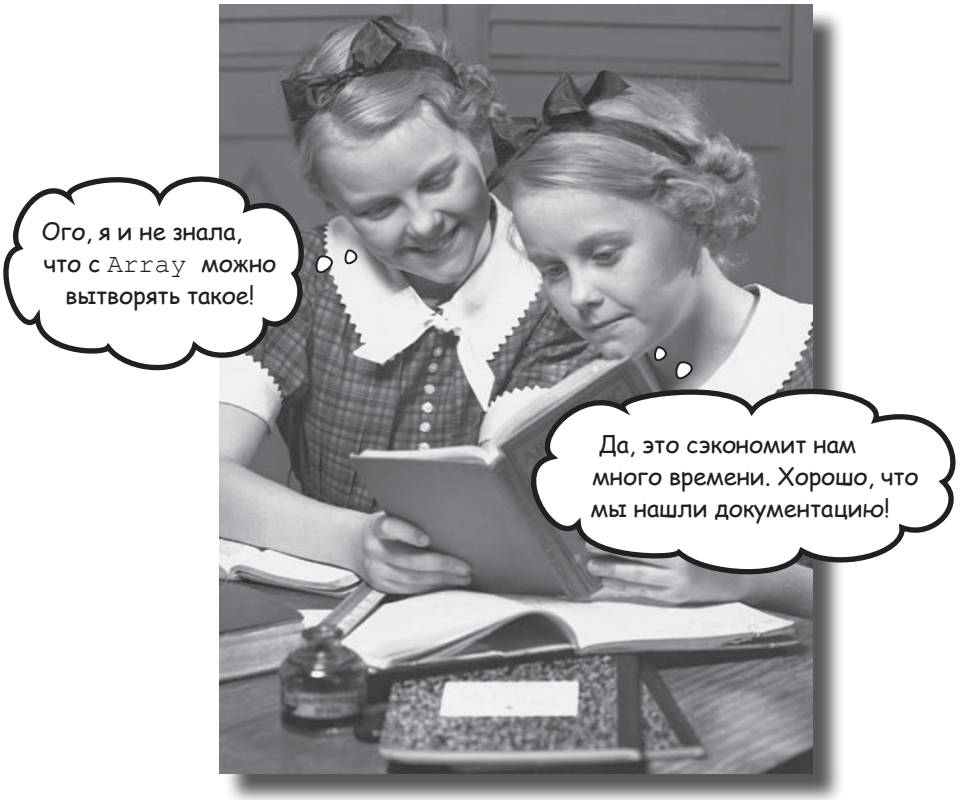
Далее в программе…

В следующей главе вы узнаете, откуда *мы* взяли информацию обо всех этих замечательных классах, модулях и методах: из документации Ruby. И вы научитесь пользоваться документацией!

КЛЮЧЕВЫЕ МОМЕНТЫ

- Константа представляет ссылку на объект, который никогда не изменяется.
- Константу можно определить присваиванием значения — по аналогии с созданием новой переменной.
- Имена констант принято записывать в верхнем регистре, с разделением слов символами подчеркивания (`ALL_CAPS`).
- Методы <, <=, ==, >, >= и `between?` добавляются в классы `Numeric` и `String` включением модуля `Comparable`.
- Оператор <=> («космический корабль») сравнивает выражения в левой и правой части.
- <=> возвращает `-1`, если выражение слева меньше выражения справа; `0`, если два выражения равны; и `1`, если выражение слева больше выражения справа.
- Методы `Comparable` используют оператор <=> для определения того, какой из двух объектов больше (или они равны).
- Многие классы коллекций Ruby (например, `Array` и `Hash`) получают свои методы, относящиеся к работе с коллекциями, из модуля `Enumerable`.
- Методы `Enumerable` вызывают метод `each` включающего класса для получения элементов коллекции.

Читайте документацию



В книге не хватит места, чтобы рассказать все о Ruby. Есть старая поговорка: «Дайте человеку рыбу, и он будет сыт один день. Научите его ловить рыбу, и он будет сыт всю жизнь». До сих пор мы *давали вам рыбу*: показали, как использовать некоторые классы и модули Ruby. Однако существуют десятки других классов и модулей, которые могут пригодиться в вашей работе, но нам не хватит места, чтобы описать их в книге. Пришло время *научить вас ловить рыбу*. Существует превосходная бесплатная **документация** по всем классам, модулям и методам Ruby. Вам просто нужно знать, где ее найти и как ее интерпретировать. Именно об этом пойдет речь в этой главе.

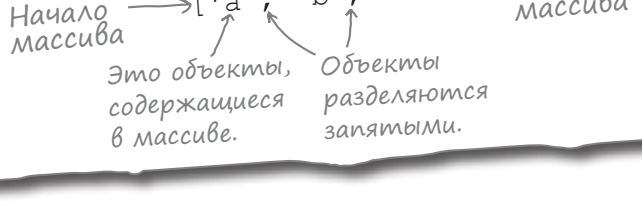
Как узнать больше?

Ваши коллеги рады переходу на Ruby. На них произвели впечатление классы `Array` и `Hash`, возможности модулей `Comparable` и `Enumerable`. Но у одного разработчика возникли сомнения...

Мы знаем только то, что прочитали в книге. Но как нам найти информацию о классах, модулях и методах **самостоятельно?**

Все-все, даже основные возможности... Скажем, класс `Array` был представлен в главе 5. А откуда **вы** узнали о массивах?

Массивы используются для хранения коллекций объектов. Коллекция может иметь произвольный размер.



Хороший вопрос... Пожалуй, пришло время поговорить о документации Ruby.

Базовые классы и модули Ruby

Как мы уже говорили, Ruby включает обширный набор классов и модулей для выполнения самых разнообразных задач программирования. Многие из них автоматически загружаются при каждом запуске Ruby, без необходимости загрузки каких-либо дополнительных библиотек; эти классы и модули Ruby называются **базовыми**.

Базовые классы и модули Ruby загружаются автоматически при каждом запуске Ruby.

Перед вами *неполный* список базовых классов и модулей:

Классы и модули, уже знакомые вам, подчеркнуты.

Array	FalseClass	MatchData	Rational
BasicObject	Fiber	Math	Regexp
Bignum	File	Method	Signal
Binding	FileTest	Module	String
Class	Fixnum	Mutex	Struct
Comparable	Float	NilClass	Symbol
Complex	GC	Numeric	Thread
Dir	Hash	Object	ThreadGroup
Encoding	Integer	ObjectSpace	Time
Enumerable	Interrupt	Proc	TracePoint
Enumerator	IO	Process	TrueClass
Errno	Kernel	Random	UnboundMethod
Exception	Marshal	Range	

Документация

Есть только одно препятствие: нужно узнать, *что делают эти классы* и *как ими пользоваться*. В этой главе мы покажем, как получить нужную информацию из *документации*.

Конечно, вам не придется перерывать сообщения в блогах или на форумах поддержки. Документация языка программирования имеет конкретный стандартизированный формат. На верхнем уровне в ней перечисляются все доступные классы и модули (с описаниями), а далее приводятся списки всех методов, доступных для каждого класса/модуля, также с описаниями.

Если вам доводилось работать с документацией в других языках, то вы увидите, что документация Ruby выглядит примерно так же. Впрочем, в системе обозначений *есть* свои особенности, о которых необходимо знать. В этой главе вы узнаете, на что следует обратить внимание.

В этой главе рассматриваются следующие темы:

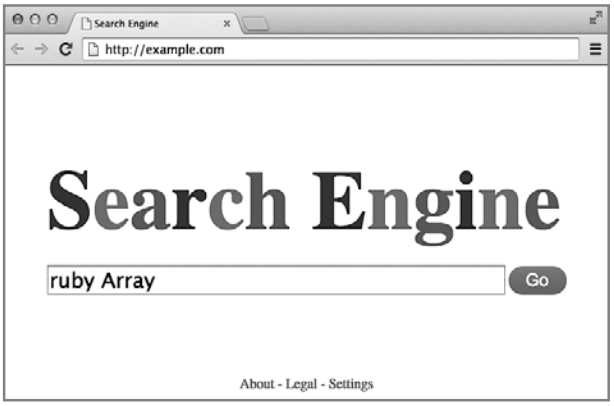
- Документация по классам и модулям.
- Документация по методам.
- Добавление вашей документации в базу Ruby.

Итак, за дело!

Вам не обязательно знать обо всех этих классах и модулях прямо сейчас.

После того как вы научитесь работать с документацией Ruby, мы советуем почитать ее — просто для того, чтобы лучше освоиться. С другой стороны, некоторые из этих классов и модулей вам *никогда* не понадобятся, так что не стоит думать, что вам непременно нужно изучить их от начала до конца.

Документация в формате HTML



Как будет показано позднее в этой главе, разработчики Ruby включают документацию прямо в исходный код с помощью комментариев в особом формате. Существуют программы для извлечения этой документации в разные форматы, среди которых наибольшей популярностью пользуется формат HTML.

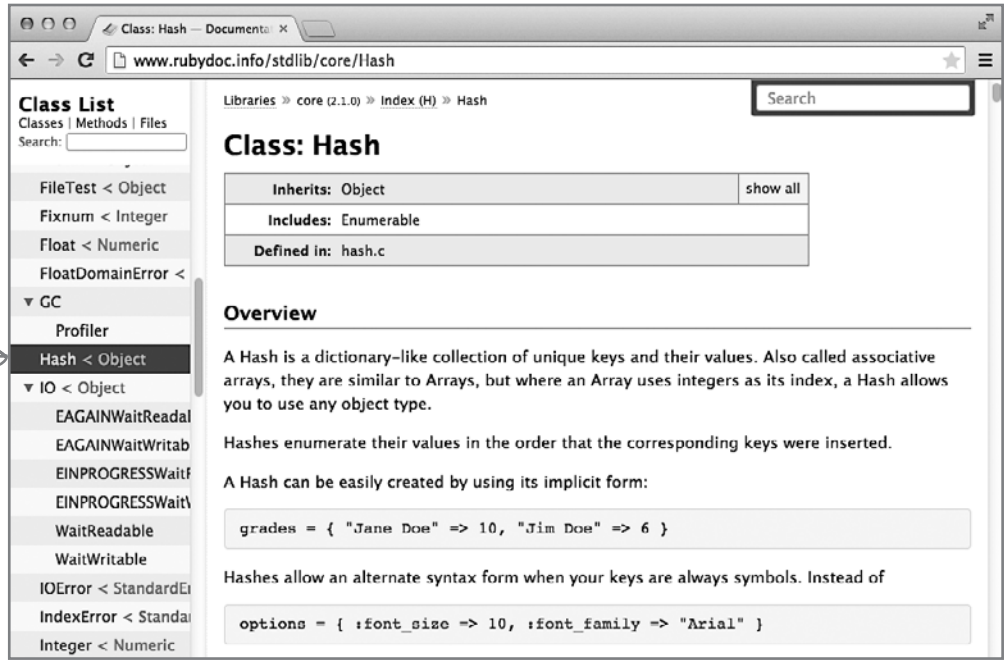
Благодаря сайтам, на которых размещается документация в формате HTML, информация о новых классах и модулях обычно находится прямо под рукой — на расстоянии одного поиска в Интернете. Просто запустите свою любимую поисковую систему и введите строку **`ruby`** и имя класса, модуля или метода, о котором вы хотите больше узнать. (Включение слова *ruby* помогает отфильтровать одноименные классы из других языков программирования.)

Для каждого класса или модуля в документации приводится описание, примеры использования, а также списки методов класса или экземпляра.

Популярные сайты с документацией Ruby:

- *http://docs.ruby-lang.org*
- *http://ruby-doc.org*
- *http://www.rubydoc.info*

Документация класса Hash



Впрочем, чтобы эффективно работать с документацией HTML, необходимо кое-что знать заранее. Мы расскажем об этом на нескольких ближайших страницах.

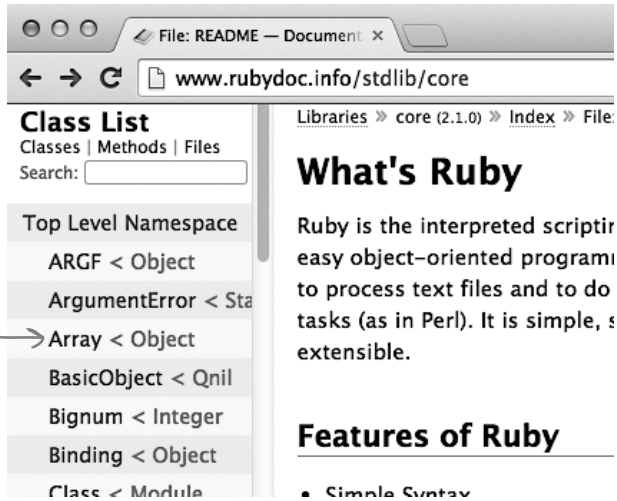
Список доступных классов и модулей

На сайтах с документацией Ruby обычно присутствует список доступных классов и модулей.

Например, если вы откроете в своем браузере следующую страницу:

http://www.rubydoc.info/stdlib/core

...то обнаружите список классов на левой панели. Просто щелкните на интересующем вас классе.

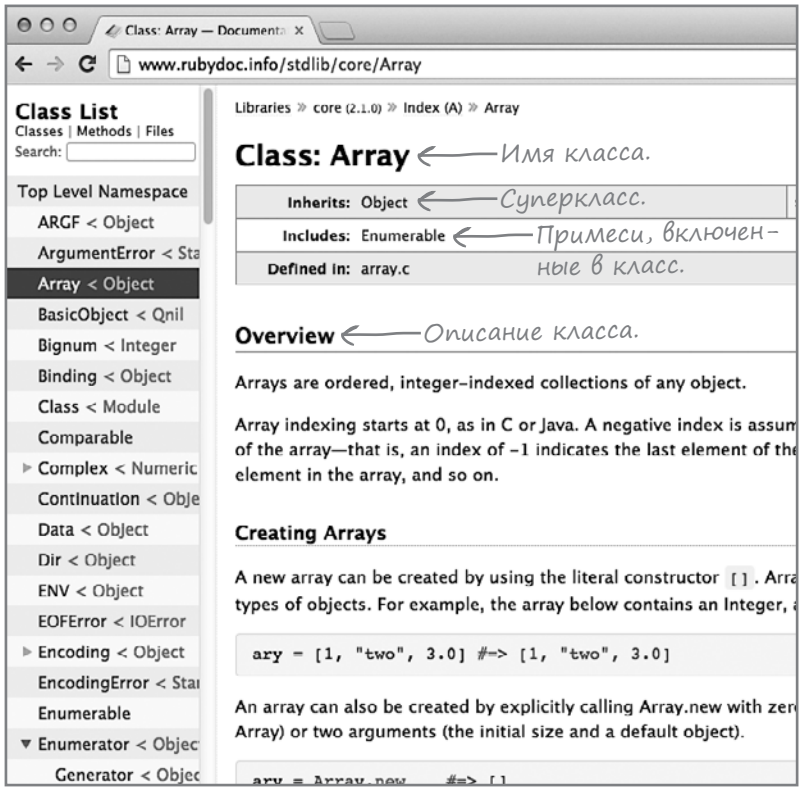


Имя класса представляет собой ссылку; щелкните на нем!

Открывается страница с подробной информацией о выбранном классе, включающая:

- Имя класса.
- Имя суперкласса и ссылку на документацию.
- Ссылки на документацию всех модулей, включаемых классом.
- Описание предназначения класса, обычно с примерами кода, демонстрирующими его использование.

Страница с документацией класса.



Поиск информации о методах экземпляра

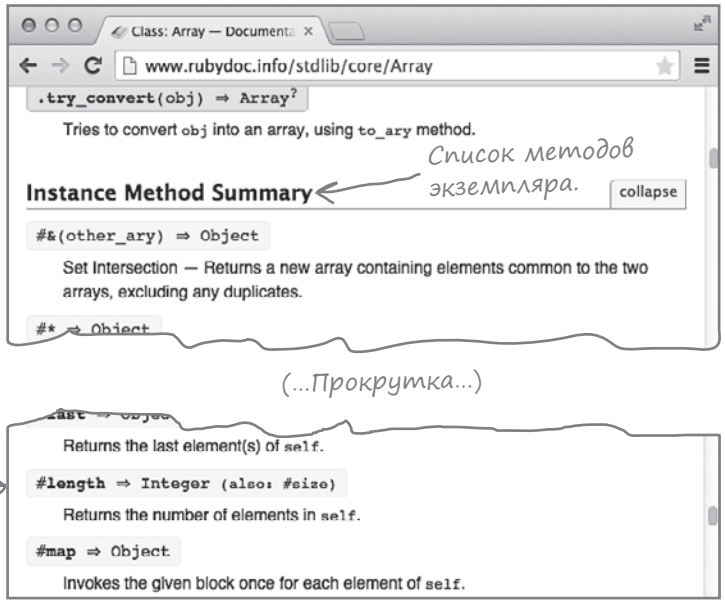
Ладно, документация класса выглядит красиво... Но как насчет методов из этих классов? Например, когда вы показывали, как перебирать элементы массива, откуда вы узнали о методе `length`?

```
index = 0                          # Начинаем с индекса 0.
while index < prices.length        # Перебирать элементы, пока не будет достигнут конец массива.
  puts prices[index]               # Обращение к элементу с текущим индексом.
  index += 1                       # Переход к следующему элементу массива.
end
```

```
3.99
25.0
8.99
```

В документацию классов Ruby включена информация об их методах экземпляра.

На странице документации каждого класса находится список всех методов экземпляра этого класса. (Чтобы увидеть этот список на сайте rubydoc.info, прокрутите страницу вниз.)



Как и в списках имен классов, каждое имя метода представляет собой ссылку на документацию метода. Итак, просто щелкните на имени метода, о котором вы хотите узнать больше.

Щелкните на имени нужного метода.

Методы экземпляра обозначены в документации символом

Это соглашение используется в документации Ruby для обозначения методов экземпляра.

В документации методы экземпляра должны отличаться от методов класса, так как они вызываются по-разному. По этой причине методы экземпляра принято помечать решеткой (#) в начале имени.

Обозначает методы экземпляра.

```
#length ⇒ Integer (also: #size)
    Returns the number of elements in self.
#map ⇒ Object
    Invokes the given block once for each element of self.
```

Итак, если вы встречаете в документации упоминание `Array#length`, то эту запись следует читать «метод экземпляра `length` класса `Array`».

Обратите внимание: символ # используется для обозначения методов экземпляра *в документации*. В коде Ruby символ # открывает комментарий. Не пытайтесь включать в свой код конструкции вида `[1, 2, 3]#length`! В этом случае Ruby будет рассматривать `#length` как комментарий и проигнорирует его.

Будьте осторожны!

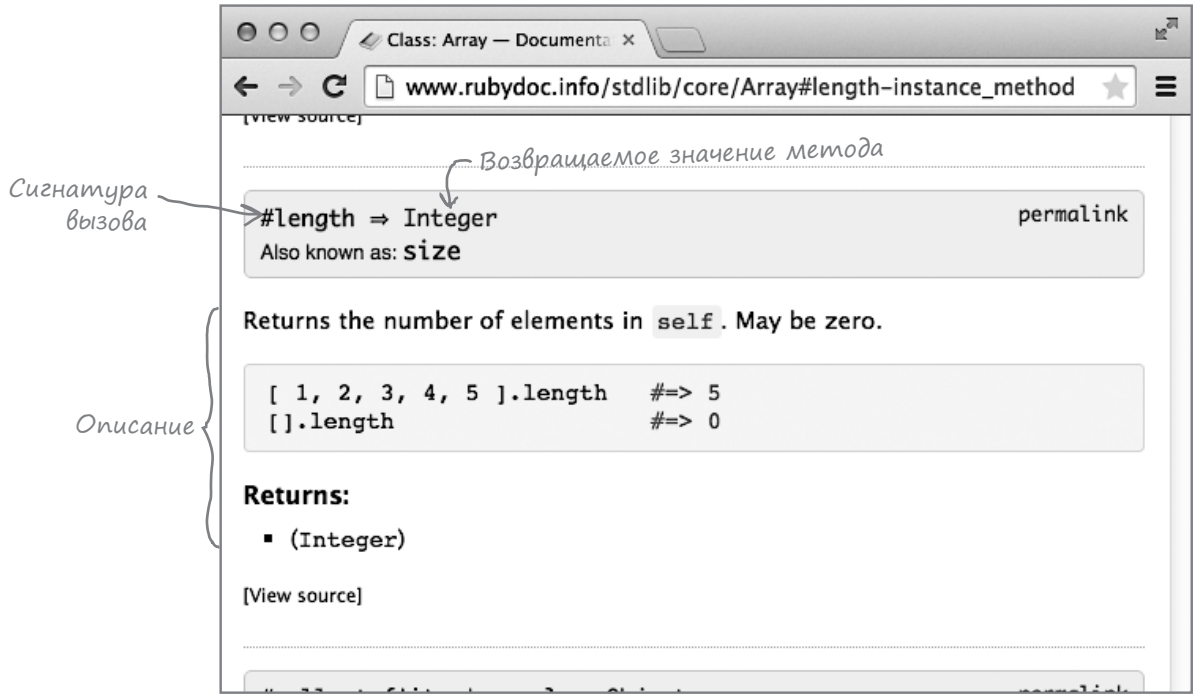
Не используйте символ # для обозначения метода экземпляра в реальном коде!

В коде Ruby символ # является признаком комментария (если он не находится в строке). При попытке использовать # вместо оператора «точка» Ruby решит, что все символы после # образуют комментарий, и скорее всего, выдаст сообщение об ошибке!

Документация по методам экземпляров

Когда вы найдете нужный метод экземпляра и щелкнете на его имени, открывается документация с подробным описанием этого метода.

Например, документация метода экземпляра `length` объектов `Array` выглядит так:



Обычно в документации метода кратко указано, для чего нужен этот метод, а также приводятся примеры кода, демонстрирующие его использование.

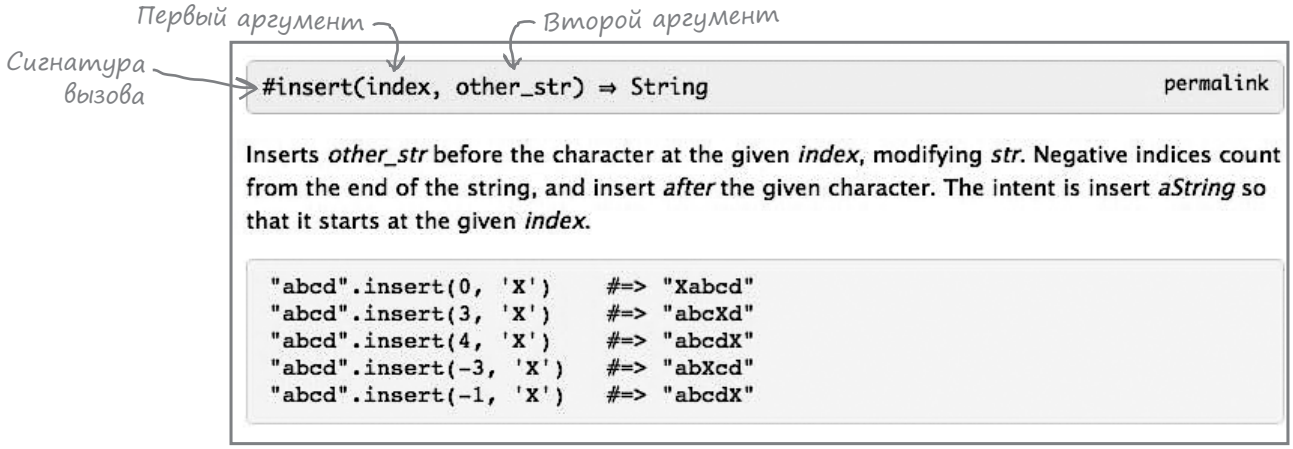
В начале документации метода приводится его *сигнатура вызова*, которая показывает, как должен вызываться данный метод и какое возвращаемое значение следует ожидать.

Если метод не получает никаких аргументов или блока (как метод `length` в приведенном примере), его сигнатура вызова состоит из имени метода и класса возвращаемого значения.

Аргументы в сигнатурах вызова

Если метод *получает* аргументы, то они включаются в сигнатуру вызова.

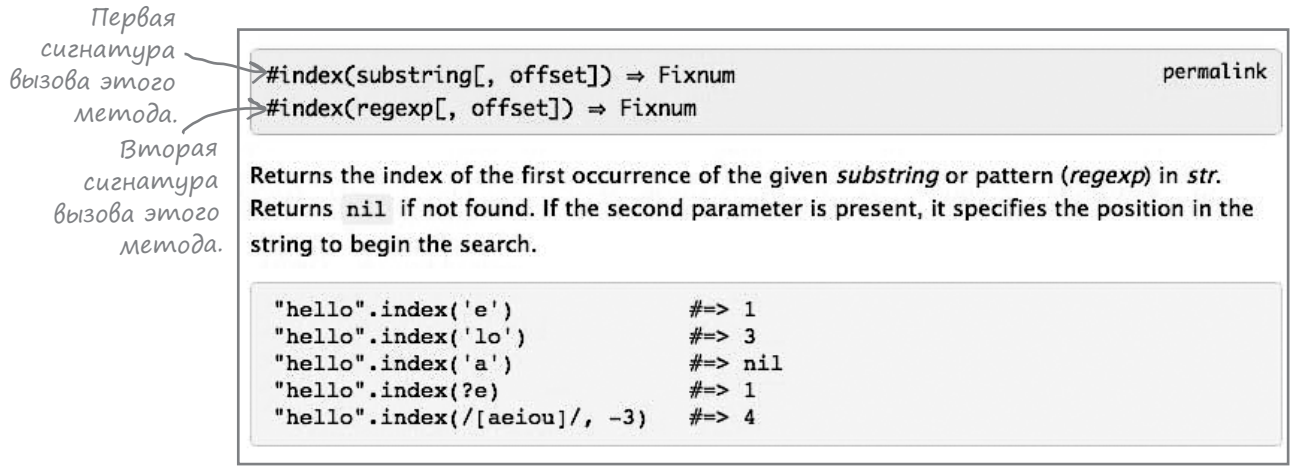
Ниже изображена страница документации для метода экземпляра `insert` класса `String`, который вставляет одну строку в середину другой. В первом аргументе передается целочисленный индекс символа, перед которым вставляется строка, а во втором — сама строка.



Иногда один метод может вызываться несколькими разными способами. В начале документации таких методов указываются несколько вариантов сигнатуры.

Для примера возьмем метод экземпляра `index` для строк. Его первым аргументом может быть либо строка, либо регулярное выражение. (Регулярные выражения кратко рассматриваются в приложении; пока вам достаточно знать, что они относятся к другому классу, нежели строки.) Поэтому в документации приводятся две сигнатуры вызова `index`: для строки и для регулярного выражения в первом аргументе.

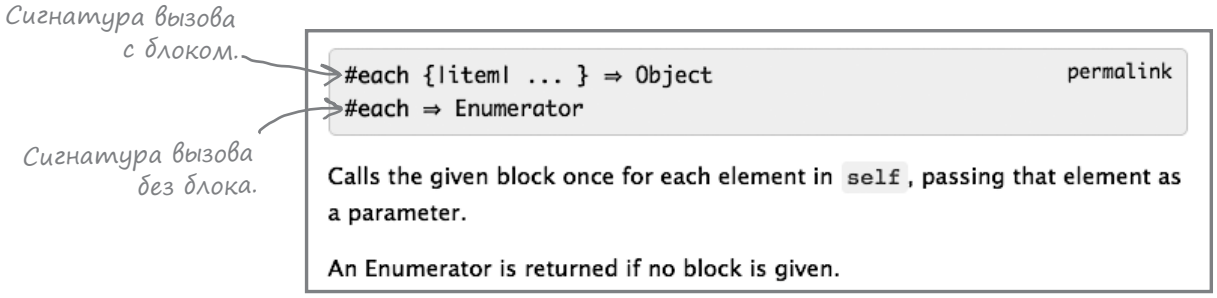
Квадратные скобки (`[]`), в которые заключен второй аргумент, показывают, что этот аргумент не является обязательным.



Блоки в сигнатурах вызова

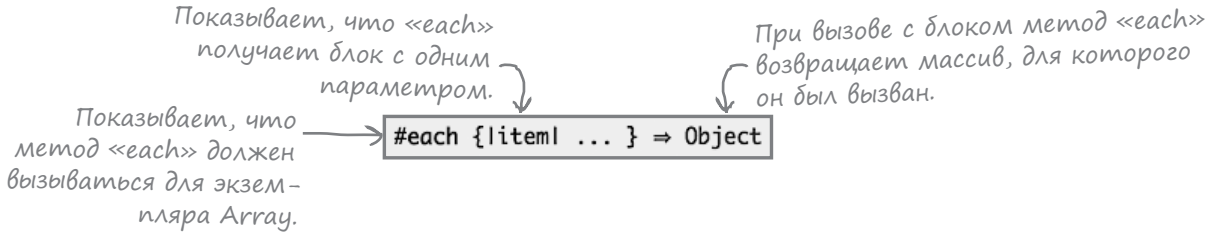
Если метод получает блок, этот факт также будет отражен в сигнатуре вызова. Если блок не является обязательным, то в документации будут указаны сигнатуры как с блоком, так и без него.

Например, в методе экземпляра `each` для массивов блок не обязателен, поэтому в документации присутствуют два вызова (с блоком и без).



Эти два вызова также имеют разные возвращаемые значения, потому что вызов *с* блоком возвращает тот же массив, для которого был вызван метод `each`, тогда как вызов *без* блока возвращает экземпляр класса `Enumerator`. (Этот класс выходит за рамки книги, но по сути он предоставляет средства для перебора коллекций без использования блока.)

Присмотримся повнимательнее к сигнатуре вызова, которая описывает способ вызова метода `each`, использовавшийся в книге. Из сигнатуры видно, что метод получает блок с одним параметром. Он также возвращает объект — массив, для которого был вызван метод `each`.



Обратите внимание: сигнатуры вызова записаны на *псевдокоде*, то есть неформальном коде, который невозможно запустить. Этот код *похож* на код Ruby, но в нем упоминаются имена несуществующих переменных, и обычно он не подходит для вставки в реальную программу. Тем не менее сигнатуры вызова позволяют быстро составить представление о том, как должен использоваться метод.

Также прочитайте документацию суперкласса и примесей!

Я легко нашла документацию по методам экземпляра `length` и `each` класса `Array`. Но вы также показывали метод `find_all`, а его я найти никак не могу!

```
relevant_lines = lines.find_all { |line| line.include?("Truncated") }
```

Сокращенная версия кода работает точно так же, как предыдущая: в новый массив копируются только строки, содержащие подстроку «Truncated»!

```
puts relevant_lines
```

```
Normally producers and directors would stop this kind of
garbage from getting published. Truncated is amazing in that
it got past those hurdles.
    --Joseph Goldstein, "Truncated: Awful," New York Minute
Truncated is funny: it can't be categorized as comedy,
romance, or horror, because none of those genres would want
to be associated with it.
    --Liz Smith, "Truncated Disappoints," Chicago Some-Times
```

Чтобы найти описания методов, взятых из примеси или суперкласса, обратитесь к документации этой примеси или суперкласса.

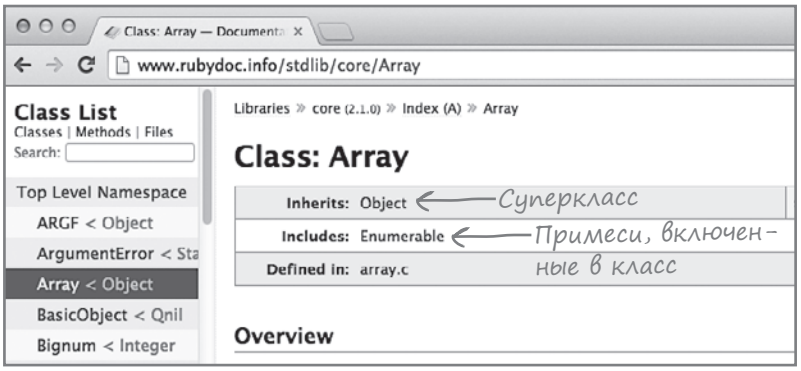
Помните, в главе 10 мы узнали, что метод экземпляра `find_all` класса `Array` появился из примеси `Enumerable`? В документации модуля `Enumerable` также можно найти документацию метода `find_all`!

В документации класса не повторяются методы, унаследованные от суперкласса или внесенные в модуле. (Иначе документация получилась бы слишком объемистой, и в ней было бы слишком много повторений.) Итак, когда вы ищете информацию о новом классе, обязательно прочитайте документацию его суперкласса и всех включаемых модулей!

Также прочитайте документацию суперкласса и примесей! (продолжение)

В документацию HTML также включены удобные ссылки на документацию суперкласса и всех примесей. Щелкните на имени класса или модуля, чтобы перейти к соответствующему разделу.

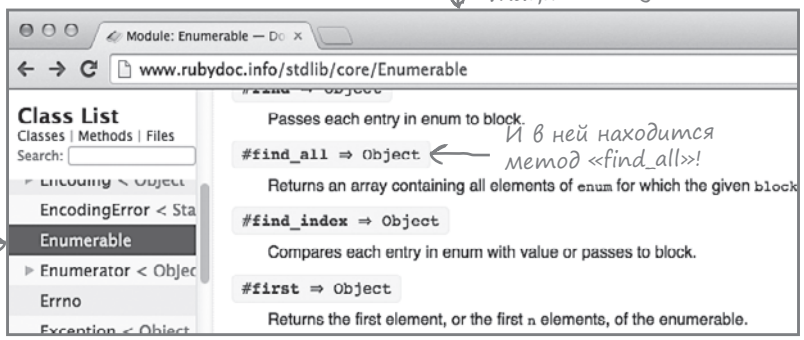
Страница с документацией класса.



Если щелкнуть на ссылке `Enumerable` и прокрутить страницу к списку методов экземпляра модуля, вы найдете в нем метод `find_all`. (И тогда, как и прежде, щелчок на имени метода открывает полную информацию.)

Страница с документацией модуля.

Открывается документация примеси.



Часто Задаваемые Вопросы

В: Метод `find_all` не документирован в классе `Array`, а `map` и несколько других методов `Enumerable` — документированы! Как это понимать?

О: Вы помните, что метод, существующий в классе, переопределяет все одноименные методы примеси? Некоторые базовые классы Ruby включают модуль `Enumerable`, но переопределяют часть его методов по соображениям эффективности. В этом случае документация метода *также* присутствует в классе. Будьте внимательны и сверьтесь с документацией модуля, чтобы ничего не упустить!

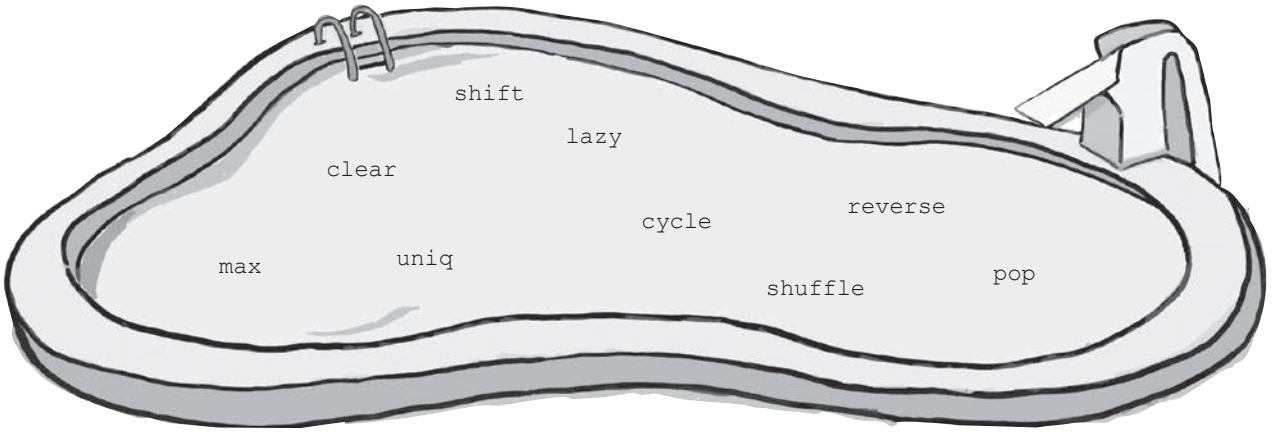
У бассейна

Выловите из бассейна вызовы методов экземпляра и расставьте их в пустых местах в коде. Проблема в том, что все эти методы в книге **не рассматривались (**по крайней мере в подробностях). Откройте страницу *http://www.rubydoc.info/stdlib/core/Array* (или проведите поиск в Интернете по условию «ruby array»), найдите информацию о методах из бассейна и определите, какие из них должны вызываться для получения указанного результата. Не забудьте заглянуть в документацию суперкласса и примеси `Array`! Ваша **задача** — составить код, который будет нормально выполняться и выдавать приведенный ниже результат.

```
array = [10, 5, 7, 3, 9]

first = array._____
puts "We pulled #{first} off the start of the array."

last = array._____
puts "We pulled #{last} off the end of the array."

largest = array._____
puts "The largest remaining number is #{largest}."
```

Результат:

```
We pulled 10 off the start of the array.
We pulled 9 off the end of the array.
The largest remaining number is 7.
```



У бассейна. Решение

```
array = [10, 5, 7, 3, 9]

first = array.shift
puts "We pulled #{first} off the start of the array."

last = array.pop
puts "We pulled #{last} off the end of the array."

largest = array.max
puts "The largest remaining number is #{largest}."
```

Результат:

```
We pulled 10 off the start of the array.
We pulled 9 off the end of the array.
The largest remaining number is 7.
```

Описания методов класса

Найти информацию о методах экземпляра несложно. Но как насчет методов класса? Вроде метода `File.open`, который мы видели в главе 6...

Объект File передается блоку в параметре.

```
File.open("reviews.txt") do |review_file|
  lines = review_file.readlines
end
```

При завершении блока Ruby автоматически закроет файл за вас!

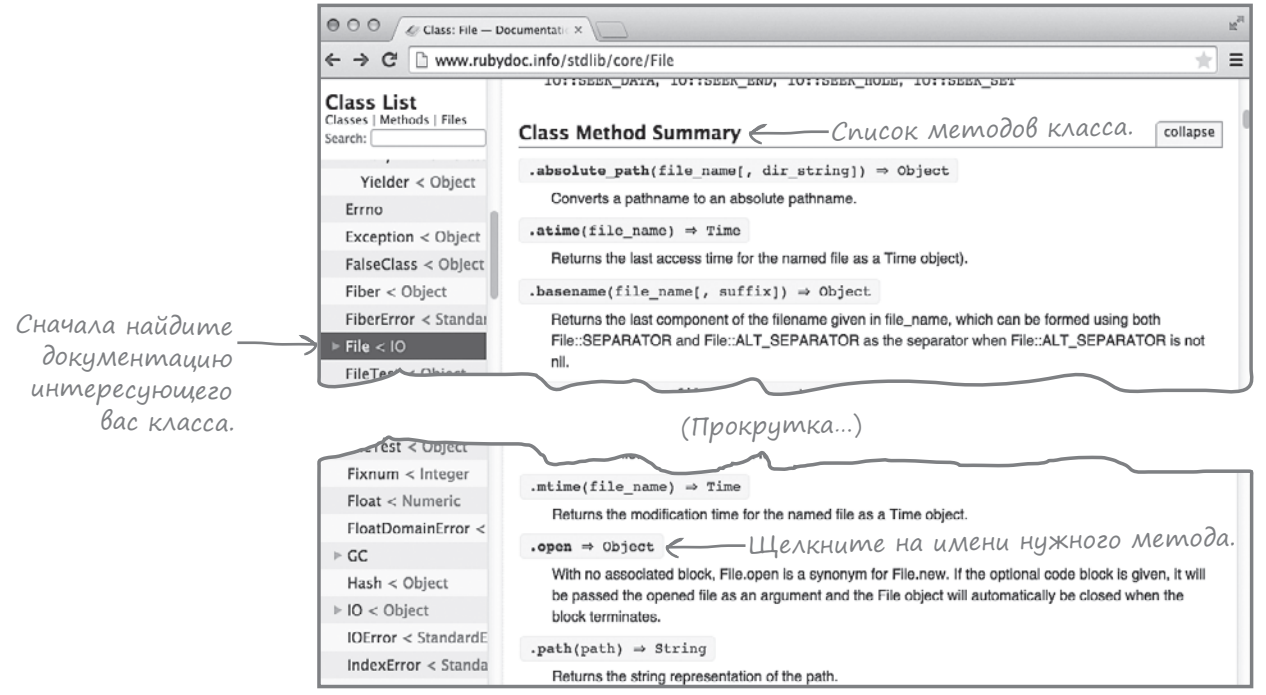
Методы класса объединены в отдельный список на странице документации класса.

Описания методов класса (продолжение)

Поиск документации методов класса мало чем отличается от поиска документации методов экземпляра, с парой исключений:

- Методы класса не смешиваются с методами экземпляра, а образуют отдельный список.
- Методы класса помечаются другим префиксным символом.

Итак, если вы захотите найти описание метода класса `File.open`, откройте документацию класса `File` и прокрутите страницу до списка методов класса. (Он находится непосредственно перед списком методов экземпляра.) Затем найдите в списке метод `open` и щелкните на его имени, чтобы перейти к подробной документации.



Очень важно, чтобы разработчик при работе с документацией сразу отличал методы класса от методов экземпляра: ведь метод `open` должен вызываться для класса `File` в целом, а не для его отдельного экземпляра. Если методы экземпляра помечаются префиксом `#`, то для обозначения методов класса в качестве префикса используется оператор «точка» (`.`), как в приведенном примере.

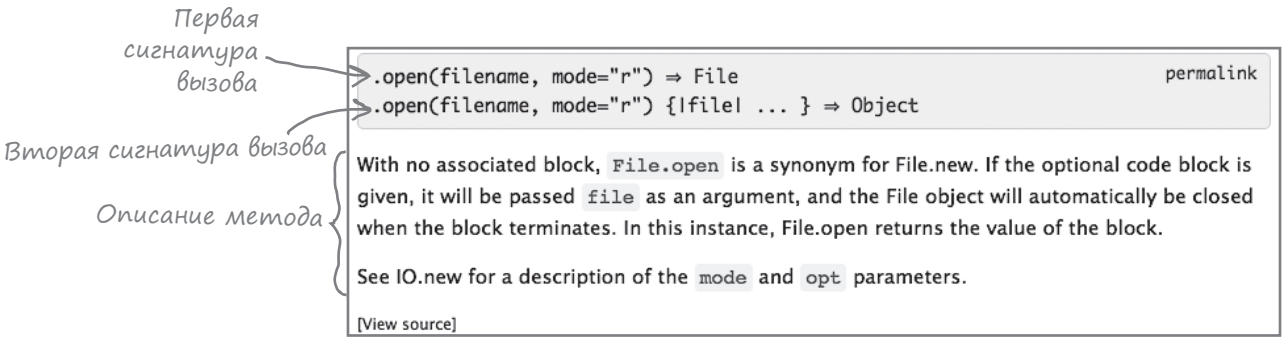
(Как вы вскоре увидите, в другой документации HTML методы класса помечаются оператором `::`, используемым для обращения к константам или методам внутри класса или модуля. Если перед именем метода стоят символы `.` или `::`, значит, перед вами метод класса.)

Документация методов класса

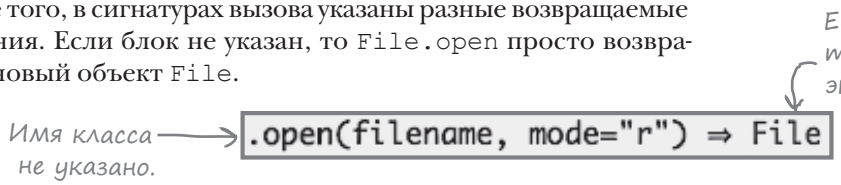
После того как вы щелкнете на ссылке в списке методов класса для просмотра информации о конкретном методе, открывшаяся документация мало чем отличается от документации методов экземпляра.

В верхней части описания приведены сигнатуры вызовов метода на псевдокоде. Напомним, что имя метода помечается префиксом `.` вместо префикса `#`, используемого для методов экземпляра. Имя класса в сигнатурах вызова часто не указывается, но вы должны включать его в свой код. За сигнатурами вызова следует описание метода, а также обычно примеры кода, демонстрирующие его использование.

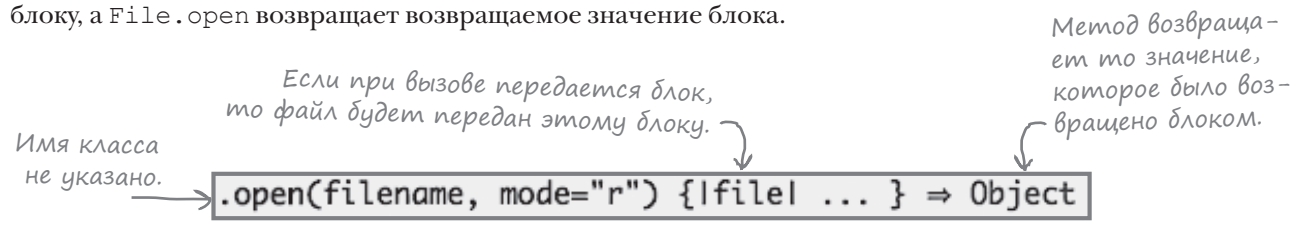
Перед вами документация метода `File.open`. Передавать блок при вызове `File.open` не обязательно, поэтому приведены две сигнатуры вызова: с блоком и без.



Кроме того, в сигнатурах вызова указаны разные возвращаемые значения. Если блок не указан, то `File.open` просто возвращает новый объект `File`.



Но если при вызове передается блок, то объект `File` передается блоку, а `File.open` возвращает возвращаемое значение блока.



Документация несуществующих классов?!

Я в замешательстве. Я нашла в Интернете документацию для класса `Date` — кажется, очень полезного...

Class: Date

Inherits:	Object
Includes:	Comparable

Документация класса Date.

И верно, класс `Date` широко используется в Ruby.

Класс `Date` содержит метод класса `today`, который создает объект, представляющий текущую дату...

Документация метода класса «today»

```
.today([start = Date::ITALY]) ⇒ Object

Date.today #=> #<Date: 2011-06-11 ..>

Creates a date object denoting the present day.
```

...а также методы экземпляра `year`, `month` и `day`, предназначенные для получения года, месяца и дня.

Документация метода экземпляра «month».

Документация метода экземпляра «day».

```
#year ⇒ Integer

Returns the year.
```

Документация метода экземпляра «year».

```
#mon ⇒ Fixnum
#month ⇒ Fixnum

Returns the month (1–12).
```

```
#mday ⇒ Fixnum
#day ⇒ Fixnum

Returns the day of the month (1–31).
```

Прекрасно, но когда я пытаюсь использовать класс `Date` в своем коде, я получаю сообщение об ошибке! Что происходит?

```
today = Date.today
puts "#{today.year}-#{today.month}-#{today.day}"
```

Ошибка

```
uninitialized constant Date (NameError)
```

Этот класс не входит в число базовых классов Ruby, а это означает, что он не загружается автоматически.

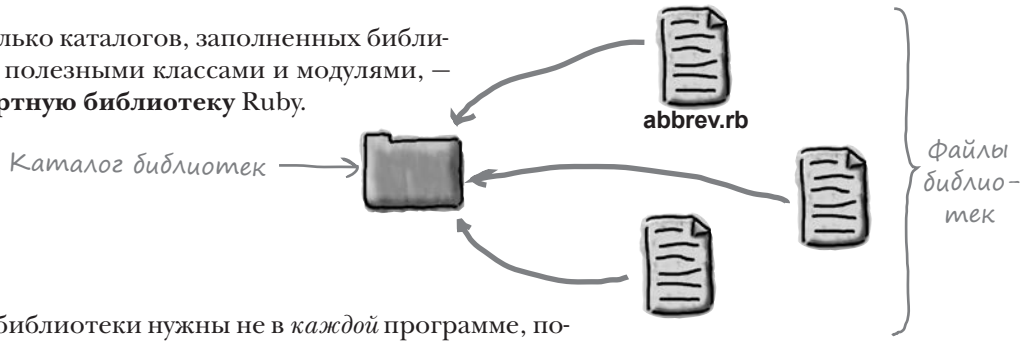
Стандартная библиотека Ruby

Стандартная библиотека — набор классов и модулей, распространяемых в составе Ruby, но не загружаемых автоматически при запуске.

В начале главы мы упоминали о том, что «базовые» классы и модули Ruby загружаются в память при каждом выполнении программы. Но если бы при этом загружались *все* доступные классы и модули, это было бы ужасно; запуск Ruby занимал бы намного больше времени и расходовал бы намного больше памяти. По этой причине те классы или модули, которые не будут с большой вероятностью использоваться в каждой программе, не загружаются автоматически.

Вместо этого программы, которым понадобятся эти специализированные классы, должны загружать их явно. Для загрузки дополнительного кода Ruby используется метод `require`.

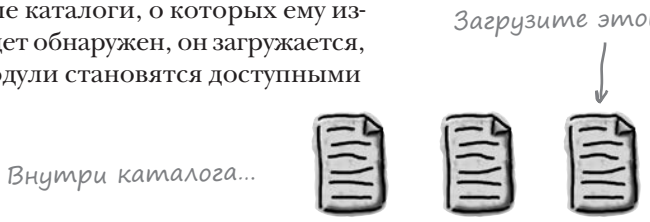
Ruby содержит несколько каталогов, заполненных библиотечными файлами с полезными классами и модулями, — они образуют **стандартную библиотеку** Ruby.



Классы стандартной библиотеки нужны не в *каждой* программе, поэтому они не загружаются автоматически. Тем не менее вы можете легко загрузить их самостоятельно. Включите в свой код вызов `require` и передайте ему строку с именем загружаемого файла. Расширение файла (символы, следующие после точки) можно опустить.

```
require 'date'
```
← *Расширение «.rb» указывать не нужно.*

Ruby просматривает все библиотечные каталоги, о которых ему известно. Когда запрашиваемый файл будет обнаружен, он загружается, а все содержащиеся в нем классы и модули становятся доступными для использования в программе.



Чтобы избавиться от ошибки, просто вызовите метод `require` с передачей строки `'date'`. Тем самым вы загрузите класс `Date`, с которым потом можно будет делать все, что угодно!

```
require 'date'
today = Date.today
puts "#{today.year}-#{today.month}-#{today.day}"
```
← *Загрузите класс Date.*

```
2015-10-17
```

Стандартная библиотека Ruby (продолжение)

Стандартная библиотека Ruby содержит еще много разных классов и модулей. Ниже приведен неполный список:

Файл для require	Классы/модули
'abbrev'	Abbrev
'base64'	Base64
'benchmark'	Benchmark
'bigdecimal'	BigDecimal
'cgi'	CGI
'complex'	Complex
'coverage'	Coverage
'csv'	CSV
'curses'	Curses
'date'	Date DateTime
'dbm'	DBM
'delegate'	Delegator
'digest/md5'	Digest::MD5
'digest/sha1'	Digest::SHA1
'drb'	DRb
'erb'	ERB
'fiber'	Fiber
'fiddle'	Fiddle
'fileutils'	FileUtils
'find'	Find
'forwardable'	Forwardable
'getoptlong'	GetoptLong
'gserver'	GServer
'ipaddr'	IPAddr
'irb'	IRB
'json'	JSON

Файл для require	Классы/модули
'logger'	Logger
'matrix'	Matrix
'minitest'	MiniTest
'monitor'	Monitor
'net/ftp'	Net::FTP
'net/http'	Net::HTTP
'net/imap'	Net::IMAP
'net/pop'	Net::POP3
'net/smtp'	Net::SMTP
'net/telnet'	Net::Telnet
'nkf'	NKF
'observer'	Observable
'open-uri'	OpenURI
'open3'	Open3
'openssl'	OpenSSL
'optparse'	OptionParser
'ostruct'	OpenStruct
'pathname'	Pathname
'pp'	PP
'prettyprint'	PrettyPrint
'prime'	Prime
'pstore'	PStore
'pty'	PTY
'readline'	Readline
'rexml'	REXML
'rinda'	Rinda
'ripper'	Ripper

Файл для require	Классы/модули
'rss'	RSS
'set'	Set
'shellwords'	Shellwords
'singleton'	Singleton
'socket'	TCPServer TCPSocket UDPSocket
'stringio'	StringIO
'strscan'	StringScanner
'syslog'	Syslog
'tempfile'	Tempfile
'test/unit'	Test::Unit
'thread'	Thread
'thwait'	ThreadsWait
'time'	Time
'timeout'	Timeout
'tk'	Tk
'tracer'	Tracer
'tsort'	TSort
'uri'	URI
'weakref'	WeakRef
'webrick'	WEBrick
'win32ole'	WIN32OLE
'xmlrpc/client'	XMLRPC::Client
'xmlrpc/server'	XMLRPC::Server
'yaml'	YAML
'zlib'	ZLib

И снова скажем: изучать все это прямо сейчас не обязательно.

Разумеется, вы можете просмотреть описания классов и модулей, которые вас заинтересуют, — но не думайте, что вам нужно изучать их *все*. Этот список приводится здесь лишь для того, чтобы вы получили представление о возможностях стандартной библиотеки.

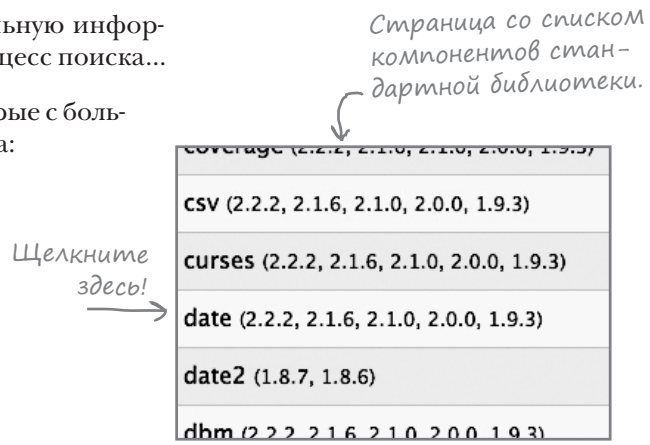
Поиск классов и модулей стандартной библиотеки

Стандартная библиотека Ruby содержит слишком много классов, чтобы мы могли их здесь описать. Как обычно, необходимую информацию легко можно найти в Интернете.

Предположим, вы захотели получить дополнительную информацию о классе `Date`. Вот как может выглядеть процесс поиска...

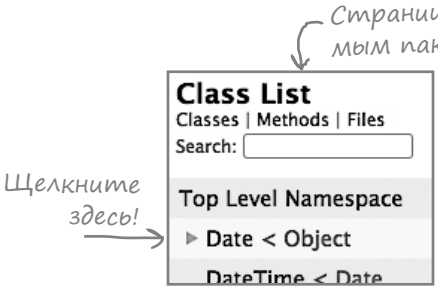
Перед вами одна из многочисленных страниц, которые с большой вероятностью встретятся в результатах поиска:

http://www.rubydoc.info/stdlib



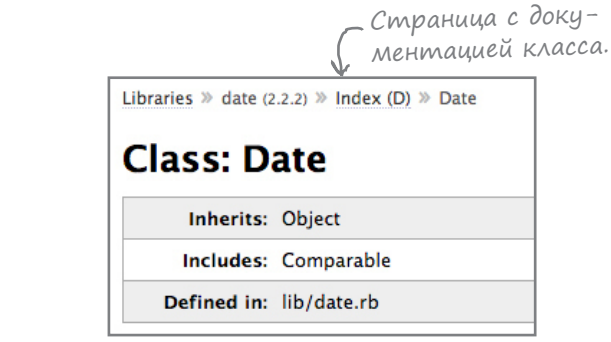
Щелкните здесь!

1. При посещении страницы в левой части выводится список пакетов. Щелкните на интересующем вас пакете; открывается новая страница.



Щелкните здесь!

2. Как и в случае с базовыми классами Ruby, вы увидите список классов и модулей (хотя в данном случае список будет намного меньше, так как он ограничивается содержимым выбранного пакета). Щелкните на интересующем вас классе или модуле.



3. Открывается страница с документацией класса. На этой странице приведена полная информация о доступных методах класса и экземпляра!

Откуда берется документация Ruby: rdoc

Потрясающе! Похоже, из этой документации я смогу узнать все о классах и модулях Ruby. Но как насчет **моего** кода? Как другие люди научатся пользоваться им?

Программа *rdoc*, включенная в поставку Ruby, может использоваться для генерирования документации к вашему коду. Вы передаете rdoc исходный файл с кодом Ruby, а rdoc строит файлы в формате HTML с документацией.

Ниже приведен код класса `WordSplitter`, созданного в главе 10. Попробуем сгенерировать документацию HTML для этого класса при помощи rdoc.

1. Сохраните этот код в файле с именем *word_splitter.rb* (если это не было сделано ранее).

```
class WordSplitter

  include Enumerable

  attr_accessor :string

  def each
    string.split(" ").each do |word|
      yield word
    end
  end

end
```

word_splitter.rb

Откуда берется документация Ruby: rdoc (продолжение)

2 В терминальном окне перейдите в каталог, в котором был сохранен файл. Затем введите следующую команду:

```
rdoc word_splitter.rb
```

Перейдите в каталог, в котором был сохранен файл word_splitter.rb.

Введите команду «rdoc» с именем исходного файла для обработки.

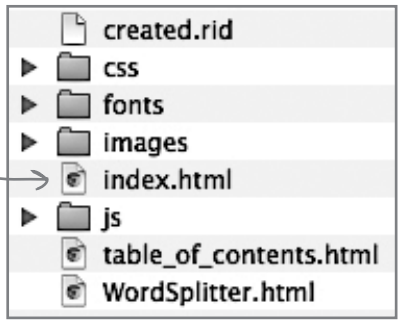
Создается подкаталог с именем doc для хранения документации в формате HTML.

```
File Edit Window Help
$ cd /code
$ rdoc word_splitter.rb
Parsing sources...
100% [ 1/ 1]  word_splitter.rb

Generating Darkfish format into /code/doc...

  Files:      1

  Classes:    1 (1 undocumented)
  Modules:    0 (0 undocumented)
  Constants:  0 (0 undocumented)
  Attributes: 1 (1 undocumented)
  Methods:    1 (1 undocumented)

  Total:      3 (3 undocumented)
    0.00% documented

  Elapsed: 0.1s
$
```

На иллюстрации показано, как выглядит вывод rdoc в процессе обработки кода. Программа создает в текущем каталоге новый подкаталог (по умолчанию ему присваивается имя *doc*) и записывает в него файлы HTML.

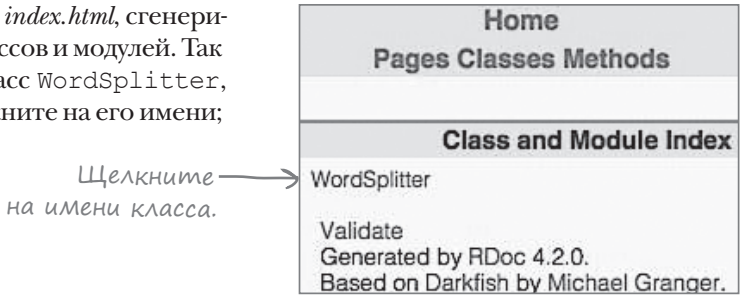
3 Среди файлов, которые rdoc создает в подкаталоге *doc*, вы найдете файл с именем *index.html*. Откройте его в браузере (обычно для этого следует дважды щелкнуть на файле).

Откройте этот файл в браузере.



Какую информацию может вывести rdoc

Первое, что вы увидите при открытии файла *index.html*, сгенерированного программой rdoc, — это список классов и модулей. Так как наш исходный файл содержит только класс `WordSplitter`, это будет единственный класс в списке. Щелкните на его имени; открывается страница с документацией.

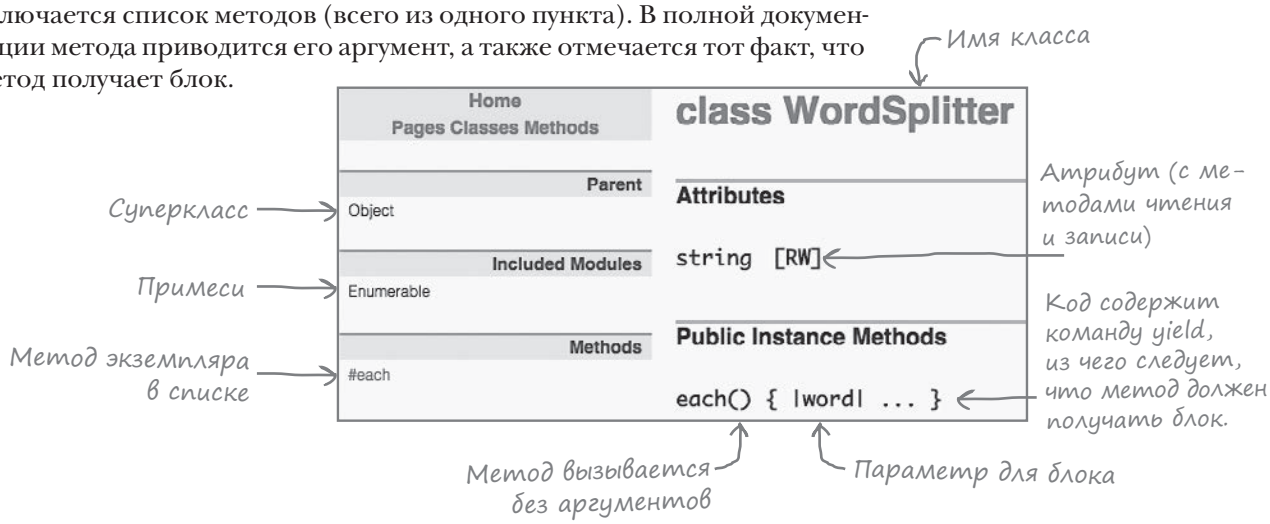


Во время обработки кода rdoc собирает информацию о нем. Конечно, имя класса идентифицируется легко. Менее очевиден тот факт, что суперкласс *не объявляется*, — это означает, что суперклассом `WordSplitter` является класс `Object`; rdoc также отмечает и этот факт. Также выделяются все модули, включаемые в класс, и все объявленные атрибуты (с информацией о том, доступен ли атрибут для чтения и/или записи).

Когда rdoc добирается до определений методов экземпляра, анализ становится *действительно* подробным. Конечно, rdoc запоминает имя метода, а также то, определены ли для метода параметры (в нашем примере их нет). Программа даже заглядывает в тело метода и находит команду `yield`, по наличию которой заключает, что метод ожидает получить блок. Также запоминается имя переменной, передаваемой блоку.

```
class WordSplitter                          # Имя класса

  include Enumerable                        # Примеси

  attr_accessor :string                     # Атрибут (с методами чтения и записи)

  def each                                  # Метод экземпляра
    string.split(" ").each do |word|        # Передача управления блоку
      yield word                            # Аргумент, передаваемый блоку
    end
  end

end
```

Вся собранная информация используется в сгенерированной документации класса: имя класса, суперкласс, примеси, атрибуты. В документацию включается список методов (всего из одного пункта). В полной документации метода приводится его аргумент, а также отмечается тот факт, что метод получает блок.



Добавление комментариев для создания документации

Впрочем, новый пользователь, желающий побольше узнать о классе `WordSplitter`, с таким же успехом может получить всю эту информацию из исходного кода. То, что нам нужно *на самом деле* – это описание класса и его методов на естественном языке.

К счастью, rdoc позволяет легко добавить такие описания в HTML — при помощи самых обычных комментариев Ruby! Встроенная документация находится у разработчика под рукой, где ее удобно обновлять. К тому же она помогает тем людям, которые читают исходный код вместо HTML.

Чтобы включить документацию для класса, просто добавьте комментарии в строках, непосредственно предшествующих ключевому слову `class`. (Если комментарий занимает несколько строк, в документации он объединяется в одну строку.) Атрибуты документируются комментариями в строках, находящихся непосредственно перед объявлением атрибута. Наконец, методы класса и экземпляра документируются в строках, предшествующих ключевому слову `def` для метода.

Описание класса (будет объединено в одну строку).

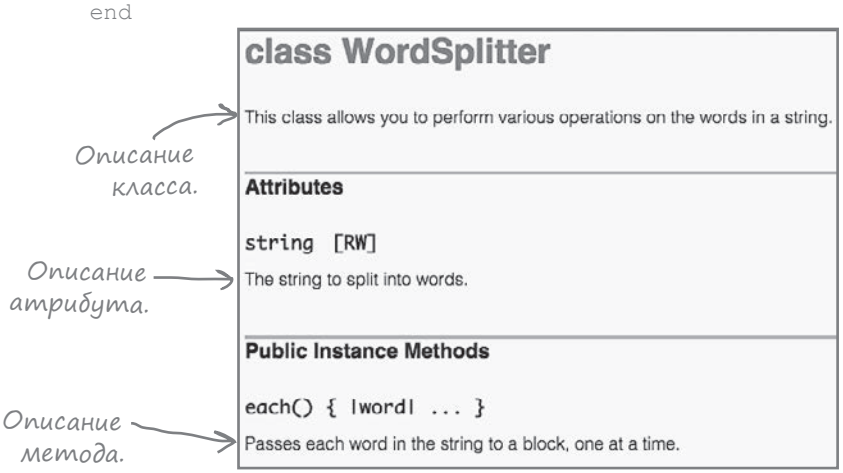
```
# This class allows you to perform various
# operations on the words in a string.
class WordSplitter

  include Enumerable

  # The string to split into words.
  attr_accessor :string
```

Описание атрибута.

Описание метода.

```
  # Passes each word in the string to a block
  # one at a time.
  def each
    string.split(" ").each do |word|
      yield word
    end
  end

end
```

Если повторно запустить rdoc для того же исходного файла, то файлы HTML будут заменены новой версией. Откройте новый файл, и вы увидите в нем описания класса, атрибута и метода, находящиеся в соответствующих разделах!



Описание класса.

Описание атрибута.

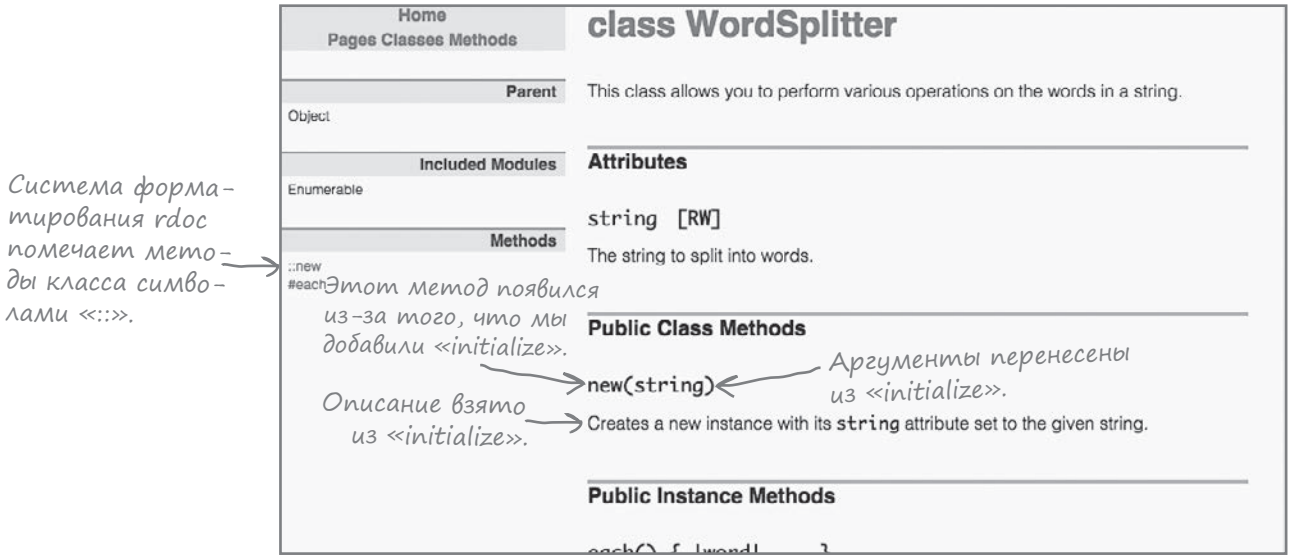
Описание метода.

Метод экземпляра «initialize» представляется методом класса «new»

Как вы уже знаете, если добавить в класс метод `initialize`, он не будет вызываться напрямую. Вместо этого он вызывается через метод класса `new`. Этот особый случай также учитывается в сгенерированной документации. Попробуем добавить в класс `WordSplitter` метод `initialize` с описанием и посмотрим, что произойдет...

```
class WordSplitter
  ...
  # Creates a new instance with its string
  # attribute set to the given string.
  def initialize(string)
    self.string = string
  end
  ...
end
```

Если снова выполнить rdoc и заново загрузить документ HTML, вы увидите, что вместо документирования метода экземпляра `initialize` rdoc добавляет метод класса `new`. (Система форматирования rdoc помечает методы класса в списке методов символами `::`.) Наше описание и аргумент `string` также копируются в метод `new`.

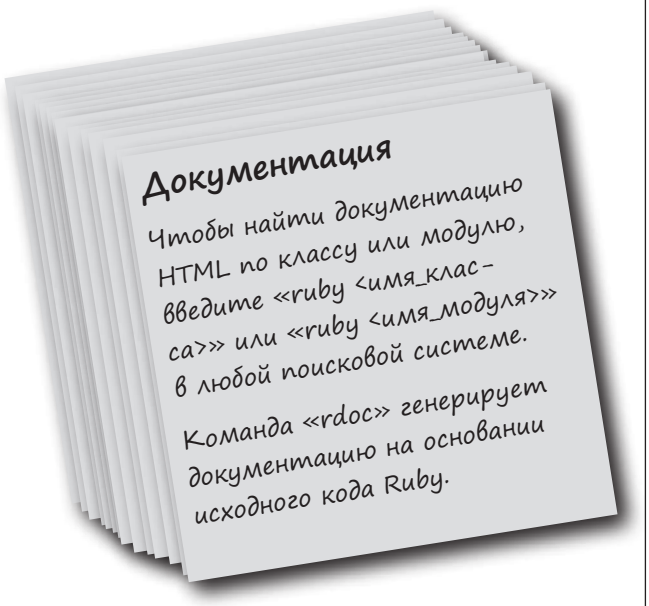


Вот и все! Добавление нескольких простых комментариев в код — все, что требуется для построения удобной, профессиональной документации в формате HTML для ваших классов, модулей и методов.

Если вы хотите, чтобы другие пользователи нашли и использовали ваш код, — без качественной документации не обойтись. Ruby упрощает создание и сопровождение такой документации.

Ваш инструментарий Ruby

Глава 11 осталась позади, а ваш инструментарий Ruby пополнился навыками работы с документацией.



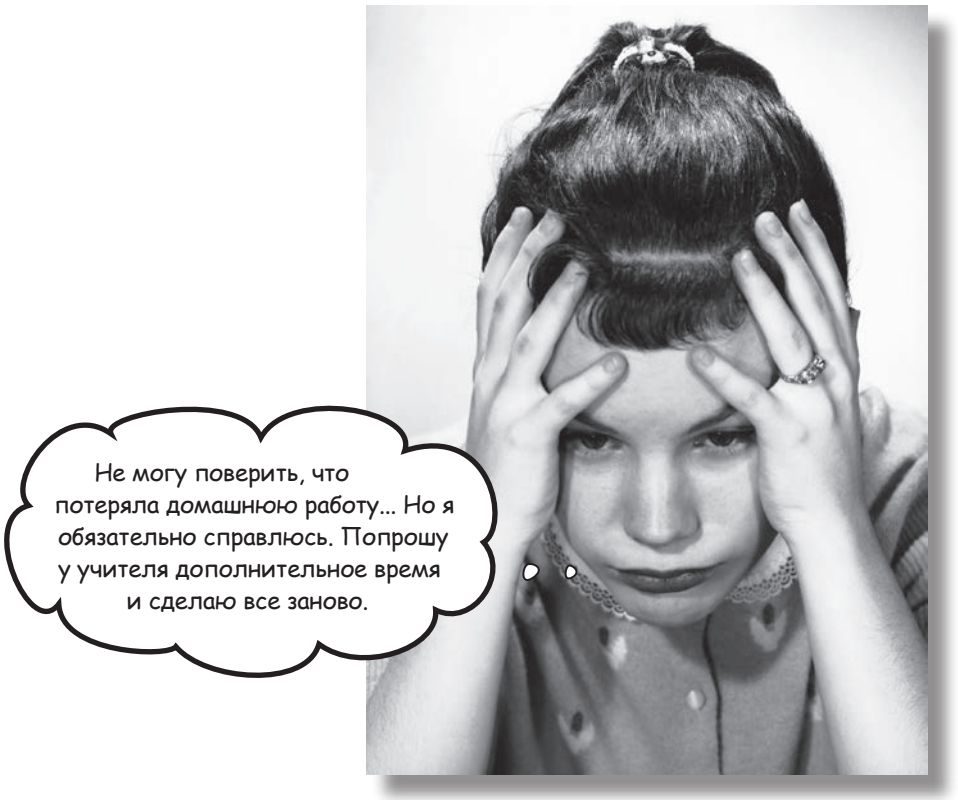
Далее в программе...

Что должна делать программа, если что-то пошло не так? Вероятно, ошибки *возможно* обнаружить и исправить с использованием только тех средств, которые мы вам уже показали, но такой код быстро становится некрасивым и запутанным. В следующей главе мы представим более эффективный механизм обработки аномальных ситуаций: *исключения*.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Базовые классы и модули Ruby загружаются автоматически при каждом запуске Ruby.
- Документация класса в формате HTML включает ссылки на документацию его суперкласса, а также всех включаемых модулей.
- Документация HTML включает списки всех методов класса и экземпляра, а также ссылки на подробную документацию методов.
- Методы экземпляра обычно помечаются «решеткой» (#). Такая запись не может использоваться в коде Ruby; она встречается только в документации.
- Методы класса обычно помечаются оператором «точка» (`.`) или оператором области видимости (`::`).
- Документация метода включает одну или несколько сигнатур вызова, каждая из которых демонстрирует отдельный вариант вызова метода (комбинацию аргументов и блоков).
- Для разных сигнатур вызова метод иногда имеет разные возвращаемые значения. Такие случаи также отражаются в сигнатуре вызова.
- В документацию класса не включаются методы, унаследованные от суперкласса или внесенные в примесях. С другой стороны, документация содержит ссылки на документацию суперкласса и примесей, чтобы вы могли просмотреть полный набор доступных методов.
- В каждую установленную копию Ruby включается стандартная библиотека Ruby: набор классов и модулей, которые не загружаются автоматически, но могут быть загружены вызовом метода `require`.
- Команда `rdoc исходный_файл.rb`, введенная в терминальном окне, генерирует документацию HTML для классов и модулей в исходном файле Ruby.
- Если ваш класс содержит метод экземпляра `initialize`, он будет представлен в документации методом класса `new` (потому что именно так он будет вызываться пользователями).
- Чтобы добавить документацию для класса, модуля, метода чтения/записи атрибута или обычного метода, разместите комментарий непосредственно перед их определением в исходном коде.

Непредвиденное препятствие



В реальном мире порой происходит нечто неожиданное. Кто-то удаляет файл, который пытается загрузить ваша программа, или сервер, с которым программа пытается связаться, выходит из строя. Вы можете проверять такие исключительные ситуации в своем коде, но такие проверки перемешиваются с кодом, выполняемым в ходе нормальной работы. (И в результате возникает жуткая мешанина, в которой невозможно разобраться.)

Эта глава посвящена средствам обработки исключений в Ruby, которые позволяют писать код для обработки аварийных ситуаций и размещать его отдельно от нормального кода.

Не используйте возвращаемые значения методов для передачи информации об ошибках

Всегда существует риск того, что пользователь допустит ошибку при вызове методов в вашем коде. Возьмем хотя бы простой класс, моделирующий духовку. Пользователь создает новый экземпляр, включает духовку вызовом метода `turn_on`, задает атрибуту `contents` нужное блюдо (наш класс моделирует маленькую духовку, в которой помещается только одно блюдо), а затем вызывает метод `bake`, чтобы запекать блюдо до появления золотистой корочки.

Однако пользователь может забыть включить духовку перед вызовом `bake`... Или вызвать `bake`, когда атрибут `contents` содержит `nil`. Мы встроили в программу обработку ошибок для обоих сценариев. Вместо того чтобы возвращать приготовленное блюдо, класс возвращает строку с сообщением об ошибке.

```
class SmallOven

  attr_accessor :contents

  def turn_on
    puts "Turning oven on."
    @state = "on"
  end
  def turn_off
    puts "Turning oven off."
    @state = "off"
  end

  def bake
    unless @state == "on"                                    # Если печь не была включена, предупредить пользователя.
      return "You need to turn the oven on first!"
    end
    if @contents == nil                                      # Если духовка пуста, предупредить пользователя.
      return "There's nothing in the oven!"
    end
    "golden-brown #{contents}"
  end

end
```

Если же пользователь не забудет включить духовку и поставить туда блюдо, все прекрасно работает!

```
dinner = ['turkey', 'casserole', 'pie']
oven = SmallOven.new
oven.turn_on
dinner.each do |item|                  # Обрабатываем каждый пункт меню.
  oven.contents = item                 # Блюдо ставится в духовку.
  puts "Serving #{oven.bake}."         # Блюдо готовится и подается на стол.
end
```

```
Turning oven on.
Serving golden-brown turkey.
Serving golden-brown casserole.
Serving golden-brown pie.
```

Но как вы сейчас убедитесь, такая схема далеко не идеальна. Использование возвращаемого значения метода для передачи информации об ошибке (как в приведенном коде) может создать *еще больше* проблем...

Не используйте возвращаемые значения методов для передачи информации об ошибках (продолжение)

Итак, что произойдет, если мы забудем поставить блюдо в духовку? Если атрибут `contents` содержит `nil`, то наш код «подаст на стол» предупреждающее сообщение!

Забыли поставить блюдо в духовку!

```
dinner = ['turkey', nil, 'pie']
oven = SmallOven.new
oven.turn_on
dinner.each do |item|
  oven.contents = item
  puts "Serving #{oven.bake}."
end
```

Выглядит не-аппетитно...

```
Turning oven on.
Serving golden-brown turkey.
Serving There's nothing in the oven!.
Serving golden-brown pie.
```

Но здесь испорчено только *одно* блюдо. Гораздо хуже, если пользователь забудет включить духовку — тогда испорченным может оказаться *весь обед*!

```
dinner = ['turkey', 'casserole, 'pie']
oven = SmallOven.new
oven.turn_off ← Случайно выключили духовку («off») вместо того, чтобы включить («on»)!
dinner.each do |item|
  oven.contents = item
  puts "Serving #{oven.bake}."
end
```

Да это вообще несъедобно!

```
Turning oven off.
Serving You need to turn the oven on first!.
Serving You need to turn the oven on first!.
Serving You need to turn the oven on first!.
```

Но самая серьезная проблема в другом: при возникновении ошибки программа продолжает работать, словно ничего не произошло. К счастью, в главе 2 вы уже узнали, как решаются подобные проблемы...

Помните метод `raise`? Он прерывает выполнение программы с сообщением об ошибке при обнаружении аномального состояния. Похоже, такой способ намного надежнее передачи сообщений об ошибках в возвращаемых значениях методов.

```
class Dog

  attr_reader :name, :age

  def name=(value)
    if value == ""
      raise "Name can't be blank!"
    end
    @name = value
  end
```

Если значение «value» недействительно...

...выполнение прерывается в этой точке.

Эта команда не будет выполнена, если был вызван метод «raise».

Использование «raise» для передачи информации об ошибках

Попробуем заменить в классе `SmallOven` коды ошибок в возвращаемых значениях вызовами `raise`:

```
class SmallOven

  attr_accessor :contents

  def turn_on
    puts "Turning oven on."
    @state = "on"
  end
  def turn_off
    puts "Turning oven off."
    @state = "off"
  end

  def bake
    unless @state == "on"
      raise "You need to turn the oven on first!"
    end
    if @contents == nil
      raise "There's nothing in the oven!"
    end
    "golden-brown #{contents}"
  end

end
```

Если пользователь пытается приготовить блюдо при выключенной духовке, происходит ошибка.

Если пользователь пытается готовить с пустой духовкой, происходит ошибка.

Эта строка не выполняется, если ранее произошла ошибка.

Если теперь вы попытаетесь готовить с выключенной или пустой духовкой, вместо «подачи на стол» сообщения об ошибке происходит настоящая ошибка...

```
oven = SmallOven.new
oven.turn_off
oven.contents = 'turkey'
puts "Serving #{oven.bake}."
```

Случайно выключили духовку.

```
Turning oven off.
oven.rb:16:in `bake': You need to turn the oven on first! (RuntimeError)
        from oven.rb:29:in `<main>'
```

```
oven = SmallOven.new
oven.turn_on
oven.contents = nil
puts "Serving #{oven.bake}."
```

Духовка пуста.

```
Turning oven on.
oven.rb:19:in `bake': There's nothing in the oven! (RuntimeError)
        from oven.rb:29:in `<main>'
```

Использование «raise» создает новые проблемы

В предыдущей версии, когда мы использовали возвращаемые значения для передачи сообщений об ошибках в методе `bake`, такие сообщения иногда возникали и для нормально приготовленных блюд.

Забыли поставить блюдо в духовку!

```
dinner = ['turkey', nil, 'pie']
oven = SmallOven.new
oven.turn_on
dinner.each do |item|
  oven.contents = item
  puts "Serving #{oven.bake}."
end
```

Выглядит неаппетитно...

```
Turning oven on.
Serving golden-brown turkey.
Serving There's nothing in the oven!.
Serving golden-brown pie.
```

Но у старой программы есть и несомненное достоинство: после «подачи на стол» сообщения об ошибке она хотя бы продолжает подавать другие блюда. А в новой версии, в которой `raise` сообщает об ошибках в методе `bake`, программа завершается сразу же после обнаружения ошибки. Десерта не будет!

Тот же самый код...

```
dinner = ['turkey', nil, 'pie']
oven = SmallOven.new
oven.turn_on
dinner.each do |item|
  oven.contents = item
  puts "Serving #{oven.bake}."
end
```

Выполнение останавливается на «nil», без обработки «pie»!

```
Turning oven on.
Serving golden-brown turkey.
oven.rb:19:in `bake': There's nothing in the oven! (RuntimeError)
        from oven.rb:31:in `block in <main>'
        from oven.rb:29:in `each'
        from oven.rb:29:in `<main>'
```

И сообщение об ошибке выглядит уродливо. Возможно, упоминания номеров строк программы пригодятся разработчикам, но рядового пользователя они только собьют с толку.

Если уж мы собираемся использовать `raise` в методе `bake`, эти недостатки нужно исправить. А для этого необходимо поближе познакомиться с *исключениями*...

Исключения: когда происходит что-то не то

Если просто вызвать `raise` в сценарии, результат будет выглядеть примерно так:

```
raise "oops!"
```

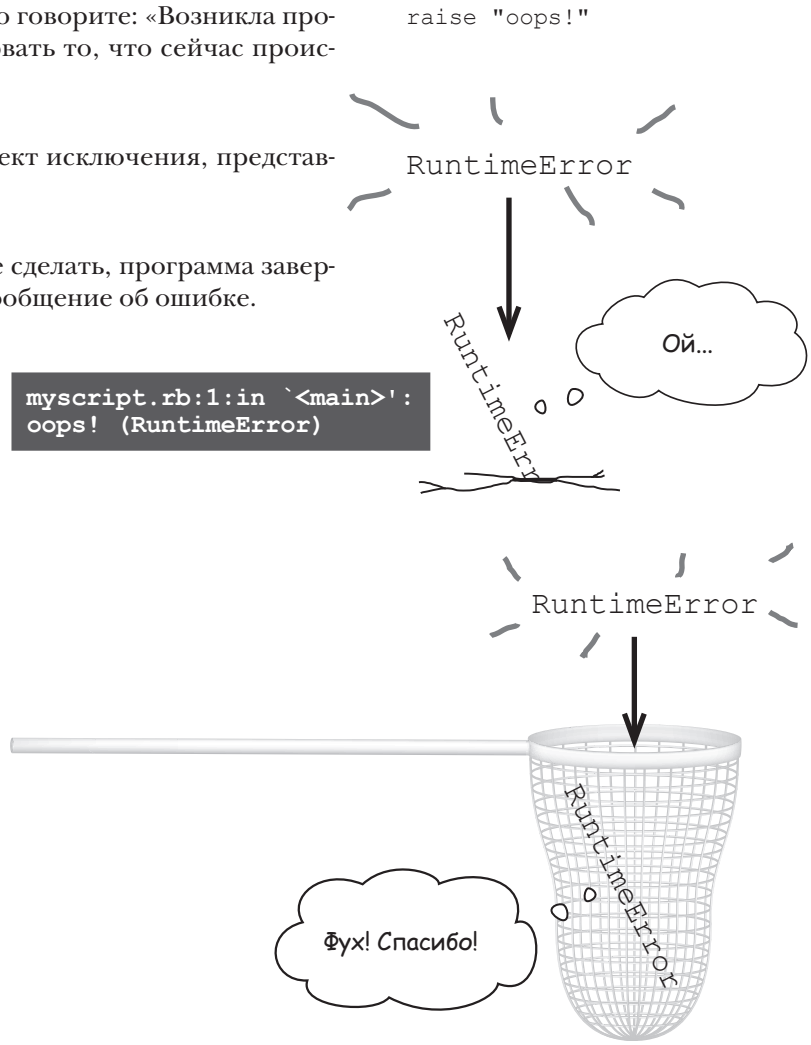
```
myscript.rb:1:in `<main>': oops! (RuntimeError)
```

Метод `raise` в действительности создает объект **исключения** — то есть объект, представляющий ошибку. Если исключение не будет обработано, то программа аварийно завершается.

Исключение — объект, представляющий ошибку.

Вот как это происходит:

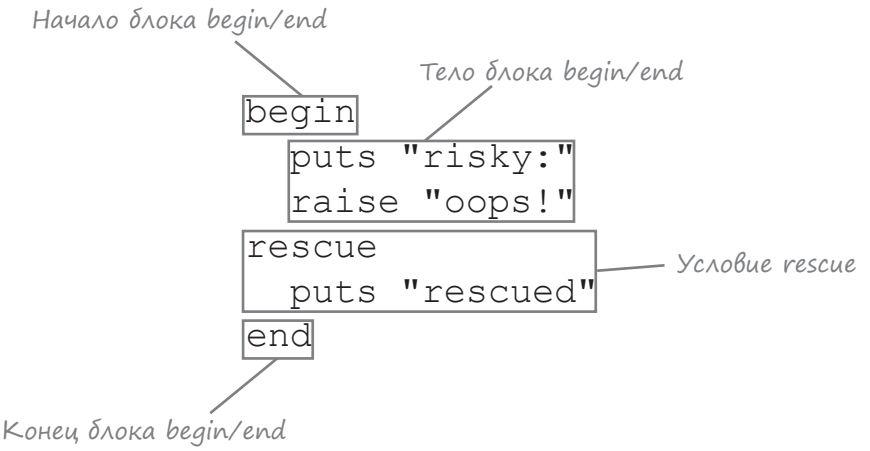
1. Вызывая `raise`, вы словно говорите: «Возникла проблема. Нужно *срочно* прервать то, что сейчас происходит».
2. Метод `raise` создает объект исключения, представляющий ошибку.
3. Если с ошибкой ничего не сделать, программа завершается, а Ruby выводит сообщение об ошибке.



Впрочем, исключение также можно **обработать**: другими словами, ошибка перехватывается программой. Вы можете вывести другое сообщение, более удобное для пользователя, а иногда даже решить проблему и продолжить выполнение.

Условия rescue: пытаемся исправить проблему

Если у вас имеется некий код, в котором могут возникать исключения, можно попытаться заключить его в блок `begin/end` и добавить одно или несколько условий `rescue`, выполняемых при возникновении исключения. Условие `rescue` может содержать код для регистрации ошибки в журнальном файле, попытки повторного создания сетевого подключения... словом, любых действий, направленных на корректное решение проблемы.



Если выражение в теле блока `begin/end` выдает исключение, управление немедленно передается в подходящее условие rescue (если его удается найти).

```
begin
  puts "I'll be run."
  raise "oops!"
  puts "I'll be skipped."
rescue
  puts "Rescued an exception!"
end
```

Выражения в этом блоке нормально выполняются, пока...

...не возникнет исключение!

Весь остальной код пропускается.

А управление передается сюда.

```
I'll be run.
Rescued an exception!
```

После завершения условия rescue выполнение продолжается с кода, следующего за блоком `begin/end`. Предполагается, что вам удалось решить проблему в условии rescue, поэтому завершать программу не нужно.

Как Ruby ищет условие rescue

Исключения *можно* инициировать в основной программе (вне каких-либо методов), но на практике исключения гораздо чаще инициируются *внутри* методов. В таких случаях Ruby сначала ищет условие `rescue` внутри метода. Если поиск оказывается безрезультатным, то метод немедленно завершается (без возвращаемого значения).

```
def risky_method
  raise "oops!"
  puts "I'll be skipped."
end

risky_method
```

Выполнение метода немедленно прерывается.

Никогда не выполняется.


```
myscript.rb:2:in `risky_method': oops! (RuntimeError)
        from myscript.rb:6:in `<main>'
```

При выходе из метода Ruby также ищет условие `rescue` там, откуда был вызван метод. Таким образом, если вы вызываете метод, который (как вы подозреваете) может инициировать исключение, заключите вызов в блок `begin/end` и добавьте условие `rescue`.

```
def risky_method
  raise "oops!"
  puts "I'll be skipped."
end

begin
  risky_method
rescue
  puts "Rescued an exception!"
end
```

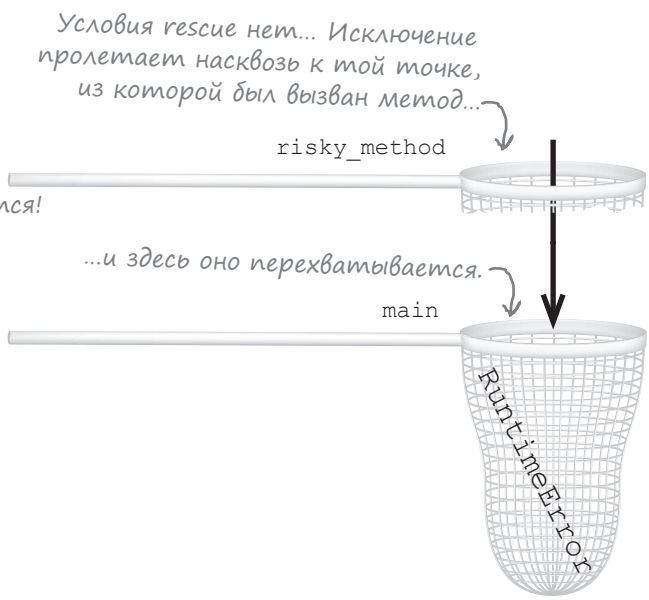
Выполнение метода немедленно прерывается.

Никогда не выполняется.

Инициирует исключение.

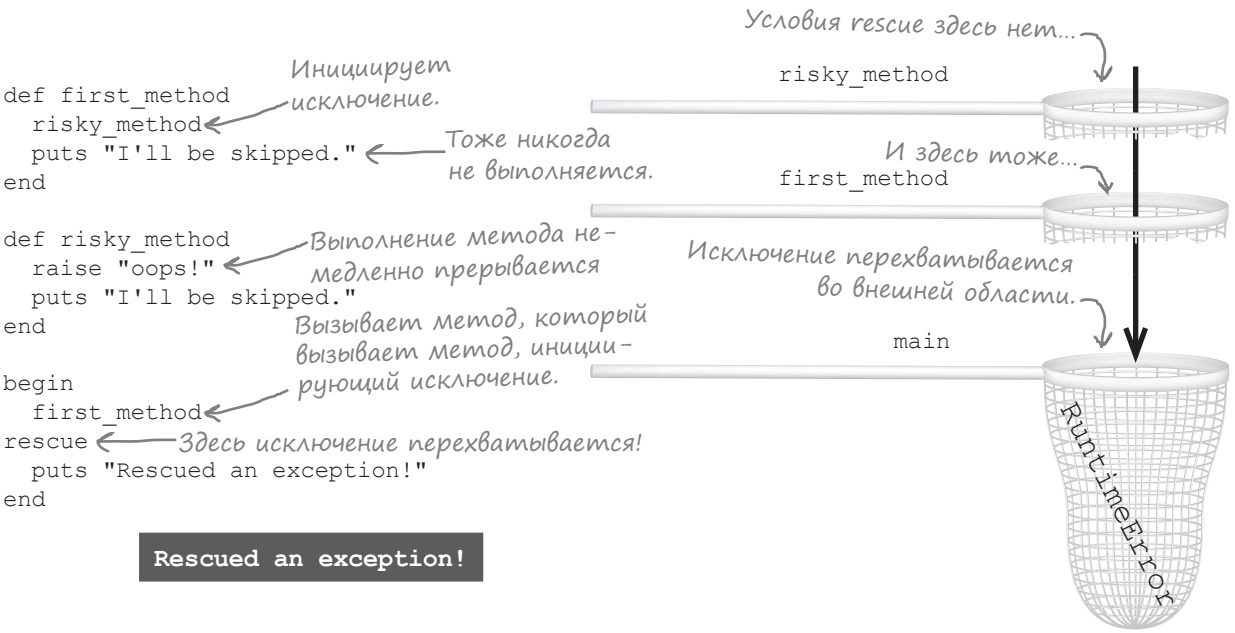
А здесь исключение перехватывается!

```
Rescued an exception!
```



Как Ruby ищет условие rescue (продолжение)

Исключение может передаваться по цепочке из нескольких методов. Если в области вызова метода не найдется подходящего условия `rescue`, то Ruby немедленно выходит *из этого* метода и ищет условие rescue в области *его* вызова. Это продолжается по цепочке до тех пор, пока не будет найдено подходящее условие `rescue`. (Если условие *не будет* найдено, происходит аварийное завершение работы программы.)



Ruby ищет условие `rescue` в методе, в котором произошло исключение. Если поиск оказывается безрезультатным, то Ruby продолжает поиск в месте, из которого был вызван метод, и так далее.

Использование условия rescue с классом SmallOven

В данный момент при вызове метода экземпляра `bake` нашего класса `SmallOven` без задания атрибута `contents` экземпляра выводится невразумительное сообщение об ошибке. Выполнение программы также немедленно прерывается, без обработки остальных элементов массива.

```
class SmallOven
  ...
  def bake
    unless @state == "on"
      raise "You need to turn the oven on first!"
    end
    if @contents == nil
      raise "There's nothing in the oven!"
    end
    "golden-brown #{contents}"
  end
end

dinner = ['turkey', nil, 'pie']
oven = SmallOven.new
oven.turn_on
dinner.each do |item|
  oven.contents = item
  puts "Serving #{oven.bake}."
end
```

Забыли поставить блюдо в духовку!

Все останавливается на элементе «nil» без обработки «pie»!

```
Turning oven on.
Serving golden-brown turkey.
oven.rb:19:in `bake': There's nothing in the oven! (RuntimeError)
        from oven.rb:31:in `block in <main>'
        from oven.rb:29:in `each'
        from oven.rb:29:in `<main>'
```

А теперь добавим условие `rescue`, чтобы выводилось более понятное сообщение об ошибке.

```
dinner = ['turkey', nil, 'pie']
oven = SmallOven.new
oven.turn_on
dinner.each do |item|
  begin
    oven.contents = item
    puts "Serving #{oven.bake}."
  rescue
    puts "Error: There's nothing in the oven!"
  end
end
```

Код остался неизменным, но теперь он заключен в блок begin/end

Перехватывает любые исключения, возникающие в двух предыдущих строках.

Выводим сообщение об ошибке.

```
Turning oven on.
Serving golden-brown turkey.
Error: There's nothing in the oven!
Serving golden-brown pie.
```

Первое блюдо готово.

Для блюда «nil» выводится сообщение об ошибке.

Переходим к приготовлению третьего блюда.

Так гораздо лучше! Все элементы в массиве обработаны, а для некорректного значения выводится понятное сообщение об ошибке — без всех рисков, связанных с возвращением строкового сообщения об ошибке из метода!

Описание проблемы от источника

Если в методе `bake` обнаруживается ошибка, строка с описанием проблемы передается методу `raise`:

```
class SmallOven
  ...
  def bake
    unless @state == "on"
      raise "You need to turn the oven on first!"
    end
    if @contents == nil
      raise "There's nothing in the oven!"
    end
    "golden-brown #{contents}"
  end

end
```

Если духовка выключена, это сообщение передается «raise»...

Если духовка пуста, то передается другое сообщение...

Пока в программе эти сообщения не используются; вместо этого в условии `rescue` всегда выводится одна строка, которая сообщает, что духовка пуста.

Но если духовка на самом деле *выключена*, а не пуста, то будет выведено неправильное сообщение об ошибке!

```
dinner = ['turkey', 'casserole', 'pie']
oven = SmallOven.new
oven.turn_off
dinner.each do |item|
  begin
    oven.contents = item
    puts "Serving #{oven.bake}."
  rescue
    puts "Error: There's nothing in the oven!"
  end
end
```

Если духовка выключена...

...программа игнорирует сообщения, переданные «raise», и использует одно сообщение во всех случаях!

Неправильно! Проблема в том, что духовка выключена!

```
Turning oven off.
Error: There's nothing in the oven!
Error: There's nothing in the oven!
Error: There's nothing in the oven!
```

Вместо этого нужно вывести сообщение, переданное `raise`...

Сообщения об исключениях

Если при вызове `raise` передается строка, то метод использует эту строку для назначения атрибута `message` созданного им исключения:

```
unless @state == "on"
  raise "You need to turn the oven on first!"
end
if @contents == nil
  raise "There's nothing in the oven!"
end
```

Создает объект исключения и задает его атрибут «message».

То же самое.

Нужно сделать совсем немного: удалить фиксированное сообщение об ошибке и вывести вместо него текущее содержимое атрибута `message` объекта исключения.

Чтобы сохранить исключение в переменной, нужно добавить в строку `rescue` символы =>, за которыми следует имя нужной переменной. (Обозначение => используется в некоторых литералах хешей, но в данном контексте оно не имеет никакого отношения к хешам.) Если объект исключения доступен, мы можем вывести его атрибут `message`.

```
begin
  raise "oops!"
rescue => my_exception
  puts my_exception.message
end
```

Создает исключение со строкой «oops!» в атрибуте message.

Исключение сохраняется в переменной.

Выводим сообщение из исключения.

```
oops!
```

А теперь обновим код так, чтобы исключение сохранялось в переменной, а его сообщение выводилось для пользователя:

```
dinner = ['turkey', 'casserole', 'pie']
oven = SmallOven.new
oven.turn_off
dinner.each do |item|
  begin
    oven.contents = item
    puts "Serving #{oven.bake}."
  rescue => error
    puts "Error: #{error.message}"
  end
end
```

Духовка выключена.

Исключение сохраняется в переменной.

Выводим сообщение, которое хранится в исключении.

Проблема описана в сообщении из исключения.

```
Turning oven off.
Error: You need to turn the oven on first!
Error: You need to turn the oven on first!
Error: You need to turn the oven on first!
```

Проблема решена. Теперь программа выводит то сообщение из исключения, которое было передано методу `raise`.

Развлечения с магнитами

В программе на языке Ruby, выложенной на холодильнике, часть магнитов перепуталась. Сможете ли вы расставить фрагменты кода по местам, чтобы программа выводила указанный результат?

```
def
end
end
begin
if destination == "Hawaii"
(destination)
drive
end
"You can't drive to Hawaii!"
puts error.message
=>
rescue
drive("Hawaii")
error
raise
```

Результат:

```
File Edit Window Help
You can't drive to Hawaii!
```

Развлечения с магнитами. Решение

В программе на языке Ruby, выложенной на холодильнике, часть магнитов перепуталась. Сможете ли вы расставить фрагменты кода по местам, чтобы программа выводила указанный результат?

```
def drive (destination)
    if destination == "Hawaii"
        raise "You can't drive to Hawaii!"
    end
end

begin
    drive("Hawaii")
rescue => error
    puts error.message
end
```

Результат:

```
File Edit Window Help
You can't drive to Hawaii!
```

Что было сделано...

С того момента, как мы занялись усовершенствованием обработки ошибок в коде моделирования духовки, мы уже многое сделали! Давайте еще раз припомним, что же было сделано за это время.

В метод `bake` класса `SmallOven` были добавлены команды `raise`, инициирующие исключение при обнаружении проблемы. Значение атрибута `message` объекта задавалось по-разному в зависимости от причины (пустая или выключенная духовка).

```
class SmallOven

  attr_accessor :contents

  def turn_on
    puts "Turning oven on."
    @state = "on"
  end
  def turn_off
    puts "Turning oven off."
    @state = "off"
  end

  def bake
    unless @state == "on"
      raise "You need to turn the oven on first!"
    end
    if @contents == nil
      raise "There's nothing in the oven!"
    end
    "golden-brown #{contents}"
  end

end
```

Исключение выдается в том случае, если пользователь пытается готовить при выключенной духовке.

Исключение выдается в том случае, если духовка пуста.

В коде, вызывающем метод `bake`, создается условие `rescue` с сохранением объекта исключения в переменной с именем `error`. Затем выводится содержимое атрибута `message` объекта исключения, которое сообщает, что же именно пошло не так.

```
dinner = ['turkey', 'casserole', 'pie']
oven = SmallOven.new
oven.turn_on
dinner.each do |item|
  begin
    oven.contents = item
    puts "Serving #{oven.bake}."
  rescue => error
    puts "Error: #{error.message}"
  end
end
```

Исключение сохраняется в переменной.

Выводим сообщение, содержащееся в исключении.

Что было сделано... (продолжение)

Если атрибут `contents` объекта `SmallOven` содержит `nil`, выводится одно сообщение об ошибке:

Атрибуту contents присваивается «nil»!

```
dinner = ['turkey', nil, 'pie']
oven = SmallOven.new
oven.turn_on
dinner.each do |item|
  begin
    oven.contents = item
    puts "Serving #{oven.bake}."
  rescue => error
    puts "Error: #{error.message}"
  end
end
```

```
Turning oven on.
Serving golden-brown turkey.
Error: There's nothing in the oven!
Serving golden-brown pie.
```

Сообщение из исключения.

...а если духовка выключена, выводится другое сообщение.

```
dinner = ['turkey', 'casserole', 'pie']
oven = SmallOven.new
oven.turn_off
dinner.each do |item|
  begin
    oven.contents = item
    puts "Serving #{oven.bake}."
  rescue => error
    puts "Error: #{error.message}"
  end
end
```

Духовка выключена.

Одно исключение инициируется трижды.

```
Turning oven off.
Error: You need to turn the oven on first!
Error: You need to turn the oven on first!
Error: You need to turn the oven on first!
```

Разная логика rescue для разных исключений

Как-то странно получается — если духовка выключена, мы трижды выдаем одно и то же исключение, по одному разу для каждого блюда. А может, просто... **включить духовку?**

Одно и то же исключение инициируется три раза.

```
Turning oven off.
Error: You need to turn the oven on first!
Error: You need to turn the oven on first!
Error: You need to turn the oven on first!
```

Было бы неплохо, если бы наша программа могла обнаружить проблему, включить духовку и снова попытаться приготовить обед.

Но мы не можем просто включить духовку и повторить попытку для *любого* полученного исключения. Если атрибут `contents` содержит `nil`, было бы глупо снова ставить *ничто* во включенную духовку!

```
class SmallOven
  ...
  def bake
    unless @state == "on"
      raise "You need to turn the oven on first!"
    end
    if @contents == nil
      raise "There's nothing in the oven!"
    end
    "golden-brown #{contents}"
  end

end
```

Включить духовку и попробовать снова? Вполне разумный план для обработки этого исключения...

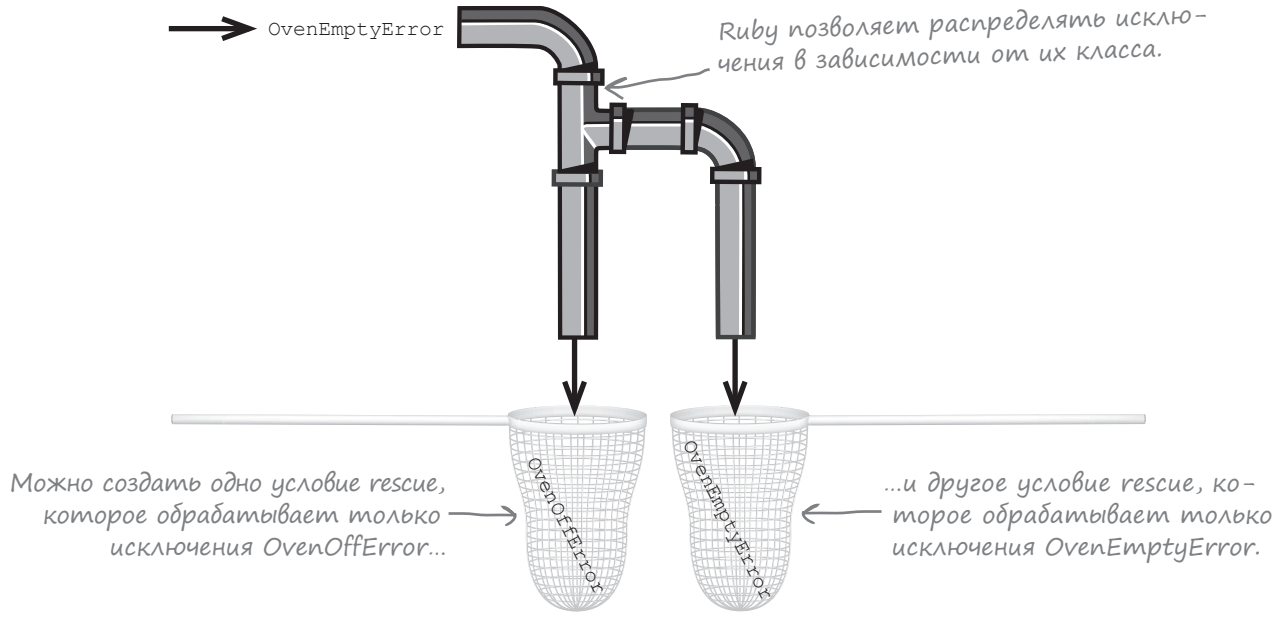
...но только не для этого!

Необходимо как-то различать исключения, которые может выдавать метод `bake`, чтобы по-разному обрабатывать их. А для *этого* можно использовать класс исключения...

Разная логика rescue для разных исключений (продолжение)

Ранее мы уже упоминали о том, что исключения представляют собой объекты. Но ведь каждый объект — экземпляр некоторого класса, верно? Вы можете указать, какие классы исключений обрабатываются каждым конкретным условием `rescue`. В этом случае условие `rescue` будет игнорировать любые исключения, не являющиеся экземпляром указанного класса (или одного из его субклассов).

Эта возможность позволяет направить исключение условию `rescue`, которое сможет обработать его так, как вам нужно.



Но если мы собираемся по-разному обрабатывать разные классы исключений, то для начала нужно разобраться в том, как *назначить* класс исключения.

Классы исключений

Когда мы вызываем метод `raise`, он создает объект исключения... и если это исключение не будет перехвачено, вы увидите класс объекта исключения при завершении программы.

Объект исключения является экземпляром класса RuntimeError.

```
raise "oops!"
```

```
myscript.rb:1:in `<main>': oops! (RuntimeError)
```

По умолчанию `raise` создает экземпляр класса `RuntimeError`. Впрочем, при желании можно выбрать другой класс, который будет использован методом `raise`. Просто передайте имя класса в первом аргументе, перед строкой, которая определяет сообщение данного исключения.

Укажите имя класса.

```
raise ArgumentError, "This method takes a String!"
```

```
myscript.rb:1:in `<main>':
This method takes a String! (ArgumentError)
```

Инициируется исключение заданного класса!

Укажите класс.

```
raise ZeroDivisionError, "Can't cut a pie into 0 portions!"
```

```
myscript.rb:1:in `<main>':
Can't cut a pie into 0 portions! (ZeroDivisionError)
```

Происходит исключение заданного класса!

Вы даже можете создавать и инициировать собственные классы исключений. Впрочем, при попытке использовать такой класс так, как вы это делали с другими классами, вы получите ошибку (совершенно не такую, которая вам нужна):

```
class MyError
end

raise MyError, "oops!"
```

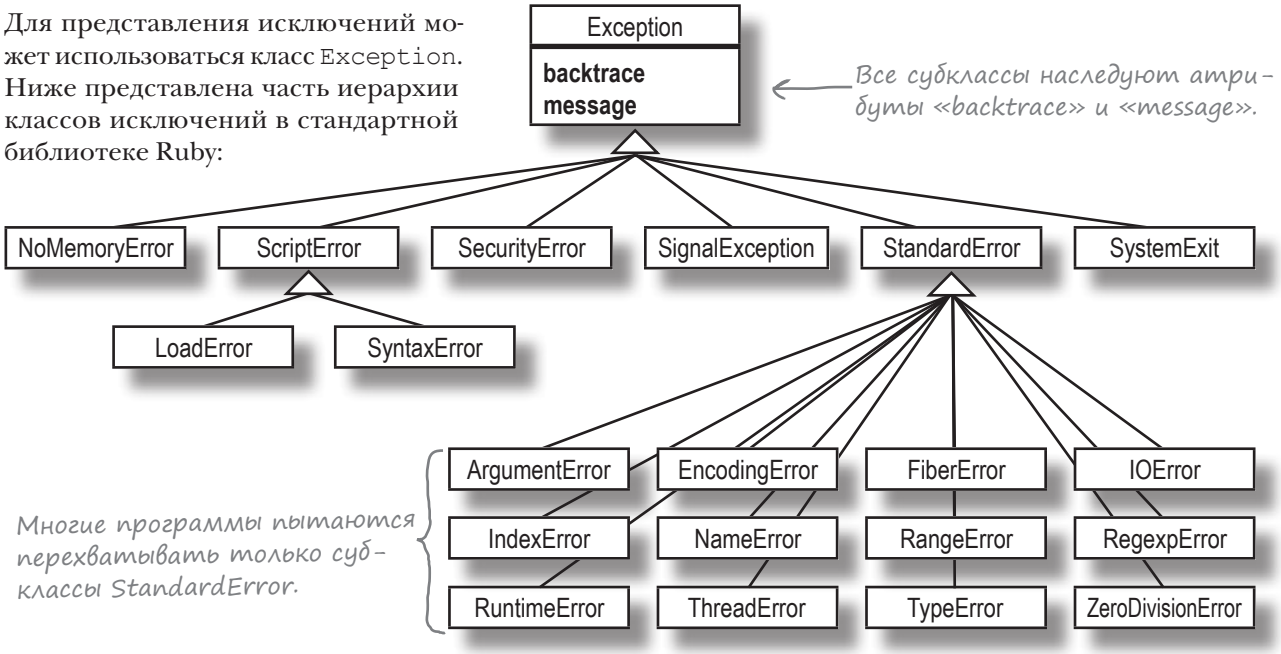
Не работает!

```
myscript.rb:4:in `raise': exception
class/object expected (TypeError)
```

Метод «raise» инициирует собственное исключение!

Классы исключений (продолжение)

Для представления исключений может использоваться класс `Exception`. Ниже представлена часть иерархии классов исключений в стандартной библиотеке Ruby:



Итак, если вы сделаете свой класс исключения субклассом `Exception`, он будет работать с `raise`...

```
class MyError < Exception
end

raise MyError, "oops!"
```

Должен быть субклассом Exception.

Наше исключение!

```
myscript.rb:4:in `<main>': oops! (MyError)
```

...но учтите, что в Ruby классы исключений обычно являются субклассами `StandardError`, а не прямыми субклассами `Exception`. По общепринятым соглашениям `StandardError` представляет категорию ошибок, которые могут быть обработаны типичной программой. Другие субклассы `Exception` представляют проблемы, неподконтрольные вашей программе, — например, нехватку памяти в системе или завершение ее работы.

Итак, хотя вы *можете* использовать `Exception` как суперкласс для своих исключений, обычно вместо него следует использовать `StandardError`.

```
class MyError < StandardError
end

raise MyError, "oops!"
```

Обычно стоит субклассировать StandardError, а не Exception.

```
myscript.rb:4:in `<main>': oops! (MyError)
```

Назначение класса исключения для условия rescue

Теперь, когда вы научились создавать собственные классы исключений, можно заняться перехватом *нужных* классов. Указывая имя класса за ключевым словом `rescue` в условии rescue, вы сообщаете, что условие должно перехватывать только исключения, являющиеся экземплярами этого класса (или одного из его субклассов).

В этом коде инициированное исключение (`PorridgeError`) не совпадает с типом, указанным в условии `rescue` (`BeddingError`), поэтому исключение не перехватывается:

```
class PorridgeError < StandardError
end
class BeddingError < StandardError
end

def eat
  raise PorridgeError, "too hot"
end
def sleep
  raise BeddingError, "too soft"
end

begin
  eat
rescue BeddingError => error
  puts "This bed is #{error.message}!"
end
```

Перехватывает только исключения BeddingError.

Инициирует исключение PorridgeError.

Исключение PorridgeError остается необработанным.

```
goldilocks.rb:7:in `eat': too hot (PorridgeError)
        from goldilocks.rb:14:in `<main>'
```

...но все исключения, которые *соответствуют* классу, указанному в условии `rescue`, *будут* обработаны:

```
begin
  sleep
rescue BeddingError => error
  puts "This bed is #{error.message}!"
end
```

Перехватывает исключения BeddingError.

Инициирует исключение BeddingError.

Подходящее исключение успешно перехватывается.

```
This bed is too soft!
```

Старайтесь всегда указывать тип исключения в условиях `rescue`. В этом случае ваша программа будет перехватывать только те исключения, которые она умеет обрабатывать.

Несколько условий rescue в одном блоке begin/end

Этот код перехватывает все экземпляры BeddingError, но игнорирует исключения PorridgeError. А мы хотим, чтобы перехватывались *оба* типа исключений...

```
class PorridgeError < StandardError
end
class BeddingError < StandardError
end

def eat
  raise PorridgeError, "too hot"
end
def sleep
  raise BeddingError, "too soft"
end

begin
  eat
rescue BeddingError => error
  puts "This bed is #{error.message}!"
end
```

Инициирует исключение PorridgeError.

Перехватывает только исключения BeddingError.

Исключение PorridgeError остается необработанным.

```
goldilocks.rb:7:in `eat': too hot (PorridgeError)
        from goldilocks.rb:14:in `<main>'
```

В один блок begin/end можно добавить несколько условий rescue, в каждом из которых указывается отдельный тип исключения.

```
begin
  eat
rescue BeddingError => error
  puts "This bed is #{error.message}!"
rescue PorridgeError => error
  puts "This porridge is #{error.message}!"
end
```

Сюда направляются исключения BeddingError.

Сюда направляются исключения PorridgeError.

```
This porridge is too hot!
```

Это позволяет выполнять разный код в зависимости от того, к какому типу относится перехваченное исключение.

```
begin
  sleep
rescue BeddingError => error
  puts "This bed is #{error.message}!"
rescue PorridgeError => error
  puts "This porridge is #{error.message}!"
end
```

На этот раз инициируется исключение BeddingError.

Оно обрабатывается здесь.

```
This bed is too soft!
```

Переход на пользовательские классы исключений в классе SmallOven

Итак, вы научились инициировать исключения созданных вами классов и обрабатывать исключения разных классов в разных местах. Теперь попробуем обновить наш код, моделирующий работу духовки. Если духовка выключена, нужно включить ее, а если пуста — предупредить об этом пользователя.

Мы создадим два новых класса исключений, представляющих две разновидности аномальных ситуаций, и сделаем их субклассами `StandardError`. Затем для каждого класса исключения будет добавлено отдельное условие `rescue`.

Определяем два новых класса исключений.

```
class OvenOffError < StandardError
end
class OvenEmptyError < StandardError
end
```

Инициируем исключение одного типа, если духовка выключена...

...и исключение другого типа, если духовка пуста.

```
class SmallOven
  ...
  def bake
    unless @state == "on"
      raise OvenOffError, "You need to turn the oven on first!"
    end
    if @contents == nil
      raise OvenEmptyError, "There's nothing in the oven!"
    end
    "golden-brown #{contents}"
  end
end

dinner = ['turkey', nil, 'pie']
oven = SmallOven.new
oven.turn_off
dinner.each do |item|
  begin
    oven.contents = item
    puts "Serving #{oven.bake}."
  rescue OvenEmptyError => error
    puts "Error: #{error.message}"
  rescue OvenOffError => error
    oven.turn_on
  end
end
```

Снова забыли поставить блюдо в духовку!

А духовка была выключена!

Перехватывает только исключения OvenEmptyError.

Выводит сообщение из исключения, как и в предыдущей версии кода.

Перехватывает только исключения OvenOffErrors.

Раз духовка выключена, мы включаем ее.

Условие rescue для OvenOffError включает духовку.

Условие rescue для OvenEmptyError выводит предупреждение.

```
Turning oven off.
Turning oven on.
Error: There's nothing in the oven!
Serving golden-brown pie.
```

Работает! Когда при выключенной духовке инициируется исключение `OvenOffError`, вызывается соответствующее условие `rescue`, а духовка включается. А когда по значению `nil` инициируется `OvenEmptyError`, условие `rescue` для *этого* исключения выводит предупреждение.

Перезапуск после исключения

Впрочем, мы кое-что упустили из вида... Наше условие `rescue` для `OvenOffError` снова включило духовку, и *остальные* блюда были приготовлены успешно. Но так как исключение `OvenOffError` произошло в то время, когда мы пытались приготовить блюдо, эта часть обеда была в итоге пропущена! Нужно вернуться и снова попытаться приготовить индейку (turkey) после того, как духовка была включена.

Условие rescue для OvenOffError включает духовку. Но тогда одно блюдо оказывается пропущенным!

```
Turning oven off.
Turning oven on.
There's nothing in the oven!
Serving golden-brown pie.
```

Ключевое слово `retry` делает именно то, что нужно. Когда вы включаете `retry` в условие `rescue`, выполнение возвращается к началу блока `begin/end`, и находящиеся там команды выполняются снова.

Например, если исключение возникло из-за попытки деления на 0, можно изменить делитель и попытаться выполнить операцию снова:

После исправления делителя управления возвращается к началу блока «begin».

```
amount_won = 100
portions = 0                    # Вызовет исключение ZeroDivisionError.
begin
  portion_size = amount_won / portions
  puts "You get $#{portion_size}."
rescue ZeroDivisionError
  puts "Revising portion count from 0 to 1."
  portions = 1                  # Исправляем проблему, породившую исключение.
  retry
end
```

```
Revising portion count from 0 to 1.
You get $100.
```

Будьте внимательны при использовании `retry`. Если вы не исправите проблему, породившую исключение (или если в коде `rescue` допущена ошибка), то исключение будет выдано снова, управление снова передастся retry... и программа войдет в бесконечный цикл! В таком случае нажмите клавиши Ctrl-C, чтобы прервать работу Ruby.

Если добавить в приведенный выше код `retry` без исправления проблемы с делителем, произойдет зацикливание:

Если пропустить выражение «portions = 1»...

```
amount_won = 100
portions = 0
begin
  portion_size = amount_won / portions
  puts "You get $#{portion_size}."
rescue ZeroDivisionError
  puts "Revising portion count from 0 to 1."
  retry                         # ...перезапуск произойдет, а проблема сохранится!
end
```

Бесконечный цикл!

```
...
Revising portion count from 0 to 1.
Revising portion count from 0 to 1.
Revising portion count from 0 to 1.
^Cwin.rb:4:in `new': Interrupt
```

Нажмите Ctrl-C, чтобы прервать работу программы.

Обновление кода с «retry»

Добавим `retry` в условие `rescue` после включения духовки и посмотрим, будет ли пропущенное блюдо обрабатываться на этот раз:

```
dinner = ['turkey', nil, 'pie']
oven = SmallOven.new
oven.turn_off
dinner.each do |item|
  begin
    oven.contents = item
    puts "Serving #{oven.bake}."
  rescue OvenEmptyError => error
    puts "Error: #{error.message}"
  rescue OvenOffError => error
    oven.turn_on
    retry
  end
end
```

Перезапускаем блок «begin» после включения духовки.

Условие rescue для OvenOffError включает духовку.

Блок «begin» перезапускается, а индейка успешно готовится.

```
Turning oven off.
Turning oven on.
Serving golden-brown turkey.
Error: There's nothing in the oven!
Serving golden-brown pie.
```

Получилось! Мы не только исправили ошибку, приводившую к исключению, но и смогли заново обработать проблемный объект (на этот раз успешно)!

Упражнение

Заполните пропуски в коде, чтобы он выдавал указанный результат.

```
class _____ < StandardError
end

score = 52
begin
  if score > 60
    puts "passing grade"
  else
    _____ TestScoreError, "failing grade"
  end
rescue _____ => error
  puts "Received #{error._____}. Taking make-up exam..."
  score = 63
  _____
end
```

Результат:

```
Received failing grade. Taking make-up exam...
passing grade
```

Упражнение
Решение

Заполните пропуски в коде, чтобы он выдавал указанный результат.

```
class TestScoreError < StandardError
end

score = 52
begin
  if score > 60
    puts "passing grade"
  else
    raise TestScoreError, "failing grade"
  end
rescue TestScoreError => error
  puts "Received #{error.message}. Taking make-up exam..."
  score = 63
  retry
end
```

Результат:

```
Received failing grade. Taking make-up exam...
passing grade
```

Код, который должен выполняться в любом случае

Но теперь стоит обратить внимание на один важный момент в предыдущих примерах: после завершения работы духовка никогда *не выключается*.

И эту проблему не решить простым добавлением одной строки кода. Духовка все равно останется включенной, потому что перед вызовом `turn_off` инициируется исключение:

```
begin
  oven.turn_on
  oven.contents = nil
  puts "Serving #{oven.bake}."
  oven.turn_off
rescue OvenEmptyError => error
  puts "Error: #{error.message}"
end
```

Инициирует исключение.

Никогда не выполняется!

```
Turning oven on.
Error: There's nothing in the oven!
```

Духовка не выключается!

В принципе, можно добавить вызов `turn_off` и в условие `rescue`...

```
begin
  oven.turn_on
  oven.contents = nil
  puts "Serving #{oven.bake}."
  oven.turn_off
rescue OvenEmptyError => error
  puts "Error: #{error.message}"
  oven.turn_off
end
```

Код копируется отсюда...

...в условие rescue.

```
Turning oven on.
Error: There's nothing in the oven!
Turning oven off.
```

Духовка выключается, несмотря на исключение.

...но такое дублирование кода тоже нежелательно.

Условие ensure

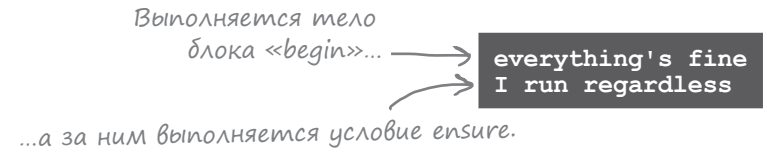
Если у вас имеется код, который должен быть выполнен *независимо* от того, происходило исключение или нет, его можно разместить в условии `ensure`. Условие `ensure` должно находиться в блоке `begin/end` после всех условий `rescue`. Все команды, заключенные между ключевыми словами `ensure` и `end`, гарантированно будут выполнены до выхода из блока.

Условие `ensure` будет выполнено, если в программе происходит исключение:

```
begin
  raise "oops!"
rescue
  puts "rescued an exception"
ensure
  puts "I run regardless"
end
```

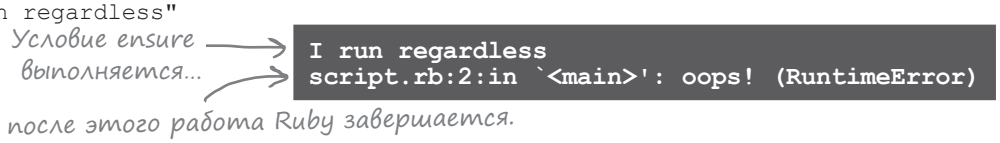
Условие `ensure` выполняется и в том случае, если исключение *не* происходит:

```
begin
  puts "everything's fine"
rescue
  puts "rescued an exception"
ensure
  puts "I run regardless"
end
```



Даже если исключение не перехватывается, условие `ensure` все равно будет выполнено до прерывания работы Ruby!

```
begin
  raise "oops!"
ensure
  puts "I run regardless"
end
```



Ситуации, в которых требуется выполнить некий завершающий код независимо от того, успешно завершилась операция *или нет*, достаточно часто встречаются в программировании. Например, файлы следует закрывать даже в том случае, если их содержимое повреждено. Сетевые подключения должны закрываться даже в том случае, если вы не получили данные. А духовка должна выключаться даже в том случае, если обед подгорел. Условие `ensure` идеально подходит для размещения такого кода.

Гарантированное выключение духовки

Попробуем переместить вызов `oven_off` в условие `ensure` и посмотрим, что из этого получится...

```
begin
  oven.turn_on
  oven.contents = 'turkey'
  puts "Serving #{oven.bake}."
rescue OvenEmptyError => error
  puts "Error: #{error.message}"
ensure
  oven.turn_off
end
```

Достаточно оставить эту строку в условии ensure.

Блок «begin» завершается...

```
Turning oven on.
Serving golden-brown turkey.
Turning oven off.
```

...за ним следует условие ensure.

Работает! Условие `ensure` будет выполнено сразу же после завершения тела `begin/end`, и духовка выключается.

Даже если происходит исключение, духовка все равно будет выключена. Сначала выполняется условие `rescue`, а за ним выполняется условие `ensure`, в котором вызывается `turn_off`.

```
begin
  oven.turn_on
  oven.contents = nil
  puts "Serving #{oven.bake}."
rescue OvenEmptyError => error
  puts "Error: #{error.message}"
ensure
  oven.turn_off
end
```

Происходит исключение.

Достаточно оставить эту строку в условии ensure.

Выполняется условие rescue...

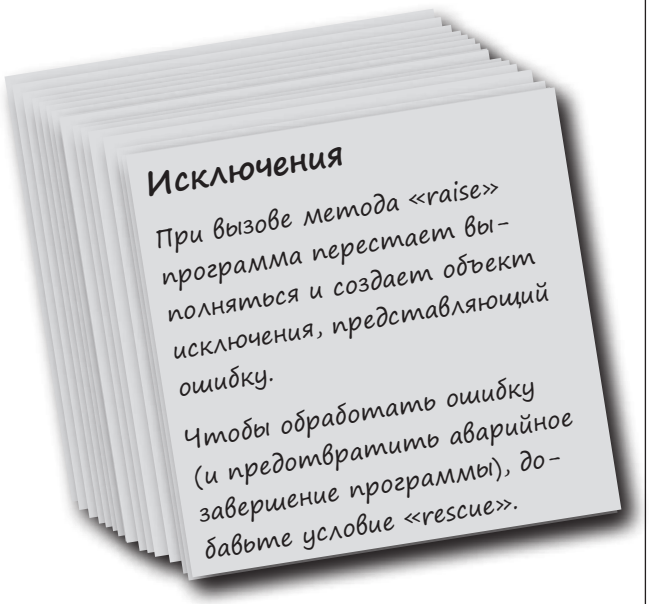
```
Turning oven on.
Error: There's nothing in the oven!
Turning oven off.
```

...а за ним следует условие ensure.

В реальном мире не все идет по плану, поэтому-то и существуют исключения. Когда-то возникающие исключения приводили к аварийному завершению вашей программы. Но теперь, когда вы научились обрабатывать исключения, вы убедитесь, что исключения — мощный механизм, обеспечивающий безотказное выполнение вашего кода!

Ваш инструментарий Ruby

Глава 12 осталась позади, а ваш инструментарий пополнился обработкой исключений.



- Если передать методу `raise` имя субкласса `Exception`, то метод создаст исключение этого класса.
- У экземпляров класса `Exception` и его субклассов имеется атрибут `message`, который может использоваться для передачи дополнительной информации о проблеме.
- Если передать строку во втором аргументе метода `raise`, то она будет присвоена атрибуту `message` объекта исключения.
- В блок `begin`/`end` можно добавить условие `rescue`. Если в коде, следующем за `begin`, происходит исключение, то выполняется код в условии `rescue`.
- После выполнения кода в условии `rescue` выполнение продолжается в обычном порядке после блока `begin`/`end`.
- Если исключение происходит внутри метода, и в этом методе нет условия `rescue`, Ruby выходит из метода и ищет условие `rescue` в том месте, в котором метод был вызван.
- Вы можете указать, какие классы исключений должно обрабатывать конкретное условие `rescue`. Исключения, не являющиеся экземплярами заданного класса (или его субклассов), перехватываться этим условием не будут.
- В один блок `begin`/`end` можно добавить несколько условий `rescue` для разных типов исключений.
- Вы можете определять собственные классы исключений. Большинство исключений, обрабатываемых программами Ruby, являются субклассами `StandardError`, а этот класс является субклассом `Exception`.
- Чтобы снова выполнить код в окружающем блоке `begin`/`end`, добавьте ключевое слово `retry` без условия `rescue`.
- В конец блока `begin`/`end` можно добавить условие `ensure`. Оно будет выполнено перед выходом из блока `begin`/`end` независимо от того, происходило исключение или нет.

Далее в программе…

Программисты тоже ошибаются; именно поэтому так важно тщательно тестировать программы. Впрочем, ручное тестирование занимает слишком много времени, и, откровенно говоря, это очень скучное занятие. В следующей главе будет представлен более эффективный способ организации тестирования: *автоматизированные тесты*.

Контроль качества кода

А вы уверены, что ваша программа работает правильно? Действительно уверены? Прежде чем передавать новую версию пользователям, вы, вероятно, опробовали новые функции и убедились в том, что они работают. Но проверили ли вы *старые* функции, чтобы убедиться, что они не перестали работать? *Все* старые функции? Если этот вопрос не дает вам покоя, значит, вашей программе необходимо автоматизированное тестирование. Автоматизированные тесты гарантируют, что основные компоненты программы продолжают правильно работать даже после изменения кода. **Модульные тесты** — самая распространенная, самая важная разновидность автоматизированных тестов. В поставку Ruby входит библиотека **MiniTest**, предназначенная для модульного тестирования. В этой главе вы узнаете все, что действительно необходимо знать об этой библиотеке!

Автоматизированные тесты помогают найти ошибки до того, как их найдут другие

Разработчик А сталкивается с разработчиком Б в ресторанчике, в котором они оба часто бывают...

Разработчик А:

Как твоя новая работа?

Ничего себе. И как *такое* могло случиться на сервере счетов?

Целых два месяца... И ваши тесты не обнаружили ошибку?

Да, автоматизированные тесты. Они продолжали нормально проходить после появления ошибки?

Что?!

Разработчик Б:

Бывает и получше. Мне пришлось возвращаться в офис после ужина. Обнаружилась ошибка, из-за которой некоторым клиентам счета выставлялись вдвое чаще положенного.

Мы думаем, что ошибка появилась пару месяцев назад. Один из наших разработчиков тогда вносил изменения в код выставления счетов..

Тесты?

Да у нас их и не было.

Ваши клиенты зависят от качества вашего кода. Ошибки могут привести к катастрофическим последствиям. Страдает репутация вашей компании, а *вам* приходится работать по ночам, чтобы исправить ошибку.

Вот почему были изобретены автоматизированные тесты. **Автоматизированный тест** представляет собой отдельную программу, которая выполняет компоненты *основной программы* и *проверяет*, что они работают именно так, как ожидается.

Нет, если только при этом вы не тестируете все старые функции — вдруг внесенные изменения что-то нарушили? Автоматизированные тесты экономят время по сравнению с ручными тестами, и вдобавок они обычно работают более тщательно.

Программа, для которой необходимы автоматизированные тесты

Рассмотрим пример ошибки, которая может быть обнаружена автоматизированными тестами. Ниже приведен простой класс, который объединяет массив строк в список по правилам английского языка. Если список состоит из двух элементов, они соединяются словом *and* (например, «apple and orange»). При большем количестве элементов добавляются запятые (например, «apple, orange and pear»).

```
class ListWithCommas
  attr_accessor :items
  def join
    last_item = "and #{items.last}"
    other_items = items.slice(0, items.length - 1).join(', ')
    "#{other_items} #{last_item}"
  end
end
```

Присваивается массив объединяемых элементов.

Добавляем слово «and» перед последним элементом.

Берем элементы с первого до предпоследнего и соединяем их с добавлением запятых.

Возвращаем результат в виде одной строки.

Внутри метода `join` мы берем последний элемент в массиве и добавляем к нему слово *and*. Затем метод экземпляра `slice` массива используется для получения всех элементов, *кроме* последнего.

Пожалуй, стоит сказать пару слов о методе `slice`. Он выделяет из массива «сегмент», начинающийся с заданного индекса и состоящий из заданного количества элементов. Выбранные элементы возвращаются в виде нового массива.

Начиная со второго элемента... *...извлекаются три элемента.*

```
p ['a', 'b', 'c', 'd', 'e'].slice(1, 3)
```

```
["b", "c", "d"]
```

Наша цель — выбрать все элементы, кроме последнего. Соответственно, сегмент должен начинаться с индекса 0, а для получения длины сегмента длина массива уменьшается на 1.

```
array = ['a', 'b', 'c', 'd', 'e']
p array.slice(0, array.length - 1)
```

```
["a", "b", "c", "d"]
```

Выбираем элементы, начиная с первого... *...и до предпоследнего.*

Проверим, как работает этот класс, на примере пары списков:

```
two_subjects = ListWithCommas.new
two_subjects.items = ['my parents', 'a rodeo clown']
puts "A photo of #{two_subjects.join}"
three_subjects = ListWithCommas.new
three_subjects.items = ['my parents', 'a rodeo clown', 'a prize bull']
puts "A photo of #{three_subjects.join}"
```

Похоже, метод `join` работает! Вернее, работает, пока мы не внесем в него изменение...

```
A photo of my parents and a rodeo clown
A photo of my parents, a rodeo clown and a prize bull
```

Программа, для которой необходимы автоматизированные тесты (продолжение)

Впрочем, в выходных данных программы обнаруживается одна небольшая проблема...

```
A photo of my parents, a rodeo clown and a prize bull
```

Пожалуй, такой список выглядит несколько двусмысленно. И такое форматирование списков может привести к другим недоразумениям.

Чтобы избежать путаницы, мы изменим наш код и добавим дополнительную запятую перед *and* (как в строке «apple, orange, and pear»). Опробуем новую версию метода `join` на трех элементах.

```
class ListWithCommas
  attr_accessor :items
  def join
    last_item = "and #{items.last}"
    other_items = items.slice(0, items.length - 1).join(', ')
    "#{other_items}, #{last_item}"
  end
end

three_subjects = ListWithCommas.new
three_subjects.items = ['my parents', 'a rodeo clown', 'a prize bull']
puts "A photo of #{three_subjects.join}"
```

Перед последним элементом добавляется запятая.

Вот добавленная запятая.

```
A photo of my parents, a rodeo clown, and a prize bull
```

Превосходно! Теперь никаких недоразумений, всем все ясно.

Постойте! Вы протестировали код для списка из **трех** элементов, но не попытались снова проверить его для **двух** элементов. И в программе появилась ошибка!

Что, серьезно? Снова проверяем для списка из двух элементов...

```
two_subjects = ListWithCommas.new
two_subjects.items = ['my parents', 'a rodeo clown']
puts "A photo of #{two_subjects.join}"
```

Запятой здесь быть не должно!

```
A photo of my parents, and a rodeo clown
```

Прежде метод `join` возвращал строку `"my parents and a rodeo clown"` для списка из двух элементов, но лишняя запятая появилась и здесь! Мы были так поглощены исправлением недочетов для списка из *трех* элементов, что забыли проверить другие сценарии.

Программа, для которой необходимы автоматизированные тесты (продолжение)

Если бы у нас были автоматизированные тесты для этого класса, то этой проблемы удалось бы избежать.

Автоматизированный тест выполняет ваш код для конкретного набора входных данных и проверяет, был ли получен конкретный результат. Если результат совпадает с ожидаемым значением, тест считается «пройденным».

Но предположим, что в коде была случайно допущена ошибка (как в случае с лишней запятой). Результат работы кода не будет совпадать с ожидаемым значением, и тест не проходит. Вы немедленно узнаете о существовании ошибки.

Тест проходит.

☑ Для элементов `['apple', 'orange', 'pear']` метод `join` должен вернуть `"apple, orange, and pear"`.

Тест не проходит!

☒ Для элементов `['apple', 'orange']` метод `join` должен вернуть `"apple and orange"`.

Наличие автоматизированных тестов фактически означает, что ваш код анализируется на предмет наличия ошибок — полностью, автоматически — при внесении каждого изменения!

Типы автоматизированных тестов

Существует много разновидностей автоматизированных тестов, применяемых на практике. Ниже перечислены наиболее популярные типы:

- *Хронометражные тесты (тесты производительности)* измеряют скорость выполнения программы.
- *Интеграционные тесты* выполняют всю программу и убеждаются в том, что все методы, классы и другие компоненты правильно работают в сочетании друг с другом.
- *Модульные тесты* выполняют отдельные компоненты (модули) вашей программы, обычно на уровне отдельных методов.

При желании вы сможете найти и загрузить библиотеки, предназначенные для создания таких видов тестов. Но поскольку в поставку Ruby включена библиотека, предназначенная конкретно для модульного тестирования, в этой главе основное внимание уделяется именно этому виду тестов.

MiniTest: стандартная библиотека модульного тестирования Ruby

В стандартную библиотеку Ruby входит инфраструктура модульного тестирования, которая называется **MiniTest**. (Раньше в Ruby включалась другая библиотека с именем **Test::Unit**. Новая библиотека называется MiniTest, потому что она делает многое из того, что делала библиотека Test::Unit, с меньшим объемом кода.

Начнем с написания простейшего теста. Ничего полезного этот тест не делает; сначала мы хотим просто показать, как работает MiniTest. Затем можно будет заняться тестированием реального кода Ruby.

Прежде всего следует выполнить команду require для `'minitest/autorun'` из стандартной библиотеки; команда загружает библиотеку MiniTest и настраивает ее для автоматического запуска загружаемых тестов.

Теперь можно перейти к созданию теста. Создайте новый субкласс `Minitest::Test` и присвойте ему любое имя. Субклассы `Minitest::Test` могут выполняться как модульные тесты.

При запуске теста MiniTest перебирает класс теста, находит все методы экземпляра, имена которых начинаются с `test_`, и вызывает их. (Вы можете добавить методы с другими именами, но они не будут рассматриваться как тестовые.) Мы добавим в наш класс два тестовых метода.

В этих двух тестовых методах выполняются два вызова метода `assert`. Метод `assert` — один из многих методов, унаследованных от `Minitest::Test`, — проверяет, соответствует ли поведение вашего кода ожиданиям. Принцип его работы очень прост: если ему передается истинное значение, то тест проходит успешно, а если ложное — то весь тест прерывается и немедленно останавливается. Попробуем передать `true` в одном тесте, и `false` в другом.

```
require 'minitest/autorun'          <-- Загружаем MiniTest.

class TestSomething < Minitest::Test   <-- Создаем субкласс Minitest::Test.

  def test_true_assertion           <-- Первый тестовый метод.
    assert(true)
  end                               <-- Этот тест пройдет.

  def test_false_assertion          <-- Второй тестовый метод.
    assert(false)
  end                               <-- Этот тест не пройдет.

end
```

test_something.rb

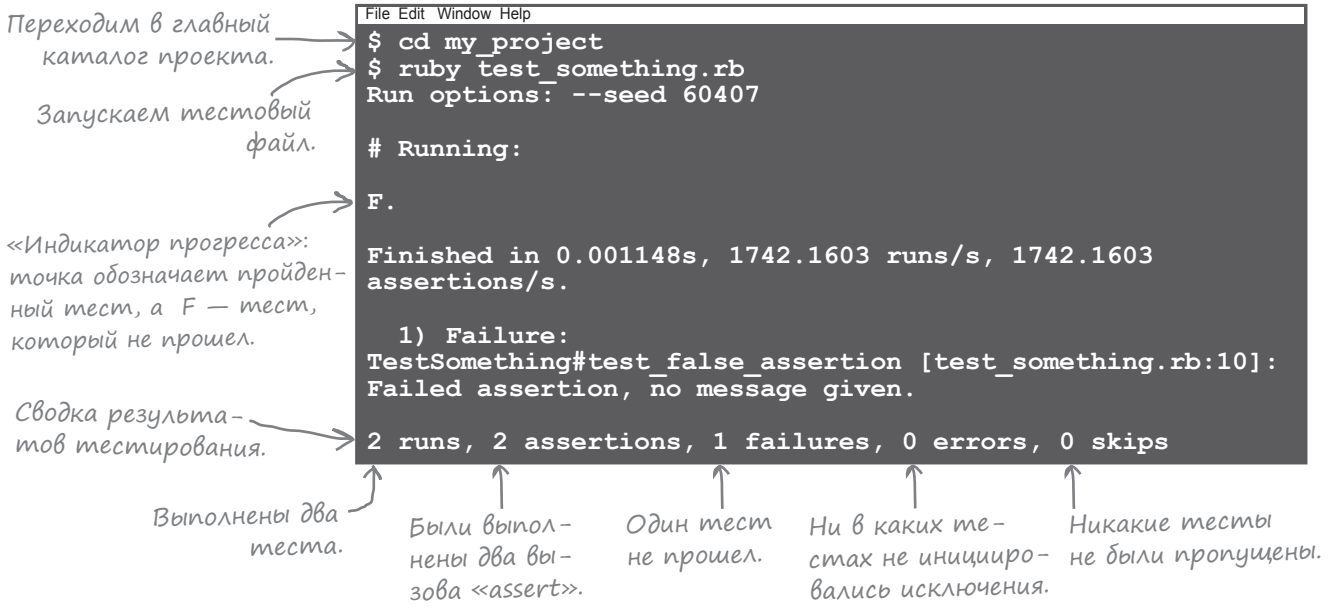
Сохраните этот код в файле с именем *test_something.rb*. Честно говоря, ничего выдающегося там нет. Но попробуем выполнить его и посмотрим, что получится!

Выполнение теста

В терминальном окне перейдите в каталог, в котором был сохранен файл *test_something.rb*. Запустите его командой:

```
ruby test_something.rb
```

Тесты выполняются автоматически, и вы получаете сводку результатов.



Со временем количество тестов увеличивается, а их выполнение занимает время, поэтому MiniTest отображает «индикатор прогресса», выводя один символ при выполнении каждого теста. Если текущий тест пройден, выводится точка, а если не пройден — выводится символ `F`.

После завершения тестов выводится отчет с описаниями всех сбойных тестов. В нем приводится имя и номер строки тестового метода, а также причина отказа. Все эти ошибки более подробно рассматриваются на нескольких ближайших страницах.

Самая важная часть выходных данных — сводка тестирования. В сводке указано количество выполненных тестовых методов, количество вызовов `assert` и аналогичных методов, количество непройденных тестов и количество неперехваченных исключений (ошибок). Если количество ошибок и количество непройденных тестов равно 0, значит, ваш код работает правильно. (Конечно, если вы не допустили ошибки в самих тестах.)

Так как в одном из тестов `assert` было передано ложное значение, в сводку включается сообщение о непройденном тесте. Если бы это был настоящий тест, то эта информация означала бы, что в коде необходимо что-то исправить.

Тестирование класса

Итак, вы примерно представляете, как написать и запустить тест в MiniTest. Впрочем, сам по себе тест особой пользы не приносит. Теперь, когда вы понимаете механику MiniTest, мы напишем модульный тест для реального класса Ruby.

По общепринятым соглашениям (и чтобы не разводить беспорядок) код модульных тестов следует хранить в специальных файлах отдельно от основного кода программы. А это означает, что подготовка и запуск тестов потребуют некоторых дополнительных шагов...

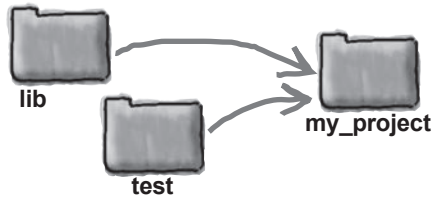
1. Сохраните приведенный ниже простой класс в файле с именем *person.rb*.

```
class Person
  attr_accessor :name
  def introduction
    "Hello, my name is #{name}!"
  end
end
```

2. Создайте и сохраните тестовый класс в отдельном файле с именем *test_person.rb*. (Вскоре содержимое этого файла будет рассмотрено более подробно.)

```
require 'minitest/autorun'
require 'person'

class TestPerson < Minitest::Test
  def test_introduction
    person = Person.new
    person.name = 'Bob'
    assert(person.introduction == 'Hello, my name is Bob!')
  end
end
```

3. Создайте в основном каталоге проекта два подкаталога. По общепринятым соглашениям одному подкаталогу присваивается имя *lib*, а другому — имя *test*.



Тестирование класса (продолжение)

4 Переместите файл с тем классом, который требуется протестировать, в подкаталог *lib*. Переместите тестовый файл в подкаталог *test*.

5 В терминальном окне перейдите в основной каталог проекта (в котором находятся каталоги *lib* и *test*). Введите следующую команду:

```
ruby -I lib test/test_person.rb
```

Флаг `-I lib` добавляет *lib* в список каталогов, в которых Ruby проводит поиск при вызове `require`; это необходимо для загрузки файла *person.rb*. Запись *test/test_person.rb* означает, что Ruby будет искать файл с именем *test_person.rb* в подкаталоге *test*.

Ваши модульные тесты выполняются и выдают результат вроде приведенного ниже.

(О том, как автоматизировать эту часть, рассказано в приложении!)

Переходим в основной каталог проекта.

Запускаем тестовый файл в Ruby.

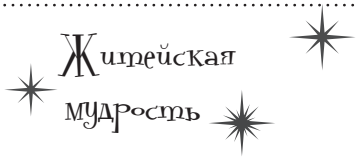
Точка означает, что тест прошел успешно.

Сводка всех результатов тестов.

```
File Edit Window Help
$ cd my_project
$ ruby -I lib test/test_person.rb
Run options: --seed 48563

# Running:

.

Finished in 0.000999s, 1001.0010 runs/s,
1001.0010 assertions/s.

1 runs, 1 assertions, 0 failures, 0 errors,
0 skips
```

Из сводки в нижней части видно, что количество ошибок и непрошедших тестов равно 0. Тестирование прошло успешно!

Тестовый код более подробно рассматривается на следующей странице...



Создайте каталог с именем *lib* для хранения файлов, содержащих классы и модули. Создайте отдельный каталог с именем *test* для хранения файлов модульных тестов.

Подробнее о тестовом коде

А теперь повнимательнее присмотримся к коду модульного теста. В файле *person.rb* из каталога *lib* находится простой класс, который необходимо протестировать. Этот класс состоит из одного атрибута и одного метода экземпляра.

```
class Person
  attr_accessor :name
  def introduction
    "Hello, my name is #{name}!"
  end
end
```

lib/person.rb

А в файле *test_person.rb* из каталога *test* хранится код теста.

```
require 'minitest/autorun'   <-- Загружаем MiniTest.
require 'person'             <-- Загружаем тестируемый класс.

class TestPerson < Minitest::Test   <-- Определяем тестовый класс.
  def test_introduction             <-- Определяем тестовый метод.
    person = Person.new
    person.name = 'Bob'
    assert(person.introduction == 'Hello, my name is Bob!')
  end
end
```

test/test_person.rb

После загрузки MiniTest подключается файл `'person'` для загрузки класса `Person`. Вызов `require` работает потому, что при выполнении теста в командную строку был добавлен флаг `-I lib`. Он добавляет каталог *lib* в список каталогов, в которых Ruby ищет загружаемые файлы.

После того как все необходимые классы будут загружены, можно переходить к определению теста. Мы создаем новый субкласс `Minitest::Test` с именем `TestPerson` и добавляем в него один тестовый метод (не забудьте, что имя метода должно начинаться с `test_`).

И в этом тестовом методе мы наконец-то проверяем, правильно ли работает наш код...

Часто Задаваемые Вопросы

В: В программе нигде не вызывается метод `TestPerson.new` или какой-либо из тестовых методов. Как тесты выполняются сами по себе?

О: При выполнении `require 'minitest/autorun'` тестовые классы автоматически запускаются сразу же после того, как они будут загружены.

Подробнее о тестовом коде (продолжение)

Тестирование кода в тестовом методе очень похоже на его вызов из обычной программы.

Если бы мы создали класс `Person`, задали его атрибуту `name` значение `'Bob'`, а затем провели проверку равенства, чтобы узнать, равно ли возвращаемое значение его метода `introduction` строке `'Hello, my name is Bob!'`, то результат этого сравнения, естественно, был бы истинным:

```
person = Person.new
person.name = 'Bob'
puts person.introduction == 'Hello, my name is Bob!'
```

true

Собственно, все это и происходит в методе `test_introduction`. Мы создаем обычный экземпляр `Person`, как если бы класс использовался в настоящем приложении. Его атрибуту `name` присваивается значение, как в приложении. И затем вызывается его метод `introduction`, как в обычной программе.

Единственное различие проявляется в том, что затем возвращаемое значение сравнивается с возвращаемой строкой. Если они равны, то `assert` передается значение `true`, и тест проходит успешно. Если же они *не* равны, то `assert` будет передано значение `false`, а тест не пройдет. (А мы узнаем о том, что в нашем коде необходимо что-то исправить!)

Создание экземпляра Person с именем «Bob».

Для экземпляра Person с именем «Bob» должна возвращаться эта строка.

```
def test_introduction
  person = Person.new
  person.name = 'Bob'
  assert(person.introduction == 'Hello, my name is Bob!')
end
```

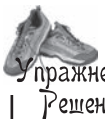
Упражнение

Заполните пропуски в коде, чтобы он выдавал указанный результат.

```
require '_________/autorun'

class TestMath < _________________
  def ____test_truth
    _______(2 + 2 == 4)
  end
  def ____test_fallacy
    _______(2 + 2 == 5)
  end
end
```

Результат:

```
File Edit Window Help
$ ruby test_math.rb
Run options: --seed 55914

# Running:

.F

Finished in 0.000863s, 2317.4971 runs/s,
2317.4971 assertions/s.

  1) Failure:
TestMath#test_fallacy [test_math.rb:8]:
Failed assertion, no message given.

2 runs, 2 assertions, 1 failures,
0 errors, 0 skips
```



Заполните пропуски в коде, чтобы он выдавал указанный результат.

```
require 'minitest/autorun'

class TestMath < Minitest::Test
  def test_truth
    assert(2 + 2 == 4)
  end
  def test_fallacy
    assert(2 + 2 == 5)
  end
end
```

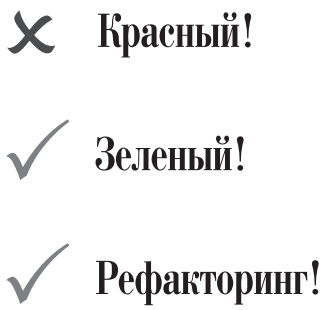
Результат:

```
File Edit Window Help
$ ruby test_math.rb
Run options: --seed 55914

# Running:

.F

Finished in 0.000863s, 2317.4971 runs/s,
2317.4971 assertions/s.

  1) Failure:
TestMath#test_fallacy [test_math.rb:8]:
Failed assertion, no message given.

2 runs, 2 assertions, 1 failures,
0 errors, 0 skips
```

Красный, зеленый, рефакторинг

Когда у вас появится опыт использования модульного тестирования, вероятно, вы начнете использовать рабочий цикл, который часто описывают фразой «красный, зеленый, рефакторинг»:

- **Красная фаза:** Вы пишете тест для *нужной* функции, хотя на данный момент эта функция еще не существует. Вы проводите тест и убеждаетесь в том, что он *не проходит*.
- **Зеленая фаза:** Вы реализуете функцию в своем основном коде. Не беспокойтесь, если написанный вами код окажется неэлегантным или неэффективным; ваша единственная цель — заставить его работать. После этого вы проводите тест и убеждаетесь в том, что он *проходит*.
- **Фаза рефакторинга:** Теперь можно заняться *рефакторингом* (переработкой) кода, изменяя и совершенствуя его на ваше усмотрение. Вы видели, что исходный тест *не проходил* — значит, он не пройдет, если в коде вашего приложения что-то сломается. Вы видели, что тест *проходил* — значит, он будет проходить, если ваш код работает правильно.



С модульными тестами вы можете *изменять* код, не беспокоясь о возможных последствиях, — именно поэтому они так важны. Если вы видите, что ваш код можно сократить или упростить, не упускайте такую возможность. После завершения работы вы просто проведете тесты заново и будете уверены в том, что программа работает.

Тесты для класса ListWithCommas

Теперь вы знаете, как написать и выполнить модульные тесты с использованием MiniTest, и мы можем заняться поиском проблем в классе `ListWithCommas`.

Класс `ListWithCommas` работает нормально, если передать ему список из *трех* элементов:

```
three_subjects = ListWithCommas.new
three_subjects.items = ['my parents', 'a rodeo clown', 'a prize bull']
puts "A photo of #{three_subjects.join}"
```

```
A photo of my parents, a rodeo clown, and a prize bull
```

Но если передать ему список всего из *двух* элементов, появляется лишняя запятая.

```
two_subjects = ListWithCommas.new
two_subjects.items = ['my parents', 'a rodeo clown']
puts "A photo of #{two_subjects.join}"
```

Запятой здесь быть не должно!

```
A photo of my parents, and a rodeo clown
```

А теперь напишем тесты, которые показывают, что мы *ожидаем* получить от метода `join`. Выполним их и убедимся в том, что в данный момент они не проходят. Затем класс `ListWithCommas` будет изменен так, чтобы тесты проходили. И когда тесты будут проходить, мы будем знать, что ошибка в коде исправлена!

Мы напишем два теста: для соединения двух и трех слов. В каждом случае мы будем создавать экземпляр `ListWithCommas` и присваивать массив его атрибуту `items`, как и в реальной программе. Затем мы вызываем метод `join` и проверяем, что возвращаемое значение совпадает с ожидаемым.

```
require 'minitest/autorun'          ← Загружаем MiniTest.
require 'list_with_commas'          ← Загружаем тестируемый класс.

class TestListWithCommas < Minitest::Test

  def test_it_joins_two_words_with_and          ← Первый тестовый метод.
    list = ListWithCommas.new
    list.items = ['apple', 'orange']            ← Тестируем «join» с двумя элементами.
    assert('apple and orange' == list.join)     ← Тест проходит, ЕСЛИ «join» возвращает ожидаемую строку.
  end

  def test_it_joins_three_words_with_commas     ← Второй тестовый метод.
    list = ListWithCommas.new
    list.items = ['apple', 'orange', 'pear']    ← Тестируем «join» с тремя элементами.
    assert('apple, orange, and pear' == list.join)
  end                                           ← Тест проходит, ЕСЛИ «join» возвращает ожидаемую строку.

end
```

Тесты для класса ListWithCommas (продолжение)

Тестовый класс готов. Пора подготовить его к выполнению!

1. Сохраните класс `ListWithCommas` в файле с именем *list_with_commas.rb*.

```
class ListWithCommas
  attr_accessor :items
  def join
    last_item = "and #{items.last}"
    other_items = items.slice(0, items.length - 1).join(', ')
    "#{other_items}, #{last_item}"
  end
end
```

list_with_commas.rb

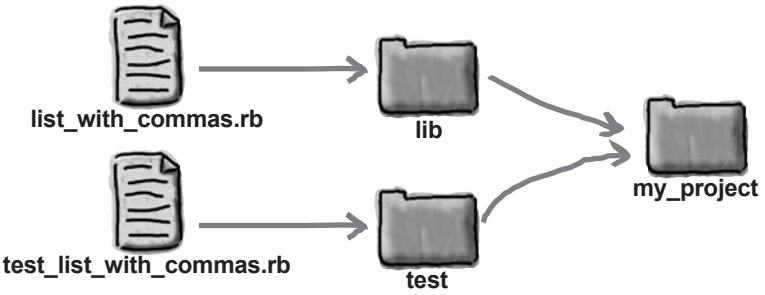
2. Сохраните класс `TestListWithCommas` в отдельном файле с именем *test_list_with_commas.rb*.

```
require 'minitest/autorun'
require 'list_with_commas'

class TestListWithCommas < Minitest::Test

  def test_it_joins_two_words_with_and
    list = ListWithCommas.new
    list.items = ['apple', 'orange']
    assert('apple and orange' == list.join)
  end

  def test_it_joins_three_words_with_commas
    list = ListWithCommas.new
    list.items = ['apple', 'orange', 'pear']
    assert('apple, orange, and pear' == list.join)
  end

end
```

test_list_with_commas.rb

Тесты для класса ListWithCommas (продолжение)

3. Как и в случае с предыдущим тестом, файл *list_with_commas.rb* должен быть сохранен в каталоге *lib*, а файл *test_list_with_commas.rb* — в каталоге *test*. В свою очередь, эти каталоги должны находиться в одном каталоге проекта.



4. После того как файлы будут сохранены в правильных каталогах, в терминальном окне перейдите в основной каталог проекта. Затем введите следующую команду:

```
ruby -I lib test/test_list_with_commas.rb
```

Ваши модульные тесты выполняются и выводят результат, сходный с приведенным ниже.

Переходим в основной каталог проекта.

Выполняем тестовый файл в Ruby.

Один тест прошел, другой не прошел.

Информация о непрошедшем тесте отображается в сводке.

```
File Edit Window Help
$ cd my_project
$ ruby -I lib test/test_list_with_commas.rb
Run options: --seed 39924

# Running:

.F

Finished in 0.000804s, 2487.5622 runs/s,
2487.5622 assertions/s.

  1) Failure:
TestListWithCommas#test_it_joins_two_words_with_and
[test/test_list_with_commas.rb:8]:
Failed assertion, no message given.

2 runs, 2 assertions, 1 failures, 0 errors, 0 skips
```

Из сводки видно, что тест для трех элементов был выполнен успешно, но тест для двух элементов не прошел. Мы достигли «красной» фазы цикла «красный, зеленый, рефакторинг»! При наличии работоспособного теста ошибка в классе `ListWithCommas` исправляется достаточно легко.

☑ Для элементов `['apple', 'orange', 'pear']` метод `join` должен возвращать строку `"apple, orange, and pear"`.

Тест проходит.

☒ Для элементов `['apple', 'orange']` метод `join` должен возвращать строку `"apple and orange"`.

Тест не проходит!

Исправление ошибки

Сейчас для класса `ListWithCommas` написаны два теста. Тест для *трех* элементов в списке проходит, а тест для *двух* элементов не проходит:

```
class ListWithCommas
  attr_accessor :items
  def join
    last_item = "and #{items.last}"
    other_items = items.slice(0, items.length - 1).join(', ')
    "#{other_items}, #{last_item}"
  end
end
```

Тест проходит. ☑ Для элементов `['apple', 'orange', 'pear']` метод `join` должен возвращать строку `"apple, orange, and pear"`.

Тест не проходит! ☒ Для элементов `['apple', 'orange']` метод `join` должен возвращать строку `"apple and orange"`.

Это происходит из-за того, что метод `join` класса `ListWithCommas` включает лишнюю запятую при выводе списка, состоящего всего из двух элементов.

```
two_subjects = ListWithCommas.new
two_subjects.items = ['my parents', 'a rodeo clown']
puts "A photo of #{two_subjects.join}"
```

Запятая здесь не нужна!

```
A photo of my parents, and a rodeo clown
```

Изменим `join` так, чтобы список, состоящий всего из двух элементов, просто соединялся связкой *and*. Полученная строка возвращается без выполнения остального кода.

```
class ListWithCommas
  attr_accessor :items
  def join
    if items.length == 2
      return "#{items[0]} and #{items[1]}"
    end
    last_item = "and #{items.last}"
    other_items = items.slice(0, items.length - 1).join(', ')
    "#{other_items}, #{last_item}"
  end
end
```

Если в списке всего два элемента...

...соединить их связкой «and» и пропустить остальной код.

list_with_commas.rb

Исправление ошибки (продолжение)

Мы внесли изменения в код, но стал ли он от этого работать правильно? Тесты немедленно дадут ответ на этот вопрос! Как и прежде, введите следующую команду в терминальном окне:

```
ruby -I lib test/test_list_with_commas.rb
```

Оба теста теперь проходят успешно!

Запускаем тестовый файл.

Оба теста проходят!

Ни ошибок, ни проблем.

```
File Edit Window Help
$ ruby -I lib test/test_list_with_commas.rb
Run options: --seed 18716

# Running:

..

Finished in 0.001321s, 1514.0045 runs/s, 1514.0045
assertions/s.

2 runs, 2 assertions, 0 failures, 0 errors, 0 skips
```

Тест проходит. ☑ Для элементов `['apple', 'orange', 'pear']` метод `join` должен возвращать строку `"apple, orange, and pear"`.

Тест проходит. ☑ Для элементов `['apple', 'orange']` метод `join` должен возвращать строку `"apple and orange"`.

Тесты проходят, разработка находится в «зеленой» фазе! Можно с уверенностью сказать, что `join` работает со списками из двух элементов, потому что соответствующий модульный тест теперь проходит. И нам не нужно беспокоиться о том, не нарушили ли внесенные изменения работоспособность другого кода: у нас имеется модульный тест, который подтверждает, что все работает правильно.

Теперь мы можем использовать новый класс, не сомневаясь в его надежности!

```
two_subjects = ListWithCommas.new
two_subjects.items = ['my parents', 'a rodeo clown']
puts "A photo of #{two_subjects.join}"
three_subjects = ListWithCommas.new
three_subjects.items = ['my parents', 'a rodeo clown', 'a prize bull']
puts "A photo of #{three_subjects.join}"
```

Для двух элементов лишняя запятая не включается.

И с тремя элементами работает правильно.

```
A photo of my parents and a rodeo clown
A photo of my parents, a rodeo clown, and a prize bull
```

Еще одна ошибка

Вполне может оказаться, что класс `ListWithCommas` будет использоваться со списком, содержащим всего один элемент. Однако метод `join` в таком случае ведет себя некорректно: он обходится с единственным элементом так, словно тот находится в конце более длинного списка:

```
one_subject = ListWithCommas.new
one_subject.items = ['a rodeo clown']
puts "A photo of #{one_subject.join}"
```

Класс обходится с единственным элементом так, словно он находится в конце списка!

```
A photo of , and a rodeo clown
```

Как метод `join` *должен* вести себя в таком случае? Для списка из одного элемента не нужно вообще ничего — ни запятых, ни слова *and*. Метод должен просто вернуть этот единственный элемент.

```
A photo of a rodeo clown
```

Список из одного элемента должен выглядеть примерно так.

Попробуем представить это требование в виде модульного теста. Создадим экземпляр `ListWithCommas` и назначим его атрибуту `items` массив, содержащий только один элемент. Затем добавим проверку тестового условия, согласно которой метод `join` должен возвращать строку, состоящую только из этого элемента.

```
require 'minitest/autorun'
require 'list_with_commas'

class TestListWithCommas < Minitest::Test

  def test_it_prints_one_word_alone
    list = ListWithCommas.new
    list.items = ['apple']
    assert('apple' == list.join)
  end

  ...

end
```

Присваиваем список, состоящий из одного элемента.

Следует ожидать строку, состоящую только из этого элемента.

test_list_with_commas.rb

Сообщения об ошибках

Давайте проверим новый тест в деле.

Запускаем тестовый файл.

В сводке присутствует сообщение о том, что тест не прошел.

```
File Edit Window Help
$ ruby -I lib test/test_list_with_commas.rb
...
  1) Failure:
TestListWithCommas#test_it_prints_one_word_alone
[test/test_list_with_commas.rb:9]:
Failed assertion, no message given.

3 runs, 3 assertions, 1 failures, 0 errors, 0 skips
```

Тест не прошел!

К сожалению, никакой другой информации, описывающей причину сбоя, мы не получаем:

```
Failed assertion, no message given.
```

Как же получить более подробную информацию? Существуют два разных способа.

Во-первых, можно создать сообщение с описанием о сбое теста. Метод `assert` получает необязательный второй параметр с сообщением, которое должно выводиться в случае сбоя теста. Попробуем добавить это сообщение:

Это сообщение будет выведено в том случае, если тест не проходит.

```
...
class TestListWithCommas < Minitest::Test

  def test_it_prints_one_word_alone
    list = ListWithCommas.new
    list.items = ['apple']
    assert('apple' == list.join, "Return value didn't equal 'apple'")
  end
  ...
end
```

test_list_with_commas.rb

При попытке выполнения обновленного теста новое сообщение об ошибке выводится в составе сводки.

Наше новое сообщение.

```
File Edit Window Help
$ ruby -I lib test/test_list_with_commas.rb
...
  1) Failure:
TestListWithCommas#test_it_prints_one_word_alone
[test/test_list_with_commas.rb:9]:
Return value didn't equal 'apple'

3 runs, 3 assertions, 1 failures, 0 errors, 0 skips
```

Другой способ проверки равенства двух значений

И хотя новое сообщение об ошибке оказывается более содержательным, оно по-прежнему не сообщает, *почему* произошел сбой. Было бы полезно, если бы сообщение показывало, что *именно* вернул метод `join`, чтобы это значение можно было сравнить с *ожидаемым*...

Второй (и более простой) способ получения более содержательного сообщения об ошибке основан на использовании другого метода проверки. Метод `assert` = всего лишь один из многих методов, наследуемых тестовыми классами от `Minitest::Test`.

Также существует метод `assert_equal`, который получает два аргумента и проверяет, что они равны. Если они не равны, тест не проходит, как и в случае с `assert`. Но важнее другое: при этом в сводке выводится как фактическое, так и ожидаемое значение, что позволяет нам легко сравнить их.

Так как вызовы `assert` в нашей программе фактически выполняют проверку равенства, их можно заменить вызовами `assert_equal`. В первом аргументе `assert_equal` должно передаваться ожидаемое значение, а во втором — фактическое значение, возвращаемое тестируемым кодом.

```
require 'minitest/autorun'
require 'list_with_commas'

class TestListWithCommas < Minitest::Test

  def test_it_prints_one_word_alone
    list = ListWithCommas.new
    list.items = ['apple']
    assert_equal('apple', list.join)
  end

  def test_it_joins_two_words_with_and
    list = ListWithCommas.new
    list.items = ['apple', 'orange']
    assert_equal('apple and orange', list.join)
  end

  def test_it_joins_three_words_with_commas
    list = ListWithCommas.new
    list.items = ['apple', 'orange', 'pear']
    assert_equal('apple, orange, and pear', list.join)
  end

end
```

Ожидается строка, состоящая только из этого элемента.

test_list_with_commas.rb

Другой способ проверки равенства двух значений (продолжение)

Попробуем снова выполнить тест и посмотрим, не будет ли результат более содержательным.

```
File Edit Window Help
$ ruby -I lib test/test_list_with_commas.rb
Run options: --seed 39624

# Running:

..F

Finished in 0.001493s, 2009.3771 runs/s, 2009.3771
assertions/s.

  1) Failure:
TestListWithCommas#test_it_prints_one_word_alone
[test/test_list_with_commas.rb:9]:
Expected: "apple"
  Actual: ", and apple"

3 runs, 3 assertions, 1 failures, 0 errors, 0 skips
```

Два теста проходят, один не проходит — как и прежде.

В сообщении о сбое теста приводятся два значения: ожидаемое и фактическое.

Да, вот они: ожидаемое значение (`"apple"`) и то, которое было реально получено в программе (`", and apple"`)!

Теперь понятно, что именно пошло не так, и мы сможем легко исправить ошибку. Для этого в код `ListWithCommas` будет включено еще одно условие `if`. Если список содержит только один элемент, нужно просто вернуть этот элемент.

Заменим «if» на «elsif».

```
class ListWithCommas
  attr_accessor :items
  def join
    if items.length == 1
      return items[0]
    elsif items.length == 2
      return "#{items[0]} and #{items[1]}"
    end
    last_item = "and #{items.last}"
    other_items = items.slice(0, items.length - 1).join(', ')
    "#{other_items}, #{last_item}"
  end
end
```

Если в списке всего один элемент...

...возвращаем этот элемент, пропуская весь оставшийся код.

list_with_commas.rb

Повторив тестирование, мы видим, что все тесты проходят успешно!

```
3 runs, 3 assertions, 0 failures, 0 errors, 0 skips
```

Другие методы проверки условий

Как упоминалось ранее, тестовые классы наследуют многие методы проверки условий от класса `Minitest::Test`. Вам уже знаком метод `assert`, который проходит нормально при получении истинного значения и выдает сбой при получении ложного значения:

```
assert(true)   # Проходит.
assert(false)  # Не проходит!
```

И вы видели метод `assert_equal`, который получает два значения. Проверка не проходит, если эти значения не равны:

```
assert_equal(1, 1)  # Проходит.
assert_equal(1, 2)  # Не проходит!
```

Ниже кратко описаны другие методы проверки условий...

Метод `assert_includes` получает в первом аргументе коллекцию, а во втором произвольный объект. Проверка не проходит, если заданный объект отсутствует в коллекции.

```
assert_includes(['apple', 'orange'], 'apple')   # Проходит, потому что массив включает значение «apple».
assert_includes(['apple', 'orange'], 'pretzel') # Не проходит, потому что в массиве нет значения «pretzel»!
```

Метод `assert_instance_of` получает в первом аргументе класс, а во втором произвольный объект. Проверка не проходит, если объект не является экземпляром заданного класса.

```
assert_instance_of(String, 'apple')  # Проходит, потому что «apple» является экземпляром String.
assert_instance_of(Fixnum, 'apple')  # Не проходит, потому что «apple» НЕ ЯВЛЯЕТСЯ экземпляром Fixnum!
```

Наконец, метод `assert_raises` получает в аргументах один или несколько классов исключений. Также метод получает блок. Если блок *не* инициирует исключение, соответствующее одному из заданных классов, тест не проходит. (Это может быть полезно, если вы написали код, который должен инициировать ошибку в определенных обстоятельствах, и хотите убедиться в том, что ошибка действительно возникает в положенный момент.)

```
assert_raises(ArgumentError) do  # Проходит, потому что блок инициирует ArgumentError.
  raise ArgumentError, "That didn't work!"
end

assert_raises(ArgumentError) do  # Не проходит, потому что блок НЕ инициирует ArgumentError.
  "Everything's fine!"
end
```

Упражнение

Все приведенные ниже фрагменты взяты из тестовых методов MiniTest. Рядом с каждой проверкой условия напишите, пройдет или не пройдет этот тест.

```
.........  assert_equal('apples', 'apples')

.........  assert_includes([1, 2, 3, 4, 5], 3)

.........  assert_instance_of(String, 42)

.........  assert_includes(['a', 'b', 'c'], 'd')

.........  assert_raises(RuntimeError) do
             raise "Oops!"
           end

.........  assert('apples' == 'oranges')

.........  assert_raises(StandardError) do
             raise ZeroDivisionError, "Oops!"
           end

.........  assert_instance_of(Hash, {})
```

Упражнение
Решение

Все приведенные ниже фрагменты взяты из тестовых методов MiniTest. Рядом с каждой проверкой условия напишите, пройдет или не пройдет этот тест.

Pass
```
assert_equal('apples', 'apples')
```

Pass
```
assert_includes([1, 2, 3, 4, 5], 3)
```

Fail
```
assert_instance_of(String, 42)
```

Fail
```
assert_includes(['a', 'b', 'c'], 'd')
```

Pass
```
assert_raises(RuntimeError) do
  raise "Oops!"
end
```

Помните: если класс исключения не указан, то по умолчанию инициируется RuntimeError.

Fail
```
assert('apples' == 'oranges')
```

Fail
```
assert_raises(StandardError) do
  raise ZeroDivisionError, "Oops!"
end
```

Инициирует исключение, тип которого отличается от ожидаемого!

Pass
```
assert_instance_of(Hash, {})
```

Устранение дублирования кода из тестов

В наших тестах присутствует повторяющийся код... Каждый тест начинается с создания экземпляра `ListWithCommas`.

```
require 'minitest/autorun'
require 'list_with_commas'

class TestListWithCommas < Minitest::Test

  def test_it_prints_one_word_alone
    list = ListWithCommas.new <-- Повторение
    list.items = ['apple']
    assert_equal('apple', list.join)
  end

  def test_it_joins_two_words_with_and
    list = ListWithCommas.new <-- Повторение
    list.items = ['apple', 'orange']
    assert_equal('apple and orange', list.join)
  end

  def test_it_joins_three_words_with_commas
    list = ListWithCommas.new <-- Повторение
    list.items = ['apple', 'orange', 'pear']
    assert_equal('apple, orange, and pear', list.join)
  end

end
```

test_list_with_commas.rb

Вполне естественно, что при проведении множественных тестов для одного типа объекта для настройки каждого теста потребуются сходные действия. По этой причине в классе MiniTest был предусмотрен механизм, который предотвращает повторение кода...

Метод «setup»

MiniTest ищет в тестовом классе метод экземпляра с именем `setup`, и если такой метод будет найден, выполняет его перед каждым тестом.

```
require 'minitest/autorun'

class TestSetup < Minitest::Test
  def setup
    puts "In setup"
  end
  def test_one
    puts "In test_one"
  end
  def test_two
    puts "In test_two"
  end
end
```

Метод «setup» выполняется перед первым тестом. →

Метод «setup» снова выполняется перед вторым тестом. →

```
...
In setup
In test_one
In setup
In test_two
...
```

Метод `setup` может использоваться для подготовки объектов для вашего теста.

```
class TestSetup < Minitest::Test
  def setup
    @oven = SmallOven.new
    @oven.turn_on
  end
  def test_bake
    @oven.contents = 'turkey'
    assert_equal('golden-brown turkey', @oven.bake)
  end
  def test_empty_oven
    @oven.contents = nil
    assert_raises(RuntimeError) { @oven.bake }
  end
end
```

Создаем объект для каждого теста. ←

Используем объект из «setup». ←

Используем объект из «setup». ←

Обратите внимание: если вы собираетесь использовать `setup`, очень важно сохранять создаваемые объекты в переменных *экземпляра*. Если использовать *локальные* переменные, они выйдут из области видимости при выполнении тестового метода.

```
class TestSetup < Minitest::Test
  def setup
    oven = SmallOven.new
    oven.turn_on
  end
  def test_bake
    oven.contents = 'turkey'
    assert_equal('golden-brown turkey', oven.bake)
  end
end
```

Не используйте локальные переменные! ←

Переменная «oven» вышла из области видимости! ←

Error →

```
undefined local variable or method `oven'
```

Метод «teardown»

MiniTest также ищет в тестовом классе второй метод экземпляра с именем `teardown`. Если такой метод присутствует, он будет выполняться *после* каждого теста.

```
require 'minitest/autorun'

class TestSetup < Minitest::Test
  def teardown
    puts "In teardown"
  end
  def test_one
    puts "In test_one"
  end
  def test_two
    puts "In test_two"
  end
end
```

Метод «teardown» выполняется после первого теста.

Метод «teardown» снова выполняется после второго теста.

```
...
In test_one
In teardown
In test_two
In teardown
...
```

Метод `teardown` пригодится в тех ситуациях, когда после выполнения каждого теста необходимо провести некоторые завершающие действия («зачистку»).

```
class TestSetupAndTeardown < Minitest::Test
  def setup
    @oven = SmallOven.new
    @oven.turn_on
  end
  def teardown
    @oven.turn_off
  end
  def test_bake
    @oven.contents = 'turkey'
    assert_equal('golden-brown turkey', @oven.bake)
  end
  def test_empty_oven
    @oven.contents = nil
    assert_raises(RuntimeError) { @oven.bake }
  end
end
```

Вызывается после выполнения каждого теста.

Методы `setup` и `teardown` выполняются до и после *каждого* теста, а не однократно. И хотя существует только одна копия вашего подготовительного кода, для каждого выполняемого теста создается новый, «чистый» объект. (Только представьте, какая путаница возникнет, если бы изменения, внесенные в объект предыдущим тестом, могли влиять на результат следующего теста.)

А теперь воспользуемся новыми знаниями и попробуем избавиться от дублирования кода в тестах `ListWithCommas`...

Обновление кода для использования метода «setup»

В написанном ранее коде в каждом тестовом методе создавался новый экземпляр `ListWithCommas`. Давайте переместим повторяющийся код в метод `setup`. Объект каждого теста будет храниться в переменной экземпляра `@list`, к которой мы будем затем обращаться из тестовых методов.

```
require 'minitest/autorun'
require 'list_with_commas'

class TestListWithCommas < Minitest::Test

  def setup
    @list = ListWithCommas.new
  end

  def test_it_prints_one_word_alone
    @list.items = ['apple']
    assert_equal('apple', @list.join)
  end

  def test_it_joins_two_words_with_and
    @list.items = ['apple', 'orange']
    assert_equal('apple and orange', @list.join)
  end

  def test_it_joins_three_words_with_commas
    @list.items = ['apple', 'orange', 'pear']
    assert_equal('apple, orange, and pear', @list.join)
  end

end
```

Создание экземпляра ListWithCommas перемещается сюда.

Переходим на использование переменной экземпляра.

Переходим на использование переменной экземпляра.

Переходим на использование переменной экземпляра.

test_list_with_commas.rb

Код стал гораздо чище! А если запустить тесты, вы увидите, что все они проходят точно так же, как прежде.

```
File Edit Window Help
$ ruby -I lib test/test_list_with_commas.rb
Run options: --seed 13205

# Running:

...

Finished in 0.000769s, 3901.1704 runs/s, 3901.1704
assertions/s.

3 runs, 3 assertions, 0 failures, 0 errors, 0 skips
```

Без сбоев и ошибок

У бассейна

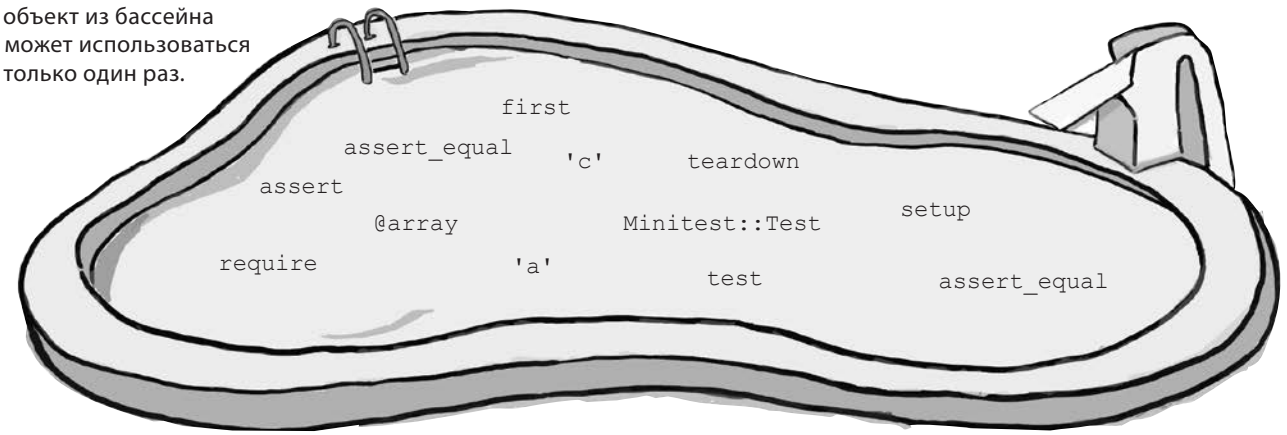
Выловите из бассейна фрагменты кода и расставьте их в пустых местах в коде. Каждый фрагмент может использоваться **только один** раз, причем использовать все фрагменты не обязательно. Ваша **задача** — составить код, который будет нормально выполняться и выдавать приведенный ниже результат.

```
_______ 'minitest/autorun'

class TestArray < _______________

  def _____
    @array = ['a', 'b', 'c']
  end

  def test_length
    _____________(3, _______.length)
  end

  def test_last
    assert_equal(___, @array.last)
  end

  def test_join
    _____________('a-b-c', @array.join('-'))
  end

end
```

Результат:

```
$ ruby test_setup.rb
Run options: --seed 60370

# Running:

...

Finished in 0.000752s, 3989.3617
runs/s, 3989.3617 assertions/s.

3 runs, 3 assertions, 0 failures,
0 errors, 0 skips
```

Примечание: каждый объект из бассейна может использоваться только один раз.



У бассейна. Решение

```
require 'minitest/autorun'

class TestArray < Minitest::Test

  def setup
    @array = ['a', 'b', 'c']
  end

  def test_length
    assert_equal(3, @array.length)
  end

  def test_last
    assert_equal('c', @array.last)
  end

  def test_join
    assert_equal('a-b-c', @array.join('-'))
  end

end
```

Результат:

```
$ ruby test_setup.rb
Run options: --seed 60370

# Running:

...

Finished in 0.000752s, 3989.3617
runs/s, 3989.3617 assertions/s.

3 runs, 3 assertions, 0 failures,
0 errors, 0 skips
```

Ваш инструментарий Ruby

Глава 13 осталась позади, а ваш инструментарий пополнился методами модульного тестирования.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Загрузка MiniTest командой `require 'minitest/autorun'` обеспечивает автоматический запуск тестов после загрузки.
- Чтобы создать модульный тест MiniTest , необходимо сначала определить субкласс `Minitest::Test`.
- MiniTest ищет методы экземпляра тестового класса, начинающиеся с `test_`, и запускает их. Каждый такой метод представляет модульный тест.
- В тестовом методе можно выполнить тест и передать его результат методу `assert`. Если `assert` получает ложное значение, тест не проходит. Если `assert` получает истинное значение, тест проходит.
- По общепринятым соглашениям основной код программы должен храниться в файлах в подкаталоге проекта с именем *lib*. Модульные тесты должны храниться в подкаталоге *test*.
- При вызове `assert` можно передать необязательный второй аргумент с сообщением об ошибке. Если тест не пройдет, то сообщение будет выведено в сводке результатов тестирования.
- Метод `assert_equal` получает два аргумента и сообщает о сбое теста, если аргументы не равны.
- Многие другие методы проверки условий — такие, как `assert_includes`, `assert_raises` и `assert_instance_of` — наследуются от `Minitest::Test`.
- Если добавить в тестовый класс метод экземпляра `setup`, он будет вызываться *до* выполнения каждого теста. Метод может использоваться для создания объектов, используемых в тестах.
- Если добавить в тестовый класс метод экземпляра `teardown`, он будет вызываться *после* выполнения каждого теста. Метод может использоваться для выполнения завершающих действий после тестов.

Далее в программе...

Последние страницы все ближе! Пора проверить ваши новые навыки на практике. В следующих двух главах мы создадим полноценное веб-приложение. Не пугайтесь; в этом нам поможет Sinatra — библиотека, которая существенно упрощает весь процесс!

На раздаче HTML

На дворе XXI век. Пользователи требуют веб-приложения. И Ruby поможет вам в этом! Существуют библиотеки, которые помогут вам развернуть собственные веб-приложения и обеспечить доступ к ним из любого браузера. Итак, в двух последних главах этой книги мы покажем вам, как построить полноценное веб-приложение.

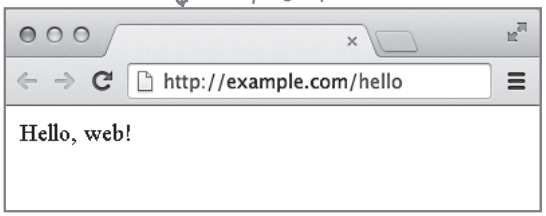
Прежде всего вам понадобится Sinatra — независимая библиотека для создания веб-приложений. Но не беспокойтесь, мы научим вас использовать программу RubyGems (включенную в поставку Ruby) для автоматизации загрузки и установки библиотек! Затем будет рассмотрена разметка HTML — ровно столько, сколько необходимо для создания собственных веб-страниц. И конечно, мы покажем, как предоставить эти страницы браузеру!

Написание веб-приложений на Ruby

Пора нам распрощаться с терминальным окном...

```
File Edit Window Help
$ ruby hello.rb
Hello, world!
```

...и поздороваться с браузером!

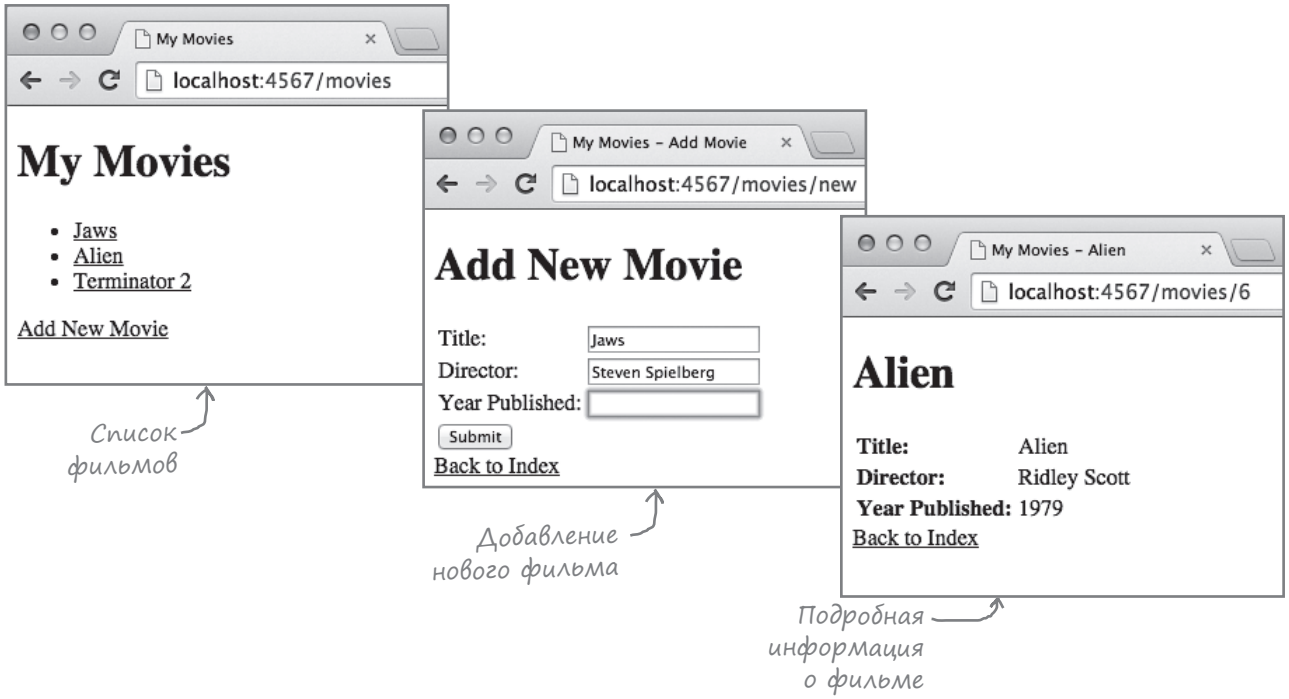


Приложения, работающие в терминальном окне, очень удобны — когда ими пользуетесь только вы. Но обычные пользователи уже избалованы возможностями Интернета и Всемирной Паутины. Они не захотят учиться пользоваться терминальным окном, чтобы работать с вашим приложением. Они даже не пожелают его устанавливать. Они хотят, чтобы приложение было готово к работе в тот момент, когда они щелкнут на ссылке в браузере.

Но не беспокойтесь! Ruby поможет вам и в создании веб-приложений.

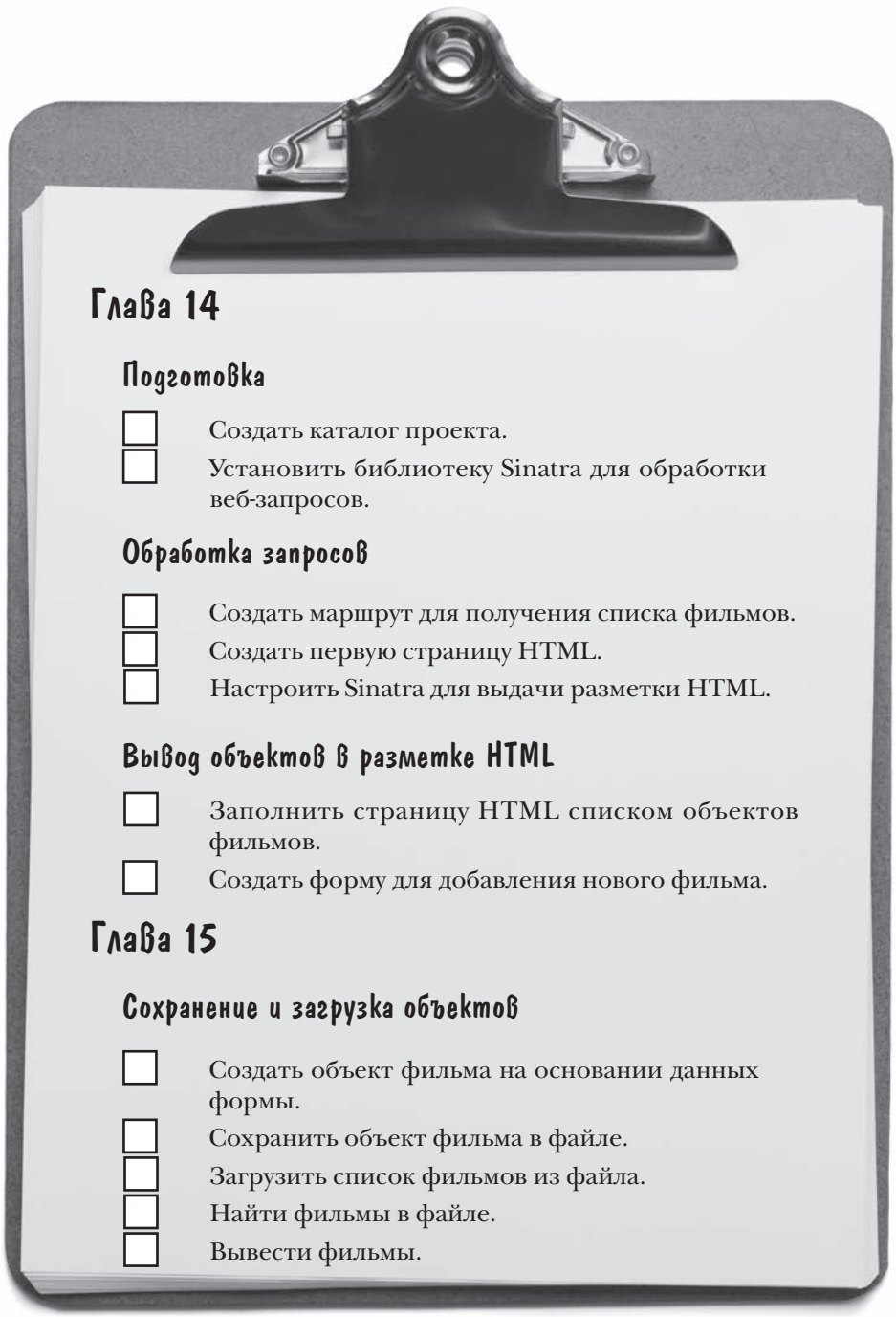
Будем откровенны: написание полноценного веб-приложения — непростая задача, даже если вы используете Ruby. Она потребует от вас всего, чему вы научились раньше, а также ряда новых навыков. Но в Ruby существуют превосходные библиотеки, которые упрощают этот процесс настолько, насколько это возможно!

В двух последних главах книги мы построим простую онлайновую базу данных фильмов. Вы сможете ввести подробную информацию о фильмах, и эта информация будет сохранена в файле. Сайт будет предоставлять список ссылок на все введенные фильмы, а щелчок на ссылке откроет информацию о соответствующем фильме.



Список задач

В этих двух главах нам предстоит довольно серьезная работа. К счастью, весь процесс будет разбит на небольшие шаги. Посмотрим, что же вообще нужно сделать...



Глава 14

Подготовка

- [ ] Создать каталог проекта.
- [ ] Установить библиотеку Sinatra для обработки веб-запросов.

Обработка запросов

- [ ] Создать маршрут для получения списка фильмов.
- [ ] Создать первую страницу HTML.
- [ ] Настроить Sinatra для выдачи разметки HTML.

Вывод объектов в разметке HTML

- [ ] Заполнить страницу HTML списком объектов фильмов.
- [ ] Создать форму для добавления нового фильма.

Глава 15

Сохранение и загрузка объектов

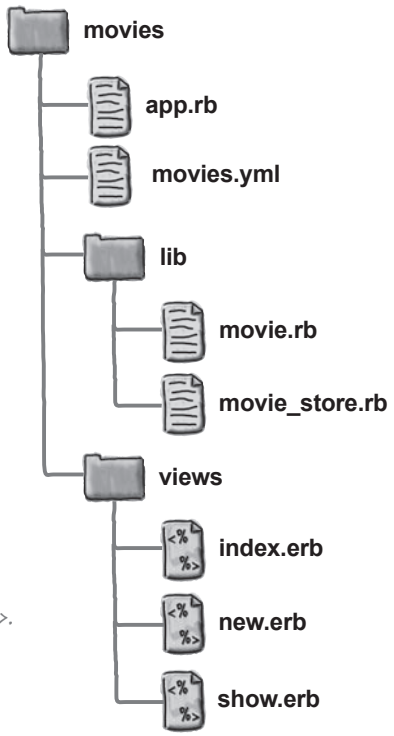
- [ ] Создать объект фильма на основании данных формы.
- [ ] Сохранить объект фильма в файле.
- [ ] Загрузить список фильмов из файла.
- [ ] Найти фильмы в файле.
- [ ] Вывести фильмы.

В этой главе основное внимание уделяется созданию страниц HTML и их отображению в браузере. Глава завершается заполнением формы, на которой пользователи вводят новые данные фильмов...

Затем в главе 15 мы покажем, как использовать данные формы для настройки атрибутов объектов Ruby и как сохранить эти объекты в файле. После этого вы сможете снова загрузить данные и вывести их в том формате, который вам покажется наиболее уместным.

Структура каталогов проекта

К концу главы 15 папка нашего проекта будет выглядеть примерно так:

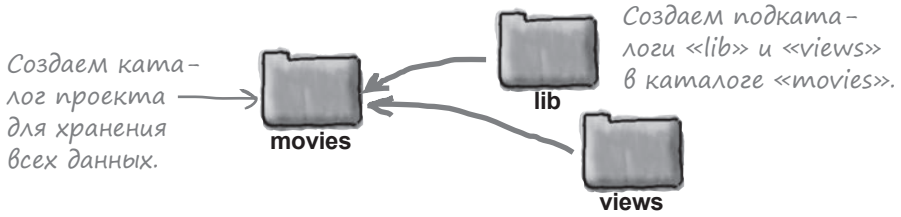


В ходе работы в нашем приложении появится целый набор файлов, поэтому для хранения всего проекта потребуется отдельный каталог. Присвойте ему любое имя на свое усмотрение; мы выбрали для своего каталога проекта имя *movies*.

В подкаталоге проекта также должны располагаться два подкаталога. Первый, *lib*, используется для хранения файлов с исходным кодом Ruby (по аналогии с каталогом *lib*, созданным в главе 13).

Также потребуются средства для просмотра данных приложения. Пользователь будет просматривать данные в браузере, а это означает, что мы будем использовать файлы в формате HTML. Для них в каталоге проекта рядом с каталогом *lib* создается второй каталог с именем *views*.

Итак, создайте каталог с именем *movies*, а в нем создайте два каталога с именами *lib* и *views*. (Пока эти каталоги остаются пустыми; они будут заполняться файлами в процессе работы.)



Вот и все, что нужно сделать для подготовки структуры каталогов проекта. Первая задача выполнена!

Подготовка

- [x] Создать каталог проекта.
- [ ] Установить библиотеку Sinatra для обработки веб-запросов.

На следующем шаге необходимо «установить библиотеку Sinatra для обработки веб-запросов». Но что это за библиотека? И что такое «веб-запрос»?

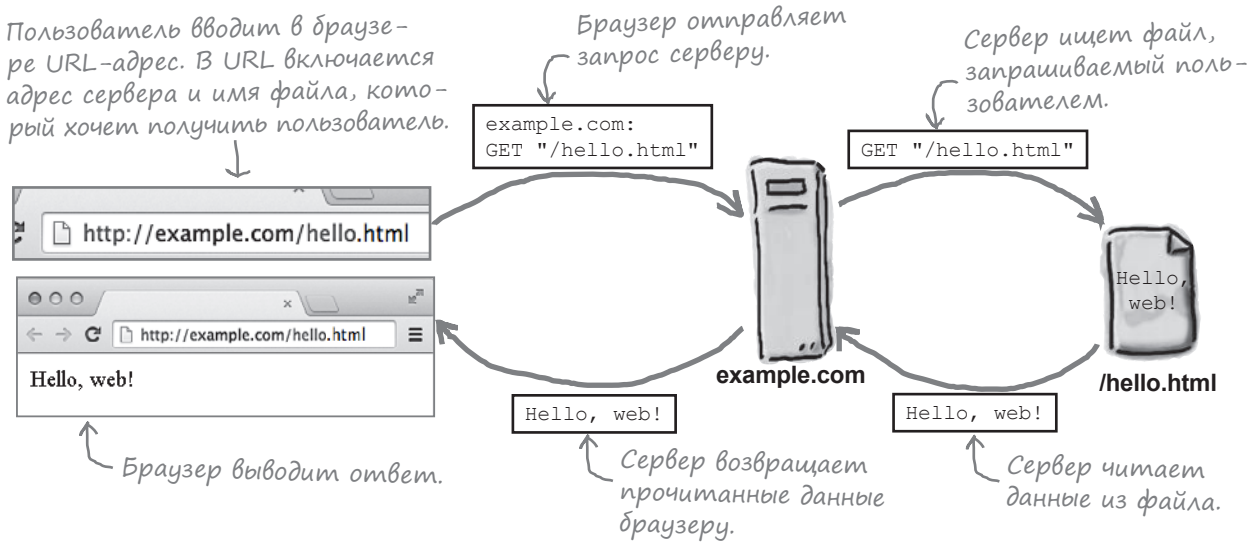
Чтобы создать это приложение, не обязательно знать HTML.

Мы расскажем вам о языке HTML ровно столько, сколько необходимо для понимания примеров. Если же вы захотите заняться веб-программированием на более высоком уровне, мы рекомендуем обратиться к книге Элизабет Робсон и Эрика Фримена *Изучаем HTML, XHTML и CSS. 2-е изд.*

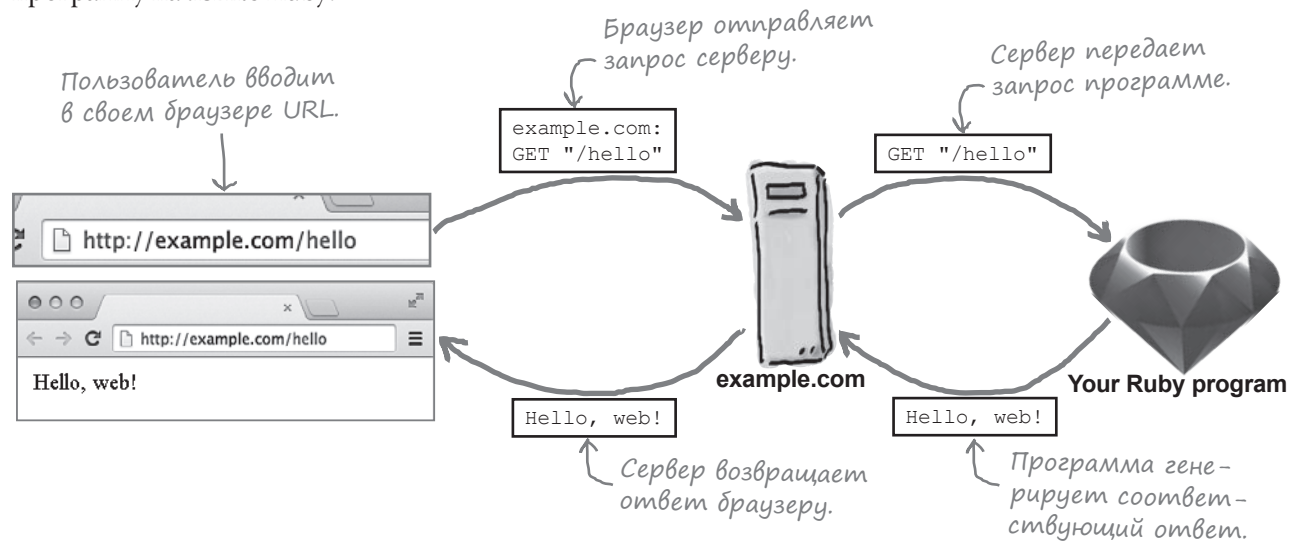
Браузеры, запросы, серверы и ответы

Когда вы вводите URL-адрес в своем браузере, вы на самом деле отправляете *запрос* на получение веб-страницы. Этот запрос передается *серверу*. Задача сервера — получить соответствующую страницу и вернуть браузеру *ответ* с этой страницей.

Когда Всемирная Паутина делала свои первые шаги, сервер обычно читал содержимое файла HTML на жестком диске сервера и отправлял разметку HTML браузеру.



Но в наши дни сервер гораздо чаще взаимодействует с *программой* для выполнения запросов, а не читает данные из файлов. И ничто не мешает написать такую программу на языке Ruby!



Sinatra получает запросы

Обработка запросов, полученных от браузера, — непростое дело. К счастью, нам не придется делать все самостоятельно. Библиотека Sinatra возьмет на себя технические подробности получения запросов и возвращения ответов. Все, что от нас потребуется, — написать код, *генерирующий* эти ответы.

Однако Sinatra не входит в базовую часть Ruby (классы и модули, загружаемые при каждом запуске Ruby). Эта библиотека даже не входит в стандартную библиотеку Ruby (классы и библиотеки, которые распространяются вместе с Ruby, но должны явно загружаться программой). Sinatra — независимый продукт, разработкой которого занимается сторонняя фирма.

Часто Задаваемые Вопросы

В: Разве для написания веб-приложений не нужно использовать Ruby on Rails?

О: Многие разработчики уже слышали о Ruby on Rails — это самая популярная инфраструктура веб-программирования, написанная на Ruby. Но это *не единственная* инфраструктура. Библиотека Sinatra тоже чрезвычайно популярна, отчасти из-за своей простоты. Если полноценное приложение Rails состоит из десятков классов и файлов с исходным кодом, приложение Sinatra может состоять из нескольких строк кода. Приложения Sinatra обычно считаются более понятными и доступными, чем приложения Rails. Вот почему для этого вводного приложения была выбрана библиотека Sinatra. Если вы собираетесь в будущем заняться изучением Rails — не беспокойтесь! То, что вы узнаете в этих главах, пригодится и в Rails!

Загрузка и установка библиотек с использованием RubyGems

Когда язык Ruby только начинал свое существование, пользователю, желающему использовать библиотеку (например, Sinatra), приходилось основательно потрудиться:

- Найти и загрузить файлы с исходным кодом.
- Распаковать найденные файлы.
- Найти подходящий каталог на жестком диске для их размещения.
- Добавить этот каталог в `LOAD_PATH` (список каталогов, в которых Ruby ищет загружаемые файлы).
- Если же библиотека зависела от *других* библиотек, пользователю приходилось повторять те же действия для каждой из *этих* библиотек.

Не стоит и говорить, что все это было очень хлопотно. Вот почему Ruby теперь распространяется вместе со служебной программой RubyGems, которая берет на себя все технические сложности. С RubyGems процесс требует куда меньших усилий со стороны пользователя:

- Авто библиотеки использует программу командной строки RubyGems для упаковки своих исходных файлов в один пакет, предназначенный для дальнейшего распространения.
- Программа RubyGems отправляет пакет на центральный сервер по адресу rubygems.org. (Сайт является бесплатным; он существует на пожертвования сообщества.)
- Пользователь — также при помощи программы RubyGems — указывает имя нужного пакета.
- RubyGems загружает пакет, распаковывает его, устанавливает в центральном каталоге пакетов на жестком диске пользователя и включает его в список `LOAD_PATH`. Затем те же действия выполняются для всех остальных пакетов, от которых зависит запрашиваемый пакет.

Программа RubyGems включена в поставку Ruby, так что она уже установлена в вашей системе. И она открывает вам доступ к огромной подборке библиотек; одно из недавних посещений rubygems.org показало, что свыше 6500 пакетов уже доступно на сервере и новые пакеты добавляются каждую неделю.

Установка библиотеки Sinatra

Давайте воспользуемся RubyGems для установки Sinatra. Для этого шага вам потребуется активное подключение к Интернету.

RubyGems запускается командой `gem`. Введите в терминальном окне команду **`gem install sinatra`**. (Неважно, какой каталог при этом является текущим.) RubyGems загружает и устанавливает несколько других пакетов, от которых зависит Sinatra, после чего устанавливает саму библиотеку Sinatra.

Команда для установки Sinatra.

RubyGems загружает и устанавливает другие пакеты, от которых зависит Sinatra.

Загрузка и установка самой библиотеки Sinatra.

```
File Edit Window Help
$ gem install sinatra
Fetching: rack-1.6.4.gem (100%)
Successfully installed rack-1.6.4
Fetching: rack-protection-1.5.3.gem (100%)
Successfully installed rack-protection-1.5.3
Fetching: tilt-2.0.1.gem (100%)
Successfully installed tilt-2.0.1
Fetching: sinatra-1.4.6.gem (100%)
Successfully installed sinatra-1.4.6
4 gems installed
$
```

После того как пакет будет установлен, достаточно включить в любую программу на языке Ruby строку `require 'sinatra'` — библиотека Sinatra будет загружаться при запуске этой программы!

Будьте осторожны!

Некоторые операционные системы не разрешают обычному пользователю устанавливать файлы в каталог *gems*.

По соображениям безопасности Mac OS X, Linux и другие системы на базе Unix обычно разрешают сохранение файлов в каталоге, в котором хранятся пакеты, только программам с правами администратора. Если ваша ОС принадлежит к их числу, может появиться сообщение об ошибке:

```
$ gem install sinatra
ERROR:  While executing gem ... (Gem::FilePermissionError)
    You don't have write permissions into the /var/lib/gems directory.
```

В таком случае попробуйте выполнить команду `gem` *от имени администратора; для этого в начало команды добавляется команда* `sudo` *(сокращение от "super-user do"):* `sudo gem install sinatra`. *Возможно, ОС запросит у вас пароль, после чего начнется обычный процесс установки.*

```
$ sudo gem install sinatra
[sudo] password for jay:
Fetching: rack-1.6.4.gem (100%)
Successfully installed rack-1.6.4
...
4 gems installed
```

Простое приложение Sinatra

Пакет Sinatra установлен, вторая задача тоже выполнена.

Теперь можно перейти к обработке запросов, полученных от браузера.

Впрочем, нам еще предстоит разобраться в основах Sinatra. Прежде чем браться за приложение, для разминки потратим несколько страниц на рассмотрение более простого примера.

Подготовка

- [x] Создать каталог проекта.
- [x] Установить библиотеку Sinatra для обработки веб-запросов.

Обработка запросов

- [ ] Создать маршрут для получения списка фильмов.
- [ ] Создать первую страницу HTML.
- [ ] Настроить Sinatra для выдачи разметки HTML.

Ниже приведен код законченного приложения Sinatra, которое возвращает короткий ответ браузеру. Сохраните код в файле с именем *hello_web.rb*.

Сохраните этот код в файле.

```
require 'sinatra'

get('/hello') do
  'Hello, web!'
end
```

hello_web.rb

Несложно, правда? Sinatra берет на себя все сложности обработки веб-запросов. Вскоре мы опишем происходящее более подробно, но сначала опробуем его на практике. В терминальном окне перейдите в каталог, в котором был сохранен файл, и введите команду:

```
ruby hello_web.rb
```

Запускаем сервер.

Сообщения от сервера.

```
File Edit Window Help
$ ruby hello_web.rb
[2015-07-10 20:05:45] INFO  WEBrick 1.3.1
== Sinatra (v1.4.6) has taken the stage on 4567
for development with backup from WEBrick
[2015-07-10 20:05:45] INFO
WEBrick::HTTPServer#start: pid=6742 port=4567
```

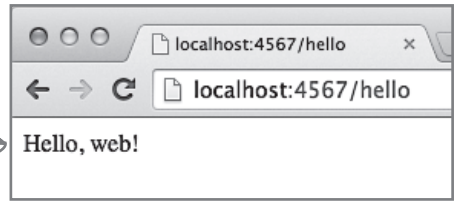
Приложение немедленно начинает выводить на консоль диагностические сообщения, чтобы мы знали, что оно работает. Теперь нужно подключить к нему браузер и протестировать. Откройте браузер и введите в адресной строке следующий URL-адрес. (Если он покажется вам немного странным, не беспокойтесь: вскоре мы объясним, что все это значит.)

```
http://localhost:4567/hello
```

Браузер отправляет запрос серверу, который отвечает сообщением `"Hello, web!"`. Запрос также отображается в списке сообщений в терминальном окне. Вы только что впервые передали данные браузеру!

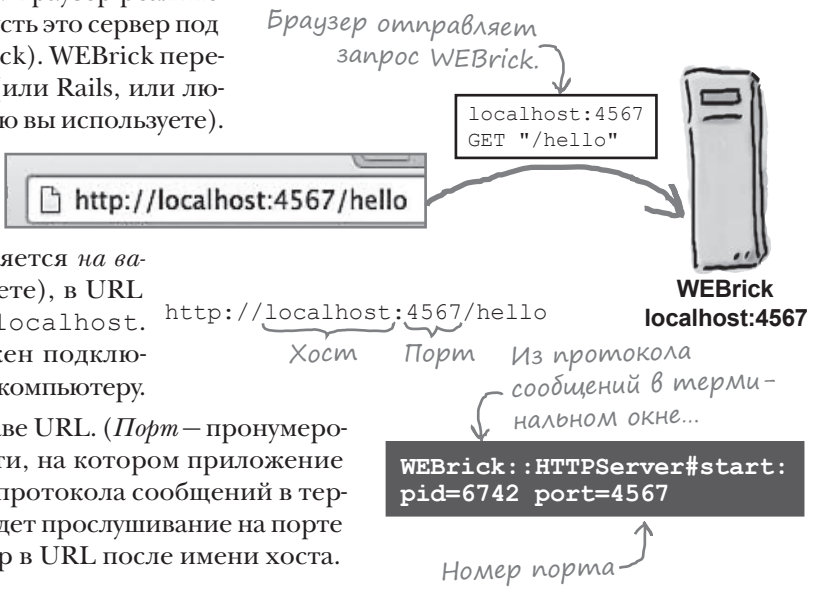
Sinatra продолжает прослушивать входящие запросы до тех пор, пока этот процесс не будет прерван. Когда вам надоест рассматривать страницу, нажмите Ctrl-C в терминальном окне, чтобы приказать Ruby прервать выполнение программы.

Ответ нашего приложения!



Ваш компьютер разговаривает сам с собой

Когда вы запустили приложение Sinatra, оно запустило свой собственный веб-сервер прямо на вашем компьютере — для этого оно использовало библиотеку WEBrick (часть стандартной библиотеки Ruby). Браузер реально устанавливает связь с веб-сервером — пусть это сервер под управлением Sinatra (такой, как WEBrick). WEBrick передает все полученные запросы Sinatra (или Rails, или любой другой веб-инфраструктуре, которую вы используете).



Так как серверная программа выполняется *на вашем компьютере* (а не где-то в Интернете), в URL используется специальное имя хоста `localhost`. Оно сообщает браузеру, что он должен подключиться *с* вашего компьютера *к* тому же компьютеру.

Также необходимо указать порт в составе URL. (*Порт* — пронумерованный канал передачи данных по сети, на котором приложение может прослушивать сообщения.) Из протокола сообщений в терминальном окне видно, что WEBrick ведет прослушивание на порте 4567, поэтому мы включаем этот номер в URL после имени хоста.

Часто Задаваемые Вопросы

В: Я получаю сообщение об ошибке, в котором говорится, что браузер не смог подключиться к серверу!

О: Вероятно, ваш сервер не работает. Просмотрите сообщения об ошибках в терминальном окне. Также присмотритесь повнимательнее к имени хоста и номеру порта в браузере — возможно, вы допустили опечатку.

В: Почему в URL нужно включать номер порта? На других сайтах это не нужно!

О: Веб-серверы обычно ведут прослушивание на порте 80, потому что именно этот порт используется по умолчанию браузерами. Но во многих операционных системах по соображениям безопасности для запуска службы, ведущей прослушивание на порте 80, необходимы специальные разрешения. Для повседневной работы это создаст слишком много хлопот, поэтому для приложений, находящихся в процессе разработки, Sinatra настраивает WEBrick для прослушивания порта 4567.

В: В моем браузере загружается страница с текстом: «Sinatra doesn't know this ditty».

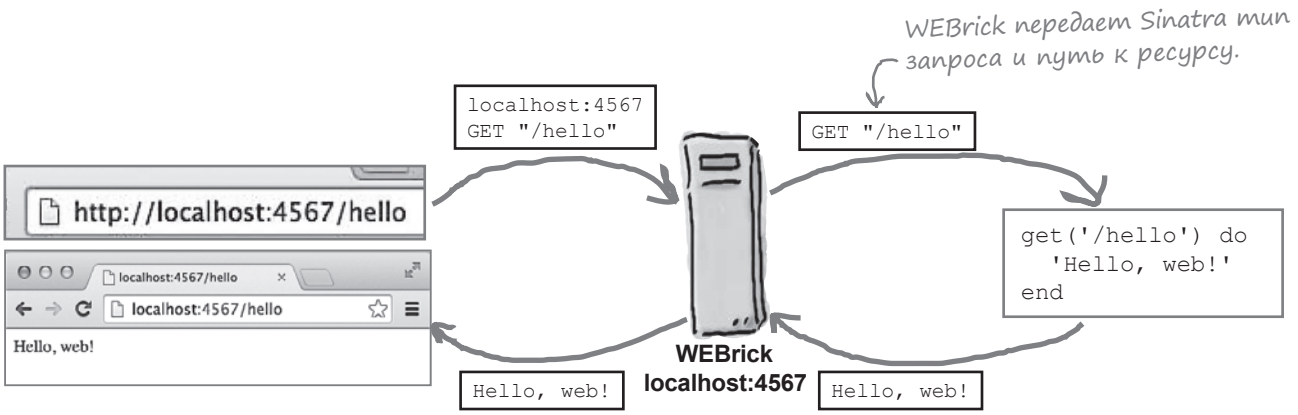
О: Это ответ от сервера, и это хорошо — но он также означает, что запрошенный вами ресурс не найден. Проверьте, что URL-адрес завершается строкой */hello*, а в имени файла *hello_web.rb* нет опечатки.

В: При попытке запустить приложение я получаю сообщение о том, что порт 4567 уже используется, или исключение «Адрес уже используется»!

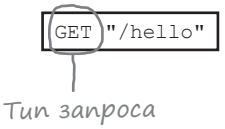
О: Ваш сервер Sinatra пытается вести прослушивание на том же порте, что и другая программа (чего ОС делать не разрешает). Запускали несколько экземпляров Sinatra? В таком случае нажимали ли вы Ctrl-C в терминальном окне, чтобы остановить старый экземпляр перед запуском нового? Также можно задать другой порт при помощи параметра командной строки `-p` (таким образом, команда `sinatra myserver.rb -p 8080` заставляет Sinatra вести прослушивание на порте 8080).

Тип запроса

Итак, благодаря присутствию имени хоста и порта в URL запрос браузера передается WEBrick. Затем WEBrick передает запрос Sinatra для генерирования запроса.



При формировании запроса библиотека Sinatra должна учитывать *тип* запроса. Тип запроса указывает, какое *действие* хочет выполнить браузер.



Во взаимодействиях браузеров и серверов используется протокол HTTP (сокращение от «**H**yper**T**ext **T**ransfer **P**rotocol»). Протокл HTTP определяет несколько *методов* (которые не являются методами Ruby — просто одинаковые названия), которые могут использоваться запросом. Наиболее распространенные методы HTTP:

- GET: используется тогда, когда браузер хочет что-то *получить* от сервера, — обычно из-за того, что вы ввели URL или щелкнули на ссылке. Это может быть страница HTML, изображение или другой ресурс.
- POST: используется тогда, когда браузер хочет *добавить* данные на сервер, — обычно при отправке пользователем формы с новыми данными.
- PUT: браузер хочет *изменить* существующие данные на сервере — обычно при отправке пользователем формы с измененными данными.
- DELETE: браузер хочет *удалить* данные с сервера.

Наш текущий запрос является запросом типа GET. Мы вводим URL-адрес, а браузер знает, что мы хотим что-то *получить*. Соответственно, он выдает запрос нужного типа.

Путь к ресурсу

Но *что* мы должны получить в ответ на запрос GET? На сервере обычно хранится множество разных ресурсов, которые он может отправить браузеру, включая страницы HTML, графические изображения и т. д.

Ответ находится в конце введенного URL-адреса, после имени хоста и порта:

```
http://localhost:4567/hello
                      Путь
```

Это *путь* к ресурсу. Он сообщает серверу, с каким из его многочисленных ресурсов вы хотите выполнить действие. Sinatra берет путь, завершающий URL-адрес, и использует его для принятия решения о том, как же ответить на запрос.

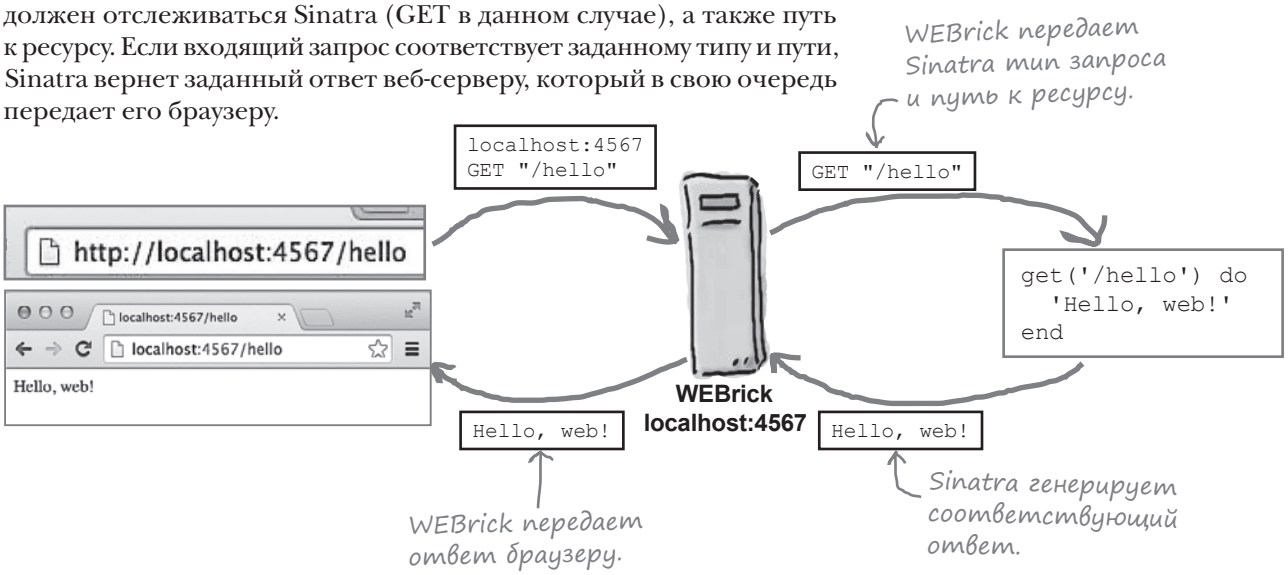
```
GET "/hello"
    Путь к ресурсу
```

Итак, Sinatra получит запрос GET для пути `'/hello'`. Как ответит Sinatra? А это решаете *вы* в *своем* коде!

Каждый запрос HTTP включает тип запроса (метод) и путь к ресурсу (тот ресурс, к которому вы обращаетесь).

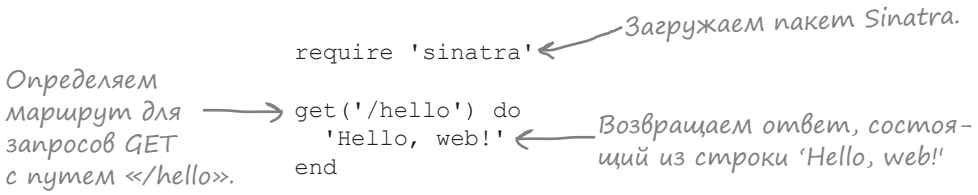
Маршруты Sinatra

Sinatra использует *маршруты* (routes) для принятия решений о том, как следует отвечать на запрос. Вы указываете тип запроса, который должен отслеживаться Sinatra (GET в данном случае), а также путь к ресурсу. Если входящий запрос соответствует заданному типу и пути, Sinatra вернет заданный ответ веб-сервер, который в свою очередь передает его браузеру.



Наш сценарий начинается с команды `require 'sinatra'`, которая загружает библиотеку Sinatra. При этом также определяются некоторые методы для создания маршрутов, включая метод с именем `get`.

Мы вызываем метод `get`, передавая аргумент со строкой `'/hello'` и блок. При этом также создается маршрут для запросов GET с путем `'/hello'`. Sinatra сохраняет переданный блок и вызывает этот блок каждый раз, когда получает запрос GET с путем `'/hello'`.



Какое бы значение ни вернул блок, Sinatra вернет это значение веб-серверу в своем ответе. Мы определяем этот блок так, чтобы он возвращал строку `'Hello, web!'` — и именно эта строка отображается в браузере!

Множественные маршруты в одном приложении Sinatra

Впрочем, ваше приложение не может отвечать `'Hello, web!'` на каждый входящий запрос. Оно должно по-разному реагировать на разные пути запросов.

Для этого можно создать разные маршруты для каждого пути. Просто вызовите `get` для каждого пути и предоставьте блок, возвращающий соответствующий ответ. Тогда ваше приложение сможет реагировать на запросы по любому из этих путей.

Определяем маршрут для запросов GET с путем «/hello».

Возвращаем ответ, состоящий из строки 'Hello, web!'

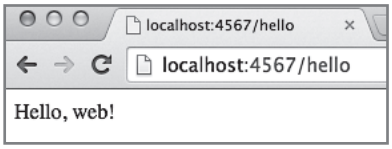
Определяем маршрут для запросов requests с путем «/salut».

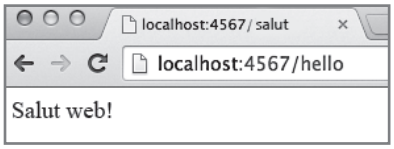
Возвращаем ответ, состоящий из строки 'Salut web!'

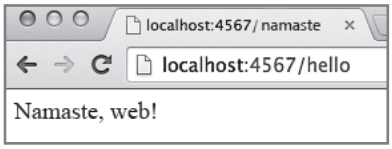
Определяем маршрут для запросов GET с путем «/namaste».

Возвращаем ответ, состоящий из строки 'Namaste, web!'

```
require 'sinatra'

get('/hello') do
  'Hello, web!'
end

get('/salut') do
  'Salut web!'
end

get('/namaste') do
  'Namaste, web!'
end
```







Часто Задаваемые Вопросы

В: Я изменил код своего приложения, но веб-страница не обновляется, когда я щелкаю на кнопке Reload в браузере!

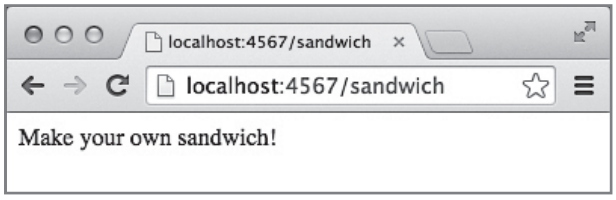
О: Чтобы загрузить изменения в память, необходимо перезапустить приложение. (В Ruby on Rails предусмотрена автоматическая перезагрузка кода при его изменении, но Sinatra не делает это по умолчанию.) Вам придется нажать клавиши Ctrl-C в терминальном окне, чтобы прервать работу Ruby, а затем перезапустить ваше приложение.

Развлечения с магнитами

На холодильнике разложена программа на языке Ruby. Сможете ли вы расставить фрагменты так, чтобы у вас получилось приложение Sinatra, которое получает запросы GET для пути `'/sandwich'` и выдает приведенный ниже результат?

```
'Make your own sandwich!'
'sinatra'
end
get
do
require
('/sandwich')
```

Результат:

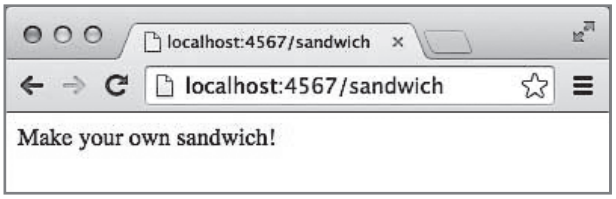


Развлечения с магнитами. Решение

На холодильнике разложена программа на языке Ruby. Сможете ли вы расставить фрагменты так, чтобы у вас получилось приложение Sinatra, которое получает запросы GET для пути `'/sandwich'` и выдает приведенный ниже результат?

```
require 'sinatra'

get ('/sandwich') do
  'Make your own sandwich!'
end
```

Результат:



Маршрут для списка фильмов

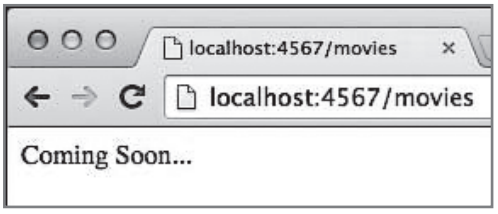
Теперь, когда вы знаете, как создаются маршруты Sinatra, мы можем создать маршрут для списка фильмов! Сохраните этот код в файле с именем *app.rb*. Разместите файл *app.rb* в каталоге проекта, рядом с подкаталогами *lib* и *views*.

Тем самым вы создаете маршрут для запросов GET к пути `'/movies'`. Пока мы используем простой текстовый заполнитель.

Чтобы запустить приложение, перейдите в каталог, в котором оно находится, и введите команду **`app.rb`**. Когда Sinatra запустится, введите адрес `http://localhost:4567/movies` для просмотра страницы.

```
require 'sinatra'

get('/movies') do
  'Coming Soon...'
end
```


```
File  Edit  Window  Help
$ cd /tmp/movies
$ ruby app.rb
[2015-07-16 23:17:35] INFO
WEBrick 1.3.1
== Sinatra (v1.4.6) has taken
 the stage...
```



Очередная задача выполнена. На следующем шаге временный текст будет заменен реальной разметкой HTML!

Обработка запросов

- [x] Создать маршрут для получения списка фильмов.
- [ ] Создать первую страницу HTML.
- [ ] Настроить Sinatra для выдачи разметки HTML.

Создание списка фильмов в формате HTML

До настоящего момента мы передавали браузеру только фрагменты текста. Чтобы применить к тексту форматирование, необходимо использовать разметку HTML. В языке HTML для определения форматирования текста используются теги.

Даже если вы никогда не имели дела с HTML, ничего страшного: мы кратко изложим основы работы с разметкой на ходу.

Прежде чем пытаться отправлять HTML из Sinatra, попробуем создать простой файл HTML в текстовом редакторе. Сохраните приведенный ниже текст в файле с именем *index.html*.

Некоторые важные теги HTML, встречающиеся в файле:

- `<title>`: название страницы, отображаемое на вкладке браузера.
- `<h1>`: заголовок первого уровня. Обычно выводится крупным, жирным шрифтом.
- `<ul>`: неупорядоченный список (обычно выводится в виде маркированного списка).
- `<li>`: элемент списка. Несколько таких тегов, вложенных в теги `<ul>`, образуют список.
- `<a>`: определяет ссылку.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset='UTF-8' />
    <title>My Movies</title>
  </head>
  <body>
    <h1>My Movies</h1>
    <ul>
      <li>movies</li>
      <li>go</li>
      <li>here</li>
    </ul>
    <a href="/movies/new">Add New Movie</a>
  </body>
</html>
```

Название страницы, которое будет отображаться на вкладке в браузере.

Заголовок

Элемент списка

Элемент списка

Элемент списка

Ссылка

index.html

Теперь попробуем просмотреть разметку HTML в браузере. Запустите свой любимый браузер, выберите в меню команду «Открыть файл...» и откройте файл HTML, который вы только что сохранили.

Обратите внимание на соответствие элементов страницы и разметки HTML...

Вы можете попробовать щелкнуть на ссылке, но сейчас это приведет только к выдаче ошибки «страница не найдена». Проблема будет решена позднее, когда мы добавим страницу для создания новых фильмов.

Обращение к разметке HTML из Sinatra

Файл HTML отлично работает в браузере. Еще одна задача выполнена!

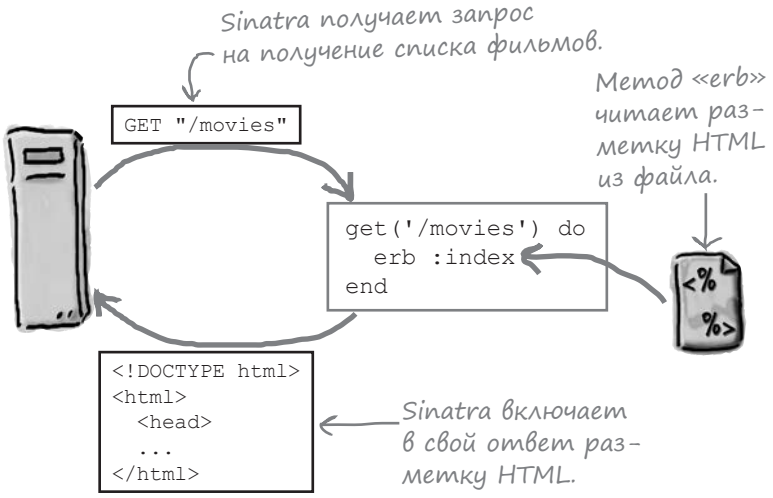
Но *никто другой* не сможет увидеть разметку HTML, пока мы не предоставим доступ к ней приложению Sinatra. К счастью, у этой проблемы есть простое решение: метод Sinatra `erb`.

Обработка запросов

- [x] Создать маршрут для получения списка фильмов.
- [x] Создать первую страницу HTML.
- [ ] Настроить Sinatra для выдачи разметки HTML.

Метод `erb` читает содержимое файла в каталоге app, чтобы вы могли включить это содержимое в ответ Sinatra. Он также использует библиотеку ERB (часть стандартной библиотеки Ruby), чтобы вы могли встроить код Ruby в файл для настройки содержимого каждого запроса. (ERB означает «**e**mbedded **Ruby**», то есть «встроенный Ruby».)

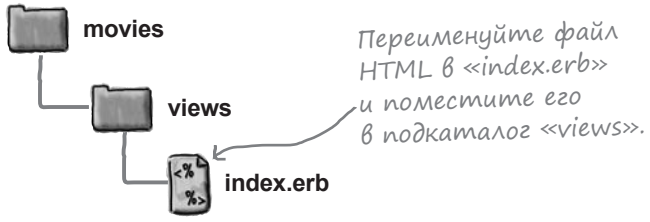
Итак, нам остается лишь поместить разметку HTML в конкретный файл, а затем вызвать `erb` для включения HTML в ответ Sinatra!



Метод `erb` получает в аргументе символическое имя и преобразует его в имя файла. По умолчанию в подкаталоге с именем *views* ищется файл с именем, завершающимся суффиксом *.erb*. (Такие файлы называются *шаблонами ERB*.)

При вызове:	*будет загружен следующий шаблон:*
`erb :index`	*movies/views/index.erb*
`erb :new`	*movies/views/new.erb*
`erb :show`	*movies/views/show.erb*

Мы хотим, чтобы при вызове `erb :index` загружалась наша разметка HTML. Значит, файл *index.html* необходимо переименовать в *index.erb*. Переместите файл в подкаталог *views* в каталоге проекта.



Обращение к разметке HTML из Sinatra (продолжение)

Разметка HTML была сохранена в шаблоне *index.erb* в подкаталоге *views*:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset='UTF-8' />
    <title>My Movies</title>
  </head>
  <body>
    <h1>My Movies</h1>
    <ul>
      <li>movies</li>
      <li>go</li>
      <li>here</li>
    </ul>
    <a href="/movies/new">Add New Movie</a>
  </body>
</html>
```

views/index.erb

А теперь настроим приложение, чтобы оно включало HTML в свой ответ. В созданном ранее файле *app.rb* замените временный текст `'Coming soon...'` вызовом `erb :index`.

Метод `erb` возвращает содержимое шаблона в виде строки. Поскольку вызов `erb` является последним выражением в блоке, он также определяет и возвращаемое значение блока. А то, что возвращает блок, становится ответом Sinatra для веб-сервера.

```
require 'sinatra'

get('/movies') do
  erb :index
end
```

Отвечает на запросы GET для пути «/movies».

Загружает «views/index.erb».

app.rb

А теперь посмотрим, работает ли программа. В терминальном окне перейдите в каталог проекта и запустите приложение командой **`ruby app.rb`**. Затем в браузере введите адрес:

```
http://localhost:4567/movies
```

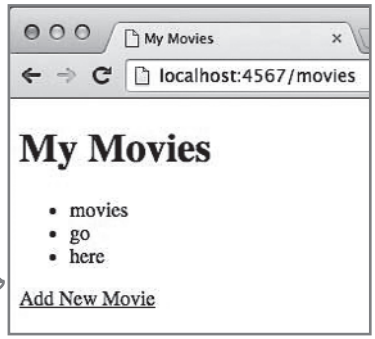
Sinatra выдает ответ с содержимым шаблона *index.erb*!

Переходим в каталог проекта.

Запускаем приложение.

```
File Edit Window Help
$ cd movies
$ ruby app.rb
[2015-07-10 20:05:45]
INFO  WEBrick 1.3.1
== Sinatra (v1.4.6) has
taken the stage on 4567
...
```

Sinatra выдает ответ с содержимым index.erb.



Класс для хранения данных фильмов

Шаблон ERB позволяет загрузить страницу HTML из Sinatra. Еще одна задача выполнена.

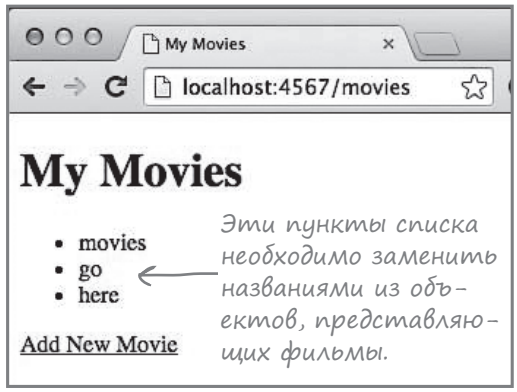
Обработка запросов

- [x] Создать маршрут для получения списка фильмов.
- [x] Создать первую страницу HTML.
- [x] Настроить Sinatra для выдачи разметки HTML.

Вывод объектов в разметке HTML

- [ ] Заполнить страницу HTML списком объектов фильмов.
- [ ] Создать форму для добавления нового фильма.

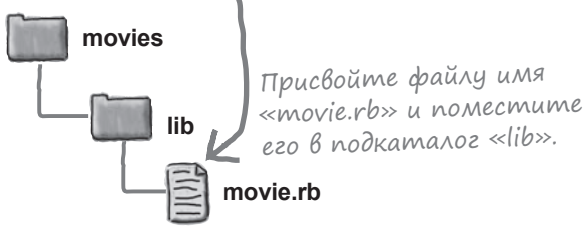
Однако в отображаемой разметке HTML там, где должен располагаться список фильмов, находится только временный текст. Нашей *следующей* задачей должно стать создание объектов Ruby для хранения данных фильмов и последующее использование этих объектов для заполнения списка в формате HTML.



Конечно, для создания объектов фильмов потребуется специальный класс. Сейчас мы создадим его. Это очень простой класс, имеющий всего три атрибута: `title`, `director` и `year`.

```
class Movie
  attr_accessor :title, :director, :year
end
```

Не стоит перемешивать код этого класса с кодом Sinatra, чтобы не создавать путаницы. Лучше сохраните код `Movie` в отдельном файле с именем *movie.rb* и поместите его в каталог *lib*.



Создание объекта Movie в приложении Sinatra

Давайте создадим объект `Movie` в нашем приложении Sinatra.

Чтобы использовать класс `Movie`, необходимо загрузить файл *movie.rb*. Для этого следует добавить вызов `require` в начало *app.rb*. (Помните: суффикс *.rb* указывать не обязательно; `require` добавит его за вас.)

Когда это будет сделано, мы можем создать новый экземпляр `Movie` в блоке для маршрута `'/movies'`. (Пока начнем с одного фильма, а построением списка займемся позднее.) Вскоре мы покажем, как использовать данные объекта `Movie` при генерировании страницы HTML.

Обратите внимание: объект `Movie` хранится в переменной *экземпляра*, а не в локальной переменной. Вскоре мы расскажем, чем объясняется такое решение.

```
require 'sinatra'
require 'movie'          <- Загружаем файл с классом Movie.

get('/movies') do
  @movie = Movie.new     <- Создаем новый объект Movie.
  @movie.title = "Jaws"
  erb :index
end
```

app.rb

В новой версии, которая пытается загрузить файл из каталога *lib*, приложение уже нельзя просто запустить командой `ruby app.rb`. Если вы попытаетесь это сделать, Ruby сообщит об ошибке при попытке загрузить *movie.rb*:

Файл movie.rb не найден!

```
File Edit Window Help
$ cd movies
$ ruby app.rb
in `require': cannot load such file -- movie (LoadError)
$
```

Как вы, возможно, помните из главы 13, необходимо добавить параметр `-I lib` в командную строку, чтобы каталог *lib* был включен в список каталогов, в которых Ruby ищет загружаемые файлы.

Добавляем «lib» в список каталогов, из которых Ruby может загружать файлы.

```
File Edit Window Help
...
$ ruby -I lib app.rb
[2015-07-17 15:58:12] INFO  WEBrick 1.3.1
== Sinatra (v1.4.6) has taken the stage on 4567 for
development with backup from WEBrick
```

Итак, объект получен. Как перенести его данные на страницу HTML? Сейчас узнаете...

Теги внедрения в ERB

На данный момент наша страница состоит из «статической» (неизменяемой) разметки HTML, а в том месте, где должны находиться названия фильмов, располагается временный текст. Как же вывести данные на странице?

Вспомните, что перед вами не обычный файл HTML, а шаблон с «встроенным Ruby». Это означает, что в разметку можно включить код Ruby. Когда метод `erb` загружает шаблон, он выполняет встроенный код. А код Ruby может использоваться для включения названий фильмов в HTML!

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset='UTF-8' />
    <title>My Movies</title>
  </head>
  <body>
    <h1>My Movies</h1>
    <ul>
      <li>movies</li>
      <li>go</li>
      <li>here</li>
    </ul>
    <a href="/movies/new">Add New Movie</a>
  </body>
</html>
```

Этот временный текст необходимо заменить.

views/index.erb

Однако код Ruby нельзя просто взять и вставить в середину файла. Система разбора ERB не будет знать, какие части шаблона содержат HTML, а в каких частях находится код Ruby. (Если интерпретатор Ruby попытается обработать разметку HTML — поверьте, ничего хорошего не получится.)

Если разметка HTML будет обрабатываться как код Ruby, произойдет ошибка!

Мы не можем просто взять и вставить код Ruby в середину разметки HTML!

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset='UTF-8' />
    <title>My Movies</title>
  </head>
  <body>
    <h1>My Movies</h1>
    <ul>
      <li>
        @movie.title
      </li>
      <li>go</li>
      <li>here</li>
    </ul>
    <a href="/movies/new">Add New Movie</a>
  </body>
</html>
```

ERB позволяет встраивать код Ruby в текстовый шаблон при помощи тегов внедрения.

```
index.erb:1: syntax error, unexpected '<'
<!DOCTYPE html>
 ^
index.erb:2: syntax error, unexpected '<'
<html>
...
```

Вместо этого код необходимо включить в специальные *теги*, которые используются для пометки кода Ruby в шаблонах ERB.

Тег внедрения вывода в ERB

Одним из наиболее часто используемых тегов ERB является тег внедрения вывода <%= %>. Встречая этот тег, ERB выполняет содержащийся в нем код Ruby, преобразует результат в строку (если потребуется) и подставляет эту строку в окружающий текст (вместо тега).

Вот как выглядит обновленная версия шаблона *index.erb* с несколькими тегами внедрения вывода:

```
<!DOCTYPE html>
<html>
  <body>
    <ul>
      <li><%= "A string" %></li>
      <li><%= 15.0 / 6.0 %></li>
      <li><%= Time.now %></li>
    </ul>
  </body>
</html>
```

Вставляется обычная строка.

Вставляется результат математической операции.

Вставляется текущее время.

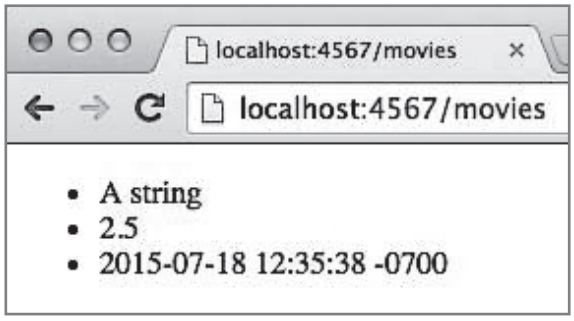
views/index.erb

А вот как выглядит шаблон после преобразования в HTML:

Все теги внедрения вывода заменяются результатом выполнения кода Ruby.

```
<!DOCTYPE html>
<html>
  <body>
    <ul>
      <li>A string</li>
      <li>2.5</li>
      <li>2015-07-18 12:35:38 -0700</li>
    </ul>
  </body>
</html>
```

А вот как выглядит разметка HTML в браузере:



Внедрение названия фильма в разметку HTML

А почему в блоке маршрута мы сохранили объект Movie в переменной **экземпляра**, а не в локальной переменной?

```
require 'sinatra'
require 'movie'

get('/movies') do
  @movie = Movie.new
  @movie.title = "Jaws"
  erb :index
end
```

Сохраняем новый объект Movie в переменной экземпляра.

app.rb

Переменные экземпляра, определенные в блоках маршрутов Sinatra, также доступны в шаблонах ERB. Это позволяет коду из *app.rb* работать в сочетании с кодом из шаблона ERB.

Сохраняя объект `Movie` в переменной экземпляра `@movie` внутри блока маршрута Sinatra, мы делаем ее доступной для использования внутри шаблона ERB! Это позволит нам интегрировать данные из приложения в HTML.

Заменим временный текст из *index.erb* тегом внедрения вывода с атрибутом `title` объекта `@movie`. Убедитесь в том, что приложение работает, перезагрузите страницу — и вы увидите название фильма!

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset='UTF-8' />
    <title>My Movies</title>
  </head>
  <body>
    <h1>My Movies</h1>
    <ul>
      <li><%= @movie.title %></li>
    </ul>
    <a href="/movies/new">Add New Movie</a>
  </body>
</html>
```

Обращаемся к объекту, определенному в app.rb.

views/index.erb

Данные из app.rb вставляются в index.erb!

Будьте осторожны!

Не пытайтесь обращаться к локальным переменным из блока маршрута Sinatra в шаблоне ERB!

Только переменные экземпляра из основного приложения будут находиться в области видимости в шаблоне. (Возможность присваивания локальных переменных в шаблонах существует, но информацию по этой теме вы найдете в документации Sinatra на сайте http://sinatrarb.com.*) Если вы попытаетесь обратиться к переменной, локальной для блока маршрута Sinatra, то получите сообщение об ошибке!*

У бассейна

Выловите из бассейна фрагменты кода и расставьте их в пустых местах в трех файлах. Каждый фрагмент может использоваться **только один** раз, причем использовать все фрагменты не обязательно. Ваша **задача** — составить приложение Sinatra, которое будет нормально выполняться и возвращать приведенные ниже ответы браузеру.

```
require _____

___('/addition') do
  @first = 3
  _____ = 5
  @result = _____ + @second
  erb _____
end

get(_____) do
  _____ = 2
  @second = 6
  _____ = @first * @second
  ___ :multiplication
end
```

app.rb

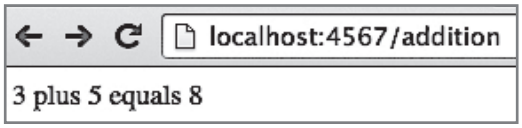
```
<!DOCTYPE html>
<html>
  <body>
    <%= @first %> plus
    <%= @second %> equals
    <%= _____ %>
  </body>
</html>
```

addition.erb

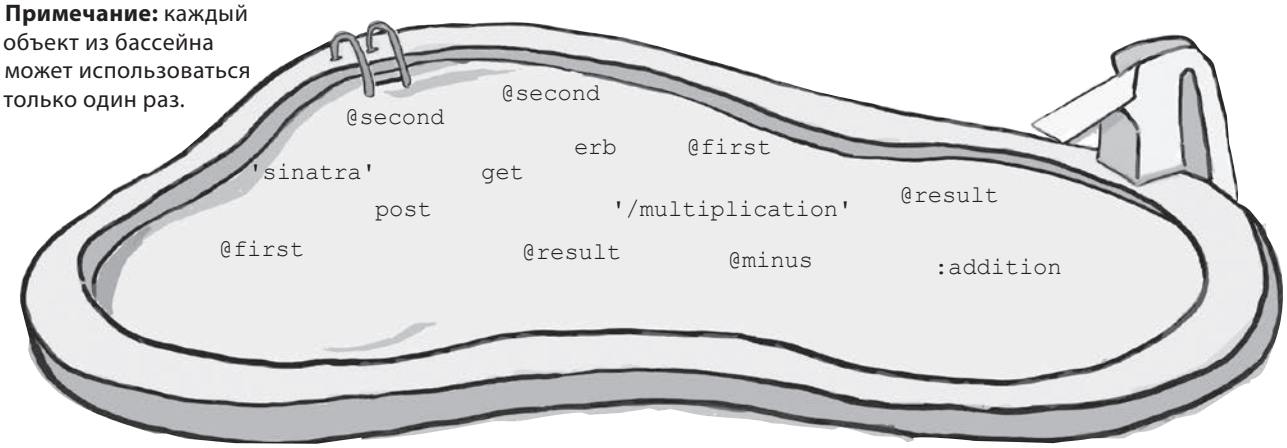
```
<!DOCTYPE html>
<html>
  <body>
    <%= @first %> times
    <%= _____ %> equals
    <%= @result %>
  </body>
</html>
```

multiplication.erb

Ответы браузеру:



Примечание: каждый объект из бассейна может использоваться только один раз.



У бассейна. Решение

Выловите из бассейна фрагменты кода и расставьте их в пустых местах в трех файлах. Каждый фрагмент может использоваться **только один** раз, причем использовать все фрагменты не обязательно. Ваша **задача** — составить приложение Sinatra, которое будет нормально выполняться и возвращать приведенные ниже ответы браузеру.

```
require 'sinatra'

get('/addition') do
  @first = 3
  @second = 5
  @result = @first + @second
  erb :addition
end

get('/multiplication') do
  @first = 2
  @second = 6
  @result = @first * @second
  erb :multiplication
end
```

app.rb

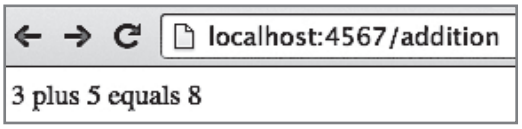
```
<!DOCTYPE html>
<html>
  <body>
    <%= @first %> plus
    <%= @second %> equals
    <%= @result %>
  </body>
</html>
```

addition.erb

```
<!DOCTYPE html>
<html>
  <body>
    <%= @first %> times
    <%= @second %> equals
    <%= @result %>
  </body>
</html>
```

multiplication.erb

Ответы браузеру:



Часто Задаваемые Вопросы

В: Я попытался использовать `puts` в теге внедрения вывода: `<%= puts "hello" %>`. Но никакого вывода я не получил! Почему?

О: Метод `puts` может вывести строку `"hello"` в *терминале*, но его *возвращаемое значение* равно `nil`, и именно *это* значение ERB использует в HTML (`nil` преобразуется в пустую строку). Не используйте `puts` в тегах; скорее всего, он сделает не то, что вам нужно.

Нормальный тег внедрения

Тег внедрения вывода `<%= %>` позволяет включить в разметку *один* фильм, а нужно включить целый *список*. Значит, шаблон ERB необходимо переработать.

На втором месте по частоте использования стоит тег ERB — `<% %>` — нормальный тег внедрения. (В отличие от тега внедрения вывода, он не содержит знака равенства.) Тег содержит код Ruby, который должен быть выполнен, но результаты которого не будут напрямую вставляться в вывод ERB.

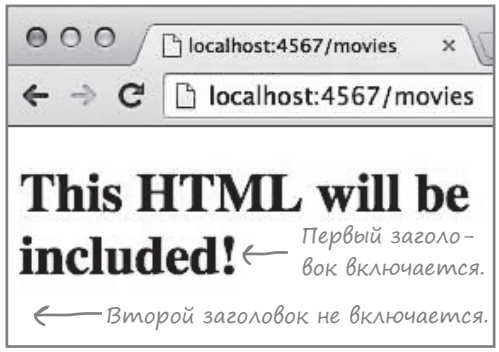
В частности, тег `<% %>` часто используется для включения фрагментов HTML только в том случае, если некоторое условие истинно (с использованием команд `if` и `unless`). Например, если включить следующий код в *index.erb*, то первый тег `<h1>` будет включен в вывод, а второй будет пропущен:

```
<!DOCTYPE html>
<html>
  <body>
    <% if true %>
      <h1>This HTML will be included!</h1>
    <% end %>
    <% if false %>
      <h1>This won't!</h1>
    <% end %>
  </body>
</html>
```

Разметка HTML до тега «<% end %>» будет включена.

Разметка HTML до тега «<% end %>» не включается.

views/index.erb



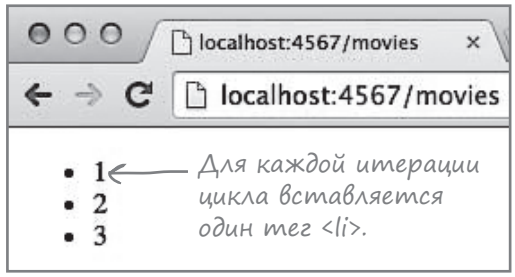
Нормальные теги внедрения также часто используются с циклами. Если встроить цикл в шаблон ERB, то вся разметка HTML или теги внедрения вывода в цикле тоже будут повторяться.

Если использовать приведенный ниже шаблон в *index.erb*, он переберет все элементы массива и вставит тег HTML `<li>` для каждого элемента. А поскольку значение параметра блока изменяется с каждой итерацией цикла, тег внедрения вывода будет каждый раз вставлять новое число.

```
<!DOCTYPE html>
<html>
  <body>
    <ul>
      <% [1, 2, 3].each do |number| %>
        <li><%= number %></li>
      <% end %>
    </ul>
  </body>
</html>
```

Вставляется три раза с разными номерами.

views/index.erb



Перебор названий фильмов в разметке HTML

Теперь, когда вы знаете, как использовать теги `<% %>` для включения элементов в массив, можно перейти к обработке всего массива объектов `Movie`. Итак, создадим массив для обработки. (Пока его содержимое будет жестко фиксировано — для начала. А потом массив фильмов будет загружаться из файла.)

В блоке маршрута `get('/movies')` в файле *app.rb* переменная экземпляра `@movie` будет заменена переменной `@movies` для хранения массива. Затем в массив будут добавлены объекты `Movie`.

```
require 'sinatra'
require 'movie'

get('/movies') do
  @movies = []
  @movies[0] = Movie.new
  @movies[0].title = "Jaws"
  @movies[1] = Movie.new
  @movies[1].title = "Alien"
  @movies[2] = Movie.new
  @movies[2].title = "Terminator 2"
  erb :index
end
```

Создаем массив фильмов.

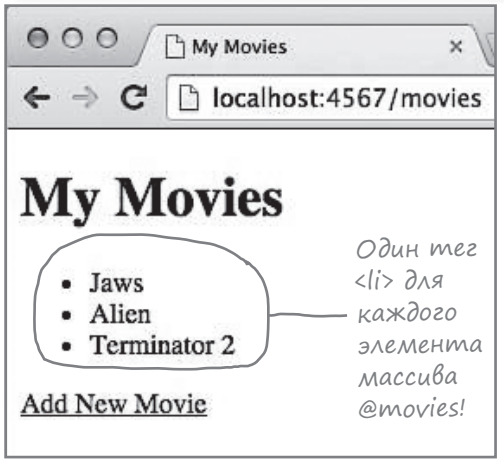
app.rb

Также изменения необходимо внести в шаблон ERB. В файл *index.erb* будут добавлены теги `<% %>`, которые перебирают каждый объект `Movie` в массиве. Между тегами `<% %>` добавляется тег HTML `<li>` и тег ERB для вывода названия текущего фильма.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset='UTF-8' />
    <title>My Movies</title>
  </head>
  <body>
    <h1>My Movies</h1>
    <ul>
      <% @movies.each do |movie| %>
        <li><%= movie.title %></li>
      <% end %>
    </ul>
    <a href="/movies/new">Add New Movie</a>
  </body>
</html>
```

views/index.erb

Перезапустите Sinatra и перезагрузите страницу. Вы увидите, что для каждого элемента массива `@movies` был вставлен тег HTML `<li>` с названием фильма.



Перебор названий фильмов в разметке HTML (продолжение)

Мы создали теги внедрения ERB для отображения списка названий фильмов в разметке HTML. Еще одна задача выполнена!

Вывод объектов в разметке HTML

- [x] Заполнить страницу HTML списком объектов фильмов.
- [ ] Создать форму для добавления нового фильма.

Упражнение

Ниже приведен шаблон ERB. Определите, какой результат он выведет, и заполните пропуски справа.

(Мы заполнили первое поле за вас.)

```
<!DOCTYPE html>
<html>
  <body>
    <% [1, 2, 3, 4].each do |number| %>
      <% if number.even? %>
        <li><%= number %> is even.</li>
      <% else %>
        <li><%= number %> is odd.</li>
      <% end %>
    <% end %>
  </body>
</html>
```

Результат:

```
<!DOCTYPE html>
<html>
  <body>
    <li>1 is odd.</li>
    .....
    .....
    .....
  </body>
</html>
```

Упражнение
Решение

Ниже приведен шаблон ERB. Определите, какой результат он выведет, и заполните пропуски справа.

```
<!DOCTYPE html>
<html>
  <body>
    <% [1, 2, 3, 4].each do |number| %>
      <% if number.even? %>
        <li><%= number %> is even.</li>
      <% else %>
        <li><%= number %> is odd.</li>
      <% end %>
    <% end %>
  </body>
</html>
```

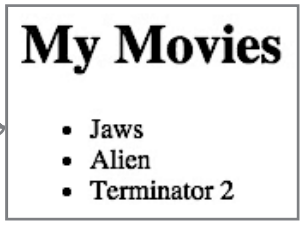
Результат:

```
<!DOCTYPE html>
<html>
  <body>
    <li>1 is odd.</li>
    <li>2 is even.</li>
    <li>3 is odd.</li>
    <li>4 is even.</li>
  </body>
</html>
```

Ввод данных на форме HTML

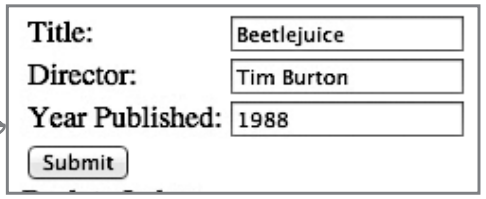
Сейчас наше приложение умеет выводить список фильмов в формате HTML, но он состоит всего из трех пунктов. Вряд ли кому-нибудь захочется создавать закладку на такой сайт. Чтобы приложение стало более интересным, нужно предоставить пользователю возможность вводить собственные описания фильмов.

Сейчас приложение выглядит так...



Чтобы пользователи могли добавлять фильмы на сайт, нам понадобятся *формы* HTML. Форма обычно содержит одно или несколько полей, в которых пользователи вводят данные, и кнопку для отправки данных на сервер.

Форма HTML.



Ниже приведена разметка HTML для очень простой формы. В ней встречаются теги, которые нам еще раньше не попадались:

- `<form>`: тег вмещает все остальные компоненты формы.
- `<label>`: метка одного из полей формы. Значение атрибута `for` должно соответствовать атрибуту `name` одного из элементов формы.
- `<input>` с атрибутом `type="text"`: текстовое поле для ввода строки. Его атрибут `name` будет использоваться для пометки значения поля в данных, передаваемых серверу (своего рода ключ хеша).
- `<input>` с атрибутом `type="submit"`: создает кнопку, при нажатии которой данные формы передаются серверу.

Метка для текстового поля «food».

Текст метки будет выводиться на странице.

```
<!DOCTYPE html>
<html>
  <body>
    <form>
      <label for="food">Enter your favorite food:</label>
      <input type="text" name="food">
      <input type="submit">
    </form>
  </body>
</html>
```

Текстовое поле «food».

Кнопка для отправки данных формы.

Если загрузить эту разметку HTML в браузере, она будет выглядеть примерно так:

Не так уж сложно! Давайте настроим наше приложение для загрузки формы HTML с данными фильма и отображения ее в браузере.

Получение формы HTML для добавления фильма

Сейчас при вводе адреса `http://localhost:4567/movies` приложение вызывает `erb :index` и выводит разметку HTML из файла *index.erb* в каталоге *views*. Теперь необходимо добавить вторую страницу с формой HTML для ввода описания нового фильма...

Для этого нужно добавить в *app.rb* второй маршрут, сразу же после первого. Мы настроим его для обработки запросов GET с путем `'/movies/new'`. В блок маршрута добавляется вызов `erb :new`, чтобы он загружал шаблон ERB из файла *new.erb* в каталоге *views*.

```
require 'sinatra'
require 'movie'

get('/movies') do
  @movies = []
  ...
  erb :index
end

get('/movies/new') do
  erb :new
end
```

Добавляем второй маршрут с другим путем.

Загружаем «views/new.erb».

app.rb

А теперь создадим шаблон ERB для нашей формы. Встраивать объекты Ruby в него не нужно, так что файл будет содержать простую разметку HTML без каких-либо тегов ERB.

В начале страницы тег `<title>` определяет ее название, а тег `<h1>` определяет заголовок.

Затем тег `<form>` обозначает начало формы. Вместо одного текстового поля на этой форме размещаются целых три поля: для названия фильма, для режиссера и для года выпуска. Все теги `<input>` имеют разные атрибуты `name`, по которым их можно отличить друг от друга. Каждое текстовое поле также имеет соответствующую метку `<label>`. Как и прежде, форма завершается кнопкой отправки данных.

В конце страницы размещается ссылка на путь */movies*. Щелчок на этой ссылке возвращает пользователя к списку фильмов.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset='UTF-8' />
    <title>My Movies — Add Movie</title>
  </head>
  <body>
    <h1>Add New Movie</h1>
    <form>
      <label for="title">Title:</label>
      <input type="text" name="title">
      <label for="director">Director:</label>
      <input type="text" name="director">
      <label for="year">Year Published:</label>
      <input type="text" name="year">
      <input type="submit">
    </form>
    <a href="/movies">Back to Index</a>
  </body>
</html>
```

Для этой страницы используется другое название документа и заголовок.

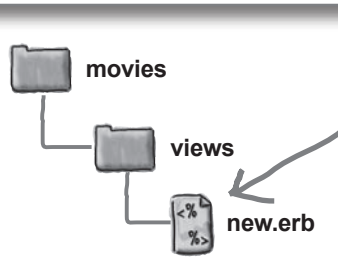
Все поля и метки размещаются внутри тега <form>.

Метки полей

Текстовые поля.

Кнопка отправки данных.

Ссылка на список фильмов.



Как и прежде, шаблон следует сохранить в том каталоге, в котором его сможет найти метод `erb`. Сохраните шаблон в файле с именем *new.erb* в каталоге *views*.

Сохраните приведенный выше файл с именем «new.erb» в каталоге «views».

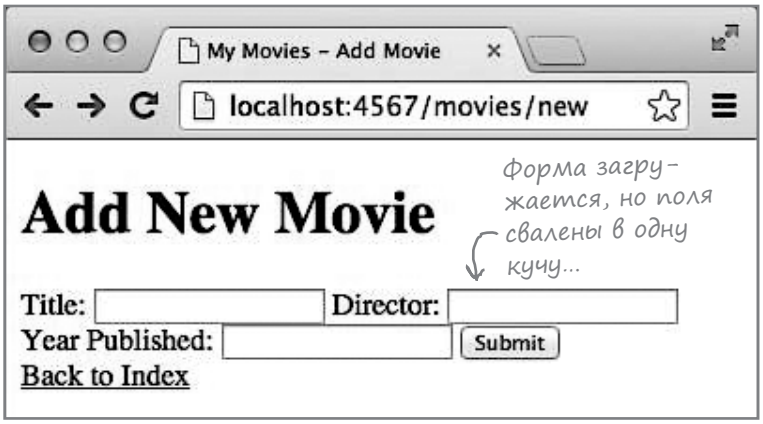
Таблицы HTML

Пора опробовать новую форму в деле! Перезапустите приложение Sinatra и введите URL:

```
http://localhost:4567/movies/new
```

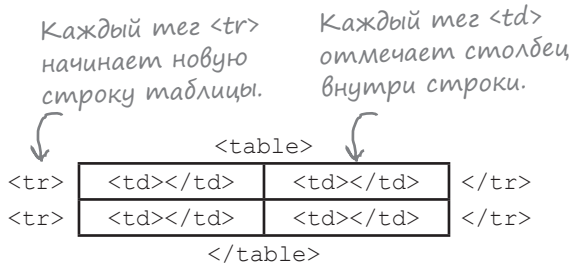
...в адресной строке браузера. (Или просто щелкните на ссылке Add New Movie в конце страницы со списком; ссылка должна работать, так как мы добавили для нее маршрут.)

Форма загружается, но выглядит просто ужасно! Все поля просто свалены в одну кучу. Чтобы исправить ситуацию, мы воспользуемся таблицами.



Таблицы HTML используются для размещения текста, полей форм и т. д. по строкам и столбцам. На практике для создания таблиц наиболее часто используются следующие теги HTML:

- `<table>`: тег вмещает все остальные компоненты таблицы.
- `<tr>`: строка таблицы (от "Table Row.") Каждая строка содержит данные в одном или нескольких столбцах.
- `<td>`: данные таблицы (от "Table Data.") Обычно один элемент `<tr>` содержит несколько встроенных элементов `<td>`, каждый из которых описывает данные одного столбца.



Ниже приведена разметка простой таблицы HTML и ее внешний вид в браузере. (Обычно границы в таблицах остаются скрытыми, но мы добавили их для наглядности.)

```
<table>
  <tr>                                  <-- Начало первой строки.
    <td>Row 1, Column 1</td>            <-- Первый столбец.
    <td>Row 1, Column 2</td>            <-- Второй столбец.
  </tr>
  <tr>                                  <-- Начало второй строки.
    <td>Row 2, Column 1</td>            <-- Первый столбец.
    <td>Row 2, Column 2</td>            <-- Второй столбец.
  </tr>
</table>
```

Результат. (Границы ячеек изображены для большей наглядности.)

Row 1, Column 1	Row 1, Column 2
Row 2, Column 1	Row 2, Column 2

Размещение компонентов формы в таблице HTML

Чтобы форма не выглядела как беспорядочное нагромождение полей, воспользуемся таблицей. Приведите файл *new.erb* к следующему виду:

```
<!DOCTYPE html>
<html>
  ...
  <body>
    <h1>Add New Movie</h1>
    <form>
      <table>
        <tr>
          <td><label for="title">Title:</label></td>
          <td><input type="text" name="title"></td>
        </tr>
        <tr>
          <td><label for="director">Director:</label></td>
          <td><input type="text" name="director"></td>
        </tr>
        <tr>
          <td><label for="year">Year Published:</label></td>
          <td><input type="text" name="year"></td>
        </tr>
        <tr>
          <td><input type="submit"></td>
        </tr>
      </table>
    </form>
    <a href="/movies">Back to Index</a>
  </body>
</html>
```

Таблица HTML вкладывается в форму.

Для каждой пары «метка/поле» создается новая строка.

Метка размещается в первом столбце.

Текстовое поле размещается во втором столбце.

Новая строка для следующего поля.

Новая строка для следующего ввода.

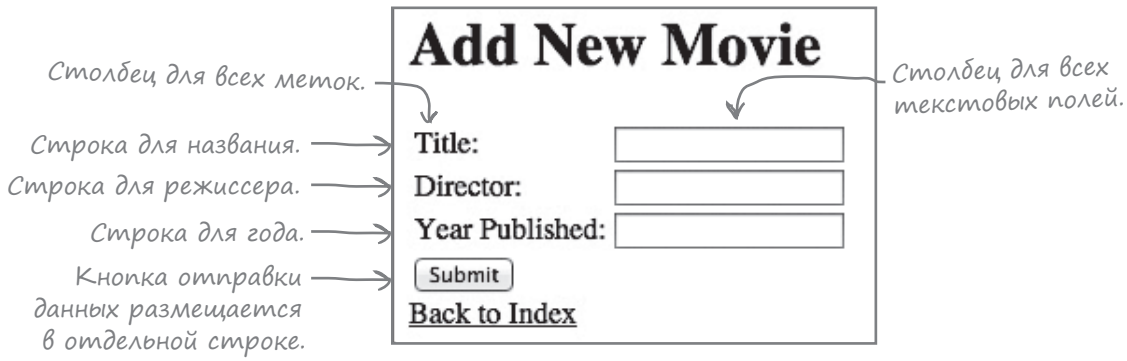
Под кнопку отправки данных выделяется отдельная строка.

Второго столбца нет.

Таблица должна быть завершена до завершения формы.

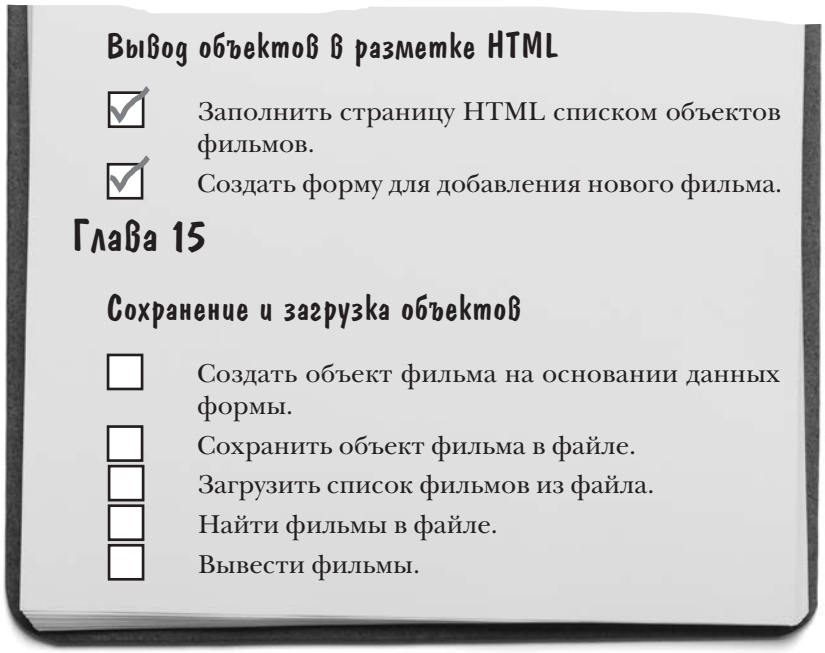
views/new.erb

Попробуйте снова открыть путь `'/movies/new'`. На этот раз поля формы размещаются намного аккуратнее!

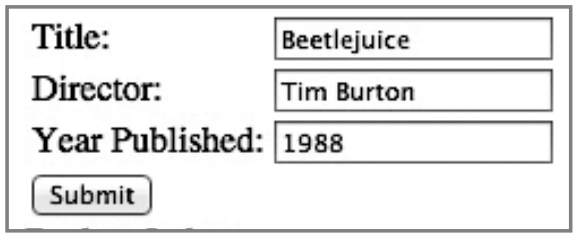


Работа еще не закончена

Маршрут `'/movies/new'` отвечает отображением новой формы HTML. Последняя задача этой главы выполнена!



Но... если заполнить форму и нажать кнопку Submit, ничего не происходит! Мы научили приложение Sinatra реагировать на запросы HTTP GET для загрузки формы, но форма еще не умеет отправлять данные обратно на сервер.



Если нажать эту кнопку, ничего не происходит!

Похоже, предстоит серьезная работа. Не беспокойтесь — эта проблема (вместе с другими!) будет решена в следующей главе!

Ваш инструментарий Ruby

Глава 14 осталась позади, а ваш инструментарий пополнился библиотекой Sinatra и шаблонами ERB.

Sinatra

Библиотека Sinatra создана независимыми разработчиками и распространяется отдельно от Ruby. Такие пакеты загружаются и устанавливаются автоматически с помощью программы RubyGems.

Для определения того, как должны обрабатываться запросы от браузера, в Sinatra используются маршруты.

ERB

ERB — библиотека для выполнения кода Ruby и подстановки результатов в текстовый шаблон.

Теги ERB <% %> обычно используются с условными командами или циклами.

Теги ERB <%= %> используются для подстановки значений в выходные данные.

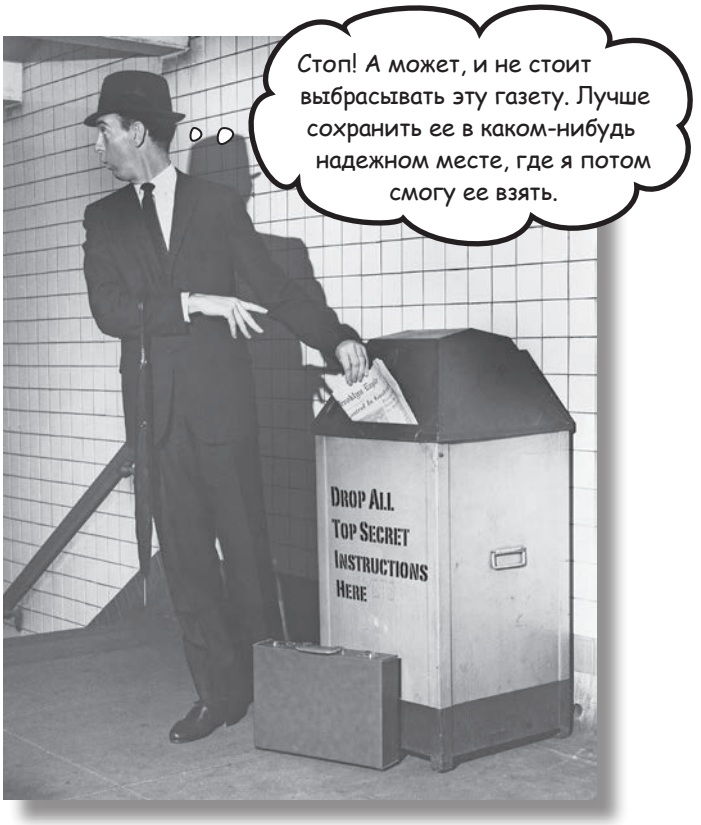
КЛЮЧЕВЫЕ МОМЕНТЫ

- При вводе URL в браузере или щелчке на ссылке браузер отправляет веб-серверу запрос HTTP GET.
- Запросы GET включают путь к ресурсу, который указывает, к какому ресурсу должен обратиться браузер.
- В приложениях Sinatra для создания маршрута для запросов GET вызывается метод `get` со строкой пути к ресурсу и блоком. С этого момента запросы GET к этому пути обрабатываются переданным вами блоком.
- Если блок маршрута возвращает строковое значение, то строка будет отправлена браузеру в ответе.
- Как правило, ответ Sinatra должен определяться в формате HTML (HyperText Markup Language). Язык HTML используется для определения структуры веб-страницы.
- Метод `erb` загружает из подкаталога *views* файл с суффиксом *.erb*. Затем он выполняет встроенный код Ruby и возвращает результат в виде строки.
- Шаблон ERB, обрабатываемый блоком маршрута Sinatra, может обращаться к переменным экземпляра из этого блока.
- Формы HTML предназначены для ввода данных на веб-страницах.
- Таблицы HTML предназначены для размещения данных в табличной структуре со строками и столбцами.

Далее в программе...

Книга почти закончена! Следующая глава — последняя (если не считать приложения). В этой главе мы показали, как вывести в браузере форму для ввода данных пользователем; остается показать, как *сохранить* введенные данные, и снова *загрузить* их позднее.

Сбережения пользователя



Сейчас наше веб-приожение просто выбрасывает данные, введенные пользователем. Мы создали форму, на которой пользователь *вводит* данные. Пользователь ожидает. что данные будут *сохранены*, чтобы их можно было *прочитать* и *вывести* позднее. Но сейчас ничего подобного не происходит! Все введенные данные просто *пропадают*.

В последней главе книги приложение будет подготовлено к сохранению данных, введенных пользователем. Мы покажем, как настроить приложение для получения данных формы, как преобразовать эти данные в объекты Ruby, как сохранить их в файле и как загрузить нужный объект, когда пользователь захочет увидеть его. Готовы? Так доделаем наше приложение!

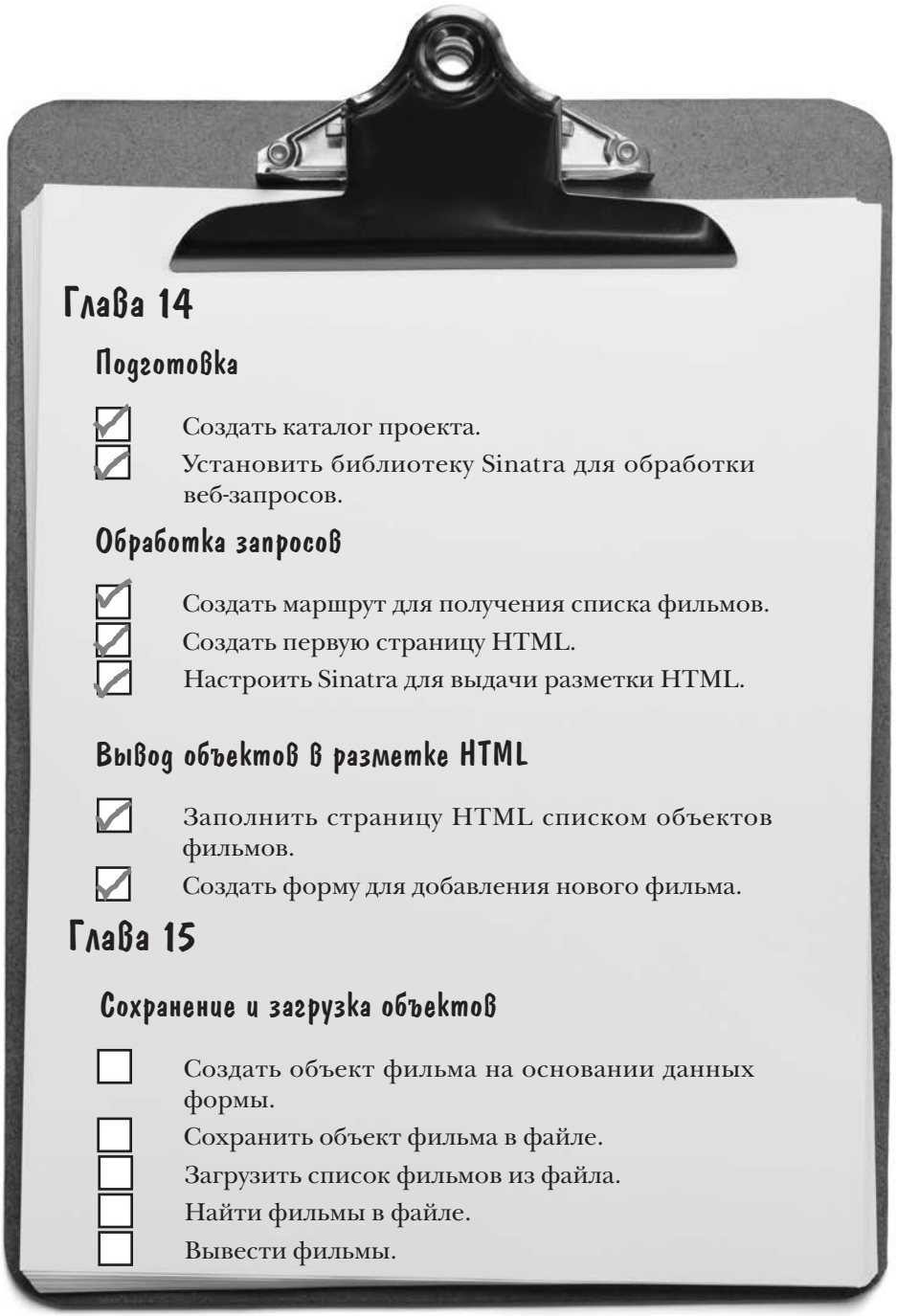
Сохранение и загрузка данных формы

В предыдущей главе вы научились использовать библиотеку Sinatra для выдачи ответов на запросы HTTP GET от браузера. Мы построили класс `Movie` и встроили данные фильмов в страницу HTML.

Вы также узнали, как вывести форму HTML, чтобы пользователь мог ввести данные нового фильма.

Но это все, на что способны запросы HTTP GET. Они не позволят вам отправить форму на сервер — впрочем, даже если бы это было возможно, все равно непонятно, как сохранить данные формы.

В этой главе все эти проблемы будут решены! Вы узнаете, как получить данные с формы и преобразовать их в объекты Ruby для удобства хранения. Также мы научим вас сохранять такие объекты в файле, а потом читать их из файла для вывода. В общем, в этой главе ваши данные наконец-то заработают в полной мере!

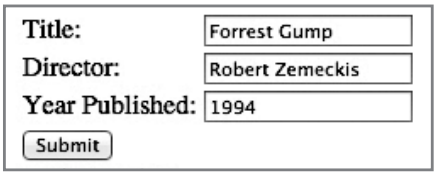


Сохранение и загрузка данных формы (продолжение)

Итак, пользователи будут вводить данные фильмов на форме. Необходим удобный формат для хранения данных, чтобы позднее мы могли прочитать эти данные и вывести их в приложении. Мы преобразуем данные формы в объекты `Movie` и присвоим каждому объекту `Movie` уникальный идентификатор. Затем объект `Movie` будет сохранен в файле.

Позднее мы сможем перебрать содержимое файла и создать набор ссылок, содержащих все идентификаторы фильмов. Когда пользователь щелкает на ссылке, приложение извлекает идентификатор из ссылки, на которой был сделан щелчок, и читает соответствующий объект `Movie`.

1 Пользователь отправляет данные формы.



2 Sinatra получает форму. На основании данных формы создается объект `Movie`, которому присваивается уникальный идентификатор.

```
#<Movie:0x007ffc391ee988
@title="Forrest Gump",
@director="Robert Zemeckis",
@year=1994
@id=3>
```

«В дальнейшем мы будем использовать термин 'фильм №3'».

3 Объект Ruby сохраняется в файле.

Сохраняем фильм №3!

4 Идентификатор объекта используется для генерирования списка ссылок на все фильмы из файла. Пользователь щелкает на одной из ссылок.

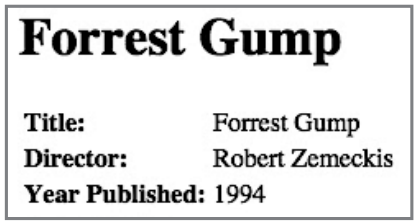
```
<a href="/movies/1">Star Wars</a>
<a href="/movies/2">Ghostbusters</a>
<a href="/movies/3">Forrest Gump</a>
```

«Ага, это интересно» (Щелк!)

5 Идентификатор фильма из ссылки используется для чтения соответствующего объекта `Movie` из файла.

«Покажи мне описание фильма №3!»

6 Прочитанный объект `Movie` преобразуется в разметку HTML и передается браузеру.



Браузер может ПОЛУЧИТЬ форму...

На последних страницах предыдущей главы мы добавили форму в приложение Sinatra. Браузер может выдать запрос GET для пути `'/movies/new'`, а Sinatra отвечает формой HTML. Но когда пользователь нажимает кнопку отправки данных Submit, ничего не происходит!

```
require 'sinatra'
require 'movie'
...
get('/movies/new') do
  erb :new
end
```

app.rb

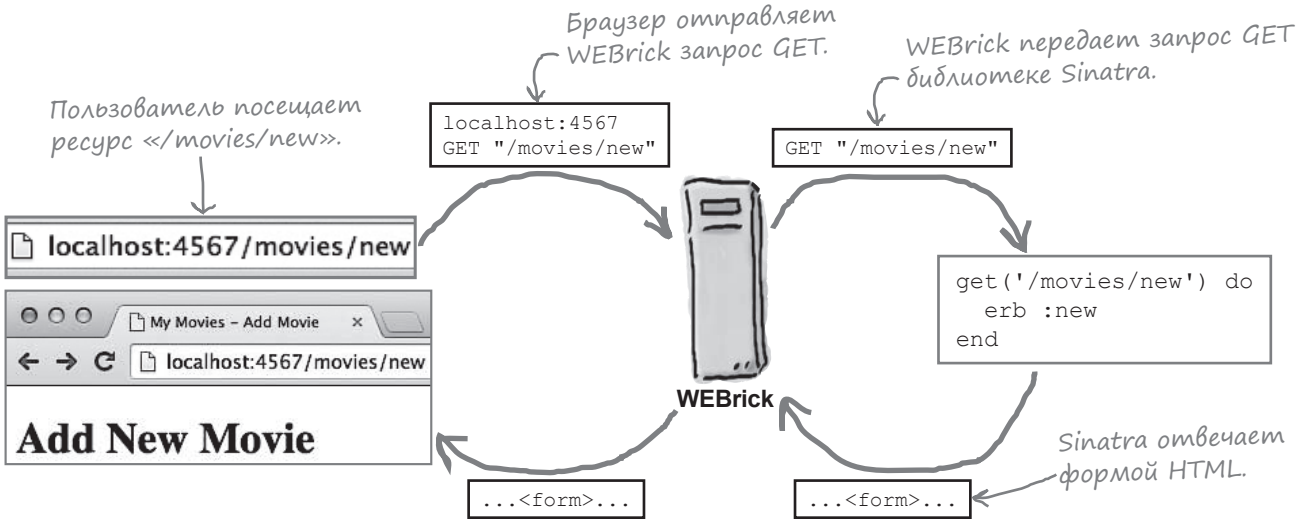
```
<h1>Add New Movie</h1>
<form>
  <table>
    <tr>
      <td><label for="title">Title:</label></td>
      <td><input type="text" name="title"></td>
      ...
```

views/new.erb

Проблема заключается в том, что передача формы HTML требует *двух* запросов к серверу: для *получения* формы и для *отправки* введенных данных серверу.

Мы уже организовали обработку запроса GET для формы:

1. Пользователь запрашивает ресурс `'/movies/new'`, вводя URL-адрес или щелкая на ссылке.
2. Браузер отправляет запрос HTTP GET для ресурса `'/movies/new'` серверу (WEBrick).
3. Сервер передает запрос GET библиотеке Sinatra.
4. Sinatra передает управление блоку для маршрута `get('/movies/new')`.
5. Блок выдает ответ с разметкой HTML формы.

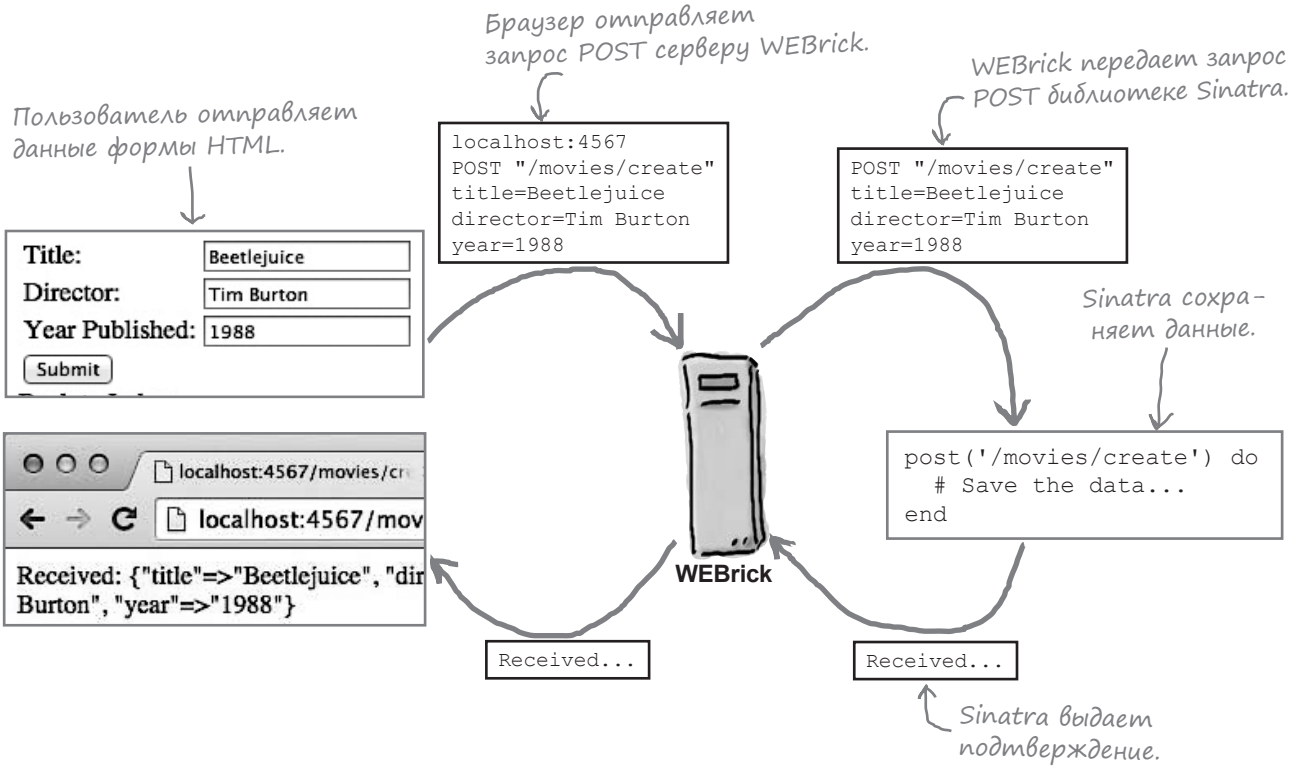


...Но ответ он должен ОТПРАВИТЬ

Теперь нужно настроить Sinatra для обработки запросов HTTP *POST*, чтобы обработать содержимое формы. Если запросы GET *получают данные от* сервера, запросы POST *добавляют данные на сервер.*

Процесс выглядит так:

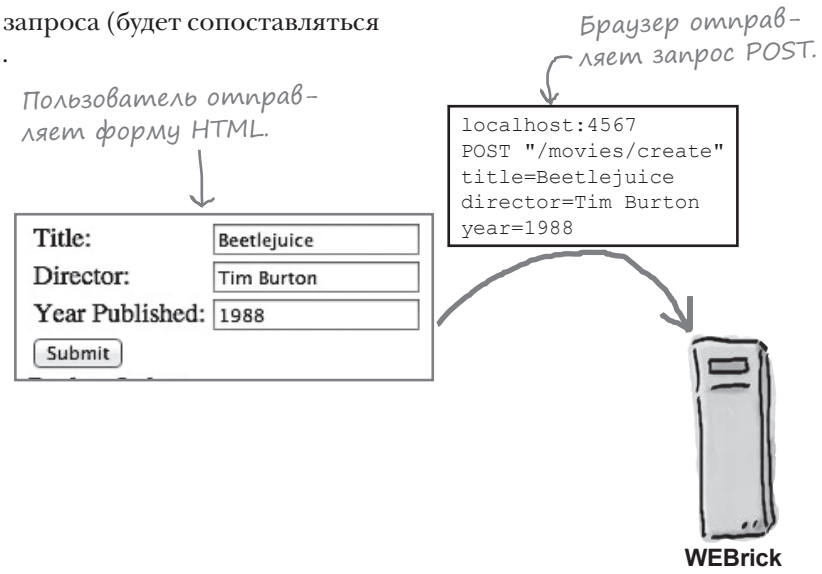
1. Пользователь заполняет форму с данными и нажимает кнопку отправки Submit.
2. Браузер отправляет запрос HTTP POST для ресурса `'/movies/create'` серверу (WEBrick). Запрос включает все данные формы.
3. Сервер передает запрос POST библиотеке Sinatra.
4. Sinatra вызывает блок для маршрута `post('/movies/create')`.
5. Блок получает данные формы из запроса и сохраняет их.
6. Блок отвечает разметкой HTML, которая означает, что данные были получены успешно.



Настройка формы HTML для отправки запроса POST

Первое, что необходимо сделать для обработки данных формы, — обеспечить передачу данных серверу. А для этого нужно настроить форму для отправки запроса POST. Для этого необходимо добавить в тег `<form>` в разметке HTML два атрибута:

- `method`: используемый метод запроса HTTP.
- `action`: путь к ресурсу для запроса (будет сопоставляться с маршрутом Sinatra route).



Изменим разметку в файле *new.erb*, чтобы добавить эти атрибуты к форме. Так как мы собираемся использовать метод POST, атрибуту `method` присваивается значение `"post"`. После этого в атрибуте `action` задается путь к ресурсу `"/movies/create"`.

```
<!DOCTYPE html>
<html>
  ...
  <body>
    <h1>Add New Movie</h1>
    <form method="post" action="/movies/create">
      <table>
        <tr>
          <td><label for="title">Title:</label></td>
          <td><input type="text" name="title"></td>
        </tr>
        ...
```

Чтобы отправить данные формы, следует отправить запрос POST к ресурсу «/movies/create».

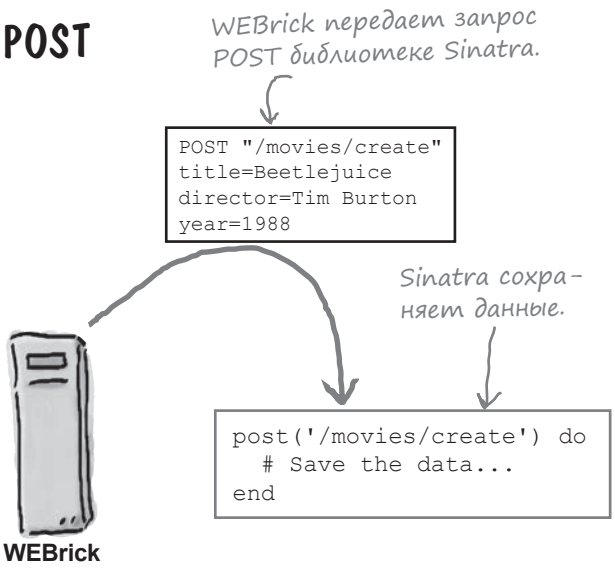
views/new.erb

В результате форма будет настроена для *отправки* запроса POST, но все это не имеет никакого отношения к *обработке* запроса. Этим мы займемся на следующем шаге...

Создание маршрута Sinatra для запроса POST

Итак, мы создали форму HTML, которая отправляет запросы POST для пути `'/movies/create'`. Теперь нужно настроить Sinatra на обработку этих запросов.

Чтобы создать маршрут Sinatra для запросов HTTP GET, следует вызвать метод `get`. А чтобы создать маршрут для запросов POST, следует вызвать — да, угадали! — метод `post`. Он работает точно так же, как и метод `get`; имя метода представляет тип искомого запроса, а сам метод получает строковый аргумент с путем, который должен отслеживаться в запросе. Как и `get`, метод `post` получает блок, который вызывается при получении каждого подходящего запроса.



Вызвав внутри блока маршрута `post` метод `params`, вы получите хеш с данными формы из запроса.

Давайте создадим простой маршрут `post` для простого просмотра данных формы. В файле *app.rb* метод `post` будет вызываться с путем к ресурсу, соответствующим тому, который был задан в форме: `'/movies/create'`. Блок просто возвращает браузеру строку со значением `params.inspect`.

```
require 'sinatra'
require 'movie'

get('/movies') do
  @movies = []
  ...
  erb :index
end

get('/movies/new') do
  erb :new
end

post('/movies/create') do
  "Received: #{params.inspect}"
end
```

Обрабатывает запросы POST для пути '/movies/create'.

Браузеру возвращается строка с данными формы.

app.rb

Создание маршрута Sinatra для запроса POST (продолжение)

Мы подготовили форму HTML для отправки запроса POST...

```
<form method="post" action="/movies/create">
  ...
</form>
```

И создали соответствующий маршрут в приложении Sinatra...

```
post('/movies/create') do
  "Received: #{params.inspect}"
end
```

А теперь опробуем новый маршрут на практике. Перезапустите приложение и загрузите страницу формы в браузере. Заполните форму любыми данными и нажмите кнопку отправки данных Submit.

Заполните форму и нажмите кнопку Submit.

Форма отправляет Sinatra запрос POST, на который Sinatra отвечает простой строкой, представляющей содержимое хеша `params`.

Хеш «params» с данными формы.

Убедившись в том, что данные возвращаются правильно, мы создадим объект `Movie` на основании содержимого хеша `params`. Для этого нужно изменить блок маршрута из *app.rb*, чтобы переменной экземпляра присваивался новый объект `Movie`. Затем каждое значение из хеша присваивается соответствующему атрибуту `Movie`.

```
require 'sinatra'
require 'movie'
...
post('/movies/create') do
  @movie = Movie.new
  @movie.title = params['title']
  @movie.director = params['director']
  @movie.year = params['year']
end
```

Создаем новый экземпляр Movie.

Содержимое полей формы присваивается атрибутам объекта.

app.rb

Создание маршрута Sinatra для запроса POST (продолжение)

Мы написали маршрут Sinatra, который получает данные формы и использует их для заполнения атрибутов нового объекта `Movie`. Еще одна задача выполнена!

Впрочем, пока объект `Movie` нигде не сохраняется. Решением этой проблемы мы займемся на следующем шаге.

Сохранение и загрузка объектов

- [x] Создать объект фильма на основании данных формы.
- [ ] Сохранить объект фильма в файле.
- [ ] Загрузить список фильмов из файла.
- [ ] Найти фильмы в файле.
- [ ] Вывести фильмы.

Ниже приведены файлы приложения Sinatra и шаблона ERB. Заполните пропуски в обоих файлах, чтобы браузер отображал форму по запросу URL `http://localhost:4567/form` и выводил показанный результат при отправке данных формы.

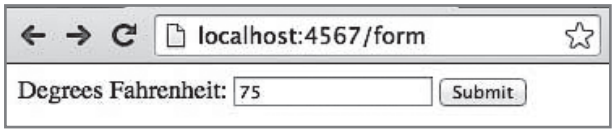
```
require 'sinatra'

get(_______) do
  erb _____
end

____('/convert') do
  fahrenheit = _______['temperature'].to_f
  celsius = (fahrenheit - 32) / 1.8
  format("%0.1f degrees Fahrenheit is %0.1f degrees Celsius.", fahrenheit, celsius)
end
```

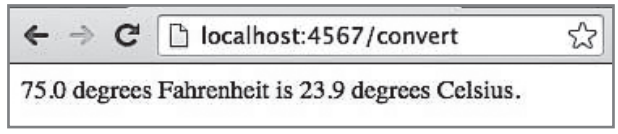
app.rb

```
<!DOCTYPE html>
<html>
  <body>
    <form _______="post" action="_________">
      <label for="temperature">Degrees Fahrenheit:</label>
      <input type="text" name="____________">
      <input type="submit">
    </form>
  </body>
</html>
```

views/form.erb

Ответы:





Ниже приведены файлы приложения Sinatra и шаблона ERB. Заполните пропуски в обоих файлах, чтобы браузер отображал форму по запросу URL `http://localhost:4567/form` и выводил показанный результат при отправке данных формы.

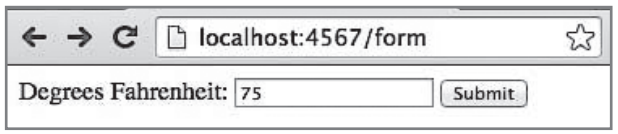
```
require 'sinatra'

get('/form') do
  erb :form
end

post('/convert') do
  fahrenheit = params['temperature'].to_f
  celsius = (fahrenheit - 32) / 1.8
  format("%0.1f degrees Fahrenheit is %0.1f degrees Celsius.", fahrenheit, celsius)
end
```

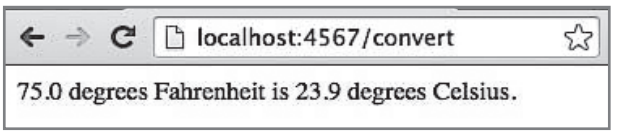
app.rb

```
<!DOCTYPE html>
<html>
  <body>
    <form method="post" action="/convert">
      <label for="temperature">Degrees Fahrenheit:</label>
      <input type="text" name="temperature">
      <input type="submit">
    </form>
  </body>
</html>
```

views/form.erb

Ответы:





Преобразование объектов в строки и YAML

Мы добавили код преобразования данных формы в объект `Movie`:

```
post('/movies/create') do
  @movie = Movie.new
  @movie.title = params['title']
  @movie.director = params['director']
  @movie.year = params['year']
end
```

Но этот объект пропадает сразу же после создания. Его необходимо где-то сохранить!

В этом нам поможет библиотека `YAML`, которая является частью стандартной библиотеки Ruby. YAML (рекурсивное сокращение от «YAML Ain't Markup Language», то есть «YAML не является языком разметки») — стандарт представления объектов и других данных в строковой форме. Такие строки можно сохранять в файлах, а потом преобразовывать их в объекты. За дополнительной информацией о YAML обращайтесь по адресу:

http://yaml.org

Но прежде чем заниматься сохранением YAML в файле, попробуем преобразовать объекты в строки, чтобы вы поняли, как выглядит формат YAML. Не беспокойтесь, *знать* YAML не обязательно. Библиотека YAML преобразует объекты Ruby в формат YAML и обратно за вас!

Модуль `YAML` содержит метод `dump`, который преобразует почти любой объект Ruby в строковое представление. Следующий фрагмент создает объект `Movie`, а затем преобразует его в строку:

```
require 'movie'
require 'yaml'

movie = Movie.new
movie.title = "Fight Club"
movie.director = "David Fincher"
movie.year = 1999

puts YAML.dump(movie)
```

Загружаем класс Movie.

Загружаем модуль YAML.

Создаем объект Movie.

Класс объекта.

```
--- !ruby/object:Movie
title: Fight Club
director: David Fincher
year: 1999
```

Атрибуты объекта и их значения.

Модуль `YAML` также содержит метод `load`, который получает строку с данными YAML и преобразует ее в объект. Этот метод пригодится нам позднее для загрузки сохраненных объектов.

Этот код преобразует объект `Movie` в строку YAML, а затем преобразует строку обратно в объект; при этом все значения атрибутов остаются неизменными:

```
movie_yaml = YAML.dump(movie)
copy = YAML.load(movie_yaml)
puts copy.title, copy.director, copy.year
```

Строка YAML сохраняется в переменной.

Строка YAML преобразуется обратно в объект.

```
Fight Club
David Fincher
1999
```

Все атрибуты старого объекта успешно восстановлены!

YAML::Store и сохранение объектов в файле

Преобразование объектов в строки и обратно — всего лишь половина решения; нам по-прежнему нужна возможность сохранения данных для использования в будущем. Библиотека YAML включает класс с именем `YAML::Store`, который умеет сохранять объекты Ruby на диске и загружать их позднее.

Код добавления объектов в экземпляр `YAML::Store` и их последующей загрузки очень похож на код работы с хешем. Вы указываете ключ и значение, которое должно ассоциироваться с этим ключом. Позднее вы обращаетесь к ключу и получаете исходное значение.

Конечно, главное различие заключается в том, что `YAML::Store` сохраняет ключи и значения в файле. Вы можете перезапустить свою программу или даже прочитать содержимое файла из совершенно другой программы — ключи и значения останутся на своих местах.

Чтобы использовать `YAML::Store` в коде Ruby, необходимо сначала загрузить библиотеку:

```
require 'yaml/store'
```

После этого можно создать экземпляр `YAML::Store`. Новый метод получает аргумент с именем файла, в который объект должен записывать и/или читать данные. (Один объект `Store` может использоваться как для чтения, так *и* для записи в файл.)

```
store = YAML::Store.new('my_file.yml')
```

Создаем экземпляр YAML::Store для записи в файл с именем my_file.yml.

Прежде чем добавлять или читать объекты из хранилища, необходимо вызвать метод `transaction`. (Если это не будет сделано, класс `YAML::Store` инициирует ошибку.) Почему это необходимо? Если одна программа запишет данные в файл, пока другая программа читает из этого файла, то прочитанные данные могут быть повреждены. Метод `transaction` защищает от этой опасности.

Итак, чтобы записать данные в файл, мы вызываем метод `transaction` экземпляра `YAML::Store` и передаем ему блок. В блоке нужное значение ассоциируется с ключом, как и при работе с хешем.

```
store.transaction do
  store["my key"] = "my value"
  store["key two"] = "value two"
end
```

Запрещает запись в файл из других программ, пока не произойдет выход из блока.

Присваиваем значения ключам. Эти данные будут сохранены в файле!

Процесс чтения значений выглядит так же: вызываем `transaction`, затем в блоке присваиваем нужное значение.

```
store.transaction do
  puts store["my key"]
end
```

Также необходимо вызвать «transaction» перед чтением.

Читаем значение из файла.

```
my value
```

YAML::Store и сохранение описаний фильмов в файле

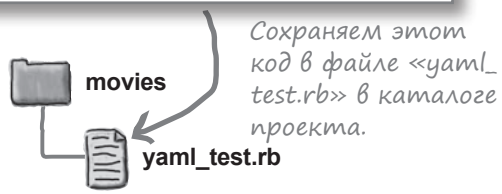
На нескольких ближайших страницах мы создадим класс, который будет управлять экземпляром `YAML::Store`, записывать в него объекты `Movie` и читать их оттуда. А пока мы разбираемся в этой теме, напишем простой сценарий, использующий хранилище `YAML::Store` для сохранения объектов фильмов в файле.

В начале сценария необходимо выполнить кое-какую подготовку... Так как классы `Movie` и `YAML::Store` становятся доступными только после того, как они будут загружены в программе, все начинается с вызовов `require` для `'movie'` и `'yaml/store'`. Затем создается экземпляр `YAML::Store`, который читает и записывает данные в файл с именем *test.yml*. Мы создаем пару экземпляров `Movie` и задаем значения всех их атрибутов.

А теперь самое интересное: мы вызываем метод `transaction` экземпляра `YAML::Store` и передаем ему блок. В этом блоке выполняется пара операций:

- Объекты фильмов ассоциируются с ключами в хранилище.
- Мы читаем значение, ассоциированное с одним из сохраненных ранее ключей, и выводим его.

```
require 'movie'                          # Загружаем класс Movie.
require 'yaml/store'                     # Загружаем класс YAML::Store.

store = YAML::Store.new('test.yml')      # Создаем хранилище для записи
                                         # объектов в файл с именем test.yml.
first_movie = Movie.new                  # Создаем объект с данными фильма.
first_movie.title = "Spirited Away"
first_movie.director = "Hayao Miyazaki"
first_movie.year = 2001

second_movie = Movie.new                 # Создаем второй объект.
second_movie.title = "Inception"
second_movie.director = "Christopher Nolan"
second_movie.year = 2010

store.transaction do                     # Запрещаем другим программам запись в файл.
  store["Spirited Away"] = first_movie   # Сохраняем два объекта.
  store["Inception"] = second_movie

  p store["Inception"]                   # Выводим одно из значений из хранилища.
end
```



А теперь опробуем этот код в деле! Сохраните сценарий с именем *yaml_test.rb*. Файл должен храниться в каталоге проекта Sinatra, рядом с каталогом *lib*, чтобы мы могли загрузить файл *movie.rb* при его запуске.

YAML::Store и сохранение описаний фильмов в файле (продолжение)

В терминальном окне перейдите в каталог проекта и запустите сценарий командой:

```
ruby -I lib yaml_test.rb
```

Сценарий создает файл *test.yml* и сохраняет в нем два объекта фильмов. Затем он обращается к одному из объектов и выводит для него отладочную строку.

Переходим в каталог проекта.

Запускаем сценарий.

Один из хранимых объектов.

```
File  Edit  Window  Help
$ cd movies
$ ruby -I lib yaml_test.rb
#<Movie:0x007ffc391ee988 @title="Inception",
  @director="Christopher Nolan", @year=2010>
```

Открыв файл *test.yml* в текстовом редакторе, вы увидите объекты `Movie` в формате YAML, а также ключи, с которыми эти объекты были сохранены.

При выполнении yaml_test.rb создается этот файл.

Ключ, под которым был сохранен первый объект.

Первый объект Movie в формате YAML.

Ключ, под которым был сохранен второй объект.

Второй объект Movie.

```
---
Spirited Away: !ruby/object:Movie
  title: Spirited Away
  director: Hayao Miyazaki
  year: 2001
Inception: !ruby/object:Movie
  title: Inception
  director: Christopher Nolan
  year: 2010
```

test.yml

Развлечения с магнитами

На холодильнике разложена программа на языке Ruby. Сможете ли вы переставить фрагменты так, чтобы получить работоспособную программу, которая бы выдавала приведенный ниже результат? Программа должна создавать файл с именем *books.yml*, содержимое которого приведено ниже.

```
"Head First Ruby"
store.transaction
end
book.pages =
store
YAML::Store.new
end
do
450
book.title =
book = Book.new
book
'yaml/store'
store =
class Book
require
=
attr_accessor :title, :pages
["HFRB"]
('books.yml')
```

Результат:

```
---
HFRB: !ruby/object:Book
  title: Head First Ruby
  pages: 450
```

books.yml

Развлечения с магнитами. Решение

На холодильнике разложена программа на языке Ruby. Сможете ли вы переставить фрагменты так, чтобы получить работоспособную программу, которая бы выдавала приведенный ниже результат? Программа должна создавать файл с именем *books.yml*, содержимое которого приведено ниже.

```
require 'yaml/store'

class Book
    attr_accessor :title, :pages
end

book = Book.new
book.title = "Head First Ruby"
book.pages = 450

store = YAML::Store.new ('books.yml')

store.transaction do
    store ["HFRB"] = book
end
```

Результат:

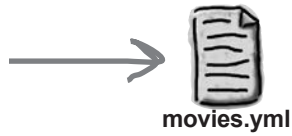
```
---
HFRB: !ruby/object:Book
  title: Head First Ruby
  pages: 450
```

books.yml

Поиск объектов Movie в YAML::Store

Почти все готово для сохранения объектов `Movie` в `YAML::Store`!

```
#<Movie:0x007ffc391ee988
@title="Forrest Gump",
@director="Robert Zemeckis",
@year=1994>
```



Но сначала стоит задать себе вопрос: как потом мы будем читать эти объекты?

Как вы уже знаете, мы собираемся сгенерировать список ссылок на все фильмы в базе данных. Щелкая на ссылке, пользователь отправляет запрос приложению для получения информации об этом фильме. Итак, необходимо иметь возможность найти объект `Movie` в хранилище `YAML::Store` исключительно по информации, содержащейся в ссылке.

```
<a href="/movies/????">Star Wars</a>
<a href="/movies/????">Ghostbusters</a>
<a href="/movies/????">Forrest Gump</a>
```

«Ага, это интересно!» (Щелк!)

Найти фильм с идентификатором ????.

Вывод данных в формате HTML.

Какое же значение использовать в качестве идентификатора? Вроде бы атрибут `title` является очевидным кандидатом. В предыдущем сценарии мы использовали названия в качестве ключей `YAML::Store`:

```
store["Spirited Away"] = first_movie
store["Inception"] = second_movie
```

Сохраняем два фильма.

Но с названиями фильмов не все так просто. Во-первых, они часто содержат пробелы, а в URL этот символ не разрешен. Можно обойти проблему, используя кодировку символов, но такое решение выглядит некрасиво:

```
<a href="/movies/Forrest%20Gump">Forrest Gump</a>
```

Последовательность символов, представляющая пробел.

Во-вторых, названия фильмов не являются уникальными идентификаторами. Да, в 1997 году вышел фильм *«Титаник»*, но и в 1953 году вышел фильм, который *тоже* назывался *«Титаник»*!

Для идентификации фильмов придется поискать что-то другое...

Числовые идентификаторы

По этим и другим подобным причинам в большинстве современных веб-приложений для идентификации записей в базах данных используются простые числовые значения. С ними проще работать, и они более эффективны. Мы тоже будем использовать числовые идентификаторы в качестве ключей `YAML::Store`.

Числовой идентификатор

Числовой идентификатор

```
---
1: !ruby/object:Movie
   title: Star Wars
   director: George Lucas
   year: '1977'
   id: 1
2: !ruby/object:Movie
   title: Ghostbusters
   director: Ivan Reitman
   year: '1984'
   id: 2
```

Чтобы упростить и ускорить связывание объекта `Movie` с ключом `YAML::Store`, откройте файл *movie.rb* в каталоге *lib* и добавьте атрибут `id` в класс `Movie`:

```
class Movie
  attr_accessor :title, :director, :year, :id
end
```

Добавляем атрибут «id».

lib/movie.rb

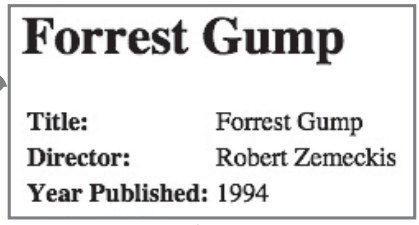
Назначение числового идентификатора упростит такие операции, как последующее генерирование URL-адресов.

```
<a href="/movies/1">Star Wars</a>
<a href="/movies/2">Ghostbusters</a>
<a href="/movies/3">Forrest Gump</a>
```

«Ага, это интересно!» (Щелк!)

movies.yml

Найти фильм с атрибутом «id», равным 3.



Вывод данных в формате HTML.

Поиск следующего доступного идентификатора

Предположим, мы создали новый объект `Movie` с данными, введенными на форме, и идентификатора у него еще нет. Какой ключ `YAML::Store` следует ему присвоить? Необходимо перебрать существующие ключи и найти свободный.

Объекту необходим уникальный идентификатор.

```
#<Movie:0x007ffc391ee988
@title="Beetlejuice",
@director="Tim Burton",
@year=1988
@id=nil>
```

Значение 1 занято...

Значение 2 занято...

```
---
1: !ruby/object:Movie
  title: Jaws
  director: Steven Spielberg
  year: '1975'
  id: 1
2: !ruby/object:Movie
  title: Goodfellas
  director: Martin Scorsese
  year: '1990'
  id: 2
```

Значение 3 свободно; используем 3.

Для обработки ключей можно воспользоваться методом экземпляра `roots` из `YAML::Store`. Метод `roots` возвращает все ключи хранилища в виде массива:

```
require 'yaml/store'

store = YAML::Store.new('numeric_keys.yml')
store.transaction do
  store[1] = 'Jaws'
  store[2] = 'Goodfellas'
  p store.roots
end
```

Присваиваем пару числовых ключей.

Получаем ключи в виде массива.

```
[1, 2]
```

Ключи YAML::Store в виде массива.

Теперь нужно найти в этом массиве наибольшее число. У массивов имеется метод `max`, который возвращает наибольшее значение среди элементов массива:

```
p [1, 2, 9, 5].max
```

```
9
```

Наибольшее значение в массиве.

Итак, мы просто вызываем `roots` для получения массива ключей, вызываем `max` для получения наибольшего числа и увеличиваем наибольшее число на 1 для получения нового идентификатора.

```
require 'yaml/store'

store = YAML::Store.new('numeric_keys.yml')
store.transaction do
  p store.roots.max + 1
end
```

Находим наибольший ключ в хранилище и увеличиваем его на 1.

```
3
```

Поиск следующего доступного идентификатора (продолжение)

Но тут кроется одна загвоздка. Что, если мы работаем с пустым хранилищем `YAML::Store`? (А хранилище *будет* пустым до того, как в нем будет сохранен первый фильм...)

В этом случае метод `roots` вернет пустой массив. Вызов `max` для пустого массива возвращает `nil`. А при попытке прибавить 1 к `nil` происходит ошибка!

```
require 'yaml/store'

store = YAML::Store.new('empty_store.yml')
store.transaction do
  p store.roots
  p store.roots.max
  p store.roots.max + 1
end
```

В этом файле еще нет ни одного ключа!

Возвращает пустой массив!

Возвращает «nil»!

Приводит к ошибке!

```
[]
nil
undefined method `+' for nil:NilClass
```

Чтобы предотвратить ошибку, необходимо проверить, не возвращает ли `max` значение `nil`, и в таком случае использовать значение `0`. А это можно легко и быстро сделать при помощи логического оператора `||` (ИЛИ). Этот оператор уже встречался в главе 9: если значение слева от `||` равно `false` или `nil` оно игнорируется и вместо него используется выражение в правой части. Итак, мы можем просто написать:

```
store.roots.max || 0
```

...и получим `0` при отсутствии ключей в хранилище или же наибольший существующий ключ, если ключи *существуют*.

```
require 'yaml/store'

store = YAML::Store.new('empty_store.yml')
store.transaction do
  highest_id = store.roots.max || 0
  p highest_id + 1
end
```

Файл все еще пуст.

Если выражение «store.roots.max» возвращает «nil», вместо него используется 0.

```
1
```

Класс для управления YAML::Store

Итак, мы написали безопасный код присваивания идентификаторов для новых объектов фильмов в `YAML::Store`. Пожалуй, добавлять этот код в приложение Sinatra не стоит; оно и без того достаточно загромождено. Лучше напишем отдельный класс для сохранения объектов в `YAML::Store`.

Значение 1 занято...

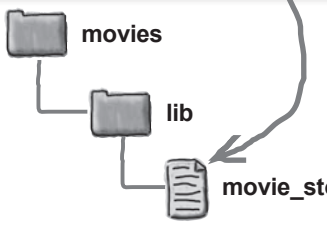
Значение 2 занято...

```
---
1: !ruby/object:Movie
  title: Jaws
  director: Steven Spielberg
  year: '1975'
  id: 1
2: !ruby/object:Movie
  title: Goodfellas
  director: Martin Scorsese
  year: '1990'
  id: 2
```

Значение 3 свободно; используем 3.

Создадим файл с именем *movie_store.rb* в подкаталоге *lib*. В этом файле будет определен класс с именем `MovieStore`. Его метод `initialize` получает имя файла и создает экземпляр `YAML::Store` для записи в этот файл. Затем будет добавлен метод `save`, получающий объект `Movie`. Если объекту `Movie` еще не присвоен идентификатор, `save` находит следующий свободный идентификатор и присваивает его объекту `Movie`. После того как `Movie` получит идентификатор, метод `save` записывает его в хранилище с ключом, совпадающим с идентификатором.

```
require 'yaml/store'                          ← Загружаем класс YAML::Store.

class MovieStore

  def initialize(file_name)                   ← Создаем хранилище для чтения/
    @store = YAML::Store.new(file_name)         записи файла с заданным именем.
  end

  def save(movie)                             ← Сохраняет объект Movie в хранилище.
    @store.transaction do                     ← Вызов transaction необходим...
      unless movie.id                         ← Если фильму еще не присвоен идентификатор...
        highest_id = @store.roots.max || 0    ← ...находим наибольший ключ...
        movie.id = highest_id + 1             ← ...и увеличиваем его на 1.
      end
      @store[movie.id] = movie                ← Фильм сохраняется
    end                                         с ключом, совпадающим
  end                                           с идентификатором.

end
```



Сохраните код в файле «movie_store.rb» в подкаталоге «lib».

Использование класса MovieStore в приложении Sinatra

Нам пришлось создать специальный класс `MovieStore` для хранения кода, работающего с `YAML::Store`. Но теперь мы можем извлечь из него пользу: работать с `MovieStore` в приложении Sinatra стало проще простого!

В начало файла *app.rb* необходимо включить вызов `require 'movie_store'` для загрузки нового класса. Мы также создадим новый экземпляр `MovieStore` и передадим строку `'movies.yml'` с именем файла, который должен использоваться для чтения и записи данных.

Наш блок для маршрута `post('/movies/create')` уже подготовлен для создания объекта `Movie` на основании данных, введенных на форме. Следовательно, остается только передать объект `Movie` методу `save` экземпляра `MovieStore`.

```
require 'sinatra'
require 'movie'
require 'movie_store'          ← Загружаем класс MovieStore.

store = MovieStore.new('movies.yml')   ← Создаем объект MovieStore для обновления файла movies.yml.

...

get('/movies/new') do
  erb :new
end

post('/movies/create') do      ← Сюда поступают отправленные данные формы.
  @movie = Movie.new
  @movie.title = params['title']
  @movie.director = params['director']   ← Содержимое полей формы присваивается атрибутам объекта.
  @movie.year = params['year']
  store.save(@movie)           ← Сохраняем объект!
  redirect '/movies/new'       ← Отобразить новую, пустую форму.
end
```

app.rb

После сохранения фильма необходимо что-то вывести в браузере, поэтому мы вызываем метод Sinatra, который нам еще не встречался: `redirect`. Метод `redirect` получает строку с путем к ресурсу (или весь URL-адрес, если потребуется) и отправляет ответ браузеру с приказанием загрузить этот ресурс. Мы используем путь `'/movies/new'` для того, чтобы браузер снова загрузил форму нового фильма.

Тестирование MovieStore

Вы уже знаете, что делать: перейдите в каталог проекта в терминальном окне и перезапустите приложение. Откройте страницу для добавления нового фильма:

```
http://localhost:4567/movies/new
```

Введите фильм на форме и нажмите кнопку отправки данных Submit. В браузере вы увидите лишь то, что форма очищается (потому что ответ redirect приказывает браузеру снова загрузить страницу формы)...

Введите на форме данные нового фильма и нажмите кнопку отправки данных Submit.

Но заглянув в каталог проекта, вы найдете в нем новый файл *movies.yml*. А если вы откроете этот файл, то увидите в нем введенные данные в формате YAML!

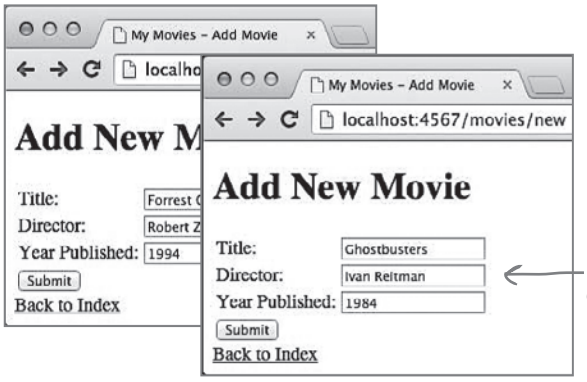
Если продолжить ввод фильмов, они тоже будут добавлены в файл YAML. Вы также увидите, что значение идентификатора увеличивается для каждого нового фильма.

Данные фильмов, введенных на форме, добавляются в файл!

```
---
1: !ruby/object:Movie
   title: Star Wars
   director: George Lucas
   year: '1977'
   id: 1
2: !ruby/object:Movie
   title: Ghostbusters
   director: Ivan Reitman
   year: '1984'
   id: 2
3: !ruby/object:Movie
   title: Forrest Gump
   director: Robert Zemeckis
   year: '1994'
   id: 3
```

movies.yml

Идентификатор будет увеличиваться с каждым новым фильмом.



Вводим другие фильмы...

Загрузка всех фильмов из MovieStore

Пришлось немного потрудиться, но мы наконец-то можем сохранять объекты фильмов в хранилище `YAML::Store`. Еще одна задача выполнена!

На следующем шаге требуется загрузить все фильмы из файла и вывести их в списке.

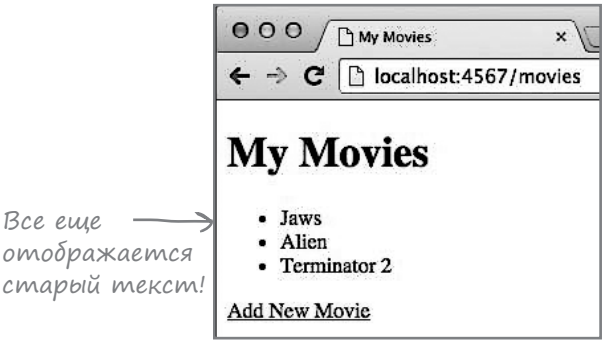
Сохранение и загрузка объектов

- [x] Создать объект фильма на основании данных формы.
- [x] Сохранить объект фильма в файле.
- [ ] Загрузить список фильмов из файла.
- [ ] Найти фильмы в файле.
- [ ] Вывести фильмы.

Если посетить страницу со списком прямо сейчас:

```
http://localhost:4567/movies
```

...вы увидите заготовки, которые были созданы ранее (вместо сохраненных фильмов). Эти данные все еще жестко запрограммированы в блоке маршрута `get('/movies')` в *app.rb*.



Все еще отображается старый текст!

Если мы хотим, чтобы вместо него в списке отображались сохраненные названия фильмов...

Необходимо добавить в `MovieStore` еще один метод, который возвращает массив всех значений из `YAML::Store`. Но если метод `roots` возвращает массив всех *ключей*, удобного метода для получения всех значений не существует...

Но это неважно! Так как метод `roots` возвращает массив, мы можем просто вызвать для него метод `map`. Как вы, возможно, помните из главы 6, метод `map` передает каждый элемент массива блоку и возвращает новый массив со всеми возвращаемыми значениями блока. Следовательно, мы можем просто передать `map` блок, который возвращает значение, ассоциированное с каждым ключом в хранилище.

Следующий простой сценарий демонстрирует эту возможность для уже знакомого вам хранилища *numeric_keys.yml*:

```
require 'yaml/store'

store = YAML::Store.new('numeric_keys.yml')
store.transaction do
  store[1] = 'Jaws'
  store[2] = 'Goodfellas'
  p store.roots
  p store.roots.map { |key| store[key] }
end
```

Выводим все ключи.

Создаем новый массив со значениями всех ключей.

```
[1, 2]
["Jaws", "Goodfellas"]
```

Загрузка всех фильмов из MovieStore (продолжение)

А теперь применим ту же идею в классе `MovieStore`. Мы создадим новый метод с именем `all`, который возвращает каждый объект `Movie` в хранилище. Метод `map` будет вызываться для массива, возвращаемого `roots`, для получения объектов `Movie`, ассоциированных со всеми ключами.

```
require 'yaml/store'

class MovieStore

  def initialize(file_name)
    @store = YAML::Store.new(file_name)
  end

  def all
    @store.transaction do
      @store.roots.map { |id| @store[id] }
    end
  end

  ...

end
```

Читает все фильмы из хранилища.

Для обращения к хранилищу необходим вызов transaction.

Создаем массив со значениями всех ключей.

lib/movie_store.rb

Часто Задаваемые Вопросы

В: Как метод `all` возвращает массив объектов `Movie`?

О: Метод `map` возвращает массив объектов `Movie`. *Блок* `transaction` возвращает этот массив. *Метод* `transaction` возвращает то, что возвращает его *блок*. А возвращаемое значение метода `transaction` становится возвращаемым значением метода `all`.

Вот как выглядит более подробный эквивалент приведенного выше кода:

```
def all
  transaction_return_value = @store.transaction do
    block_return_value = @store.roots.map { |id| @store[id] }
    block_return_value
  end
  transaction_return_value
end
```

block_return_value возвращается методом «transaction».

Возвращает массив Movies.

Массив объектов Movie возвращается из блока.

Возвращаемое значение «transaction» становится возвращаемым значением метода «all».

Честно говоря, мы считаем, что анализировать все это шаг за шагом слишком хлопотно. Мы предпочитаем игнорировать вызов `transaction` — делать вид, что его вообще нет. Обычно это ни на что не влияет — код работает точно так же!

Загрузка всех фильмов в приложении Sinatra

После добавления метода загрузки всех фильмов из хранилища `MovieStore` приложение Sinatra почти закончено.

Ранее в приложении использовался фиксированный набор фильмов, хранившихся в переменной экземпляра `@movies`, для шаблона *index.erb*. Все, что потребуется — заменить этот набор объектов `Movie` вызовом `store.all`.

```
get('/movies') do
  @movies = []
  @movies[0] = Movie.new
  @movies[0].title = "Jaws"
  @movies[1] = Movie.new
  @movies[1].title = "Alien"
  @movies[2] = Movie.new
  @movies[2].title = "Terminator 2"
  erb :index
end
```

Этот код заменяется...

```
require 'sinatra'
require 'movie'
require 'movie_store'

store = MovieStore.new('movies.yml')

get('/movies') do
  @movies = store.all
  erb :index
end

get('/movies/new') do
  erb :new
end

post('/movies/create') do
  @movie = Movie.new
  @movie.title = params['title']
  @movie.director = params['director']
  @movie.year = params['year']
  store.save(@movie)
  redirect '/movies/new'
end
```

...этим!

app.rb

Объекты `Movie`, хранящиеся в `YAML::Store`, загружаются в переменную экземпляра `@movies`. Они готовы для построения списка HTML!

После внесения этих изменений перезапустите приложение в терминальном окне. Старые временные названия фильмов будут заменены названиями из файла *movies.yml*!

Названия фильмов загружаются из файла movies.yml!

Построение ссылок HTML на фильмы

Получить список всех фильмов в файле было относительно просто. Теперь необходимо сделать следующий шаг — построить ссылки на отдельные фильмы.

Сохранение и загрузка фильмов

- [x] Создать объект фильма на основании данных формы.
- [x] Сохранить объект фильма в файле.
- [x] Загрузить список фильмов из файла.
- [ ] Найти фильмы в файле.
- [ ] Вывести фильмы.

На странице со списком фильмов выводятся значения атрибутов `title` всех объектов `Movie`. Однако на форме также вводились значения атрибутов `director` и `year`, и их тоже необходимо где-то вывести...

Значит, нужно создать страницу для вывода подробной информации о фильме. На этой странице все атрибуты объекта `Movie` будут представлены в табличной форме.

На странице выводится подробная информация о фильме.

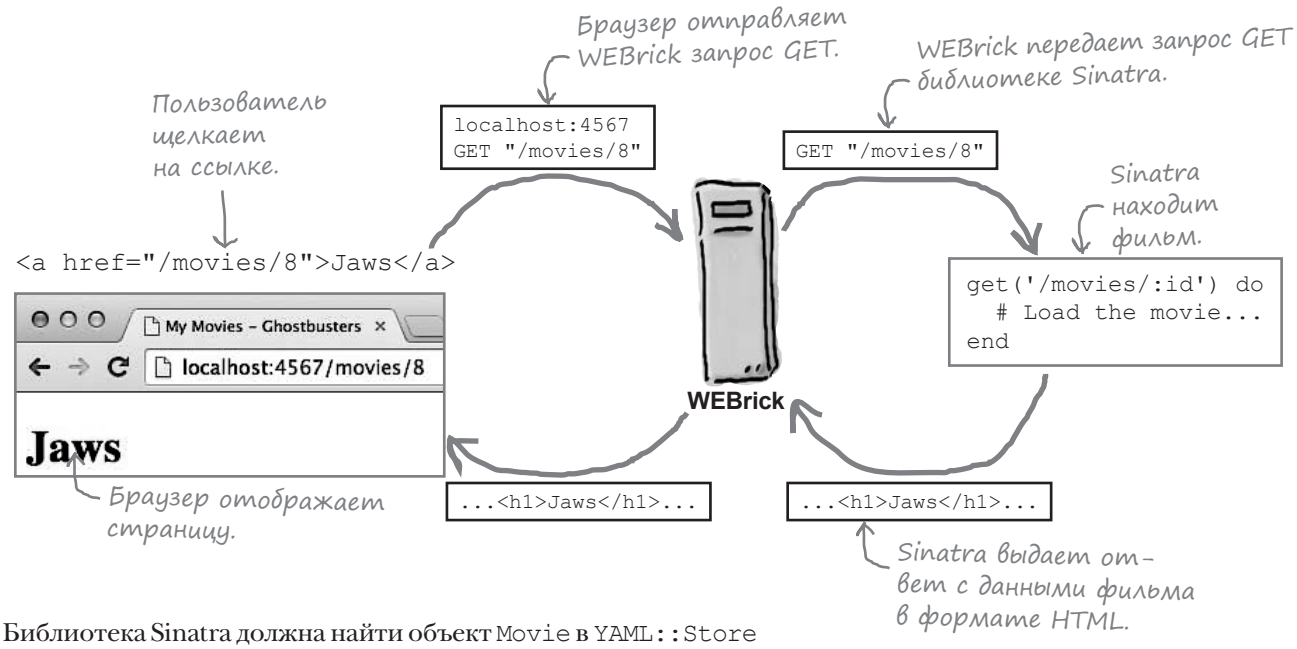
Также необходимо понять, как пользователь будет переходить к страницам конкретных фильмов... На странице со списком название каждого фильма будет преобразовано в ссылку для перехода к странице этого фильма.

Вместо обычного текста список должен содержать ссылки на страницы с подробной информацией о каждом фильме.

Построение ссылок HTML на фильмы (продолжение)

Чтобы ссылка на страницу с подробной информацией работала, ее путь должен содержать достаточно информации для нахождения ресурса (объект `Movie`), на который ведет ссылка.

В атрибуте `href` каждой ссылки HTML содержится путь к ресурсу. Когда пользователь щелкает на ссылке, она отправляет серверу запрос GET на получение этого ресурса (так же, как если бы пользователь ввел URL в адресной строке).



Библиотека Sinatra должна найти объект `Movie` в `YAML::Store` *исключительно* по пути к ресурсу в ссылке.

Какую же информацию следует включить для определения правильного объекта `Movie`? Как насчет атрибута `id`?

Мы можем встроить атрибут `id` в создаваемые ссылки... Позднее, когда пользователь щелкает на ссылке, этот идентификатор станет частью пути к ресурсу, отправленного серверу. Мы просто находим этот ключ в `YAML::Store` и используем возвращенный им объект `Movie` для построения страницы HTML!

```
---
1: !ruby/object:Movie
  title: Star Wars
  director: George Lucas
  year: '1977'
  id: 1
2: !ruby/object:Movie
  title: Ghostbusters
  director: Ivan Reitman
  year: '1984'
  id: 2
3: !ruby/object:Movie
  title: Forrest Gump
  director: Robert Zemeckis
  year: '1994'
  id: 3
```

Атрибут «id» объекта Movie включается в ссылку!

```
<a href="/movies/1">Star Wars</a>
<a href="/movies/2">Ghostbusters</a>
<a href="/movies/3">Forrest Gump</a>
```

Построение ссылок HTML на фильмы (продолжение)

Обновим разметку HTML для страницы со списком фильмов, чтобы на ней отображались ссылки на страницы с подробной информацией. Откройте файл *index.erb* в подкаталоге *views*. В блоке `each`, обрабатывающем каждый объект `Movie`, заключите название фильма в теги `<a>`. При помощи тега ERB `<%= %>` задайте атрибуту `href` ссылку на атрибут id текущего объекта `Movie`.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset='UTF-8' />
    <title>My Movies</title>
  </head>
  <body>
    <h1>My Movies</h1>
    <ul>
      <% @movies.each do |movie| %>
        <li>
          <a href="/movies/<%= movie.id %>"><%= movie.title %></a>
        </li>
      <% end %>
    </ul>
    <a href="/movies/new">Add New Movie</a>
  </body>
</html>
```

В цикле, в котором обрабатывается каждый фильм...

...добавляем ссылку на путь, содержащий идентификатор фильма...

...при этом название фильма используется в качестве отображаемого текста ссылки.

views/index.erb

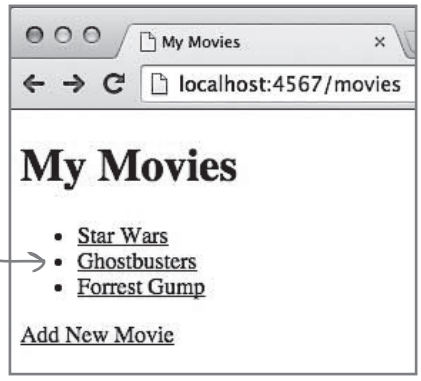
Если вы откроете адрес `http://localhost:4567/movies` в своем браузере, обновленная разметка HTML для списка фильмов будет выглядеть примерно так:

Идентификаторы фильмов в пути к ресурсу.

Названия фильмов как тексты ссылок.

```
<ul>
  <li>
    <a href="/movies/1">Star Wars</a>
  </li>
  <li>
    <a href="/movies/2">Ghostbusters</a>
  </li>
  <li>
    <a href="/movies/3">Forrest Gump</a>
  </li>
</ul>
```

Названия будут преобразованы в (неработающие) ссылки.



...а названия фильмов будут преобразованы в ссылки. Пока эти ссылки никуда не ведут, но сейчас мы решим эту проблему!

Именованные параметры в маршрутах Sinatra

Нам пришлось добавить в приложение Sinatra маршрут для обработки запросов списка фильмов:

```
get('/movies') do
  @movies = store.all
  erb :index
end
```

И еще один маршрут для обработки запросов формы для добавления фильма:

```
get('/movies/new') do
  erb :new
end
```

Так что вас, скорее всего, не удивит, что для страниц отдельных фильмов также понадобятся свои маршруты. Но конечно, добавлять маршрут для *каждого* фильма было бы нереально:

```
get('/movies/1') do
  # Загрузить фильм 1
end
get('/movies/2') do
  # Загрузить фильм 2
end
get('/movies/3') do
  # Загрузить фильм 3
end
```

Sinatra позволяет создать один маршрут, который будет обрабатывать запросы к нескольким ресурсам; для этого в путь к ресурсу включается *именованный параметр*. Чтобы обозначить именованный параметр, поставьте после косой черты (`/`) в строке пути двоеточие (`:`), за которым следует имя. Этот маршрут будет обрабатывать запросы к любому пути, соответствующему шаблону маршрута, а сегменты пути, совпадающие с именованными параметрами, сохраняются в хеше `params` и могут использоваться в коде.

Вероятно, проще продемонстрировать именованные параметры на конкретном примере. Если вы выполняете код Sinatra следующего вида:

```
require 'sinatra'

get('/zipcodes/:state') do
  "Postal codes for #{params['state']}..."
end
```

Маршрут содержит параметр с именем «state».

Отвечает строкой, включающей параметр «state».

...то при обращении к следующему URL:

`http://localhost:4567/zipcodes/Nebraska`

...Sinatra ответит строкой `"Postal codes for Nebraska..."`. А при обращении к URL:

`http://localhost:4567/zipcodes/Ohio`

...Sinatra ответит строкой `"Postal codes for Ohio..."`. Любая строка, следующая за `'/zipcodes/'` в URL-адресе, сохраняется с ключом `'state'` в хеше `params`.

Использование именованного параметра для получения идентификатора

Итак, мы хотим, чтобы библиотека Sinatra отвечала на любой URL-адрес в формате:

```
http://localhost:4567/movies/1
http://localhost:4567/movies/2
http://localhost:4567/movies/3
```

...но при этом *не хотим* писать код следующего вида:

```
get('/movies/1') do
  # Загрузить фильм 1
end
get('/movies/2') do
  # Загрузить фильм 2
end
...
```

Добавим один маршрут, использующий именованный параметр `'id'` в пути запроса. Он будет обрабатывать все запросы, URL-адреса которых соответствуют приведенному формату. Также при этом будет сохранен идентификатор из пути, и мы сможем использовать его для поиска соответствующего объекта `Movie`.

Определите новый маршрут `get` с путем `'/movies/:id'` в конце файла *app.rb*. (Важно, чтобы этот маршрут следовал *после* других маршрутов; вскоре мы объясним, почему.)

```
...
get('/movies') do
  ...
end

get('/movies/new') do
  ...
end

post('/movies/create') do
  ...
end

get('/movies/:id') do
  "Received a request for movie ID: #{params['id']}"
end
```

Проследите за тем, чтобы этот маршрут был ПОСЛЕДНИМ в приложении!

В параметре с именем «id» сохраняется часть пути запроса.

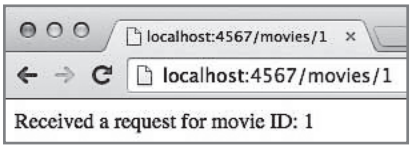
В ответ включается полученный параметр.

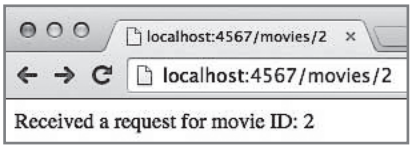
app.rb

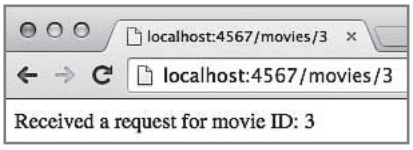
Затем попробуйте обратиться по каждому из этих URL-адресов:

```
http://localhost:4567/movies/1
http://localhost:4567/movies/2
http://localhost:4567/movies/3
```

...или любому другому идентификатору на ваш выбор. Sinatra отвечает на все такие запросы, включая в свой ответ значение параметра `'id'`.







Определение маршрутов в порядке приоритетов

Мы упоминали о том, что маршрут `get('/movies/:id')` должен определяться *после* других маршрутов в приложении Sinatra. Почему же это важно? А вот почему: запрос обрабатывается *первым* маршрутом Sinatra, соответствующим этому запросу. Все остальные маршруты игнорируются.

Предположим, имеется приложение для обработки заказов. В него включен маршрут для запроса формы нового заказа, а также другой маршрут с параметром `'part'` для просмотра всех существующих заказов, включающих определенную деталь. И допустим, что маршрут с именованным параметром определяется *до* другого маршрута...

```
require 'sinatra'

get('/orders/:part') do
  "Orders for Part: #{params['part']}"
end

get('/orders/new') do
  erb :new
end
```

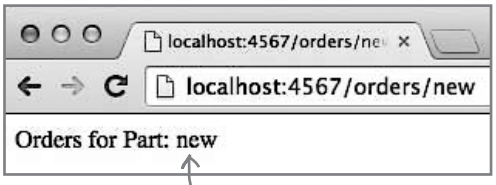
Этот маршрут на самом деле должен определяться последним...

...потому что он замещает этот маршрут!

При попытке обратиться к форме нового заказа:

```
http://localhost:4567/orders/new
```

...активизируется маршрут, определенный первым, и Sinatra попробует выдать заказы на деталь с `'id'='new'`!



«new» интерпретируется как идентификатор!

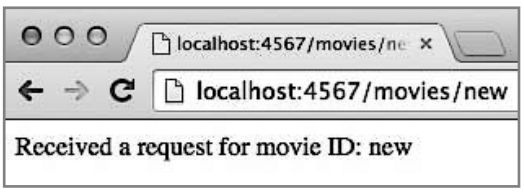
Если в приложении с фильмами определить маршрут `get('/movies/:id')` *до* маршрута `get('/movies/new')`, возникает та же проблема. Любая попытка загрузить форму нового фильма:

```
http://localhost:4567/movies/new
```

...будет интерпретирована как запрос на вывод подробной информации о фильме с `'id'='new'`.

```
...
get('/movies/:id') do
  "Received a request for movie ID: #{params['id']}"
end

get('/movies/new') do
  ...
end
...
```

Запросы к этому маршруту будут перехвачены предыдущим маршрутом!



Из этого следует, что в приложении сначала должны определяться более конкретные маршруты Sinatra, а менее конкретные должны определяться после них. Если какой-либо маршрут содержит именованные параметры, вероятно, он должен определяться среди последних.

Это приложение Sinatra работает не совсем правильно. Соедините каждый из трех URL-адресов, приведенных ниже, с ответом, который выдаст приложение. (Один из ответов останется лишним.)

```
require 'sinatra'

get('/hello') do
  "Hi there!"
end

get('/:greeting') do
  greeting = params['greeting']
  "Sinatra says #{greeting}!"
end

get('/goodbye') do
  "See you later!"
end
```

..... http://localhost:4567/hello

..... http://localhost:4567/ciao

..... http://localhost:4567/goodbye

A See you later!

B Sinatra says ciao!

C Sinatra says goodbye!

D Hi there!

Упражнение
Решение

Это приложение Sinatra работает не совсем правильно. Соедините каждый из трех URL-адресов, приведенных ниже, с ответом, который выдаст приложение. (Один из ответов останется лишним.)

```
require 'sinatra'

get('/hello') do
  "Hi there!"
end

get('/:greeting') do
  greeting = params['greeting']
  "Sinatra says #{greeting}!"
end

get('/goodbye') do
  "See you later!"
end
```

Этот маршрут содержит именованный параметр. Он должен определяться последним!

Этот маршрут замещается предшествующим маршрутом!

D http://localhost:4567/hello

B http://localhost:4567/ciao

C http://localhost:4567/goodbye

A See you later!

B Sinatra says ciao!

C Sinatra says goodbye!

D Hi there!

Поиск объекта Movie в YAML::Store

Мы встроили идентификаторы фильмов в ссылки на сайте, а также сохранили параметр с идентификатором из запроса HTTP GET. Пришло время использовать этот идентификатор для нахождения объекта `Movie`.

Идентификаторы фильмов.

```
<a href="/movies/1">Star Wars</a>
<a href="/movies/2">Ghostbusters</a>
<a href="/movies/3">Forrest Gump</a>
```

Все фильмы хранятся в `YAML::Store` с использованием идентификаторов в качестве ключей. Найти нужный фильм в хранилище будет несложно.

Атрибут «id» каждого объекта Movie совпадает с ключом, с которым объект был сохранен в YAML::Store.

```
---
1: !ruby/object:Movie
  title: Star Wars
  director: George Lucas
  year: '1977'
  id: 1
2: !ruby/object:Movie
  title: Ghostbusters
  director: Ivan Reitman
  year: '1984'
  id: 2
```

movies.yml

Добавим метод экземпляра `find` в класс `MovieStore`. Метод получает идентификатор, используемый в качестве ключа, и возвращает значение (объект `Movie`), ассоциированное с ключом в `YAML::Store`.

Как и другие операции `YAML::Store`, эта операция должна выполняться в блоке метода `transaction`.

```
require 'yaml/store'

class MovieStore

  def initialize(file_name)
    @store = YAML::Store.new(file_name)
  end

  def find(id)
    @store.transaction do
      @store[id]
    end
  end

  def all
    @store.transaction do
      @store.roots.map { |id| @store[id] }
    end
  end

  ...

end
```

Передается идентификатор, используемый в качестве ключа.

Должно происходить внутри блока transaction...

Возвращает объект Movie, хранящийся с этим ключом.

movie_store.rb

Готово! Проблема с поиском сохраненных фильмов решена.

Шаблон ERB для отдельного фильма

В классе `MovieStore` появился новый метод `find`, готовый вернуть данные отдельного фильма. Остается лишь использовать `find` в приложении Sinatra и добавить разметку HTML для вывода информации о фильме. Работа почти завершена!

- [x] Загрузить список фильмов из файла.
- [x] Найти фильмы в файле.
- [ ] Вывести фильмы.

Мы решили проблемы с загрузкой отдельного объекта `Movie`, так что мы можем вывести его атрибуты, но нам все равно нужен шаблон HTML, в котором эти данные должны отображаться. Создайте файл *show.erb* в подкаталоге *views* и добавьте в него следующую разметку HTML.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset='UTF-8' />
    <title>My Movies – <%= @movie.title %></title>
  </head>
  <body>
    <h1><%= @movie.title %></h1>
    <table>
      <tr>
        <td><strong>Title:</strong></td>
        <td><%= @movie.title %></td>
      </tr>
      <tr>
        <td><strong>Director:</strong></td>
        <td><%= @movie.director %></td>
      </tr>
      <tr>
        <td><strong>Year Published:</strong></td>
        <td><%= @movie.year %></td>
      </tr>
    </table>
    <a href="/movies">Back to Index</a>
  </body>
</html>
```

Название фильма встраивается в название страницы.

Заголовок с названием фильма.

Таблица для хранения атрибутов фильма.

Метки выводятся жирным шрифтом.

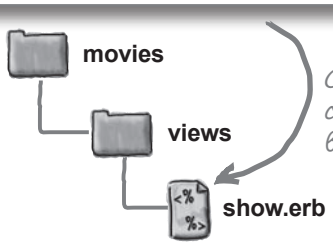
Название фильма.

Режиссер.

Год выпуска.

Ничего принципиально нового в этой странице нет. Мы обращаемся к объекту `Movie`, хранящемуся в переменной экземпляра `@movie` (это будет сделано в маршруте Sinatra). Теги ERB `<%= %>` используются для внедрения атрибутов фильма в HTML.

Тег HTML `<table>` используется для вывода атрибутов фильма по строкам. В первом столбце каждой строки хранится метка атрибута. Тег HTML `<strong>` (который нам ранее не встречался) просто выводит текст **жирным шрифтом**. Во втором столбце располагается значение атрибута.



Сохраните в файле с именем «show.erb» в подкаталоге «views».

В итоге страница должна выглядеть примерно так.

TITLE HERE

Title:	TITLE HERE
Director:	DIRECTOR HERE
Year Published:	9999

Back to Index

Завершение маршрута Sinatra для отдельных фильмов

Мы добавили в `MovieStore` метод `find` для загрузки объекта `Movie` по атрибуту `id`, а также файл *show.erb* для вывода объекта `Movie`. Пора связать эти компоненты воедино. В приложении Sinatra изменим маршрут (`'/movies/:id'`), чтобы он загружал информацию фильма и отображал ее в формате HTML.

```
require 'sinatra'
require 'movie'
require 'movie_store'

store = MovieStore.new('movies.yml')

get('/movies') do
  @movies = store.all
  erb :index
end

get('/movies/new') do
  erb :new
end

post('/movies/create') do
  @movie = Movie.new
  @movie.title = params['title']
  @movie.director = params['director']
  @movie.year = params['year']
  store.save(@movie)
  redirect '/movies/new'
end

get('/movies/:id') do
  id = params['id'].to_i
  @movie = store.find(id)
  erb :show
end
```

Параметр 'id' преобразуется из строки в целое число.

Идентификатор используется для загрузки объекта из хранилища.

Данные фильма встраиваются в разметку HTML из show.erb и возвращаются браузеру.

app.rb

Чтобы загрузить объект `Movie` из `MovieStore`, необходимо взять параметр `'id'` из пути к ресурсу. Однако параметр будет содержать строку, а ключи `MovieStore` являются целыми числами. Следовательно, первое, что нужно сделать в блоке route — преобразовать строку в целое число с помощью метода `to_i`.

Имеющийся идентификатор в форме целого числа можно передать его методу `find` экземпляра `MovieStore`. Метод возвращает объект `Movie`, который будет сохранен в переменной экземпляра `@movie` (для использования в шаблоне ERB).

Наконец, вызов метода `erb :show` загружает шаблон *show.erb* из каталога *views*, встраивает в него атрибуты объекта `@movie` и возвращает полученную разметку HTML браузеру.

Нам пришлось основательно потрудиться, чтобы организовать хранение данных, но зато подключить их к приложению оказалось достаточно просто. Приложение Sinatra готово!

Сохранение и загрузка объектов

- ☑ Создать объект фильма на основании данных формы.
- ☑ Сохранить объект фильма в файле.
- ☑ Загрузить список фильмов из файла.
- ☑ Найти фильмы в файле.
- ☑ Вывести фильмы.

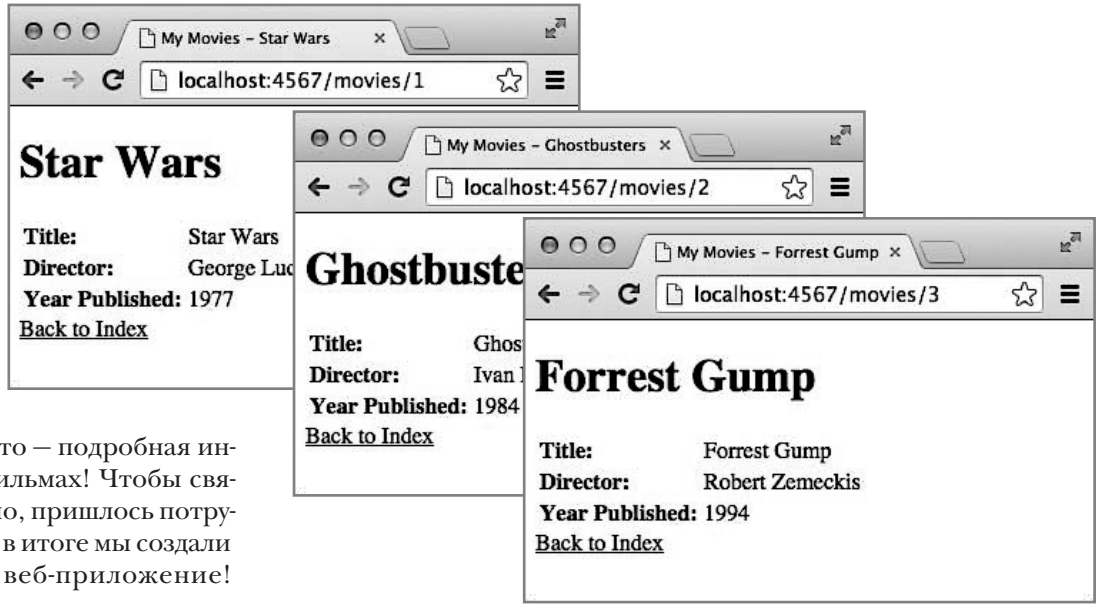
А теперь проверим, как работает приложение!

Готовы? Мы проделали изрядную работу, так что это великий момент...

Перезапустите приложение из терминального окна и откройте в браузере страницу `http://localhost:4567/movies`. Щелкните на любой ссылке в списке.

Щелкните на любой ссылке.

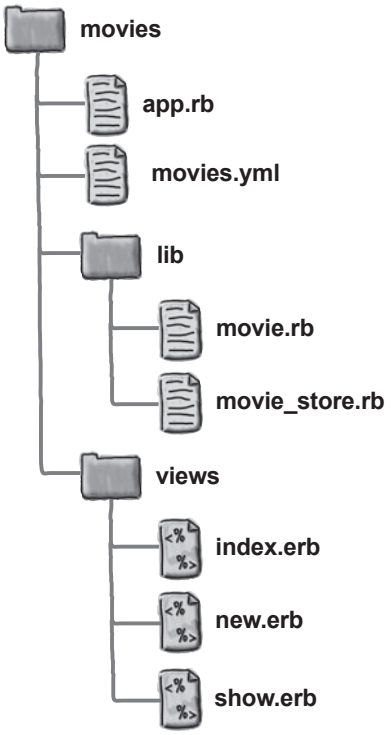
Приложение получает идентификатор фильма из URL, загружает экземпляр `Movie` из `YAML::Store`, встраивает его атрибуты в *show.erb* и отправляет полученную разметку HTML браузеру.



И тут наконец-то — подробная информация о фильмах! Чтобы связать все воедино, пришлось потрудиться, но зато в итоге мы создали полноценное веб-приложение!

Полный код приложения

Вот как выглядит структура каталогов приложения:



Файл *app.rb* содержит основной код приложения. В нем определяются все маршруты Sinatra.

```
require 'sinatra'
require 'movie'
require 'movie_store'

store = MovieStore.new('movies.yml')

get('/movies') do
  @movies = store.all
  erb :index
end

get('/movies/new') do
  erb :new
end

post('/movies/create') do
  @movie = Movie.new
  @movie.title = params['title']
  @movie.director = params['director']
  @movie.year = params['year']
  store.save(@movie)
  redirect '/movies/new'
end

get('/movies/:id') do
  id = params['id'].to_i
  @movie = store.find(id)
  erb :show
end
```

Создаем экземпляр MovieStore, работающий с файлом movies.yml.

Загружаем все объекты Movie из movies.yml.

Встраиваем данные фильмов в разметку HTML из шаблона views/index.erb и возвращаем результат.

Возвращаем разметку HTML в views/new.erb.

Сюда передаются отправленные данные формы.

Создаем объект для сохранения данных формы.

Данные формы добавляются в объект.

Сохраняем объект!

Отображается новая пустая форма.

Параметр 'id' преобразуется из строки в целое число.

Используем идентификатор для загрузки фильмов из хранилища.

Встраиваем данные фильма в разметку HTML из show.erb и возвращаем результат браузеру.

app.rb

В файле *movies.yml* хранятся все данные фильмов, сохраненные классом `MovieStore`. (Конкретное содержимое зависит от того, какие данные были введены на форме HTML.)

```
---
1: !ruby/object:Movie
  title: Star Wars
  director: George Lucas
  year: '1977'
  id: 1
2: !ruby/object:Movie
  ...
```

Атрибут «id» каждого объекта Movie совпадает с тем ключом, с которым этот объект хранится в YAML::Store.

movies.yml

Полный код приложения (продолжение)

Класс `Movie` просто определяет несколько атрибутов для каждого объекта фильма.

```
class Movie
  attr_accessor :title, :director, :year, :id
end
```

lib/movie.rb

Класс `MovieStore` отвечает за хранение объектов `Movie` в файле YAML и их последующую загрузку из хранилища.

```
require 'yaml/store'          # Загружаем класс YAML::Store.

class MovieStore

  def initialize(file_name)   # Создаем хранилище для чтения/записи в файл
    @store = YAML::Store.new(file_name)   # с заданным именем.
  end

  def find(id)                # Метод находит объект Movie с заданным идентификатором.
    @store.transaction do     # Для работы с хранилищем необходим вызов transaction.
      @store[id]              # Возвращает объект Movie, ассоциированный с этим ключом.
    end
  end

  def all                     # Метод загружает все фильмы из хранилища
    @store.transaction do     # Тоже необходим вызов transaction...
      @store.roots.map { |id| @store[id] }   # Создаем массив со значениями, ассоциированными со всеми ключами.
    end
  end

  def save(movie)             # Метод сохраняет объект Movie в хранилище.
    @store.transaction do     # Тоже необходим вызов transaction...
      unless movie.id         # Если фильму еще не присвоен идентификатор...
        highest_id = @store.roots.max || 0   # ...находим наибольший ключ...
        movie.id = highest_id + 1            # ...и увеличиваем его.
      end
      @store[movie.id] = movie   # Сохраняем фильм с ключом, совпадающим с его идентификатором.
    end
  end

end
```

lib/movie_store.rb

Полный код приложения (продолжение)

Файл *show.erb* из подкаталога *views* содержит шаблон ERB с разметкой HTML для внедрения данных одного фильма.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset='UTF-8' />
    <title>My Movies – <%= @movie.title %></title>
  </head>
  <body>
    <h1><%= @movie.title %></h1>
    <table>
      <tr>
        <td><strong>Title:</strong></td>
        <td><%= @movie.title %></td>
      </tr>
      <tr>
        <td><strong>Director:</strong></td>
        <td><%= @movie.director %></td>
      </tr>
      <tr>
        <td><strong>Year Published:</strong></td>
        <td><%= @movie.year %></td>
      </tr>
    </table>
    <a href="/movies">Back to Index</a>
  </body>
</html>
```

Название фильма встраивается в название страницы.

Заголовок с названием фильма.

Название фильма.

Режиссер.

Год выпуска.

views/show.erb

Файл *index.erb* содержит шаблон для создания ссылки на каждый отдельный фильм.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset='UTF-8' />
    <title>My Movies</title>
  </head>
  <body>
    <h1>My Movies</h1>
    <ul>
      <% @movies.each do |movie| %>
        <li>
          <a href="/movies/<%= movie.id %>"><%= movie.title %></a>
        </li>
      <% end %>
    </ul>
    <a href="/movies/new">Add New Movie</a>
  </body>
</html>
```

Обрабатываем все фильмы...

...добавляем ссылку на путь, содержащий идентификатор фильма...

...название фильма используется в качестве отображаемого текста ссылки.

views/index.erb

Полный код приложения (продолжение)

Наконец, файл *new.erb* содержит форму HTML для ввода данных нового фильма. При отправке данных формы описание фильма передается в запросе HTTP POST с путем `'/movies/create'`. Маршрут Sinatra в *app.rb* использует данные для создания нового объекта `Movie` и его сохранения в `MovieStore`.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset='UTF-8' />
    <title>My Movies — Add Movie</title>
  </head>
  <body>
    <h1>Add New Movie</h1>
    <form method="post" action="/movies/create">
      <table>
        <tr>
          <td><label for="title">Title:</label></td>
          <td><input type="text" name="title"></td>
        </tr>
        <tr>
          <td><label for="director">Director:</label></td>
          <td><input type="text" name="director"></td>
        </tr>
        <tr>
          <td><label for="year">Year Published:</label></td>
          <td><input type="text" name="year"></td>
        </tr>
        <tr>
          <td><input type="submit"></td>
        </tr>
      </table>
    </form>
    <a href="/movies">Back to Index</a>
  </body>
</html>
```

Данные формы отправляются запросом POST с путем «/movies/create».

Метка поля.

Текстовое поле.

Метка поля.

Текстовое поле.

Метка поля.

Текстовое поле.

Кнопка отправки данных формы.

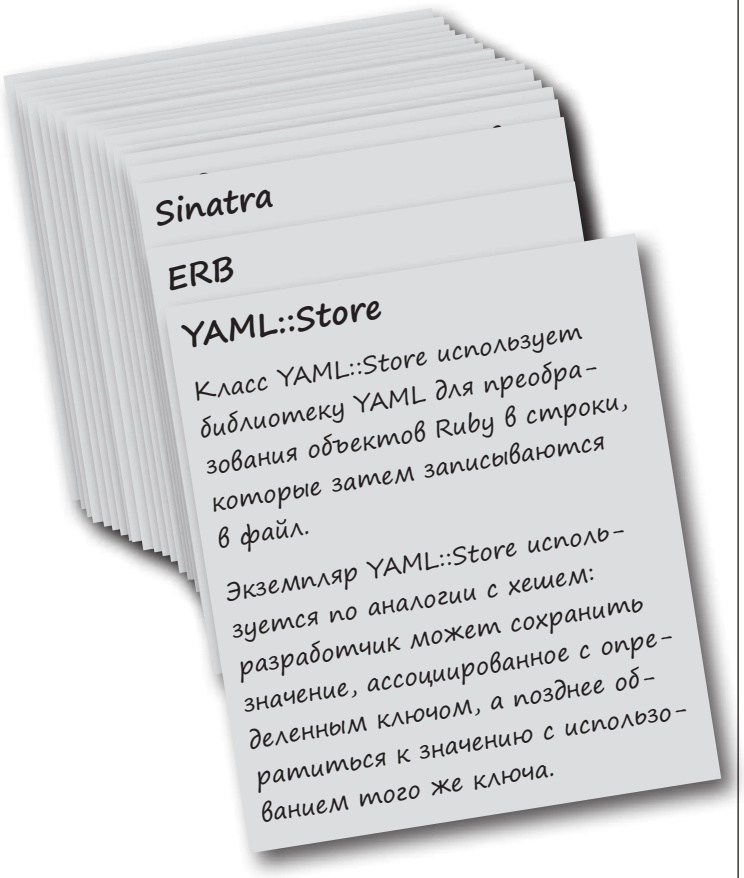
views/new.erb

Вот и все — мы создали полноценное веб-приложение, которое умеет сохранять данные, введенные пользователем, и загружать их из хранилища.

Написание веб-приложений может быть исключительно сложным делом, но библиотека Sinatra существенно упрощает его для вас!

Ваш инструментарий Ruby

Глава 15 осталась позади, а ваш инструментарий пополнился методами работы с хранилищем YAML::Store.



КЛЮЧЕВЫЕ МОМЕНТЫ

- Когда методу attribute формы HTML присвоено значение `"post"` и пользователь отправляет данные формы, браузер передает данные формы серверу в запросе HTTP POST.
- У форм также имеется атрибут `action`, задающий путь к ресурсу. Путь включается в запросы POST (как и в случае с запросами GET).
- В Sinatra определен метод `post`, который используется для определения маршрутов для запросов POST.
- Вызов метода `params` в блоке маршрута `post` возвращает хеш с данными формы запроса.
- Метод `YAML::Store.new` получает строку с именем файла, с которым выполняются операции чтения и/или записи.
- У экземпляров `YAML::Store` имеется метод `transaction`, предотвращающий запись в файл со стороны других программ. Метод `transaction` получает блок, в котором можно вызывать любые необходимые методы `YAML::Store`.
- Метод экземпляра `roots` класса `YAML::Store` возвращает массив со всеми ключами хранилища.
- Sinatra позволяет включить в путь маршрута именованные параметры. Часть пути запроса, находящаяся в позиции именованного параметра, сохраняется и становится доступной в хеше `params`.
- Если один запрос подходит для нескольких маршрутов Sinatra, он будет обработан тем маршрутом, который был определен ранее других.
- Маршруты с именованными параметрами обычно следует определять последними, чтобы избежать случайного замещения других маршрутов.

Далее в программе…

Это еще не все! Ограниченный объем книги не позволил нам рассмотреть многие важные темы, поэтому мы добавили приложение с краткой сводкой наиболее важных вопросов, а также перечнем ресурсов, которые помогут вам подготовиться к вашему *следующему* проекту на Ruby. Читайте дальше!

Поздравляем!

Вы дочитали последнюю главу.

Конечно, еще есть приложение.
И еще веб-сайт...
В общем, никуда вы от нас не денетесь.

Десять основных тем (не рассмотренных в книге)

Мы прошли долгий путь, и книга почти закончена. Мы будем скучать по вам, но было бы неправильно расставаться и отпускать вас в самостоятельное путешествие без еще *нескольких* напутственных слов. Нам при всем желании не удалось бы уместить все, что вам еще нужно знать о Ruby, на этих страницах… (Вообще-то сначала мы *уместили* все необходимое, уменьшив размер шрифта в 25 000 раз. Все поместилось, но текст было не прочитать без микроскопа, поэтому материал пришлось изрядно сократить.) Но мы оставили все самое лучшее для этого приложения.

И это уже *действительно* конец книги!

1. Другие полезные библиотеки

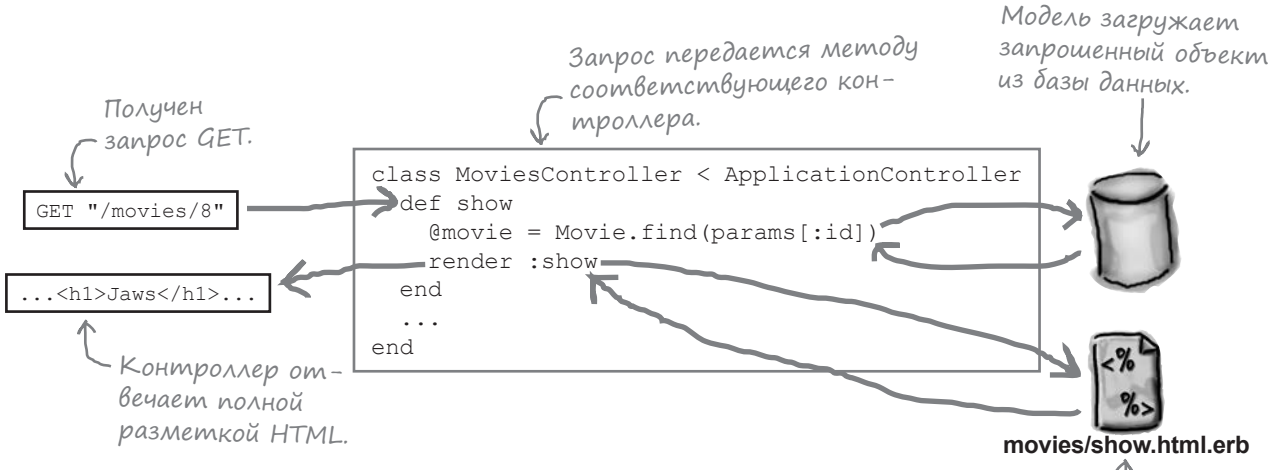
Ruby on Rails

Библиотека Sinatra (рассмотренная в главах 14 и 15) прекрасно подходит для построения простых веб-приложений. Но разработчики всегда стараются дополнить свои продукты новыми возможностями, так что приложения со временем увеличиваются. Со временем одним местом для хранения шаблонов ERB дело уже не ограничится. Вам также понадобится место для размещения конфигурации базы данных, место для размещения кода JavaScript и стилей CSS, место для размещения кода, связывающее все это воедино... и многое другое.

Именно это является сильной стороной Ruby on Rails: стандартизация мест для размещения чего-либо.

Все начинается с фундаментальной архитектуры любого приложения Rails, которое строится на базе популярного паттерна *Модель, Представление, Контроллер* (MVC):

- *Модель* используется для размещения данных приложения. Rails умеет автоматически сохранять объекты модели в базе данных и загружать их позднее. (В этом отношении она напоминает классы `Movie` и `MovieStore`, созданные нами для приложения Sinatra.)
- В *представлении* размещается код отображения данных модели для пользователей. По умолчанию Rails использует шаблоны ERB для отображения данных в формате HTML (или JSON, или XML). (Опять же по аналогии с приложением Sinatra.)
- В *контроллере* размещается код обработки запросов браузера. Контроллер получает запрос, обращается к модели за нужными данными, обращается к представлению для отображения нужного ответа, а затем возвращает ответ браузеру.



Выбрать подходящее место для этого кода — задача не из простых. Настроить Ruby так, чтобы он знал, где все это можно найти, будет еще сложнее. Вот почему Rails активно использует принцип «соглашений по конфигурации». Если код модели, представления и контроллера будет размещаться в стандартных местах, где его будет искать Rails, то вам не придется заниматься настройкой. Все будет делаться автоматически. Вот почему веб-инфраструктура Rails пользуется такой популярностью!

За дополнительной информацией о Rails обращайтесь по адресу *http://rubyonrails.org/*.

1. Другие полезные библиотеки (продолжение)

dRuby

dRuby (часть стандартной библиотеки Ruby) наглядно демонстрирует мощь Ruby. Название происходит от сокращения «distributed Ruby» (распределенный Ruby); эта библиотека позволяет организовать доступ по сети к *любому* объекту Ruby. Разработчик просто создает объект и приказывает dRuby открыть доступ к нему как к сетевой службе. После этого методы объекта могут вызываться из сценариев Ruby, работающих на *другом компьютере*. Вам не придется писать специальный код сетевой службы. Код просто работает благодаря уникальной особенности Ruby: возможности передачи вызовов методов от одного объекта к другому (вскоре мы поговорим об этом подробнее).

Перед вами короткий сценарий, который предоставляет доступ к обычному массиву по сети. Мы передаем dRuby массив и задаем URL-адрес (включающий номер порта), по которому он должен быть доступен. Мы также добавили код повторного вывода массива, чтобы вы могли проследить за тем, как он изменяется клиентскими программами.

```
require 'drb/drb'                                  ← Загружаем библиотеку dRuby.

my_object = []                                     ← Создаем пустой массив.
DRb.start_service("druby://localhost:8787", my_object)
20.times do                                        ← Цикл выполняется 20 раз.
  sleep 10                                         ← Ожидаем 10 секунд.
  p my_object                                      ← Выводим массив.
end
DRb.thread.join                                    ← Ожидаем завершения сервера перед выходом.
```

URL-адрес для доступа к массиву.

server.rb

А теперь мы приведем отдельный сценарий, который выполняет функции клиента. Он подключается к серверному сценарию по сети и получает объект, действующий как *заместитель* удаленного объекта.

```
require 'drb/drb'
DRb.start_service
remote_object = DRbObject.new_with_uri("druby://localhost:8787")
remote_object.push "hello", "network"              ← Вызываем метод массива.
p remote_object.last                               ← Вызываем другой метод массива.
```

Подключаемся к порту, указанному выше.

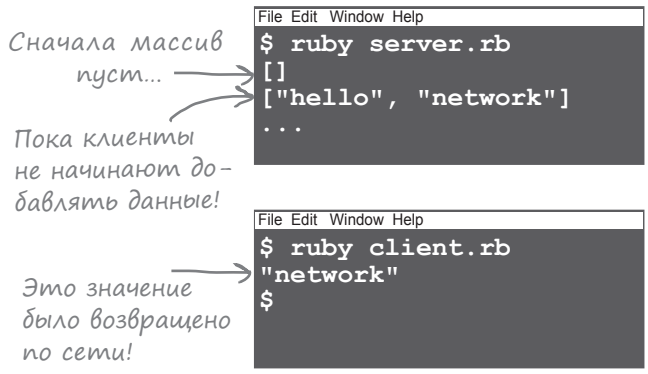
client.rb

Любой вызов метода заместителя передается по сети и воспроизводится для удаленного объекта. Возвращаемые значения, полученные при вызове, передаются по сети и возвращаются заместителем.

Чтобы проверить, как работает dRuby, откройте терминальное окно и запустите **`server.rb`**. Сценарий начинает выводить содержимое массива `my_object` каждые 10 секунд.

Теперь в *другом* терминальном окне запустите **`client.rb`**. Переключившись в первое терминальное окно, вы увидите, что клиент добавил новые данные в массив!

Возникают ли при этом проблемы, связанные с безопасностью? Еще бы. Проследите за тем, чтобы при использовании dRuby программа была защищена брандмауэром. За дополнительной информацией обращайтесь к описанию dRuby в документации стандартной библиотеки Ruby.



1. Другие полезные библиотеки (продолжение)

CSV

Если вам когда-либо доводилось заниматься офисной работой, вы наверняка работали с данными в электронных таблицах — вашими или подготовленными кем-то другим. В электронных таблицах могут определяться формулы, но их синтаксис весьма примитивен (и плохо запоминается).

Программы для работы с электронными таблицами обычно могут экспортировать данные в формате CSV (сокращение от «**C**omma-**S**eparated **V**alues») — простом текстовом формате, в котором значения разбиваются по строкам и столбцам и отделяются друг от друга запятыми.

Файл в формате CSV.

```
Associate,Sale Count,Sales Total
"Boone, Agnes",127,1710.26
"Howell, Marvin",196,2245.19
"Rodgers, Tonya",400,3032.48
```

sales.csv

Библиотека CSV, упрощающая обработку файлов CSV, входит в состав стандартной библиотеки Ruby. Следующий короткий сценарий использует CSV для вывода имен и общих продаж из предыдущего файла. Он пропускает строку заголовка, но использует имена в заголовке как ключи для обращения к значениям столбцов, словно каждая строка файла CSV представляет собой маленький хеш.

Загружаем библиотеку.

Первая строка интерпретируется как набор имен столбцов.

Обрабатываем каждую строку файла.

```
require 'csv'
CSV.foreach("sales.csv", headers: true) do |row|
  puts "#{row['Associate']}: #{row['Sales Total']}"
end
```

Обращаемся к данным столбцов, используя имена в заголовках как ключи.

```
Boone, Agnes: 1710.26
Howell, Marvin: 2245.19
Rodgers, Tonya: 3032.48
```

К сожалению, здесь мы смогли только упомянуть об этих трех приложениях. Чтобы узнать о десятках других полезных возможностей стандартной библиотеки, воспользуйтесь поиском в Интернете. А на сайте *http://rubygems.org* вы найдете буквально *тысячи* других полезных пакетов.

2. Компактные версии if и unless

Мы все время говорим, что Ruby помогает выполнять больше полезной работы при меньшем объеме кода. Этот принцип встроен на уровне самого языка. Одним из примеров такого рода служат встроенные условные конструкции.

Конечно, вам уже знакомы традиционные формы `if` и `unless`:

```
if true
  puts "I'll be printed!"
end
```

```
unless true
  puts "I won't!"
end
```

Но если код условной конструкции занимает всего одну строку, условие можно переместить в конец строки. Следующие выражения работают точно так же, как приведенные выше:

```
puts "I'll be printed!" if true

puts "I won't!" unless true
```

3. Приватные методы

Когда вы создаете новый класс, вероятно, какое-то время вы будете его единственным пользователем. Но так (хочется верить) будет не всегда. Другие разработчики увидят ваш класс и поймут, что он может пригодиться для решения их задач. Однако их задачи не всегда точно совпадают с вашими; может оказаться, что они будут использовать ваш класс так, как не было предусмотрено исходным планом. И в этом нет ничего страшного — до тех пор, пока вы не решите внести изменения в свой класс.

Предположим, вам потребовалось увеличить на 15% все суммы в счетах ваших клиентов. Вы создали класс `Invoice` с атрибутом `subtotal` и методом `total`, вычисляющим общую сумму счета. И чтобы избежать чрезмерного усложнения метода `total`, вы выделили вычисления процентов в отдельный метод `fees`, вызываемый из `total`.

```
class Invoice
  attr_accessor :subtotal
  def total
    subtotal + fees(subtotal, 0.15)
  end
  def fees(amount, percentage)
    amount * percentage
  end
end

invoice = Invoice.new
invoice.subtotal = 500.00
p invoice.total
```

Значение subtotal увеличивается на 15%.

Значение amount умножается на коэффициент.

```
575.0
```

А потом вдруг выясняется, что коммерческий отдел заодно решил добавить ко всем счетам фиксированную сумму $25. Вы добавляете в метод `fees` параметр `flat_rate`...

```
class Invoice
  attr_accessor :subtotal
  def total
    subtotal + fees(subtotal, 0.15, 25.00)
  end
  def fees(amount, percentage, flat_rate)
    amount * percentage + flat_rate
  end
end
```

Добавляем фиксированную доплату $25.

В метод «fees» включается новый параметр.

И все отлично работает — пока вам не позвонят разгневанные пользователи из другого отдела, которые хотят знать, почему их код перестал работать из-за вашего класса. Похоже, они начали использовать метод `fees` для вычисления своей доплаты, составляющей 8%. Но их код написан в предположении, что метод `fees` вызывается с *двумя* параметрами, тогда как в новой версии он получает *три*!

```
fee = Invoice.new.fees(300, 0.08)
p fee
```

```
in 'fees': wrong number of arguments (2 for 3)
```

3. Приватные методы (продолжение)

Проблема в том, что другие разработчики вызывают ваш метод `fees` *извне*, тогда как на самом деле он должен был вызываться только *внутри* вашего класса. Теперь приходится решать: то ли придумывать, как заставить метод `fees` работать и для ваших целей, *и* для целей других разработчиков, которые используют его, то ли вернуть код к прежнему состоянию и больше никогда не изменять его.

Впрочем, подобные ситуации можно предотвратить. Если вы знаете, что метод должен использоваться только внутри вашего класса, пометьте его ключевым словом `private`. **Приватные методы** могут вызываться только из кода класса, в котором они были определены. Ниже приведена обновленная версия класса `Invoice`, в которой метод `fees` помечен ключевым словом `private`:

```
class Invoice
  attr_accessor :subtotal
  def total
    subtotal + fees(subtotal, 0.15, 25.00)
  end
private
  def fees(amount, percentage, flat_rate)
    amount * percentage + flat_rate
  end
end
```

Приватные методы могут вызываться из других методов Invoice.

Все методы, определяемые после этого ключевого слова, будут приватными для класса Invoice.

Теперь при попытке вызова метода `fees` за пределами класса `Invoice` происходит ошибка с выдачей сообщения о том, что вызываемый метод объявлен приватным.

```
fee = Invoice.new.fees(300, 0.08)
```

```
private method `fees' called for #<Invoice:0x007f97bb02ba20>
```

При этом ваш метод `total` (метод экземпляра того же класса, к которому принадлежит `fees`) все равно может вызывать этот метод.

```
invoice = Invoice.new
invoice.subtotal = 500.00
p invoice.total
```

```
600.0
```

Вызывает «fees» и включает результат в свое возвращаемое значение.

Другому отделу придется поискать другой способ вычисления доплаты, это верно. Но по крайней мере вы предотвратили все будущие недоразумения такого рода. И вы сможете вносить любые необходимые изменения в метод `fees`! Приватные методы делают ваш код более чистым и упрощают его обновление.

4. Аргументы командной строки

Допустим, вы хотите включить в свое сообщение электронной почты цитату из сообщения, на которое вы отвечаете. Для этого перед каждой строкой цитируемого текста выводится символ >. По этому признаку получатель легко поймет, о чем идет речь в сообщении.

Перед вами простой сценарий, который читает содержимое текстового файла и вставляет символ «> » перед каждой строкой:

```
file = File.open("email.txt") do |file|
  file.each do |line|
    puts "> " + line
  end
end
```

Открываем файл с заданным именем.

Метод «each» объекта File последовательно передает каждую строку блоку.

quote.rb

Но в этом варианте нам придется открывать сценарий и изменять имя файла каждый раз, когда мы захотим использовать его с новым входным файлом. Было бы неплохо иметь возможность менять имя файла *за пределами* сценария.

```
> Jay,
>
> Do you have any idea how far past deadline we are?
> What am I supposed to tell the copy editor?
>
> -Meghan
```

И такая возможность существует! Достаточно использовать *аргументы командной строки*. Программы, выполняемые в терминальном окне, часто позволяют указать аргументы после имени программы (по аналогии с аргументами при вызове метода), и сценарии Ruby не являются исключением. Аргументы, переданные при вызове сценария, доступны в массиве `ARGV`, заполняемом при каждом запуске. Первый аргумент хранится в элементе `ARGV[0]`, второй — в элементе `ARGV[1]`, и так далее.

Приведенный ниже короткий сценарий *args_test.rb* демонстрирует эту возможность. Если запустить ее в терминальном окне, все данные, следующие за именем сценария, будут выведены в ходе выполнения.

```
p ARGV[0]
p ARGV[1]
```

args_test.rb

```
File Edit Window Help
$ ruby args_test.rb hello terminal
"hello"
"terminal"
```

Мы можем использовать массив `ARGV` в *quote.rb*, чтобы задать любой входной файл при запуске программы. Нужно лишь заменить жестко запрограммированное имя файла на `ARGV[0]`. И с этого момента при запуске *quote.rb* в терминальном окне мы можем просто указать имя входного файла после имени сценария!

Первый аргумент командной строки определяет имя открываемого файла.

```
file = File.open(ARGV[0]) do |file|
  file.each do |line|
    puts "> " + line
  end
end
```

quote.rb

В итоге имя обрабатываемого файла становится аргументом сценария!

```
File Edit Window Help
$ ruby quote.rb reply.txt
> Tell them I'm really sorry!
> Just ONE more week!
>
> -Jay
```

5. Регулярные выражения

Регулярные выражения предназначены для поиска *по шаблону* в тексте. Поддержка регулярных выражений реализована во многих языках программирования, но в Ruby с ними особенно удобно работать.

Например, если вы ищете адрес электронной почты в тексте, при помощи регулярного выражения можно сформулировать следующий критерий: «Нужно найти последовательность из какого-то количества букв, за которыми следует знак @, потом еще несколько букв, потом точка и еще сколько-то букв».

```
/\w+@\w+\.\w+/ <-- Регулярное выражение для поиска адреса электронной почты.
```

Если вы ищете конец предложения, смысл регулярного выражения будет таким: «Нужно найти точку, вопросительный или восклицательный знак, за которыми следует пробел».

```
/[.?!]\s/ <-- Регулярное выражение для поиска конца предложения.
```

Регулярные выражения обладают невероятной мощью, но они также бывают сложными и неудобочитаемыми. Впрочем, даже если вы освоите только простейшие возможности регулярных выражений, они все равно принесут немало пользы. Здесь мы постараемся лишь дать представление о том, на что способны регулярные выражения, и порекомендуем пару ресурсов для самостоятельного изучения темы.

Предположим, имеется строка, в которой нужно найти телефонный номер:

```
"Tel: 555-0199"
```

Создайте регулярное выражение, которое найдет его для вас. Литералы регулярных выражений начинаются и заканчиваются символом косой черты (/).

```
/555-0199/ <-- Литерал регулярного выражения.
```

Впрочем, регулярное выражение само по себе еще ничего не делает. При помощи оператора =~ в условном выражении можно проверить, существует ли совпадение у регулярного выражения в строке:

```
                    Проверяем, существует ли у регулярного выражения (справа) совпадение в строке (слева).
if "Tel: 555-0199" =~ /555-0199/
  puts "Found phone number."
end
```

```
Found phone number.
```

Чтобы программировать на Ruby, не обязательно знать регулярные выражения.

Регулярные выражения чрезвычайно мощны, но при этом они весьма сложны, а на практике применяются не так уж часто. Но даже если вы ограничитесь только основными возможностями, они могут принести огромную пользу, если вы интенсивно работаете со строками!

5. Регулярные выражения (продолжение)

На данный момент наше регулярное выражение может совпасть только с одним телефонным номером: 555–0199. Чтобы оно могло совпадать и с другими номерами, можно воспользоваться *символьным классом* `\d`, который совпадает с любой цифрой от `0` до `9`. (Также существуют и другие символьные классы — например, `\w` для символов в словах или `\s` для пропусков.)

```
if "Tel: 555-0148" =~ /\d\d\d-\d\d\d\d/ <— Совпадает с любой цифрой.
  puts "Found phone number."
end
```

```
Found phone number.
```

Вместо того чтобы многократно вводить `\d` в этом коде, можно указать один экземпляр `\d`, за которым следует символ `+` — признак одного *и более* вхождений предыдущего совпадения.

```
if "Tel: 555-0148" =~ /\d+-\d+/ <— Совпадает с одной или несколькими цифрами.
  puts "Found phone number."
end
```

```
Found phone number.
```

Еще удобнее использовать другую конструкцию — число, заключенное в фигурные скобки. Она обозначает предыдущее совпадение, повторяющееся *заданное количество раз*.

```
if "Tel: 555-0148" =~ /\d{3}-\d{4}/ <— Три цифры, затем дефис и еще четыре цифры.
  puts "Found phone number."
end
```

```
Found phone number.
```

Конкретный фрагмент текста, с которым совпало регулярное выражение, может быть сохранен с помощью *сохраняющей группы*. Если заключить часть регулярного выражения в круглые скобки, эта часть совпадения будет сохранена в специальной переменной с именем `$1`. Например, можно вывести значение из `$1`, чтобы узнать, с чем совпала эта часть выражения.

```
                                  Совпадающая часть строки со-
if "Tel: 555-0148" =~ /(\d{3}-\d{4})/ <— храняется в переменной «$1».
  puts "Found phone number: #{$1}"
end
```

```
Found phone number: 555-0148
```

Регулярные выражения в Ruby также являются объектами, что позволяет передавать их в аргументах методов. У строк имеется метод `sub`, который ищет совпадение регулярного выражения в строке и заменяет совпадающий фрагмент новой строкой. Следующий вызов `sub` удаляет все телефонные номера в строке:

```
puts "Tel: 555-0148".sub(/\d{3}-\d{4}/, '***-****')
```

```
Tel: ***-****
```

И это всего лишь *крошечная* доля того, на что способны регулярные выражения. Функциональности гораздо больше, чем мы могли бы здесь описать, и вполне может статься, что за всю вашу карьеру программиста вам так и не потребуется изучать все тонкости регулярных выражений. Но даже если вы ограничитесь хотя бы основными возможностями, это сэкономит вам много времени и усилий!

А если вы захотите узнать больше, обращайтесь к литературе или просмотрите описание класса `Regexp` в базовой документации Ruby.

6. Синглетные методы

Многие объектно-ориентированные языки позволяют определять методы экземпляра, доступные для *всех* экземпляров класса. Но Ruby принадлежит к числу немногочисленных языков, позволяющих определять методы экземпляра для *одного* экземпляра. Такие методы называются **синглетными**.

Имеется класс `Person` с единственным методом экземпляра `speak`. При создании экземпляра `Person` этот экземпляр не содержит ничего, кроме метода экземпляра `speak`.

```
class Person
  def speak
    puts "Hello, there!"
  end
end

person = Person.new
person.speak
```

```
Hello, there!
```

Но язык Ruby позволяет определить метод экземпляра, доступный только в одном объекте. За ключевым словом `def` указывается ссылка на объект, оператор «точка» и имя определяемого синглетного метода.

Следующий код определяет метод `fly` для объекта в переменной `superhero`. Метод `fly` может вызываться точно так же, как любой другой метод экземпляра, но доступен он будет *только* в `superhero`.

```
superhero = Person.new
def superhero.fly
  puts "Up we go!"
end
superhero.fly
```

Для объекта определяется синглетный метод с именем «fly».

```
Up we go!
```

Вызываем метод «fly».

Также возможно переопределять методы, определяемые классом, синглетными методами. Этот код переопределяет метод экземпляра `speak` из класса `Person` версией, уникальной для `superhero`:

```
def superhero.speak
  puts "Off to fight crime!"
end
superhero.speak
```

Переопределяет метод «speak» из класса Person.

```
Off to fight crime!
```

Вызываем переопределенный метод.

Эта возможность может пригодиться при написании модульных тестов, когда метод объекта *не должен* работать так, как обычно. Например, если объект всегда создает файлы для хранения вывода, но вы не хотите, чтобы он загромождал жесткий диск файлами от многочисленных запусков теста, — переопределите метод создания файла в экземпляре, задействованном в тестировании.

Синглетные методы оказываются неожиданно полезными, хотя это всего лишь один пример выдающейся гибкости Ruby!

7. Вызов любых (даже неопределенных) методов

При вызове метода экземпляра, *не определенного* для объекта, Ruby вызывает для этого объекта метод с именем `method_missing`. Версия `method_missing`, наследуемая всеми объектами от класса `Object`, просто инициирует исключение:

```
object = Object.new
object.win
```

```
undefined method `win' for #<Object:0x007fa87a8311f0> (NoMethodError)
```

Но если переопределить метод `method_missing` в классе, вы можете создавать экземпляры этого класса и вызывать для них неопределенные методы без выдачи исключений. Более того, в этих «фантомных методах» можно делать много всего интересного...

Ruby всегда передает `method_missing` как минимум один аргумент: имя вызванного метода в форме символического имени. Кроме того, возвращаемое значение `method_missing` будет интерпретироваться как возвращаемое значение фантомного метода. Например, следующий класс возвращает строку с именем неопределенного метода, вызванного для этого класса.

Параметр содержит имя вызванного метода.

```
class AskMeAnything
  def method_missing(method_name)
    "You called #{method_name.to_s}"
  end
end
```

Имя метода преобразуется из символического имени в строку.

```
object = AskMeAnything.new
p object.this_method_is_not_defined
p object.also_undefined
```

Вызов неопределенного метода...

...и еще один.

```
"You called this_method_is_not_defined"
"You called also_undefined"
```

Все аргументы, переданные неопределенному методу, передаются `method_missing`, так что мы также можем вывести информацию о них...

Первый аргумент сохраняется здесь.

Второй аргумент сохраняется здесь.

```
class AskMeAnything
  def method_missing(method_name, arg1, arg2)
    "You called #{method_name.to_s} with #{arg1} and #{arg2}."
  end
end

object = AskMeAnything.new
p object.with_args(127.6, "hello")
```

```
"You called with_args with 127.6 and hello."
```

7. Вызов любых (даже неопределенных) методов (продолжение)

Ниже приведен класс `Politician`, экземпляры которого готовы пообещать вам что угодно. Вызовите любой неопределенный метод, передайте ему аргумент — и `method_missing` выведет как имя метода, так и аргумент.

```
class Politician
  def method_missing(method_name, argument)
    puts "I promise to #{method_name.to_s} #{argument}!"
  end
end

politician = Politician.new
politician.lower("taxes")
politician.improve("education")
```

Символическое имя преобразуется в строку.

```
I promise to lower taxes!
I promise to improve education!
```

Впрочем, даже это еще не все... Помните код dRuby, приведенный несколько страниц назад? В этом коде мы создавали объект-заместитель, который вызывал методы другого объекта *по сети*?

```
require 'drb/drb'
DRb.start_service
remote_object = DRbObject.new_with_uri("druby://localhost:8787")
remote_object.push "hello", "network"
p remote_object.last
```

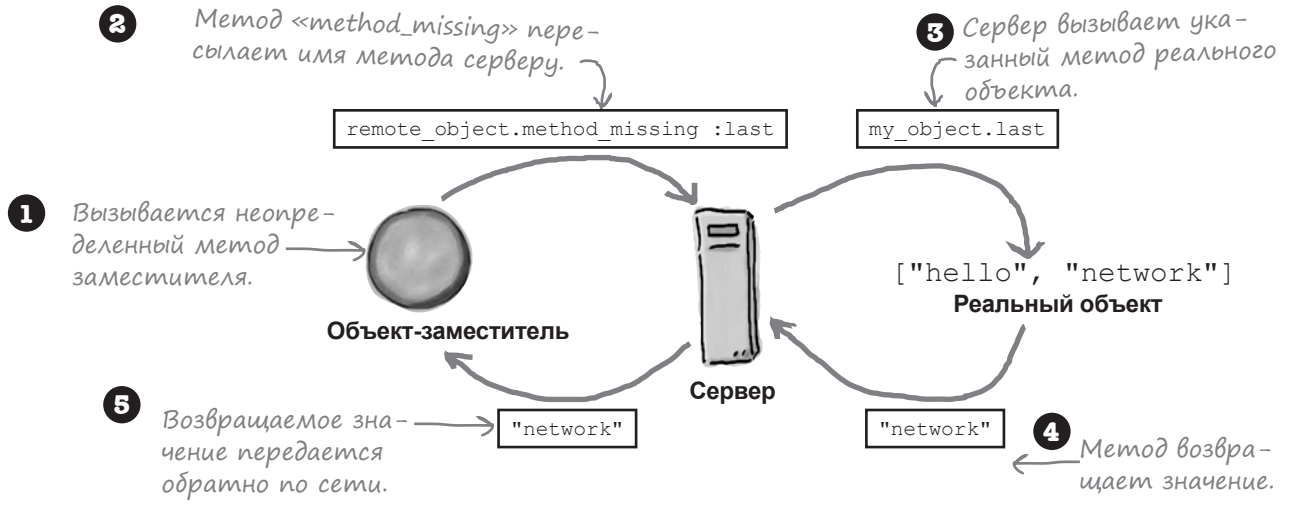
Подключаемся к удаленному серверу и получаем заместителя для массива.

Вызываем метод массива.

Вызываем другой метод массива.

Для объектов-заместителей в dRuby можно вызывать *любые методы*. Так как заместитель почти не определяет собственных методов, эти вызовы перенаправляются методу `method_missing` заместителя. Здесь имя вызванного метода и все переданные аргументы пересылаются серверу по сети.

Сервер вызывает метод для *реального* объекта и отправляет возвращаемое значение по сети, где оно возвращается из метода `method_missing` объекта-*заместителя*.



Процесс на первый взгляд сложный, но благодаря `method_missing` все сводится к простому вызову метода заместителя!

8. Rake и автоматизация задач

Вспомните, о чем говорилось в главе 13: чтобы выполнить модульные тесты в подкаталоге *lib*, нам приходилось включать в командную строку ключ `-I lib`. А также приходилось указывать файл с тестами, которые нужно было запустить...

```
File Edit Window Help
$ ruby -I lib test/test_list_with_commas.rb
Run options: --seed 18716
...
3 runs, 3 assertions, 0 failures, 0 errors, 0 skips
```

Пока неудобства невелики... А если с ростом проекта появятся десятки тестовых файлов? Выполнять их по одному станет практически нереально, а ведь тестирование — одна из задач, которые должны выполняться регулярно. Со временем также возникнет необходимость в построении документации, упаковки проекта в пакет и т. д.

В поставку Ruby входит программа Rake, которая может упростить все эти операции. Вы можете выполнить команду `rake` в терминальном окне, она ищет файл с именем «Rakefile» (без расширения) в каталоге проекта. Этот файл должен содержать код Ruby для создания *задач*, которые Rake может выполнить за вас.

Ниже приведен файл Rakefile, который создает задачу для автоматизации выполнения всех тестов. Для этого он использует класс `Rake::TestTask`, входящий в поставку Rake и предназначенный для выполнения тестов. Класс `TestTask` настроен на загрузку файлов из каталога *lib* (не нужно добавлять ключ `-I lib`) и запуск всех тестовых файлов в каталоге *test* (не нужно перечислять все тестовые файлы по одному).

```
require "rake/testtask"
```
Загружаем специализированную задачу Rake для запуска тестов.

Создается задача с именем «test».

```
Rake::TestTask.new(:test) do |t|
  t.libs << "lib"
  t.test_files = FileList['test/**/test_*.rb']
end
```
Настраивается для загрузки из каталога «lib».

Запускаются все файлы в каталоге «test».

Rakefile

После того как файл Rakefile будет сохранен в каталоге проекта, вы можете перейти в этот каталог в терминальном окне и выполнить команду `rake` с именем выполняемой задачи. В нашем примере определена только одна задача `test`, поэтому мы выполняем ее.

Переходим в основной каталог проекта.

Выполняем задачу Rake «test».

Будет выполнен каждый файл в подкаталоге «test».

```
File Edit Window Help
$ cd my_project
$ rake test
Run options: --seed 20843
...
3 runs, 3 assertions, 0 failures, 0 errors, 0 skips
```

Всего одна простая команда заставляет Rake найти и выполнить все файлы в подкаталоге *test*!

Впрочем, Rake упрощает не только запуск тестов, но и другие операции. Существуют обобщенные задачи, которые могут использоваться для выполнения любых нужных вам команд. За дополнительной информацией о Rake обращайтесь к документации стандартной библиотеки Rake.

9. Bundler

В начале главы 14 мы использовали команду `gem install sinatra` в терминальном окне для загрузки и установки пакета Sinatra. Тогда это было достаточно просто, потому что нам был нужен всего один пакет.

Но предположим, нам также понадобился пакет *i18n*, упрощающий интернационализацию (перевод приложения на другие языки), и пакет *erubis* для расширенной поддержки ERB. Вдобавок нужно проследить за тем, чтобы использовалась *более старая* версия i18n, потому что наш код зависит от метода, исключенного в обновленной версии пакета. Внезапно установка пакетов перестает казаться простым делом.

Но как обычно, находится полезный инструмент, который упростит работу: Bundler. Bundler позволяет каждому приложению в системе иметь собственный набор пакетов. Также программа помогает управлять этими пакетами, своевременно загружать и устанавливать правильные версии каждого пакета, а также предотвращает конфликты между разными версиями одного пакета.

Программа Bundler *не* включается в установку Ruby; она сама распространяется в виде пакета. Следовательно, ее сначала необходимо установить командой **`gem install bundler`** (или **`sudo gem install bundler`**, если этого требует ваша система) в терминальном окне. Команда устанавливает библиотеку Bundler и команду `bundle`, которая может выполняться в терминальном окне.

Устанавливаем Bundler.

```
File Edit Window Help
$ gem install bundler
Fetching: bundler-1.10.6.gem (100%)
Successfully installed bundler-1.10.6
...
```

Bundler берет имена устанавливаемых пакетов из файла с именем «Gemfile» (обратите внимание на сходство с «Rakefile») в папке проекта. Итак, на следующем шаге нужно создать файл Gemfile.

Как и Rakefile, файл Gemfile содержит код Ruby, но в этом файле вызываются методы библиотеки Bundler. Начнем с вызова метода `source` с URL-адресом сервера, с которого должны загружаться пакеты. Если ваша компания не использует собственный сервер пакетов, используйте сервер сообщества Ruby: `'https://rubygems.org'`. Затем для каждого устанавливаемого пакета вызывается метод `gem` с указанием имени и версии пакета в аргументах.

```
source 'https://rubygems.org'

gem "sinatra", "1.4.6"
gem "i18n", "0.6.11"
gem "erubis", "2.7.0"
```

Пакеты загружаются с сервера rubygems.org.

Используем версию 1.4.6 пакета Sinatra.

Используем версию 0.6.11 пакета i18n.

Используем версию 2.7.0 пакета Erubis.

Gemfile

После сохранения файла Gemfile перейдите в каталог проекта в терминальном окне и выполните команду **`bundle install`**. Bundler автоматически загрузит и установит конкретные версии пакетов, указанные в файле Gemfile.

Переходим в каталог проекта.

Выполняем команду «bundle install».

Bundler устанавливает пакеты, перечисленные в файле Gemfile.

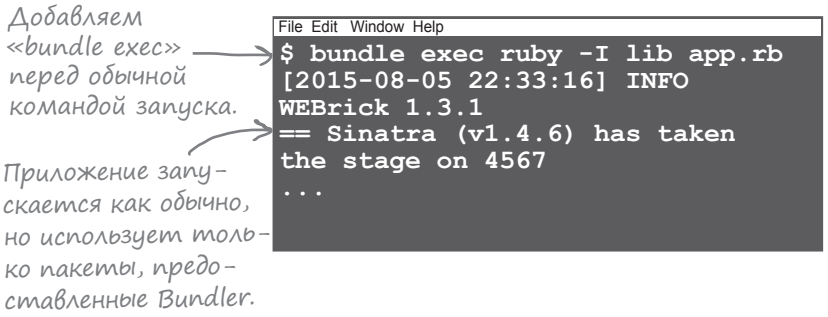
```
File Edit Window Help
$ cd my_project
$ bundle install
Fetching gem metadata from
https://rubygems.org/...
Resolving dependencies...
Installing erubis 2.7.0
Installing i18n 0.6.11
...
```

9. Bundler (продолжение)

После того как Bundler установит пакеты из файла Gemfile, вы можете обращаться к этим пакетам в своем коде так, словно вы установили их самостоятельно. Впрочем, при *запуске* приложения потребуются небольшие изменения; команда в терминальном окне должна начинаться с префикса `bundle exec`.

И вот почему: если вы *не используете* команду `bundle` для запуска приложения, то Bundler не сможет контролировать процесс выполнения. Даже в этом случае на первый взгляд все может идти нормально. Однако одна из задач Bundler — следить за тем, чтобы старые версии пакетов в системе не конфликтовали с пакетами, установленными Bundler. Без запуска Bundler такой контроль невозможен.

Итак, для надежности при использовании Bundler всегда вводите команду `bundle exec` при запуске приложения (или поищите информацию о других способах, гарантирующих запуск приложения в среде Bundler). Возможно, когда-нибудь это избавит вас от больших неприятностей!



Дополнительную информацию о Bundler можно найти по адресу: *http://bundler.io/*.

10. Литература

Эта книга о Ruby подошла к концу, но ваше путешествие в страну Ruby всего лишь начинается. Мы хотим порекомендовать пару превосходных книг, которые вам наверняка пригодятся.

Programming Ruby 1.9 & 2.0 **(Fourth Edition)**. Дэйв Томас (Dave Thomas), Чед Фаулер (Chad Fowler) и Энди Хант (Andy Hunt)

В сообществе Ruby эту книгу прозвали «книгой с киркой» из-за изображения кирки на обложке. Это известное и популярное название.

Технические книги можно условно разделить на две категории: учебники (как та книга, которую вы держите в руках) и справочники (как *Programming Ruby*). «Книга с киркой» — превосходный справочник: в ней рассматриваются все темы, для которых у нас не хватило места в этой книге. А в конце книги приводится документация по всем важным классам и модулям из стандартной библиотеки Ruby. (Так что вам даже не понадобится электронная документация HTML.)

The Well-Grounded Rubyist **(Second Edition)**. Дэвид Блэк (David A. Black)

Если вы хотите получить представление об элегантной простоте внутренних механизмов Ruby — это именно то, что вам нужно. А вы знали, что в Ruby на самом деле никаких методов класса не существует? (Так называемые «методы класса» в действительности являются синглетными методами объекта класса.) А вы знали, что классы — всего лишь модули с возможностью определения переменных экземпляра? Дэвид Блэк рассматривает аспекты языка, которые на первый взгляд кажутся сложными и непоследовательными, и раскрывает основополагающие принципы, с которыми все становится на свои места.

Благодарности

Основателям серии:

Прежде всего мы должны поблагодарить основателей серии Head First **Кэти Сьерра** (Kathy Sierra) и **Берта Бэйтса** (Bert Bates). Мне очень понравилась эта серия при первом знакомстве, которое произошло лет десять назад, но мне и в голову не приходило, что я буду для нее писать. Спасибо вам за то, что вы создали этот замечательный стиль изложения!

Берт заслуживает двойной благодарности за подробные отзывы о ранних вариантах рукописи, в которых я еще не совсем освоил Путь Head First. Берт, твои усилия сделали лучше эту книгу — и меня как автора!

Сотрудникам O'Reilly:

Я бесконечно благодарен нашему редактору **Меган Бланшет** (Meghan Blanchette). Она непреклонно работает над тем, чтобы повысить качество книг, и ее решимость поддерживала меня во время работы над вторым (и третьим) вариантом книги. Спасибо **Майку Лукидесу** (Mike Loukides) за то, что он помог мне установить первые контакты в издательстве O'Reilly, и **Кортни Нэш** (Courtney Nash), запустившей этот проект.

Также спасибо всем сотрудникам O'Reilly, благодаря которым все это стало возможным, особенно **Рэчел Монахан** (Rachel Monaghan), **Мелани Ярбро** (Melanie Yarbrough) и всей остальной производственной группе.

Научным редакторам:

Все люди совершают ошибки. К счастью, со мной работали технические рецензенты **Авди Гримм** (Avdi Grimm), **Сонда Сенгупта** (Sonda Sengupta), **Эдвард Ю Шун Вон** (Edward Yue Shung Wong) и **Оливье Лакан** (Olivier Lacan), которые помогли обнаружить все мои ошибки. Вы никогда не узнаете, сколько было таких ошибок, потому что я быстро уничтожил все доказательства. Но их помощь и отзывы были безусловно необходимы, и я никогда об этом не забуду!

Другие благодарности:

Спасибо **Райану Бенедетти** (Ryan Benedetti), который снова привел проект в движение, когда он забуксовал на ранних главах. **Дебора Робинсон** (Deborah Robinson) оказала помощь с набором текста. **Дженет Макгаврен** (Janet McGavren) и **Джон Макгаврен** (John McGavren) помогали с корректурой. **Ленни Макгаврен** (Lenny McGavren) подготовил фотографии. Спасибо читателям первого варианта книги, особенно **Эду Фреско** (Ed Fresco) и **Джону Ларкину** (John Larkin), за найденные опечатки и другие недостатки. Спасибо участникам форума **Ruby Rogues Parley** за отзывы о примерах кода.

И пожалуй, самое важное — спасибо **Кристине**, **Кортни**, **Брайану**, **Ленни** и **Джереми** за их терпение и поддержку. Я снова с ними!