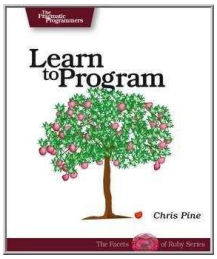
Learn to Program

улучшенная и

РАСШИРЕННАЯ

версия



« оригинальный учебник »

Крис Пайн

"Учись программировать"

(Перевод на японский - Син Нисияма.)

(Перевод на французский - Жан-Пьер Ангель.)

(Перевод на русский: HTML, TAR, PDF - Михаил В. Шохирев.)

Отправная точка для будущего программиста

Полагаю, что всё это началось ещё в 2002 году. Я думал о преподавании программирования и о том, каким превосходным языком для обучения программированию стал бы Ruby. В том смысле, что мы все были в восхищении от Ruby, потому что это мощный, элегантный и действительно просто интересный язык. И мне подумалось, что было бы здорово начать знакомство с программированием именно с него.

К сожалению, в то время было немного документации по Ruby, предназначенной для новичков. Некоторые из нашего сообщества говорили о том, что очень пригодился бы учебник наподобие "Ruby для новичка", или в более широком смысле — для обучения программированию вообще. Чем больше я думал обо всём этом, тем больше накапливалось того, что мне нужно было высказать (и это меня несколько удивило). Наконец, кто-то сказал: "Крис, почему бы тебе тогда просто не написать учебник вместо того, чтобы всё время говорить о нём?" Так я и сделал.

Но получалось не слишком хорошо. У меня была масса идей, которые были хороши *в теории*, но в реальной задаче написать классный учебник для не-программистов таился гораздо больший вызов, чем я мог осознать. (Я имею в виду, что он казался хорошим для меня, но я-то уже знал, как программировать.)

Спасло меня вот что: я устроил так, что людям было действительно легко связаться со мной, и я всегда старался помочь людям, когда они на чём-нибудь застревали. Когда я видел, что многие застревали на одном и том же месте, я переписывал его. Работы было много, но учебник потихоньку становился всё лучше и лучше.

Спустя пару лет, он был уже весьма хорош. :-) В самом деле, настолько хорош, что я был готов сказать, что он закончен, и двигаться дальше к чему-то ещё. И как раз примерно тогда появилась возможность превратить этот учебник в книгу. Поскольку он был в основном почти готов, я посчитал, что трудностей не возникнет. Я только подчищу в нём некоторые места, добавлю несколько упражнений, ну, возможно, ещё несколько упражнений, и ещё несколько глав, и дам его на просмотр ещё 50-ти рецензентам...

Это заняло у меня ещё один год, но теперь я думаю, что он действительно *в самом деле* хорош, в основном благодаря сотням храбрецов, что помогли мне написать его.

То, что размещено на этом сайте, — это исходный вариант учебника, почти совсем не изменённый с 2004 года. Чтобы получить самую свежую и улучшенную версию, вам, возможно, захочется обратиться вот к этой книге.

Соображения для преподавателей

Было несколько руководящих принципов, которых я старался придерживаться. Я думаю, они делают процесс обучения гораздо более лёгким — ведь учиться

программировать и так довольно тяжело. Если вы преподаёте или наставляете кого-то на путь благородного хакерства, то эти идеи могут помочь и вам.

Во-первых, я старался как можно больше разделять понятия так, чтобы учащемуся требовалось изучать каждый раз только одно понятие. Это было трудно сначала, но уж *слишком* легко после того, как я приобрёл некоторый опыт. Некоторые вещи должны изучаться прежде других, но я был поражён, насколько мало в действительности имеется таких иерархических зависимостей. В конечном счёте, я просто выбирал порядок изложения и старался скомпоновать материал таким образом, чтобы каждый новый раздел основывался на предыдущих.

Другой принцип, о котором я всё время помнил, — учить только одному способу делать что-либо. В этом очевидное преимущество учебника для тех, кто никогда не программировал ранее. С одной стороны, один способ делать что-то легче выучить, чем два. Хотя, возможно, более важная выгода от этого состоит в том, что, чем меньшему числу приёмов вы учите начинающего программиста, тем более изобретательным и сообразительным ему придётся быть в своих программах. Поскольку основная деятельность в программировании связана с решением задач, критически важным становится поощрять это насколько возможно на каждом этапе разработки.

Я постарался переводить понятия программирования на те понятия, которые уже есть у начинающего программиста, с целью представить идеи таким образом, чтобы нагрузка ложилась более на его интуицию, нежели на учебник. Объектно-ориентированное программирование подходит для этого весьма хорошо. Мне можно было достаточно рано начать ссылаться в учебнике на "объекты" и различные "типы объектов", невинно роняя такого рода фразы в подходящие моменты. Я не говорил что-либо подобное "всё в Ruby является объектами," или "числа и строки — это разновидности объектов", поскольку эти утверждения в действительности ничего не значат для начинающего программиста. Вместо этого я предпочёл говорить о строках (а не о "строковых объектах"), но иногда я упоминал "объекты", имея в виду просто "вот эти вещи в этих программах". А то, что все эти *вещи* в Ruby *фактически являются* объектами, позволило этим уловкам с моей стороны хорошо сработать.

Хотя я желал избежать ненужного объектно-ориентированного жаргона, я хотел быть уверенным, что если им в самом деле нужно было узнать какой-то термин, они выучат правильное слово. (Я не хотел, чтобы им пришлось учить его дважды, верно?) Вот почему я применял слово "строки", а не "текст". Методы тоже нужно было как-нибудь назвать, и я называл их "методы".

Что касается упражнений, думаю, что я приготовил несколько удачных, но их никогда не бывает слишком много. Если честно, могу поспорить, что я половину времени провёл, просто пытаясь подобрать забавные и интересные упражнения. Скучные упражнения напрочь убивают всякое желание программировать, в то время как от безупречно подходящих упражнений появляется профессиональный зуд, перед которым начинающий программист вряд ли сможет устоять. Короче говоря, невозможно потратить слишком много времени на подготовку хороших упражнений.

Об оригинальном учебнике

Страницы этого учебника (и даже эта самая страница) сгенерированы большой программой — конечно, написанной на Ruby. :-) И поэтому, в оригинальном учебнике есть несколько тонкостей. Например, все примеры программного кода в действительности исполняются каждый раз, когда вы просматриваете страницу, и показанный результат — это результат, сгенерированный при выполнении. Думаю, это лучший, самый лёгкий и, конечно, самый крутой способ удостовериться, что весь представленный мной код отрабатывает *в точности* так, как я утверждаю. Вам не нужно беспокоиться о том, что я мог ошибочно скопировать выдачу какого-нибудь из примеров или забыл протестировать какую-то часть кода: он весь проверяется каждый раз, когда вы его просматриваете. Так, в разделе по генераторам случайных чисел, если вы перезагрузите страницу, то увидите, что числа каждый раз будут изменяться... *Прекрасно*. (Я применил похожий трюк для примеров кода, когда писал книгу, но очевидно, это наиболее заметно именно в этом учебнике.)

Благодарности

И наконец, мне бы хотелось поблагодарить всех в списке рассылки ruby-talk за их мысли и ободрение; всех моих замечательных рецензентов за их помощь сделать эту книгу гораздо лучше, чем я сделал бы это один; в особенности — мою дорогую жену за то, что она была моим главным рецензентом / тестером / подопытным кроликом / музой; Мацумото — за создание этого сказочного языка программирования; издательство Pragmatic Programmers — за то, что сообщили мне об этом и, конечно, за публикацию моей книги!

Если вы заметили какие-то ошибки или опечатки, или у вас есть замечания, предложения или хорошие упражнения, которые можно добавить, пожалуйста, сообщите мне.

[А если у вас появятся замечания или уточнения по поводу перевода на русский язык, можете написать переводчику. — *Прим. перев.*]

0. Начинаем

Когда вы пишете программу для компьютера, вы должны "говорить" на языке, который ваш компьютер понимает: на языке программирования. Есть много, очень много разных языков, и многие из них — превосходные. В этом учебнике я выбрал для использования мой любимый язык программирования — *Ruby*.

Помимо того, что Ruby — мой любимый язык, это также самый лёгкий язык программирования из всех, с которыми я знаком (а знаком я весьма со многими). На самом деле, это и есть настоящая причина, по которой я пишу этот учебник: не то чтобы я решил писать учебник и остановился на Ruby как на своём любимом языке; наоборот, я нахожу Ruby настолько лёгким, что решил: обязательно должен быть хороший учебник для начинающих и непременно с его использованием. Именно простота Ruby побудила меня написать этот учебник, а вовсе не то, что это — мой любимый язык. (Чтобы написать подобный учебник на примере другого языка, такого как C++ или Java, потребовались бы сотни и сотни страниц.) Но не думайте, что Ruby — это язык для начинающих, раз он очень лёгкий! Это мощный язык программирования такого профессионального уровня, который вряд ли существовал ранее.

Когда вы что-нибудь пишете на естественном языке, написанное называется текстом. Когда вы пишете что-нибудь на компьютерном языке, написанное называется кодом. Я включил множество примеров кода на языке Ruby на протяжении учебника, большинство из них — это законченные программы, которые вы можете выполнять на своём компьютере. Чтобы легче было читать код, я окрасил части кода в различные цвета. (Например, числа всегда **зелёные**.) Всё, что предполагается вводить в программу, заключается в `белую рамку`, а всё, что выводит программа, помещается в голубую рамку.

Если вам встретится что-либо, что вы не понимаете, или у вас возникнет вопрос, на который вы не находите ответа, запишите его и продолжайте читать! Вполне возможно, что ответ найдется в одной из следующих глав. Однако, если ваш вопрос останется без ответа до последней главы, я скажу вам, куда можно отправиться, чтобы задать его. Есть множество чудесных людей, которые с желанием помогут вам; нужно только знать, где их найти.

Но сначала вам нужно загрузить и установить Ruby на ваш компьютер.

Установка под Windows

Установить Ruby под Windows легче лёгкого. Сначала вам нужно загрузить установщик Ruby. Обычно имеется пара версий на выбор; в этом учебнике используется версия 1.8.4, так что удостоверьтесь, что вы загрузили версию как минимум не старее этой. (Я бы просто взял самую свежую версию из имеющихся.) Затем просто запустите программу-установщик. Он запросит у вас, куда вы хотите установить Ruby. Если у вас нет серьёзных причин для обратного, я бы посоветовал установить его в каталог по умолчанию.

Чтобы программировать, вам нужно иметь возможность писать программы и выполнять программы. Для этого вам понадобится текстовый редактор и командная строка.

Установщик Ruby поставляется с прекрасным текстовым редактором под названием `SciTE` (Scintilla Text Editor). Вы можете запустить `SciTE`, выбрав его из меню "Пуск". Если вы хотели бы расцветить код, как в примерах этого учебника, загрузите эти файлы и поместите их в каталог редактора `SciTE` (`c:/ruby/scite`, если вы выбрали каталог по умолчанию):

- Глобальные настройки
- Настройки для Ruby

Неплохой идеей будет создать где-нибудь каталог, чтобы хранить там все ваши программы. Убедитесь, что, когда вы сохраняете программу, вы сохраняете её именно в этот каталог.

Чтобы вам добраться до командной строки, выберите "Командная строка" из подменю "Стандартные" в меню "Пуск". Вам захочется переходить в каталог, где вы храните ваши программы. Команда `cd ..` переместит вас в каталог уровнем выше, а по команде `cd foldername` вы окажетесь в каталоге под названием `foldername`. Чтобы увидеть все подкаталоги в текущем каталоге, введите команду `dir /ad`.

Вот и всё! Вы готовы, чтобы учиться программировать.

Установка под Macintosh

Если у вас стоит Mac OS X 10.2 (Jaguar), значит Ruby уже установлен в вашей системе! Что может быть проще? К сожалению, я не думаю, что вы сможете использовать Ruby под Mac OS X 10.1 или более ранней версией.

Чтобы программировать, вам нужно иметь возможность писать программы и выполнять программы. Для этого вам понадобится текстовый редактор и командная строка.

Командная строка доступна посредством терминального приложения ("Terminal application"), которое находится в разделе меню "Приложения / Утилиты" ("Applications / Utilities").

В качестве текстового редактора вы можете использовать тот, что вам более знаком или более удобен. Однако, если вы используете `TextEdit`, убедитесь, что вы сохраняете ваши программы в виде обычного текста! Иначе ваши программы *не будут работать*. Другими альтернативами для составления программ могут быть редакторы `emacs`, `vi` или `pico`: все они доступны из командной строки.

Вот и всё! Вы готовы, чтобы учиться программировать.

Установка под Linux

Во-первых, вы захотите убедиться, не был ли Ruby уже установлен у вас. Наберите на консоли команду `which ruby`. Если она выведет что-нибудь наподобие /usr/bin/which: no ruby in (...), значит вам нужно будет загрузить Ruby. Иначе посмотрите, какая версия Ruby установлена, набрав `ruby -v`. Если версия старее, чем последняя стабильная сборка с упомянутой выше страницы загрузки, то вам, возможно, захочется обновить её.

Если вы работаете как пользователь root, тогда вам, наверное, не нужны будут дальнейшие указания по установке Ruby. Если это не так, вам нужно будет попросить вашего системного администратора установить его для вас. (При этом способе установки все пользователи в этой системе смогут пользоваться Ruby.)

Или же вы можете просто установить его так, чтобы только вы могли использовать его. Переместите файл, который вы загрузили, во временный каталог, например, в `$HOME/tmp`. Если файл называется `ruby-1.6.7.tar.gz`, вы сможете открыть его командой `tar zxvf ruby-1.6.7.tar.gz`. Перейдите в каталог, который только что был вами создан (в нашем примере — в каталог `cd ruby-1.6.7`).

Сконфигурируйте вашу установку, набрав команду `./configure --prefix=$HOME`). Затем наберите на консоли команду `make`, которая соберёт ваш интерпретатор Ruby. Это может занять несколько минут. После того, как всё будет выполнено, напечатайте `make install`, чтобы установить его.

Потом вы пожелаете добавить каталог `$HOME/bin` в список путей для поиска команд, отредактировав ваш файл `$HOME/.bashrc` с настройками командного интерпретатора. (Вам нужно будет выйти из системы и зайти снова, чтобы изменения вступили в силу.) После того, как вы сделали это, проверьте успешность установки командой `ruby -v`. Если она сообщит, какая версия Ruby у вас установлена, тогда вы можете удалить файлы в каталоге `$HOME/tmp` (или там, куда вы их разместили).

Вот и всё! Вы готовы, чтобы учиться программировать.

1. Числа

А сейчас, когда вы всё установили, давайте напишем программу! Откройте ваш любимый текстовый редактор и наберите на клавиатуре следующее:

```
puts 1+2
```

Сохраните вашу программу (да, это программа!) под именем `calc.rb` (**.rb** — это то, что мы обычно пишем в конце имени программы, написанной на Ruby). А теперь выполните вашу программу, набрав `ruby calc.rb` в командной строке. Она должна вывести на экране число 3. Видите, программировать не так уж трудно, не правда ли?

Знакомство с puts

Так что же происходит в этой программе? Я уверен, что вы смогли догадаться, что делает **1+2**, и наша программа, в принципе, такая же, как эта:

```
puts 3
```

слово **puts** просто выводит на экран всё, что следует за ним.

Целые и плавающие

В большинстве языков программирования (и Ruby не является исключением) числа без десятичной точки называются целыми, а числа с десятичной точкой обычно называются числами с плавающей точкой или проще — плавающими. [По сути, это дробные десятичные числа, представленные в компьютере в двоичном виде. — *Прим. перев.*]

Вот несколько целых чисел:

```
5

-205

99999999999999999999999999

0
```

А вот несколько чисел с плавающей точкой:

```
54.321

0.001

-205.3884
```

```
0.0
```

На практике в большинстве программ плавающие числа не используются, только целые. (В конце концов, никто не хочет прочитать 7.4 сообщений электронной почты, просмотреть 1.8 web-страниц или послушать 5.24 любимых песен...) Числа с плавающей точкой больше используются для научных целей (физических экспериментов и тому подобное) и для 3D-графики. Даже большинство программ для денежных расчётов применяют целые числа: в них просто учитываются суммы в копейках! [в оригинале: "в пенсах"— *Прим. перев.*]

Простая арифметика

К этому моменту у нас есть всё необходимое, чтобы сделать простой калькулятор. (В калькуляторах всегда используются плавающие числа, поэтому если вы хотите, чтобы ваш компьютер просто работал как калькулятор, вы также должны использовать числа с плавающей точкой.) Для сложения и вычитания мы используем `+` и `-`, что мы уже видели в программе. Для умножения мы используем `*`, а для деления мы используем `/`. На большинстве клавиатур эти клавиши располагаются среди цифровых клавиш в правой её части. Если у вас уменьшенная клавиатура или ноутбук, то вы можете просто нажать клавиши `Shift` и `8` или `/` (та же клавиша, что и клавиша `?`). Давайте попробуем немного расширить нашу программу calc.rb. Наберите на клавиатуре, а затем выполните следующее:

```
puts 1.0 + 2.0

puts 2.0 * 3.0

puts 5.0 - 8.0

puts 9.0 / 2.0
```

Вот что возвращает эта программа:

```
3.0

6.0

-3.0

4.5
```

(Пробелы в программе не важны; они просто делают код легче для чтения.) Что ж, в том, что мы увидели, нет ничего удивительного. А теперь давайте испробуем на целых числах:

```
puts 1+2

puts 2*3

puts 5-8

puts 9/2
```

В основном, то же самое, правда?

```
3

6

-3

4
```

Хм... кроме последнего! Но когда вы выполняете арифметические действия с целыми числами, вы получаете целочисленные ответы. Когда ваш компьютер не может получить "правильный" ответ, он всегда округляет его. (Конечно, 4 и *есть* правильный ответ в целочисленной арифметике для выражения `9/2`; хотя, возможно, это не тот ответ, который вы ожидали.)

Возможно, вы недоумеваете, зачем может понадобиться целочисленное деление. Ну, скажем, вы собираетесь в кино, но у вас только 9 долларов. У нас в Портленде вы можете посмотреть фильм в кинотеатре "Багдад" за 2 бакса. Сколько фильмов вы сможете посмотреть там? `9/2`... 4 фильма. 4.5 — это в данном случае, конечно, *неправильный* ответ; вам не позволят посмотреть полфильма и не позволят половине вас посмотреть целый фильм... Некоторые вещи нельзя разделить.

А теперь поэкспериментируйте сами с несколькими программами! Если вы хотите применить более сложные выражения, можете использовать скобки. Например:

```
puts 5 * (12-8) + -15

puts 98 + (59872 / (13*8)) * -52
```

```
5

-29802
```

Попробуйте ещё кое-что

Напишите программу, которая сообщит вам:

- сколько часов в году?
- сколько минут в десятилетии?
- ваш возраст в секундах?
- сколько шоколадок вы надеетесь съесть за свою жизнь?
  Предупреждение: *Вычисления в этой части программы могут потребовать очень много времени!*
- А вот задание потруднее: Если я прожил 943 миллиона секунд, то каков мой возраст?

Когда вы закончите развлекаться с числами, давайте взглянем на буквы.

2. Буквы

Итак, мы узнали всё о числах, а как же насчёт букв? слов? текста?

Группы букв в программе мы называем строками. (Вы можете считать, что напечатанные буквы нанизаны друг за другом на бечёвку, как флажки.) Чтобы было легче увидеть, какие части кода — строки, я буду выделять их **красным** цветом. Вот несколько строк:

```
'Привет.'

'Ruby - потрясающий.'

'5 - это моё любимое число... а какое ваше?'

'Снупи восклицает: "#%^?&*@!", когда он запинается.'

'     '

''
```

Как видите, строки могут содержать знаки пунктуации, цифры, символы и пробелы... не только одни буквы. А в последней строке совсем ничего нет; такую строку называют пустой строкой.

Мы использовали команду **puts**, чтобы напечатать числа; давайте опробуем её с несколькими строками:

```
puts 'Привет, мир!'

puts ''

puts 'До свидания.'
```

```
Привет, мир!

До свидания.
```

Всё отлично отработало. А теперь попробуйте вывести несколько своих строк.

Строковая арифметика

Точно так же, как вы выполняли арифметические действия с числами, вы можете выполнять арифметические действия со строками! Ну, почти так же... Во всяком случае, вы можете складывать строки. Давайте попробуем сложить две строки и посмотреть, что нам покажет **puts**.

```
puts 'Я люблю' + 'яблочный пирог.'
```

```
Я люблюяблочный пирог.
```

Опаньки! Я забыл поместить пробел между `'Я люблю'` и `'яблочный пирог.'`. Пробелы обычно не имеют значения, но они имеют значение внутри строк. (Правду говорят: "компьютеры делают не то, что вы *хотите*, чтобы они делали, а только то, что вы *велите* им делать".) Давайте попробуем снова:

```
puts 'Я люблю ' + 'яблочный пирог.'

puts 'Я люблю' + ' яблочный пирог.'
```

```
Я люблю яблочный пирог.

Я люблю яблочный пирог.
```

(Как видите, неважно, к какой из строк добавлять пробел.)

Итак, вы можете складывать строки, но вы также можете умножать их! (На число, во всяком случае.) Посмотрите:

```
puts 'миг ' * 4
```

```
мигаю глазами
```

(Шучу, конечно... на самом деле, она выводит вот что:)

```
миг миг миг миг
```

Если подумать, это имеет безупречный смысл. В конце концов, `7*3` действительно означает `7+7+7`, поэтому `'му'*3` означает в точности `'му'+'му'+'му'`.

`12` или `'12'`

Прежде, чем мы двинемся дальше, мы должны убедиться, что понимаем различие между *числами* и *цифрами*. `12` это число, а `'12'` это — строка из двух цифр.

Давайте немного поиграем с этим:

```
puts  12  +  12

puts '12' + '12'

puts '12  +  12'
```

```
24

1212

12  +  12
```

А как насчёт этого:

```
puts  2  *  5

puts '2' *  5
```

```
puts '2  *  5'
```

```
10

22222

2  *  5
```

Эти примеры были довольно простыми. Однако, если вы не будете очень осторожными, смешивая в своей программе строки и числа, у вас могут возникнуть...

Сложности

К этому моменту вы может быть попробовали некоторые вещи, которые *не желают* работать. Если ещё нет, то вот несколько из них:

```
puts '12' + 12

puts '2' * '5'
```

```
#<TypeError: can't convert Fixnum into String>
```

[#<Ошибка типа: невозможно преобразовать Целое к Строке> — *Прим. перев.*] Мммдааа... сообщение об ошибке. Дело в том, что вы действительно не можете прибавить число к строке или умножить строку на другую строку. В этом не больше смысла, чем вот в таких выражениях:

```
puts 'Бетти' + 12

puts 'Фред' * 'Джон'
```

Вот ещё чего следует остерегаться: вы можете написать в программе `'поросёнок'*5`, поскольку это просто обозначает `5` экземпляров строки `'поросёнок'`, соединённых вместе. Однако, вы *не можете* написать `5*'поросёнок'`, поскольку это обозначает `'поросёнок'` экземпляров числа `5`, что просто-напросто глупо.

И, наконец: а что, если я хочу, чтобы программа напечатала 'Ты шикарная!'? Можно попробовать так:

```
puts ''Ты шикарная!''
```

Ну так вот, *это* не будет работать; я даже не буду пробовать выполнить это. Компьютер подумал, что мы закончили строку. [После второго апострофа. — *Прим. перев.*] (Вот почему хорошо иметь текстовый редактор, который расцвечивает синтаксис для вас.) Так как же дать понять компьютеру, что мы желаем остаться внутри строки? Нам нужно экранировать символ апострофа вот так:

```
puts '\'Ты шикарная!\''
```

```
'Ты шикарная!'
```

Обратная косая черта — это символ экранирования (escape character). Другими словами, если в строке стоит обратная черта и другой символ [которые образуют так

называемую "escape-последовательность". — *Прим. перев.*], то они оба иногда трансформируются в один новый символ. Но две вещи, которые обратная черта всё-таки экранирует, это апостроф и сама обратная черта. (Если хорошенько подумать, то символы экранирования должны всегда экранировать себя.) Ещё несколько примеров, я думаю, будут здесь к месту:

```
puts '\'Ты шикарная!\''

puts 'обратная черта в конце строки:  \\'

puts 'вверх\\вниз'

puts 'вверх\вниз'
```

```
'Ты шикарная!'

обратная черта в конце строки:  \

вверх\вниз

вверх\вниз
```

Поскольку обратная черта *не* экранирует `'в'`, но *экранирует* себя, две последних строки идентичны. Они не выглядят одинаковыми в коде программы, но в вашем компьютере они действительно одинаковы.

Если у вас есть другие вопросы, просто продолжайте читать дальше! В конце концов, я не могу отвечать на каждый вопрос на *этой* странице.

3. Переменные и присваивание

До сих пор, каждый раз, когда мы выводили с помощью **puts** строку или число, всё, что мы выводили, исчезало. Я имею в виду, если мы хотели напечатать что-то дважды, мы должны были вводить это дважды с клавиатуры:

```
puts '...ты можешь сказать это снова...'

puts '...ты можешь сказать это снова...'
```

```
...ты можешь сказать это снова...

...ты можешь сказать это снова...
```

Вот было бы прекрасно, если бы мы могли просто ввести что-то один раз и потом обращаться к нему... сохранить его где-нибудь. Ну, конечно же, мы можем это сделать, иначе я бы не заговорил об этом!

Чтобы сохранить строку в памяти вашего компьютера, нам нужно дать строке какое-то имя. Программисты часто говорят об этом процессе как о присваивании, а имена они называют переменными. Имя переменной — это просто любая последовательность латинских букв и цифр, но первый символ должен быть буквой в нижнем регистре. Давайте снова испробуем нашу последнюю программу, но на этот раз я дам строке имя **myString** (хотя я с таким же успехом мог назвать её **str** или **myOwnLittleString** или **henryTheEighth**).

```
myString = '...ты можешь сказать это снова...'

puts myString

puts myString
```

```
...ты можешь сказать это снова...

...ты можешь сказать это снова...
```

Каждый раз, когда вы пытаетесь что-нибудь сделать с **myString**, программа вместо этого делает это со строкой **'...ты можешь сказать это снова...'**. Вы можете думать о переменной **myString** как об "указателе на" строку **'...ты можешь сказать это снова...'**. А вот немного более интересный пример:

```
name = 'Патриция Розанна Джессика Милдред Оппенгеймер'

puts 'Меня зовут ' + name + '.'

puts 'Ого! ' + name + ' - это правда длинное имя!'
```

```
Меня зовут Патриция Розанна Джессика Милдред Оппенгеймер.

Ого! Патриция Розанна Джессика Милдред Оппенгеймер - это правда длинное
имя!
```

Кроме того: так же просто, как мы можем *присвоить* объект переменной, мы можем *переприсвоить* другой объект этой же переменной. (Вот почему мы называем их переменными: потому что то, на что они указывают, может изменяться.)

```
composer = 'Моцарт'

puts composer + ' был "сенсацией" в своё время.'

composer = 'Бетховен'

puts 'Но мне лично больше нравится ' + composer + '.'
```

```
Моцарт был "сенсацией" в своё сремя.

Но мне лично больше нравится Бетховен.
```

Конечно, переменные могут указывать на объект любого типа, а не только на строки:

```
var = 'ещё одна ' + 'строка'

puts var

var = 5 * (1+2)

puts var
```

```
ещё одна строка

15
```

Фактически переменные могут указывать почти на всё... кроме других переменных. А что же произойдёт, если мы попробуем?

```
var1 = 8

var2 = var1

puts var1

puts var2

puts ''

var1 = 'восемь'

puts var1

puts var2
```

```
8

8

восемь

8
```

Сначала, когда мы попытались сделать, чтобы переменная **var2** указывала на **var1**, она в действительности стала вместо этого указывать на **8** (на то же число, на которое

указывала переменная **var1**). Затем мы сделали так, чтобы **var1** стала указывать на **'восемь'**, но поскольку **var2** никогда не указывала на **var1**, она продолжает указывать на **8**.

Ну а теперь, когда у нас есть переменные, числа и строки, давайте научимся, собирать их все вместе!

4. Собираем всё вместе

Мы рассмотрели несколько различных видов объектов (числа и буквы), мы создавали переменные, указывающие на них; а следующее, что мы хотим сделать — заставить их всех дружно работать вместе.

Мы уже знаем, что если мы хотим, чтобы программа напечатала 25, то следующий код **не** *будет* работать, поскольку нельзя складывать числа и строки:

```
var1 = 2

var2 = '5'

puts var1 + var2
```

Частично трудность заключается в том, что ваш компьютер не знает, пытаетесь ли вы получить 7 (**`2 + 5`**), или вы хотите получить 25 (**`'2' + '5'`**).

Прежде чем мы сможем сложить их вместе, нам нужно каким-то образом получить строковую версию значения **`var1`** или целочисленную версию значения **`var2`**.

Преобразования

Чтобы получить строковую версию объекта, мы просто записываем после него **`.to_s`**:

```
var1 = 2

var2 = '5'

puts var1.to_s + var2
```

```
25
```

Подобным же образом **`to_i`** возвращает целочисленную версию значения объекта, а **`to_f`** возвращает плавающую версию. Давайте взглянем на то, что эти три метода делают (и что *не* делают) чуть более пристально:

```
var1 = 2

var2 = '5'

puts var1.to_s + var2

puts var1 + var2.to_i
```

```
25

7
```

Обратите внимание, что даже после того, как мы получили строковую версию **`var1`**,

вызвав **to_s**, переменная **var1** продолжает постоянно указывать на **2**, и никогда — на **'2'**. До тех пор, пока мы явно не переприсвоим значение **var1** (что требует применение знака =), она будет указывать на **2** до конца работы программы.

А теперь давайте испробуем несколько более интересных (и несколько просто-таки странных) преобразований:

```
puts '15'.to_f

puts '99.999'.to_f

puts '99.999'.to_i

puts ''

puts '5 - это моё любимое число!'.to_i

puts 'Кто вас спрашивал о 5 или о чём-нибудь подобном?'.to_i

puts 'Ваша мамочка.'.to_f

puts ''

puts 'строковое'.to_s

puts 3.to_i
```

```
15.0

99.999

99

5

0

0.0

строковое

3
```

Итак, это, возможно, вызвало у вас некоторое удивление. Первый пример весьма обычен и даёт в результате 15.0. После этого мы преобразовали строку **'99.999'** в число с плавающей точкой и в целое. Плавающее получилось таким, как мы и ожидали; целое было, как всегда, округлено.

Далее, у нас было несколько примеров, когда несколько ... *необычных* строк преобразовывались в числа. **to_i** игнорирует всё, начиная с первой конструкции, которая не распознана как число, и далее до конца строки. Таким образом, первая строка была преобразована в **5**, а остальные, поскольку они начинались с букв, были полностью проигнорированы, так что компьютер вывел только ноль.

Наконец, мы увидели, что последние два наших преобразования не изменили ровным счётом ничего — в точности, как мы и предполагали.

Другой взгляд на puts

Есть что-то странное в нашем любимом методе... Посмотрите-ка вот на это:

```
puts 20

puts 20.to_s

puts '20'
```

```
20

20

20
```

Почему эти три вызова puts выводят одно и то же? Ну, положим, последние два так и должны выводить, поскольку **`20.to_s`** *и есть* **`'20'`**. А как насчёт первого, с целым числом **`20`**? Собственно говоря, что же это значит: написать *целое число* 20? Когда вы пишете *2*, а затем *0* на листе бумаги, вы записываете строку, а не целое. *Число* 20 — это количество пальцев у меня на руках и ногах; это не *2* с последующим *0*.

Что ж, у нашего знакомого, **`puts`**, есть большой секрет: прежде, чем метод **`puts`** пытается вывести объект, он использует **`to_s`**, чтобы получить строковую версию этого объекта. Фактически, *s* в слове **`puts`** обозначает *string*; **`puts`** на самом деле значит put string ["вывести строку" — *Прим. перев.*].

Сейчас это может показаться не слишком впечатляющим, но в Ruby есть много, *много* разнообразных объектов (вы даже научитесь создавать собственные объекты!), и было бы неплохо узнать, что произойдёт, если вы попытаетесь вывести с помощью **`puts`** действительно причудливый объект, например, фотографию вашей бабушки, музыкальный файл или что-нибудь ещё. Но об этом позже...

А тем временем, у нас есть для вас ещё несколько методов, и они позволят нам писать разнообразные забавные программы...

Методы gets и chomp

Если **`puts`** обозначает put string, уверен, что вы догадаетесь, что обозначает **`gets`**. И так же, как **`puts`** всегда "выплёвывает" строки, **`gets`** считывает только строки. А откуда он берёт их?

От вас! Ну, по крайней мере, с вашей клавиатуры. Так как ваша клавиатура может производить только строки, всё это прекрасно работает. В действительности происходит вот что: **`gets`** просто сидит себе и считывает всё, что вы вводите, пока вы не нажмёте `Enter`. Давайте попробуем:

```
puts gets
```

```
Здесь есть эхо?
```

```
Здесь есть эхо?
```

Конечно же, что бы вы ни вводили с клавиатуры, будет просто выведено вам обратно. Выполните пример несколько раз и попробуйте вводить разные строки.

Теперь мы можем делать интерактивные программы! Например, эта, когда вы введёте своё имя, поприветствует вас:

```
puts 'Приветик, ну и как Вас зовут?'

name = gets

puts 'Вас зовут ' + name + '?  Какое прекрасное имя!'

puts 'Рад познакомиться с Вами, ' + name + '.  :)'
```

Ммм-да! Я только что выполнил её — я ввёл своё имя, и вот что получилось:

```
Приветик, ну и как Вас зовут?

Chris

Вас зовут Chris

?  Какое прекрасное имя!

Рад познакомиться с Вами, Chris

.  :)
```

Хммм... похоже, когда я ввел буквы "`К`", "`р`", "`и`", "`с`" и затем нажал на `Enter`, **`gets`** воспринял все буквы моего имени *и* символ `Enter`! К счастью, имеется метод как раз для подобных случаев: **`chomp`**. Он убирает все символы `Enter`, которые "болтаются" в конце вашей строки. Давайте снова проверим эту программу, но на сей раз призовём на помощь **`chomp`**:

```
puts 'Приветик, ну и как Вас зовут?'

name = gets.chomp

puts 'Вас зовут ' + name + '?  Какое прекрасное имя!'

puts 'Рад познакомиться с Вами, ' + name + '.  :)'
```

```
Приветик, ну и как Вас зовут?

Chris

Вас зовут Chris?  Какое прекрасное имя!

Рад познакомиться с Вами, Chris.  :)
```

Гораздо лучше! Обратите внимание, что поскольку **`name`** указывает на **`gets.chomp`**, нам не требуется писать **`name.chomp`**; значение **`name`** уже было **`chomp`**-нуто.

Попробуйте ещё кое-что

• Напишите программу, которая спрашивает у человека имя, затем отчество, затем фамилию. В результате она должна поприветствовать человека, называя его полным именем.

• Напишите программу, которая спрашивает у человека его любимое число. Пусть ваша программа прибавит единицу к этому числу, а затем предложит результат в качестве *большего и лучшего* любимого числа. (Однако будьте при этом тактичными.)

После того, как вы закончите эти две программы (и любые другие, которые вы пожелаете попробовать), давайте изучим ещё несколько методов (и узнаем о них ещё что-нибудь).

5. Ещё немного о методах

Пока что мы видели несколько различных методов: **puts** и **gets** и так далее (**Быстрый тест:** *Перечислите все методы, которые мы узнали до сих пор! Их десять; ответ приводится ниже.*), но мы совсем не говорили о том, что из себя представляют методы. Мы знаем, что они делают, но мы не знаем, "что они такое".

И вот что они *есть* на самом деле: нечто, которое выполняет что-либо. Если объекты (такие как строки, целые и плавающие числа) являются существительными в языке Ruby, то методы подобны глаголам. И совсем также, как в английском [и в русском — *Прим. перев.*] языке, вы не используете глагол без существительного, чтобы *выполнить* действие, обозначаемое глаголом. Например, тиканье не совершается само по себе; настенные часы (или наручные или что-нибудь ещё) должны производить его. На естественном языке мы бы сказали: "Часы тикают." На Ruby мы бы сказали **clock.tick** (естественно, предполагая, что **clock** — это объект Ruby). Программисты могли бы сказать, что мы "вызвали метод **tick** объекта **clock**" или что мы "вызвали **tick** у **clock**".

Ну что, вы выполнили тест? Хорошо. Что ж, я уверен, что вы вспомнили методы **puts**, **gets** и **chomp**, так как мы только что разобрали их. Вы, возможно, также усвоили наши методы преобразования: **to_i**, **to_f** и **to_s**. Однако, знаете ли вы остальные четыре? Ну конечно же, это не что иное, как старые добрые арифметические действия: **+**, **-**, ***** и **/** !

Как я уже говорил ранее, также как каждому глаголу нужно существительное, так и каждому методу требуется объект. Обычно легко сказать, какой объект выполняет метод: тот, что стоит непосредственно перед точкой, как в примере с **clock.tick** или в **101.to_s**. Иногда же это не столь очевидно, например, в арифметических методах. Выясняется, что **5 + 5** это на самом деле просто сокращённый способ записи **5.+ 5**. Например:

```
puts 'привет '.+ 'мир'

puts (10.* 9).+ 9
```

```
привет мир

99
```

Выглядит не слишком привлекательно, поэтому мы больше не будем записывать методы в таком виде. Но нам ведь важно понимать, что же происходит *в действительности*. (На моей машине, эта программа также выдаёт мне такое предупреждение:

warning: parenthesize argument(s)for future_version
[предупреждение: заключите аргумент(ы) в скобки для будущих версий — *Прим. перев.*].

Этот код прекрасно выполнился, но мне было сказано, что возникли трудности при выяснении, что я имею в виду, поэтому на будущее рекомендуется использовать дополнительно скобки.) И это также даёт нам более глубокое понимание, почему мы можем выполнить `'pig'*5`, но не можем выполнить `5*'pig'`: `'pig'*5` указывает `'pig'` выполнить умножение, а `5*'pig'` предписывает числу `5` выполнить умножение. Строка `'pig'` знает, как сделать `5` собственных копий и объединить их вместе; однако, числу `5` будет затруднительно сделать `'pig'` копий *самого себя* и сложить их вместе.

И, конечно, нам всё ещё нужно выяснить про **`puts`** и **`gets`**. Где же их объекты? В английском [и в русском — *Прим. перев.*] языке, вы можете иногда опустить существительное; например, если злодей завопит "Умри!", неявным существительным будет тот, кому он кричит. В Ruby, если я говорю **`puts 'быть или не быть'`**, на самом деле я говорю: **`self.puts 'быть или не быть'`**. Но что же такое **`self`**? Это специальная переменная, которая указывает на тот объект, в котором вы находитесь. Мы пока что не знаем, как находиться *внутри* объекта, но покуда мы это не выяснили, мы всегда будем находиться в большом объекте, которым является... вся наша программа! И к счастью для нас, у этой программы есть несколько собственных методов, наподобие **`puts`** и **`gets`**. Посмотрите:

```
iCantBelieveIMadeAVariableNameThisLongJustToPointToA3 = 3

puts iCantBelieveIMadeAVariableNameThisLongJustToPointToA3

self.puts iCantBelieveIMadeAVariableNameThisLongJustToPointToA3
```

```
3

3
```

Если вы не совсем въехали во всё это, это нормально. Самое важное, что нужно из всего этого уяснить, это то, что каждый метод выполняется некоторым объектом, даже если перед ним не стоит точка. Если вы понимаете это, то вы вполне готовы двигаться дальше.

Забавные строковые методы

Давайте изучим несколько забавных строковых методов. Вам не нужно их все запоминать; достаточно просто ещё раз взглянуть на эту страницу, если вы их позабудете. Я только хочу показать вам *небольшую* часть того, что могут делать строки. На самом деле, я и сам не могу запомнить даже половины строковых методов — но это нормально, потому что в Интернете есть замечательные справочники, где перечислены и объяснены все строковые методы. (Я покажу вам, где их найти в конце этого учебника.) Серьёзно, я даже *не хочу* знать все строковые методы: это всё равно, что знать каждое слово в словаре. Я прекрасно могу говорить по-английски, не зная всех слов в словаре... и ведь не в этом же заключается сам смысл словаря? Вам ведь не *требуется* знать всё, что в нём содержится?

Итак, наш первый строковый метод это **`reverse`**, который выдаёт значение строки, перевёрнутое задом наперёд:

```
var1 = 'барк'

var2 = 'телекс'

var3 = 'Вы можете произнести это предложение наоборот?'

puts var1.reverse

puts var2.reverse

puts var3.reverse

puts var1

puts var2

puts var3
```

```
краб

скелет

?торобоан еинежолдерп отэ итсензиорп етежом ыВ

барк

телекс

Вы можете произнести это предложение наоборот?
```

Как видите, **`reverse`** не переворачивает значение *исходной* строки, он просто создаёт её новую перевёрнутую копию. Вот почему в **`var1`** по-прежнему содержится **`'барк'`** даже после того, как мы вызвали **`reverse`** у **`var1`**.

Другой строковый метод это **`length`**, который сообщает нам количество символов (включая пробелы) в строке:

```
puts 'Как Ваше полное имя?'

name = gets.chomp

puts 'Вы знаете, что Ваше имя состоит из '+name.length+' символов,
'+name+'?'
```

```
Как Ваше полное имя?

Christopher David Pine

#<TypeError: can't convert Fixnum into String>
```

Ой-ё-ёй! Что-то не сработало, и, кажется, это случилось где-то после строки **`name = gets.chomp`**... Вы понимаете, в чём дело? Поглядим, сможете ли вы разобраться с этим.

Причина заморочки — в методе **`length`**: он выдаёт нам число, а нам нужна строка. Исправить это довольно просто: мы только воткнём **`to_s`** (и скрестим пальцы на удачу):

```
puts 'Как Ваше полное имя?'

name = gets.chomp

puts 'Вы знаете, что Ваше имя состоит из '+name.length.to_s+' символов,
'+name+'?'
```

```
Как Ваше полное имя?

Christopher David Pine

Вы знаете, что Ваше имя состоит из 22 символов, Christopher David Pine?
```

Нет, я этого не знал. **Внимание:** это количество *символов* в моём имени, а не количество *букв* (сосчитайте их). Думаю, мы могли бы написать программу, которая спрашивает ваше имя, отчество и фамилию по отдельности, а затем складывает их длины... эй, почему бы вам это не сделать? Давайте, я подожду.

И что, сделали? Хорошо! Нравится программировать, не так ли? А вот после нескольких следующих глав вы сами изумитесь тому, что вы сможете делать.

Итак, есть ещё несколько строковых методов, которые изменяют регистр букв (заглавных или строчных) в вашей строке. **upcase** изменяет каждую строчную букву на заглавную, а **downcase** изменяет каждую заглавную букву на строчную. **swapcase** переключает регистр каждой буквы в строке, и наконец, **capitalize** работает совсем как **downcase** за исключением того, что он переводит первую букву в заглавную (если это буква).

```
letters = 'aAbBcCdDeE'

puts letters.upcase

puts letters.downcase

puts letters.swapcase

puts letters.capitalize

puts ' a'.capitalize

puts letters
```

```
AABBCCDDEE

aabbccddee

AaBbCcDdEe

Aabbccddee

 a

aAbBcCdDeE
```

Довольно обычные средства. Как видно из строки **puts ' a'.capitalize**, метод **capitalize** переводит в заглавную только первый *символ*, а не первую *букву*. И также, как мы уже видели раньше, при вызове всех этих методов, значение **letters** остаётся неизменным. Мне не хотелось бы слишком вас мучить этим, но это важно

понимать. Есть ещё несколько методов, которые *действительно* изменяют ассоциированные с ними объекты, но мы их пока что не видели и не увидим ещё некоторое время.

[Чтобы преобразовывать строки с русскими буквами потребуется установить одну из дополнительных библиотек, например, **active_support**, и тогда можно будет воспользоваться методами **"строка".chars.downcase**, **"строка".chars.upcase**, **"строка".chars.capitalize**. — *Прим. перев.*]

Остальные из этих забавных строковых методов, рассматриваемых нами, предназначены для визуального форматирования. Первый из них, **center**, добавляет пробелы в начало и в конец строки, чтобы отцентрировать её. Однако, также как вам требовалось указать методу **puts**, что вы хотите напечатать, а методу +, что вы хотите сложить, вам нужно указать методу **center**, какой ширины должна быть ваша отцентрированная строка. Так, если бы мне захотелось расположить по центру строки стихотворения, я бы сделал это примерно так [вместо оригинального английского шестистишия приводится другой подходящий лимерик Эдварда Лира в переводе Е. В. Клюева — *Прим. перев.*]:

```
lineWidth = 50

puts(  'Вот вам юная мисс из России:'.center(lineWidth))

puts(   'Визг её был ужасен по силе.'.center(lineWidth))

puts(        'Он разил, как кинжал,-'.center(lineWidth))

puts(          'Так никто не визжал,'.center(lineWidth))

puts('Как визжала та Мисс из России.'.center(lineWidth))
```

```
            Вот вам юная мисс из России:

            Визг её был ужасен по силе.

               Он разил, как кинжал,-

                так никто не визжал,

           Как визжала та Мисс из России.
```

Хммм... Не думаю, что этот детский стишок звучит именно так, но мне просто лень уточнить по книге. (Кроме того, я хотел выровнять части строк программы, где встречается **.center(lineWidth)**, поэтому я вставил эти лишние пробелы перед строками. Это сделано просто потому, что мне кажется, что так красивее. У программистов часто вызывает сильные чувства обсуждение того, что в программе является красивым, и они часто расходятся во мнениях об этом. Чем больше вы программируете, тем больше вы следуете своему собственному стилю.) Что касается разговоров о лени, то лень в программировании — это не всегда плохо. Например, видите, что я сохранил ширину стихотворения в переменной **lineWidth**? Это для того, чтобы, если я захочу позже вернуться к программе и сделать стих шире, мне нужно будет изменить только самую верхнюю строку программы вместо того, чтобы менять каждую из строк, где есть центрирование. При достаточно длинном стихотворении это может сэкономить мне немало времени. Вот такая разновидность лени — это действительно добродетель в программировании. [Однако автор не

поленился назвать эту переменную достаточно длинным, но зато осмысленным именем **`lineWidth`**, так как он сознаёт, насколько важно для читающего программу правильно понимать назначение переменных. — *Прим. перев.*]

Так вот, о центрировании... Как вы могли заметить, оно не столь прекрасно, как его мог бы сделать текстовый процессор. Если вы действительно хотите идеальное центрирование (и, возможно, более симпатичный шрифт), тогда вы просто должны использовать текстовый процессор! Ruby — это удивительный инструмент, но нет ни одного инструмента идеального для *любой* работы.

Два других метода форматирования строк — это **`ljust`** и **`rjust`**, названия которых обозначают left justify (выровнять влево) и right justify (выровнять вправо). Они похожи на **`center`** за исключением того, что они добавляют к строке пробелы соответственно с левой или с правой стороны. Давайте посмотрим все три метода в действии:

```
lineWidth = 40

str = '--> текст <--'

puts str.ljust  lineWidth

puts str.center lineWidth

puts str.rjust  lineWidth

puts str.ljust (lineWidth/2) + str.rjust (lineWidth/2)
```

```
--> текст <--

               --> текст <--

                           --> текст <--

--> текст <--              --> текст <--
```

Попробуйте ещё кое-что

• Напишите программу "Злой Начальник". Он должен грубо спрашивать, чего вы хотите. Что бы вы ему ни ответили, Злой Начальник должен орать вам это же самое в ответ, а затем увольнять вас. Например, если вы введёте: `Я хочу повышения зарплаты.`, он должен прокричать в ответ: ЧТО ЗНАЧИТ: "Я ХОЧУ ПОВЫШЕНИЯ ЗАРПЛАТЫ."?!? ВЫ УВОЛЕНЫ!!

• А вот здесь для вас есть кое-что, чтоб ещё поиграть с **`center`**, **`ljust`** и **`rjust`**: напишите программу, которая будет отображать "Содержание" так, чтобы это выглядело следующим образом:

```
                    Содержание



Глава 1:  Числа                                 page 1
```

Высшая математика

(Этот раздел — совершенно необязательный. Он предполагает некоторый уровень математических знаний. Если вам не интересно, вы можете без малейших затруднений перейти прямо к Управлению выполнением. Однако, быстрый взгляд на раздел о **случайных числах** *может весьма пригодиться.)*

Числовых методов не настолько много, как строковых методов (хотя я всё равно не держу их все в своей голове). Здесь мы рассмотрим оставшиеся арифметические методы, генератор случайных чисел и объект **Math** с его тригонометрическими и трансцендентальными методами.

Снова арифметика

Ещё два арифметических метода — это ** (возведение в степень) и % (деление по модулю). Так что, если вы хотите сказать на Ruby "пять в квадрате", вы просто запишите это как **5**2**. Вы также можете использовать в качестве степени числа с плавающей точкой, так что если вам нужен квадратный корень из пяти, вы можете написать **5**0.5**. Метод % (деление по модулю) выдаёт остаток от деления на число. Так, например, если я разделю 7 на 3, то получу 2 и 1 в остатке. Давайте посмотрим, как это работает в программе:

```
puts 5**2

puts 5**0.5

puts 7/3

puts 7%3

puts 365%7
```

```
25

2.23606797749979

2

1

1
```

Из последней строки мы узнали, что любой (не високосный) год состоит из некоторого количества недель плюс один день. Так что, если ваш день рождения был во вторник в этом году, на следующий год он будет в среду. Вы также можете применять в методе деления по модулю числа с плавающей точкой. В основном, он выполняет вычисления наиболее возможным осмысленным способом... но я предоставляю вам самим поиграть с этим.

Остаётся упомянуть ещё один метод прежде, чем мы проверим работу генератора

случайных чисел: **abs**. Он просто берёт абсолютное значение указанного числа:

```
puts((5-2).abs)

puts((2-5).abs)
```

```
3

3
```

Случайные числа

Ruby поставляется с довольно хорошим генератором случайных чисел. Метод, возвращающий случайно выбранное число, называется **rand**. Если вы вызовете **rand** как есть (без аргументов), вы получите дробное число, большее или равное **0.0** и меньшее **1.0**. Если вы дадите методу **rand** целое (например, **5**), он вернёт вам целое число, большее или равное **0** и меньшее, чем **5** (то есть одно из пяти возможных чисел, от **0** до **4**).

Давайте посмотрим **rand** в действии. (Если вы перезагрузите эту страницу [на оригинальном англоязычном сервере — *Прим. перев.*], то эти числа будут каждый раз другими. Вы ведь знали, что я на самом деле *выполнял* все эти программы, не так ли?)

```
puts rand

puts rand

puts rand

puts(rand(100))

puts(rand(100))

puts(rand(100))

puts(rand(1))

puts(rand(1))

puts(rand(1))

puts(rand(99999999999999999999999999999999999999999999999999999999))

puts('Синоптик сказал, что с вероятностью в '+rand(101).to_s+'% пойдёт
дождь,')

puts('но никогда не стоит доверять синоптикам.')
```

```
0.053950924931684

0.975039266747952

0.436084118016833

63
```

```
40

38

0

0

0

54350491927962189206794015651522429182285732200948685516886

Синоптик сказал, что с вероятностью в 22% пойдёт дождь,

но никогда не стоит доверять синоптикам.
```

Обратите внимание, что я использовал **rand(101)**, чтобы получить числа от **0** до **100**, и что **rand(1)** всегда возвращает **0**. Непонимание того, каков диапазон возможных возвращаемых значений, является, по-моему, самой частой ошибкой, которую делают при работе с **rand**; даже профессиональные программисты и даже в завершённых программных продуктах, которые вы покупаете в магазине. У меня даже однажды был CD-проигрыватель, который в режиме "Случайное воспроизведение" проигрывал все песни, кроме последней... (Я гадал, что бы могло произойти, если я бы вставил в него CD с одной-единственной песней?)

Иногда вы можете захотеть, чтобы **rand** возвращал *те же самые* случайные числа в той же последовательности при двух разных запусках вашей программы. (Например, я однажды использовал случайно сгенерированные числа для создания случайно сгенерированного мира в компьютерной игре. Если я обнаружил мир, который мне по-настоящему понравился, возможно, мне захочется снова сыграть с ним или отправить его другу.) Чтобы проделать это, вам нужно задать *seed* [зерно случайной последовательности — *Прим. перев.*], что можно сделать методом **srand**. Вот так:

```
srand 1776

puts(rand(100))

puts(rand(100))

puts(rand(100))

puts(rand(100))

puts(rand(100))

puts ''

srand 1776

puts(rand(100))

puts(rand(100))

puts(rand(100))

puts(rand(100))

puts(rand(100))
```

```
24

35

36

58

70

24

35

36

58

70
```

Одно и то же будет выдаваться каждый раз, когда вы "посеете" то же самое число в качестве зерна. Если вы снова хотите получать различные числа (также, как происходит, если вы не применяли до этого **srand**), то просто вызовите **srand 0**. Этим вызовом в генератор "засевается" действительно причудливое число с использованием (кроме всего прочего) текущее время в вашем компьютере, с точностью до миллисекунды.

Объект Math

Наконец, давайте рассмотрим объект **Math**. Наверное, нам следует сразу окунуться в примеры:

```
puts(Math::PI)

puts(Math::E)

puts(Math.cos(Math::PI/3))

puts(Math.tan(Math::PI/4))

puts(Math.log(Math::E**2))

puts((1 + Math.sqrt(5))/2)
```

```
3.14159265358979

2.71828182845905

0.5

1.0

2.0

1.61803398874989
```

Первое, что вы, возможно, заметили, это символы **::** в обозначениях констант. Объяснение оператора пределов видимости (а это именно он) на самом деле выходит

за, хм... пределы этого учебника. Я не хотел каламбурить. Честное слово. Достаточно сказать, что вы можете просто использовать **`Math::PI`** в ожидаемом вами значении.

Как видите, в **`Math`** есть всё, что вы предполагаете иметь в приличном научном калькуляторе. И как и прежде, дробные числа *действительно близко* представляют правильные результаты. [Точность вычислений десятичных чисел ограничена их двоичным представлением в компьютере. — *Прим. перев.*]

А теперь перейдём к выполнению!

6. Управление выполнением

Аааaа, управление выполнением... Вот где всё соединяется воедино. И хотя эта глава короче и легче, чем глава о методах, она откроет вам целый мир программных возможностей. После этой главы мы сможем писать по-настоящему интерактивные программы; до этого мы создавали программы, которые *выводили* разные вещи в зависимости от вашего ввода с клавиатуры, но после этой главы они также будут действительно *делать* разные вещи. Но прежде, чем мы сможем сделать это, нам нужно иметь возможность сравнивать объекты в наших программах. Нам нужны...

Методы сравнения

Давайте быстро пробежимся по этому разделу, чтобы можно было перейти к следующему разделу, **"Ветвления"**, где будут происходить самые классные вещи. Так, чтобы узнать, что один объект больше или меньше другого, мы применяем методы > и <, вот так:

```
puts 1 > 2

puts 1 < 2
```

```
false

true
```

Ничего сложного. Подобным же образом мы можем выяснить, что объект больше или равен другому (или меньше или равен) с помощью методов >= и <= .

```
puts 5 >= 5

puts 5 <= 4
```

```
true

false
```

И наконец, мы можем выяснить, равны ли два объекта или нет, используя == (что обозначает "они равны?") и != (что обозначает "они различны?"). Важно не путать = с ==. = используется, чтобы заставить переменную указывать на объект (присваивание), а == используется для ответа на вопрос: "Эти два объекта равны?"

```
puts 1 == 1

puts 2 != 1
```

```
true

true
```

Конечно, мы также можем сравнивать строки. Когда сравниваются строки, они

сопоставляются в лексикографическом порядке, что обычно означает словарную упорядоченность. **cat** в словаре идёт перед **dog**, поэтому:

```
puts 'cat' < 'dog'
```

```
true
```

Однако, здесь таится ловушка: обычно в компьютерах устроено так, что устанавливается такой порядок букв, в котором заглавные располагаются перед строчными буквами. (Так хранятся буквы в шрифтах, например: сначала все заглавные буквы, а за ними — строчные.) Это означает, что компьютер будет думать, что **'Zoo'** располагается перед **'ant'**, так что, если вы хотите выяснить, какое слово идёт первым в настоящем словаре, обязательно примените метод **downcase** (или **upcase** или **capitalize**) к обоим словам прежде, чем вы попытаетесь их сравнить.

Последнее замечание перед **"Ветвлением"**: методы сравнения не возвращают нам строки **'true'** и **'false'**; они возвращают нам специальные объекты **true** и **false**. (Конечно, **true.to_s** вернёт нам **'true'**, именно поэтому **puts** напечатал **'true'**.) **true** и **false** всегда используются для...

Ветвления

Ветвление — это простое, но очень мощное понятие. Фактически, оно настолько простое, что, ей-Богу, мне совсем даже и не нужно объяснять его; я просто покажу его вам:

```
puts 'Привет, как Вас зовут?'

name = gets.chomp

puts 'Привет, ' + name + '.'

if name == 'Chris'

  puts 'Какое милое имя!'

end
```

```
Привет, как Вас зовут?

Chris

Привет, Chris.

Какое милое имя!
```

Но если мы введём другое имя...

```
Привет, как Вас зовут?

Chewbacca

Привет, Chewbacca.
```

Вот это и есть ветвление. Если то, что находится после **if**, имеет значение **true**, мы выполняем код между **if** и **end**. Если то, что находится после **if**, имеет значение **false**, то не выполняем. Ясно и просто.

Я выделил отступом код между **if** и **end** просто потому, что, полагаю, таким образом легче отслеживать ветвление. Почти все программисты делают так, независимо от того, на каком языке они программируют. В этом простом примере может показаться, что от этого не слишком много пользы, но когда программа становится более сложной, это существенно меняет дело.

Часто нам бы хотелось, чтобы программа выполняла что-то, если выражение равно **true**, и нечто другое, если оно равно **false**. Для этого имеется **else**:

```
puts 'Я - предсказатель судьбы.  Скажите мне своё имя:'

name = gets.chomp

if name == 'Chris'

  puts 'Я предвижу у Вас в будущем великие дела.'

else

  puts 'Ваше будущее... О Боже!  Посмотрите-ка на часы!'

  puts 'На самом деле мне пора уходить, извините!'

end
```

```
Я предсказатель будущего.  Скажите мне своё имя:

Chris

Я предвижу у Вас в будущем великие дела.
```

А теперь давайте попробуем другое имя...

```
Я предсказатель будущего.  Скажите мне своё имя:

Ringo

Ваше будущее... О Боже! Посмотрите-ка на часы!

На самом деле мне пора уходить, извините!
```

Ветвление — это как будто мы подошли к развилке в коде: мы выберем дорожку к людям, чьё имя — Крис (**name == 'Chris'**), или же (**else**) мы выберем другой путь.

И совсем как ветви на дереве, у вас могут быть ветвления, которые сами содержат ветвления:

```
puts 'Здравствуйте и добро пожаловать в 7-й класс на урок английского.'

puts 'Меня зовут миссис Габбард.  А тебя зовут...?'

name = gets.chomp
```

```
if name == name.capitalize
  puts 'Садись, пожалуйста, ' + name + '.'
else
  puts name + '?  Ты имел в виду: ' + name.capitalize + ', не так ли?'
  puts 'Ты что, даже не знаешь, как пишется твоё имя??'
  reply = gets.chomp

  if reply == 'да'
    puts 'Хммм!  Ну хорошо, садись!'
  else
    puts 'УБИРАЙСЯ ВОН!!'
  end
end
```

```
Здравствуйте и добро пожаловать в 7-й класс на урок английского.
Меня зовут миссис Габбард.  А тебя зовут...?

chris

chris?  Ты имел в виду: Chris, не так ли?
Ты что, даже не знаешь, как пишется твоё имя??

да

Хммм!  Ну хорошо, садись!
```

Прекрасно, напишу его с заглавной буквы...

```
Здравствуйте и добро пожаловать в 7-й класс на урок английского.
Меня зовут миссис Габбард.  А тебя зовут...?

Chris

Садись, пожалуйста, Chris.
```

Иногда можно запутаться, пытаясь "вычислить", куда же ведут все эти **`if`**-ы, **`else`-ы** и **`end`**-ы. Я делаю так: пишу **`end`** *сразу же*, когда напишу **`if`**. Так, когда я писал приведённую выше программу, сначала она выглядела вот так:

```
puts 'Здравствуйте и добро пожаловать в 7-й класс на урок английского.'
puts 'Меня зовут миссис Габбард.  А тебя зовут...?'
name = gets.chomp
```

```
if name == name.capitalize

else

end
```

Затем я вставил комментарии — те вещи в коде, которые компьютер проигнорирует:

```
puts 'Здравствуйте и добро пожаловать в 7-й класс на урок английского.'

puts 'Меня зовут миссис Габбард.  А тебя зовут...?'

name = gets.chomp

if name == name.capitalize

  #  Она говорит вежливо.

else

  #  Она постепенно свирепеет.

end
```

Всё, что стоит после знака #, считается *комментарием* (конечно, если только он не находится внутри строки). После этого я заменил комментарии работающим кодом. Некоторым нравится оставлять комментарии; лично я думаю, что хорошо написанный код обычно говорит сам за себя. Раньше я применял больше комментариев, но чем более "бегло" я пишу на Ruby, тем меньше я их использую. На самом деле я нахожу их в большинстве случаев отвлекающими внимание. Но это каждый выбирает для себя сам; и у вас сложится свой собственный (обычно развивающийся) стиль. И вот, мой следующий шаг выглядел так:

```
puts 'Здравствуйте и добро пожаловать в 7-й класс на урок английского.'

puts 'Меня зовут миссис Габбард.  А тебя зовут...?'

name = gets.chomp

if name == name.capitalize

  puts 'Садись, пожалуйста, ' + name + '.'

else

  puts name + '?  Ты имел в виду: ' + name.capitalize + ', не так ли?'

  puts 'Ты что, даже не знаешь, как пишется твоё имя??'

  reply = gets.chomp


  if reply == 'да'

  else

  end

end
```

Снова я написал **if**, **else** и **end** одновременно. Это на самом деле помогает мне отслеживать, "где я нахожусь" в тексте программы. К тому же от этого задача кажется немного легче, поскольку я могу сфокусироваться на небольшой части программы — на вставке кода между **if** и **else**. Ести и другое преимущество от того, что я поступаю именно так. Оно состоит в том, что компьютер может понять программу на любом этапе разработки. Каждая из незаконченных версий программы, которые я показывал вам, будет выполняться. Они не были закончены, но это были работающие программы. Таким образом, я мог проверять её по ходу её написания, что помогало увидеть, как обстоят дела, и где она всё ещё нуждалась в доработке. Когда она прошла все проверки, я и выяснил, что я её закончил!

Эти подсказки помогут вам писать программы с ветвлением, но они также помогут освоить другой основной тип управления выполнением:

Циклы

Довольно часто вам может захотеться, чтобы ваш компьютер выполнял одно и то же снова и снова — в конце концов, предполагается, что именно этим компьютеры занимаются лучше всего.

Когда вы говорите вашему компьютеру повторять что-либо, вам также нужно сказать ему, когда остановиться. Компьютеру никогда не может наскучить, поэтому если вы не скажете ему остановиться, он и не остановится. Мы можем быть уверены, что подобное не случится, если скажем компьютеру повторять отдельные части программы пока (**while**) определённое условие является истинным. Это работает очень похоже на то, как работает **if**:

```
command = ''

while command != 'пока'

  puts command

  command = gets.chomp

end

puts 'Приходите ещё!'
```

```
Ау?

Ау?

Привет!

Привет!

Рад познакомиться.

Рад познакомиться.

О... как приятно!
```

```
О... как приятно!

пока

Приходите ещё!
```

Вот это и есть цикл. (Вы, наверное, заметили пустую строку в в начале вывода; она — от первого **puts**, перед первым **gets**. Как бы вы изменили программу, чтобы избавиться от этой пустой строки? Проверьте её! Она работает *в точности*, как программа выше, не считая первой пробельной строки?)

Циклы позволяют вам делать самые разные интересные вещи, какие только можно себе вообразить. Они, однако, могут также вызвать и неприятности, если вы сделаете ошибку. Что если ваш компьютер попадётся в ловушку бесконечного цикла? Если вы думаете, что именно так, возможно, и случилось, то просто, удерживая клавишу `Ctrl`, нажмите на клавишу `C`. [Это прервёт выполнение вашей программы. — *Прим. перев.*]

Но прежде, чем мы начнём забавляться с циклами, давайте изучим несколько вещей, которые облегчат нам жизнь.

Чуть-чуть логики

Давайте снова взглянем на нашу первую программу с ветвлением. Что если моя жена пришла домой, увидела эту программу, попробовала выполнить её, и она не сказала ей, какое милое имя *у неё*? Я не хотел бы обидеть её чувства (и спать на кушетке), так что давайте перепишем её:

```
puts 'Привет, как Вас зовут?'
name = gets.chomp
puts 'Привет, ' + name + '.'
if name == 'Chris'
  puts 'Какое милое имя!'
else
  if name == 'Katy'
    puts 'Какое милое имя!'
  end
end
```

```
Привет, как Вас зовут?

Katy

Привет, Katy.

Какое милое имя!
```

Ну да, она работает... но это не очень-то красивая программа. Почему не очень? Ну,

потому, что лучшее правило, которое я когда-либо узнал в программировании, — это правило DRY (Don't Repeat Yourself), то есть "Не повторяйся!" Возможно. я бы мог написать небольшую книгу только о том, почему это такое хорошее правило. В нашем случае, мы повторили строку **`puts 'Какое милое имя!'`**. Почему это так важно? Ладно, а что если я сделал опечатку, когда переписывал её? Что если я хотел изменить с **`'милое'`** на **`'прекрасное'`** в обеих строках? Я же ленивый, помните? По сути, если я хочу, чтобы программа делала одно и то же, когда она получает **`'Chris'`** или **`'Katy'`**, тогда она должна действительно *делать одно и то же*:

```
puts 'Привет, как Вас зовут?'

name = gets.chomp

puts 'Привет, ' + name + '.'

if (name == 'Chris' or name == 'Katy')

  puts 'Какое милое имя!'

end
```

```
Привет, как Вас зовут?

Katy

Привет, Katy.

Какое милое имя!
```

Гораздо лучше. Чтобы заставить это заработать, я применил **`or`**. Другие *логические операции* это **`and`** и **`not`**. ["ИЛИ", "И" и "НЕ" — *Прим. перев.*] И когда работаешь с ними, лучшим решением будет всегда использовать скобки. Давайте посмотрм, как они работают:

```
iAmChris  = true

iAmPurple = false

iLikeFood = true

iEatRocks = false

puts (iAmChris  and iLikeFood)

puts (iLikeFood and iEatRocks)

puts (iAmPurple and iLikeFood)

puts (iAmPurple and iEatRocks)

puts

puts (iAmChris  or iLikeFood)

puts (iLikeFood or iEatRocks)

puts (iAmPurple or iLikeFood)
```

```
puts (iAmPurple or iEatRocks)

puts

puts (not iAmPurple)

puts (not iAmChris )
```

```
true

false

false

false

true

true

true

false

true

false
```

Единственная из них, которая может обмануть вас, — это операция **or**. В английском языке мы часто используем "or" в значении "один или другой, но не оба". Например, ваша мама могла бы сказать: "На десерт ты можешь съесть торт или пирожное". Она *не* предполагала, что вы можете съесть и то и другое! Компьютер, напротив, использует **or** в значении "один или другой или оба". (Другой способ выразить это: "по крайней мере, один из них — истинный".) Вот почему с компьютерами гораздо веселее, чем с мамами.

Попробуйте ещё кое-что

• *"99 бутылок пива на стене..."* Напишите программу, которая печатает стихи этой излюбленной классической походной песни: "99 бутылок пива на стене".
• Напишите программу "Глухая бабуля". Что бы вы ни говорили бабуле (чтобы вы ни вводили с консоли), она должна отвечать: АСЬ?! ГОВОРИ ГРОМЧЕ, ВНУЧЕК!, если только вы не кричите ей (вводите слова заглавными буквами). Если вы кричите, она вас слышит (или по крайней мере думает, что слышит) и вопит в ответ: НЕТ, НИ РАЗУ С 1938 ГОДА! Чтобы сделать вашу программу *действительно* правдоподобной, пусть бабуля кричит каждый раз другой год; например, любой случайный год между 1930 и 1950. (Эта часть необязательная, и вам будет гораздо легче, если вы прочтёте раздел о генераторе случайных чисел в Ruby в конце главы о методах.) Вы не можете остановить разговор с бабулей, пока не прокричите `ПОКА`.

Подсказка: *Не забывайте о* **chomp**! **'ПОКА'** *с символом Enter это не то же самое, что* **'ПОКА'** *без него!*
Подсказка 2: *Попробуйте обдумать, какие части вашей программы должны происходить снова и снова. Все они должны находиться внутри цикла* **while**.

• Улучшите вашу программу "Глухая бабуля": Что если бабуля не хочет, чтобы вы уходили? Когда вы кричите `ПОКА`, она может притвориться, что не слышит вас. Измените вашу предыдущую программу так, чтобы вам нужно было прокричать `ПОКА` три раза *в одной строке*. Удостоверьтесь в правильности вашей программы: если вы прокричите `ПОКА` три раза, но не в одной строке, вы должны дальше разговаривать с бабулей.
• Високосные годы. Напишите программу, которая будет спрашивать начальный год и конечный год, а затем выдавать с помощью **`puts`** все високосные годы между ними (и включая их, если они также високосные). Високосные годы — это годы, нацело делящиеся на 4 (как 1984 и 2004). Однако, годы, нацело делящиеся на 100, — *не* високосные (как 1800 и 1900) *если только* они не делятся нацело на 400 (как 1600 и 2000, которые действительно были високосными). *(Да, это всё довольно запутанно, но не настолько запутанно, как если бы июль был в середине зимы, что иногда случалось бы.)* [если бы не было високосных годов. — *Прим. перев.*]

Когда вы это закончите, сделайте перерыв! Вы уже многое изучили. Поздравляю! Вас удивляет, сколько разных вещей вы можете заставить делать компьютер? Ещё несколько глав, и вы сможете запрограммировать почти всё, что угодно. Серьёзно! Только взгляните на все эти штуки, что вы можете делать сейчас и которые не смогли бы сделать без циклов и ветвления.

А теперь давайте изучим новую разновидность объектов, которые отвечают за списки других объектов: массивы.

7. Массивы и итераторы

Давайте напишем программу, которая просит нас ввести сколько угодно слов (по одному слову в строке до тех пор, пока мы не нажмём `Enter` на пустой строке) и которая затем повторяет нам эти слова в алфавитном порядке. Идёт?

Так... сначала мы — ээ... ну... хммм... Хорошо, мы могли бы — ээ... ну...

Вы знаете, я не думаю, что мы сможем это сделать. Нам нужен способ хранить неизвестное количество слов и как-то держать их все вместе, чтобы они не смешивались с другими переменными. Нам нужно поместить их все в какое-то подобие списка. Нам нужны массивы.

Массив — это просто список в вашем компьютере. Каждая ячейка в списке ведёт себя как переменная: вы можете посмотреть, на какой объект указывает определённая ячейка, и вы можете сделать так, чтобы она указывала на другой объект. Давайте посмотрим на некоторые массивы:

```
[]                              #  пустой массив

[5]                             #  массив из одного числа

['Привет', 'До свидания']       #  массив из 2-х строк

flavor = 'ванильный'            #  Это не массив, конечно...

[89.9, flavor, [true, false]]   #  ...но это — массив.
```

Итак, сначала у нас есть пустой массив, затем массив, содержащий единственное число, затем массив, содержащий две строки. Потом у нас имеется простое присваивание; затем — массив, содержащий три объекта, последний из которых — это массив **`[true, false]`**. Помните, что переменные — это не объекты, поэтому наш последний массив в действительности указывает на число с плавающей точкой, на *строку* и на массив. Даже если бы мы сделали, чтобы переменная **`flavor`** указывала на что-нибудь другое, это не изменило бы этот массив.

Чтобы помочь нам с поиском конкретных объектов в массиве, каждой ячейке даётся номер — индекс. Программисты (и, по случайному совпадению, большинство математиков) начинают считать их с нуля, так что первая ячейка в массиве — это ячейка номер ноль. Вот как мы должны адресоваться к объектам в массиве:

```
names = ['Ада', 'Белль', 'Крис']

puts names

puts names[0]

puts names[1]

puts names[2]
```

```
puts names[3]  #  Это вне диапазона массива.
```

```
Ада

Белль

Крис

Ада

Белль

Крис

nil
```

Итак, мы видим, что **puts names** выводит каждое имя в массиве **names**. Затем мы используем **puts names[0]**, чтобы напечатать "первое" имя в массиве и **puts names[1]**, чтобы напечатать "второе"... Уверен, что это кажется запутанным, но вы *непременно* привыкните к этому. Вы просто должны действительно начать *думать*, что отсчёт начинается с нуля, и перестать использовать такие слова, как "первый" и "второй". Если вы идёте на обед из пяти блюд, не говорите о "первом" блюде; говорите о блюде ноль (а у себя в голове думайте о **course[0]**). На правой руке у вас пять пальцев с номерами 0, 1, 2, 3 и 4. Моя жена и я умеем жонглировать. Когда мы жонглируем шестью булавами, мы жонглируем булавами 0-5. Надеемся, что через несколько месяцев мы сможем жонглировать булавой номер 6 (и таким образом будем жонглировать между собой семью булавами). Вы поймёте, что привыкли к этому, когда начнёте использовать слово "нулевой". :-) Да, в самом деле есть такое слово: спросите у любого программиста или математика.

Наконец, мы попытались выполнить **puts names[3]**, просто чтобы посмотреть, что же получится. Вы ожидали ошибку? Иногда, когда вы задаёте вопрос, ваш вопрос не имеет смысла (по крайней мере для вашего компьютера); и тогда вам выдаётся ошибка. Иногда, однако, вы можете задать вопрос и ответ на него будет: *ничего*. Что в ячейке номер три? Ничего. Что в **names[3]**? **nil**: таким способом Ruby говорит "ничего". **nil** — это особенный объект, который обычно означает "никакой из объектов".

Если вся эта смешная нумерация ячеек массивов начинает вас "доставать", не бойтесь! Часто мы сможем обойтись совершенно без неё, используя различные методы массивов, как вот этот:

Метод each

each позволяет нам делать что-нибудь (всё, что мы хотим) с **каждым** объектом, на который указывает массив. Так, если мы хотим сказать что-либо приятное о языке в приведённом ниже массиве, мы сделаем так:

```
languages = ['английский', 'немецкий', 'Ruby']

languages.each do |lang|

  puts 'Мне нравится ' + lang + '!'
```

```
  puts 'А вам?'

end

puts 'А теперь давайте послушаем мнение о C++!'

puts '...'
```

```
Мне нравится английский!

А вам?

Мне нравится немецкий!

А вам?

Мне нравится Ruby!

А вам?

А теперь давайте послушаем мнение о C++!

...
```

Так что же сейчас произошло? Вот, мы смогли обойти каждый из объектов в массиве совсем без использования номеров, и это определённо приятно. В переводе на русский, приведённую выше программу можно прочитать примерно так: для каждого (**each**) объекта в **languages**, пусть переменная **lang** указывает на этот объект и затем выполнится (**do**) всё, что я скажу вам, пока вы не дойдёте до конца (**end**). (Чтобы вы знали: C++ это ещё один язык программирования. Его гораздо труднее изучить, чем Ruby; обычно, программа на C++ будет во много раз длиннее, чем программа на Ruby, которая делает то же самое.)
Вы, должно быть, думаете: "Это очень похоже на циклы, о которых мы узнали до этого". Ну да, похоже. Одно важное отличие заключается в том, что метод **each** — это просто-напросто метод. **while** и **end** (также как **do**, **if**, **else** и все другие **синие** слова) это не методы. Они являются основополагающей частью языка Ruby, как и операция = или скобки; они похожи на знаки пунктуации в английском [или русском — *Прим. перев.*] языке.

Но не **each**; **each** — это просто ещё один метод массива. Такие методы, как **each**, которые "ведут себя как" циклы, часто называют итераторами.

Нужно отметить ещё одну вещь об итераторах: за ними всегда следует **do**...**end**. Около **while** и **if** никогда не бывает **do**; мы используем **do** только с итераторами.

А вот другой маленький симпатичный итератор, но это не метод массива... Это метод целого числа!

```
3.times do

  puts 'Гип-гип-ура!'

end
```

```
Гип-гип-ура!

Гип-гип-ура!
```

```
Гип-гип-ура!
```

Другие методы массивов

Итак, мы изучили **each**, но у массивов есть много других методов... почти столько же много, как методов у строк! В сущности, некоторые из них (такие, как **length**, **reverse**, **+** и *****) работают так же, как и со строками, только они оперируют с ячейками массива, а не с буквами в строке. Другие, например, **last** и **join**, характерны только для массивов. Третьи, такие как **push** и **pop**, фактически изменяют массив. И так же, как с методами строк, вам не нужно помнить их все, покуда вы помните, где можно узнать о них (прямо здесь).

Сначала давайте взглянем на **to_s** и **join**. Метод **join** работает во многом схоже с методом **to_s**, за исключением того, что он добавляет строку между объектами массива. Давайте посмотрим:

```
foods = ['артишок', 'бриошь', 'карамель']

puts foods

puts

puts foods.to_s

puts

puts foods.join(', ')

puts

puts foods.join('  :)  ') + '  8)'

200.times do

  puts []

end
```

```
артишок

бриошь

карамель

артишокбриошькарамель

артишок, бриошь, карамель

артишок  :)  бриошь  :)  карамель  8)
```

Как видите, метод **puts** обращается с массивами не так, как с другими объектами: он просто вызывает **puts** для каждого из объектов в массиве. Вот почему вывод через **puts** пустого массива 200 раз ничего не делает: массив ни на что не указывает, так что нечего выводить с помощью **puts**. (Ничего не делать 200 раз — значит всё равно ничего не делать.) Попробуйте вывести методом **puts** массив, содержащий другие массивы: он делает то, что вы ожидали?

Кроме того, вы заметили, что я не употреблял в **puts** пустые строки, когда хотел вывести чистую строку? Получилось то же самое.

Теперь давайте посмотрим на **push**, **pop** и **last**. Методы **push** и **pop** в каком-то смысле противоположны, как + и -. **push** добавляет объект в конец вашего массива, а **pop** удаляет последний объект из массива (и сообщает вам, что это был за объект). **last** похож на **pop** в том, что он сообщает вам, что находится в конце массива, только он оставляет массив нетронутым. Повторю снова: **push** и **pop** *действительно изменяют массив*:

```
favorites = []

favorites.push 'капли дождя на розах'

favorites.push 'капли виски на котах'

puts favorites[0]

puts favorites.last

puts favorites.length

puts favorites.pop

puts favorites

puts favorites.length
```

```
капли дождя на розах

капли виски на котах

2

капли виски на котах

капли дождя на розах

1
```

Попробуйте ещё кое-что

• Напишите программу, о которой мы говорили в самом начале этой главы.

> **Подсказка:** *Есть прекрасный метод массива, который вернёт вам отсортированную версию массива:* **sort**. *Используйте его!*

• Попробуйте написать указанную программу *без* использования метода **sort**. Большая часть программирования - это преодоление сложностей, так что практикуйтесь чаще, насколько это возможно!

• Перепишите вашу программу "Содержание" (из главы о методах). Начните программу с массива, содержащего всю информацию для вашей таблицы с содержанием (названия глав, номера страниц и т. д.). Затем напечатайте информацию из массива в виде красиво отформатированного содержания.

До сих пор мы изучили довольно много разных методов. А сейчас пора научиться, как сделать свои собственные.

8. Пишем свои методы

Как мы уже видели, циклы и итераторы позволяют нам делать одно и то же (выполнять тот же самый код) снова и снова. Однако, иногда мы хотим сделать одно и то же несколько раз, но в разных частях программы. Например, мы бы разрабатывали, скажем, программу опроса для студента-психолога. Судя по разговорам со знакомыми студентами-психологами и по опросам, которые они мне предлагали, она, наверное, должна быть примерно такой:

```
puts 'Здравствуйте! И спасибо, что Вы нашли время, чтобы'
puts 'помочь мне в этом исследовании.  Мое исследование'
puts 'связано с изучением того, как люди относятся к'
puts 'мексиканской еде. Просто думайте о мексиканской еде'
puts 'и попробуйте отвечать на все вопросы честно,'
puts 'только словами "да" или "нет".  Моё исследование'
puts 'не имеет ничего общего с ночным недержанием мочи.'
puts
#  Мы задаём эти вопросы, но игнорируем ответы на них.
goodAnswer = false
while (not goodAnswer)
  puts 'Вам нравится есть такос?'
  answer = gets.chomp.downcase
  if (answer == 'да' or answer == 'нет')
    goodAnswer = true
  else
    puts 'Пожалуйста, отвечайте "да" или "нет".'
  end
end
goodAnswer = false
while (not goodAnswer)
  puts 'Вам нравится есть бурритос?'
  answer = gets.chomp.downcase
  if (answer == 'да' or answer == 'нет')
```

```
    goodAnswer = true
  else
    puts 'Пожалуйста, отвечайте "да" или "нет".'
  end
end
#  Мы, однако, обращаем внимание на *этот* вопрос.
goodAnswer = false
while (not goodAnswer)
  puts 'Вы мочитесь в постель?'
  answer = gets.chomp.downcase
  if (answer == 'да' or answer == 'нет')
    goodAnswer = true
    if answer == 'да'
      wetsBed = true
    else
      wetsBed = false
    end
  else
    puts 'Пожалуйста, отвечайте "да" или "нет".'
  end
end
goodAnswer = false
while (not goodAnswer)
  puts 'Вам нравится есть чимичангас?'
  answer = gets.chomp.downcase
  if (answer == 'да' or answer == 'нет')
    goodAnswer = true
  else
    puts 'Пожалуйста, отвечайте "да" или "нет".'
  end
end
puts 'И ещё несколько вопросов...'
```

```
goodAnswer = false
while (not goodAnswer)
  puts 'Вам нравится есть сопапиллас?'
  answer = gets.chomp.downcase
  if (answer == 'да' or answer == 'нет')
    goodAnswer = true
  else
    puts 'Пожалуйста, отвечайте "да" или "нет".'
  end
end
#  Задайте много других вопросов о мексиканской еде.
puts
puts 'ПОЯСНЕНИЕ:'
puts 'Спасибо за то, что Вы нашли время, чтобы помочь'
puts 'этому исследованию.  На самом деле, это исследование'
puts 'не имеет ничего общего с мексиканской едой.  Это'
puts 'исследование ночного недержания мочи.  Мексиканская еда'
puts 'присутствовала только затем, чтобы усыпить Вашу бдительность'
puts 'в надежде, что Вы будете отвечать более'
puts 'правдиво.  Ещё раз спасибо.'
puts
puts wetsBed
```

```
Здравствуйте! И спасибо, что Вы нашли время, чтобы
помочь мне в этом исследовании.  Моё исследование
связано с изучением того, как люди относятся к
мексиканской еде.  Просто думайте о мексиканской еде
и попробуйте отвечать на все вопросы честно,
только словами "да" или "нет".  Моё исследование
не имеет ничего общего с ночным недержанием мочи.
Вам нравится есть такос?

да

Вам нравится есть бурритос?
```

```
да

Вы мочитесь в постель?

никогда!

Пожалуйста, отвечайте "да" или "нет".

Вы мочитесь в постель?

нет

Вам нравится есть чимчангас?

да

И ещё несколько вопросов...

Вам нравится есть сопапиллас?

да

ПОЯСНЕНИЕ:

Спасибо за то, что Вы нашли время, чтобы помочь

этому исследованию.  На самом деле, это исследование

не имеет ничего общего с мексиканской едой.  Это

исследование ночного недержания мочи.  Мексиканская еда

присутствовала только затем, чтобы усыпить Вашу бдительность

в надежде, что Вы будете отвечать более

правдиво.  Ещё раз спасибо.

false
```

Это была довольно длинная программа со многими повторениями. (Все разделы программы с вопросами о мексиканской еде были одинаковыми, а вопрос о недержании мочи отличался совсем немного.) Повторение — это нехорошая штука. И всё же, мы не можем поместить его в один большой цикл или итератор, поскольку иногда нам нужно кое-что сделать между вопросами. В подобных ситуациях лучше всего написать метод. Вот так:

```
def sayMoo  #  скажи: "Му"

  puts 'мууууууу...'

end
```

Ээ... наша программа не выполняет **sayMoo**. Почему же? Потому что мы не сказали ей это делать. Мы сказали ей, *как* мычать методом **sayMoo**, но мы фактически так и не сказали ей *сделать* это. Давайте попытаемся по-другому:

```
def sayMoo  #  скажи: "Му"
```

```
  puts 'мууууууу...'
end

sayMoo

sayMoo

puts 'куан-куан'

sayMoo

sayMoo
```

```
мууууууу...

мууууууу...

куан-куан

мууууууу...

мууууууу...
```

Ааа, гораздо лучше. (Если вы не говорите по-французски, поясняю: в середине программы была французская утка. Во Франции утки говорят: *"куан-куан"*.)

Итак, с помощью **def** мы определили метод **sayMoo**. (Имена методов, как и имена переменных, начинаются со строчной буквы. Есть, однако, несколько исключений таких, как + или ==.) Но разве методы не должны всегда ассоциироваться с объектами? Ну да, должны, и в этом случае (как и в случаях с **puts** и **gets**) метод просто ассоциируется с объектом, представляющим всю программу. В следующей главе мы увидим, как добавлять методы к другим объектам. Но сначала...

Параметры метода

Вы, должно быть, заметили, что некоторые методы (такие, как **gets**, **to_s**, **reverse**...) просто вызываются у какого-нибудь объекта. Однако, другие методы (такие, как **+**, **-**, **puts**...) принимают параметры для указания объекту, как выполнять метод. Например, вы не скажете просто **5+**, правда? Этим вы говорите числу **5** прибавить, но не говорите ему *что* прибавить.

Чтобы определить параметр для метода **sayMoo** (скажем, количество мычаний), мы должны сделать так:

```
def sayMoo numberOfMoos

  puts 'мууууууу...'*numberOfMoos

end

sayMoo 3

puts 'хрю-хрю'

sayMoo  #  Это должно вызвать ошибку, потому что параметр отсутствует.
```

```
муууууууу...муууууууу...муууууууу...

хрю-хрю

#<ArgumentError: wrong number of arguments (0 for 1)>
```

[#<Ошибка аргументов: неверное число аргументов (0 вместо 1)> — *Прим. перев.*]

numberOfMoos — это переменная, которая указывает на параметр, переданный в метод. Я повторю это ещё раз, хотя это всё равно звучит немного запутанно: **numberOfMoos** — это переменная, которая указывает на параметр, переданный в метод. Так, если я напишу **sayMoo 3**, то параметр равен **3**, а переменная **numberOfMoos** указывает на **3**.

Как видите, параметр теперь *обязателен*. В конце концов, каким образом **sayMoo** должен повторять **'муууууууу...'**, если вы не передадите ему параметр? Ваш бедный компьютер не сообразит.

Если объекты в Ruby подобны существительным в английском языке, а методы подобны глаголам, то вы можете думать о параметрах, как о наречиях (как в случае с **sayMoo**, где параметр говорит нам *как* выполнить **sayMoo**) или иногда, как о прямом дополнении (как в случае с **puts**, где параметр — это то, *что* выводится через **puts**).

Локальные переменные

В следующей программе имеется две переменные:

```
def doubleThis num

  numTimes2 = num*2

  puts num.to_s+' дважды будет '+numTimes2.to_s

end

doubleThis 44
```

```
44 дважды будет 88
```

Переменные — это **num** и **numTimes2**. Обе они расположены внутри метода **doubleThis**. Эти (и все другие переменные, которые вы видели до сих пор) являются локальными переменными. Это означает, что они "живут" внутри метода и недоступны снаружи. А если вы попытаетесь выполнить следующий код, вам будет выдана ошибка:

```
def doubleThis num

  numTimes2 = num*2

  puts num.to_s+' дважды будет '+numTimes2.to_s

end
```

```
doubleThis 44

puts numTimes2.to_s
```

```
44 дважды будет 88

#<NameError: undefined local variable or method `numTimes2' for
#<StringIO:0x82ba924>>
```

[#<Ошибка имени: неопределённая локальная переменная или метод `numTimes2' для #<StringIO: 0x82ba924>> — *Прим. перев.*]

Неопределённая локальная переменная... Фактически, мы *определили* эту локальную переменную, но она не является локальной там, где мы попытались её использовать; она локальная внутри метода.

Это может показаться неудобным, но на самом деле это очень даже приятно. Хотя это означает, что у вас нет доступа к переменным внутри методов, это также означает, что у них нет доступа к *вашим* переменным, и, таким образом, их нельзя испортить:

```
def littlePest var

  var = nil

  puts 'ХА-ХА!  Я уничтожил твою переменную!'

end

var = 'Ты не можешь даже притронуться к моей переменной!'

littlePest var

puts var
```

```
ХА-ХА!  Я уничтожил твою переменную!

Ты не можешь даже притронуться к моей переменной!
```

Фактически, в этой маленькой программе *две* переменные **var**, а именно: та, что внутри метода **littlePest**, и та, что вне его. Когда мы вызвали **littlePest var**, мы в действительности просто передали строку из одной переменной **var** в другую так, что обе указывали на одну и ту же строку. Затем в методе **littlePest** его собственная *локальная* **var** стала указывать на **nil**, но это не повлияло на **var** вне метода.

Возвращаемые значения

Должно быть, вы заметили, что некоторые методы возвращают вам что-нибудь, когда вы их вызываете. Например, **gets** возвращает строку (ту строку, что вы ввели с клавиатуры), а метод **+** в выражении **5+3**, (а это на самом деле **5.+(3)**) возвращает **8**. Арифметические методы чисел возвращают числа, а арифметические методы строк возвращают строки.

Важно понять отличие между методами, возвращающими значение туда, где этот метод был вызван, и выводом вашей программой информации на экран монитора, как

это делает **puts**. Заметьте, что **5+3** возвращает **8**; он **не** выводит 8.

А что же тогда *возвращает* **puts**? Мы никогда до этого не задумывались об этом, но давайте взглянем сейчас:

```
returnVal = puts 'Это вернул метод puts:'

puts returnVal
```

```
Это вернул метод puts:

nil
```

Итак, первый **puts** вернул **nil**. И хотя мы этого не проверяли, второй **puts** вернул то же; **puts** всегда возвращает **nil**. Каждый метод должен возвращать что-нибудь, даже если это просто **nil**.

Прервитесь ненадолго и напишите программу, чтобы выяснить, что же возвращает **sayMoo**.

Вы удивлены? Хорошо, вот как это всё работает: значение, возвращаемое из метода, — это просто значение последней строки метода. В случае с **sayMoo** это означает, что он возвращает **puts 'муууууууу...'*numberOfMoos**, то есть просто **nil**, поскольку **puts** всегда возвращает **nil**. Если бы мы хотели, чтобы все наши методы возвращали строку **'жёлтая подводная лодка'**, нам бы просто нужно было поместить *это* в конце каждого из них:

```
def sayMoo numberOfMoos

  puts 'муууууууу...'*numberOfMoos

  'жёлтая подводная лодка'

end

x = sayMoo 2

puts x
```

```
муууууууу...муууууууу...

жёлтая подводная лодка
```

Итак, давайте снова вернёмся к нашему психологическому исследованию, но на этот раз мы напишем метод, который будет задавать для нас вопросы. Нужно, чтобы он принимал вопрос в качестве параметра и возвращал **true**, если ответ был `да`, или **false**, если ответ был `нет`. (Даже если в большинстве случаев мы просто игнорируем ответ, всё равно это неплохая идея, чтобы наш метод возвращал результат. Поступая так, мы сможем использовать его и для вопроса о ночном недержании.) Я также собираюсь сократить приветствие и пояснение просто для того, чтобы программу было легче читать:

```
def ask question  #  задать вопрос
```

```
  goodAnswer = false
  while (not goodAnswer)
    puts question
    reply = gets.chomp.downcase

    if (reply == 'да' or reply == 'нет')
      goodAnswer = true
      if reply == 'да'
        answer = true
      else
        answer = false
      end
    else
      puts 'Пожалуйста, отвечайте "да" или "нет".'
    end
  end

  answer  #  Это то, что мы возвращаем (true или false).
end
puts 'Здравствуйте! И спасибо, что Вы...'
puts
ask 'Вам нравится есть такос?' # Мы игнорируем возвращаемое значение.
ask 'Вам нравится есть бурритос?'
wetsBed = ask 'Вы мочитесь в постель?' # Мы сохраняем возвращаемое
значение.
ask 'Вам нравится есть чимичангас?'
ask 'Вам нравится есть сопапиллас?'
ask 'Вам нравится есть тамалес?'
puts 'И ещё несколько вопросов...'
ask 'Вам нравится пить хорчата?'
ask 'Вам нравится есть флаутас?'
puts
```

```
puts 'ПОЯСНЕНИЕ:'

puts 'Спасибо за то...'

puts

puts wetsBed
```

```
Здравствуйте! И спасибо, что Вы...

Вам нравится есть такос?

да

Вам нравится есть бурритос?

да

Вы мочитесь в постель?

никогда!

Пожалуйста, отвечайте "да" или "нет".

Вы мочитесь в постель?

нет

Вам нравится есть чимичангас?

да

Вам нравится есть сопапиллас?

да

Вам нравится есть тамалес?

да

И ещё несколько вопросов...

Вам нравится пить хорчата?

да

Вам нравится есть флаутас?

да

ПОЯСНЕНИЕ:

Спасибо за то...

false
```

Неплохо, а? Мы смогли добавить больше вопросов (ведь добавлять вопросы теперь *легко*), но наша программа к тому же ещё и стала немного короче! Это значительное усовершенствование — мечта каждого ленивого программиста.

Ещё один большой пример

Думаю, было бы очень полезно привести ещё один пример метода. Мы назовём его **englishNumber**. Он будет принимать число, например, **22**, и возвращать его английское название (в данном случае, строку **'twenty-two'**). Для начала, пусть он работает только с целыми числами от **0** до **100**.

> *(**ПРИМЕЧАНИЕ:** Этот метод использует новую хитрость для преждевременного возврата из метода с помощью ключевого слова **return**, а также представляет новый поворот в ветвлении: **elsif**. Из контекста должно быть ясно, как они работают.)*

```
def englishNumber number

  #  Нам нужны только числа в диапазоне 0-100.

  if number < 0

    return 'Пожалуйста, введите число ноль или больше.'

  end

  if number > 100

    return 'Пожалуйста, введите число 100 или меньше.'

  end


  numString = ''  #  Эту строку мы будем возвращать.


  #  "left" - сколько от числа нам ещё осталось вывести.

  #  "write" - часть числа, которую мы выводим сейчас.

  #  write и left... поняли?  :)

  left  = number

  write = left/100          #  Сколько сотен осталось вывести?

  left  = left - write*100  #  Вычтем эти сотни.


  if write > 0

    return 'one hundred'

  end


  write = left/10          #  Сколько десятков осталось вывести?

  left  = left - write*10  #  Вычтем эти десятки.
```

```
if write > 0
  if write == 1  #  Охо-хо...
    #  Поскольку мы не можем вывести "tenty-two" вместо "twelve",
    #  нам нужно сделать особые исключения для них.
    if    left == 0
      numString = numString + 'ten'
    elsif left == 1
      numString = numString + 'eleven'
    elsif left == 2
      numString = numString + 'twelve'
    elsif left == 3
      numString = numString + 'thirteen'
    elsif left == 4
      numString = numString + 'fourteen'
    elsif left == 5
      numString = numString + 'fifteen'
    elsif left == 6
      numString = numString + 'sixteen'
    elsif left == 7
      numString = numString + 'seventeen'
    elsif left == 8
      numString = numString + 'eighteen'
    elsif left == 9
      numString = numString + 'nineteen'
    end
    #  Поскольку уже мы позаботились о цифре для единиц,
    #  нам не осталось ничего для вывода.
    left = 0
  elsif write == 2
    numString = numString + 'twenty'
  elsif write == 3
```

```
    numString = numString + 'thirty'
  elsif write == 4
    numString = numString + 'forty'
  elsif write == 5
    numString = numString + 'fifty'
  elsif write == 6
    numString = numString + 'sixty'
  elsif write == 7
    numString = numString + 'seventy'
  elsif write == 8
    numString = numString + 'eighty'
  elsif write == 9
    numString = numString + 'ninety'
  end

  if left > 0
    numString = numString + '-'
  end
end

write = left  #  Сколько единиц осталось вывести?
left  = 0     #  Вычтем эти единицы.

if write > 0
  if    write == 1
    numString = numString + 'one'
  elsif write == 2
    numString = numString + 'two'
  elsif write == 3
    numString = numString + 'three'
  elsif write == 4
    numString = numString + 'four'
```

```
    elsif write == 5
      numString = numString + 'five'
    elsif write == 6
      numString = numString + 'six'
    elsif write == 7
      numString = numString + 'seven'
    elsif write == 8
      numString = numString + 'eight'
    elsif write == 9
      numString = numString + 'nine'
    end
  end

  if numString == ''
    #  Только в одном случае "numString" может быть пустой -
    #  если "number" равно 0.
    return 'zero'
  end

  #  Если мы дошли досюда, то у нас имеется число где-то
  #  между 0 и 100, поэтому нам нужно вернуть "numString".
  numString
end
puts englishNumber(  0)
puts englishNumber(  9)
puts englishNumber( 10)
puts englishNumber( 11)
puts englishNumber( 17)
puts englishNumber( 32)
puts englishNumber( 88)
puts englishNumber( 99)
puts englishNumber(100)
```

```
zero

nine

ten

eleven

seventeen

thirty-two

eighty-eight

ninety-nine

one hundred
```

И всё-таки, определённо имеется несколько моментов в этой программе, которые мне не нравятся. Во-первых, в ней слишком много повторений. Во-вторых, она не обрабатывает числа больше 100. В-третьих, в ней слишком много особых случаев, слишком много возвратов по **return**. Давайте используем несколько массивов и попробуем её немного подчистить:

```
def englishNumber number
  if number < 0  #  Без отрицательных чисел.
    return 'Пожалуйста, введите неотрицательное число.'
  end
  if number == 0
    return 'zero'
  end

  #  Больше нет особых случаев!  Больше нет возвратов по return!

  numString = ''  #  Эту строку мы будем возвращать.

  #  единицы
 onesPlace = ['one',     'two',       'three',    'four',     'five',
              'six',     'seven',     'eight',    'nine']
  #  десятки
  tensPlace = ['ten',     'twenty',    'thirty',   'forty',    'fifty',
               'sixty',   'seventy',   'eighty',   'ninety']
  teenagers = ['eleven',  'twelve',    'thirteen', 'fourteen', 'fifteen',
```

```
            'sixteen', 'seventeen', 'eighteen', 'nineteen']

#  "left" - сколько от числа нам ещё осталось вывести.
#  "write"  - часть числа, которую мы выводим сейчас.
#  write и left... поняли?  :)
left  = number
write = left/100          #  Сколько сотен осталось вывести?
left  = left - write*100  #  Вычтем эти сотни.

if write > 0
  #  Вот здесь действительно хитрый фокус:
  hundreds  = englishNumber write
  numString = numString + hundreds + ' hundred'
  #  Это называется "рекурсия".  Так что же я только что сделал?
  #  Я велел этому методу вызвать себя, но с параметром "write" вместо
  #  "number".  Помните, что "write" это (в настоящий момент) число
  #  сотен, которые нужно вывести.  Прибавив "hundreds" к "numString",
  #  мы добавляем после неё строку ' hundred'.  Так, например, если
  #  мы сначала вызвали englishNumber с 1999 (т.е. "number" = 1999),
  #  затем в этой точке "write" будет равен 19, а "left" равен 99.
  #  Наиболее лениво в этом месте было бы заставить englishNumber
  #  вывести нам 'nineteen', а затем мы выведем ' hundred',
  #  и потом оставшаяся часть englishNumber выведет 'ninety-nine'.

  if left > 0
    #  Так, мы не выводим 'two hundredfifty-one'...
    numString = numString + ' '
  end
end

write = left/10           #  Сколько десятков осталось вывести?
left  = left - write*10   #  Вычтем эти десятки.
```

```
  if write > 0
    if ((write == 1) and (left > 0))
      #  Поскольку мы не можем вывести "tenty-two" вместо "twelve",
      #  нам нужно сделать для них особую обработку.
      numString = numString + teenagers[left-1]
      #  "-1" здесь потому, что teenagers[3] это 'fourteen', а не
'thirteen'.

      #  Поскольку уже мы позаботились о цифре для единиц,
      #  нам не осталось ничего для вывода.
      left = 0
    else
      numString = numString + tensPlace[write-1]
      #  "-1" потому, что tensPlace[3] это 'forty', а не 'thirty'.
    end

    if left > 0
      #  Так, мы не выводим 'sixtyfour'...
      numString = numString + '-'
    end
  end

  write = left  #  Сколько единиц осталось вывести?
  left  = 0     #  Вычтем эти единицы.

  if write > 0
    numString = numString + onesPlace[write-1]
    #  "-1" потому, что onesPlace[3] это 'four', а не 'three'.
  end

  #  А теперь мы просто возвращаем "numString"...
```

```
  numString
end
puts englishNumber(  0)
puts englishNumber(  9)
puts englishNumber( 10)
puts englishNumber( 11)
puts englishNumber( 17)
puts englishNumber( 32)
puts englishNumber( 88)
puts englishNumber( 99)
puts englishNumber(100)
puts englishNumber(101)
puts englishNumber(234)
puts englishNumber(3211)
puts englishNumber(999999)
puts englishNumber(1000000000000)
```

```
zero
nine
ten
eleven
seventeen
thirty-two
eighty-eight
ninety-nine
one hundred
one hundred one
two hundred thirty-four
thirty-two hundred eleven
ninety-nine hundred ninety-nine hundred ninety-nine
one hundred hundred hundred hundred hundred hundred
```

Aaaaa.... Это гораздо, гораздо лучше. Программа довольно компактная, вот почему я добавил в неё так много комментариев. Она даже работает с большими числами... хотя не совсем так хорошо, как можно было надеяться. Например, я полагаю, для

последнего числа было бы гораздо лучше вернуть значение `'one trillion'`, или даже `'one million million'` (хотя все три значения правильные). В сущности, вы можете сделать это прямо сейчас...

Попробуйте ещё кое-что

• Доработайте **englishNumber**. Во-первых, добавьте тысячи. Так она должна возвращать `'one thousand'` вместо `'ten hundred'`, а также `'ten thousand'` вместо `'one hundred hundred'`.

• Ещё доработайте **englishNumber**. Теперь добавьте миллионы, чтобы вам возвращалось `'one million'` вместо `'one thousand thousand'`. Затем попробуйте добавить миллиарды и триллионы. Насколько далеко вы сможете зайти?

• А как насчёт **weddingNumber**? Она должна работать почти также, как **englishNumber**, но только она должна вставлять повсюду слово "and", возвращая что-то наподобие `'nineteen hundred and seventy and two'`, или как там это должно выглядеть в приглашениях на свадьбу? Я бы привёл вам больше примеров, но я сам не совсем это понимаю. Вам возможно, понадобится обратиться за помощью к устроителю свадеб.

• *"Девяносто девять бутылок пива..."* Используя **englishNumber** и вашу старую программу, напечатайте стихи этой песни, на этот раз *правильно*. Накажите ваш компьютер: пусть она начнётся с 9999. (Однако не задавайте слишком большое число, так как выводить всё это на экран займёт у компьютера достаточно много времени. Сто тысяч бутылок пива занимает приличное время; а если вы зададите миллион, вы накажете и себя тоже!

Мои поздравления! К этому моменту, вы стали настоящим программистом! Вы изучили всё, что нужно, чтобы составлять огромные программы с чистого листа. Если у вас есть идеи, какие бы программы вы сами хотели бы написать, попробуйте воплотить их!

Конечно, составлять всё с чистого листа может оказаться довольно медленным процессом. Зачем же тратить время на написание кода, который кто-то уже написал? Вы бы хотели, чтоб ваша программа оправляла электронную почту? Вы бы хотели сохранять и загружать файлы на свой компьютер? А как насчёт генерирования web-страниц для учебника, где примеры кода в самом деле выполняются каждый раз, когда загружается страница? ;) В Ruby есть много различных видов объектов, которые мы можем использовать и которые помогут нам писать программы лучше и быстрее.

9. Классы

До сих пор мы видели несколько различных видов, или классов, объектов: строки, целые числа, дробные числа, массивы, а также несколько особых объектов (**true**, **false** и **nil**), о которых мы поговорим позже. В Ruby эти классы всегда записываются с заглавной буквы: **String**, **Integer**, **Float**, **Array**... и т. д. В общем случае, если мы хотим создать новый объект определённого класса, мы используем **new**:

```
a = Array.new  + [12345]  #  Сложение массивов.

b = String.new + 'hello'  #  Сложение строк.

c = Time.new

puts 'a = '+a.to_s

puts 'b = '+b.to_s

puts 'c = '+c.to_s
```

```
a = 12345

b = hello

c = Wed Jun 28 02:11:24 GMT 2006
```

Так как мы можем создавать массивы и строки с помощью **[...]** и **'...'** соответственно, мы редко создаём их с помощью **new**. (Хотя это и не совсем очевидно из предыдущего примера, **String.new** создаёт пустую строку, а **Array.new** создаёт пустой массив.) Кроме того, числа являются особыми исключениями: вы не можете создать целое число с помощью **Integer.new**. Вам придётся просто записать число.

Класс Time

И что же особенного в этом классе **Time**? Объекты класса **Time** представляют моменты времени. Вы можете прибавлять числа к (или вычитать из) объектов времени, чтобы получить новые моменты времени: прибавление **1.5** к моменту времени создаст новый момент, который на полторы секунды позже первого:

```
time  = Time.new   #  Момент, когда вы получили эту web-страницу.

time2 = time + 60  #  Одной минутой позже.

puts time

puts time2
```

```
Wed Jun 28 02:11:24 GMT 2006

Wed Jun 28 02:12:24 GMT 2006
```

Вы также можете создавать объект времени, соответствующий определённому моменту, используя **Time.mktime**:

```
puts Time.mktime(2000, 1, 1)          #  Двухтысячный год (Y2K).

puts Time.mktime(1976, 8, 3, 10, 11)  #  Когда я родился.
```

```
Sat Jan 01 00:00:00 GMT 2000

Tue Aug 03 10:11:00 GMT 1976
```

Обратите внимание: момент моего рождения задан по Тихоокеанскому летнему времени (Pacific Daylight Savings Time, PDT). Хотя, когда наступил 2000-й год, было Тихоокеанское стандартное время (Pacific Standard Time, PST), по крайней мере для нас, жителей Западного берега. Скобки здесь нужны, чтобы сгруппировать параметры метода **mktime**. Чем больше параметров вы указываете, тем более точным становится ваше время.

Вы можете сравнивать время с помощью методов сравнения (более раннее время *меньше, чем* более позднее время), а если вы вычтете одно время из другого, вы получите разницу между ними в секундах. Поиграйте с этим немного!

Попробуйте ещё кое-что

• Один миллиард секунд... Выясните точно, в какую секунду вы родились (если сможете). Вычислите, когда вы достигнете (или, возможно, уже достигли?) возраста в один миллиард секунд. А потом отметьте это в вашем календаре.

• С днём рождения! Спросите, в каком году человек родился, затем в каком месяце, потом в какой день. Вычислите, сколько ему лет, и выдайте ему большой ХЛОП! на каждый его день рождения.

Класс Hash

Другой полезный класс — это класс **Hash**. Хэши во многом похожи на массивы: в них имеется набор слотов, которые могут указывать на различные объекты. Однако в массиве слоты выстроены в ряд, и каждый из них пронумерован (начиная с нуля). В хэше слоты не располагаются подряд (они просто как-то беспорядочно свалены вместе), и вы можете использовать для обращения к слоту *любой* объект, а не только число. Использовать хэши хорошо тогда, когда у вас есть набор каких-нибудь вещей, которые вы хотите обрабатывать, но они совсем не укладываются в нумерованный список. Например, цвета, которые я использую для различных частей кода, положенного в основу этого учебника:

```
colorArray = []  #  то же, что Array.new

colorHash  = {}  #  то же, что Hash.new

colorArray[0]       = 'красный'

colorArray[1]       = 'зелёный'
```

```
colorArray[2]       = 'синий'

colorHash['строки']         = 'красный'

colorHash['числа']          = 'зелёный'

colorHash['ключевые слова'] = 'синий'

colorArray.each do |color|

  puts color

end

colorHash.each do |codeType, color|

  puts codeType + ':  ' + color

end
```

```
красный

зелёный

синий

строки:  красный

ключевые слова:  синий

числа:  зелёный
```

Если я использую массив, мне нужно помнить, что слот **0** предназначен для строк, слот **1** — для чисел и т. д. Но если я использую хэш, то всё просто! Слот **'строки'**, конечно, содержит цвет строк. Ничего не нужно запоминать. Вы, должно быть, заметили, что когда мы применяли **each**, объекты из хэша выдавались не в том порядке, в котором мы их в него помещали. (По крайней мере, так было, когда я это писал. Возможно, сейчас будет по-другому... С этими хэшами никогда ничего не знаешь наперёд.) Для содержания чего-то в определённом порядке предназначены массивы, а не хэши.

Хотя для именования слотов в хэше обычно используются строки, вы могли бы использовать объект любого типа, даже массивы и другие хэши (хотя не могу представить себе, зачем бы вам захотелось это делать...):

```
weirdHash = Hash.new

weirdHash[12] = 'обезьян'

weirdHash[[]] = 'пустота'

weirdHash[Time.new] = 'текущее время и никакое другое'
```

Хэши и массивы хороши для разных применений; вам решать, что из них лучше всего подходит для конкретной задачи.

Расширение классов

В конце предыдущей главы вы написали метод, выдающий английскую фразу для заданного целого числа. Однако, это не был метод целых чисел; это был просто метод "программы вообще". Но разве не было бы прекрасно, если вы могли бы написать что-то вроде **22.to_eng** вместо **englishNumber 22**? Вот как вы бы это сделали:

```
class Integer


  def to_eng
    if self == 5
      english = 'five'
    else
      english = 'fifty-eight'
    end


    english
  end
end
#  Хорошо бы протестировать его на паре чисел...
puts 5.to_eng
puts 58.to_eng
```

```
five
fifty-eight
```

Ну вот, я его протестировал; кажется, он работает. ;)

Вот так мы определили метод для целых чисел: "заскочили" в класс **Integer**, описали там метод и "выскочили" из него обратно. Теперь у всех целых чисел есть этот (хотя и немного недоделанный) метод. Фактически, если вам не нравится, как работает какой-нибудь встроенный метод, например **to_s**, вы могли бы просто переопределить его примерно таким же образом... но я не советую это делать! Лучше всего оставить старые методы в покое и создавать новые, когда вам хочется сделать что-нибудь новенькое.

Ну что... всё ещё непонятно? Давайте, я ещё раз пройдусь по последней программе. До сих пор, когда мы выполняли какой-нибудь код или определяли какие-то методы, мы делали это в объекте по умолчанию под названием "программа". В нашей последней программе мы впервые покинули этот объект и проникли в класс **Integer**. Мы определили в нём метод (поэтому он стал методом для целых чисел), и все целые числа могут его использовать. Внутри этого метода мы использовали **self**, чтобы

ссылаться на объект (целое число), использующий этот метод.

Создание классов

Мы уже видели достаточно много объектов различных классов. Однако, легко предоставить себе такие объекты, которых в Ruby нет. К счастью, создать новый класс так же просто, как расширить существующий. Скажем, мы бы хотели сделать на Ruby игральные кости. Вот как мы могли бы создать класс Die:

```
class Die  #  игральная кость
  def roll
    1 + rand(6)
  end


end
#  Давайте создадим пару игральных костей...
dice = [Die.new, Die.new]
#  ...и бросим их.
dice.each do |die|
  puts die.roll
end
```

```
5

6
```

(Если вы пропустили раздел о случайных числах: **`rand(6)`** просто возвращает случайное число между **`0`** и **`5`**.)

Вот так! Наши собственноручно созданные объекты. Бросьте кости несколько раз (нажимая на кнопку "Обновить") и понаблюдайте, что при этом появится.

Мы можем определить для наших объектов самые разные методы... но здесь чего-то явно не хватает. Работа с этими объектами сильно напоминает программирование до того, как мы узнали о переменных. Взгляните на наши кости, например. Мы можем бросать их, и каждый раз при этом они выдают нам другое число. Но если мы хотели бы задержаться на этом числе, нам бы пришлось создать переменную, указывающую на это число. Кажется, любая порядочная игральная кость должна иметь возможность *хранить* число, а бросание кости должно изменять это число. Если мы уже отслеживаем состояние самой кости, то нам уже не нужно отслеживать где-то ещё число, которое она показывает.

Однако, если мы попытаемся сохранить полученное число в (локальной) переменной метода **`roll`**, оно исчезнет, как только закончится **`roll`**. Нам нужно хранить число в

переменной другого типа -

Переменные экземпляра

Обычно, когда мы хотим что-то сказать о строке, мы просто называем её строкой. Однако, мы могли также назвать её строковым объектом. Некоторые программисты могли бы назвать её экземпляром класса **String**, но это просто причудливый и длинный (так что можно запыхаться) способ сказать: "строка". Экземпляр класса — это просто объект этого класса.

Так что переменные экземпляра — это просто переменные объекта. Локальные переменные метода действуют до завершения метода. С другой стороны, переменные экземпляра каждого объекта будут действительны, пока существует объект. Чтобы отличить переменные экземпляра от локальных переменных, перед их именами ставится символ @:

```
class Die  #  игральная кость
  def roll
    @numberShowing = 1 + rand(6)
  end

  def showing
    @numberShowing
  end

end
die = Die.new
die.roll
puts die.showing
puts die.showing
die.roll
puts die.showing
puts die.showing
```

```
6
6
3
3
```

Очень хорошо! Так, метод **roll** бросает кость, а **showing** сообщает нам, какое число выпало. Однако, что же будет, если мы попытаемся посмотреть, что выпало прежде, чем мы бросили кость (прежде, чем мы задали значение **@numberShowing**)?

```
class Die  #  игральная кость
  def roll
    @numberShowing = 1 + rand(6)
  end

  def showing
    @numberShowing
  end

end
#  Поскольку я не собираюсь снова использовать эту кость,
#  мне не нужно сохранять её в переменной.
puts Die.new.showing
```

```
nil
```

Хммм... ладно, по крайней мере, она не выдала нам ошибку. Однако, в самом деле нет никакого смысла в том, что кость "не была брошена", или что бы там ни означало значение nil в этом случае. Было бы хорошо, если бы мы могли задать значение для нашего нового объекта "кость" сразу после того, как он был создан. Вот зачем нужен метод **initialize**:

```
class Die  #  игральная кость
  def initialize
    #  я просто брошу эту кость, хотя мы
    #  могли бы сделать что-нибудь ещё, если бы хотели,
    #  например, задать, что выпало число 6.
    roll
  end

  def roll
    @numberShowing = 1 + rand(6)
  end
```

```
  def showing
    @numberShowing
  end


end
puts Die.new.showing
```

```
2
```

Когда объект создаётся, всегда вызывается его метод **initialize** (если он у него определён).

Наши игральные кости теперь почти безупречны. Может быть, единственное, чего не хватает, так это способа задать, какой стороной выпала кость... Почему бы вам не написать метод **cheat**, который как раз это и делает! Вернётесь к чтению, когда закончите его (и, конечно, когда проверите, что он работает). Убедитесь, что невозможно задать, чтобы на кости выпало **7**!

Итак, мы только что прошли весьма крутой материал. Однако же, он довольно сложный, поэтому позвольте мне дать вам другой, более интересный пример. Ну, скажем, мы хотим сделать простое виртуальное домашнее животное — дракончика. Как большинство детей, он должен быть способен есть, спать и "гулять", что означает, что нам нужно будет иметь возможность кормить его, укладывать спать и выгуливать. Внутри себя нашему дракону понадобится отслеживать, когда он голоден, устал или ему нужно на прогулку; но у нас не будет возможности узнать это, когда мы будем общаться с нашим драконом: точно так же вы не можете спросить человеческого младенца: "Ты хочешь есть?". Мы также предусмотрим несколько других забавных способов для общения с нашим дракончиком, а когда он родится, мы дадим ему имя. (Что бы вы ни передали в метод **new**, для вашего удобства будет передано в метод **initialize**.) Ладно, давайте попробуем:

```
class Dragon


  def initialize name
    @name = name
    @asleep = false
    @stuffInBelly     = 10  #  Он сыт.
    @stuffInIntestine =  0  #  Ему не надо гулять.


    puts @name + ' родился.'
  end
```

```
def feed
  puts 'Вы кормите ' + @name + '(а).'
  @stuffInBelly = 10
  passageOfTime
end

def walk
  puts 'Вы выгуливаете ' + @name + '(а).'
  @stuffInIntestine = 0
  passageOfTime
end

def putToBed
  puts 'Вы укладываете ' + @name + '(а) спать.'
  @asleep = true
  3.times do
    if @asleep
      passageOfTime
    end
    if @asleep
      puts @name + ' храпит, наполняя комнату дымом.'
    end
  end
  if @asleep
    @asleep = false
    puts @name + ' медленно просыпается.'
  end
end

def toss
  puts 'Вы подбрасываете ' + @name + '(а) в воздух.'
```

```
  puts 'Он хихикает, обжигая при этом вам брови.'
  passageOfTime
end

def rock
  puts 'Вы нежно укачиваете ' + @name + '(а).'
  @asleep = true
  puts 'Он быстро задрёмывает...'
  passageOfTime
  if @asleep
    @asleep = false
    puts '...но просыпается, как только вы перестали качать.'
  end
end

private

#  "private" означает, что определённые здесь методы являются
#  внутренними методами этого объекта.  (Вы можете кормить
#  вашего дракона, но не можете спросить его, голоден ли он.)

def hungry?  #  голоден?
  #  Имена методов могут заканчиваться знаком "?".
  #  Как правило, мы называем так только, если метод
  #  возвращает true или false, как здесь:
  @stuffInBelly <= 2
end

def poopy?  #  кишечник полон?
 @stuffInIntestine >= 8
end
```

```
  def passageOfTime # проходит некоторое время
    if @stuffInBelly > 0
      #  Переместить пищу из желудка в кишечник.
      @stuffInBelly     = @stuffInBelly     - 1
      @stuffInIntestine = @stuffInIntestine + 1
    else  #  Наш дракон страдает от голода!
      if @asleep
        @asleep = false
        puts 'Он внезапно просыпается!'
      end
      puts @name + ' проголодался! Доведённый до крайности, он съедает
ВАС!'
      exit  #  Этим методом выходим из программы.
    end

    if @stuffInIntestine >= 10
      @stuffInIntestine = 0
      puts 'Опаньки!  ' + @name + ' сделал нехорошо...'
    end

    if hungry?
      if @asleep
        @asleep = false
        puts 'Он внезапно просыпается!'
      end
      puts 'В желудке у ' + @name + '(а) урчит...'
    end

    if poopy?
      if @asleep
        @asleep = false
        puts 'Он внезапно просыпается!'
```

```
      end
      puts @name + ' подпрыгивает, потому что хочет на горшок...'
    end
  end

end
pet = Dragon.new 'Норберт'
pet.feed
pet.toss
pet.walk
pet.putToBed
pet.rock
pet.putToBed
pet.putToBed
pet.putToBed
pet.putToBed
```

```
Норберт родился.
Вы кормите Норберт(а).
Вы подбрасываете Норберт(а) в воздух.
Он хихикает, обжигая при этом вам брови.
Вы выгуливаете Норберт(а).
Вы укладываете Норберт(а) спать.
Норберт храпит, наполняя комнату дымом.
Норберт храпит, наполняя комнату дымом.
Норберт храпит, наполняя комнату дымом.
Норберт медленно просыпается.
Вы нежно укачиваете Норберт(а).
Он быстро задрёмывает...
...но просыпается, как только вы перестали качать.
Вы укладываете Норберт(а) спать.
Он внезапно просыпается!
В желудке у Норберт(а) урчит...
```

```
Вы укладываете Норберт(а) спать.

Он внезапно просыпается!

В желудке у Норберт(а) урчит...

Вы укладываете Норберт(а) спать.

Он внезапно просыпается!

В желудке у Норберт(а) урчит...

Норберт подпрыгивает, потому что хочет на горшок...

Вы укладываете Норберт(а) спать.

Он внезапно просыпается!

Норберт проголодался! Доведённый до крайности, он съедает ВАС!
```

Вот тебе и раз! Конечно, было бы лучше, если бы это была интерактивная программа, но эти изменения вы можете сделать попозже. Я просто попытался показать те части программы, которые непосредственно относятся к созданию нового класса Dragon.

В этом примере мы увидели несколько новых конструкций. Первая достаточно проста: **`exit`** заканчивает программу "здесь и сейчас". Вторая — это ключевое слово **`private`**, которое мы вставили прямо в середину описания нашего класса. Я мог бы обойтись без него, но я хотел подчеркнуть мысль о том, что одни методы — это то, что *вы* можете делать с драконом, а другие — то, что просто происходит *внутри* дракона. Вы можете считать, что эти методы скрыты "под капотом": если вы не работаете автомехаником, всё, что на самом деле вам нужно знать, это педаль газа, педаль тормоза и рулевое колесо. Программист назвал бы их открытым интерфейсом вашей машины. Однако то, каким образом ваша аварийная подушка знает, когда наполниться воздухом, является внутренним поведением машины; обычному пользователю (водителю) не нужно знать об этом.

А сейчас в качестве более конкретного примера, иллюстрирующего эти строки, давайте поговорим о том, как вы могли бы представить автомобиль в видео-игре (чем я как раз, по случайному совпадению, и занимаюсь). Во-первых, вы захотели бы решить, каким должен выглядеть ваш внешний интерфейс; другими словами, какие методы смогут вызывать пользователи у ваших объектов-автомобилей? Ну, им понадобится нажимать на педаль газа и педаль тормоза, но им также понадобится указывать, с какой силой они нажимают на педаль. (Есть большая разница между "вдавить в пол" и "дотронуться".) Им также понадобится управлять, и снова потребуется возможность сказать, насколько сильно они поворачивают руль. Я полагаю, вы могли бы продолжить и добавить сцепление, сигналы поворота, реактивную установку, форсаж, конденсатор временного потока [главная деталь машины времени — *Прим. перев.*] и так далее... Это зависит от того, какую разновидность игры вы делаете.

Однако, нужно, чтобы внутри объекта-автомобиля происходило много чего другого; автомобилю нужны будут такие вещи, как скорость, направление и положение (и это только самые основные). Эти атрибуты могут изменяться нажатием на педали газа и тормоза и, конечно, поворачиванием руля, но пользователь не должен иметь возможности непосредственно устанавливать положение (что было бы подобно

сверхсветовому перемещению). Вы, должно быть, также пожелаете отслеживать боковые заносы и повреждения, отрыв всех колёс от земли и так далее. Всё это будет внутренностями вашего автомобильного объекта.

Попробуйте ещё кое-что

• Сделайте класс для апельсинового дерева — **`OrangeTree`**. У него должен быть метод **`height`**, возвращающий его высоту, и метод **`oneYearPasses`**, который при вызове увеличивает возраст дерева на один год. Каждый год дерево становится выше (как по-вашему, на какую высоту в год должно вырастать апельсиновое дерево?), а после определённого числа лет (опять же, как вы считаете) дерево должно умереть. Первые несколько лет оно не должно плодоносить, но через некоторое время должно, и мне кажется, что более старые деревья приносят каждый год больше плодов, чем молодые... настолько, насколько вы считаете это разумным. И, конечно, вам нужно иметь возможность сосчитать апельсины методом **`countTheOranges`** (который возвращает число апельсинов на дереве) и собирать их методом **`pickAnOrange`** (который уменьшает **`@orangeCount`** на единицу и возвращает строку с описанием, насколько вкусен был апельсин, или же он просто сообщает вам, что больше нет апельсинов для сбора в этом году). Удостоверьтесь, что те апельсины, что вы не собрали в этом году, опадут до следующего года.

• Напишите программу, в которой вы смогли бы взаимодействовать с вашим дракончиком. У вас должна быть возможность вводить такие команды, как `feed` и `walk`, и чтобы при этом вызывались нужные методы вашего дракона. Конечно, поскольку то, что вы вводите, это просто строки, вам понадобится некое подобие диспетчера методов, где ваша программа проверяет, какая строка была введена и затем вызывает соответствующий метод.

Вот почти что и всё, что можно сказать об этом! Нет, подождите секундочку... Я же не рассказал вам обо всех этих классах, которые выполняют самые разнообразные вещи: отправляют электронную почту, сохраняют и загружают файлы на ваш компьютер, или же создают окна и кнопки (и даже 3-хмерные миры) и всё прочее! Что ж, попросту имеется *настолько много* классов, которые вы можете использовать, что мне никак невозможно показать вам их все; я даже не знаю, что большинство из них из себя представляют! А что я *могу* сказать вам о них, так это то, где можно разузнать о них поподробнее, чтобы вы смогли изучить те из них, которые вы захотите применить в ваших программах. Однако, прежде, чем вы отправитесь на самостоятельное изучение, вам ещё следует узнать о других важных особенностях Ruby, которых нет в большинстве других языков, но без которых я просто не смог бы жить: о блоках и процедурных объектах.

10. Блоки и процедурные объекты

Это определённо одна из самых крутых возможностей Ruby. В некоторых других языках тоже есть такие возможности, хотя они могут называться как-нибудь по-другому (например, замыкания), но в большинстве даже более популярных языков, к их стыду, они отсутствуют.

Так что же это за новая крутая возможность? Это способность принимать блок кода (то есть код между **do** и **end**), обёртывать его в объект (называемый процедурным объектом или proc по-английски), сохранять его в переменной или передавать его в метод, а затем исполнять код этого блока, когда бы вы ни пожелали (более одного раза, если хотите). Таким образом, блок напоминает настоящий метод за исключением того, что он не привязан ни к какому объекту (он сам *является* объектом), и вы можете сохранять его или передавать его как параметр подобно тому, как вы это делаете с любым другим объектом. Думаю, настало время привести пример:

```
toast = Proc.new do

  puts 'Ваше здоровье!'

end

toast.call

toast.call

toast.call
```

```
Ваше здоровье!

Ваше здоровье!

Ваше здоровье!
```

Итак, я создал объект `proc` (это название, полагаю, означает сокращение от "procedure", т. е. "процедура", но гораздо более важно, что оно рифмуется с "`block`"), который содержит блок кода, затем я с помощью **`call`** вызвал `proc`-объект три раза. Как видите, это очень напоминает метод.

На самом деле, это даже более походит на метод, чем в показанном мной примере, так как блоки могут принимать параметры:

```
doYouLike = Proc.new do |aGoodThing|

  puts 'Я *действительно* люблю '+aGoodThing+'!'

end

doYouLike.call 'шоколад'

doYouLike.call 'рубин'
```

```
Я *действительно* люблю шоколад!

Я *действительно* люблю рубин!
```

Хорошо, вот мы узнали, что из себя представляют блоки и `proc`-и [читается: "проки" — *Прим. перев.*], и как их можно использовать, но в чём же здесь дело? Почему бы просто не использовать методы? Ну потому, что некоторые вещи вы просто не сможете сделать с помощью методов. В частности, вы не можете передавать методы в другие методы (но вы можете передавать в методы процедурные объекты), и методы не могут возвращать другие методы (но они могут возвращать `proc`-объекты). Это возможно просто потому, что `proc`-и являются объектами, а методы — нет.

(Между прочим, вам это не кажется знакомым? Вот-вот, вы уже видели блоки раньше... когда вы изучали итераторы. Но давайте поговорим об этом ещё чуточку попозже.)

Методы, принимающие процедурные объекты

Когда мы передаём процедурный объект в метод, мы можем управлять тем, как, в каком случае или сколько раз мы вызываем `proc`-объект. Например, имеется, скажем, нечто, что мы хотим сделать перед и после выполнения некоторого кода:

```
def doSelfImportantly someProc
  puts 'Всем немедленно ЗАМЕРЕТЬ!  Мне нужно кое-что сделать...'
  someProc.call
  puts 'Внимание всем, я закончил.  Продолжайте выполнять свои дела.'
end
sayHello = Proc.new do
  puts 'привет'
end
sayGoodbye = Proc.new do
  puts 'пока'
end
doSelfImportantly sayHello
doSelfImportantly sayGoodbye
```

```
Всем немедленно ЗАМЕРЕТЬ!  Мне нужно кое-что сделать...
привет
Внимание всем, я закончил.  Продолжайте выполнять свои дела.
Всем немедленно ЗАМЕРЕТЬ!  Мне нужно кое-что сделать...
пока
```

```
Внимание всем, я закончил.  Продолжайте выполнять свои дела.
```

Возможно, это не выглядит так уж особенно потрясающим... но это так и есть. :-) В программировании слишком часто имеются строгие требования к тому, что должно быть сделано и когда. Если вы хотите, например, сохранить файл, вам нужно открыть файл, записать туда информацию, которую вы хотите в нём хранить, а затем закрыть файл. Если вы позабудете закрыть файл, могут случиться "Плохие Вещи"™. Но каждый раз, когда вы хотите сохранить или загрузить файл, вам требуется делать одно и то же: открывать файл, выполнять то, что вы *действительно* желаете сделать, затем закрывать файл. Это утомительно и легко забывается. В Ruby сохранение (или загрузка) файлов работает подобно приведённому выше коду, поэтому вам не нужно беспокоиться ни о чём, кроме того, что вы действительно хотите сохранить (или загрузить). (В следующей главе я покажу вам, где разузнать, как делать такие вещи, как сохранение и загрузка файлов.)

Вы также можете написать методы, которые будут определять, сколько раз (или даже *при каком условии*) вызывать процедурный объект. Вот метод, который будет вызывать переданный ему `proc`-объект примерно в половине случаев, и ещё один метод, который будет вызывать его дважды:

```
def maybeDo someProc  # Условный вызов
 if rand(2) == 0
    someProc.call
  end
end
def twiceDo someProc  # Двойной вызов
 someProc.call
  someProc.call
end
wink = Proc.new do
  puts '<подмигнуть>'
end
glance = Proc.new do
  puts '<взглянуть>'
end
maybeDo wink
maybeDo glance
twiceDo wink
twiceDo glance
```

```
<подмигнуть>
```

```
<подмигнуть>

<взглянуть>

<взглянуть>
```

(Если вы перезагрузите эту страницу несколько раз [имеется ввиду страница оригинального учебника — *Прим. перев.*], то вы увидите другие результаты.) Это самые распространённые применения процедурных объектов, которые дают нам возможность делать такие вещи, которые мы просто не могли бы сделать, используя только методы. Конечно, вы могли бы написать метод, чтобы подмигнуть два раза, но вы не смогли бы написать метод, чтобы просто делать дважды *что-нибудь*!

Прежде, чем мы продолжим, давайте посмотрим на последний пример. До сих пор все передаваемые процедурные объекты были довольно похожи друг на друга. В этот раз они будут совсем другими, и вы увидите, насколько сильно подобный метод зависит от тех процедурных объектов, что были ему переданы. Наш метод примет обычный объект и процедурный объект, и вызовет процедурный объект с обычным объектом в качестве параметра. Если процедурный объект вернёт **`false`**, мы закончим выполнение, иначе мы вызовем процедурный объект с возвращённым объектом. Мы будем продолжать так делать, пока процедурный объект не вернёт **`false`** (что ему лучше сделать в конце концов, иначе программа "загнётся"). Этот метод вернёт последнее значение, возвращённое процедурным объектом, не равное **`false`**.

```
def doUntilFalse firstInput, someProc

  input  = firstInput

  output = firstInput



  while output

    input  = output

    output = someProc.call input

  end



  input

end

buildArrayOfSquares = Proc.new do |array| # Создание массива квадратов
чисел

  lastNumber = array.last

  if lastNumber <= 0

    false

  else

    array.pop                          # Уберём последнее число...
```

```
    array.push lastNumber*lastNumber # ...и заменим его на его квадрат...

    array.push lastNumber-1          # ...за которым идет предыдущее
число.

  end

end

alwaysFalse = Proc.new do |justIgnoreMe|

  false

end

puts doUntilFalse([5], buildArrayOfSquares).inspect

puts doUntilFalse('Я пишу это в 3 часа утра; кто-то меня вырубил!',
alwaysFalse)
```

```
[25, 16, 9, 4, 1, 0]

Я пишу это в 3 часа утра; кто-то меня вырубил!
```

Хорошо, признаю, что это был довольно странный пример. Но он показывает, насколько по-разному ведёт себя наш метод, когда ему передают совсем разные процедурные объекты.

Метод **inspect** во многом похож на **to_s** за исключением того, что возвращаемая им строка — это попытка показать код на Ruby для создания объекта, который вы ему передали. Здесь он показывает нам весь массив, возвращённый при нашем первом вызове метода **doUntilFalse**. Вы, должно быть, также заметили, что мы сами никогда не возводили в квадрат этот **0** в конце массива, но поскольку **0** в квадрате всегда равен **0**, нам это и не нужно было делать. А так как **alwaysFalse**, как вы знаете, возвращает всегда **false**, метод **doUntilFalse** ничего не делал, когда мы вызвали его во второй раз; он просто вернул то, что ему было передано.

Методы, возвращающие процедурные объекты

Ещё одна из крутых возможностей, которые можно делать с процедурными объектами, это то, что их можно создавать в методах, а затем возвращать их. Это делает возможным разнообразные сумасшедшие, но мощные программистские штучки (с впечатляющими названиями наподобие ленивое вычисление, бесконечные структуры данных и карринг). Но дело в том, что я почти никогда не использовал это на практике, а также не припомню, чтобы видел, как кто-либо применял это в своём коде. Думаю, это не такого рода вещи, которые обычно нужно делать на Ruby, а, может быть, Ruby просто подталкивает вас находить другие решения — не знаю. В любом случае, я только кратко коснусь этого.

В этом примере метод **compose** принимает два процедурных объекта и возвращает новый процедурный объект, который, будучи вызван, вызывает первый процедурный объект и передаёт его результат во второй.

```
def compose proc1, proc2
```

```
  Proc.new do |x|
    proc2.call(proc1.call(x))
  end
end
squareIt = Proc.new do |x|
  x * x
end
doubleIt = Proc.new do |x|
  x + x
end
doubleThenSquare = compose doubleIt, squareIt
squareThenDouble = compose squareIt, doubleIt
puts doubleThenSquare.call(5)
puts squareThenDouble.call(5)
```

```
100
50
```

Обратите внимание, что вызов **proc1** должен быть внутри скобок при вызове **proc2**, чтобы он был выполнен первым.

Передача блоков (не proc-объектов) в методы

Ну, хорошо, этот подход представляет чисто академический интерес, к тому же применять его несколько затруднительно. В основном трудность состоит в том, что здесь вам приходится выполнить три шага (определить метод, создать процедурный объект и вызвать метод с процедурным объектом); тогда как есть ощущение, что должно быть только два (определить метод и передать *блок* непосредственно в этот метод, совсем не используя процедурный объект), поскольку в большинстве случаев вы не хотите использовать процедурный объект / блок после того, как вы передали его в метод. Что ж, да будет вам известно, что в Ruby всё это уже сделано за нас! Фактически, вы уже делали это каждый раз, когда использовали итераторы.

Сначала я быстро покажу вам пример, а затем мы обсудим его.

```
class Array


  def eachEven(&wasABlock_nowAProc)
    isEven = true # Мы начинаем с "true", т.к. массив начинается с 0, а он
чётный.
```

```
    self.each do |object|
      if isEven
        wasABlock_nowAProc.call object
      end

      isEven = (not isEven)  #  Переключиться с чётного на нечётное или
наоборот.
    end
  end
end
['яблоками', 'гнилыми яблоками', 'вишней', 'дурианом'].eachEven do |fruit|
  puts 'Мммм!  Я так люблю пирожки с '+fruit+', а вы?'
end
#  Помните, что мы берём элементы массива с чётными номерами,
#  все из которых оказываются нечётными числами; это
#  просто потому, что мне захотелось создать подобные трудности.
[1, 2, 3, 4, 5].eachEven do |oddBall|
  puts oddBall.to_s+' - НЕ чётное число!'
end
```

```
Мммм!  Я так люблю пирожки с яблоками, а вы?
Мммм!  Я так люблю пирожки с вишней, а вы?
1 - НЕ чётное число!
3 - НЕ чётное число!
5 - НЕ чётное число!
```

Итак, всё, что мы должны сделать, чтобы передать блок в метод **eachEven**, это "прилепить" блок после метода. Подобным же образом вы можете передать блок в любой метод, хотя многие методы просто проигнорируют блок. Чтобы заставить ваш метод *не* игнорировать блок, а взять его и превратить его в процедурный объект, нужно поместить имя процедурного объекта в конце списка параметров вашего метода и поставить перед ним амперсанд (**&**). Конечно, это немного мудрёно, но не слишком, и вам придётся сделать это только один раз (когда вы описываете метод). А затем вы можете использовать этот метод снова и снова точно так же, как и встроенные методы, принимающие блоки такие, как **each** и **times**. (Помните, **5.times do**...?)

Если для вас это слишком запутанно, просто помните, что должен сделать **eachEven**:

вызвать переданный ему блок для каждого чётного элемента в массиве. После того, как однажды вы написали метод и убедились, что он работает, вам уже не нужно думать о том, что в действительности делается "под капотом" ("какой блок и когда вызывается??"). На самом деле, именно *поэтому* мы пишем подобные методы: чтобы нам никогда не приходилось снова думать о том, как они работают. Мы просто используем их.

Помню, один раз я захотел сделать, чтобы можно было измерять, сколько времени выполняются различные секции программы. (Это также известно как профилирование программного кода.) И я написал метод, который засекает время перед исполнением кода, затем выполняет его, в конце снова засекает время и вычисляет разницу. Сейчас я не могу найти этот метод, но мне он и не нужен; он, возможно, выглядел примерно так:

```
def profile descriptionOfBlock, &block  #  Описание блока и сам блок
  startTime = Time.now

  block.call

  duration = Time.now - startTime

  puts descriptionOfBlock+':  '+duration.to_s+' сек.'
end
profile '25000 удваиваний' do
  number = 1

  25000.times do
    number = number + number
  end

  puts number.to_s.length.to_s+' цифр' # Да, это число цифр в таком
ГИГАНТСКОМ числе.
end
profile 'сосчитать до миллиона' do
  number = 0

  1000000.times do
    number = number + 1
```

```
  end

end
```

```
7526 цифр

25000 удваиваний:  0.304966 сек.

сосчитать до миллиона:  0.615216 сек.
```

Как просто! Как элегантно! Теперь с помощью этого крошечного метода я легко могу измерить время работы любой секции в любой программе, в какой только захочу: я просто закину код в блок и отправлю его методу **`profile`**. Что может быть проще? В большинстве языков мне понадобилось бы явно добавлять код для измерения времени (тот, что написан в **`profile`**) до и после каждой секции, которую я хотел бы захронометрировать. В то время как в Ruby я всё держу в одном-единственном месте и (что более важно) отдельно от всего остального!

Попробуйте ещё кое-что

• *Дедушкины часы*. Напишите метод, который принимает блок и вызывает его один раз для каждого часа, который прошёл сегодня. Таким образом, если я бы передал ему блок **`do puts 'БОМ!' end`**, он бы отбивал время (почти) как дедушкины часы. Проверьте ваш метод с несколькими различными блоками (включая тот, что я вам дал). **Подсказка:** *Вы можете использовать* **`Time.now.hour`**, *чтобы получить текущий час. Однако, он возвращает число между* **`0`** *и* **`23`**, *поэтому вам придётся изменить эти числа, чтобы получить обычные числа, как на циферблате (от* **`1`** *до* **`12`**).

• *Протоколирование программ*. Напишите метод под названием **`log`**, который принимает строку описания блока и, конечно, сам блок. Подобно методу **`doSelfImportantly`**, он должен выводить с помощью **`puts`** строку, сообщающую, что он начал выполнение блока, и ещё одну строку в конце, сообщающую, что он закончил выполнение блока, а также сообщающую вам, что вернул блок. Проверьте ваш метод, отправив ему блок кода. Внутри этого блока поместите *другой* вызов метода **`log`**, передав ему другой блок. (Это называется вложенностью.) Другими словами, ваш вывод должен выглядеть примерно так:

```
Начинаю "внешний блок"...

Начинаю "другой небольшой блок"...

..."другой небольшой блок" закончен, вернул:  5

Начинаю "ещё один блок"...

..."ещё один блок" закончен, вернул:  Мне нравится тайская еда!

..."внешний блок" закончен, вернул:  false
```

• *Улучшенное протоколирование*. Вывод из предыдущего метода **`log`** было трудновато читать, и было бы тем хуже, чем больше была бы вложенность. Было бы гораздо легче читать, если бы он делал отступы в строках для внутренних блоков. Чтобы это сделать, вам понадобится проверять, насколько глубоко вложен вызов

метода перед тем, как **log** хочет что-нибудь напечатать. Чтобы сделать это, примените глобальную переменную, т. е. переменную, которую вы можете видеть из любого места вашего кода. Чтобы сделать переменную глобальной, просто поставьте перед именем вашей переменной символ **$**, вот так: **$global**, **$nestingDepth** и **$bigTopPeeWee**. В конце концов, ваша программа протоколирования должна выводить примерно вот что:

```
Начинаю "внешний блок"...

  Начинаю "другой небольшой блок"...

    Начинаю "маленький-премаленький блок"...

    ..."маленький-премаленький блок" закончен, вернул:  море любви

  ..."другой небольшой блок" закончен, вернул:  42

  Начинаю "ещё один блок"...

  ..."ещё один блок" закончен, вернул:  Я люблю индийскую еду!

..."внешний блок" закончен, вернул:  true
```

Ну вот почти и всё, что вы намеревались узнать из этого учебника. Мои поздравления! Вы изучили *очень много*! Возможно, вам не кажется, что вы помните всё, или же вы пропустили некоторые части... Ну и ладно, это нормально. Программирование — это не то, что вы знаете; это то, что вы можете вычислить. Покуда вы знаете, где найти то, что вы позабыли, у вас будет всё в порядке. Надеюсь, вы не думаете, что я написал всё это, не заглядывая куда-нибудь время от времени? Я именно так и делал. Мне также много помогали с кодом, выполняющим все примеры в этом учебнике. Но куда же *я* заглядывал и кого *я* просил о помощи? Давайте, я покажу вам...

11. Что не вошло в учебник

Итак, куда мы сейчас отправимся? Кого можно спросить, если у вас возник вопрос? А если вы хотите, чтобы ваша программа открывала web-страничку, отправляла электронную почту или масштабировала цифровое изображение? Что ж, есть полным-полно мест, где найдется помощь по Ruby. Но, к сожалению, такой ответ вам не слишком поможет, не так ли? :-)

Что касается меня, то на самом деле есть только три места, где я ищу помощи по Ruby. Если это небольшой вопрос, и я полагаю, что я могу сам поэкспериментировать, чтобы найти на него ответ, то я использую irb. Если это вопрос посерьёзнее, я обращаюсь к своей киркомотыге. А если я никак не могу разобраться с ним сам, то я прошу помощи в ruby-talk.

IRB: интерактивный интерпретатор Ruby

Если вы установили Ruby, то вы также установили `irb`. Чтобы его запустить, нужно просто перейти в командное окно и напечатать `irb`. Когда вы находитесь в сеансе `irb`, вы можете вводить любые выражения языка Ruby, какие пожелаете, а он будет выдавать вам их значения. Введите `1 + 2`, и он выдаст вам 3. (Обратите внимание, что вам не нужно использовать **`puts`**.) Это похоже на гигантский калькулятор на Ruby. Когда вы закончите, просто введите команду `exit`.

В `irb` имеется гораздо больше этого, но обо всём этом вы можете узнать в "киркомотыге".

Киркомотыга: "Программирование на Ruby"

Абсолютно *именно та* книга по Ruby, которая нужна всем — это "Программирование на Ruby: Руководство прагматичного программиста", написанная Дэвидом Томасом и Эндрю Хантом (Программисты-прагматики). Хотя я очень рекомендую достать 2-е издание этой замечательной книги, где освещены все последние возможности Ruby, вы также можете взять немного более старую (но до сих пор в основном подходящую) версию, бесплатно доступную в Сети. (На самом деле, если вы установили версию Ruby для Windows, то она у вас уже имеется.)

В этой книге вы можете найти о руби Ruby почти всё, от основ до самых передовых возможностей. Она легко читается; она исчерпывающая; она почти безупречна. Хотелось бы, чтобы для всякого языка имелась бы книга подобного качества. В конце книги вы найдёте огромный раздел, где подробно описан каждый метод каждого класса, приведены объяснения и примеры. Я просто полюбил эту книгу!

Её можно найти в нескольких местах (включая собственный сайт Программистов-прагматиков), но мне больше всего нравится сайт ruby-doc.org. В этой версии слева имеется симпатичное оглавление, а также предметный указатель. (На ruby-doc.org

есть также много другой документации, например, о базовом API и стандартной библиотеке... В основном, там есть готовые к использованию документы обо всём, что касается Ruby. Проверьте сами.)

А почему же она называется "киркомотыга" ("the pickaxe")? Ну, там на обложке книги есть картинка киркомотыги. Мне кажется, это глупое название, но оно уже "прилипло".

Ruby-Talk: список рассылки о Ruby

Даже имея irb и *киркомотыгу*, вы иногда всё-таки не можете разобраться с чем-нибудь. Или, возможно, вы хотите знать, не делал ли уже кто-нибудь то, над чем вы сейчас работаете, чтобы выяснить, можно ли вам этим воспользоваться. В этих случаях вам нужно обратиться именно в ruby-talk, список рассылки о Ruby. В нём полно дружелюбных, умных, отзывчивых людей. Чтобы побольше узнать о нём или подписаться на него, взгляните сюда.

ПРЕДУПРЕЖДЕНИЕ: В этой рассылке каждый день приходит *много* почты. У меня она автоматически пересылается в другой почтовый ящик, поэтому она мне не мешает. Если же вы не желаете иметь дело со всей этой почтой, то это вам и не нужно! Список рассылки ruby-talk зеркалируется в новостную группу comp.lang.ruby, и наоборот, так что там вы можете увидеть те же самые сообщения. Любым из этих способов вы увидите одни и те же сообщения, просто немного в разных форматах. [Web-интерфейс к архиву этого списка рассылки находится по адресу: www.ruby-forum.com/forum/4. — *Прим. перев.*]

Tim Toady

То, от чего я старался уберечь вас, но с чем вы непременно скоро столкнётесь, это принцип TMTOWTDI (произносится "Тим Тоуди") или "There's More Than One Way To Do It", что значит "Есть не один способ сделать что-либо".

В то время как одни будут говорить вам, какая замечательная вещь этот TMTOWTDI, другие относятся к нему совсем по-другому. На самом деле, у меня в основном не возникает по этому поводу никаких сильных эмоций, но я считаю, что это *ужасный* метод обучить кого-то, как нужно программировать. (Как будто научить делать что-то одним способом это само по себе не достаточно трудоёмкое и сложное дело!)

Однако теперь, когда вы выходите за рамки того учебника, вы будете читать гораздо более разнообразные программы. Например, мне приходят на ум по крайней мере пять различных способов создания строки (помимо заключения некоторого текста в одинарные кавычки), и каждый из них работает немного по-другому. Я показал вам только самый простой из этих шести.

А когда мы говорили о ветвлении, я показал вам **`if`**, но не показал **`unless`**. Предоставляю вам выяснить что это такое в `irb`.

Ещё одно приятное сокращение, которое вы можете использовать для **`if`**, **`unless`** и **`while`**, это симпатичная однострочная версия:

```ruby
#  Эти слова взяты из программы, которую я написал для генерирования

#  англоподобной болтовни.  Круто, да?

puts 'grobably combergearl kitatently thememberate' if 5 == 2**2 + 1**1

puts 'enlestrationshifter supposine follutify blace' unless 'Chris'.length
== 5
```

```
grobably combergearl kitatently thememberate
```

И наконец, есть ещё один способ писать методы, которые принимают блоки (а не процедурные объекты). Мы видели это, когда мы захватывали блок и превращали его в процедурный объект, используя трюк с **`&block`** в списке параметров при определении функции. Тогда, чтобы вызвать блок, вы просто используете **`block.call`**. Ладно, есть способ покороче (хотя лично я нахожу его более запутанным). Вместо этого:

```ruby
def doItTwice(&block)

  block.call

  block.call

end

doItTwice do

  puts 'murditivent flavitemphan siresent litics'

end
```

```
murditivent flavitemphan siresent litics

murditivent flavitemphan siresent litics
```

...вы делаете так:

```ruby
def doItTwice

  yield

  yield

end

doItTwice do

  puts 'buritiate mustripe lablic acticise'

end
```

```
buritiate mustripe lablic acticise

buritiate mustripe lablic acticise
```

Ну, не знаю... а что вы об этом думаете? Возможно, так только я так считаю, но... **`yield`**?! Если бы это было что-нибудь наподобие **`call_the_hidden_block`** или что-то в этом роде, что бы имело *немного* больше смысла для меня. Многие люди говорят, что **`yield`** для них имеет смысл. Но я думаю, что принцип TMTOWTDI

предполагает вот что: они делают что-то по-своему, а я делаю это по-моему.

КОНЕЦ

Используйте всё это во благо, а не во зло. :-) И если вы находите этот учебник полезным (или запутанным, либо если вы нашли ошибку), дайте мне знать!