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, тикание 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
Глава 2:  Буквы                         page 72
Глава 3:  Переменные                    page 118
```

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!