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 подминуть 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 быдут 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 нигогда 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...