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.