Глава 2
Программирование == Радость: Увлекательный экскурс в Ruby и Объектное Ориентирование.

В зависимости от того, кого вы спрашиваете, программирование - это и наука, и искусство. Чтобы указывать компьютерам, что делать с компьютерными программами, нужно уметь мыслить аналитически, как ученый, и концептуально, как художник. Быть художником важно для того, чтобы придумывать грандиозные идеи и быть достаточно гибким, чтобы применять уникальные подходы. Быть ученым важно для понимания того, как и почему необходимо учитывать определенные методологии, и подходить к тестированию и отладке с логической точки зрения, а не с эмоциональной.

К счастью, вам не обязательно быть художником или ученым. Как и в случае с тренировкой тела, упражнения по программированию и размышления о том, как решать проблемы, тренируют ум, чтобы стать лучшим программистом. Программировать может научиться каждый. Самыми большими препятствиями являются отсутствие мотивации и целеустремленности или излишний уровень сложности на ранних этапах. Ruby - один из самых простых языков программирования для изучения, поэтому остается мотивация и целеустремленность. Вы, вероятно, купили эту книгу с целью создания определенная программа, веб-приложение или решение определенной задачи - вот ваша мотивация, остается только приверженность делу. Чтобы помочь охватить аспект приверженности, мы постараемся, чтобы все было гладко и несложно.

Я надеюсь, что к тому времени, когда вы дочитаете эту главу до конца, вы сможете ощутить всю прелесть, которая ждет вас впереди, благодаря
знанию мощного, но обманчиво простого языка программирования, и вам захочется создавать свои собственные вещи!

■ Обратите внимание, что в отличие от последующих глав, эта глава не имеет формата инструкции. Вместо этого я буду быстро переходить от концепции к концепции, чтобы дать вам представление о Ruby как о языке, прежде чем мы перейдем к деталям позже.

Маленькие шаги
В главе 1 вы сосредоточились на установке Ruby, чтобы ваш компьютер мог понимать язык. В конце главы вы загрузили программу под названием irb.

irb: Interactive Ruby

irb расшифровывается как “Интерактивный Ruby”. “Интерактивный” означает, что как только вы вводите что-либо и нажимаете *Enter,* ваш компьютер немедленно пытается это обработать. Иногда такого рода среду называют непосредственной или интерактивной средой.

■ Примечание. Если вы не можете вспомнить, как загрузить *irb*, обратитесь к разделу главы 1, посвященному операционной системе вашего компьютера.

Запустите *irb* и убедитесь, что появилось приглашение, например:
irb(main):001:0>

Это приглашение не такое сложное, как кажется. Все, что оно означает, это то, что вы находитесь в программе *irb* и вводите вашу первую строку (001), и вы находитесь на глубине 0. В данный момент вам не нужно придавать значения элементу глубины.

Введите это значение после предыдущего запроса и нажмите *Enter*:

```
1 + 1
```

Результат должен появиться быстро: 2. Весь процесс выглядит следующим образом:

```
irb(main):001:0> 1 + 1

result: 2

irb(main):002:0>
```

Теперь Ruby готов принять от вас другую команду или выражение.

Как начинающий программист на Ruby, вы потратите много времени на тестирование концепций *irb* и ознакомление с Ruby. Это идеальная среда для настройки и тестирования языка.

Интерактивная среда *irb* дает вам возможность получать мгновенную обратную связь — важный инструмент при обучении. Вместо того чтобы писать программу в текстовом редакторе, сохранять ее, запускать на компьютере, а затем просматривать ошибки, чтобы понять, где вы допустили ошибку, вы можете просто ввести небольшие фрагменты кода, нажать *Enter* и сразу увидеть, что произойдет.

Если вы хотите поэкспериментировать дальше, попробуйте другие арифметические действия, такие как *100 * 5, 57 + 99, 10 – 50*, или *100/10* (если последнее кажется вам непривычным, то в Ruby символ косой черты / является оператором для деления).

Ruby - это “Английский для компьютеров”.

На самом низком уровне компьютерные процессоры состоят из транзисторов, которые реагируют на электронные сигналы и воздействуют на них, но обдумывание выполнения операций на этом уровне требует много времени и сложностей, поэтому мы склонны использовать “языки” более высокого уровня для передачи наших намерений, во многом так же, как мы это делаем с естественными языками, такими как английский.

Компьютеры могут понимать языки, хотя и несколько иным образом, чем это делает большинство людей. Компьютерам требуются языки с логическими структурами и четко определенным синтаксисом, чтобы было логически понятно, что вы приказываете компьютеру делать.

Ясность необходима, потому что почти все, что вы передаете компьютеру во время программирования, является инструкцией (или командой). Инструкции являются основными строительными блоками всех программ, и для того, чтобы компьютер мог выполнять их должным образом, намерения программиста должны быть ясными и четкими. Многие сотни всех этих инструкций объединены в программы, которые выполняют определенные задачи, а это означает, что вероятность ошибки невелика.

Вам также необходимо учитывать, что другим программистам нужно будет поддерживать написанные вами компьютерные программы. (Если вы не программируете конечно просто для развлечения), но важно, чтобы ваши программы были просты для понимания, чтобы вы могли разобраться в них, когда вернетесь к ним позже.

Почему Ruby - отличный язык программирования

Хотя английский был бы плохим языком программирования, из-за своей неоднозначности и сложности Ruby иногда может казаться удивительно похожим на английский. Ruby - всего лишь один из сотен языков программирования, но это особенный, потому что для многих программистов он очень похож на естественный язык, но при этом обладает ясностью, необходимой компьютерам. Рассмотрим этот пример кода:

10.times do print "Hello, world!" end

Прочитайте этот код вслух (это действительно помогает!). Он звучит не так хорошо, как английский, но смысл должен быть понятен сразу. Программа просит компьютер “10 раз” “напечатать” *"Hello, world!"* на экране. Это работает. Если у вас запущен irb, введите предыдущий код и нажмите Enter, чтобы увидеть результаты:

result: Hello, world!Hello, world!Hello, world!Hello, world!Hello, world!Hello, world!Hello, world!Hello, world!Hello, world!Hello, world!

Если вы прочтете код вслух, то результат (“Привет, мир!”, напечатанный десять раз) не должен вас удивлять.

■ Примечание. Опытные программисты могут удивиться, почему в конце предыдущего примера кода нет точки с запятой. В отличие от многих других языков, таких как Perl, PHP, C и C++, в Ruby точка с запятой в концестрок не нужна (хотя, если вы ее используете, это не повредит). Поначалу может потребоваться некоторое время, чтобы привыкнуть к этому, но для начинающим программистам изучение Ruby становится еще проще.

Вот гораздо более сложный пример, который может возникнуть в реальном веб-приложении:

user = User.find_by_email('me@privacy.net')
user.country = 'Belgium'

■ Примечание
Не копируйте и не вставляйте этот код. Он не будет работать вне контекста конкретного приложения.

Этот код далеко не так очевиден, как в примере с *"Hello, world!"* , но вы все равно сможете догадаться, что он делает. Сначала он сообщает компьютеру, что вы хотите работать с понятием, называемым u*ser*. Затем он пытается найти пользователя с указанным адресом электронной почты. И, наконец, он изменяет данные о стране пользователя на Бельгию. Не беспокойтесь о том, как хранятся данные пользователей на данный момент; это произойдет позже.

Это достаточно продвинутый и абстрактный пример, но он демонстрирует единую концепцию потенциально сложного приложения, в котором вы можете иметь дело с различными понятиями, такими как “пользователи”. В конце этой главы вы увидите, как вы можете создавать свои собственные концепции в Ruby и оперировать ими аналогично приведенному в этом примере. Ваш код будет читаться почти так же легко, как и английский.

Познавательные маршруты.

Обучение само по себе может быть увлекательным занятием, но простое чтение о чем-то не сделает вас экспертом в этом деле. Я прочитала несколько кулинарных книг, но, кажется, это не улучшает качество моей готовки, хотя я время от времени пробую это делать. Не хватает только экспериментов и тестирования, так как без них ваши усилия в лучшем случае будут носить академический характер.

Помня об этом, важно настроиться на эксперименты и тестирование с первого дня использования Ruby. На протяжении всей книги я буду просить вас опробовать различные блоки кода и поиграть с ними, чтобы убедиться, что вы получите желаемые результаты. Время от времени вы будете удивлять себя, а иногда и загонять свой код в тупик; все это - часть удовольствия. Что бы ни случилось, все хорошие программисты учатся на экспериментах, и вы можете овладеть языком и концепциями программирования, только экспериментируя по ходу дела.

Эта книга проведет вас через лес кода и концепций, но без самостоятельного тестирования и подтверждения правильности кода вы можете быстро заблудиться. Часто используйте irb и другие инструменты, о которых я расскажу, и экспериментируйте с кодом как можно больше, чтобы закрепить полученные знания.

Введите следующий код в командной строке irb и нажмите Enter:

```
print "test"
```

Результат будет простым:

```
test => nil
```

Логично, что при выводе "*test"* результаты теста будут выведены на экран. Однако суффикс *=> nil* является результатом работы вашего кода в виде выражения (подробнее об этом в главе 3). Это происходит потому, что все строки кода в Ruby состоят из выражений, возвращающих значения. Однако функция *print* выводит данные на экран, а не возвращает какое-либо значение в виде выражения, поэтому вы получаете значение *nil.* Подробнее об этом в главе 3. На данном этапе это вполне нормально, если вы немного запутались в этом вопросе.

Давайте попробуем что-нибудь другое:

```
print "2+3 is equal to " + 2 + 3
```

На первый взгляд эта команда кажется логичной. Если 2 + 3 равно 5 и вы добавляете это в конец строки "2+3 равно ", то должно получиться "2+3 равно 5", верно? К сожалению, вместо этого вы получаете эту ошибку:

```
TypeError: no implicit conversion of Fixnum into String
from (irb):45:in `+'
from (irb):45
from :0
```

Ruby жалуется, когда вы допускаете ошибку, а здесь он жалуется на то, что вы не можете преобразовать число в строку (где “строка” - это набор текста, такой как это предложение). Числа и строки не могут быть смешаны таким образом. Расшифровка причины пока не важна, но эксперименты, подобные этому, помогут вам узнать и запомнить о Ruby больше, чем чтение этой книги в одиночку. При возникновении подобной ошибки вы можете использовать сообщение об ошибке как подсказку к решению, независимо от того, найдете ли вы его в этой книге, в Интернете или обратитесь к другому разработчику.

В качестве побочного действия скопируйте и вставьте ошибку *“no implicit conversion of Fixnum into String”* в Google и посмотрите, что произойдет. Большинство программистов часто делают это.

Промежуточным решением предыдущей задачи было бы сделать это:

```
print "2+3 is equal to "
print 2 + 3
```

Или это:

```
print "2+3 is equal to " + (2 + 3).to_s
```

Попробуйте оба варианта.

Давайте попробуем еще один пример. Как насчет того, чтобы разделить 10 на 3?

```
irb(main):002:0> 10/3
```

result: 3

Предполагается, что компьютеры должны быть точными, но любой, кто обладает базовыми навыками арифметики, знает, что 10, деленное на 3, равно 3,33, а не 3!

Причина любопытного результата в том, что по умолчанию Ruby принимает число, такое как 10 или 3, за число, равное *integer* — целое число. Арифметика с целыми числами в Ruby дает целочисленные результаты, поэтому необходимо указать В Ruby число с плавающей запятой (число с десятичными разрядами), чтобы получить ответ с плавающей запятой, например 3.33. Вот пример того, как это сделать:

```
Irb (main):001:0> 10.0 / 3
result: 3.3333333333333
```

Подобные неочевидные результаты делают тестирование и эксперименты не только хорошим инструментом обучения, но и необходимой тактикой при разработке более крупных программ.

На данный момент достаточно ошибок. Давайте создадим что-нибудь полезное!

Воплощая идеи в коде на Ruby.

Часть мастерства программирования заключается в умении воплощать свои идеи в компьютерные программы. Как только вы овладеете языком программирования, вы сможете воплощать свои идеи непосредственно в коде. Однако, прежде чем вы сможете это сделать, вам нужно понять, как Ruby понимает концепции реального мира и как вы можете передать свои идеи в форме, которая будет понятна Ruby.

Как Ruby понимает концепции с помощью объектов и классов

Ruby - это объектно-ориентированный язык программирования. В простейшем смысле это означает, что ваш Ruby программы могут определять понятия и оперировать ими таким образом, чтобы имитировать то, как мы могли бы работать с понятиями в реальном мире. Ваша программа может содержать такие понятия, как “люди”, “ящики”, “билеты”, “карты” или любые другие понятия, с которыми вы хотите работать. Объектно-ориентированные языки упрощают реализацию этих концепций таким образом, что вы можете создавать объекты на их основе. Будучи объектно-ориентированным языком, Ruby может использовать и понимать взаимосвязи между этими концепциями любым доступным вам способом.

Например, вы можете захотеть создать приложение, которое может управлять бронированием билетов на спортивные мероприятия. К таким понятиям относятся “события”, “люди”, “билеты”, “места проведения” и так далее. Ruby позволяет вам внедрять эти концепции непосредственно в ваши программы, создавать их экземпляры объектов (экземплярами “события” могут быть Суперкубок или финал Чемпионата мира по футболу 2018 года), а также выполнять операции и определять отношения между ними. Используя все эти концепции в своей программе, вы можете быстро связать “мероприятия” с “местами проведения” и это означает, что ваш код с самого начала формирует логическую систему.

Если вы раньше мало программировали, идея взять реальные концепции и использовать их непосредственно в компьютерной программе может показаться очевидным способом упростить разработку программного обеспечения. Однако объектное ориентирование - довольно новая идея в разработке программного обеспечения (эта концепция была разработана в 1960-х годах, но в массовом программировании она стала популярной только в 1990-х годах). В случае с не объектно–ориентированными языками программисту приходится использовать более ручной подход для обработки концепций и взаимосвязей между ними, и хотя это добавляет больше контроля, оно также создает дополнительную сложность.

The Making of a Person

Давайте сразу перейдем к исходному коду, демонстрирующему простую концепцию *person:*

```
class Person
  attr_accessor :name, :age , :gender
end
```

Раньше Ruby казался очень похожим на английский, но при определении понятий он не очень похож на английский.
Давайте рассмотрим это шаг за шагом:

class Person

С этой строки вы начнете определять понятие “*person*”. Когда мы определяем понятия в Ruby (или в большинстве других объектно-ориентированных языков, если на то пошло), мы называем их классами. Класс — это «заготовка» для штамповки объектов. Имена классов в Ruby всегда начинаются с заглавной буквы, поэтому в конечном итоге ваши программы будут содержать классы с такими именами, как User, Person, Place, Topic, Message и т. д.

attr_accessor :name, :age, :gender

В предыдущей строке указаны три атрибута для класса Person. У отдельного пользователя есть имя, возраст и пол, и эта строка создает эти атрибуты. *attr* расшифровывается как “атрибут”, а *accessor* примерно означает “сделать эти атрибуты доступными для установки и изменения по своему усмотрению”. Это означает, что когда вы работая с объектом *Person* в своем коде, вы можете изменить имя, возраст и пол этого человека (или, точнее, атрибуты имени, возраста и пола объекта).

end

Конечная строка должна быть явно полезной. Это совпадает с определением класса в первой строке и сообщает Ruby, что вы больше не определяете класс *Person*.

Напомним, что класс определяет концепцию (например, личность), а объект - это отдельная вещь, основанная на классе (например, “*Chris*” или “*Mrs. Smith*”).

Итак, давайте поэкспериментируем с нашим классом Person. Перейдите в свой запрос irb и введите найденный класс Person ранее. Ваши усилия должны выглядеть примерно так:

i*rb(main):001:0> class Person*
irb(main):002:?>
attr_accessor :name, :age, :gender
irb(main):003:?>
end
=> nil
irb(main):004:0>

Вы заметите, что *irb* распознает, когда вы были “внутри” определения класса, потому что автоматически создает отступы в вашем коде.

Как только вы закончите определение класса и Ruby обработает его, будет возвращено значение *nil*, потому что определение класса не приводит к возвращаемому значению, а *nil* - это способ представления Ruby “ничего”. Поскольку ошибок не было, ваш класс Person теперь существует в Ruby, так что давайте что-нибудь с ним сделаем:

person_instance = Person.new
=> #<Person:0x007fbb0c625f88>

Что делает первая строка, так это создает “новый” экземпляр класса *Person*, таким образом, вы создаете “нового человека” и присваиваете ему значение *person_instance* — заполнитель, представляющий нового человека, известный как переменная.
Вторая строка - это реакция Ruby на создание нового пользователя, и на данном этапе она не важна. Бит *0x358ea8* будет отличаться от компьютера к компьютеру и представляет собой только внутреннюю ссылку, которую Ruby присвоил новому пользователю. Вам вовсе не обязательно принимать это во внимание.

Давайте сразу же сделаем что-нибудь с *person_instance*:
person_instance.name = "Christine"

В этом простом примере вы ссылаетесь на атрибут person_instance name и присваиваете ему значение *"Christine"*. Вы только что присвоили своему персонажу имя. Класс *Person* имеет два других атрибута: возраст и пол. Давайте установим их:

person_instance.age = 52
person_instance.gender = "female"

Вот так просто! Вы присвоили *person_instance* базовые идентификаторы. Как насчет того, чтобы вывести имя пользователя обратно на экран?

puts person_instance.name

При нажатии клавиши Enter появляется имя: *"Christine"*. Попробуйте проделать то же самое с *age* и *gender*.

■ Обратите внимание, что в предыдущих примерах вы использовали print для отображения объектов на экране. В предыдущем примере вы использовали *puts*. Разница между *print* и *puts* заключается в том, что *puts* автоматически перемещает курсор вывода на следующей строки (то есть добавляется символ новой строки для начала новой строки), в то время как *print* продолжает выводить текст на ту же строку, что и в предыдущий раз. Как правило, вы

захотите использовать *puts*, но я использовал *print*, чтобы сделать предыдущие примеры более понятными при чтении их вслух.

Основные переменные

В предыдущем разделе вы создали пользователя и присвоили ему переменную с именем *person_instance*.

Переменные - важная часть программирования, и их легко понять, особенно если вы обладаете минимальными знаниями в области алгебры. Рассмотрим это:

x = 10

В этом коде переменной x присваивается значение 10. Поскольку x теперь равно 10, вы можете сделать что-то вроде этого:

*x * 2*

result: 20

■ Примечание. Некоторых начинающих программистов может смутить определение = как средства, присваивающего значение, а не как показателя равенства. Когда мы говорим, что x = 10, мы **не** имеем в виду, что x и 10 равны, но теперь следует считать, что x относится к значению 10.

Переменные в Ruby могут относиться к любым понятиям, связанным со значениями, которые понимает Ruby, таким как числа, текст и другие структуры данных, о которых я расскажу в этой книге. В предыдущем разделе *person_instance* был переменной который ссылается на экземпляр объекта класса Person, подобно тому, как x - это переменная, содержащая число 10. Проще говоря, рассмотрим *person_instance* как имя, которое ссылается на конкретный, уникальный объект Person.

Когда вы хотите сохранить что-либо и использовать это в нескольких строках программы, вы будете использовать переменные в качестве временных хранилищ для данных, с которыми работаете.

От людей до домашних животных

Ранее вы создали простой класс (*Person*), создали объект этого класса и присвоили ему значение *person_instance* переменной и присвоили ему идентификатор (мы назвали его *Christine*), который вы запрашивали. Если эти концепции кажутся вам простыми, молодец — понимаешь основные принципы объектного ориентирования! Если нет, перечитайте предыдущий раздел , а также прочтите этот раздел, поскольку я собираюсь углубиться в него.

Вы начали с класса *Person*, но теперь вам нужно что-то более сложное, поэтому давайте создадим несколько “домашних животных”, которые будут жить в Ruby. Вы создадите несколько кошек, собак и змей. Первым шагом будет определение классов. Вы могли бы сделать что-то вроде этого:

```
class Cat
attr_accessor :name, :age, :gender, :color
end
class Dog
```

```
attr_accessor :name, :age, :gender, :color
end
class Snake
attr_accessor :name, :age, :gender, :color
end
```

Это похоже на создание класса *Person*, но с умножением на трех разных животных. Вы можете продолжить, создав животных с помощью кода, например, *lassie = Dog.new* или *sammy = Snake.new* и задав атрибуты для питомцев с помощью кода, например, *lassie.age = 12* или *sammy.color = "Green"*. Введите предыдущий код и попробуйте, если хотите.

Однако при создании классов таким образом была бы **упущена** одна из наиболее интересных особенностей объектно-ориентированного программирования - **наследование**.

Наследование позволяет различным классам связывать друг с другом и группировать понятия по их сходству. В данном случае кошки, собаки и змеи - это домашние животные. Наследование позволяет вам создать “родительский” класс домашних животных, а затем позволить вашим классам кошек, собак и змей наследовать (из родительского класса) функции, которыми обладают все домашние животные.

Почти все в реальной жизни имеет структуру, схожую с вашими классами. Кошки могут быть домашними животными, которые, в свою очередь, являются животными; которые, в свою очередь, являются живыми существами; которые, в свою очередь, являются объектами, существующими во Вселенной. Иерархия классов существует повсюду, и объектно-ориентированные языки позволяют определять эти взаимосвязи в коде.

■ Примечание. В главе 6 приведена полезная диаграмма, демонстрирующая концепцию наследования между различными формами жизни, такими как млекопитающие, растения и т. д.

Логически структурируйте своих питомцев

Теперь, когда у нас есть несколько идей по улучшению нашего кода, давайте перепишем его с нуля. Чтобы полностью очистить и перезагрузить то, над чем вы работаете, вы можете перезапустить irb. irb не запоминает информацию между разными периодами использования. Итак, перезапустите irb (чтобы выйти из irb, введите exit и нажмите Enter) и перепишите определения классов следующим образом:

```
class Pet
  attr_accessor :name, :age, :gender, :color
end

class Cat  < Pet
end

class Dog < Pet
end

class Snake < Pet
end
```

■ Примечание

В списках кода в этой главе любой код, находящийся внутри классов, имеет отступ, как в случае с *attr_accessor* в предыдущем(родительском) классе *Pet*. Это всего лишь вопрос стиля, и это облегчает чтение кода.
Когда вы вводите его в irb, нет необходимости повторять эффект, так как это сделает некоторые отступы за вас. Вы можете просто ввести то, что видите. Как только вы начнете использовать текстовый редактор для написания более длинных программ, вам захочется сделать отступы в коде, чтобы его было легче читать, но пока это не важно.

Сначала вы создаете класс *Pet* и определяете атрибуты имени, возраста, пола и цвета, доступные для питомца объекты. Затем вы определяете классы *Cat, Dog u Snake*, которые наследуются от класса *Pet* (оператор < в данном случае указывает, от какого класса происходит наследование). Это означает, что все объекты *cat, dog u snake* будут иметь атрибуты *name, age, gender u color*, но поскольку функциональность этих атрибутов унаследована от класса *Pet*, их функциональность не обязательно создавать специально для каждого класса. Это упрощает обслуживание и обновление кода, если вы хотите сохранить больше информации о домашних животных или добавить другой тип животных.

Как быть с признаками, которые применимы не ко всем животным? Что делать, если вы хотите сохранить длину змей, но не хотите сохранять длину собак или кошек? К счастью, наследование дает вам множество преимуществ, но не имеет недостатков. Вы все равно можете добавить код, относящийся к конкретному классу, куда захотите. Повторно введите класс *Snake* следующим образом:

```
class Snake < Pet
  attr_accessor :length
end
```

У класса *Snake* теперь есть атрибут длины. Однако он добавлен к атрибутам, унаследованным *Snake* от *Pet*, поэтому у *Snake* есть атрибуты имени, возраста, пола, цвета и длины, в то время как у кошек и собак есть только первые четыре атрибута. Вы можете протестировать это следующим образом (некоторые выходные строки были удалены для наглядности):

```
irb(main):001:0> snake = Snake.new
irb(main):002:0> snake.name = "Sammy"
irb(main):003:0> snake.length = 500
irb(main):004:0> lassie = Dog.new
irb(main):005:0> lassie.name = "Lassie"
irb(main):006:0> lassie.age = 20
irb(main):007:0> lassie.length = 10
```

result: *NoMethodError: undefined method 'length=' for #<Dog:0x32fddc @age=20, @name="Lassie">*

Здесь вы создали собаку и змею. Вы дали змее длину 500, прежде чем попытаться дать собаке длину 10 (единицы измерения не важны). Попытка присвоить собаке длину **error of undefined method 'length=',** поскольку вы присвоили атрибут длины только классу *Snake*.

Попробуйте поиграть с другими атрибутами и создать других домашних животных. Попробуйте использовать атрибуты, которых не существует, и посмотрите, что это за сообщения об ошибках.

■ Примечание. Возможно, вам интересно, почему мы используем здесь такие искусственные примеры, как кошки, собаки и змеи. Они были выбраны для того, чтобы представить простую для понимания и легко визуализируемую в уме модель работы классов. В своих будущих приложениях вы будете работать с такими вещами, как различные типы пользователей, события, продукты, фотографии и так далее, и они будут работать примерно одинаково. Не стесняйтесь создавать свои собственные классы, используя концепции , соответствующие вашим запланированным программам, и следовать им, заменяя названия классов, где это уместно.

Контролируйте своих питомцев

До сих пор вы создавали классы и объекты с различными изменяемыми атрибутами. Атрибуты - это данные, относящиеся к отдельным объектам. Змея может иметь длину, собака - имя, а кошка — определенный цвет. Как насчет инструкций, о которых я говорил ранее? Как вы даете своим объектам инструкции для выполнения? Вы определяете методы для каждого класса.

Методы важны в Ruby. Они позволяют вам указывать объектам на выполнение действий. Например, вы можете захотеть добавить в свой класс Dog метод bark, который при вызове объекта Dog выводит Woof! на экран. Вы могли бы написать это так:

```
class Dog < Pet
  def bark
    puts "Woof!"
  end
end
```

После ввода этого кода все собаки, которых вы создадите, теперь смогут лаять. Давайте попробуем:

```
irb(main):0> a_dog = Dog.new
irb(main):0> a_dog.bark
result: Woof!
```

Эврика! Вы заметите, что способ, которым вы заставляете собаку лаять, заключается в простом обращении к собаке (в данном случае - *a_dog*) и добавлении точки (.), за которой следует название метода лая, с помощью которого ваша собака “лает”. Давайте разберем, что именно произошло.

Сначала вы добавили метод bark в свой класс Dog. Для этого вы определили метод. Чтобы определить метод, вы используете слово def, за которым следует название метода, который вы хотите определить. Это означает строка *def bark*. Это означает, что “я определяю метод bark в этом классе, пока не скажу end”. Затем в следующей строке на экране просто отображается слово “Гав!”, а последняя строка метода завершает определение этого метода. Последний *end* завершает определение класса (вот почему полезно использовать отступы, чтобы вы могли видеть, какой *end* совпадает с каким определением). Затем класс Dog содержит новый метод под названием *bark*, который вы использовали ранее.

Подумайте о том, как бы вы создали методы для других классов *Pet* или для самого класса *Pet*. Есть ли здесь методы, общие для всех домашних животных? Если да, то они относятся к классу *Pet*. Существуют ли методы, специфичные для кошек?
Они относятся к классу Cat.

Все является объектом

В этой главе мы рассмотрели, как Ruby может понимать концепции в виде классов и объектов. Мы создали виртуальных кошек и собак, дали им имена и запустили их методы (например, метод *bark*). Эти базовые понятия составляют основу объектно-ориентированного программирования, и вы будете постоянно использовать их на протяжении всей книги. Собаки и кошки - всего лишь пример гибкой объектного ориентирования предложения, но концепции, которые мы использовали до сих пор, могут быть применимы к большинству концепций, независимо от того, даем ли мы “билету” команду изменить его цену или “пользователю” команду сменить свой пароль. Начните думать о программах, которые вы хотите разработать, с точки зрения их общих концепций и о том, как вы можете превратить их в классы, которыми можно манипулировать с помощью Ruby.

Даже среди объектно-ориентированных языков программирования Ruby достаточно уникален тем, что почти все в этом языке является объектом, даже концепции, относящиеся к самому языку. Рассмотрим следующую строку кода:

puts 1 + 10

Если вы введете это в *irb* и нажмете *Enter*, то в ответ увидите цифру 11. Вы попросили Ruby вывести на экран результат, равный 1 + 10. Это кажется достаточно простым, но, хотите верьте, хотите нет, в этой простой строке используются два объекта. 1 - это объект, как и 10. Это объекты класса *Fixnum*, и в этом встроенном классе уже определены методы для выполнения операций с числами, таких как сложение и вычитание.

Мы рассмотрели, как понятия могут быть связаны с разными классами. Наши питомцы служат хорошим примером. Однако даже определение понятий, которые программисты используют при написании компьютерных программ, как классов и объектов имеет смысл. Когда вы записываете простую сумму, такую как 2 + 2, вы ожидаете, что компьютер сложит два числа вместе, чтобы получить 4. В своем объектно-ориентированном виде Ruby рассматривает два числа (2 и 22) как числовые объекты. Таким образом, 2 + 2 - это просто сокращение для запроса первого числового объекта добавить к себе второй числовой объект. На самом деле, знак + - это метод сложения! (Это правда, 2.+(2) будет работать просто отлично!)

Вы можете доказать, что все в Ruby является объектом, спросив объект, к какому классу он принадлежат. В приведенном ранее примере с домашним животным вы могли бы заставить *a_dog* сообщать вам, к какому классу он принадлежит, с помощью следующего кода:

puts a_dog.class

result: Dog

class - это не метод, который вы создали сами, например, метод *bark*, а метод, который Ruby предоставляет по умолчанию для всех объектов. Это означает, что вы можете задать любому объекту, к какому классу он принадлежит, используя метод его класса. Таким образом, a_dog.class равно Dog.

Что будет, если вы спросите число, к какому классу оно относится? Попробуйте:

puts 2.class

result: Fixnum

Число 2 является объектом класса *Fixnum*. Это означает, что все, что нужно сделать Ruby, - это реализовать логику и код для сложения чисел в классе *Fixnum*, подобно тому, как вы создали метод *bark* для своего класса Dog, и тогда Ruby будет знать, как складывать любые

два числа вместе! Но еще лучше то, что вы можете добавлять свои собственные методы в класс *Fixnum* и обрабатывать числа любым удобным для вас способом.

Методы ядра(Kernel Methods)

Kernel — это особый класс (на самом деле, модуль, но не беспокойтесь об этом до главы 6!), методы которого доступны в каждом классе и области видимости в Ruby (если это звучит сложно, рассмотрите методы ядра как те, которые доступны в любой ситуации в обязательном порядке). Вы уже использовали ключевой метод, предоставляемый Ядром.

Рассмотрим метод *puts*. Вы использовали метод *puts* для вывода данных на экран, например, таким образом:

```
puts "Hello, world!"
```

Однако, в отличие от методов в ваших собственных классах, *puts* не имеет префикса перед именем класса или объекта, для которого выполняется выполнение метода. Казалось бы, логично, что полная команда должна быть чем-то вроде *Screen.puts* или *Display.puts*, поскольку *puts* помещает текст на экран. Однако на самом деле *puts* — это метод, доступный из модуля ядра, - особый тип класса, который содержит стандартные, часто используемые методы, упрощающие чтение и запись вашего кода.

■ Обратите внимание, что модуль ядра в Ruby не имеет никакого отношения к ядрам операционных систем или ядру Linux. Как и в случае с ядром и его операционной системой, модуль ядра является частью “ядра” Ruby, но за пределами этого нет никакой связи. Слово “ядро” используется исключительно в традиционном смысле.

Когда вы вводите puts "*Hello, world!*", Ruby может определить, что в нем нет никакого класса или объекта, поэтому он просматривает свои стандартные классы и модули в поисках метода с именем *puts*, находит его в модуле ядра и выполняет свою работу. Когда вы видите строки кода, в которых нет очевидного класса или объекта, найдите время, чтобы подумать, куда направляется вызов метода.

Чтобы гарантировать, что вы используете метод Kernel puts, вы можете обратиться к нему явно, хотя это редко делается с помощью *puts*:

```
Kernel.puts "Hello, world!"
```

Передача данных в методы

Попросить собаку залаять или указать объекту его класс в Ruby просто. Вы просто ссылаетесь на класс или объект иуказываете после него точку (.) и название метода, например, *a_dog.bark*, *2.class* или *Dog.new*. Однако бывают ситуации, когда вы не хотите выполнять простую команду, но хотите связать с ней некоторые данные.

Давайте создадим очень простой класс, представляющий dog:

```
class Dog
  def bark
    puts “Woof!”
  end
end
```

Теперь мы можем просто заставить собаку лаять, вызвав соответствующий метод:

```
my_dog = Dog.new
```

my_dog.bark

result: *“Woof!”*

Это просто, но что делать, если у нас есть действие, для которого был бы полезен некоторый пользовательский ввод? Мы можемнаписать методы для приема данных при их вызове. Например:

```
class Dog
  def bark(i)
    i.times do
      puts “Woof!”
    end
  end
end
```

На этот раз мы можем заставить собаку лаять определенное количество раз, передав значение методу bark:

```
my_dog = Dog.new
my_dog.bark(3)
```

```
result: Woof!
        Woof!
        Woof!
```

Когда мы указываем аргумент 3 в *my_dog.bark(3)*, он передается методу *bark* и помещается в в определенный параметр i. Затем мы можем использовать i в качестве исходного значения для выполнения команды *puts* три раза (или, точнее, i раз), используя блок *times*.

На этом раннем этапе необходимо учитывать еще несколько моментов. Во-первых, вы можете указать множество различных параметров, которые могут быть приняты методом. Например:

```
class Dog
  def say(a, b, c)
    puts a
    puts b
    puts c
  end
end
```

Теперь мы можем передать три аргумента:

```
my_dog = Dog.new
my_dog.say(“Dogs”, “can’t”, “talk!”)
```

```
result: Dogs
        can't
        talk!
```

Вы также должны знать, что круглые скобки вокруг аргументов в конце вызова метода необязательны, если есть только один аргумент и вызов метода не связан ни с какими другими. Например, ранее вы уже видели такой код:

```
puts "Hello"
```

Но вы могли бы с таким же успехом написать:

```
puts("Hello")
```

В этой книге вы еще встретите множество примеров вызова методов и передачи им аргументов. Обратите внимание на различные способы, которыми это происходит, с аргументами и без них, со скобками и без них.

Используя методы класса String

Вы уже играли с собаками и числами, но со строками текста (strings) тоже может быть интересно поиграть:

```
puts "This is a test".length
result: 14
```

Вы задали строку "This is a test", которая является объектом класса String (подтвердите это с помощью "This is a test".class), чтобы вывести его длину на экран, используя метод length. Метод *length* доступен для всех строк, поэтому вы можете заменить "This is a test" любым текстом, который вам нужен, и вы получите правильный ответ.

Запрос длины строки - не единственное, что вы можете сделать. Рассмотрим это:

```
puts "This is a test".upcase
result: THIS IS A TEST
```

Класс *String* имеет множество методов, о которых я расскажу в следующей главе, но пока попробуйте использовать некоторые из них: *capitalize, downcase, chop, next, reverse, sum* и *swapcase.*

В таблице 2.1 показаны некоторые методы, доступные для *strings.*

Таблица 2-1. Результаты использования различных методов в строке “Test” или “test”

Вывод	**выражения**
"Test" + "Test"	TestTest
"test".capitalize	Test
"Test".downcase	test
"Test".chop	Tes
"Test".next	Tesu
"Test".reverse	tseT
"Test".sum	416
"Test".swapcase	tEST
"Test".upcase	TEST
"Test".upcase.reverse	TSET
"Test".upcase.reverse.next	TSEU

Некоторые из примеров в таблице 2-1 очевидны, например, изменение регистра в тексте или его реверсирование, но последние два примера представляют особый интерес. Вместо того, чтобы обрабатывать текст одним методом, вы обрабатываете два или три метода подряд. Причина, по которой вы можете это сделать, заключается в том, что методы будут возвращать исходный объект после того, как он был скорректирован методом, так что у вас есть новый строковый объект, на основе которого можно обрабатывать другой метод.
"Test".upcase приводит к возвращению строки TEST, после чего вызывается обратный метод, приводящий к TSET, после чего вызывается следующий метод, который “увеличивает” последний символ, приводя к TSEU.

В следующей главе мы рассмотрим строки более подробно, но концепция объединения методов в цепочки для получения быстрых результатов является важной в Ruby. Вы можете прочитать предыдущие примеры вслух, и они имеют смысл. Не многие другие языки программирования могут дать вам такой уровень мгновенного знакомства!

Использование Ruby в не объектно-ориентированном стиле

До сих пор в этой главе мы рассматривали несколько достаточно сложных концепций. В некоторых языках программирования объектное ориентирование является чем-то второстепенным, и в книгах для начинающих по этим языкам не рассматривается объектное ориентирование до тех пор, пока читатели не разберутся с основами языка (особенно с Perl и PHP и другие популярные языки веб-разработки). Однако для Ruby это не работает, поскольку Ruby — это чисто объектно-ориентированный язык, и вы можете получить значительные преимущества перед пользователями других языков, сразу поняв эти концепции.

Однако корни Ruby уходят в другие языки. На Ruby оказали сильное влияние такие языки, как Perl и C, которые обычно считаются процедурными, не объектно–ориентированными языками (хотя Perl обладает некоторыми объектно-ориентированными функциями). Таким образом, несмотря на то, что почти все в Ruby является объектным, вы можете использовать Ruby аналогично объектно–ориентированному языку, если хотите, даже если он далек от идеала. По сути, вы бы “игнорировали” объектно-ориентированные функции Ruby, даже если бы они все еще работали под капотом.

Обычная демонстрационная программа для таких языков, как Perl или C, включает в себя создание подпрограммы (по сути, своего рода метода, который не имеет связанного объекта или класса) и ее вызов, подобно тому, как вы вызывали метод *bark* для своих объектов *Dog*. Вот похожая программа, написанная на Ruby:

```
def dog_barking
  puts "Woof!"
end

dog_barking
```

Это сильно отличается от ваших предыдущих экспериментов.
Создается впечатление, что вы не определяете метод внутри класса, а определяете его сам по себе, совершенно независимо. Метод является общим и, похоже, не привязан к какому-либо конкретному классу или объекту. В таких языках, как Perl или C, этот метод назывался бы процедурой, функцией или подфункцией, поскольку метод - это слово, обычно используемое для обозначения действия , которое может выполняться над объектом. В Ruby этот метод все еще определяется для класса (*Object class*), но мы можем игнорировать это в данном контексте.

После определения метода — он по-прежнему называется методом, хотя в других языках он рассматривается как подпрограмма или функция - он становится доступным для

немедленного использования без указания имени класса или объекта, подобно тому, как *puts* доступен без обращения непосредственно к модулю ядра. Вы вызываете метод, просто используя его собственное имя, как в последней строке предыдущего примера. Ввод предыдущего кода в irb
приводит к вызову метода *dog_barking*, что приводит к следующему результату:

result: Woof!

В Ruby почти все является объектом, и это включает в себя волшебное пространство, где в конечном итоге оказываются бесклассовые методы! На данном этапе не важно, где именно это происходит, но всегда полезно помнить об объектно-ориентированных методах Ruby, даже если вы пытаетесь не использовать объектно-ориентированные методы!

■ Примечание
Если вы хотите поэкспериментировать, вы найдете *dog_barking в Object.dog_barking*.

Резюме

В этой главе вы узнали о нескольких важных концепциях не только для программирования на Ruby, но и для программирования в целом. Если эти концепции уже кажутся вам логичными, вы на верном пути к тому, чтобы стать опытным разработчиком на Ruby. Давайте кратко рассмотрим основные концепции, прежде чем двигаться дальше:

•***Class***: Класс - это определение понятия в объектно-ориентированном языке, таком как Ruby. Мы создали классы под названием *Pet, Dog, Cat, Snake и Person*. Классы могут наследоватьобъекты из других классов, но все же обладают собственными уникальными свойствами.

•***Object***: Объект - это единственный экземпляр класса (или, в зависимости от обстоятельств, экземпляр самого класса). Объект класса *Person* - это отдельный человек. Объектом класса *Dog* является отдельная собака. Думайте об объектах как о реальных объектах. Класс - это классификация, в то время как объект - это сам фактический объект или “вещь”.

•***Object orientation***: Объектное ориентирование - это подход, основанный на использовании классов и объектов для моделирования реальных концепций на языке программирования, таком как Ruby.

•***Variable***: В Ruby переменная является заполнителем для одного объекта, который может быть числом, строкой, списком (других объектов) или экземпляром определенного вами класса, например, в этой главе - домашним животным.

•***Method***: Метод представляет собой набор кода (содержащий несколько команд и инструкций) внутри класса и/или объекта. Например, в наших объектах класса *Dog* был метод *bark*, который выводил на экран “*Woof!*”. Методы также могут быть непосредственно связан с классами, как в случае с *fred = Person.new*, где new - это метод, который создает новый объект на основе класса Person. Методы также могут принимать данные, известные как аргументы или параметры, заключенные в круглые скобки после названия метода, как в случае с *puts("Test").*

•***Arguments/parameters***: это данные, передаваемые методам в круглых скобках (или, как в некоторых случаях, после названия метода без круглых скобок, как в случае с *puts "Test"*). Технически, вы передаете аргументы методам, а методы получают параметры, но для практических целей эти термины взаимозаменяемы.

•***Kernel***: для использования некоторых методов не требуется указывать имя класса или модуля, например , *puts*. Обычно это встроенные, распространенные методы, которые не имеют очевидной связи с какими-либо классами или модулями. Многие из этих методов включены в Ruby. Модуль ядра - модуль, предоставляющий функции, которые работают из любой точки мира. Код на Ruby без явных ссылок (глобальный “набор полезных методов”, если хотите).

• ***Experimentation****:* Одна из самых приятных сторон программирования заключается в том, что вы можете воплотить свои мечты в реальность. Количество необходимых вам навыков зависит от вашего мечты, но, как правило, если вы хотите разработать определенный тип приложения или сервиса, вы можете попробовать. Большинство программ создаются по необходимости или в соответствии с мечтой, поэтому важно внимательно следить за тем, что вы, возможно, захотите разработать. Это еще более важно, когда вы впервые получаете практические знания о новом языке, как это происходит во время чтения этой книги. Если вам в голову придет идея, разбейте ее на мельчайшие компоненты, которые вы можете представить в виде классов Ruby, и посмотрите, сможете ли вы собрать воедино строительные блоки, используя Ruby, который вы уже изучили. Ваши навыки программирования могут совершенствуйтесь только с практикой.

В следующих нескольких главах мы рассмотрим темы, кратко затронутые в этой главе, более подробно.