1 Интерпретатор Ruby, rb

Для запуска на выполнение программы на Ruby нужно запустить ее из командной строки Windows:

```
X:\path\ruby>ruby foo.rb
```

Такой запуск приведет к выполнению программы и выводу результата ее выполнения.

Программа irb представляет собой REPL языка Ruby. Ее можно запусть просто дав команду `irb`.

```
X:\path\ruby>irb
irb(main):001:0>
```

Приглашением интерпретатора для ввода команды является `irb(main):001:0>`.

Выход из интерпретатора осуществляется по команде `exit`.

После каждого нажатия Enter интерпретатор анализирует введенную команду и в случае, если она введена полностью, выполняет ее и результат выполнения выводит на терминал.

Образец диалога:

```
irb(main):001:0> 3
=> 3
irb(main):002:0> 2 > 5
=> false
irb(main):003:0> 2 < 5
=> true
irb(main):004:0> x = 2 + 5
=> 7
irb(main):005:0> y = x * x - 3
=> 46
```

В качестве команды REPL может получать любое выражение языка.

При написании программы на Ruby каждая команда (выражение) пишется на отдельной строке. Можно несколько команд языка написать на одной строке, отделив их друг от друга точкой с запятой.

Для включения в тело программы на языке Ruby комментариев следует использовать однострочные комментарии, начинающиеся с символа # и продолжающиеся до конца строки.

Блок многострочных комментариев начинаются строкой, в начале которой стоит `=begin`, и заканчивается строкой с `=end`

2 Класс-ориентированное ООП

В языке Ruby все описывается в терминах объектно-ориентированного программирования (ООП) следующим образом:

1. Все значения (результаты вычисления выражений) являются объектами.

2. Общение с заданным объектом происходит посредством вызовов его методов (отсылкой сообщения объекту). При обработке такого сообщения метод может отправлять сообщения другим объектам.

3. Каждый объект имеет находится в своем приватном состоянии. Только методы объекта имеют прямой доступ к чтению или изменению информации об этом состоянии.

4. Каждый объект является экземпляром класса.

5. Класс объекта определяет поведение этого объекта. Класс содержит определения методов, устанавливающие в чем будет состоять реакция объекта на вызов метода.

3 Определение классов и методов

Поскольку каждый объект принадлежит какому-то классу, то сначала нужно давать определения классам, а затем создавать их экземпляры (объект класса `C` является экземпляром `C`).

Базовый синтаксис для создания класса `Foo` с методами `m1`, `m2`, ..., `mn` может быть:

```
class Foo
  def m1
    ...
  end
  def m2 (x,y)
    ...
  end
  ...
  def mn z
    ...
  end
end
```

Имена классов должны начинаться с заглавной буквы.

Определение класса включают в себя определения методов. Метод может определяться для любого количества аргументов, включая 0. В приведенном выше примере `m1` — метод от 0 аргументов, `m2` принимает два аргумента и `mn` принимает 1 аргумент. Тело каждого метода в примере пропущено.

Результатом вызова метода является результат последнего вычисляемого выражения в нем. Можно использовать конструкцию `return` для возвращения результата (но ставить `return` в качестве последнего выполняемого выражения в методе считается плохим стилем программирования).

Аргументы метода могут иметь значения по умолчанию. В таком случае, при вызове метода может быть указано меньшее количество фактических параметров, чем описано, а вместо тех аргументов, значения которых не указаны при вызове, будут заменены значениями по умолчанию. Например,

```
1  def myMethod (x, y, z=0, w="hi")
2    ...
3  end
```

Если аргумент метода является необязательным (имеет значение по умолчанию), то все аргументы справа от него также должны иметь значение по умолчанию.

4 Вызов методов

Вызов метода

```
e0.m(e1, ..., en)
```

вычисляет значения `e0`, `e1`, ..., `en` до объектов, после чего вызывает метод `m` для результата вычисления `e0`, передавая ему результаты `e1`, ..., `en` в качестве параметров.

Есть особенности синтаксиса, касающиеся написания скобок. В вызове метода круглые скобки, в которые заключаются аргументы, необязательны. В частности вызов метода `m`, имеющего 0 аргументов можно записать

```
e0.m
```

или

```
e0.m()
```

а самый первый вызов упоминаемый на этой странице

```
e0.m e1, ..., en
```

Если методу передается более одного аргумента и аргументы заключаются в скобки, то не должно быть пробела, отделяющего имя вызываемого метода от скобки с аргументами, т. е. вызов

```
e0.m (e1,...,en)
```

приведет к ошибке.

Если выполняемый метод принадлежит некоторому объекту и вызывает метод `m` этого-же объекта, то для вызова можно писать

```
self.m(...)
```

или просто

```
m(...)
```

5 Переменные экземпляра (Instance variables)

Объект имеет класс, который определяет свои методы. Он может также иметь переменные экземпляров, содержащие значения (то есть, объекты). Добавление переменных экземпляра в объект происходит динамически, по мере выполнения программы: если происходит присваивание такой переменной некоторого значения, а переменная с таким именем в данном экземпляре не существует, то она создается.

Все переменные экземпляра должны начинаться с символа `@` (например, `@foo`), чтобы отличать их от переменных, являющихся локальными для метода.

Каждый объект имеет свои собственные переменные экземпляра. Переменные экземпляра являются изменяемыми. Выражение в теле метода может прочитать значение переменной экземпляра просто обратившись к ней

```
@foo
```

или задать новое значение переменной экземпляра с помощью конструкции

```
@foo = newValue
```

При извлечении значения несуществующей переменной экземпляра в качестве результата возвращается объект `nil`.

Переменные экземпляра являются приватными для объекта. Не существует способа непосредственного доступа переменной экземпляра любого другого объекта.

В Ruby можно задать переменные класса. Их имена начинаются с `@@` (например, `@@foo`). Переменные класса являются общими для всех экземпляров класса, но не доступны непосредственно из объектов других классов.

6 Создание объектов

Чтобы создать новый экземпляр класса `Foo`, нужно вызвать

```
Foo.new (...)
```

где `(...)` содержит некоторое количество аргументов. Вызов `Foo.new` создаст новый экземпляр класса `Foo` и затем, прежде чем вернуть созданный объект в качестве результата, вызывается метод `initialize` нового объекта, которому передаются все аргументы, переданные в `Foo.new`. То есть метод `initialize` — особенный и выполняет ту же роль, как и конструкторы в других объектно-ориентированных языках.

Типичное поведение для инициализации — создание и инициализация переменных экземпляра.

7 Выражения и локальные переменные

Как и переменные экземпляра, локальные переменные для метода не должны как-то специально объявляться: переменная создается в ходе операции присваивания, если на момент начала присваивания переменной с заданным именем отсутствовала. Попытка извлечения значения несуществующей переменной приводит к ошибке времени выполнения программы.

Областью видимости переменной является весь метод.

Большинство выражений в Ruby являются на самом деле вызовами методов. Даже

```
e1 + e2
```

— просто синтаксический сахар для

```
e1. + e2
```

то есть, вызвать метод `+` на результат `e1` с `e2`. Другим примером является вызов

```
puts e
```

который выводит значение `e` (после вызова его метода `to_s`, чтобы преобразовать его в строку), после чего делает переход новой строке. Оказывается, `puts` — это метод во всех объектах (он определен в классе `Object`, для которого все классы являются подклассами), поэтому

```
puts e
```

является просто

```
self.puts e
```

Не каждое выражение является вызовом метода. Другой наиболее часто используемой формой выражения является условное выражение. Существуют различные способы записи условных конструкций:

```
if x == y
  f1(x)
else
  f2(y,x)
end
```

```
if x == y then f1(x) else f2(y,x) end
```

```
if x == y
  f1(x)
elsif x < y
  f1(x,y-x)
else
  f3(y,x)
end
```

Конструкции цикла в Ruby используются редко.

8 Константы класса и методов классов

Константы класса в отличие от переменных класса, имеют следующие особенности

- имя константы класса начинается с заглавной буквы вместо `@@`;
- значение константы класса никогда не изменяется;
- значение константы класса видимо за пределами класса.

За пределами экземпляра класса `C` можно получить доступ к константе `Foo` указав имя класса и имя константы `C::Foo`. Часто используемым примером такой константы является `Math::PI`.

Методы класса, по отношению к обычным методам (так называемым методам экземпляра) имеют следующие особенности

- не имеют доступа к каким-либо переменным экземпляра или методам экземпляра
- вызываются за пределами класса с указанием имени класса.

Например, метод `method_name` класса `C` за пределами класса `C` можно вызвать

```
C.method_name(...)
```

Существуют различные способы для определения метода класса. Наиболее распространенным является указание слова `self` в имени метода в его описании.

```
1  def self.method_name args
2    ...
3  end
```

9 Видимость

Как упоминалось выше, переменные экземпляра являются приватными для объекта: только методы этого-же экземпляра могут считывать или записывать значения этих переменных. Даже другие экземпляры того же класса не может получить доступ к переменным экземпляра данного класса.

Один объект может взаимодействовать с другим объектом только путем отправки ему сообщения (вызова метода этого объекта).

Для методов можно описывать различные степени ограничения доступа к ним (область видимости каждого метода). Значение по умолчанию — public (общий), что означает, что любой объект любого класса может вызвать такой метод.

Описание метода как private (частный, приватный), как в случае с переменными экземпляров, позволяет только методам самого объекта вызывать такой метод.

Описание метода как protected (защищенный) означает, что такой метод может вызываться любым объектом, который является экземпляром того же класса или любого подкласса данного класса.

Существуют различные способы, чтобы задать видимость метода. Самый простой — в рамках определения класса, можно указать ключевое слово `public`, `private` или `protected` перед описанием соответствующего блока методов. Предполагается неявное указание `public` перед описанием первого метода в классе.

Одна синтаксическая особенность. Если метод `m` объявлен как `private`, то его можно вызывать только как `m(args)`. Вызов как `x.m(args)` или даже `self.m(args)` не допускаются.

10 Getter/setter-методы

Чтобы сделать содержимое переменной экземпляра доступным или изменяемым, можно определить getter/setter-методы, которые по соглашению могут иметь то же имя, что и переменная экземпляра. Например:

```
1  def foo
2    @foo
3  end
4
5  def foo= x
6    @foo = x
7  end
```

Если эти методы являются public-методами, то любой код может получить доступ переменной экземпляра `@foo` косвенно, путем вызова `foo` или `foo=`.

Синтаксическим сахаром является соглашение, что если имя метода заканчивается на символ «=», то перед этим символом при вызове метода разрешается вставлять пробел. Таким образом, можно писать

```
e.foo = bar
```

вместо

```
e.foo= bar
```

Getter/setter-методы не обязаны на самом деле иметь доступ/устанавливать значение переменной экземпляра. По большому счету — это обычные методы, хотя используя их вызывающий код может считать, что он имеет дело с переменной экземпляра. Например, setter-метод который таковым не является:

```
1  def celsius_temp= x
2    @kelvin_temp = x + 273.15
3  end
```

Getter/setter-методы встречаются настолько часто, что есть более короткая синтаксическая конструкция для их определения. Например, чтобы определить getter-методы для переменных `@x`, `@y` и `@z` и setter-метод для `@x`, в определение класса можно просто включить:

```
1  attr_reader :y :z # определяет getter-методы
2  attr_accessor :x  # определяет getter и setter-методы
```

11 Все является объектом

Все является объектом, включая числа, логические значения и `nil`. Например, `-42.abs` равен `42` потому что класс `Fixnum` определяет метод `abs` для вычисления абсолютного значения и `-42` является экземпляром класса `Fixnum`.

Все объекты имеют метод `nil?`, который в классе объекта `nil` определен для возврата `true`, но в других классах возвращает значение `false`.

Каждое выражение возвращает результат результат. Но когда нет смысла возвращать какой-то конкретный результат, более предпочтительным стилем является возврат объекта `nil`.

Часто бывает удобно для методов вернуть `self`, чтобы последующие вызовы методов этого же объекта могли быть скомпонованы в единый вызов. Например, если метод `foo` возвращает `self`, то можно написать

```
x.foo(14).bar("hi")
```

вместо

```
1 x.foo(14)
2 x.bar("hi")
```

Существует множество методов, определенных для всех объектов, для поддержки отражений (reflection) — средств получения информации об объектах и их определении во время выполнения программы. Например, метод `methods` возвращает массив имен методов, определенных для объекта, а метод `class` возвращает класс объекта. Такие средства, в частности, полезны в REPL при изучении языка или для отладки.

12 Top-level (верхний уровень)

Можно определить методы, переменные и т. д. вне явного класса определения. Такие методы неявно добавляются в класс `Object`, который делает их доступными из имеющихся методов любого объекта.

Выражения верхнего уровня вычисляются в порядке их описания в программе.

13 Массивы

Класс `Array` (массив) — один из наиболее часто используемых классов в программах на Ruby. Существует несколько особенностей синтаксиса по сравнению с массивами в других языках программирования.

Экземпляр класса `Array` — стандартный выбор для какой-либо коллекции объектов.

Формально, массив является отображением номеров (индексов) в объекты.

Выражение `[e1, e2, e3, e4]` создает новый массив с четырьмя объектами в нем: результат `e1` располагается в массиве по индексу 0, результат `e2` — по индексу 1, и так далее.

Есть другие способы для создания массивов. К примеру, `Array.new(x)` создает массив длиной `x`, изначально в котором каждый индекс сопоставляется `nil`. Кроме того, можно передать блок в метод `Array.new` для инициализации элементов массива. Например,

```
Array.new(x) {0}
```

создает массив длиной `x` со всеми элементами, инициализируется значением `0`, а выражение

```
Array.new(5) {|i| -i}
```

создает массив `[0, -1, -2, -3, -4]`.

Формат выражения для получения и установки элементов массива похож на формат подобных выражений во многих других языках программирования: выражение `a[i]` возвращает элемент массива, располагающийся по индексу `i`, `a[i] = e` меняет значение элемента массива по этому индексу.

Вот несколько возможностей использования массивов Ruby:

- Как обычно в языках с динамической типизацией, массив может содержать объекты, являющиеся экземплярами разных классов, например, `[14, "hi", false, 34]`.

- Отрицательные индексы интерпретируются как номера элементов, отсчитанных с конца массива. Так что `a[-1]` извлекает последний элемент массива `a`, `a[-2]` получает предпоследний элемент, и т. д.

- Если при обращении к элементу массива `a[i]` массив a содержит меньше элементов, чем `i + 1`, то результатом будет `nil`. Запись значения по такому индексу приводит к динамическому расширению массива до массива длины `i`.

Массивы Ruby часто используются там, где в других языках используются кортежи, стеки или очереди.

Для работы с массивом как со стеком определены методы `push` и `pop`. Для работы с массивом как с очередью можно использовать метод `push` для добавления элемента в конец очереди и метод `shift` для извлечения элемента из головы очереди. Метод `push` приводит увеличение размера массива на 1 элемент. Методы `pop` и `shift` уменьшают размер массива на 1 элемент.

14 Передача блоков

Хотя в Ruby присутствуют конструкции циклов `for` и `while`, большинство программ на Ruby их не использует. Вместо этого во многих классах имеются методы, принимающие блоки. Блоки — почти замыкания.

Например, для целых чисел определен метод `times`, который принимает блок и выполняет его количество раз, представленное числом. Например,

```
x.times { puts "hi" }
```

печатает «`hi`» 3 раза если `x` равен трем в текущем окружении.

Блоки являются замыканиями в том смысле, что в них могут использоваться имена переменных того окружения, в котором блок определяется. Например, после того, как выполнится следующая программа, `y` должен равняться `10`:

```
1  y = 7
2  [4, 6, 8].each { y += 1 }
```

Здесь `[4, 6, 8]` — это массив с тремя элементами. Метод `each` для объектов класса `Array`, принимает блок и выполняет его один раз для каждого элемента массива.

Как правило, при обработке массива бывает необходимо передать каждый элемент массива обрабатывающему блоку. Это можно реализовать. Например, для того чтобы просуммировать все элементы массива и распечатать сумму накопленную для каждого элемента можно использовать следующий код:

```
sum = 0
[4, 6, 8].each { |x|
  sum += x
  puts sum
}
```

Блоки, не являются объектами. Нельзя передавать их как «регулярные» аргументы метода. В любой метод может быть передано до одного блока, отдельно от других аргументов. Как видно из приведенных выше примеров, блок просто располагается справа от вызова метода после перечисления всех аргументов. Например, метод `inject` (аналог `foldl` в других языках) должен получать 1 «регулярный» аргумент (начальное значение аккумулятора) и блок:

```
sum = [4, 6, 8].inject(0) { |acc, elt|  acc + elt }
```

Помимо использования фигурных скобок в определении блока можно использовать `do` вместо `{` и `end` вместо `}`. Обычно такое написание считается лучшим стилем для блоков, занимающих больше одной строки. Так, первый фрагмент на этой странице можно записать

```
sum = 0
[4, 6, 8].each do |x|
  sum += x
  puts sum
end
```

15 Использование блоков

Блок можно передать в любой метод. Тело метода вызывает блок с помощью ключевого слова `yield`. Например следующий фрагмент выводит «`hi`» 3 раза:

```
1  def foo x
2    if x
3      yield
4    else
5      yield
6      yield
7    end
8  end
9  foo true { puts "hi" }
10 foo false { puts "hi" }
```

Для передачи аргументов в блок, нужно передать аргументы конструкции `yield`, например,

```
yield 7
```

или

```
yield(8,"str")
```

При использовании такого подхода подразумевается тот факт, что методу обязательно должен быть передан блок. В случае если метод вызван, но блок не передан возникнет сообщение об ошибке. Такой ситуации можно избежать если использовать метод `block_given?`, чтобы узнать был ли передан блок.

16 Класс `Proc`

Блоки не совсем являются замыканиями, потому что они не являются объектами. Мы не можем хранить их в переменной, передавать их методу в качестве регулярного аргумента, положить их в массив и т. д. Поэтому мы говорим, что блоки, не являются «значениями первого класса».

Однако, Ruby имеет «настоящие» замыкания: экземпляры класса `Proc`. Метод `call` класса `Proc` является применением замыкания к конкретным параметрам, Например, `x.call` (вызов без аргументов) или `x.call(3,4)`.

Чтобы сделать экземпляр класса `Proc` из блока, можно написать

```
lambda {...}
```

где `{...}` — любой блок. Слово `lambda` не является ключевым словом. Это только лишь метод в классе `Object`, который создает экземпляр `Proc` из блока, переданного ему.

Например, следующий код создает массив «функций» одного аргумента.

```
c = a.map {|x| lambda {|y| x >= y}}
```

Здесь в результате получается массив функций, выдающих `true`, если соответствующий элемент массива `a` больше параметра. Теперь можно отправить сообщение `call` элементам массива `c`:

```
1  c[2].call 17
2  j = c.count { |x| x.call(5) }
```

17 Хеши

Классы `Hash` и `Range` — два стандартных класса. Эти классы похожи на массивы и поддерживают многие из методов-итераторов, применимых для массивов.

Хеши отличаются от массивов тем, что они сопоставляют не числовые индексы объектам, а произвольные объекты к объектам. Если сопоставляются `a` и `b`, мы называем `a` ключом и `b` значением. Таким образом хеш — коллекция, которая сопоставляет набору ключей (в котором все ключи различны) набор значений.

Можно создать хеш следующим образом:

```
{"SML" => 7, "LISP" => 12, "Ruby" => 42}
```

Это выражение создает хеш со строковыми ключами.

Можно использовать Ruby-имена для хеш-ключей, например

```
{:sml => 7, :lisp => 12, :ruby => 42}
```

Можно получить значение из хеша и записать значение в хеш, используя те же синтаксические конструкции, что и у массивов, где в качестве «индекса» может выступать любой объект, например

```
1  h1["a"] = "Found A"
2  h1[false] = "Found false"
3  h1["a"]
4  h1[false]
5  h1[42]
```

Существует множество методов, определенных для хешей. Наиболее полезные из них: `keys` (возвращает массив всех ключей), `values` (возвращает массив значений) и `delete` (для заданного ключа, удаляется ключ и соответствующее значение из хеша). Хеши поддерживают многие из итераторов, применяемых к массивам, например, `each` и `inject`.

18 Диапазоны

Диапазон представляет собой непрерывную последовательность. Задается диапазон указанием начального и конечного значения, разделенными двумя точками. Например,

```
1..100     # представляет целые числа 1, 2, 3,..., 100.
'a'..'z'   # представляет символы 'a', 'b', 'c',..., 'z'.
```

Вызов метода `each` для диапазона обычно заменяет цикл `for`.

```
(1..100).each { |i| e }
```

19 Подклассы и наследование

Создание подклассов является одной из главных концепций ООП. Если класс `C` является подклассом класса `D`, то каждый экземпляр `C` также является экземпляром `D`. Определение `C` наследует методы `D`, то есть методы `D` являются частью определения `C`. Кроме того `C` можно расширять, определяя новые методы, которые есть в `C` и отсутствуют в `D`. Кроме того в классе `C` могут быть переопределены методы, унаследованные от класса `D`.

Каждый класс в Ruby за исключением `Object` имеет один суперкласс. Классы образуют дерево (иерархию классов), где каждый узел представляет собой класс и родительский узел его суперкласс. Класс `Object` находится в корне дерева. По определению подклассов класс имеет все методы всех своих предков по дереве (то есть, все узлы между ним и корнем, включительно).

20 Некоторые особенности Ruby

В Ruby объявление того, что для класс `C` является подклассом класса `D`, происходит в момент описания класса `C` указанием `< D` в первой строке конструкции `class`.

```
1 class C < D
2   ...
3 end
```

Если `< D` не указано, то по умолчанию предполагается `< Object`.

Исследовать иерархию классов поможет метод `class`.

для каждого объекта существует метод `is_a?` и `instance_of?`. Метод `is_a?` принимает класс (например, `x.is_a? Integer`) и возвращает значение `true`, если объект является экземпляром этого класса или его подкласса. Метод `instance_of?` похож на `x.is_a?`, но возвращает `true`, только если класс, для которого вызывается метод непосредственно является экземпляром указанного класса.

Ниже приведены определения для простых классов, где первый класс описывает двумерные точки, а второй класс является его подклассом, в котором каждой точке добавляется цвет.

```
class Point
  attr_accessor :x, :y
  def initialize(x, y)
    @x = x
    @y = y
  end
  def distFromOrigin
    Math.sqrt(@x * @x + @y * @y)
  end
  def distFromOrigin2
    Math.sqrt(x * x + y * y)
  end
end

class ColorPoint < Point
  attr_accessor :color
  def initialize(x, y, c="clear")
    super(x, y)
    @color = c
  end
end
```

Метод initialize в `ColorPoint` использует ключевое слово `super`, которое позволяет переопределяющему методу вызвать метод с тем же именем в суперклассе. Это не является обязательным при переопрелении методов в Ruby, но часто желательно.

21 Примеси

Mixin (примесь) — коллекция элементов, реализующих какое-либо выделенное поведение. Эта коллекция может быть добавлена (примешана) к определению каких-либо классов.

Чтобы определить mixin, используется конструкция, похожая на конструкцию определения класса, но в которой вместо слова `class` используется ключевое слово `module`. Например, mixin для добавления методов обработки цвета к классу:

```
1  module Color
2    attr_accessor :color
3    def darken
4      self.color = "dark " + self.color
5    end
6  end
```

Этот mixin определяет три метода: `color`, `color=` и `darken`. Можно включить эти методы в определение класса с помощью ключевого слова `include` и имени примеси. Например,

```
1  class ColorPt < Point
2    include Color
3  end
```

Такое определение задает подкласс класса `Point`, который, наряду с прочим, имеет три метода, определенных в примеси `Color`.