# Ruby-расширения ООП (module, include, extend). Enumerable. Class variables, объект self.

Модули в Руби похожи на классы в том, что они содержат набор методов, константы, другие модули и определения классов. Модули задаются как классы, только слово module используется вместо class. В отличие от классов создать объекты на основе модуля нельзя, модуль не может иметь подклассы. Вместо этого вы добавляете недостающую функциональность класса или отдельного объекта с помощью модуля. Модули – одиночки, нет иерархии и наследования. (Вообще класс Module имеет суперкласс — Object, однако любой созданный модуль суперкласса не имеет).   
  
Есть два предназначения модулей. Во-первых, они служат централизованного хранения констант и методов, во-вторых, модули позволяют делить функциональность между классами, при включении (include) модуля в класс, его методы добавляются в класс. Такой способ называется примесью (mixin):

module MyModule

GOODMOOD = "happy"

BADMOOD = "grumpy"

def greet

return "I'm #{GOODMOOD}. How are you?"

end

def MyModule.greet

return "I'm #{BADMOOD}. How are you?"

end

end

class MyClass

include MyModule

def sayHi

puts( greet )

end

end

ob = MyClass.new

ob.sayHi

puts(ob.greet)

Руби (в отличие от С++) не разрешает множественное наследование, заменяют его mixin’ы.

Модули являются отличным решением для группирования методов, классов и констант вместе.  
Использую модули можно извлечь две значительные выгоды:

-Модули обеспечивают пространства имен и, следовательно, предотвращают возникающие с именованием методов, классов и констант ошибки.

-Модули реализуют примеси (**mixins**) — элегантное решение заменяющее множественное наследование.

Модули определяют пространства имен, песочницу в которой ваши методы и константы могут работать без возникновения проблем с другими методами и константами, которые имеют такие же имена.

Так выглядит модуль:

**module Identifier**  
**statement1**  
**statement2**  
**………..**  
**end**

Модульные константы именуются тем же образом, что константы в обычных классах, с первой заглавной буквой в названии. Определение методов выглядит аналогично: модульный метод объявляется так же как и метод класса.

Так же как с методами класса, мы вызываем методы модуля, но перед именами методов мы пишем имя модуля и разделитель **::**, таким образом мы ссылаемся на методы класса используя конструкцию **модуль::метод**, где двойное двоеточие разделяет имя модуля и имя метода.

Пример:

#!/usr/bin/ruby

# Модуль определен в файле trig.rb

module Trig

  PI = 3.141592654

  def Trig.sin(x)

    # ..

  end

  def Trig.cos(x)

    # ..

  end

end

Мы можем определять несколько модулей с одинаковыми именами функций (методов), но с различной функциональностью:

#!/usr/bin/ruby

# Модуль объявлен в файле moral.rb

module Moral

  VERY\_BAD = 0

  BAD = 1

  def Moral.sin(badness)

    # ...

  end

end

Мы можем определять методы модуля так же, как и методы класса, разделяя при определении методы имя метода от имени модуля точкой.

Использование метода require

В Ruby require используется так же, как и include в Си и Си++ и как import в Java. Если программа нуждается в использовании каких-либо определенных модулей, она может просто загрузить файлы с модулями используя require:

**Синтаксис:**

**require filename**  
  
Здесь не обязательно вписывать после имени файла расширение .rb.

**Пример:**

require 'trig.rb'

require 'moral'

y = Trig.sin(Trig::PI/4)

wrongdoing = Moral.sin(Moral::VERY\_BAD)

**ВАЖНО:** Здесь оба файла содержат методы с одинаковыми именами, из-за чего возникает двусмысленность кода, однако модули позволяют избежать этой двусмысленности путем указания того, какому модулю принадлежит тот или иной метод.

Использование include в Ruby

Мы можем включать в класс модули. Для включения модуля в класс используется include:

**Синтаксис:**

**include modulename**

Если модуль объявлен в отдельном файле, то он нуждается в включении этого файла используя require перед включением самого модуля в класс.

Рассмотрим следующий модуль написанный в файле Week.rb.

module Week

  FIRST\_DAY = "Sunday"

  def Week.weeks\_in\_month

    puts "You have four weeks in a month"

  end

  def Week.weeks\_in\_year

    puts "You have 52 weeks in a year"

  end

end

Теперь мы можем включить этот модуль в класс следующим образом:

#!/usr/bin/ruby

require "Week"

class Decade

  include Week

  no\_of\_yrs=10

  def no\_of\_months

    puts Week::FIRST\_DAY

    number=10\*12

    puts number

  end

end

d1=Decade.new

puts Week::FIRST\_DAY

Week.weeks\_in\_month

Week.weeks\_in\_year

d1.no\_of\_months

В результате этого получим следующий результат:

*Sunday*  
*You have four weeks in a month*  
*You have 52 weeks in a year*  
*Sunday*  
*120*

Примеси в Ruby

Я предполагаю, что у вас есть знания ООП, которые необходимы для понимания следующей части статьи.

Если класс может наследоваться от более чем одного родительского класса, то данный класс использует множественное наследование.

Ruby не поддерживает множественного наследования, но модули в Ruby дают другую, прекрасную альтернативу множественному наследованию. Эта альтернатива позволяет использовать примеси (mixins), вместо множественного наследования.

Примеси дают нам отличный контроль над добавлением функциональности классам, однако, их истинная сила проявляется когда код в примесях начинает взаимодействовать с кодом класса, который их использует.

Давайте рассмотрим следующий пример кода для понимания примесей:

module A

  def a1

  end

  def a2

  end

end

module B

  def b1

  end

  def b2

  end

end

class Sample

  include A

  include B

  def s1

  end

end

samp=Sample.new

samp.a1

samp.a2

samp.b1

samp.b2

samp.s1

Модуль А состоит из методов а1 и а2. Модуль В состоит из методов b1 и b2. Класс Sample включает в себя оба модуля A и B. Класс Sample может получать доступ ко всем четырем методам с именами a1, a2, b1 и b2. Поэтому, мы можем видеть что класс Sample наследуется от обоих модулей. Таким образом мы можем сказать, что класс Sample наследуется от нескольких модулей, при том, что для Ruby множественное наследование чуждо. Это сделано с целью избежать проблем возникающих при использовании множественного наследования в других языках программирования.

то бы добавить функциональность (код) модуля в конкретный класс можно воспользоваться двумя способами include или extend.  
Различие  между ним:

* include  делает методы модуля доступными для экземпляра класса (для объекта), в то время как
* extend делает эти методы доступными для класса непосредственно.

Вы вероятно уже знакомы с модулем Enumerable в Ruby, даже если вы видите это название в первый раз. Этот модуль добавляет некоторые методы для массивов, например map, inject, select и т.д. Вы наверняка думаете (Как и я ранее долгое время думал), что эти методы являются методами, подмешиваемыми только к массивам, однако это не так.

Массивы в Ruby используют модуль Enumerable, и вы можете абсолютно так же подмешивать его к любому желаемому классу. Конечно, класс должен представлять собой некую коллекцию, набор элементов, иначе все методы, которые представлены в модуле Enumerable не будут иметь смысла. Давайте рассмотрим пример, в котором у нас имеется класс Team, который управляет командами и членами этих комманд.

class Team

  include Enumerable

  attr\_accessor :members

  def initialize

    @members = []

  end

  def each &block

    @members.each{|member| block.call(member)}

  end

end

Enumerable требует использования в контексте вашего класса метода each который передает элементы в коллекцию @members. Все методы из модуля Enumerable полагаются на это. Для примера давайте воспользуемся методом map:

irb(main):001:0> require ‘team.rb’  
=> true  
irb(main):002:0> team = Team.new  
=> #  
irb(main):003:0> team.members = ['joshua', 'gabriel', 'jacob']  
=> ["joshua", "gabriel", "jacob"]  
irb(main):004:0> team.map{|member| member.capitalize}  
=> ["Joshua", "Gabriel", "Jacob"]

Теперь мы можем вызывать любой метод из модуля Enumerable как метод экземпляра класса Team и этот метод будет знать, что нам требуется работать внутри с массивом @members. Enumerable может быть мощной примесью к вашим собственным классам. Вы можете даже пойти дальше и сделать вызов метода map более красивым:

team.map(&:capitalize)

Вся магия состоит из двух частей: это унарный оператор амперсанд и символ, который был передан оператору.

Когда Ruby видит унарный амперсанд с последующим аргументом – символом он пытается конвертировать его в объект proc (блок кода) и запустить этот блок. Ruby делает это вызывая метод to\_proc. Немного истории: В Ruby on Rails 1.1 был добавлен метод **to\_proc** для класса Symbol, таким образом вы должны передавать символ унарному амперсанду (как например &:capitalize) и он будет конвертировать символ :capitalize в блок кода (по сути это является созданием блока кода в контексте которого вызывается метод capitalize). Начиная с Ruby 1.8.7 этот метод был включен в сам Ruby, по другим данным он был портирован в 1.8.7 уже из ветки 1.9, в любом случае вам нет необходимости использовать Ruby on Rails для того, чтобы иметь возможность воспользоваться методом to\_proc.

### ****Указатель self****

В предыдущем примере вы могли видеть указатель self. self — это указатель на текущий объект, объект, в контексте которого проиходит действие. Когда вы пишите:

class ClassName

def self.method\_name()

end

end

Это в конечном счете интерпретируется как:

class ClassName

def ClassName.method\_name()

end

end

Когда вы используете self в контексте метода экземпляра класса, то self будет ссылаться на текущий экземпляр класса, пример:

class ClassName

def obj\_info

puts "ObjectID:   #{self.object\_id}"

puts "ObjectType: #{self.class}"

end

end

o = ClassName.new

puts o.object\_id #=> 83402010

o.obj\_info

#ObjectID:   83402010

Переменные экземпляра работают так, как от них ожидают в обычных ситуациях:

class A

def set\_a( v ); @a = v; end

def get\_a; return @a; end

end

o1 = A.new

o1.instance\_variables # => []

o1.get\_a # => nil

o1.set\_a 123

o1.instance\_variables # => ["@a"]

o1.get\_a # => 123

o2 = A.new

o2.get\_a # => nil

Переменная появляется в экземпляре после ее установки, дальше все работает как по маслу — она видна только в своем экземпляре.

Значение по-умолчанию

Неожиданности с переменныой экземпляра обычно начинаются когда нужно установить значение ее по-умолчанию:

class B

attr\_accessor :b # вместо set\_b, get\_b из вышестоящего примера

@b = 1

end

o1 = B.new

o1.b # => nil

o1.instance\_variables # => []

o1.b = 2

o1.b # => 2

o1.instance\_variables # => ["@b"]

Оп-па. А где же наше значение по-умолчанию? Почему переменной вообще нет в обьекте с самого начала?  
  
Такие переменные называются *Class-Level Instance Variable* (как это по-русски «переменные экземпляра Class-уровня»?)

Классы это объекты.

На Ruby, что любой код выполняется в контексте текущего обьекта (экземпляра класса), текущий объект всегда доступен из псевдо-переменной *self*. Это базовые истинины. Но попорбуйте ответить на вопрос, в каком конексте (объекте) выполняется строка 4 и строка 2?

1: class User

2: @b = 1

3: def a

4: @a = 1

5: end

6: end

Строка 4 выполнится в экземпляре класса User, как мы и ожидаем, в объекте который вы создадите через User.new  
  
А строка 2 выполняется в экземпляре класса Class, в объекте который описывает класс User. Этот объект создатся сразу после чтения интерпретатором класса User.  
Само слово *User* это не что иное как [консанта, значение которой объект типа *Class*](http://habrahabr.ru/blogs/ruby/111738/). Этот объект описывает наш класс User. Именно к этому объекту относится переменная *@b*.

o1 = B.new

o1.b # => nil

o1.instance\_variables # => []

B.instance\_variables # => ["@b"]

На самом деле она всегда была там. Не в классе B и не в экземпляре класса B, а в экземпляре класса Class хранимом в константе B.

Применение

Что нам дают переменные экземпляра Class-уровня?  
  
Они могут работать как личные переменные класса. В отличии от настоящих переменных класса, которые доступных во всех потомках и экземплярах под одним именем, наша переменная принадлежит только тому классу в контексте которого была объявлена.

class B

@b = 1

def self.set\_b( v ); @b = v; end

def self.get\_b; @b; end

end

B.get\_b # => 1

B.set\_b 2

B.get\_b # => 2

class B1 < B; end

B1.get\_b # => 1

B1.set\_b 3

B1.get\_b # => 3

B.get\_b # => 2

То есть при каждом наследовании от класса, в котором была объявлена такая переменная будет создаваться новая независимая. Сравните с поведением переменной класса:

class C

@@c = 1

def self.set\_c( v ); @@c = v; end

def self.get\_c; @@c; end

end

C.get\_c # => 1

C.set\_b 2

C.get\_b # => 2

class C1 < C; end

C1.get\_c # => 2

C1.set\_c 3

C1.get\_c # => 3

C.get\_c # => 3

class C2 < C; @@c = 4; end

C.get\_c # => 4

C1.get\_c # => 4

C2.get\_c # => 4

Accessors

Такие переменные с помощью акцессоров создаются следующим образом:

class A

class << self

attr\_accessor :a

end

# Вместо этого такая строка: self.class.attr\_accessor :a

# невозможна только потому, что attr\_accessor это private method :(

end

A.instance\_variables # => ["@a"]

A.class\_variables # => []

A.a # => nil

A.a = 123

A.a # => 123

Обратите внимание что рельсовый *cattr\_accessor :a* создаст методы с такими же именами, только обращаться они будут к переменной класса.

class User

cattr\_accessor :a

end

User.instance\_variables # => []

User.class\_variables # => ["@@a"]