Занятие 4.

Разбор домашнего задания 2-3:

- вопросы? проблемы?
- обзор запомнившихся особенностей, связанных с Ruby

Основы ООП в Ruby. Объектная модель, переменные. Модификация доступа.

Ruby — объектно-ориентированный язык программирования, поэтому знание парадигмы объектно-ориентированного программирования (ООП) является обязательным.

Объектно — ориентированное программирование — это программирование основанное на использовании объектов — экземпляров абстрактных типов (классах). Объект в ООП — это программная модель какого-то реально существующего объекта, например вас или меня, при этом будучи объектами реального мира мы будем являться экземплярами некоторого типа (вида).

Все общее в людях и общий набор свойств определяется в классе, а все различное определяется в самих объектах.

Три кита объектно-ориентированного программирования: Инкапсуляция, Наследование, Полиморфизм.

Инкапсуляция - языковая конструкция, позволяющая связать данные с методами, предназначенными для обработки этих данных. Доступ к этим данным доступен только через специально определяемые интерфейсы, таким образом скрывается внутреннее

устройство объекта. Такие методы-интерфейсы называются accessors. Обобщая, можно сказать, что это механизм языка, позволяющий ограничить доступ одних компонентов программы от других.

Наследование заключается в эволюции классов, тобешь один класс может наследоваться от другого. Наследование заключается в передаче всей структуры класса одного класса в другой. Причём при наследовании появляются понятия: супер класс или родительский класс (тот от которого наследовались), и подкласс / сабкласс или дочерний класс (тот, который насдежовался). На практике это означает, что для наследника полностью копируется реализация класса родителя. Такое копирование и есть наследование.
Наследование — очень полезная вещь, так как: предоставляет возможность повторного использования кода, спасает от повторений в коде. Скажем имея класс, мы можем наследоваться от него любым количеством других классов, которые будут иметь абсолютно одиинаковые свойства, за исключением переопределённых свойств. Вместо написания двух новых классов, мы просто наследуеемся ими от базового класса — дополняя их новыми свойствами и методами, либо переписывая старые.

Полиморфизмом называется способность метода обрабатывать данные разных типов.
Одна сущность - несколько реализаций. Полиморфизм тесно связан с наследованием и делает его еще более мощным.
Примеры из жизни: автомобиль, но несколько разных марок; собака, но несколько пород; и т.д.
Представим, что у нас есть два класса (например, кошки и собаки) отнаследованные от общего класса “животное”, в каждом определено по объекту (конкретные кот и пёс) и методу с одинаковыми названиями («говорить», но кот мяукает, а пёс лает).
Получается, что на один и тот же метод объекты реагируют по-разному. Так вот эта возможность иметь несколько классов, содержащих одноименные методы, и называется полиморфизмом. Тобешь наследуясь, в классе — потомке (дочернем классе или подклассе или субклассе или …) вы можете переопределять унаследованные свойства и методы.

Утиная типитизация - границы использования объекта определяются его текущим набором методов и свойств, в противоположность наследованию от определённого

класса. То есть считается, что объект реализует интерфейс, если он содержит все методы этого интерфейса, независимо от связей в иерархии наследования и принадлежности к какому-либо конкретному классу.
Название термина пошло от английского «duck test» («утиный тест»), который в оригинале звучит как:
«If it looks like a duck, swims like a duck and quacks like a duck, then it probably is a duck».
(«Если нечто выглядит как утка, плавает как утка и крякает как утка, то это, вероятно, утка и есть».)

Объектная модель

- всё объект – true ООП
- объект = инстанс класса
- Object, Module, Class – три класса, которые нужно запомнить

- Object – по-умолчанию
- Class.class.class.class...
- объекты вызывают методы

- ответ всегда новый объект
- синтаксический сахар
- кэширование
- передача по ссылке

- что не объект? (блок, ключевые слова, if..end, синтаксические конструкции, определения методов)

Полезные методы:

.is_a?(ClassName) - проверить является ли заданным классом
self - указатель на текущий объект
.methods
.object_id
.clone => новый объект

Объекты классов

- всё объект
- объект есть инстанс класса
- класс – тоже объект => инстанс класса Class

-объект объекту рознь. Различия между инстансом класса, и классом всё же есть (.new / meta-data / superclass)
- классы Ruby открытые – дополняй, не хочу
- код выполняется построково, даже в теле класса
- создание инстанса класса – через .new – Это метод класса, не инстанса. И хоть мы его не определяли – он есть!

Создание объекта

- что происходит при вызове .new ?
- в .new можно передавать параметры
- метод #initialize, согласно договоренности, является тем методом, что автоматически выполняется при создании нового экземпляра класса, такие методы еще называют методами — конструкторами.
- рассказать про reader/writer

Переменные:

- local – только внутри метора, или блока
- @instance – привязаны к объекту\инстансу. У каждого свои. Шарятся между методами. По-умолчанию – nil. (В отличие от локальных)
- @@class – привязаны к классу, шаряться между объектами этого же класса. Должны быть проинициализированы перед использованием
- $global – привязаны к программе, шаряться между классами. По-умолчанию – nil
- CONS_TANT – внутри класса ведут себя как классовые, снаружи – как глобальные, но при этом “константы”. Вне класса – доступ по ::
- self – объект-получатель\текущий
- __FILE__
- __LINE__

… defaults & predefined
(ENV, RUBY_VERSION, STDIN, $LOAD_PATH, $, ……)

Наследование:

- вспоминаем, что только классы обладают методоми .new и .superclass
- метаданные классов передаются объектам инстансам по цепочке суперклассов.
- методы переписываются потомками
- потомки могут вызывать родительские методы: super & super() - показать разницу
super — это очень полезное выражение, которое позволяет в контексте метода вызывать одноименный метод из родительского класса, другими словами, super позволяет реализовать что-то вроде «наследования методов» при котором мы в контексте переопределенного метода, вызываем старый метод и дополняем его.

.class, .instance_of?, .ancestors и is_a? / kind_of? - разница и принцип работы
- сравнение классов между собой
.constants (все констаты определены на классе или модуле. Если не указано – то Object)
- практически все классы и модули – константы класса Object
.const_defined?

- множественного наследования – нет. Зато есть миксины
- перегрузки методов – нет. Не важно сколько параметров он принимает, если имя то же.
- private, protected, классовые методы – наследуются как положено. Но надо быть осторожным и иметь ввиду как резолвятся методы в Ruby. Примерно так:
https://dl.dropboxusercontent.com/u/306877/bigsoft/RubyMethodLookupFlow.png

- с @инстансами всё понятно
- константы – привязаны к классу
- классовые – меняются не интуитивно

Листинг занятия:

```
class Example
end
```

example = Example.new

#диаграмма суперклассов

Class < Module < Object < BasicObject < nil
^
Example ___|

instances
example.superclass # Exception
Kernel.superclass # Exception (потому что Kernel - это модуль)
example.new # Exception
Kernel.new # Exception
why? because they are not classes
p example.class # Example
p Kernel.class # Module

Example = Class.new

#зависимость экземпляра и его класса

example.class ___ Example.new ___
^ |
|_____|

диаграмма классов

example.class
|
Example
|
Kernel --> Module --> Class <-- Object
^
|
Class

```
class Animal
 def speak
        puts 'Yes, we can!'
 end
end

#Наследование
class Cat < Animal
end

cat = Cat.new
cat.speak

class Bird
 def make_eggs
        puts "Eggs are kids!"
 end

 def fly
        puts 'Birds are flying!'
 end
end


class Penguin < Bird
 def fly
        puts "Penguins are not flying :("
 end
end

penguin = Penguin.new
penguin.make_eggs
penguin.fly
```

```
class Animal
 def can_run
        puts "Can run"
 end

 def has_legs(amount)
        puts "Has #{amount} legs"
 end

 def has_eyes(amount)
        puts "Has #{amount} eyes"
 end

 def has_teeth(color = 'white')
        puts "Has #{color} teeth"
 end
end

class Turtle < Animal
 def can_run
        super
        puts 'But toooo slow'
 end

 def has_legs(amount)
        super
 end

 def has_eyes
        super(4) # mutant
 end

 def has_teeth(color)
        super()
 end
end
```

```
turtle = Turtle.new
turtle.can_run
turtle.has_legs(4)
turtle.has_eyes
turtle.has_teeth("yellow") # compare with an .has_legs(4)

p turtle.class
p turtle.instance_of?(Turtle)
p turtle.class.ancestors
p turtle.instance_of?(Animal)
p turtle.is_a?(Animal)
```

#классовая переменная распространяется по классам

```
class A
 @@class_variable = 'A'
 CONST = 'A_CONST'
 def show
        p [@@class_variable, CONST]
 end
end

class B < A
 @@class_variable = "B"
 CONST = "B_CONST"
 def show
        puts "METHOD OF B<A"
        p [@@class_variable, CONST]
 end
end

[10] pry(main)> A.new.show
["B", "A_CONST"]
=> ["B", "A_CONST"]
[13] pry(main)> B.new.show
METHOD OF B<A
```

```
["B", "B_CONST"]
=> ["B", "B_CONST"]
[14] pry(main)> p A::CONST
"A_CONST"
=> "A_CONST"
[16] pry(main)> p B::CONST
"B_CONST"
=> "B_CONST"
```

Интересные примеры:

Метод .allocate не вызывает конструктор, а просто выделяет память под объект:

```
klass = Class.new do
  def initialize(*args)
    @initialized = true
  end

  def initialized?
    @initialized || false
  end
end

klass.allocate.initialized? #=> false
```

Методы коллбеки - методы, вызывающиеся по какому-либо событию. Например при наследовании:

```
class Foo
   def self.inherited(subclass)
      puts "New subclass: #{subclass}"
   end
end
```

```
class Bar < Foo
end
New subclass: Bar

class Baz < Bar
end
New subclass: Baz
```

Классы

- def – методы инстанса
- self.method_name – методы класса (или через Ghost class, показать)
- при переопределении использован будет последний. Почему? - Потому что интерпретатор запомнит только его, а старый будет перезаписан
- классовые методы – не метадата, поэтому они не попадают в инстансы
- перегрузка .to_s - для удобочитаемости класса

```
class Dog
 def initialize(name)
        @name = name
 end
end

class Cat
 def initialize(name)
        @name = name
 end

 def to_s
        "Cat has name: #@name"
 end
end
```

```
dog = Dog.new("Шарик")
puts dog # => #<Dog:0x000000012bea48 @name="Шарик">
kot = Cat.new("Мурка")
=> #<Cat:0x0000000121de90 @name="Мурка">
puts kot
Cat has name: Мурка
=> nil
```

Контроль доступа:

- для инстанс методов, не переменных
- public – дефолтный(кроме initialze), открыт для всех.
- private – внутри-классовые, без получателя.
- protected – только объектами этого же класса, или его под-классов. Доступ к состоянию из других инстансов.

Приватные и protected методы ведут себя похоже:

```
class A
 def main_method
        private_method
        protected_method
 end

 private
 def private_method
        puts "hello from #{self.class} private"
 end

 protected
 def protected_method
```

```
        puts "A protected"
 end
end

class B < A
 def main_method
        private_method
        protected_method
 end
end
```

Напрямую их вызвать нельзя:

```
A.new.protected_method
NoMethodError: protected method `protected_method' called for
<A:0x00000002c08918>

A.new.private_method
NoMethodError: private method `private_method' called for #<A:0x00000002bc5eb0>
```

Наследуются:

```
A.new.main_method
hello from A private
A protected
=> nil

B.new.main_method
hello from B private
A protected
=> nil
```

Но есть различия, protected можно вызвать для инстансов того же класса или подклассов:

```
class Wallet
 def balance
   @balance
 end
 protected :balance

 def balance=(a)
   @balance = a
 end

 def check_balance(other)
         @balance > other.balance
 end
end

class Fire < Wallet
end

fire = Fire.new
fire.balance = 1
fire.balance # напрямую вызвать по-прежнему нельзя
NoMethodError: protected method `balance' called for #<Fire:0x00000002bfab60
@balance=1>


wallet1 = Wallet.new
wallet1.balance=10

wallet2 = Wallet.new
wallet2.balance = 20

p wallet1.check_balance(wallet2) # Работает! Потому что wallet2.is_a?(wallet1.class)
p fire.class # => Fire
```

```
p fire.is_a?(walet2.class) # => true
p wallet2.check_balance(fire) # => true # аналогично
```

Получаем, что protected методы могут вызываться только через методы родственных объектов (произошедших от того же класса).

Ещё пример:

```
class A
 def main_method
        method1
 end

 protected
 def method1
        puts self.inspect
        puts "hello from #{self.class}"
 end
end

class B < A
 def main_method
        puts self.inspect
        method1
 end
end

class C < A
 def main_method
        puts self.inspect
        self.method1
 end
end
```

Всё работает, так как классы связаны (наследуются от A)

A.new.main_method # => hello from A
B.new.main_method # => hello from B
C.new.main_method # => hello from C

А теперь допустим у нас появился новый класс, не связанный ни с A, ни с B, ни с C, но который пытается вызвать protected метод сам:

```
class D
 def main_method
        B.new.method1
 end
end
```

D.new.main_method # => NoMethodError: protected method `**method1'** called for #<B:0x00000002308d18>

- мы не смогли вызвать protected метод, так как на момент вызова находимся в инстансе D.new, что не связанно ни с A, ни с B, ни с C.

Классовые методы тоже наследуются:

```
class A
 def self.test
        puts "A"
 end
end
class B < A
end
B.test
```

Приемники методов

-self – дефолтный получатель сообщений
-self в каждом методе есть объект получатель этого метода

Методы пренадлежат классам и имеют приемник — объект, на котором вызываются. Если метод вызывается без указания приемника, то это означает, что приемником является текущий объект, то есть тот, в контексте которого вызывается метод. Примеры:

```
class A
def self.hello
  puts 'hello'
end
def bye
  puts 'bye'
end
end
A.hello #hello
A.new.bye #bye
```

Приемник self указывает на текущий объект, то есть метод .hello является методом класса, а метод #bye — методом объекта.

Указатель self

В предыдущем примере вы могли видеть указатель self. self — это указатель на текущий объект, объект, в контексте которого происходит 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
```

Модули:

- контейнер для методов и констант и даже классов
- неймспейсер. Что такое? Не влияют на оригиналы
- модуль не класс (в отличие от класса Module) – со всеми вытекающими(нельзя создать инстанс, нельзя наследовать)
- для поведения, не стейта

Извне (файл) можно загрузить в память через:
require / load / require_relative / autoload

Модули и классы:
- include
- extend
- @инстансы и возможные коллизии и как избежать

```
module Perimeter
 class Array
        def initialize
          @size = 400
        end
 end
end
```

Один класс - но разные объекты, ничего не поломали:

```
our_array = Perimeter::Array.new
ruby_array = Array.new

p our_array.class # => Perimeter::Array
p ruby_array.class # Array
```

Работа с несколькими модулями:

```
module Test
 TEST = 'test const'
 @@test = 'test'

 def public_module_method
        puts 'public_module_method TEST'
 end

 def unique_public_module_method
        puts 'unique public_module_method TEST'
 end

 def self.module_method
        puts 'module_method TEST ' + @@test
 end
end

module Programmer
 TEST = 'programmer const'

 def un_public_module_method
        puts 'unique public_module_method Programmer'
 end

 def public_module_method
        puts 'public_module_method Programmer'
 end

 def self.module_method
        puts 'module_method Programmer'
 end
end

p Test::TEST # => "test const"
Test.module_method # => module_method TESTtest
```

```
class Person
 include Test
 include Programmer
end

p Test::TEST # => "test const"
p Programmer::TEST # => "programmer const"
p Person::TEST # => "programmer const"
Person.new.public_module_method # => public_module_method Programmer
Person.new.unique_public_module_method # => unique public_module_method TEST
Person.new.un_public_module_method # => unique public_module_method Programmer
```

Полезные методы:

.included_modules # массив всех модулей, включённых в класс
.ancestors # массив всех классов и модулей (вся цепочка)
- показать взаимодействие модулей и классов
- ghost class

Синглтоны/Ghost class/Anonymous class:

- методы, хранятся в классах
- создание методов для инстанса происходит через Ghost класс (мета класс)
- у сиглтона класса нельзя создавать инстансы – на то он и сиглтон
- благодаря ему доступно наследование и миксины

метод .extend – для любого объекта

примеры синглтона:

```
a = ‘string’
def a.secret
 puts ‘only one object has this method’
end
a.secret # working..
‘test’.secret  # Exception
```

аналогично можно так:

```
class << a
 def secret
   #blablabla
 end
end
```

a.singleton_class # => название синглтон-класса
a.singleton_methods # => массив всех методов, определённых в классе синглтоне

открыть Ghost класс в классе:

```
class Test
 class << self
   # мы в гост-классе
 end
end
```

!Классовые методы на самом деле публичные методы синглтона класса.

Интересные примеры:

Доступ к внешней константе, при пересечении имёт с вложенной:

```
A = 1
module Kata
 A = 5
 module Dojo
   B = 9
   A = 7

   def self.inner_constant
     p A
   end

   def self.outer_constant
     p ::A
```

```
  end
 end
end

Kata::A
=> 5
Kata::Dojo::A
=> 7
Kata::Dojo.inner_constant
=> 7
Kata::Dojo.outer_constant
=> 1
```

Сравнение классов == проверка на вложенность и наследование:

```
p Fixnum < Numeric  #-> true
p Object > Integer  #-> true
p Float < Integer   #-> nil
p IO <= File            #-> false
```

Множественное присваивание:

```
a, b = 1, 2
a, b = b, a
a, b, c = [1, 2, 3]
```

Добавляем возможности итерации для своего класса:

Мы уже знакомы с модулем Enumerable в Ruby и помним, что он добавляет некоторые методы для коллекций, например map, inject, select и т.д.

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 = [])
    @members = 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:

```
team = Team.new(['joshua', 'gabriel', 'jacob'])
=> #<Team:0x00000002541968 @members=["joshua", "gabriel", "jacob"]>
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.

рассказать о module prepend и #ancestors

```
module M
 def test
   p ‘module test’
 end
end

class Test1
 include M
```

```
 def test
   p ‘class test’
 end
end

Test1.new.test # => ‘class test’

class Test2
 prepend M
 def test
   p ‘class test’
 end
end

Test2.new.test # => ‘module test’
```

Домашнее задание #4:

Теория:

- прочесть заметки лекции ещё раз, два, три…
- Почитать ещё раз об ООП, и с чем его едят.
- при возможности изучить следующие ссылки:
  - http://habrahabr.ru/post/48756/
  - http://habrahabr.ru/post/49149/
  - http://rubydev.ru/2012/09/rubydev-ruby-tutorial-class-instance-global-variables-and-constants-in-ruby/
  - http://rubydev.ru/2011/04/rubydev-ruby-tutorial-object-oriented-programming-ruby/
-
  - http://habrahabr.ru/post/111738/
  - http://rubydev.ru/2010/08/ruby_module/
  - http://rubydev.ru/2010/12/ruby-private-protected-public-methods/
  - http://habrahabr.ru/post/143483/
  - http://habrahabr.ru/post/50151/

- http://habrahabr.ru/post/49353/
- http://ruby.about.com/od/rubysbasicfeatures/ss/Load-Vs-Require.htm
- http://www.tutorialspoint.com/ruby/ruby_modules.htm

- получаем удовольствие от изучения документации:
- http://www.ruby-doc.org/core-2.2.0/Class.html
- http://www.ruby-doc.org/core-2.2.0/Module.html
- http://www.ruby-doc.org/stdlib-1.9.3/libdoc/singleton/rdoc/Singleton.html

- составить список вопросов
- выделить одну интересную и запомнившуюся особенность/метод/факт связанный с новым материалом

Практика:

Написать Ruby-скрипт, который эмулирует работу командной строки.

Условия:
- скрипт должен быть исполняемым
- у него должно быть красивое приветствие для ввода (например ‘> ’)
- он должен уметь обрабатывать следущие команды:
  - help - вывод списка доступных команд с кратким описанием
  - uptime - вывод uptime в формате “112h 53min 45sec”
  - date - вывод текущей даты и времени
  - echo - вывод первого переданного аргумента
- команды должны быть регистро-независимы
- команды должны быть обёрнуты в собственный классы, реализуя следующий интерфейс (например):

```
class HelpCommand < Command
      def self.name
            “help”
      end

      def self.description
            “print this text”
```

```
    end

    def help # no arguments for HelpCommand; might differ for others
        for cmd in Command::ALL_COMMANDS
            say "#{cmd.name} #{cmd.description}"
        end
    end

    def run
        # runs the command
    end
end
```

- Родительский класс Command должен определять protected метод “say”, который выводит текущее время, имя скрипта и переданные параметры
- Класс Command должен определять константу ALL_COMMANDS, которая должна хранить в себе массив всех доступных команд (заполнять можно вручную):

```
Command::ALL_COMMANDS.push(HelpCommand)
```

- Класс Command должен определять классовый (self) метод следующего вида:

```
self.command_by_name(name)
```

который проходит по ALL_COMMANDS и возвращает искомую команду (или nil)

- вместо for/while/и тд использовать each/map/find/ (по-возможности)
- метод #run должен уметь принимать блок от пользователя:
    - принимать два аргумента
        - первый - сообщение для пользователя
        - второй - ожидаем ли мы какой-либо ответ от него
        - если комманда нуждается в взаимодействии с пользователем, она должна вызывать блок с соответсвующими аргументами

пример:

```
... |message, wait_answer|
    puts message
    gets if wait_answer
...
```

Помощь:

- использовать #gets для пользовательского ввода

- можно использовать #print для вывода без переноса строки
- #downcase или #casecmp (http://apidock.com/ruby/String/casecmp) для сравнения комманд
- можно использовать File.read('/proc/uptime') для получения системного uptime в секундах (первое число)
- #split чтобы разбить пользовательский ввод на комманды и их аргументы
- Time.now для текущей даты и времени
- some_method(1, 2, 3, 4) == some_method(*[1, 2, 3, 4])
- в глобальной переменой $0 можно найти имя скрипта (один из способов)

Домашку присылать по адресу: aliaksandr_buhayeu@epam.com с темой письма: MTN:L_4:ИМЯ_ФАМИЛИЯ.

На этом всё, жду вас на следующем занятии!