Занятие 5.

Разбор домашнего задания 4:

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

Тема: Обработка исключений: raise-rescue & throw-catch

Исключительные ситуации (exceptions) — это ошибки возникшие в вашем коде и которые также представлены в виде специальных объектов. Например NoMethodError или NameError:

```
Array.hello
#NoMethodError: undefined method `hello' for Array:Class
hello
#NameError: undefined local variable or method `hello' for main:Object
```

Исключения

В любой программе случаются моменты, когда дела идут не по нужному пути: люди вводят неверные данные, файлы, на которые вы расчитываете не существуют (или у вас нет прав доступа к ним), заканчивается память и др. Есть несколько выходов из подобных ситуаций. Проще всего выйти из программы, когда что-то случилось не так.

Менее радикальное решение – каждый метод должен возвращать что-то вроде статусных данных, в которых указано, была ли обработка успешна, а затем необходимо протестировать полученные от метода данные. Однако тестирование каждого результата сделает код плохочитаемым.

Еще один альтернативный подход – использовать исключения. Когда что-то идет не так (т.е. появляется условие исключения) выдаются исключения. На высоком уровне в

программе будет кусочек кода (обработчик исключений), который будет следить за появлением такого сигнала и реагировать на него определенным способом.

Также в одной программе может быть множество обработчиков исключений, каждый из которых обрабатывает определенные типы ошибок. Исключение проходит через все обработчики, пока не встретит требуемый, если его нет, то программа закрывается. Такое поведение есть в C++, Java и Ruby.

Представим себе текстовый редактор. Пользователь должен ввести имя в диалоге SaveAs и нажать ОК. Так как пользователь сам решает, какие данные вводить, мы не можем знать, имеет ли он права для записи этого файла, есть ли свободное место на диске. Будем использовать исключения: (код для примера)

```
text = editor()
location = ask_user()
begin
  File.open(location, w) do |file|
        save_work(file, text)
  end
rescue
  puts "Сохранение не удалось. Ошибка: #{$!}"
end
```

Теперь если что-то пойдет не так, то программа не завершит работу, данные не потеряются и у нас будет второй шанс.
Все, что находится между begin и rescue защищено. Если появляется исключение, то контроль передается блоку между rescue и end. Глобальная переменная **$!** передает сообщение об ошибке и оно выводится на экран. Для того, чтобы контролировать только отдельные виды исключений, мы упоминаем их классы в rescue.
Например, чтобы обрабатывать только ошибки при записи файла, используем выражение rescue IOError. Если мы хотим перехватывать несколько видов исключений в один обработчик, то перечисляем их через запятую, либо (что удобнее) написать обработчик для каждого вида:

```
rescue IOError
 puts “Не удалось записать на диск -- #{$!}.”
```

```
rescue SystemCallError
 puts “Произошла ошибка системного вызова -- #{$!}.”
end
```

Ниже приведен иерархический список всех стандартных исключительных ситуаций в Ruby: https://dl.dropboxusercontent.com/u/306877/bigsoft/types.png (справа)

Вам не обязательно создавать в вашем коде ошибку, вы можете принудительно вызвать исключительную ситуацию при помощи метода raise:

```
def my_method
 raise "SomeError message ..."
end
my_method
#exceptions.rb:2:in `my_method': SomeError message ... (RuntimeError)
#from exceptions.rb:5:in `<main>'
```

Давайте разберем сообщение об ошибке. Оно содержит весьма полезную информацию, которая необходима вам для исправления ошибки: где находится ошибка (exceptions.rb:2:in `my_method’), сообщение описывающее ошибку (SomeError message …), тип ошибки (RuntimeError), и место где возникла ошибка (#from exceptions.rb:5:in `<main>’).

Обработка ошибок

Реальная польза от всех этих типов ошибок заключается в возможности их обработки. Обработка ошибок — это код, который выполняется только при условии возникновения ошибок. Код, ошибки в котором следует обрабатывать необходимо заключить в блок begin — end, а отлавливание ошибок следует производить при помощи ключевого слова rescue. Пример:

```
begin
 100 / 0
rescue
```

```
 puts "Divider is zero!"
end
#=> Divider is zero!
```

Код после rescue выполнится только после возникновения исключительной ситуации, любой исключительной ситуации!

Как уже говорилось, rescue может принимать параметры — типы исключительной ситуации для того, чтобы обрабатывать лишь один определенный тип ошибок, таким образом можно выполнять различный код для различных ошибок.
Пример:

```
begin
 some_undefined_method_call
rescue NameError
 puts "Undefined method!"
end
#=>Undefined method!
```

Иногда бывает необходимость выполнить кусок кода независимо от того была ошибка или небыло. Для этого существует ensure.
Пример:

```
begin
 some_undefined_method_call
rescue NameError
 p " Undefined method!"
ensure
 p "Ruby"
end
#=>" Undefined method!"
#=>"Ruby"
```

Выможете использовать обработчик ошибок rescue и ensure не только в контексте begin — end, но и в контексте любого блока кода, например в контексте метода или класса.
Пример:

```
def hello(msg = "")
 raise "Empty message!" if msg == ""
 puts(msg)
rescue
 puts "Some Error!"
end
hello("Ruby")
#Ruby

hello
#Some Error!
```

«Кошерная» обработка ошибок

Чтобы в обработчике ошибок иметь доступ к различной информации об ошибке, необходимо использовать следующий синтаксис:

```
def hello(msg = "")
 raise "Empty message!" if msg == ""
 puts(msg)
rescue RuntimeError => error
 puts error.inspect
end
hello #<RuntimeError: Empty message!>
```

Теперь в контексте обработчика ошибок мы имеем доступ к экземпляру ошибки, что дает нам возможность получить некоторые данные об ошибке:

```
def hello(msg = "")
 raise "Empty message!" if msg == ""
 puts(msg)
rescue RuntimeError => error
 puts error.message
 puts error.backtrace
end
hello
```

```
#=>Empty message!
#exceptions.rb:2:in `hello'
#exceptions.rb:9:in `<main>'
```

Создание собственных типов ошибок

Глядя на иерархию исключительных ситуаций можно увидить, что все исключительные ситуации происходят от класса Exception.

Доказательство:

```
puts RuntimeError.superclass #StandardError
puts RuntimeError.superclass.superclass #Exception
```

Хотя все ошибки и происходят от класса Exception, вам следует использовать класс StandartError для наследования, поскольку Exception слишком низкоуровневый класс, который обслуживает между всего прочего еще и ошибки окружения. Пример создания собственной ошибки:

```
class SomeError < StandardError
 def message
   "Some Error!"
 end
end
raise SomeError #exceptions.rb:7:in `<main>': Some Error! (SomeError)
```

Листинг занятия:

```
begin
 puts 'start'
 no_method_call
rescue
 puts 'Exception!'
end
start
Exception!
```

```
=> nil

begin
 puts 'start'
 no_method_call
rescue => a
 puts 'Exception!'
 puts [a.class, a.class.ancestors].inspect
 puts a.message
end
2.1.5 :072 > begin
2.1.5 :073 >    puts 'start'
2.1.5 :074?>   no_method_call
2.1.5 :075?>   rescue => a
2.1.5 :076?>   puts 'Exception!'
2.1.5 :077?>   puts [a.class, a.class.ancestors].inspect
2.1.5 :078?>   puts a.message
2.1.5 :079?>   end
start
Exception!
[NameError, [NameError, StandardError, Exception, Object, Kernel, BasicObject]]
undefined local variable or method `no_method_call' for main:Object
=> nil


begin
 raise "our simple error"
rescue
 puts "We got an error: #{$!}"
end
We got an error: our simple error
=> nil
```

Пример логгера ошибок: выводим что-то при свершении ошибки, но потом всё равно её вызываем:

```
def post_value(value)
```

```
 puts value
end

begin
 post_value
rescue ArgumentError
 puts 'rescue section'
 raise
end

rescue section
ArgumentError: wrong number of arguments (0 for 1)
   from (irb):104:in `post_value'
   from (irb):109
```

Именованный rescue с переменной сам определяет, какая ошибка словилась:

```
begin
 post_valuea
rescue ArgumentError, NameError => ex
 puts "rescue section: #{ex.class}"
 raise
end
rescue section: NameError
NameError: undefined local variable or method `post_valuea' for main:Object
```

Разные обработчики для разных ошибок:

```
begin
 post_value
rescue ArgumentError => ex
 puts "1: rescue section: #{ex.class}"
rescue NameError => ex
 puts "2: rescue section: #{ex.class}"
end
1: rescue section: ArgumentError
=> nil
```

Пример динамического отлова ошибок, на месте our_method может быть что-то более сложное, главное чтобы класс ошибки выводило:

```
def our_method
 NameError
end

begin
 post_valuea
rescue our_method => ex
 puts "rescue section: #{ex.class}"
end
rescue section: NameError
=> nil
```

Ошибка по-умолчанию:

```
begin
 raise
rescue
 p $!.class
end
RuntimeError
=> RuntimeError
```

По-умолчанию, но с сообщением:

```
raise "Our"
RuntimeError: Our
```

Ошибка безопасности с подробным стек-трейсом. Попробуйте у себя.

```
raise SecurityError, "Our Message", caller[1..-1]
SecurityError: Our Message
….. много информации об ошибке …..
```

Интересные примеры обработки исключений:

- default StandardError
- работает как case, только сравнивает по: $!.kind_of?(ExceptionClass) (класс или потомок)

Пример с ensure: (выполняется всегда, назависимо была ошибка или нет)

```
begin
 call_no_method
 puts 'everything is ok'
rescue => a
 puts "Exception catched: #{a.message}"
else
 puts "Congratulations-- no errors!"
ensure # должно быть после rescue
  puts 'we are ensure'
end
Exception catched: undefined local variable or method `call_no_method' for main:Object
we are ensure
=> nil
```

Пример с else но без rescue - так делать нехорошо, rescue - обязателен.

```
begin
call_no_method
puts 'everything is ok'
else
 puts "Congratulations-- no errors!"
ensure # should be after the rescue
 puts 'we are ensure'
end
(irb):103: warning: else without rescue is useless
we are ensure
NameError: undefined local variable or method `call_no_method' for main:Object
```

Пример с retry

```
file_name = 'wrong' # тут имя НЕ существующего файла, который мы будем открывать
begin
puts '*' * 50
puts "start to open the '#{file_name}' file"
puts '*' * 50
File.open(file_name, "r") do |f|
   f.each_line { |line| puts line }
end
rescue
file_name = 'testfile' #  а тут имя существующего файла
puts '*' * 50
puts 'exception handled'
retry
end
```

Задание в классе: отловить ошибку, залогировать и продолжить.

Почему плохо отлавливать Exception, а не его потомков?
Ответ тут:
https://stackoverflow.com/questions/10048173/why-is-it-bad-style-to-rescue-exception-e-in-ruby

Throw и catch**

Инструкции throw и catch являются методами класса Kernel, определяющими управляющую структуру, которую можно себе представить как многоуровневую инструкцию break.
Инструкция throw не просто передает управление из текущего цикла или блока, но фактически может передать его на любое число уровней, становясь причиной выхода из блока, при определении которого использована инструкция catch. В отношении этой

инструкции даже не требуется, чтобы она размещалась в том же методе, что и инструкция throw. Она может быть в вызывающем методе или даже где-нибудь еще дальше по стеку вызовов.

В языках, подобных Java и JavaScript, циклы могут быть поименованы или помечены произвольным префиксом. Когда это сделано, управляющая структура, известная как «помеченное прерывание» («помеченный break»), приводит к выходу из поименованного цикла. В Ruby метод catch определяет помеченный блок кода, а метод throw приводит к выходу из этого блока. Но throw и catch намного универсальнее помеченного break. К примеру, эта пара может быть использована с любой разновидностью инструкций и ее применение не ограничено циклами.

Подходя к объяснению более строго, можно сказать, что throw может распространяться вверх по стеку вызовов, чтобы привести к выходу из блока в вызывающем методе. Те, кто знаком с языками, подобными Java и JavaScript, наверное, узнают в throw и catch ключевые слова, используемые в этих языках для выдачи и обработки исключений. В Ruby работа с исключениями организована по-другому, и как мы видели, в нём используются ключевые слова raise и rescue. Но параллель с исключениями проведена неспроста.

Заметьте, что метод catch воспринимает аргумент-метку и блок. Он выполняет блок и возвращает управление по выходу из блока или когда инструкция throw применяется с указанной меткой.

Инструкция throw также предполагает применение в качестве аргумента метку и заставляет вернуть управление из вызова соответствующей инструкции catch. Если обозначению, переданному throw, не соответствует ни один из вызовов инструкции catch, выдается исключение NameError. При вызове инструкций catch и throw вместо обозначений можно использовать строковые аргументы, которые затем подвергнутся внутреннему преобразованию в метки.

Одной из особенностей throw и catch является способность работать даже в том случае, если они размещаются в разных методах. Мы можем переделать этот код, поместив наиболее глубоко вложенный цикл в отдельный метод, но поток управления все равно будет работать вполне корректно.

Если инструкция throw так и не будет вызвана, вызов catch приведет к возвращению значения последнего выражения, размещенного в ее блоке. Если инструкция throw все же будет вызвана, то значение возвращаемого соответствующей инструкцией catch выражения по умолчанию будет равно nil. Но путем передачи throw второго аргумента можно определить произвольное значение, возвращаемое инструкцией catch.

Возвращаемое catch значение поможет отличить нормальное на выдачу исключения. И тот путь, по которому throw распространяется сквозь лексический контекст, а затем вверх по стеку вызовов, является почти таким же, как и распространение и восхождение по стеку исключения. Но несмотря на схожесть с исключением, лучше все же рассматривать throw и catch в качестве универсальной (хотя и не часто используемой) управляющей структуры, а не механизма исключений. Если нужно просигнализировать об ошибке или возникновении исключительных условий, то вместо throw используется raise. В следующем коде показано, как throw и catch могут быть использованы для «прерывания» вложенных циклов. завершение блока от ненормального его завершения при использовании инструкции throw, что в свою очередь позволит вам написать код, производящий какую-нибудь специальную обработку, необходимую для ответа на throw-завершение. На практике throw и catch не нашли широкого применения. Если возникнет желание применить catch и throw внутри одного и того же метода, то лучше будет рассмотреть возможность переделки catch в отдельный метод и замены throw инструкцией return.

```
def thrower
5.times do |i|
  5.times do |j|
    puts("#{i}.#{j}")
    throw(:label, j) if i > 2 and j > 2
   end
 end
end

val = catch(:label) do
thrower
raise("should never get here")
end
```

что происходит? Блок в catch(:label) запускает метод thrower который выводит нам пары, достигнув же определённого условия, бы прекращаем это дело вернув значение переменной j

```
0.0
0.1
0.2
0.3
0.4
1.0
1.1
1.2
1.3
1.4
2.0
2.1
2.2
2.3
2.4
3.0
3.1
3.2
3.3
=> 3
[5] pry(main)> val
=> 3
```

Если же будет так:

```
val = catch(:label) do
 #thrower
 raise("should never get here")
end
```

То словим исключение и val == nil

```
def caller(a)
throw(:label, "TEST") if a == '!'
end
```

```
a = catch :label do
 i = 0
 while flow = gets
   break if flow.chomp == 'X'
   caller(flow.chomp)
   i += 1
   puts i
 end
 puts 'outside while'
 1
end
puts a
```

Что будет? Принимаем значения с ввода пользователя. Если “X” - то выходим, в противном случае передаём его в caller и там останавливаем если “!”, или не останавливаем:

Пример 1:

```
r
1
e
2
w
3
X
outside while
=> 1
```

и Пример 2:

```
r
1
f
2
!
```

=> "TEST"

А в данном примере мы выходим с :test1 до вывода, благодаря throw

```
a = catch :test1 do
 catch :test2 do
   throw :test1, 'inner'
  end
 puts 'after test2'
end

puts a
=> "inner"
```

Домашнее задание #5:

Теория:

- прочесть заметки лекции ещё раз, два, три…
- подтянуть пробелы по прошлым темам
- изучить следующие ссылки:
    - http://www.skorks.com/2009/09/ruby-exceptions-and-exception-handling/
    - http://phrogz.net/programmingruby/tut_exceptions.html
    - http://ruby.bastardsbook.com/chapters/exception-handling/

  Осмотреться в документации:
- http://www.ruby-doc.org/core-2.2.0/Exception.html

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

Практика:

- разобраться со старыми домашками
- установить Rubocop https://github.com/bbatsov/rubocop , проверить свой скрипт перед отправкой на проверку

- улучшить наш скрипт-эммулятор отлавливанием исключений:
    - скрипт должен выводить понятное сообщение об ошибке в случае невозможности прочтения системного uptime (например когда /proc/uptime нет в системе, Windows…)
    - скрипт должен выходить со статусом “0” и выводить “Good Bye!” даже при попытках завершения через SIGINT (^C).
    - метод #command_by_name должен вызывать исключение при невозможности нахождения метода переданного имени, но при этом, внешний метод\код, который его вызывает, должен эти исключения отлавливать и логгировать как-нибудь.
    - скрипт должен завершать свою работу, если во время пользовательского ввода был получен EOF. (исследовать вопрос)

Домашку присылать по адресу: aliaksandr_buhayeu@epam.com с темой письма: MTN:L_5:ИМЯ_ФАМИЛИЯ

На этом всё, жду вас на следующем занятии! (не забудьте ручки, пожалуйста)