RUBY MTN: Spring 2016, LECTURE#5

Занятие: 5

Тема: Обработка исключений

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 кусок 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 `Save As` и нажать `OK`. 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 `$!` ссылается на последнюю ошибку оно выводится на экран. Для того, чтобы контролировать только отдельные виды исключений, мы упоминаем их классы, перечисляя 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
```

Вспомним иерархию 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`:

```
begin
```

```
  some_undefined_method_call
rescue NameError
  p "Undefined method!"
ensure
  p "Ruby"
end

# =>" Undefined method!"
# =>"Ruby"
```

Вы можете 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:

```
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!


begin
  puts 'start'
  no_method_call
rescue => a
  puts 'Exception!'
  puts [a.class, a.class.ancestors].inspect
  puts a.message
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 чтото 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
```

Именнованный `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
```

Поумолчанию, но с сообщением:

```
raise "Our" # => RuntimeError: Our
```

Ошибка безопасности с подробным стектрейсом. Попробуйте у себя.

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

Интересные примеры обработки исключений:

`rescue` по-умолчанию ловит `StandardError` и 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 `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

Тема: Работа с файлами. `IO`, `File`, `Dir`, `FileUtils`

Файлы в программах играют роль хранилищ, в которые можно записать любые объекты. В отличие от привычных нам объектов, файлы позволяют хранить данные даже тогда, когда программа завершила свою работу. Именно поэтому они могут использоваться для передачи данных между разными программами или разными запусками одной и той же программы.

Как организована работа с файлами? В самом общем случае работа с файлами состоит из следующих этапов:

1. Открытие файла. Сущность этого этапа состоит в создании объекта класса `File`.
2. Запись или чтение. Вызываются привычные нам методы вывода на экран и не совсем привычные — ввода-вывода.
3. Закрытие файла. Во время закрытия файла происходят действия с файловой системой. С объектом, который создаётся при открытии файла, ничего не происходит, но после этого он указывает на закрытый файл, и производить операции чтения/записи при помощи него уже нельзя.

В стандартной библиотеке `Ruby` достаточно классов упрощающих нашу работу с файлами. Рассмотрим их.

1. Чтение при помощи класса `IO`. Класс `IO` имеет множество методов, которые позволяют производить чтение из текстовых файлов (с двоичными файлами лучше так не работать). Если нужно считать весь текстовый файл, то можно пользоваться методами класса `IO`.

2. Перенаправление потоков. Существует три предопределённые глобальные переменные: `$stdout`, `$stdin` и `$stderr`. Если им присвоить объект класса `File` (создаваемый во время открытия файла), то весь вывод пойдёт в файл, который присвоили переменной `$stdout`. Весь ввод будет браться из файла, который присволи переменной `$stdin`, а все ошибки будут сохраняться в файле, который присвоили переменной `$stderr`. Если нужно работать только с одним файлом на чтение и одним файлом на запись, то обычно используют этот способ. Также очень удобно использовать перенаправление потока ошибок (переменная `$stderr`) для программ, которые работают в составе пакетных файлов и используют только интерфейс командной строки.
3. Универсальный способ. Используется в ситуациях, когда нельзя использовать предыдущие два способа. Реализуется с помощью стандартных и дополнительных библиотек.

Подведём небольшой итог:

- Если нужно считать весь файл целиком, то можно использовать методы класса `IO`
- Если нужно работать только с одним файлом на чтение и только одним файлом на запись, то можно использовать перенаправление потока
- Если нельзя применить два вышеперечисленных способа, то можно использовать универсальный способ работы с файлами, или использовать его для удобства

Чтение при помощи класса IO

Для чтения файла целиком используется метод `#read`. Он считывает весь файл в виде строки. Во время его использования не стоит задумываться об открытии/закрытии файла, так как эти операции скрыты внутри метода.

```
config = IO.read('config.yaml')
config.class # => String
```

Имя файла — это строка.
В примере можно увидеть, как считывается файл `config.yaml` в переменную `config` типа `String`. Теперь к переменной `config` можно применять любые методы из богатого строкового арсенала.

При считывании «двоичных файлов» в операционных системах `Microsoft` использовать данный способ нельзя, так как файл будет считан не до конца. Следует использовать универсальный способ работы с файлами.

Перенаправление потока

Очень часто программист проектирует программу таким образом, чтобы ввод данных осуществлялся с клавиатуры. После сотнидругой циклов отладки программист так устаёт вводить данные, что создаёт файл, который содержит все необходимые входные данные, и перенаправляет на него поток ввода с клавиатуры, добавляя всего одну строчку в начало своей программы:

```
$stdin = File.open('входные_данные.txt')
```

А вот другая история. Программист пишет и отлаживает программу, которая все необходимые данные выводит на экран. Но в конечном итоге программа должна запускаться без участия человека, и её вывод нужно сохранять в файл для дальнейшей обработки. Переписывать всю программу лень, и поэтому в начало своей программы он вставляет парочку волшебных строчек:

```
$stdout = File.open('выходные_данные.txt', 'w')
$stderr = File.open('сообщения_об_ошибках.txt', 'a')
```

Вторым параметром метода `#open` передаётся модификатор доступа, то есть кодовое слово, по которому метод `#open` может предположить то, что вы будете делать с этим файлом. В нашем примере мы использовали модификатор `'w'` (англ. write — писать), который говорит о том, что мы будем только писать в файл.
Причём каждый раз файл будет перезаписываться.

При помощи модификатора `'a'` (англ. append — добавлять) мы указываем, что мы будем добавлять данные в файл, а не перезаписывать, как в случае с `'w'`. Ещё есть модификатор `'r'` только для чтения, например чтобы

обезопасить себя от случайной перезаписи файла, но в данном случае он нам не подходит. После этого весь вывод на экран и сообщения об ошибках записываются в соотвествующие файлы. Для того, чтобы посмотреть пример в действии, предалагаю вам выполнить следующую программу:

```
# test.rb
$stdout = File.open('out.txt', 'w')
$stderr = File.open('err.txt', 'a')

puts 'Очень важные данные,'
puts 'которые будут сохранены в файл'
raise 'Принудительно вызываем ошибку'

$ ruby test.rb

$ cat err.txt
1.rb:6:in `<main>': Принудительно вызываем ошибку (RuntimeError)

$ cat out.txt
Очень важные данные,
которые будут сохранены в файл
```

Универсальный способ работы с файлами

Универсальным способ с использованием метода `File#open`.
Дело в том, что при помощи него можно осуществлять не только считывание, запись и перезапись, но и закрытие файлов (чего нельзя сделать при использовании способа с переменными `$stdout`, `$stdin` и `$stderr`). Это позволяет несколько раз (за время выполнения программы) осуществлять операции открытия/закрытия файла. В виду того, что эта возможность нужна далеко не всегда, то и используется этот способ только тогда, когда использование всех предыдущих невозможно. Чтение из файла "входные-данные.txt" при помощи универсального метода будет выглядеть следующим образом:

```
string = File.open('входные_данные.txt', 'r') { |file| file.read }
```

Модификатор доступа `'r'` указывать необязательно, так как он устанавливается по-умолчанию. Поэтому следующий код тоже верен:

```
string = File.open('входные_данные.txt') { |file| file.read }
```

Запись данных в файл осуществляется методами `#puts`, `#write`, `#print` и тд.

```
File.open('выходные данные.txt', 'w') { |file| file.write string }
File.open('выходные данные.txt', 'a') { |file| file.puts string }
```

> Стоит не забывать, что все опреации работы с файлами относительны текущей директории.

Использование блоков

Когда вы используете метод `File#open`, вы должны закрыть вручную открытий файл при помощи метода `#close`, что несколько неудобно и является причиной ошибок. Гораздо удобнее работать с методом `File#open` при передаче файла в блок кода, который затем закроет файл самостоятельно после окончания работы с ним.

```
File.open("lib/file.rb") do |f|
 f.each do
   # делаем чтото
 end
end
```

Класс `File` является встроенным в язык классом (поэтому вам не нужно подключать его в файл с вашим кодом), и он предоставляет методы для самых популярных манипуляций с файлами. Как и многие другие `io` сущности в `Ruby`, `File` является подклассом (дочерним классом) `IO`, в который подмешивается модуль `Enumerable`.

Полезные методы:

Чтение из файла за раз

```
f = File.new("lib/file.rb")
content = f.read
f.close
```

Читать файл строка за строкой - `#readline` или `#gets`.
Разница заключается в том, что `File.readline` выбрасывает исключительную ситуацию при завершении файла, в то время как `#get`s просто возвращает `nil`.

```
f = File.new("lib/file.rb")

while line = f.gets
  puts line
end
f.close
```

Так как `File` относится к итерируемым объектам (`Enumerable`), то вы можете использовать методы `#each` или `#each_line` для обхода всего содержимого файла и передачи его строк в блок кода.

```
f = File.new("lib/file.rb")
f.each do |line|
  # делаем чтото со строкой
end
f.close
```

Вы также можете производить чтение файла посимвольно при помощи метода `#readchar`, или побайтно используя метод `#readbyte`.

Запись в файл
В режиме записи, при обращении к несуществующему файлу, файл будет создан, иначе, существующий файл будет полностью переписан.

```
f = File.new('lib/file.rb', 'w')
f.puts("a new line will be appended")
f.print("no new line")
f.print(" at all")
puts f.read
>> "a new line will be appended\nno new line at all"
f.close
```

```
# подсчет закоментированных строк
File.open("lib/file.rb").count { |line| line.starts_with?("#") }

# получение всех строк с include в массиве
File.open("lib/file.rb").grep(/^include/)

# подсчет слов в каждой строке
File.open("lib/file.rb").inject(0) do |total, line|
  total += line.split.size
end
```

Удаление файла
Удаление файла осуществляется при помощи метода `File#delete`. Например:

```
File.delete("example.txt") # => Файл был удален
(Не помещен в корзину, как во многих операционных системах!)
```

Конфигурирование с YAML

- http://www.yaml.org/
- http://ruby-doc.org/stdlib-1.9.3/libdoc/yaml/rdoc/YAML.html

Приемущества: конфиги, удобство пользования и чтения, вложенные структуры.

Наш конфиг на `YAML` может выглядеть так:

```
node:
  ip: 10.0.0.51
  user: arc
  password: xidighei
email_list:
 - arc
  - hunter
 - engineer
sms_list:
  - hunter12@sms.mob.ru
```

Как видите, разработчикам `YAML` удалось сохранить лёгкость и читабельность простейшего формата с разделителями, сделав его иерархическим за счёт использования вложенных параметров. Да и возможности работы с типами данных здесь заметно расширены.
Например, конфиг выше, это самый настоящий хэш, один элемент которого представляет собой вложенный хэш, а ещё два - массивы.

Парсим конфиги

```
require 'yaml'
config = YAML.load_file('config.yml')
# => {"node"=> {"user"=>"arc", "ip"=>"10.0.0.51", "password"=>"xidighei"},
# "sms_list"=>["hunter12@sms.mob.ru"], "email_list"=>["arc", "hunter", "engineer"]}

puts config['node']['ip']
=> 10.0.0.51
```

Часто `yaml`-файлы используются для локализации приложений, когда пользователям выводится различные сообщения, в зависимости от выбранной локали/языкка:

```
footers = YAML.load_file("./config/locales/#{locale}.footers.yml")
```

Обратная конвертация, дампим хэш в `yaml`:

```
hash = { a: 1, b: 2 }

File.open("dump.yml", 'w') { |f| f.write hash.to_yaml }
```

Домашнее задание: 5

Теория:

- прочесть заметки лекции ещё раз, два, три...
- подтянуть пробелы по прошлым темам (к тесту, возьмите, пожалуйста ручки)
- 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 `Ruby`

Практика:

- разобраться со старыми домашками
- установить `Rubocop` https://github.com/bbatsov/rubocop, проверить свой скрипт перед отправкой на проверку, должен проходить 100%

- улучшить наш скрипт-эммулятор отлавливанием исключений:
  - скрипт должен не падать и выводить понятное сообщение об ошибке в случае невозможности прочтения системного `uptime` (например когда `/proc/uptime` нет в системе, `Windows` ...), была вызвана неизвестная команда, передан неверный аргумент и тд.. Работа скрипта не должна прерываться при этом
  - скрипт должен выходить со статусом `0` и выводить "Good Bye!" даже при попытках завершения через `SIGINT (^C)` и при вводе строки команды `exit`, но при этом не запускаться, если сам содержит ошибки
  - метод `#command_by_name` должен вызывать исключение при отсутствии метода с переданным именем, но при этом, внешний метод\код, который его вызывает, должен эти исключения отлавливать
- любым возможным способом реализовать логгирование для скрипта:
  - все ошибки записываются в отдельный файл `errors.log`
  - пользовательское взаимодействие записывается в файл `<имя_скрипта>.log` в виде: `<время локальное>: <имя команды, которая была вызвана> : <результат команды> или nil`, если ничего не вызвращает
  - ** реализовать возможность запуска скрипта в `debug`-моде, при котором все ошибки выводятся на экран и логируются не просто, а с подробным stacktrace.
    Пример возможного запуска: `./script_name.rb --debug`
- ** добавить возможность конфигурирования скрипта через `<имя_скрипта>.yaml` файл, в котором можно настроить описание команд, мод-запуска, названия файлов логов, например:

```
output_log: script.log
error_log: script_errors.log
debug: true

commands:
    help: Help command prints help
    uptime: Uptime command shows uptime
    ...
```

Соответсвенно описания команд и названия лог-файлов при запуске будут новые.
По-умолчанию скрипт ищет конфиг в той же директории, что и сам, с таким же именем, только с `.yaml`, либо может получить путь к конфигу через опцию `--config=path_to_config_any_name.yaml`. Работает и без конфига, как раньше.

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

На этом всё, жду вас на следующем занятии!