# raise-rescue & throw-catch. Brief: типы исключений

Исключительные ситуации (exceptions) — это ошибки возникшие в вашем коде и которые также представлены в виде специальных объектов. Лично у меня самой популярной ошибкой, благодаря тому, что яне владею слепым набором, является NoMethodError или NameError:

Array.hello

#NoMethodError: undefined method `hello' for Array:Class

#from (irb):3

#from /home/vladimir/.rvm/rubies/ruby-1.9.2-p0/bin/irb:17:in `<main>'

hello

#NameError: undefined local variable or method `hello' for main:Object

#from (irb):4

#from /home/vladimir/.rvm/rubies/ruby-1.9.2-p0/bin/irb:17:in `<main>'

Ниже приведен иерархический список всех стандартных исключительных ситуаций в Ruby:

Exception

NoMemoryError

ScriptError

LoadError

NotImplementedError

SyntaxError

SignalException

Interrupt

StandardError

ArgumentError

IOError

EOFError

IndexError

LocalJumpError

NameError

NoMethodError

RangeError

FloatDomainError

RegexpError

RuntimeError

SecurityError

SystemCallError

SystemStackError

ThreadError

TypeError

ZeroDivisionError

SystemExit

fatal

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! rescue может принимать параметр — тип 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")

hello #Some Error!

**«Кошэрная» 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)

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 не требуется даже, 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 Ьгеак»), 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, в нем используются ключевые слова raise и rescue, которые будут рассмотрены чуть позже. Но параллель с исключениями проведена неспроста. Вызов throw очень похож Использование инструкции retry не ограничено одними блоками; ее применение всегда приводит к перевычислению вызова ближайшего охватывающего метода. Это означает, что она может быть использована (в версиях до Ruby 1.9) для написания итераторов, подобных следующему, который работает как цикл whi 1 е. Заметьте, что метод catch воспринимает аргумент-обозначение и блок. Он выполняет блок и возвращает управление по выходу из блока или когда инструкция throw применяется с указанным обозначением. Инструкция throw также предполагает применение в качестве аргумента обозначения 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 обозначения. 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. (Как происходит распространение исключения, мы рассмотрим в этой главе чуть позже.) 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.