# Ruby 初級者向けレッスン 第 13 回

okkez@Ruby関西,サカイ@小波ゼミ

# 自己紹介

- okkez（おっきーと読みます）
- 所属は Ruby 関西
- Ruby 歴は二年くらい。もうすぐ三年。
- るりま
- 最近は関数型言語にはまっています

# 自己紹介

- サカイ
- 小波ゼミ4回生
- Ruby はまだまだ初心者です
- 最近卒論のために Java を始めました
- Java と Ruby を教えてくれる人募集中
- チェコスロバキアに行きたいです

# 今回の内容

- メソッド定義
- 例外

# 今回のゴール

- メソッド定義のルールを知る
- 例外の使い方を知る

# Ruby でのメソッド定義

- メソッドとは？
- いろいろなメソッド定義の方法
- メソッドを定義してみる

# メソッドとは？

- 名前を付けた処理のかたまり
- 処理のかたまりに名前を付けることができる
- 複雑な処理を分離して隠蔽できる
- だからプログラムが書きやすくなる
- だからプログラムが読みやすくなる
- ただし適切な名前を付けることが必要

## メソッドとは？

- 名前重要

# メソッドの名前

- +, -, *, /, ** などのように一見、演算子に見えるものもメソッドとして定義されています
- 末尾が ！ で終わるメソッドは破壊的メソッド
- 末尾が ？ で終わるメソッドは true, false を返すメソッド
- 末尾が = で終わるメソッドは左辺値として使用できるメソッド

# いろいろなメソッド定義（1）

- 引数がないメソッド

# いろいろなメソッド定義（2）

- 引数が複数あるメソッド

# いろいろなメソッド定義（3）

- 引数にデフォルト値が指定されているメソッド

# いろいろなメソッド定義（4）

- 最後の引数にハッシュが指定されているメソッド

# いろいろなメソッド定義（5）

- 最後の引数に *a のように * が付いているメソッド

# いろいろなメソッド定義（6）

- 最後の引数に &b のように & が付いているメソッド

# いろいろなメソッド定義（7）

- 全部入れると？

# いろいろなメソッド定義（7）

- まとめると以下のとおり

```
def defname[ ( [ arg [ =val ], ...] [ , *vararg ] [ ,
  &blockarg ] ) ]
  body
end
```

# メソッド定義の方法まとめ

- メソッド定義の括弧は省略できるけど、大体つける
  - 引数がない場合は括弧を省略する。（ローカル変数と区別する場合は括弧を付ける）
- メソッド呼出の括弧も省略できるけど、大体つける
  - Rails では省略できるときは省略するのがスタイル
- メソッドは必ず値を返すが必ずしもその値を使用する必要はない
  - if とか case なんかも値を返しますが、それは別の話

# Rubyで学ぼう例外処理

- 例外処理とは？
- いろいろな例外
- 例外処理の実装
- 例外を発生させる
- 独自の例外クラス

# 例外処理とは？

- 実行時のエラーを処理するための方法
- エラーが起きた時にどうするかを記述できる
- エラーの処理を、検出したい処理ごとにその近くに記述する必要がなく、正常時の処理と例外の処理を分離できる
- だからプログラムが書きやすくなる
- だからプログラムが読みやすくなる

# いろいろな例外(1)

- 整数を0で割算

```
1: # sample1.rb
2: p 1/0
```

# いろいろな例外(1)

```
$ ruby sample1.rb
sample1.rb:1:in `/': divided by 0 (ZeroDivisionError)
        from sample1.rb:1
```

- sample1.rbの1行めで例外オブジェクトが発生
- 例外のクラスはZeroDivisionError
- 例外のメッセージは「divided by 0」
- 例外までの足跡（バックトレース）はsample1.rbの1行め

# いろいろな例外(2)

```
1: # sample2.rb
2: p File.read('/not/found/file')
$ ruby sample2.rb
sample2.rb:1:in `read': No such file or directory -
  /not/found/file (Errno::ENOENT)
        from sample2.rb:1
```

# いろいろな例外(3)

```
1: # sample3.rb
2: p Array.now
$ ruby sample3.rb
sample3.rb:1: undefined method `now' for Array:Class
  (NoMethodError)
```

# いろいろな例外(4)

```
1: # sample4.rb
2: p 1 /
$ ruby sample4.rb
sample4.rb:1: syntax error
```

# いろいろな例外(5)

```
1: # sample5.rb
2: def foo
3:   foo
4: end
5:
6: foo
$ ruby sample5.rb
sample5.rb:2:in `foo': stack level too deep
  (SystemStackError)
    from sample5.rb:2:in `foo'
    from sample5.rb:2:in `foo'
    ...
```

# いろいろな例外(6)

```
1: # sample5a.rb
2:
3: def foo(bar, baz)
4:   bar + baz
5: end
6:
7: foo(1)
$ ruby sample5a.rb
sample5a.rb:7:in `foo': wrong number of arguments (1
  for 2) (ArgumentError)
        from sample5a.rb:7
```

# 定義済み例外クラス

- 1.8 では、あらかじめさまざまな例外クラスが定義されている
- 1.9 では、ちょっと変わってるらしい。

# 例外処理の実装

- 文法

```
begin
  式..
[rescue [例外クラス,..] [=> 例外変数名]
  式..]..
[else
  式..]
[ensure
  式..]
end
```

# rescue節

- 単純なrescueの例

```
$ ruby sample1a.rb
Class : ZeroDivisionError
Message : divided by 0
Backtrace :
sample1a.rb:3:in `/'
sample1a.rb:3
```

# rescue節

- 複数のrescueを使用した例

```
# csv.rb:921:
begin
  str_read = read(BufSize)
rescue EOFError
  str_read = nil
rescue
  terminate
  raise
end
```

# rescueする例外の指定

- rescue節に例外クラスを指定すると（複数指定可）、指定した例外クラスとそのサブクラスの例外だけを補足する
- 無指定時はStandardErrorを指定したのと同じ
- ZeroDivisionErrorの場合

```
rescue                      #=> 補足する
rescue StandardError        #=> 補足する
rescue ZeroDivisionError    #=> 補足する
rescue TypeError            #=> 補足しない
```

# rescueする例外の指定

- LoadErrorやSyntaxErrorは、StandardErrorのサブクラスではないので、ただのrescueでは補足できない
- 何でも補足してしまうと、かえって問題に気づきにくい

# 例外オブジェクトを変数に代入

- ’rescue［例外クラス,..］=> 変数名’とすると、指定された変数に$!と同様に発生した例外が代入される

```
1: # sample1b.rb（整数を0で割算）
2: begin
3:   p 1/0
4: rescue => err
5:   puts "Class : #{err.class}"
6:   puts "Message : #{err.message}"
7:   puts "Backtrace :"
8:   puts err.backtrace.join("\n")
9: end
```

# 例外は呼び出し元にさかのぼる

- メソッドの呼び出し元にさかのぼって例外が返る

# else節

- else節は、例外が発生しなかった時に実行される

```
1: # sample1d.rb（整数を0以外で割算）
2: begin
3:   p 1 / 1
4: rescue ZeroDivisionError
5:   puts 'ZeroDivisionError raised'
6: else
7:   puts 'nothing raised'
8: end
```

# ensure節

- ensure節は、例外が発生してもしなくても実行される

```
1: # sample1e.rb（整数を0で割算するかも）
2: begin
3:   p 1 / rand(2)
4: rescue ZeroDivisionError
5:   puts 'ZeroDivisionError raised'
6: else
7:   puts 'nothing raised'
8: ensure
9:   puts 'always run ensure section'
10: end
```

# rescueの中のretry

- rescue節の中でretryを使うと、beginの最初からもう一度実行する

```
1: # sample1f.rb（整数を0で割算するかも）
2: begin
3:   p 1 / rand(2)
4: rescue ZeroDivisionError
5:   puts "try again"
6:   retry
7: end
```

- 何度やってもダメな時に使うと無限ループに

# rescue修飾子

- ’式1 rescue 式2’のようにも書ける（rescue修飾子）
- 捕捉する例外クラスは指定できない（つまりStandardErrorサブクラスだけ）

```
a = (1 / 0 rescue 2)
p a #=> 2
```

# 例外を発生させる

- 問題が起きた時に自動的に例外は発生するが、組み込み関数のraiseを用いて明示的に例外を発生させることもできる
- 文法は以下のとおり
  - raise
  - raise(例外クラス)
  - raise(メッセージ)
  - raise(例外クラス, メッセージ [, バックトレース])
  - (第一引数の「例外クラス」は「例外オブジェクト」も可)

# 例外を発生させる

- 以下のraiseの行を ’raise(TypeError)’ や ’raise(″message″)’ や ’raise(TypeError, ″message″)’ のように変えて実行してみよう

```
1: # sample6.rb（例外を発生させる）
2: begin
3:   raise
4: rescue
5:   puts "Class : #{$!.class}"
6:   puts "Message : #{$!.message}"
7:   puts "Backtrace :"
8:   puts $!.backtrace.join("\n")
9: end
```

# 例外を発生させる

- 引数がないraiseは、同じスレッドの同じブロック内で最後にrescueされた例外オブジェクト（$!）を再発生させる
- そのような例外が存在しなければRuntimeError例外（メッセージは""）を発生させる

# 例外を発生させる

```
1: # sample6a.rb（整数を0で割算）
2: begin
3:   p 1/0
4: rescue
5:   puts "Class : #{$!.class}"
6:   puts "Message : #{$!.message}"
7:   puts "Backtrace :"
8:   puts $!.backtrace.join("\n")
9:   puts '-----'
10:   raise
11: end
```

# 余談：組み込み関数

- Kernelモジュールで定義されているメソッドは、どこからでもレシーバなしに呼び出せるので「組み込み関数」とか「関数風メソッド」とか呼ばれる
- raiseの他にp, print, puts, require, systemなど

# 独自の例外クラス

- 定義済みの例外クラスの他に、独自の例外クラスを作ることができる
- ExceptionクラスやStandardErrorクラスのサブクラスとして定義する

# 演習問題

- 色々なメソッドを定義してみましょう

# Part 1

- 引数を三つ受け取り、引数の内容に応じて計算結果を返すメソッド
  - 三つの引数のうち一つは演算子を表す文字列です
  - 残りの二つは数値です
  - eval 系メソッドの使用は禁止です
  - 最低限、四則演算を実装してください。それ以上実装するのは自由です。

# Part 2

- 先ほど作成したメソッドに以下の機能を追加してください
  - ゼロ除算した場合は例外メッセージを日本語で表示させる
  - 対応していない演算子が与えられた場合は例外を発生させる

# Part 3

- 先ほどまでに作成したメソッドの引数で受け取ることができる数値の個数を無制限にしてください
  - ただし、演算子は一つでいいです。
  - 例えば +,1,2,3 のような引数が渡された場合の結果は 6 です。

# まとめ

- メソッドを定義して処理を分割しよう
- メソッドに適切名前を付けよう
- プログラムにエラー処理はつきもの
- 例外を使うと上品なプログラムになる
- rescueの対象は適宜限定しよう

# 参考文献

- プログラミングRuby 第2版 言語編
  - http://ssl.ohmsha.co.jp/cgi-bin/menu.cgi?ISBN=4-274-06642-8
- たのしいRuby 第2版
  - http://shop.sbcr.jp/bm_detail.asp?sku=4797336617

# 今後の情報源

- 公式Webサイト
  - http://www.ruby-lang.org/
- リファレンスマニュアル
  - http://www.ruby-lang.org/ja/man/
- 日本Rubyの会
  - http://jp.rubyist.net/
- okkez のブログ
  - http://typo.okkez.net