Ruby入門

- 第７章　クラスの作成を理解する -

P270 - P280

DIVE INTO CODE

もくじ

7.10.6　オープンクラスとモンキーパッチ

オープンクラスについて

- Rubyにはクラスの継承に制限がありません
- StringクラスやArreyクラスなどの組み込みライブラリに対しても独自の**クラス**を定義することが可能

※組み込みライブラリ…Ruby 本体に組み込まれているライブラリ。このライブラリに含まれるクラスやモジュールは require を書かなくても使うことが可能

```ruby
class MyString < String
end

s = String.new("sakamoto")
puts s       # "sakamoto"
puts s.class # String

n = MyString.new("nagamatsu")
puts n       # "nagamatsu"
puts n.class # "MyString"

#上記のように、組み込みライブラリのクラスに対してもクラスを継承して独自のクラスを定義することが可能
```

7.10.6　オープンクラスとモンキーパッチ

オープンクラスについて　続き

- 組み込みライブラリのクラスに対して、独自の**メソッド**を定義することが可能

※組み込みライブラリ…Ruby 本体に組み込まれているライブラリ。このライブラリに含まれるクラスやモジュールは require を書かなくても使うことが可能

```ruby
class String
  def shuffle
    #chars 文字列の各文字を配列で返す
    #shuffle 配列の要素をランダムシャッフル
    #join 文字列を連結
    chars.shuffle.join
  end
end


s = String.new("sakamoto")
puts s          # "sakamoto"
puts s.shuffle # "osaaoktm"

#上記のように、組み込みライブラリのクラスに対してメソッドを定義することが可能
```

7.10.6　オープンクラスとモンキーパッチ

モンキーパッチについて

- 既存の実装を**上書き**して、自分が期待する挙動に変更すること

```ruby
class User
  def initialize(name)
    @name = name
  end

  def hello
    "hello!,#{@name}"
  end
end

user = User.new("sakamoto")
puts user.hello  #hello!,sakamoto

class User
  def hello
    "#{@name}さん、こんにちは！"
  end
end

puts user.hello #sakamotoさん、こんにちは！

# 上記の内容より、メソッド「hello」がモンキーパッチされたことがわかる
```

7.10.6　オープンクラスとモンキーパッチ

モンキーパッチについて　応用

- 既存のメソッドをエイリアスメソッドとして残し、上書きしたメソッドの中で既存のメソッドを再利用

```ruby
class User
  def initialize(name)
    @name = name
  end

  def hello
    "hello!,#{@name}"
  end
end

class User
  alias hello_original hello  #既存メソッドをhello_originalとして呼び出せるようにする
  def hello #helloメソッドにモンキーパッチをあてて既存のhelloメソッドも再利用する
    "#{hello_original}じゃなくて、#{@name}さん、こんにちは！"
  end
end

user = User.new("sakamoto")
puts user.hello  #hello!,sakamotoじゃなくて、sakamotoさん、こんにちは！

# 上記の内容より、既存のメソッド「hello」が再利用できることがわかる
```

7.10.6　オープンクラスとモンキーパッチ

オープンクラスとモンキーパッチの弊害

オープンクラスやモンキーパッチはうまく使えば開発の効率を高めることができるが、
以下のような弊害が出る恐れがある

- Rubyに最初から実装されているメソッドを不適切に上書きしたために、プログラム全体の動きがおかしくなった
- Rubyの標準クラスに独自のメソッドを追加したが、追加した本人以外が理解できずに、かえってチーム全体の開発効率を落としてしまった
- 外部ライブラリのコードにモンキーパッチをあてて使っていたが、ライブラリのVerUPにより、予期せぬエラーが発生した

何でもできるからといって、乱用はNG！！！

7.10.7　特異メソッド

特異メソッドについて

- Rubyではクラスやメソッド単位ではなく、オブジェクト単位で挙動を変えることが可能

```ruby
sakamoto = "私はさかもとです"
nagamatsu = "私はながまつです"

def sakamoto.shuffle #オブジェクト.メソッド名で定義する
  chars.shuffle.join
end

puts sakamoto.shuffle  # "はさ私すもとかで"
puts nagamatsu.shuffle # エラーになる

# 上記のように、sakamotoにはメソッドが追加され、nagamatsuには追加されない
# このように特定のオブジェクトにだけ紐づくメソッドを特異メソッドと呼ぶ
```

7.10.8　クラスメソッドは特異メソッドの一種

DIVER クエリの使い方2【テスト駆動開発とメソッドの使用について】

def self.メソッド名　の形で定義されたメソッド

クラスメソッド

Blog.count_titleAのような形で、クラス定義から直接呼び出せる

クラス.メソッド名

```
1   class User
2   end
3
4   def User.hello #クラス構文の外部でクラスメソッドを定義
5     "hello!"
6   end
7
8   class << User #クラス構文の外部でクラスメソッドを定義
9     def hi
10      'hi.'
11    end
12  end
13
14  puts User.hello  # hello!
15  puts User.hi     # hi.
16
17  # RubyではStringやUserのようなクラスもオブジェクトのためクラス（というオブジェクト）に
18  # 特異メソッドを定義するとクラスメソッドのようにみえる
```

Ruby 3.0.0 リファレンスマニュアル > ライブラリ一覧 > 組み込みライブラリ > Regexpクラス

class Regexp

クラスの継承リスト: Regexp < Object < Kernel < BasicObject [edit]

要約

正規表現のクラス。正規表現のリテラルはスラッシュで囲んだ形式で記述します。

```
/^this is regexp/
```

Regexp.new(string) を使って正規表現オブジェクトを動的に生成することもできます。

```
str = "this is regexp"
rp1 = Regexp.new("^this is regexp")
p rp1 =~ str          # => 0
p Regexp.last_match[0] # => "this is regexp"
```

Ruby 3.0.0 から正規表現リテラルは freeze されるようになりました。

```
p /abc/.frozen?
# => true
p /a#{42}bc/.frozen?
# => true
p Regexp.new('abc').frozen?
# => false
```

正規表現 や リテラル/正規表現リテラ

目次

特異メソッド

compile　last_match　quote　union
escape　new　try_convert

APIドキュメントを参照する際は、
クラスメソッドを特異メソッドに置き換える。

7.10.9　ダックタイピング

ダックタイピングについて

- オブジェクトのクラスが何であろうとそのメソッドが呼び出せれば良しとするプログラミングスタイルのこと

```ruby
def display_name(object)
  puts "名前は#{object.name}です"
end

class User
  def name
    "坂本"
  end
end

class Product
  def name
    "鬼滅の刃"
  end
end

user = User.new
display_name(user)    # 名前は坂本です

product = Product.new
display_name(product) # 名前は鬼滅の刃です
```

7.10.9　ダックタイピング

以下のコード説明は、P277に書いている通り！！！

```ruby
class Product # スーパークラス
  def initialize(name, price)
    @name = name
    @price = price
  end

  def display_text
    stock = stock? ? 'あり' : 'なし' # stock?メソッドはサブクラスで必ず実装してもらう想定
    "商品名: #{@name} 価格: #{@price} 在庫: #{stock}"
  end
end

class DVD < Product # サブクラス
  def stock?
    true #仮にtrueとする
  end
end

product = Product.new('鬼滅',1000)
puts product.display_text  # スーパークラスはstock?を持たないためエラーになる

dvd = DVD.new('エヴァ',3000)
puts dvd.display_text      # 商品名: エヴァ 価格: 3000 在庫: あり
```

```ruby
def stock?
  raise "サブクラスにストックを実装する必要があります"
end
```

Ruby（動的型付け言語）は「コードが実行された瞬間に、そのメソッドが呼び出せるか否か」

Java、C#（静的型付け言語）は「実行前にそのメソッドが100%確実に呼び出せなければいけない」

動的型付け言語の特性を利点として、ダックタイピングなどのテクニックを使うことで非常に柔軟なプログラムを書くことができる

P278-279 コラム

```ruby
s = 'sakamoto'

# Srtingクラスはsplitメソッドを持つ
puts s.respond_to?(:split) #true

# nameメソッドは持たない
puts s.respond_to?(:name)  #false
```

```ruby
def add_ten(n)
  n.to_i + 10
end

# 整数を渡す
puts add_ten(1)    # 11

# 文字列やnilを渡す
puts add_ten('2')  # 12
puts add_ten(nil)  # 10
puts add_ten('あ') # 10
```

メソッドの有無を調べるrespond_to?
そのオブジェクトに対して、特定のメソッドが呼び出し可能か確認するメソッド

Rubyでメソッドのオーバーロード?
静的型付け言語ではオーバーロード（多重定義）機能があるが、Rubyにはない。
そのため、to_aメソッドで引数のクラスをチェックしたり、to_iメソッドで明示的に数値に変換する。
こうすることでオーバーロードと同じようなしくみを実現する

オーバーロードとは

「1つのクラスに同じ名前のメソッドを複数定義すること」です。
オーバーロードを利用すると、1つのメソッド名を知っているだけで引数によって処理を切り替えられます。