応用プログラム言語II / 演習II #13

Rubyで本格的オブジェクト指向プログラミング

情報システム学科
坂本政祐
sakapon@sit.ac.jp

はじめに

- 前回は，手軽に書けるスクリプト言語としてのRubyについて学んだ。
- 前回分では，それほど強くOOPLであることを意識しなくても書けたかと思うが，Rubyは前回言ったように「徹底してすべてがオブジェクト」なので，とてもOOPっぽく書くことはできる。
- もちろん，クラスを自作することもできる。しかもRubyの簡潔な文法で。

はじめに

- というわけで今回は，OOPLとしてのRubyについて焦点をあてて学ぶ。
- 具体的には，以前C++で作ってもらった各自のOOコードをもう一度ひっぱり出してもらって，それをRubyに移植し，やはりRubyは楽で良いねというのを実感してもらいたいというのが狙い。

メソッドの作り方

メソッド定義

- メソッドの作り方:
  - def メソッド名～endで定義
  - 返却値を返せる。
    - しかもどんなオブジェクトでも返せる。
    - 返却値が何オブジェクトであるかは，C/C++と違いメソッド名の前には書かない。(動的型付けであるがゆえ)
  - 仮引数を受け取れる。
    - どんなオブジェクトでも。何個でも。
    - やはり型は書かない。

メソッド定義の例

- 例:
  - 引数なし，返却値なし。

method-def0.rb
```ruby
def puts_foo
  puts "foo"
end

puts_foo #=> "foo"
```

  - 引数あり，返却値なし。

method-def1.rb
```ruby
def puts_args(arg0, arg1)
  puts arg0
  puts arg1
end

puts_args "foo", 356 #=> "foo"(改行)356
```


メソッド定義の例

- 例:
  - 引数: Stringオブジェクト
  - 返却値: Stringオブジェクト

method-def2.rb
```ruby
def append_foo(str)
  return str + "foo"
end

append_foo "sakamoto" #=> "sakamotofoo"
```

  - 引数: Arrayオブジェクト
  - 返却値: Arrayオブジェクト

method-def3.rb
```ruby
def jijou(ary)
  return ary.map{|e| e*e}
end

jijou [3, 5, 2] #=> [9, 25, 4]
```


自作メソッドでメソッドチェーン

- もちろんメソッドチェーンにも参加できる。

method-def3-chain.rb
```ruby
def jijou(ary)
  return ary.map{|e| e*e}
end

jijou([3, 5, 2]).sort #=> [4, 9, 25]
```

メソッド内の変数のスコープ

- スコープは?
  - メソッドの中で新規に使った変数はそのメソッド内のみで有効。

method-def3.rb
```
def dainyuu
  var = 100
end

dainyuu
p var #=> undefined local variable or method `var'
```


引数が変数である場合の扱い

- 引数を変数で渡すと，それは（他の言語で言うところの）値渡しか参照渡しか?
- つまり，以下のようなコードで最終的に str は "sakamoto" なのか "foo" なのか?

hikisuu-test.rb
```
def hikisuu_test(arg)
  arg = "foo"
end

str = "sakamoto"
hikisuu_test(str)
p str #=> ?
```


破壊的メソッドとオブジェクトID

- これはちょっと難しい話なので，まずはじめに「破壊的メソッド」と「オブジェクトID」について説明する。
  - Rubyのオブジェクトには，すべて内部的にオブジェクトIDという数値が振られている。これは object_id メソッドで知ることができる。
  - 以下のようなソースでは，String#upcase のような「レシーバを破壊しないメソッド」の場合，新たなオブジェクトIDの新たなインスタンスが作られてそれが返却される。
  - ので，str2 のオブジェクトIDはstr1とは異なっている。
  - また，str1 のオブジェクトIDは終始変わっていない。

hihakaiteki-method.rb
```
str = "sakamoto"
p str.object_id        #=> -610069088
str2 = str.upcase      #=> "SAKAMOTO"
p str2.object_id       #=> -610069128
p str                  #=> "sakamoto"
p str.object_id        #=> -610069088
```


破壊的メソッドとオブジェクトID

- 一方，String#upcase!（「!」も含めてメソッド名）というメソッドもあり，これはレシーバそのものを破壊的に変更する。
- より正確に言うと，もしレシーバが変数なら，その変数という名札が貼り付けられているオブジェクトそのものを変更する。
- その変数名札が張り付いているオブジェクトのオブジェクトIDは変わらない。

hakaiteki-method.rb
```
str = "sakamoto"
p str.object_id        #=> -610357778
str.upcase!            #=> "SAKAMOTO"
p str                  #=> "SAKAMOTO"
p str.object_id        #=> -610357778
```


引数が変数である場合の扱い

- 以上を踏まえると先程のソースはこう解釈すれば良い。
  - str は Stringオブジェクト"sakamoto" に貼り付けられた名札。
  - その名札をhikisuu_testメソッドに渡すと，その名札が張り付いているオブジェクトのオブジェクトIDがhikisuu_test側に渡る。
  - arg はそのオブジェクトIDのオブジェクトに貼り付けられた名札。つまりこの時点で，ひとつのオブジェクトに2つの名札(str, arg)が貼り付いている。これは p str.object_id と p arg.object_id をすればすぐ確認できる。
  - しかし，arg = "foo" すると，今度は arg という名札は "foo" という String オブジェクトに新たに貼り付く。
  - なので，str は "sakamoto" のまま。

hikisuu-test.rb
```
def hikisuu_test(arg)
  arg = "foo"
end

str = "sakamoto"
hikisuu_test(str)
p str #=> ?
```


引数が変数である場合の扱い

- 一方，右のようなケースでは:
  - argが貼り付いているオブジェクトを破壊的に upcase! するので，結局 str の方も "SAKAMOTO" になる。

hikisuu-test2.rb
```
def hikisuu_test(arg)
  arg.upcase!
end

str = "sakamoto"
hikisuu_test(str)
p str #=> ?
```

- 結局，結論としては:
  - 基本的には参照渡し。（というかRubyの変数はもともとみんな参照（ポインタ））。
  - が，Rubyの「代入」の仕組みをわかっていないと，場合によって参照渡しのようにも値渡しのようにも見えてしまって混乱するかも知れない，ということ。

!系のメソッドと?系のメソッド

- なお，Rubyでは一般的に
  - 末尾に！がつくメソッドは破壊的メソッド。
    - 例) String#sub!, String#gsub!, String#chomp!, Array#map!, Array#compact! ...
  - 末尾に？がつくメソッドは true か false を返すメソッド。
    - 例) String#end_with?, Array#empty?, Array#include?

メソッド定義の場所による違い

- メソッドをどこで定義するか?
  - クラス定義の中でメソッド定義: そのクラスのインスタンスメソッドになる（後述）。
  - クラス定義の中に入っていない場合（これまでの例のように）:
    - Objectクラスというすべてのクラスのスーパークラスがあって，それのインスタンスメソッドになる。
    - 普通のC/C++の「関数」のように，
      - レシーバ無しで，
      - どこからでも呼び出せる

    関数と思えばよい。

クラスの作り方

クラス定義

- 自作クラスの作り方:
  - class クラス名～endで定義。
  - クラス名は大文字で始める。(2文字目以降は小文字にすることが多い。つまりアッパーキャメルケース)
  - クラス名.new でそのクラスのインスタンスを生成することができる。
    - 以降のコード例では#05～#07で使っていたシューティングゲームの例を再び用いているので対比させて見て欲しい。

class-def0.rb
```
class Teki
end

boss = Teki.new
```


自作クラスのインスタンスメソッド

- インスタンスメソッド:
  - クラス定義の中にメソッド定義を書くと，そのクラスのインスタンスメソッドになる。

class-def1.rb
```
class Teki
  def ugokasu
    # ロジックは省略
  end
end

boss = Teki.new
```


自作クラスのコンストラクタ，インスタンス変数

- コンストラクタ，インスタンス変数:
  - コンストラクタのメソッド名はinitialize（そのクラスのクラス名ではない）。new するとinitialize メソッドが呼ばれるということ。
  - インスタンス変数は変数名を@で始める。これも特に宣言は必要ないので，コンストラクタやインスタンスメソッドの中で@付きの変数を使えば，それはすぐそのインスタンスに属する変数になる。

class-def2.rb
```
class Teki
  def initialize(x, y)
    @x = x
    @y = y
  end

  def ugokasu
    # ロジックは省略
  end
end

boss = Teki.new(600, 200)
```


アクセス指定 (メソッドに対するpublic, protected)

- アクセス指定:
  - インスタンスメソッドはデフォルトで public（この辺はC++と思想が違う)なので，特に指定がなければすべてクラスの外に見せることになる。
  - RubyのprivateはC++のそれとは意味が異なり，以下のように対応する。
    - C++のpublic: Rubyのpublic
    - C++のprivate: Rubyのprotected
  - publicやprotectedは，クラス定義の中に書くとそれ以降のメソッド定義に有効になる。publicの後ろなどには「:」は要らない。

アクセス指定(変数に対する)

- インスタンス変数のアクセス指定:
  - インスタンス変数はpublicとかprivate指定ができないので，attr_reader, attr_writer, attr_accessor で指定する。
    - 例) attr_reader :var
      - インスタンス変数@varをクラス定義外から，「インスタンス名.var」という名前で読み可能にする。
    - 例) attr_writer :var
      - インスタンス変数@varをクラス定義外から，「インスタンス名.var」という名前で書き込み可能にする。
    - 例) attr_accessor :var
      - インスタンス変数@varをクラス定義外から，「インスタンス名.var」という名前で読み書き可能にする。

変数に対するアクセス指定の例

- 例

class-def3.rb
```
class Teki
  attr_accessor :x, :y

  def initialize(x, y)
    @x = x
    @y = y
  end

  def ugokasu
    # ロジックは省略
  end
end

boss = Teki.new(600, 200)
p boss.x
boss.y = 210
```


クラスの継承

- 継承:
  - 他のクラスをスーパークラスとして継承することもできる。継承の記号は <。

class-def4.rb
```
# スーパークラス
class Kyara
  attr_accessor :x, :y

  def yomikomu(file_mei)
    # どんなキャラにも共通なファイル読み込み処理
  end

  def hyouji
    # どんなキャラにも共通な表示処理
  end
end
(続く)
```

class-def4.rb(続き)

```ruby
class Teki < Kyara
  def initialize(x, y)
    @x = x
    @y = y
  end

  def ugokasu
    # ロジックは省略
  end
end

class Jiki < Kyara
  def initialize(x, y)
    @x = x
    @y = y
  end

  def ugokasu
    # ロジックは省略
  end
end
```

(続く)

class-def4.rb(続き)

```ruby
# 各インスタンスを生成し，1つのArrayオブジェクトにまとめる
boss = Teki.new(600, 200)
zako = Teki.new(500, 320)
jiki = Jiki.new( 20, 320)
kyaras = [boss, zako, jiki]

# ポリモーフィズム
kyaras.each do |kyara|
  kyara.ugokasu
end
```


C++とRubyのクラス定義の仕方比較

class-def5.cpp

```cpp
class Maguro : public Fish
{
private:
  double omosa;

public:
  Maguro(double w)
  {
    omosa = w;
  }
};

int main(void)
{
  Maguro *magu = new Maguro(250.0);
}
```

class-def5.rb

```ruby
class Maguro < Fish
  def initialize(w)
    @omosa = w
  end
end

magu = Maguro.new(250.0)
```


その他

- クラス定義に関するその他:
  - このほかにはクラスメソッド，クラス変数等の概念があるが，とりあえずは知らなくても大丈夫。でも興味があったらぜひ調べて下さい。

オープンクラス

引数よりレシーバ形式にしよう

- 最初の方に出した以下の例だが:

method-def2.rb (再掲)

```ruby
def append_foo(str)
  return str + "foo"
end

append_foo "sakamoto"
```

method-def3.rb (再掲)

```ruby
def jijou(ary)
  return ary.map{|e| e*e}
end

jijou [3, 5, 2]
```

  - これらはObjectクラスのメソッドとして定義している。
  - 操作対象は引数で渡している。
  - しかしそうではなく，
    - 操作対象はレシーバとして，
    - やりたい操作はその操作対象のオブジェクトのクラスのインスタンスメソッドとして定義した方が，よりOOらしくなる。

オープンクラス

- そのためにRubyでは「オープンクラス」という考え方を採用している。クラスはいつでもオープンされているということ。
- 既存のクラスに対して，いつでも自分で好きなメソッドを追加できる。
- Ruby にはもともと String クラスがあるわけだから，

```ruby
class String
  ...
end
```

というクラス定義がどこかに存在するはずだ。なので，自分のスクリプト内で class String～end の間に自作メソッドを書けば，それは String クラスにメソッドを追加したことになる，ということ。

  - Stringクラスを上書きしちゃうわけではないので安心して下さい。
- このとき，レシーバは self で表す。

オープンクラス

- append_foo メソッドの改良:
  - Stringクラスにappend_fooメソッドを追加。

method-def2-in-string-class.rb

```ruby
class String
  def append_foo
    return self + "foo"
  end
end

"sakamoto".append_foo #=> "sakamotofoo"
```

オープンクラス

applied-progII #13

- jijou メソッドの改良:
  - Arrayクラスにjijouメソッドを追加。

method-def3-in-array-class.rb
```
class Array
  def jijou
    return self.map{|e| e*e}
  end
end

[3, 5, 2].jijou #=> [9, 25, 4]
```

33

自作メソッドでメソッドチェーン再び

applied-progII #13

- このようにすると，メソッドチェーンもより自然になる。

method-def3-chain2.rb
```
class Array
  def jijou
    return self.map{|e| e*e}
  end
end

[3, 5, 2].jijou.sort #=> [4, 9, 25]
```

34

レシーバと返却値は同一クラスである必要はない

applied-progII #13

- また，以上の例では:
  - append_foo は String クラスのメソッドであり，かつ String オブジェクトを返す，
  - jijou は Array クラスのメソッドであり，かつ Array オブジェクトを返す，

  というふうにたまたま作っていたが，別にレシーバと返却値が同じクラスでなければならないという義務はもちろんない。
- 現に，Array#join は Array クラスのメソッドであるが返すのは String オブジェクト。

35

オープンクラス

applied-progII #13

- 以下は，ローマ字で書かれた文字列を対象に，それを「音節」にわけるメソッド String#to_onsetsu を書いてみた例。
- Stringクラスのメソッドで，返却値は Array オブジェクト。
- なお，正規表現（/[^aiueo]*[aiueo]/という部分）については次回詳述します。

onsetsu.rb
```
class String
  def to_onsetsu
    return self.scan(/[^aiueo]*[aiueo]/)
  end
end

p "sakamoto".to_onsetsu #=> ["sa", "ka", "mo", "to"]
p "yoshioka".to_onsetsu #=> ["yo", "shi", "o", "ka"]
p "aikou".to_onsetsu    #=> ["a", "i", "ko", "u"]
p "aioi".to_onsetsu     #=> ["a", "i", "o", "i"]
```

36

オープンクラス

applied-progII #13

- こう言った書き方を突き詰めていくと，「文章のように流れる」メソッドチェーンを書けるようになる。

two-hours-from-now.rb
```
class Fixnum
  def hours
    return self*3600
  end

  def from_now
    return Time.now + self
  end
end

p 2.hours.from_now #=> 2時間後を表すTimeオブジェクト
```

37

まとめ

applied-progII #13

- Rubyは本格的OOPLではあるけれども，その文法は簡潔でとっつきやすい。「仮想関数」や「publicな継承」などの難しい概念も要らない。
- C++やJavaで本格的に大プロジェクトを書いてしまう前に，高速にプロトタイプを作ってその有効性を検証するような目的にも使うことができる。
- オープンクラスのため，既存クラスに簡単にメソッドを追加できる。

38

今日の課題

applied-progII #13

- 課題1: #07の課題1で最終的にできあがったであろう当時のC++版OOPプログラムを，Ruby版として移植して下さい。完成したらその行数の少なさも感じ取って下さい。

39

今日の課題

applied-progII #13

- 以下指針:
  - std::vector は Array オブジェクトに。
  - std::map は Hash オブジェクトに。
  - printf メソッドはあります。std::cout はありません。が，フォーマット指定が必要なければ，puts や，"#{ }" などで十分でしょう。
  - scanf の代わりに STDIN.gets を使って下さい。
    - Stringオブジェクトとして読み込みたいなら str = STDIN.gets ですが，それだと改行込みになってしまうので str = STDIN.gets.chomp がベターかもしれません。場合によりますが。
    - 数値として読み込みたいなら var = STDIN.gets.to_i です。
  - イテレータやforループは Array#each で。
  - += は使えます。++ は使えません。
  - C の continue は next，C の break は break。
  - lleval というサイトで，コードの断片を試すことができます。
  - std::cout << "a" << var << std::endl; に相当するのは，puts "a#{var}" です。
  - スーパークラスのコンストラクタをサブクラスから呼び出したいときは super とだけ書きます。
  - do {〜} while(条件); に相当するのは，while true; 〜; break unless 条件; end

40

今週の落穂拾い

applied-progII #13

- 動的型付けは便利ですが，その代わりエラーチェックもしてくれません。同じ変数として書いたつもりでも，long_variable_name = 0 と long_variabre_name = 0 では別の変数に代入したとみなされるわけです。メソッド名の間違えは，undefined method が出てくれるのでまだ良いのですが...。
- ですので，今回の課題みたいに長いコードを打つときは殊更気をつけるようにしましょう。

41