超簡約版オブジェクト指向入門
プログラム言語論

櫻井彰人

オブジェクト指向とは

- オブジェクト指向は、非常に奥深い概念
- オブジェクト指向設計:
  - ソフトウェアの上流設計に用いる考え方
- オブジェクト指向プログラミング:
  - コード設計
- ここでは、超簡約版ではあるが(だから、他人に言ってはいけない)、できるだけ具体的なオブジェクト指向プログラミングの考え方を紹介する。

オブジェクトとは

1. オブジェクトは、プログラムで処理する対象(の抽象概念)
- 個別具体的な、例えば、数値の5とか3.14, 文字列 "abcde".
- 複雑なものとしては、配列(の具体的なもの)、Ruby なら、[1,3,5]

2. オブジェクトとは
- 単に対象物だけではなく、それを操作する手段も込みに考える
- 操作する手段というのは、プログラムでは、関数(手続き)。
- 変更するとは限らず、何らかの値を求めることも含む

- 例えば、文字列への変換、Ruby なら、to_s はこの類。

クラスとインスタンス

- 例えば、オブジェクト1 については to_s は "1", オブジェクト2 については to_s は "2" という具合で、まったく規則性はないようであるが、そんなことはない。11に対しては"11"だからもっとたくさん考えれば、表ではなく手順で記述できそうである。

- つまり、整数というオブジェクトのまとまりを考えるとよい。
  - 勿論、浮動小数点数を含めた、まとまりを考えてもよい。
  - 正の整数だけは、少々無駄なような気がするがね。

- どの大きさのまとまりにすべきかは、一概にはいえない。
  - それぞれの目的に応じて、判断することになるが、絶対的にこれが正しいというものはない。

- このまとまりをクラスという。
  - 実は、クラスもオブジェクトと考えることがあるため、混同を避けるために、個別具体的なオブジェクトをインスタンスという。あるクラスの事例(インスタンス)という意味である。

メソッド

- オブジェクト指向プログラミングでは、手続きの書き方が普通の(というか他の)言語とは異なる。一番多い書き方は

  主たるオブジェクト.手続き(他のオブジェクト)

  である。Ruby の場合には、

  主たるオブジェクト.手続き(他のオブジェクト)付加的なブロック

  が使われている。付加的なブロックには、「手続き」に依存するさまざまな機能が記述される。

- こうした、オブジェクトに対応する手続きは、メソッドと呼ばれる。

メソッド 2

- 数値もオブジェクトと考える場合には、四則も、演算ではなくメソッドとなる。例えば、Ruby では正式には、5+4 は

  5.+(4)

  と書かれ、これは、オブジェクト5 に付属する「+」というメソッドを起動することをあらわす。
- そのときの副オブジェクトは4であることを示している。

```
irb(main):001:0> 5.+(4)
=> 9
irb(main):002:0> 5.+4
=> 9
irb(main):003:0> 5. + 4
=> 9
irb(main):004:0> 5.0 + 4
=> 9.0
irb(main):005:0> 5.0.+(4)
=> 9.0
irb(main):006:0> 5.0. +(4)
=> 9.0
irb(main):007:0> 5.0.
irb(main):008:0*
```


メッセージ通信とメソッド呼出

- メソッド呼出は、伝統的には、メッセージ送信と考えられている。
  - これは、並列処理を念頭におき、すなわち、各オブジェクトは自立した存在であり、その動作は、各オブジェクト並列であることを想定しているからである。
- 5.+(4) の例でいえば、オブジェクト5に「+(4)」すなわち、「4を足せ」というメッセージ（メールでもよいし、ショートメールというほうが実感がわくか？）を送ることを意味する。
  - 受け取ったオブジェクト（今は、5）は、結果である9というオブジェクトを作成して、それへのポインタ（参照）を送り返したり、他所へ送ったりするというのが、基本的な動作となる。
- 非実践的なようだが、オブジェクトが遠方にある場合には、まさにメッセージが送られると考えてよい（rpcということもあるが）

オブジェクト by Alan Kay

(I'm not against types, but I don't know of any type systems that aren't a complete pain, so I still like dynamic typing.)

The original conception of it had the following parts.
（3つのうちの一つ）
- I thought of objects being like biological cells and/or individual computers on a network, only able to communicate with messages (so messaging came at the very beginning -- it took a while to see how to do messaging in a programming language efficiently enough to be useful).

OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things.

From Dr. Alan Kay on the Meaning of "Object-Oriented Programming"

定義 by Allan Kay

1. *Everything is an object*
2. *Objects communicate by sending and receiving messages (in terms of objects)*
3. *Objects have their own memory (in terms of objects)*
4. *Every object is an instance of a class (which must be an object)*
5. *The class holds the shared behavior for its instances (in the form of objects in a program list)*
6. *To eval a program list, control is passed to the first object and the remainder is treated as its message*

http://worrydream.com/EarlyHistoryOfSmalltalk/

メッセージ通信 by Allan Kay

Smalltalk is not only NOT its syntax or the class library, it is not even about classes. I'm sorry that I long ago coined the term "objects" for this topic because it gets many people to focus on the lesser idea. (from Alan Kay on Messaging)

The big idea is "messaging" – that is what the kernel of Smalltalk/Squeak is all about. (from Alan Kay on Messaging)

The original conception of it had the following parts.
（3つのうちの一つ）
- I thought of objects being like biological cells and/or individual computers on a network, only able to communicate with messages (so messaging came at the very beginning -- it took a while to see how to do messaging in a programming language efficiently enough to be useful).

Actor model by Carl Hewitt

Intuitively, an ACTOR is an active agent which plays a role on cue according to a script.
Our formalism shows how all of these modes of behavior can be defined in terms of one kind of behaviors: sending messages to actors. (from Actor Induction and Meta-evaluation (1974))

2. *Objects communicate by sending and receiving messages (in terms of objects)*

In November (1972?), I (Alan) presented these ideas and a demonstration of the interpretation scheme to the MIT AI lab. This eventually led to Carl Hewitt's more formal "Actor" approach [Hewitt 73]. In the first Actor paper the resemblance to Smalltalk is at its closest. (from Early History of Smalltalk)

簡単な例: stack

- stack を配列で実装しよう。
  - 個別のスタックに関する操作としては、個別のスタックを作る(配列を用意して、それ専用にする)、push する、popする、空かどうかの質問に答える、が代表的のものである。
- stack は to_s に比べると厄介な点がある。
  - それは、自分の値を持っているうえに(自分の値を持つのは当然だ)、それが、外部からの手続きであるメソッドにより変更されることである。
- メソッドすなわち、プログラムは、各クラスごとに書くことになる。
  - インスタンスは、何らかのシステム関数により、このクラスから「生成」すると考える。
- Ruby では、new というメソッドがそれである。
  - お目にかかるのは、配列のときであり、Array.new や Array.new(5) などとする。Array がクラスであり、これによって、そのインスタンスが作られる。

インスタンス変数

- ところで、クラスからインスタンスを作る作業は、実際にクラスを作るときに考えないといけない。
- stack の例でいえば、stack の内容を保持する配列と、どこまでこの配列が使われているかを示すインデックスとが必要である。
- これらは、インスタンスごとに必要であるので、インスタンスごとに異なった名前にしないといけない。しかし、それ(自分で名前を作ること)は難しいし、できたにしても、それを使い分けるのは難しそうである。
- Ruby の場合には(他の言語でもだいたい同じ)、インスタンス変数という概念が用意されている。個別具体的な変数名はプログラマが考えるのだが、それがインスタンス変数であれば、同じつづり字でそれぞれに異なったインスタンスを参照できるようになっている。

インスタンス変数 2

- 混乱しそうであるが、普通は大丈夫。なぜなら、メソッド内においては、今自分が参照できるインスタンスは自分自身しかないからである。もし仮に他の(同じクラスの)インスタンスを参照したいなら、それは、メソッド呼出で呼び出し側から教えてもらうか、
- 親であるクラスから(クラスメソッドを通じて)教えてもらうことになる。
- とりあえず、stack においては、自分自身のみに関心をもてばよいので、この悩みなしで、プログラムが書ける。

コードの骨格 by Ruby

- エラーに対する処理は省略

```
class Mystack
  @@nostacks =0
  def initialize( max=10 )
    @stack= Array.new(max)
    @index= 0
    @@nostacks += 1
  end
  def push( x )
    @stack[@index] = x
    @index += 1
  end
  def pop
    if @index <=0 then return nil end
    @index -= 1
    return @stack[@index]
  end
  def empty?
    if @index <=0 then
      return true
    else
      return false
    end
  end
  def size
    @stack.length
  end
  def Mystack.howmany?
    return @@nostacks
  end
end
```


クラス変数とクラスメソッド

- Ruby の場合、new メソッドを呼び出すと、initialize という名称の関数が起動される。したがって、関数 initialize を定義しておく必要がある。
- Ruby の場合、@@ で始まる名称はクラス変数名である。
  - クラス変数は、クラスごとに定義される。
  - この例では、新規に生成されたスタックの個数を数える。
- この値を見るメソッドを作っておこう。
- クラスごとに定義されるメソッドをクラスメソッドという。
  - Ruby においては、メソッド名を クラス名.メソッド名として定義すれば、クラスメソッドとなる。

継承

- あるクラスを作るときに、親クラスを定め、その子クラスとして、当該クラスを作成するときに、親クラスのメソッドや変数を受け継ぐことができる。
- なお、単に継承できるだけではなく、それに上書きすることもできる。

カプセル化

- カプセル化(encapusulation)とは、前例のように、インスタンスの状態を表す変数を、クラス定義の中に入れてしまい、外から直接変更できない(参照できない)ようにすることである。
- 外から直接変更できないようにすることで、ガードをかけることができる。すなわち、(メソッドにより定義した)ある特定の方法でしかないようの変更ができないようにできる。
- 例えば、stack を通常の言語で、配列を使って実装すると、当該配列は、ユーザプログラムから直接見えるところにおかざるをえなくなる。

カプセル化 2

- カプセル化しない例:

```
{ int s[10]; int index[1] ;
  index[0] = 0;
  push( s, index, 123 );
  x = pop( s, index);
# ...
}
```

- これは、push や pop という専用関数を用いなくても変更できることを示している。
- しっかりした倫理観(?)をもってプログラミングを行えば、すなわち、所定の関数を用いる以外の方法で、stack内容を変更しないなら、問題は発生しないが、それは保障されるものではない。
- また、stack 用配列を宣言しているスコープ内でしか当該スタックを用いることはできない。
- オブジェクト指向プログラム言語は、こうした不便を一掃するものである。

イテレータとジェネレータ

- ともに、対象の実装に関わらず、当該対象から、一つずつ取り出す機能である
  - 関数とオブジェクトが交錯するところ
- イテレータ:
  - 列、集合、辞書(ハッシュ表)等の要素を一つずつ取り出す方法(必然的にメソッド)
  - 例えば、x.next() を実行する毎に、x の要素が一つずつ返される。
- ジェネレータ:
  - 何を返すかは、プログラム次第。従って、メソッドではなく、関数になることが多い
  - 無限個(不定個)の要素を順番に返すことも

Rubyのループと(内部)イテレータ

- 配列のイテレータ:
  - forループ

```
for i in ['A', 'B', 'C', 'D', 'E'] do
      puts i
end
```

  - イテレータ

```
['A', 'B', 'C', 'D', 'E'].each do |i|
      puts i
end
```


補足

RubyのBlock 構造

- **{..}** または **do..end** でくくったコード

```
{ puts 'This is a block' }
```

```
do
      puts 'this is another block'
end
```

- パラメータをブロック内に引渡す **||**

```
0.upto(9){|i| puts i}
```

i はブロック内に渡される

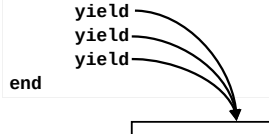
これがブロック

補足

RubyのBlock 付きのメソッド

- ブロックを 3 回呼出すメソッド

```
def tripleCall
      yield
      yield
      yield
end
```



```
tripleCall{ puts "Hello World" }
```

- この **yield** が実行されるとブロックが呼ばれる

補足

RubyのBlock 付きのメソッド

- 0 から **num** までの整数を生成するメソッド
  - さきほどの **upto** メソッドを手作り

```
def fromZeroTo(num)
      i = 0
      while i <= num
            yield i
            i += 1
      end
end
```

  - 0 から 9 までの整数の印字

```
fromZeroTo(9){|i| puts i}
```

（内部）イテレータをループのように

- "Hello World" を3回印字する
  - forループ

```
for i in 1..3 do    # do は省略可
      puts "Hello World"
end
```

  - イテレータ

```
3.times do       # do end は { } も可
      puts "Hello World"
end
```

```
3.times {puts "Hello World"}
```

30

Ruby の外部イテレータ

```
class ArrayIterator
 def initialize(array)
  @array = array
  @index = 0
 end

 def has_next?
  @index < @array.length
 end

 def next_item
  value = @array[@index]
  @index += 1
  value
 end
end
```

```
> array = ['red', 'green', 'blue']
=> ["red", "green", "blue"]
> i = ArrayIterator.new(array)
=> #<ArrayIterator:0x3357df4
@array=["red", "green", "blue"], @index=0>

> while i.has_next?
>   puts("item: #{i.next_item}")
> end
item: red
item: green
item: blue
```

https://qiita.com/kidach1/items/afa4c6c29a6eb6be487a 31

Pythonの（組込み）イテレータ

```
>>> x = iter([1, 2, 3])
>>> x
<listiterator object at 0x1004ca850>
>>> x.next()
1
>>> x.next()
2
>>> x.next()
3
>>> x.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

https://anandology.com/python-practice-book/iterators.html 33

Pythonのイテレータ

```
class yrange:
    def __init__(self, n):
        self.i = 0
        self.n = n

    def __iter__(self):
        return self

    def next(self):
        if self.i < self.n:
            i = self.i
            self.i += 1
            return i
        else:
            raise StopIteration()
```

```
>>> y = yrange(3)
>>> y.next()
0
>>> y.next()
1
>>> y.next()
2
>>> y.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 14, in next
StopIteration
```

https://anandology.com/python-practice-book/iterators.html 34

イテレータ比較 Ruby vs Python

```
class Yrange
 def initialize(n)
  @n = n
  @i = 0
 end


 def next
  if @i < @n then
   i = @i
   @i += 1
   i
  else raise RequestOurOfRange
  end
 end
end
```

```
class yrange:
    def __init__(self, n):
        self.i = 0
        self.n = n

    def __iter__(self):
        return self

    def next(self):
        if self.i < self.n:
            i = self.i
            self.i += 1
            return i
        else:
            raise StopIteration()
```

35

Rubyのジェネレータ

```
def generator()
    i = 0
    while true
        i += 1
        yield i
    end
end
generator(){ |item|
    puts( item )
    if item > 4 then break end
}
```


Python のジェネレータ

```
def generator():
    i = 0
    while True:
        i += 1
        yield i
for item in generator():
    print( item )
    if item > 4:
        break
```


ジェネレータ比較 Ruby vs Python

```
def generator()
    i = 0
    while true
        i += 1
        yield i
    end
end
generator(){ |item|
    puts( item )
    if item > 4 then
      break
    end
}
```

```
def generator():
    i = 0
    while True:
        i += 1
        yield I


for item in generator():
    print( item )
    if item > 4:
        break
```