オブジェクト指向プログラミング入門（その１）

加賀　赳寛

目的：

オブジェクト指向プログラミングがどのようなものかの感覚をつかみましょう。細かいことは後で分かってきますから、理屈は抜きに簡単なプログラムを動かして、どんな風に動くのかを感じていきます。

１．準備作業

オブジェクト指向プログラミングを理解するために Ruby を使用しますので、まずは各自のパソコンに Ruby をインストールしてください。
以下にインストールの方法を示します。

（１）Ruby のインストール
[参考：　　Rubyist Magazine 002 号
Ruby ではじめるプログラミング第 1 回　http://jp.rubyist.net/magazine/?0002-FirstProgramming]

http://rubyforge.org/projects/rubyinstaller/　から、Ruby をインストールしますので
One-Click Installer - Windows　のダウンロードをクリックしてください。

表示されたものの中から最新のモジュール例えば ruby182-15.exe をクリックして適当なフォルダへダウンロードします。

ruby182-15.exe が無事ダウンロードされたら、これをダブルクリックしてインストールしてください。

以上で Ruby が実行できるようになりました。
例題プログラムを格納するためのフォルダを作っておきます。私は説明の都合上 C:¥rubytest フォルダを作りました。

２．まずは簡単なクラスを作りましょう

< プログラム human1.rb >

```
# ----------------------------プログラムは次の行からです-----------------------------------
# 簡単なクラスを作ろう
class Human                    # Human クラス
  def initialize(name, age)    # new(name, age)した時に実行されるメソッドで initialize と書く約束
    @name = name               # @name すなわち名前を設定する インスタンス変数は@ではじまる
    @age = age                 # @age すなわち年齢を設定する インスタンス変数は@ではじまる
  end                          # メソッドの終了（def に対応している）
end                            # クラスの終了　（class　に対応している）

koizumi = Human.new("小泉純一郎", 62)
kiyohara = Human.new("清原和博", 37)
saitou = Human.new("斎藤由貴", 38)
takakura = Human.new("高倉健", 73)
fukuhara = Human.new("福原愛", 17)

p koizumi
p kiyohara
p saitou
p takakura
p fukuhara
#----------------------------プログラムはこの前までです-----------------------------------
```

ファイル名を human1.rb として rubytest フォルダに格納してください。
（注意：ファイル→名前を付けて保存の時に、ファイルの種類はすべての種類にして.txt が付かないように
してください。）

次に　スタート→プログラム→アクセサリ→コマンドプロンプトを開き

C:¥rubytest>ruby –Ks human1.rb

とコマンド入力してください。下記の結果が得られるはずです。

#<Human:0x2ac6a78 @name="小泉純一郎", @age=62>
#<Human:0x2ac6a48 @name="清原和博", @age=37>
#<Human:0x2ac6a18 @name="斎藤由貴", @age=38>
#<Human:0x2ac69e8 @name="高倉健", @age=73>
#<Human:0x2ac69b8 @name="福原愛", @age=17>

５つのインスタンスができています。
それぞれのインスタンスにはインスタンス変数@name,@age があることがわかります。

突然、オブジェクトとかクラスとかインスタンスとかの言葉が説明無しで出てきますが、後で説明する予定ですから、今はわからなくて結構です。気にしないで進んでください。
言葉から先に説明を聞くとかえってわからなくなる恐れがありますので・・・

ここで見られた一つのクラスから任意個数のインスタンスを作ることができる仕掛けは
オブジェクト指向プログラミングの優れた特徴の一つです。

Ruby 文法補足

例題<プログラム human1.rb>を元に幾つかの基本文法について補足します。

赤字のコメント部分が既に説明されています。追加した①～⑤について下の補足を参照してください。

```
# ---------------------------  プログラムは次の行からです-------------------------------------
# 簡単なクラスを作ろう ①
class Human                    # Human クラス ②
  def initialize(name, age)  # new(name, age)した時に実行されるメソッドで initialize と書く約束 ③
    @name = name             # @name すなわち名前を設定する インスタンス変数は@ではじまる ④
    @age = age               # @age すなわち年齢を設定する インスタンス変数は@ではじまる
  end                        # メソッドの終了（def に対応している）
end                          # クラスの終了 （class に対応している）

koizumi = Human.new("小泉純一郎", 62)          #  ⑤
kiyohara = Human.new("清原和博", 37)
saitou = Human.new("斎藤由貴", 38)
takakura = Human.new("高倉健", 73)
fukuhara = Human.new("福原愛", 17)

p koizumi                                      # デバッグプリント文 ⑥
p kiyohara
p saitou
p takakura
p fukuhara
#---------------------------プログラムはこの前までです-------------------------------------
```

補足１（上の①～⑤に対応してます）

① “#”はコメントの開始を意味します。コメント終了は改行です。

② クラス名は 大文字の英字で開始する。 半角英数字、ブランクも半角がベース。（コメントや"文字列"除く）

③ メソッド名は英字、数字、アンダースコア(_)を並べるが、先頭は英字小文字でなければならない。
" ."は特別な記号です（⑤を参照方）

④ 変数名は英字、数字、アンダースコア(_)を並べるが、先頭は英字小文字でなければならない。

⑤ Human というクラスに new("小泉純一郎", 62) という（関数的）メソッドを送ることを示す。
" ."の左側のオブジェクトや式に、右側のメッセージやメソッドを送ると考えてください。

⑥ p はデバッグプリント文です。

補足２ ruby実行時のエラーメッセージについて

エラーとなった場合、エラーメッセージが出て、エラーとなったプログラムの「行番号」と「エラー内容」が出力されます。
それからエラー内容を推定してください。

３．クラスにメソッドを加えてみましょう。

< プログラム human2.rb >

```
# クラスにメソッドを加えてみよう
class　Human
  def　initialize(name, age)
    @name = name
    @age = age
  end
  def　setAge(age)　　# これがメソッド setAge です
    @age = age
  end
  def　name　　# これがメソッド name です
    @name
  end
  def　age　　# これがメソッド age です
    @age
  end
end

koizumi = Human.new("小泉純一郎", 62)
kiyohara = Human.new("清原和博", 37)
saitou = Human.new("斎藤由貴", 38)
takakura = Human.new("高倉健", 73)
fukuhara = Human.new("福原愛", 17)

koizumi.setAge(63)
p koizumi

puts "総理大臣は" + koizumi.name + "です"
puts "年齢は" + koizumi.age.to_s + "です"
puts " "
puts　fukuhara.name + "ちゃんは卓球が得意です"
```

C:¥rubytest>ruby -Ks human2.rb　による実行結果は下記となります。
koizumi.name によりメソッド name を使って@name すなわち　小泉潤一郎　を得ています。
koizumi.age　により@age すなわち整数型の　63　を得て、　.to_s により文字型に変換しています。

< 実行結果 >
#<Human:0x2ac6370 @name="小泉純一郎", @age=63>
総理大臣は小泉潤一郎です
年齢は 63 です

福原愛ちゃんは卓球が得意です

４．継承を使ってみましょう

```
class　Human
  def　initialize(name, age)
    @name = name
    @age = age
  end
  def　setAge(age)　　# これがメソッド setAge です
    @age = age
  end
  def name　　# これがメソッド name です
    @name
  end
  def age　　# これがメソッド age です
    @age
  end
end

class Shokugyou < Human
  def　initialize(name, age, shoku)
    @shokugyou = shoku
    super(name, age)
  end
  def shokugyou
    @shokugyou
  end
  def setShokugyou(shoku)
    @shokugyou = shoku
  end
end

koizumi = Shokugyou.new("小泉純一郎", 62, "総理大臣")
p koizumi
puts koizumi.shokugyou
puts koizumi.age
```

```
puts "#{koizumi.name}さんは#{koizumi.shokugyou}です"

koizumi.setShokugyou("自民党総裁")
p koizumi
```

実行結果は以下となります。

```
C:¥rubytest>>ruby -Ks shokugyou.rb
#<Shokugyou:0x2ac6178 @name="小泉純一郎", @shokugyou="総理大臣", @age=62>
総理大臣
62
小泉潤一郎さんは総理大臣です
#<Shokugyou:0x2ac6178 @name="小泉純一郎", @shokugyou="自民党総裁", @age=62>
```

職業は複数あったり、職歴として過去の職業も知りたい場合があるので一つしか記憶できないのは困りますが、この対策は後で述べます。

５．関連を使ってみましょう

```
class   Human
  def   initialize(name, age)
    @name = name
    @age = age
    @humanshoku = nil          # ここに関連を設定するが、最初は値が入っていない
  end
  def   setAge(age)            # これがメソッドsetAgeです
    @age = age
  end
  def name                     # これがメソッドnameです
    @name
  end
  def age                      # これがメソッドageです
    @age
  end
  def setShokugyou(shoku)          # ここで関連をセットしている
    @humanshoku = shoku
  end
  def shokugyou
    @humanshoku.hyouji             # 関連を使って機能を実現（Shokugyouクラスに委譲）
  end
end

class Shokugyou
  def initialize(shoku)
    @shokugyou = shoku
  end
  def hyouji
    @shokugyou
  end
end

koizumi = Human.new("小泉純一郎", 62)
```

```
p koizumi
shoku1 = Shokugyou.new("総理大臣")
p shoku1
koizumi.setShokugyou(shoku1)                    # ここで関連がセットされます
p koizumi
puts "#{koizumi.name}さんは#{koizumi.shokugyou}です"
```

実行結果は下記となります。

```
C:¥rubytest>ruby -Ks kanren.rb
#<Human:0x2ac60b8 @name="小泉純一郎", @humanshoku=nil, @age=62>
#<Shokugyou:0x2ac6010 @shokugyou="総理大臣">
#<Human:0x2ac60b8 @name="小泉純一郎", @humanshoku=#<Shokugyou:0x2ac6010 @shokugy
ou="総理大臣">, @age=62>
小泉純一郎さんは総理大臣です
```

継承は差分プログラミングを実現できるので大きな注目を集めますが、継承を使いすぎると（何段階も深く継承していくと）プログラムが理解しにくくなりますし、継承するためには継承元の内容を知る必要があり、大規模なプログラムを組むのに特に必要な情報隠蔽の特質に反することにもなりかねません。

このため、オブジェクト指向プログラミングでは関連を多く使うほうが一般には推奨されます。関連はそれぞれのクラスに役割を分担させて、各クラス間は関連を持たせることにより全体としては大きな機能を実現することができます。この特質により一つ一つのクラスは小さくまとめることができるので、比較的理解しやすい、保守性の良いプログラムとすることができます。

再利用のためには当然良く考えたクラス設計をすることが重要ですが、既にできているクラスを変更することなく再利用できる仕掛けもないと実際の仕事はうまくいきません。
仮に Human クラスを変更することなく使おうとすると感覚的にはへんでしょうが、次のような関連の取り方もあり得ます。
再利用するために、元のプログラムを修正していたのでは生産効率は高まらず、品質も低下するかもしれません。オブジェクト指向プログラミングはこの観点で従来のプログラミング手法より優れています。

```
class    Human                    # Human クラスを変更できないと仮定した場合です。
  def    initialize(name, age)
    @name = name
    @age = age
  end
  def    setAge(age)         # これがメソッド setAge です
    @age = age
  end
  def name                    # これがメソッド name です
    @name
  end
  def age                     # これがメソッド age です
    @age
  end
end

class Shokugyou                #   止むを得ず、こちらに関連を持たせてみます
  def initialize(human, shoku)
    @human = human           # ここでクラス Human のインスタンスとの関連を取れるようにする
    @shokugyou = shoku
  end
  def setShokugyou(shoku)
    @shokugyou = shoku
  end
  def shokugyou
    @shokugyou
  end
  def name                      # 関連を使って機能を実現（Human クラスに委譲）
    @human.name
  end
  def age                       # 関連を使って機能を実現（Human クラスに委譲）
    @human.age
  end
  def setAge                    # 関連を使って機能を実現（Human クラスに委譲）
    @human.setAge(age)
```

```
  end
end

koizumi = Human.new("小泉純一郎", 62)
p koizumi
koizumishoku = Shokugyou.new(koizumi, "総理大臣")
p koizumishoku
puts "#{koizumi.name}さんは#{koizumishoku.shokugyou}です"
```

```
C:¥rubytest>>ruby -Ks kanren2.rb
#<Human:0x2ac6358 @name="小泉純一郎", @age=62>
#<Shokugyou:0x2ac62c8 @shokugyou="総理大臣", @human=#<Human:0x2ac6358 @name="小泉純一郎", @age=62>>
小泉純一郎さんは総理大臣です
```

６．　ポリモーフィズムってなんだろう

突然、変わった名前をだしてしまいました。ポリモーフィズム（polymorphism）の語源はギリシャ語にあるそうで、poly　“多くの”と morphism “意味の要素”の合成語だそうです。ポリモーフィズムをあえて訳せば同名異型となります。すなわち、異なる型に対して同じ名前の操作を行うことができるといった意味があります。それではオブジェクト指向プログラミングにおけるポリモーフィズムを簡単な例題によって見てみましょう。

```
class　Human
  def　initialize(name, age)
    @name = name
    @age = age
    @humanshoku = nil          # ここに関連を設定するが、最初は値が入っていない
  end
  def　setAge(age)             # これがメソッド setAge です
    @age = age
  end
  def name                     # これがメソッド name です
    @name
  end
  def age                      # これがメソッド age です
    @age
  end
  def setShokugyou(shoku)      # ここで関連をセットしている
    @humanshoku = shoku
  end
  def shokugyou                 # 関連を表示する
    @humanshoku.hyouji          #　政治家と野球選手で異なるクラス（型）だが、
  end                           #　同じメソッドで呼び出せる。これがポリモーフィズム
end

class Seijika
  def initialize(yakuwari, seitou, senkyoku)
    @yakuwari = yakuwari
    @seitou = seitou
```

```
    @senkyoku = senkyoku
  end
  def hyouji              # ポリモーフィズムを実現するために同じメソッド
    puts "職務は" +   @yakuwari   + ",所属政党は" + @seitou + ",選挙区は" + @senkyoku + "です"
  end
end

class Yakyuu
  def initialize(pos, daritsu, daten, homerun)
    @pos = pos
    @daritsu = daritsu
    @daten = daten
    @homerun = homerun
  end
  def hyouji          # ポリモーフィズムを実現するために同じメソッド
    puts "ポジションは" + @pos + ",  打率は" + @daritsu.to_s + ",  打点は" + @daten.to_s +
         ",  ホームランは" + @homerun.to_s + "です"
  end
end


koizumi = Human.new("小泉純一郎", 62)
p koizumi
shokugyou1 = Seijika.new("総理大臣", "自由民主党", "衆議院小選挙区、神奈川 13 区")
p shokugyou1
koizumi.setShokugyou(shokugyou1)
print "#{koizumi.name}さんは"
koizumi.shokugyou
puts " "


kiyohara = Human.new("清原和博", 37)
p kiyohara
pos = "一塁手"
daritsu = 0.228
daten = 27
```

```
homerun = 12
shokugyou2 = Yakyuu.new(pos, daritsu, daten, homerun)
p shokugyou2
kiyohara.setShokugyou(shokugyou2)
print "#{kiyohara.name}さんは"
kiyohara.shokugyou
```

実行結果

```
C:¥rubytest>ruby -Ks poly.rb
#<Human:0x2ac4c00 @name="小泉純一郎", @humanshoku=nil, @age=62>
#<Seijika:0x2ac4b28 @senkyoku="衆議院小選挙区、神奈川 13 区", @seitou="自由民主党"
, @yakuwari="総理大臣">
小泉純一郎さんは職務は総理大臣,所属政党は自由民主党,選挙区は衆議院小選挙区、神奈
川 13 区です

#<Human:0x2ac4978 @name="清原和博", @humanshoku=nil, @age=37>
#<Yakyuu:0x2ac48e8 @homerun=12, @daten=27, @daritsu=0.228, @pos="一塁手">
清原和博さんはポジションは一塁手, 打率は 0.228, 打点は 27, ホームランは 12 です
```

いかがでしたか、ポリモーフィズムを使うと呼び出すクラスごとに処理を分けることなく同じメソッドで指示できるので便利です。
例えば各種の図形（長方形、円、線分、グループ図形など）を表示するのに、同じメソッド　例えば hyouji と指示すれば良く、新しい図形、例えば菱形クラスを追加しても呼び出す方のプログラムは変えなくて良いという大変重要な性質を実現できます。

７．情報隠蔽

大規模なソフトウエアを作るためには「情報隠蔽」ができることが是非必要であると言われています。それではオブジェクト指向プログラミングでは「情報隠蔽」をどのようにして実現するのかをみてみましょう。
またまた、簡単な例題を見てください。クラス Person は名前と生年月日,それに加えて身長を持つものとします。このオブジェクトを使う立場からは名前と年齢および身長を知る必要があるとします。
でも他人が名前を変更したり、うその年齢を教えたりする可能性があると困りますので一般の利用者には名前を取得するメソッド name と年齢を取得するメソッド age、身長を取得するメソッド height しか公開しないというふうにします。
fukuhara = Person.new("福原愛", 1988, 11, 1)のところは一般の利用者でなく、システム管理者が行っていて、利用者は fukuhara オブジェクトを渡されていると考えてください。

年齢は誕生日がくると誰でも一つ年を取りますから今日現在の日付から生年月日をもとに計算しています。
今日現在の日付を取得するために標準ライブラリ date モジュールを組み込み、クラス Date を使いました。

身長は子供の頃には毎月、毎年伸びますので身長を書き換えるメソッド setShinchou(cm)が必要です。ただし、この setSinchou(cm)は一般には公開しないで特定のものだけが使用できるようにしたほうが良いですね。このために protected を宣言します。なお、initialize メソッドは自動的に private になります。なにも宣言してないメソッドは pubulic メソッドです。細かいことは今は分からなくて良いと思います。情報隠蔽のためのアクセス制限の仕掛けが用意されていることを理解してもらえば十分です。

- public メソッド　　--- 何の制限もなく、どこからでも呼び出せます。

- protected メソッド　--- そのメソッドが定義されたメソッドからか、そのサブクラスのメソッドからしか呼び出せません。

- private メソッド　--- 明示的にレシーバを指定して呼び出すことができません。したがって、Private メソッドが定義されたクラスのメソッドか、そのサブクラスのメソッドからしか呼び出せず、さらに、異なるインスタンスを対象にして呼び出すことはできません。

```
# ここからプログラムです
require 'date'                    # 標準ライブラリ date を要求

# クラス Date は標準ライブラリ date モジュールにあり、Date.today により
# 今日現在の日付を取得することができます。メソッドとして year mon day など
# があり、それぞれ年、月、日を得ることができます。

class Person
  def initialize(name, year, mon, day)
    @name = name
    @birthYear = year          # 生年
    @birthMon  = mon           #   月
    @birthDay  = day           #   日
    @shinchou = 0              # 初めは身長は 0 としておく
  end
  def name
    @name
  end
  def age
                # 式が複数行に跨る時は演算子を最後にするなど、続きがあることが
                # Ruby にとって分かるようにすること。

    nenrei = ((Date.today.year - @birthYear)*10000 +
              (Date.today.mon - @birthMon)*100 + (Date.today.day -
              @birthDay))/10000
  end
  def shinchou
    @shinchou
  end

protected                  # protected メソッドであることを宣言している
  def setShinchou(cm)
    @shinchou = cm
  end
end
```

```
# 選手のマネージャは Person を継承することにした。従って Person の protected メソッドを
# 使用可能である。(選手の身長の更新はマネージャはできるが一般の利用者は不可)

class Manager < Person
  def initialize (senshu)
    @senshu = senshu
    super("マネージャ", 1950, 1, 1)        # マネージャの生年月日は適当にした
  end
  def set(cm)
    @senshu.setShinchou(cm)
  end
  def information
    @senshu
  end
end

fukuhara = Person.new("福原愛", 1988, 11, 1)
fukuharaManager = Manager.new(fukuhara)
fukuharaManager.set(155)
fukuhara = fukuharaManager.information

# ここから利用者のプログラムと考えてください

puts fukuhara.name
puts fukuhara.age
puts fukuhara.shinchou

fukuhara.setShinchou(180)  # その１にはこの行はなく、その２にはこの行を含めています
```

実行結果（その１）

```
C:¥rubytest>ruby -Ks inpei.rb
```

福原愛
16
155

実行結果（その２）
C:¥rubytest>ruby -Ks inpei2.rb
福原愛
16
155
inpei2.rb:64: protected method `setShinchou' called for #<Person:0x2b6f248> (NoMethodError)

以上でオブジェクト指向プログラミングの主要な特徴を簡単に例示しました。
本当に理解するためには、まず自分で簡単な例で良いのでプログラムを書いて実行してみることです.

練習問題

個人　と　個人の財布　をクラスにしてください。
個人は少なくとも属性として名前を持つものとします。
財布はお金の入金、出金、残高の機能、すなわちメソッドを用意してください。

個人を 3 人生成してください。
３個の財布を生成してください。
各個人は一つ財布をもつように関連を設定してください。
A さんの財布に 5 万円、B さんの財布に 10 万円、C さんの財布に 5 千円を入れてください。
A,B,C さんから 3 千円の会費を集めますので、それぞれの財布から 3 千円出してください。
結果、財布の残金を求めてください。
なお、エラー処理は今回省略するものとします。