基礎プログラミングII

第5回　オブジェクト指向とクラス定義

変数のスコープ

ある場所で設定した変数が有効な範囲

例題設定

次のような簡単なお財布プログラムを考える。

1. 最初の所持金は1000円
2. ユーザからの入力を繰り返し、その数を所持金に足し続ける（マイナスなら減る）。

最も簡単なプログラム

```
#! /usr/bin/env ruby
fund = 1000
while true
   printf(“残高%d円です。何円出しますか（負の数は入金, 0で終了): ”, fund)
   inout = gets.to_i
   break if inout == 0. #ifを文の後ろにつけると1行で書ける
   fund -= inout
end
```

実行例

```
% ./inout.rb
残高1000円です。何円出しますか (負の数は入金, 0で終了): 30
残高970円です。何円出しますか (負の数は入金, 0で終了): 50
残高920円です。何円出しますか (負の数は入金, 0で終了):-100
残高1020円です。何円出しますか (負の数は入金, 0で終了):[C-d]
```

メソッドを使った場合

右のプログラムを実行するとエラーになる。どこが間違っているか？

```
#! /usr/bin/env ruby
fund = 1000

def purse(withdraw)
   fund -= withdraw
end

while true
   printf("残高%d円です。何円出しますか(負の数は入金, 0で終了): ", fund)
   inout = gets.to_i
   break if inout == 0.
   purse(inout)
end
```

```
#! /usr/bin/env ruby
fund = 1000                          #この変数fundは

def purse(withdraw)                  #メソッドpurse内からは見えない
   fund -= withdraw
end

while true
   printf("残高%d円です。何円出しますか（負の数は入金, 0で終了): ", fund)
   inout = gets.to_i
   break if inout == 0.
   purse(inout)
end
```

改良案

1. 重要な変数の値をメソッドの引数と返却値でやり取りする
2. クラス定義を用いて変数と手続きをカプセル化する

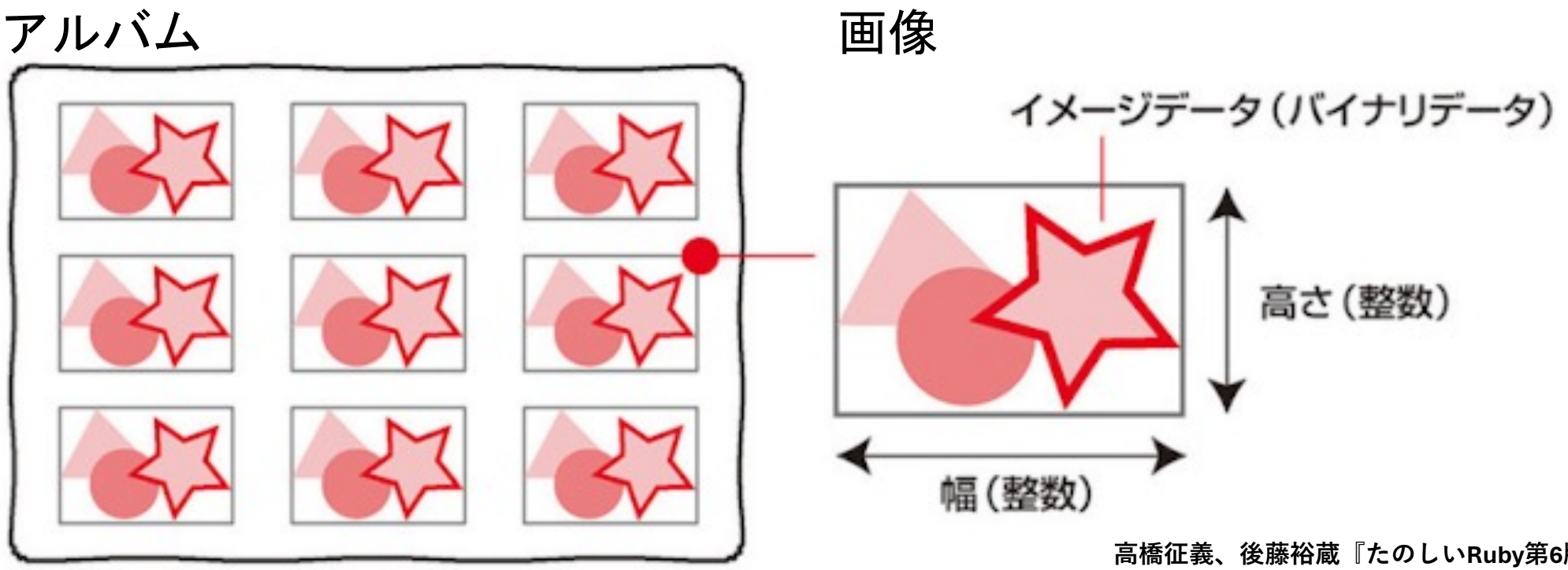
```
#! /usr/bin/env ruby
fund = 1000

def purse(total, withdraw)   # 今の総額と引き出し額を受け取る
  total -= withdraw          # その分だけ総額から引く
end

while true
  printf("残高%d円です。何円出しますか（負の数は入金, 0で終了"): ", fund)
  inout = gets.to_i
  break if inout == 0.
  fund = purse(fund, inout) # 総額と支出をpurseに渡し、結果を受け取る
end
```

オブジェクトとは

プログラムの処理の対象（データ）
データとデータを操作するための手続きをまとめたもの

オブジェクト指向プログラミング

ひとまとまりにしたデータをそれぞれオブジェクトとして扱う



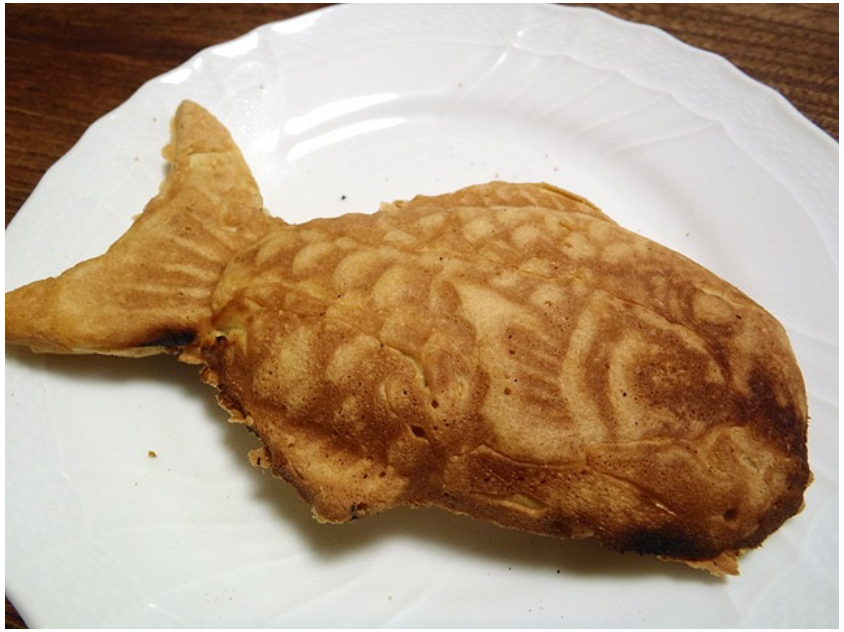
高橋征義、後藤裕蔵『たのしいRuby第6版』

クラスとインスタンス

クラス：ある性質の集合体を設計したもの
インスタンス：実際に動くものとして生み出されたもの

型（クラス）

たい焼き（インスタンス）



https://dailyportalz.jp/kiji/120217153625

お財布プログラムの例題

財布にまつわるお金のやり取りは以下のようにまとめられる。

- 財布に入れるものはお金である。
- 財布は使い始めてから使うのをやめるまでの命である。
- 使い始めるときは最初にいくらか入れた状態で始まる。
- 財布に関する操作は「お金を入れる」と「出す」である。
- 入っている額を超えない範囲で出すことができる。

お財布プログラム：クラスの作成

- 「お財布」を表すクラス名をPurseとする。
- そこに属するのは「中に入る金額」と「入れる操作」と「出す操作」。
- それらを順に変数:fund、メソッド:add、メソッド:subとする。
- 最初にお財布を導入するときの初期金額を設定できる。

クラス定義方法

```
class クラス名
  クラスの定義
end
```

initializeメソッド

```
def initialize(start=0)
  @fund = start
end
```

@で始まる変数はインスタンス変数といい、クラス内で
全てのメソッドから参照できる。

「お財布」を表すクラス

クラス定義も定義しただけでは何も起きず、メソッドと同様、プログラム内で利用する必要がある。

purse.rb

```
#! /usr/bin/env ruby
class Purse
 def initialize(start=0)
 @fund = start
 end
 def add(amount)
  if amount < 0
   STDERR.printf("マイナスはだめよ(%d)\n", amount)
  else
   @fund += amount
   printf("%d円入れました。\n", amount)
  end
 end
 def sub(amount)
  if amount > @fund
   STDERR.printf("そんなに入っていません(残高: %d)\n", @fund)
  else
   @fund -= amount
   printf("%d円出しました。\n", amount)
  end
 end
 def howmuch()                    # 現在の残高を返す
  @fund
 end
end
```

purse.rbを利用するプログラム

```
#! /usr/bin/env ruby
require_relative "purse.rb"                    # 同じディレクトリあるときは require_relative

puts "=== 今日からこの財布を使おう。5000円入れておこう: Purse.new(5000)"
saifu = Purse.new(5000)                        # .newで定義したクラスの実体を生成する。
# 生成するときに initialize メソッドが呼ばれる。

puts "=== 1598円の買い物をしよう。"
puts "=== 2000円出すぞ: saifu.sub(2000)"
saifu.sub(2000)
puts "=== おつりを402円もらった。しまおう: saifu.add(402)"
saifu.add(402)
puts "=== いまいくらあるんだろう: saifu.howmuch"
printf("残り%d円です\n", saifu.howmuch)
```

purse.rbを利用するプログラム

```
#! /usr/bin/env ruby
require_relative "purse.rb"
puts "=== 今日からこの財布を使おう。5000円入れておこう: Purse.new(5000)"
saifu = Purse.new(5000)

puts "=== 1598円の買い物をしよう。"
puts "=== 2000円出すぞ: saifu.sub(2000)"
saifu.sub(2000)
puts "=== おつりを402円もらった。しまおう: saifu.add(402)"
saifu.add(402)
puts "=== いまいくらあるんだろう: saifu.howmuch"
printf("残り%d円です\n", saifu.howmuch)
```

saifu = Purse.new(5000)
定義したPurseクラスの性質を持つインスタンスを一つ生成する(.new)。このとき引数として5000が渡され、それがクラス定義にあるinitializeメソッドに渡される。残高変数@fundに5000が格納された状態でこの財布の利用が始まる。

purse.rbを利用するプログラム

```
#! /usr/bin/env ruby
require_relative "purse.rb"
puts "=== 今日からこの財布を使おう。5000円入れておこう: Purse.new(5000)"
saifu = Purse.new(5000)

puts "=== 1598円の買い物をしよう。"
puts "=== 2000円出すぞ: saifu.sub(2000)"
saifu.sub(2000)
puts "=== おつりを402円もらった。しまおう: saifu.add(402)"
saifu.add(402)
puts "=== いまいくらあるんだろう: saifu.howmuch"
printf("残り%d円です\n", saifu.howmuch)
```

saifu.sub(2000)
saifu変数にはPurseクラスから生まれたオブジェクトが入っている。このオブジェクト内にあるsubメソッドを呼ぶ。

purse.rbを利用するプログラム

```
#! /usr/bin/env ruby
require_relative "purse.rb"
puts "=== 今日からこの財布を使おう。5000円入れておこう: Purse.new(5000)"
saifu = Purse.new(5000)

puts "=== 1598円の買い物をしよう。"
puts "=== 2000円出すぞ: saifu.sub(2000)"
saifu.sub(2000)
puts "=== おつりを402円もらった。しまおう: saifu.add(402)"
saifu.add(402)
puts "=== いまいくらあるんだろう: saifu.howmuch"
printf("残り%d円です\n", saifu.howmuch)
```

saifu.add(402)
同様にオブジェクト内にあるaddメソッドを呼ぶ

複数のインスタンス（オブジェクト）

一つのクラスから生まれるインスタンスは全て違う個体で、独立した情報を持ち、それに従った動きをする。

```
chiharu_saifu = Purse.new(5000) # 千春さんが財布を新しくして5000円から開始
hibiki_saifu  = Purse.new(8000) # 響さんが財布を新しくして8000円から開始
yu_saifu      = Purse.new(1000) # 優さんが財布を新しくして1000円から開始
```

複数のインスタンス（オブジェクト）

chiharu_saifu	
@fund変数	5000 (残高)
addメソッド	入れる処理
subメソッド	出す処理
howmuch メソッド	残高を返す処理

hibiki_saifu	
@fund変数	8000 (残高)
addメソッド	入れる処理
subメソッド	出す処理
howmuch メソッド	残高を返す処理

yu_saifu	
@fund変数	1000 (残高)
addメソッド	入れる処理
subメソッド	出す処理
howmuch メソッド	残高を返す処理

変数の隠蔽

クラス定義(classからendまで)の内側で使用された変数はその外側からは一切見えなくなること。プログラム同士が同じ変数名やメソッド名（まとめて**シンボル**という）で競合して異常動作する可能性をなくせる。

カプセル化

シンボルを隠蔽してプログラムの一定の部分の独立性を高めること。複数人、あるいは将来別のプログラムから利用することを見越したプログラムを作るときには欠かせない。

継承

既に定義されたクラスの基本機能は踏襲しつつ、機能や性質を追加すること。

継承方法

```
class クラス名 < スーパークラス名
    クラスの定義
end
```

「お財布」にカードを入れる機能を追加

purse-with-card.rb

```
require_relative "purse.rb”

class PurseWithCard < Purse
 def initialize(money=0, card=[])
  super(monkey)
  @card = card
 end
 def putinCard(card)
  @card << card
  printf("%s をしまいました。\n", card)
 end
 def drawCard(card)
  @card.delete(card)
  printf("%s を取り出しました。\n", card)
 end
 def myCards()
  @card.join(",")
 end
end
```

まとめ

- ある場所で設定した変数が有効な範囲を変数のスコープと呼ぶ。
- オブジェクトとは、データとデータを操作するための手続きをまとめたもののことである。
- クラスを定義するには、「class クラス名」で始め、「end」で閉じる。