基礎プログラミング+演習 ＃9 – オブジェクト指向

久野 靖 (電気通信大学)

2017.12.11

今回は現在のプログラミングにおいて重要な概念となっているオブジェクト指向です。

- オブジェクト指向の考え方について知る。
- クラス方式のオブジェクト指向言語による記述を学ぶ。

1 前回の演習問題解説

1.1 演習 1 — さまざまなメソッドの計算量

これは簡単に答えだけ書きましょう。

a: $O(1)$, b: $O(n)$, c: $O(n^2)$, d: $O(n)$, e: $O(\log n)$, f: $O(1)$, g: $O(2^n)$, h: $O(2^n)$

最後の方は指数の底は正確でないですが、要は指数計算量だということですね。

1.2 演習 2a — 最大公約数

2 つの数 $M, N (M < N$ とする) の最大公約数のベタなバージョン (M からカウンタを 1 ずつ減らしてゆき、それで両者が割り切れることをチェックする方法) は当然 $O(M)$ になります。

```
def gcdenumerate(x, y)
  min = x; if min > y then min = y end
  (min-1).step(1, -1) do |i|
    if x % i == 0 && y % i == 0 then return i end
  end
end
```

では「大きい方から小さい方を引いてゆく」バージョンはどうなのでしょうか?

```
def gcd(x, y)
  while x != y do
    if x > y
      x = x - y
    else
      y = y - x
    end
  end
  return x
end
```

最善の場合は $M = N$ の時で、すぐ終わります。最悪の場合は $M = 1$ のときで、$N - 1$ 回引き算をしないと終わりません。平均は…? そこで、乱数を使って実験してみました。

```
bench(10000) do gcd(rand(100)+1, rand(100)+1) end → 0.0859375
bench(10000) do gcd(rand(1000)+1, rand(1000)+1) end → 0.1640625
bench(10000) do gcd(rand(10000)+1, rand(10000)+1) end → 0.2734375
bench(10000) do gcd(rand(100000)+1, rand(100000)+1) end → 0.4453125
```

これを見ると、値が 10 倍ずつ大きくなる時に一定くらいずつ (やや多いですが) 値が増えて行きます。引き算ごとに M や N が一定比率くらいで減少して行くと考えれば $O(\log M)$ ということになるわけです。しかし M と N の比率が違うとどうでしょうか。

```
bench(10000) do gcd(rand(100)+1, rand(100)+1) end → 0.203125
bench(10000) do gcd(rand(1000)+1, rand(100)+1) end → 1.328125
bench(10000) do gcd(rand(10000)+1, rand(100)+1) end → 12.1171875
bench(10000) do gcd(rand(100000)+1, rand(100)+1) end → 130.546875
```

M の方が大きいとして、M が 10 倍になると時間も 10 倍になります (これは、M が N の 1000 倍なら 1000 回引く必要があるわけですから当り前ですね)。そうすると、計算量は全体として $O(M \log M)$ ということになるでしょうか。

では、引き算の代わりに剰余演算を使うユークリッドの互除法ではどうでしょう?

```
def gcd3(x, y)
  while true do
    if x > y
      x = x % y; if x == 0 then return y end
    else
      y = y % x; if y == 0 then return x end
    end
  end
end
```

これでまずアンバランスな方から測ってみましょう。

```
bench(10000) do gcd3(rand(100)+1, rand(100)+1) end → 0.0390625
bench(10000) do gcd3(rand(1000)+1, rand(100)+1) end → 0.046875
bench(10000) do gcd3(rand(10000)+1, rand(100)+1) end → 0.046875
bench(10000) do gcd3(rand(100000)+1, rand(100)+1) end → 0.0390625
```

まったく変わりませんね。それは、CPU の割算命令の時間は値が変わってもほとんど一定時間で実行されるからです (ただし Ruby で多倍長演算が必要なくらい大きい値になるとそれは 1 命令ではできなくなるのでこのようには行きません)。では値の大きさの影響はどうでしょうか?

```
bench(10000) do gcd3(rand(100)+1, rand(100)+1) end → 0.0390625
bench(10000) do gcd3(rand(1000)+1, rand(1000)+1) end → 0.0546875
bench(10000) do gcd3(rand(10000)+1, rand(10000)+1) end → 0.0625
bench(10000) do gcd3(rand(100000)+1, rand(100000)+1) end → 0.078125
bench(10000) do gcd3(rand(1000000)+1, rand(1000000)+1) end → 0.0859375
```

こちらは 10 倍になるごとにおよそ一定ずつ増えてゆくようです。それは先と同じで、ループを 1 回まわるごとにだいたい一定比率で 2 つの値が小さくなるからでしょう。これを総合すると、このアルゴリズムの時間計算量は $O(\log M)$ ということになります。[1]

[1]数が大きくなり 1 語に入らなくなると、除算の実行が一定時間と見なせなくなります。その場合、除算の計算に数の桁数に比例する時間、すなわち $O(\log M)$ を要するので、全体の時間計算量は $O(\log^2 M)$ となります。

1.3 演習 2b — フィボナッチ数

再帰的定義そのままの**フィボナッチ数**の計算は、fib(N) の計算に fib($N-1$) と fib($N-2$) を実行し、それらが fib($N-2$) と fib($N-3$)、fib($N-3$) と fib($N-4$) を呼ぶというふうに「倍々」になるので、時間計算量は $O(2^N)$ になります (指数時間)。これに対し、ループで計算する場合は $O(N)$ になります。

```
def fibloop(n)
  x0 = 1; x1 = 1
  (n-1).times do
    t = x0 + x1; x0 = x1; x1 = t
  end
  return x1
end

bench(10000) do fibloop(10) end → 0.0625
bench(10000) do fibloop(100) end → 0.8359375
bench(10000) do fibloop(1000) end → 11.9140625
```

「10 倍になるごとに時間も 10 倍」から外れますが、これは fib(50) くらいから多倍長計算が必要になるためでしょう。

では、行列計算で N 乗の計算を工夫する方法はどうでしょう。「工夫」の漸化式を再掲します。

$$Q^n = \begin{cases} E & (n=0) \\ QQ^{n-1} & (n \text{ が正の奇数}) \\ (Q^{\frac{n}{2}})^2 & (n \text{ が正の偶数}) \end{cases}$$

これを用いるプログラムは次のとおり。[2]

```
def mat22multvec(a, v)
  c = [0, 0]   # 結果用の 1 次元配列
  c[0] = a[0][0]*v[0] + a[0][1]*v[1]
  c[1] = a[1][0]*v[0] + a[1][1]*v[1]
  return c
end
def mat22mult(a, b)
  c = [[0,0],[0,0]]   # 結果用の 2 次元配列
  c[0][0] = a[0][0]*b[0][0] + a[0][1]*b[1][0]
  c[0][1] = a[0][0]*b[0][1] + a[0][1]*b[1][1]
  c[1][0] = a[1][0]*b[0][0] + a[1][1]*b[1][0]
  c[1][1] = a[1][0]*b[0][1] + a[1][1]*b[1][1]
  return c
end
def mat22power(a, n)
  if n < 0
    return [[1,0],[0,1]]  # 2x2 単位行列
  elsif n % 2 == 1
    return mat22mult(mat22power(a, n-1), a)
```

[2] 2×2 行列の積、行列とベクトルとの積を下請けに用意して、それを用いて上の漸化式を使った N 乗を定義し、最後にそれを用いてフィボナッチ数を計算しています。

```
    else
      b = mat22power(a, n/2); return mat22mult(b, b)
    end
  end
  def fibmat(n)
    return mat22multvec(mat22power([[1,1],[1,0]], n-1), [1,1])[0]
  end
```

実験してみましょう。

```
irb(main):101:0> bench(10000) do fibmat(10) end → 0.703125
irb(main):102:0> bench(10000) do fibmat(100) end → 1.46875
irb(main):103:0> bench(10000) do fibmat(1000) end → 2.921875
```

10 倍ごとにほぼ一定ずつ時間が増えています (最後のほうは多倍長になるため外れています)。

計算量はどうでしょうか。「正の奇数」が選ばれるのは N を 2 進表現した時に「1」が現れる回数と等しくなり、「正の偶数」が選ばれるのは N を (奇数なら 1 を引きながら) 半分ずつにしていき 0 になるまでやるので、2 進表現の桁数つまり $\log N$ が回数となります。上記「1」の数は平均すると桁数の半分くらいだから $\frac{\log N}{2}$ となります。これらを合わせると、全体として $O(\log N)$ となります。

1.4 演習 2c — 組み合わせの数

再帰的定義はフィボナッチと同様、倍々の呼び出しのため $O(2^N)$ です。「パスカルの三角形」の場合はどうでしょうか。図 1 を N 段目まで作るとなると、要素数が $\frac{N(N+1)}{2}$ ありますから、計算量としては $O(N^2)$ ということになるはずです。コードを書いてみると次のようになります。

```
  def combarray(n, r)
    a = Array.new(n+1, 1)
    1.step(n) do |i|
      (i-1).step(1, -1) do |k| a[k] = a[k-1] + a[k] end
      # p(a)
    end
    return a[r]
  end
```

これは、N 段までのパスカルの三角形を作るために、要素数 $N+1$ の「1」ばかりが詰まった配列を用意し、それを図 1 のように隣の要素どうし足すことを繰り返していくものです。内側ループを添字が大きい方から順に処理しているのは、そうしないと「1 つ前の値」を使うことができないからです。

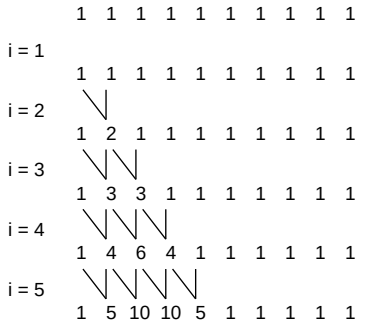

図 1: パスカルの三角形の計算

では時間を計測してみます。

```
bench(10000) do combarray(10,5) end → 0.609375
bench(10000) do combarray(20,10) end → 2.2578125
bench(10000) do combarray(30,15) end → 4.9765625
```

確かに $O(N^2)$ のようです。ところで、前にやった「普通に掛け算する」バージョンはどうでしょうか?

```
def combloop(n, r)
  result = 1
  1.step(r) do |i|
    result = result * (n - r + i) / i
  end
  return result
end
```

これならループが r 回だから ($r \sim N$ として) $O(N)$ となります。

```
bench(10000) do combloop(10,5) end → 0.0703125
bench(10000) do combloop(20,10) end → 0.1171875
bench(10000) do combloop(30,15) end → 0.2265625
```

たしかにずっと速いので、結局、ループで普通に計算するのがよいというオチでした。ただし、「何回もさまざな値を」使うのであれば、パスカルの三角形を「2 次元配列」の上で作成しておいて、そこから値を取り出すようにするのが良さそうです。

1.5 演習 6 — モンテカルロ法の誤差

関数 $y = x$ の区間 $[0, 1)$ における積分を求めてみましょう。答えは両辺が 1 の直角 2 等辺三角形の面積ですから、0.5 であることはすぐ分かります。プログラムは次のとおり。

```
def integrandom(n)
  count = 0
  n.times do
    x = rand(); y = rand()
    if y < x then count = count + 1 end
  end
  return count / n.to_f
end

irb> integrandom 100
=> 0.55              ←誤差 0.05
irb> integrandom 1000
=> 0.475             ←誤差 0.025
irb> integrandom 10000
=> 0.4933            ←誤差 0.0067
irb> integrandom 100000
=> 0.50127           ←誤差 0.00127
irb> integrandom 1000000
=> 0.500674          ←誤差 0.000674
```

試行数が 100 倍になると誤差が $\frac{1}{10}$ になるように見えます。これはなぜでしょうか。

なぜこの方法で面積が求まるのかに立ち帰って考えて見ます。このプログラムでは 1 回の**試行** (trial — サイコロを振ること) で得られるのは「打った点が関数 f の上か下か」つまり「0 か 1 か」の情報

です。そして上であれば count は増やさず (つまり 0 を足し)、下であれば 1 を足し、最後に N で割るので、この「0 か 1 か」の確率変数の平均を求めています。この確率変数が 1 である確率は関数の面積と等しいので、N を増やしていけば**大数の法則** (law of large number) により、観測される平均値は理論的平均値 (この場合は関数の面積) に近づいていきます。そして「どれくらい近づく」かは**中心極限定理** (central limit theorem) が教えてくれます。観測される平均値を $\overline{X}$、真の平均を μ とすると、次の式は $N(0,1)$ つまり平均 0、分散 1 の正規分布に収束します。

$$\sqrt{N}(\overline{X} - \mu)$$

言い換えれば、誤差を $\sqrt{N}$ 倍したものが同じ分布なのですから、試行数を N 倍にすると誤差は $\frac{1}{\sqrt{N}}$ 倍になるわけです。これは上の結果と合致しています。

1.6 演習 7 — 格子上の点で調べる

上でやったのと同じ問題を、格子上の点でやってみましょう。簡単のため、縦横の分割数を同じ値とし、この値 N を与えるようにしました。

```
def integgrid(n)
  count = 0; d = 1.0/n
  n.times do |i|
    y = i*d
    n.times do |j|
      if y <= j*d then count = count + 1 end
    end
  end
  return count / (n**2).to_f
end
```

動かしてみた結果は次のとおり。

```
irb> integgrid 10
=> 0.55
irb> integgrid 100
=> 0.505
irb> integgrid 1000
=> 0.5005
```

実際の格子点の数は 2 乗なので 100、10,000、1,000,000 となっていることに注意。しかしずいぶん「規則的」な数字に見えます。

そこで、区間 $(\frac{0}{100}, \frac{1}{100})$、$(\frac{2}{100}, \frac{3}{100})$、…、$(\frac{98}{100}, \frac{99}{100})$ で 1、それ以外で 0 を値とするというちょっといじわるな関数を考えてみましょう (図 2)。[3]

この関数を計算するメソッドと、それを格子点で積分するメソッドを用意しました。

```
def oddfunc(x)
  t = x*100 % 2
  if 0 < t && t < 1 then return 1 end
  return 0
end
def integoddgrid(n)
  count = 0; d = 1.0/n
```

[3] 区間の両端は含まれて**いない**ことに注意。

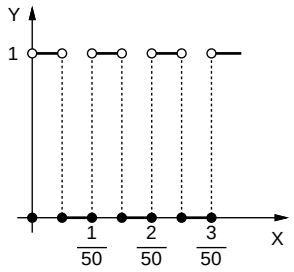

図 2: いじわるな関数

```
  n.times do |i|
    n.times do |j|
      if j*d <= oddfunc(i*d) then count = count + 1 end
    end
  end
  return count / (n**2).to_f
end
```

動かしてみると次のとおり。

```
irb> integoddgrid 10
=> 0.28
irb> integoddgrid 100
=> 0.0496
irb> integoddgrid 1000
=> 0.454546
```

ほぼ「めちゃくちゃ」ですね…(正しくは 0.5 のはずです)。モンテカルロ法ではどうでしょうか。

```
def integoddrandom(n)
  count = 0
  n.times do
    x = rand(); y = rand()
    if y <= oddfunc(x) then count = count + 1 end
  end
  return count / n.to_f
end
```

実行結果は次の通り。

```
irb> integoddrandom 100
=> 0.46
irb> integoddrandom 10000
=> 0.4982
irb> integoddrandom 1000000
=> 0.499454
```

こちらは前と変わらず、試行数を N 倍にすると誤差は $\frac{1}{\sqrt{N}}$ になっています。つまり、格子点だと先のいじわるな関数のように格子のところに段差が当たるなどすると偏った結果が出ておかしくなるわけです。それにそもそも、格子で済むのなら関数の値をもとに普通に台形公式やシンプソンで積分す

る方がずっと高速なのでした。というわけで、「おかしな」関数でもそれなりに計算できるというモンテカルロ法の利点がお分かりいただけたかと思います。

2　オブジェクト指向

2.1　オブジェクト指向とは

これまで、配列やレコード等のデータ構造を作り、それを操作するメソッドを組み合わせてアルゴリズムを実現する、という形のプログラムを作ってきました (図 3 左)。このようなモデルを**手続き型計算モデル**、このモデルに基づくプログラミング言語を**手続き型言語**と呼ぶのでした。

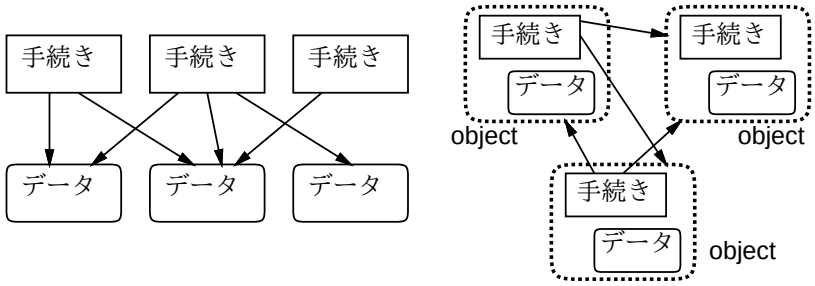

図 3: 手続き型モデルとオブジェクト指向

このプログラミングスタイルは長い間主流として使われてきましたが、近年のようにプログラムが大きくなり複雑化してくると、次のような弱点が問題になってきました。

- データ構造と手続きが分離していて、両者の対応が取りにくい。
- 手続きが複雑になると、どの部分が何を行っているかの把握が困難になる。
- 各データ構造がどの手続きからでも (原理的には) アクセスできるため、本来アクセスすべきでないデータ構造に触ってしまうことによるトラブルが起きやすい。

オブジェクト指向 object-orientation は、上記の点を克服すべく手続き型モデルを拡張した概念で (図 3 右)、プログラムが扱う対象を多様な**もの**、ないし**オブジェクト** (object) として捉える考え方です。

我々が日常扱っている「もの」にはそれぞれ固有の機能や特性があり、我々は「内部構造」には関わらなくてもこれらの「もの」の機能や特性を活用できます。たとえばイスであれば「座る」「高さを調節」「移動させる」などの操作ができますし、ペンであれば「キャップをつける/外す」「描く」などの操作ができ、それぞれ固有の色などもあります。しかし、これらを利用したり参照するのに、イスやペンの内部構造を理解している必要はありません。プログラミングもこれと同様にできれば人間にとってずっと扱いやすくなる、というのがオブジェクト指向の基本的なアイデアです。

今日では多くのプログラミング言語がオブジェクト指向を取り入れています。それらの言語 (オブジェクト指向言語) では、手続きとそれが扱うデータが組になるため対応がつけ易く、個々の手続きは自分の担当するデータのみを直接扱うため簡潔に保ちやすく、データは対応する手続き以外からは操作されないため、不用意に壊される心配が減ります。データを外部から直接アクセスされないようにすることを**カプセル化** (encapsulation) ないし**情報隠蔽** (information hiding) と呼びます。

2.2　クラスとインスタンス exam

前節で述べたような「もの」を言語上でどのように表すかを考えてみましょう。ものには種類ないし**クラス** (class) があるものと考え、その種類ごとに「どんな性質を持つか」「どのような操作ができるか」を定義していく、というのが 1 つの方法です。このような考え方に従うオブジェクト指向言語

をクラス方式 (class based) のオブジェクト指向言語と呼びます。Ruby、Java、C++などの言語はクラス方式のオブジェクト指向言語です。[4]

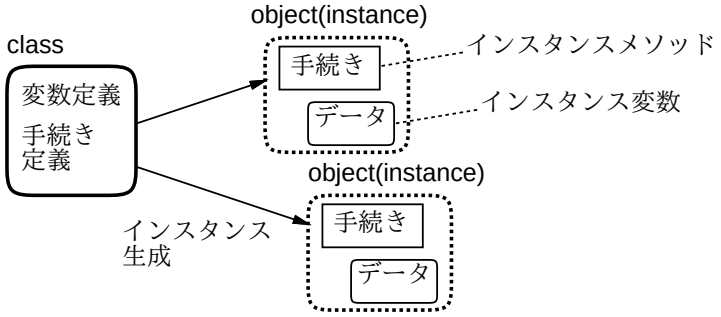

図 4: クラスからインスタンスを生成

では、ものの種類の定義、つまりクラス定義 (class definition) には何が含まれるべきでしょうか (図 4)。上述のように、それぞれの「もの」には固有のデータと固有の操作があるので、それを**変数**と**手続き**ないし**メソッド**で表すのが自然な方法です。これらをそれぞれ、**インスタンス変数** (instance variable)、**インスタンスメソッド** (instance method) と呼びます。

ただし、ここで言う変数とメソッドは、これまで使ってきた変数やメソッドとは少し違っています。つまり、あるクラスを定義し、それをもとに「そのクラスに所属するもの」 — オブジェクト指向言語の言葉で言えば**インスタンス** (instance — 実体と呼ぶこともあります) を生成したとすると、クラス内で定義した変数やメソッドは「そのクラスのインスタンスに付随した」ものとなります。

たとえば図 5 では、クラス定義 `Xyz` を「ひな型」として 2 つのインスタンスを生成し、変数 `x1` と `x2` に入れています。この時、この 2 つのインスタンスは内部に持っているインスタンス変数群も使用できるメソッド群も同じですが、インスタンスとしては別個、つまりインスタンス変数群はそれぞれ別個になっています。つまりクラス定義に書かれているとおりのインスタンス変数群とインスタンスメソッド群を持つということです。

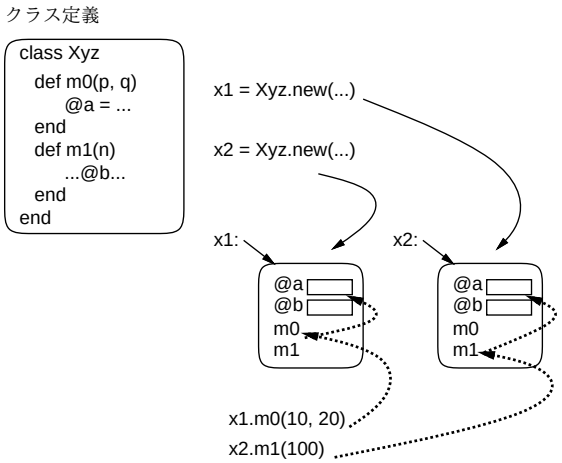

図 5: クラスとインスタンス

インスタンスメソッドを呼び出す時は、メソッド名だけでは「どのインスタンスに付随する」メソッドか特定できないので、インスタンス x に対して「x. メソッド名」の形で指定します。これを

[4]なお、別の方式として**プロトタイプ方式** (prototype based) のオブジェクト指向言語があります。これは、「お手本」となるオブジェクトを (概念的に) コピーして類似したオブジェクトを用意する方式で、JavaScript などが採用しています。

メッセージ送信記法 (message sending) と呼びます。この記法は、既に配列などさまざまな (Ruby が提供してくれている) オブジェクトの機能を呼び出す時に用いてきました。

そして、メッセージ送信記法で「`x1.m0(...)`」「`x2.m1(...)`」のようにインスタンスメソッドを呼び出すと、それらのインスタンスメソッド中でインスタンス変数を参照した時は、それぞれ `x1`、`x2` のインスタンス変数が使われます。

たとえば、「犬」というクラスを作って、そこで「名前」「走っている速さ」というインスタンス変数を持たせたとすると、どの犬もこれら 2 つのインスタンス変数を持っているという点は同じですが、そこに格納されている値、つまりそれぞれの犬の名前やそれぞれの犬の走っている速さは、どの犬かによって、つまりインスタンスによって違う、というわけです。[5]

2.3 Ruby による簡単なクラスの定義 exam

Ruby の場合についてクラス定義の方法を説明します。クラスは次の構文により定義します。

```
class クラス名
  ...
end
```

クラス名は必ず英大文字で始めることになっています。そして、この中にメソッド定義を書くと、自動的にインスタンスメソッドになり、メッセージ送信記法で呼び出せるようになります。また、インスタンス変数はこれまでの変数と異なり、名前の最初が「`@`」で始まります。

そして最後に、クラスからインスタンスを作るには「クラス名`.new(...)`」という特別なメソッド呼び出しを使います。この時、もしインスタンスメソッドの中に `initialize` という名前のものがあればそれが呼び出され、その時 `new` に渡したパラメタがそっくりそのまま渡されてきます。つまり名前どおり、初期化のためにこのような仕組みになっているわけです。[6]

では、クラス定義の例を見てみます (先に説明で使った「犬」をクラスとして定義しました)。

```
class Dog
  def initialize(name)
    @name = name; @speed = 0.0
  end
  def talk
    puts('my name is ' + @name)
  end
  def addspeed(d)
    @speed = @speed + d
    puts('speed = ' + @speed.to_s)
  end
end
```

`initialize` では名前を受け取り、その値でインスタンス変数`@name` を初期化します。インスタンス変数`@speed` は 0 に設定します。メソッド `talk` では自分の名前を喋ります (喋る犬?)。`addspeed` では渡された値だけスピードを増して、それを表示します。動かしてみましょう。

```
irb> a = Dog.new('pochi')
=> #<Dog:0x81b5b2c @name="pochi", @speed=0.0>
```

[5] Ruby ではクラスもオブジェクトなので、クラスに直接付随するメソッドである**クラスメソッド** (class method)、クラスに直接付随する変数である**クラス変数** (class variable) も存在します。クラスメソッドを呼び出す場合はメッセージ送信記法のオブジェクトのところにクラスの名前を指定します。

[6] ここでは使いませんが、クラスメソッドを定義する場合は「`def` クラス名`.` メソッド名 … `end`」のような `def` をクラスの外側に書いて定義します。また変数名を「`@@`」で始めるとその変数はクラス変数になります。

```
irb> b = Dog.new('tama')
=> #<Dog:0x81b049c @name="tama", @speed=0.0>
irb> a.talk
my name is pochi
=> nil
irb> b.talk
my name is tama
=> nil
irb> a.addspeed(5.0)
speed = 5.0
=> nil
irb> b.addspeed(8.0)
speed = 8.0
=> nil
irb> a.addspeed(10.0)
speed = 15.0
=> nil
```

ポチのインスタンスとタマのインスタンスは別であり、名前や速度を別に持つことが分かると思います。このように「もの」単位で扱えるところが、オブジェクト指向の特徴なのです。

演習 1 この例題を打ち込み動かせ。次に「ほえる」メソッド bark(引数無) と、「ほえる回数」を設定するメソッド setcount(回数を渡す) を追加せよ。最初は 3 回ほえるものとする。[7]

演習 2 次のような機能と使い方を持つクラスを作成せよ。使用例の通りに使えることを確認すること。

a. 「覚える」機能を持つクラス Memory。put(x) で与えた容を記憶し、get で取り出す。

```
irb> m1 = Memory.new            # 作る
=> #<Memory:0x81d59e0 @mem=nil>
irb> m1.put(5)                  # 5 を覚えさせる
=> 5                            # put の返値は任意
irb> m1.get                     # 取り出す
=> 5                            # 5
irb> m1.get                     # 再度取り出す
=> 5                            # やはり 5
irb> m1.put(10)                 # 10 を覚えさせる
=> 10
irb> m1.get                     # 取り出す
=> 10                           # 10
```

b. 「文字列を連結していく」クラス Concat。add(s) で文字列 s を今まで覚えているものに連結する (最初は空文字列)。get で現在覚えている文字列を返す。reset で覚えている文字列を空文字列にリセット。(文字列どうしを連結するのは「+」でできます。)

```
irb> c = Concat.new             # 作る
=> #<Concat:0x81c7e94 @str="">
irb> c.add("This")              # 追加
=> "This"
```

[7]もちろん、ほえる回数を憶えるインスタンス変数を追加する必要があるはずです。

```
irb> c.add("is")                # 追加
=> "Thisis"
irb> c.get                      # 取り出す
=> "Thisis"
irb> c.add("a")                 # 追加
=> "Thisisa"
irb> c.reset                    # リセット
=> ""
irb> c.add("pen")               # 追加
=> "pen"
irb> c.get                      # 取り出し
=> "pen"
```

c. 「最大 2 つ覚える」機能を持つクラス Memory2。put(x) で新しい内容を記憶させ、get で取り出す。2 回取り出すと 2 回目はより古い内容が出てくる。取り出した値は忘れる。覚えている以上に取り出すと nil が返る (興味があれば「最大 N 個覚える」をやってもよい)。

```
irb> m2 = Memory2.new  # 作る
=> #<Memory2:0x80fdab8 @mem2=nil, @mem1=nil>
irb> m2.put(1)    # 1 を入れる
=> 1
irb> m2.put(3)    # 3 を入れる
=> 3
irb> m2.put(5)    # 5 を入れる
=> 5
irb> m2.get       # 取り出す → 5
=> 5
irb> m2.get       # 取り出す → 3
=> 3
irb> m2.get       # 取り出す → nil (2 個が限界)
=> nil
irb> m2.put(7)    # 7 を入れる
=> 7
irb> m2.put(9)    # 9 を入れる
=> 9
irb> m2.get       # 取り出す → 9
=> 9
irb> m2.put(11)   # 11 を入れる
=> 11
irb> m2.get       # 取り出す → 11
=> 11
irb> m2.get       # 取り出す → 7
=> 7
```

2.4 例題: 有理数クラス

今度は、もう少し有用なものを作ってみます。これまで、実数の計算には誤差がつきものだという説明をしてきましたね。具体的には、浮動小数点計算では割り切れない除算は循環小数になるので、必ず誤差が生じます。

そこで代わりに、数値を $\frac{分子}{分母}$ という形で保持すれば誤差なく除算結果を保持できるはずです (もちろん、加減算の時は通分して計算し、最後に約分します。そしてもちろん、$\sqrt{2}$ や π などの無理数は扱えません)。

そのような**有理数** (rational number) クラスを作ってみましょう。このクラスでは、インスタンス変数`@a` と`@b` に分子と分母をそれぞれ保持するようにしています。

```
class Ratio
  def initialize(a, b = 1)
    @a = a; @b = b
    if b == 0 then @a = 1; return end
    if a == 0 then @b = 1; return end
    if b < 0 then @a = -a; @b = -b end
    g = gcd(a.abs, b.abs); @a = @a/g; @b = @b/g
  end
  def getDivisor
    return @b
   end
  def getDividend
     return @a
  end
  def to_s
    return "#{@a}/#{@b}"
  end

  def +(r)
    return Ratio.new(@a*r.getDivisor+r.getDividend*@b, @b*r.getDivisor)
  end
  def gcd(x, y)
    while true do
      if x > y then x = x % y; if x == 0 then return y end
      else          y = y % x; if y == 0 then return x end
      end
    end
  end
end
```

`initialize` のパラメタに代入が書いてありますが、これは**デフォルト値** (default value) つまりそのパラメタを省略した場合はこの値を使ってねという意味になります。ですから、「`Rational.new(3)`」で $\frac{3}{1}$ になるわけです。`initialize` の中身がごちゃごちゃしていますが、これは (1) 分母が 0 の時 (不定) は分子を 1 とする、(2) 分母は 0 でないなら常に正とする (負の数は分子が負)、(3) 値ゼロは $\frac{0}{1}$ で表す、(4) 必ず既約分数にする、という**正規化** (normalization — なるべく形を揃えること) を行っているからです。

分母だけ、または分子だけを取り出したい場合のために、メソッド `get_divisor`、`get_dividend` を用意しました。このような、インスタンス変数をアクセスするだけのメソッドのことを**アクセサ** (accessor) と呼びます。Ruby ではアクセサがなければインスタンス変数の内容は外部からは参照できません。これにより、カプセル化が実現され、また後で内部のデータ表現を変えた時にも外部に影響が及ばないで済みます。

また、文字列への変換メソッド `to_s` も用意しました。これは `puts` などによる打ち出しなどの時に自動的に「`a/b`」という形の文字列を生成できるので、用意しておくと便利です。そして、演算とし

てはとりあえず加算だけを用意しました。加算は「+」で表したいので、メソッド名を「+」にしてあります。このようにして、演算子を定義できるのは Ruby の特徴の 1 つです。ただし、C++などの言語でも演算子定義が可能です。

では動かしてみましょう。

```
irb> a = Ratio.new(3,5)
=> #<Ratio:0x81f978c @b=5, @a=3>
irb> puts a
3/5
=> nil
irb> b = Ratio.new(8,7)
=> #<Ratio:0x81f00d8 @b=7, @a=8>
irb> puts b
8/7
=> nil
irb> puts a+b
61/35
=> nil
```

確かに、通分して計算してくれていますね。なお、なぜ puts で打ち出しているかというと、puts は引数を文字列に変換して出力するため to_s を呼んでくれるからです。単に irb の機能で打ち出させるのだと、オブジェクトを表す「#<Ratio ....>」というのが表示されてしまいます。

演習 3 有理数クラスをそのまま打ち込んで動かせ。動いたら、四則の他の演算も追加し、動作を確認せよ。できれば、これを用いて浮動小数点では正確に行えない「実用的な」計算が正確にできることを確認してみよ。

演習 4 **複素数** (complex number) を表すクラス Comp を定義し、動作を確認せよ。これを用いて何らかの役に立つ計算をしてみられるとなおよい。[8]

演習 5 クラス定義を活用した「面白い」Ruby プログラムを作って動かせ。面白さの定義は各自に任されるものとする。

本日の課題 9A

「演習 1」〜「演習 2」で動かしたプログラム (どれか 1 つでよい) を含むレポートを提出しなさい。プログラムと、簡単な説明が含まれること。アンケートの回答もおこなうこと。

Q1. クラスの概念やその機能について納得しましたか。

Q2. オブジェクト指向というものにどのような感想を持ちましたか。

Q3. 本日の全体的な感想と今後の要望をお書きください。

次回までの課題 9B

「演習 2」〜「演習 5」の (小) 課題から選択して 1 つ以上プログラムを作り、レポートを提出しなさい。プログラムと、課題に対する報告・考察 (やってみた結果・そこから分かったことの記述) が含まれること。アンケートの回答もおこなうこと。

Q1. クラス定義が書けるようになりましたか。

Q2. オブジェクト指向について納得しましたか。

Q3. 課題に対する感想と今後の要望をお書きください。

[8] 名前を Complex にしたくなるかも知れませんが、標準ライブラリの名前と衝突するのでやめておきましょう。