情報科学 2011　久野クラス　＃ 12

久野　靖*

2012.1.20

ここまではプログラミング言語として Ruby を使用してきましたが、予告したように、今回は「さまざまなプログラミング言語」を取り上げ、また Ruby 以外の言語として Java を使ってみて頂きます。1 つ言語を学ぶと、別の言語はそれほど苦労なく学べることがお分かり頂けると思います。

1　前回の演習問題解説

最大増加部分列

この問題では元の整数列 a と同じ長さの配列 l を用意し、各 l[i] を「i 番目の値が最後にある増加列の最大の長さ」で埋めていきます。その時まず、どの数もその数単独で長さ 1 の列になりますから、l[i] の初期値は 1 です。次に、j < i なるすべての j のうち、a[i] > a[j] であるものについては、j 番目の数が最後の列の後ろに i 番目の数を追加することができますから、それによって長さ l[j] + 1 の列ができます。そのような長さのうち最大のものが l[i] の値ということになるわけです。これを Ruby コードにしたものを次に示します (トレースバックは配列 t に記録しています。showlis は後ろから求まるトレースバックを前から順に表示するためだけの再帰手続きです):

```
def lis(a)
  l = Array.new(a.length); t = Array.new(a.length)
  m = 1; p = 0
  a.each_index do |i|
    l[i] = 1; t[i] = i+1
    (i-1).step(0, -1) do |j|
      if a[i] > a[j] && l[i] < l[j]+1
        l[i] = l[j]+1; t[i] = i-j
        if m < l[i] then p = i; m = l[i] end
      end
    end
  end
  showlis(a, t, p); puts
end
def showlis(a, t, p)
  if p > 0 then showlis(a, t, p-t[p]) end
  print(" #{a[p]}")
end
```

この問題は動的計画法を使わないと、ありとあらゆる部分列を生成してみてその長さをチェックするようなことになるので、長さ L に対して $O(2^L)$ と大変計算量が大きく なってしまいます。動的計画法を使えば $O(L^2)$ と実用的な計算量 になります。

動的計画法と通常の探索の比較

動的計画法を使わない編集距離の計算方法は、あてはまりを調べる探索問題になります。ここでは再帰手続きを使ってすべての場合を試していく方法を示します。ここに示す editdist1 は「文字列 s の i 文字目までと t の j 文字目までは

*筑波大学大学院経営システム科学専攻

対応させ終わったとして、残りの部分の編集距離を計算する」メソッドになっています。

```
def editdist1(s, i, t, j)
  if i >= s.length &&  j >= t.length
    return 0
  elsif i >= s.length
    return t.length - j
  elsif j >= t.length
    return s.length - i
  else
    x = editdist1(s, i+1, t, j+1)
    if s[i] != t[j] then x += 1 end
    y = editdist1(s, i, t, j+1) + 1
    z = editdist1(s, i+1, t, j) + 1
    return [x, y, z].min
  end
end
```

単純な場合として、両方とも残りの部分がない場合はそのままで対応しているので編集距離は 0、片方が N 文字残っている場合はその文字を削除しないと対応が取れないので編集距離は N となります。

それ以外の場合は、(1)s と t とを対応させる場合、(2)t を 1 文字進める (削除する) 場合、(3)s を 1 文字進める (削除する) 場合、の 3 とおりがありますから、それぞれの場合について残り部分について自分自身を再帰呼び出しして編集距離を求めます。そして (2) と (3) の場合はそれに 1 を加え、(1) の場合は対応する文字が等しくない場合だけ (置換になるので)1 を加えます。最後にこれら 3 つの値のうちから最も小さい場合を選んで返せばよいわけです。動かしてみましょう:

```
irb> editdist1 'sassa', 0, 'wakasa', 0
=> 3
```

確かに計算できています。では時間計測してみましょう:

```
irb> bench(1) do editdist1('aaaaaa', 0, 'aaaaaa', 0) end
=> 0.0625
irb> bench(1) do editdist1('aaaaaaa', 0, 'aaaaaaa', 0) end
=> 0.296875
irb> bench(1) do editdist1('aaaaaaaa', 0, 'aaaaaaaa', 0) end
=> 1.625
irb> bench(1) do editdist1('aaaaaaaaa', 0, 'aaaaaaaaa', 0) end
=> 8.9921875
```

文字列が 1 文字長くなるごとに時間が約 5 倍になっています。コードを再度検討すると、このメソッドは 1 文字進むについて自分を 3 回呼び出すため、時間計算量は文字列の長さ N に対して指数時間 $O(C^N)$ になる 計画法ではどうでしょう (実行回数が 1 回では速すぎて測れないので、1000 回実行の時間を計測しました):

```
irb> bench(1000) do editdist('aaaaaa', 'aaaaaa') end
=> 0.3515625
irb> bench(1000) do editdist('aaaaaaa', 'aaaaaaa') end
=> 0.46875
irb> bench(1000) do editdist('aaaaaaaa', 'aaaaaaaa') end
=> 0.6015625
irb> bench(1000) do editdist('aaaaaaaaa', 'aaaaaaaaa') end
=> 0.75
```

コードを見ても分かるように、文字列の長さ M と N に対して $M \times N$ の配列を埋めるだけですから、時間計算量は $O(MN)$ で あり、こちらの方がずっと高速だと分かります。

ビタビアルゴリズムによる復号

簡単のため「-」の代わりに「2」を使うこととして、まずマルコフモデルを表すデータを用意します:

```
$ns = 3
$init = [0.0, 0.0, 1.0]
$trans = [[0.9, 0.0, 0.1], [0.0, 0.9, 0.1], [0.05, 0.05, 0.9]]
$out = [0.9, 0.05, 0.05], [0.05, 0.9, 0.05], [0.05, 0.05, 0.9]
```

出力確率$out はここでの計算では使いませんが一応用意してあります。

次にデータを作成するため、まず 0/1 から成る文字列に 2 を挿入して指定回数 n だけ反復させるメソッドを用意しました: [1]

```
def encode(s, n)
  out = '2'*n
  s.length.times do |i| out += s[i..i]*n + '2'*n end
  return out
end
```

動かしたようすは次のとおりです:

```
irb> encode('101011', 4)
=> "22221111222200002222111122220000222211112222111122222"
```

これに指定した確率でノイズを混ぜるメソッドを用意しました:

```
def noise(s, r)
  out = ''
  s.length.times do |i|
    ch = s[i..i]
    if rand() < r then j = rand(3); ch = '012'[j..j] end
    out += ch
  end
  return out
end
```

動かしたようすは次のとおりです:

```
irb> noise(encode('101011', 4), 0.1)
=> "22221111222200002202111122020001222211112121211122222"
```

これをビタビアルゴリズムで復号してみます:

```
irb> viterbi  "22221111222200002202111122020001222211112121211122222"
=> "22221111222200002222111122220001222211112221111122222"
```

ほとんど復元できていますが、ちょっとだけノイズの影響があるようです。そこで長さ 1 の列を削除した後、連続した 0 や 1 の列ごとに 0 や 1 を 1 回出力するメソッドを用意します:

```
$atm = [{'0'=>1, '1'=>5, '2'=>0},
  {'0'=>-2, '1'=>5, '2'=>0}, {'0'=>2, '1'=>3, '2'=>4},
  {'0'=>2, '1'=>-6, '2'=>0}, {'0'=>2, '1'=>5, '2'=>0},
  {'0'=>1, '1'=>-6, '2'=>0}, {'0'=>7, '1'=>6, '2'=>8},
  {'0'=>-2, '1'=>6, '2'=>0}, {'0'=>1, '1'=>6, '2'=>0}]
def filter(s)
  cur = 0; out = ''
```

[1] Ruby では「文字列*整数」で文字列を指定数だけ反復した文字列が作れます。

```
    s.length.times do |i|
      k = $atm[cur][s[i..i]];
      if k < 0 then out += s[i..i]; cur = -k else cur = k end
    end
    return out
  end
```

このメソッドは図 1 に示す有限状態オートマトンをたどることで機能を実現します。具体的には、「連続する 0 や 1 の開始」は太線の矢印に対応しているので、そこをたどる時だけ文字を出力に追加します (データ上では太線の矢印は本来の行き先にマイナス符号をつけて表しています)。このように有限状態オートマトンを用いて入力から出力を生成するものを**有限状態変換器** (finite state transducer) と呼びます。

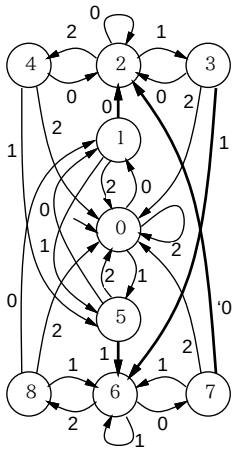


図 1: 有限状態変換器

ではこれを使って長さ 1 の列を削除してから連続する 0、1 を取り出します:

```
irb> filter "22221111222200002222111122220001222211112221111122222"
=> "101011"
```

確かに復元できました。どれくらいノイズがあっても大丈夫でしょうか:

```
irb> filter(noise(encode('101011', 4), 0.2))
=> "101011"
irb> filter(noise(encode('101011', 4), 0.3))
=> "10111"
```

さすがに 30%のノイズは無理のようですが、結構大丈夫なものですね。

2 さまざまなプログラミング言語

2.1 機械語、アセンブリ言語、高水準言語

ここまででずいぶん沢山のプログラムを Ruby で書いて来ましたが、コンピュータの内部でプログラムが実行されるしくみについてはあまり立ち入って来ませんでした。この点を少し詳しく見てみましょう (図 2)。

コンピュータの中心部分には**中央処理装置** (CPU — central processing unit) があり、その中には演算を行う**演算器** (ALU — arithmetic logc unit)、演算対象となるデータを短期的に保持する**レジスタ** (register)、処理全体を制御する**制御ロジック** (control logic) が入っています。しかし具体的な動作そのものは CPU とは別の**主記憶**に格納されたプログラムに記述されているわけです。

命令には「メモリからレジスタにデータを転送する」「レジスタからメモリにデータを転送する」「どのレジスタとどのレジスタの値にどの演算を行い結果をどのレジスタに入れる」「次の命令の番地を指定する (ジャンプ)」などのものがあります。

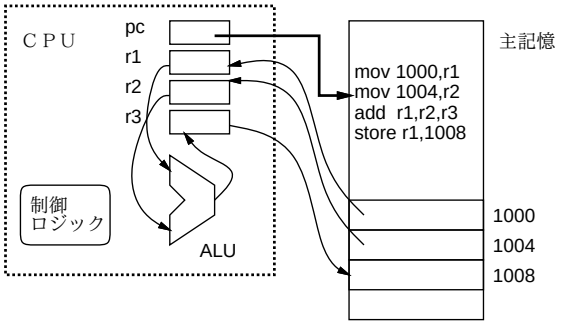


図 2: CPU と機械語プログラム

CPU 内部には**プログラムカウンタ** (program counter — pc) と呼ばれる特別なレジスタがあり、そこに「次に実行する命令の番地」が入っています。命令を取り出すごとに pc は 1 命令ずつ先に進むので、CPU はメモリ上の命令を順番に実行して行くことになります。

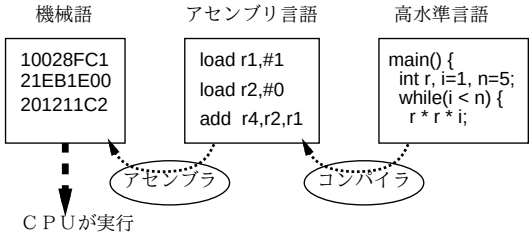


図 3: 機械語、アセンブリ言語、高水準言語

この、メモリ上に格納された命令の列のことを、「コンピュータという機械が直接実行できる形のプログラム」という意味で**機械語のプログラム** (machine language program) と呼びます (図 3 左)。機械語のプログラムは 0/1 の列なので大変読みづらく扱いづらいものです。実際、初期のコンピュータ (1945 年頃) では機械語で直接プログラムを開発していたので、短いプログラムでも大変な手間が掛かりました。

そこで、1940 年代末になると、命令やレジスタやメモリ上の番地に名前をつけてその名前を使って命令を書き表すようになりました。この記法のことを**アセンブリ言語** (assembly language) と呼びます (図 3 中)。図 2 では分かりやすくするため、メモリ上にアセンブリ言語のプログラムが書かれていますが、実際には他のデータと同じ 0/1 の列で表された機械語プログラムが格納されているわけです。

アセンブリ言語のプログラムは、**アセンブラ** (assembler) と呼ばれる言語処理系によって機械語に変換されてメモリに格納され、CPU はこれまでどおり機械語を実行します。

しかしアセンブリ言語でも、CPU に備わっている個々の命令 (加算命令、転送命令、ジャンプ命令など) を連ねてプログラムを書くのは非常に大変です。また、機械語やアセンブリ言語では、CPU の種類が違うと命令も違うので、機種を入れ替えるとプログラムが作り直しになるという問題もありました。このような、CPU の種類に依存するプログラミング言語のことを**低水準言語** (low level language) と呼びます。

このため、1950 年代以降になると、もっと人間に分かりやすい、そして特定の機種に依存しない書き方 (言語) でプログラムを書くようになりました。より正確に言えば、1950 年代になるとそのような言語の処理系が作れるくらい技術が発達してきた、ということです。

このようなプログラミング言語のことを**高水準言語** (high level language) と呼びます (図 3 右)。高水準言語は、**コンパイラ**と呼ばれるプログラムによって (場合によってはアセンブリ言語を経由して) 機械語に変換して実行するか、または**インタプリタ**と呼ばれるプログラムで直接解釈実行します。インタプリタの動作原理については、第 9 章で学びましたね。

簡単な例として、階乗を計算する C 言語プログラムを取り上げます (入出力のところはさておき、階乗の計算を行うループのところは Ruby と同じなので読めるはずです):

```
#include <stdio.h>
main() {
  int n, i = 1, r = 1;
```

```
  printf("? "); scanf("%d", &n);
  while(i <= n) {
    r = r * i; i = i + 1;
  }
  printf("%d\n", r);
}
```

これを筆者の手元のマシン (CPU は Intel 社製 32 ビット) でアセンブリ言語に直してみた結果の抜粋を示します:

```
    movl    $1, %ebx        ← i = 1
    movl    $1, %esi        ← r = 1
    pushl   $.LC0           ←パラメタ積む
    call    printf          ← printf を呼ぶ
    addl    $8, %esp
    leal    -12(%ebp), %eax ←&n の計算
    pushl   %eax            ←パラメタ積む
    pushl   $.LC1           ←パラメタ積む
    call    scanf           ← scanf を呼ぶ
    addl    $16, %esp
    cmpl    -12(%ebp), %esi ← n  <= r ?
    jg      .L6             ← > なら.L6 へ
    .p2align 2,,3
.L4:
    imull   %ebx, %esi      ← r = r * i
    incl    %ebx            ← ++i
    cmpl    -12(%ebp), %ebx ← n <= i?
    jle     .L4             ← <= なら.L4 へ
.L6:
    subl    $8, %esp
    pushl   %esi            ←パラメタ積む
    pushl   $.LC2           ←パラメタ積む
    call    printf          ← printf 実行
```

確かに高水準言語の動作に対応した命令列が生成されていることが分かります。しかしアセンブリ言語は慣れないと読むのが大変であることも確かです。

2.2 プログラミング言語の歴史

最初に作られた高水準言語は数値計算向けの言語 FORTRAN でしたが、その成功により、さまざまな言語が作られるようになりました。その流れを年代別に簡単に見てみましょう。

1950 年代

初期のプログラミング言語の時代であり、さまざまな専用目的の言語が作られました。

- **FORTRAN** — 上述のとおり、数値計算用言語。初期にはコンピュータは文字どおり「計算をする」ためのもので、そのため広く使われた。[2]
- **COBOL** — 事務処理用言語。計算を 10 進表現でやるので億でも兆でも 0.01 円でも正確に扱えるという特徴がある。[3]

[2]FORTRAN は今でも使われ改良されている言語ですが、今日では「計算をする」ことは必ずしもコンピュータの主たる用途ではなくなったため、マイナーな言語になっています。
[3]今日では COBOL で扱うような事務処理は多様なソフトウェアのごく一部となったので、これもマイナーな言語になっています。

- **Lisp** — リストと呼ばれるデータ構造と記号の扱いを中心とする。人工知能の研究に広く使われて来たが、それ以外にもこの言語が好きな人がさまざまな用途に根強く使い続けている。[4]

このほか **Algol60**、**BCPL**、**Bliss** など、多くの言語が作られました。Algol60 は FORTRAN 同様数値計算向けとされますが、さまざまなアルゴリズムの記述に多く使われました。BCPL や Bliss は**システム記述言語** (systems description languge — OS などコンピュータシステムを動かすのに必要なソフトウェアを記述するための言語) と呼ばれる種別に属します。

1960〜80 年代

新しい言語メカニズム探求の時代であり。**オブジェクト指向言語**、**論理型言語** (logic programming language)、**関数型言語** (functional programming language) など、さまざまな計算モデルとそれに基づく言語が作られました。

- **Simula**、**Smalltalk** — オブジェクト指向の黎明期の言語。
- **BASIC**、**Pascal**、**LOGO** — 教育むけに分かりやすい/使いやすい言語を作ろうとする試みの成果。
- **C** — システム記述言語であり、**Unix** オペレーティングシステムの記述に使われたことから Unix の普及とともに広がって現在に至る。
- **Prolog** — 論理型言語の最初のものであり、今でも使われている。
- **ML** — 実用に使われた最初の関数型言語で、今でも使われている。

1990 年代〜

この時期には、それ以前に考案された枠組み、とくにオブジェクト指向を土台とした「使いやすい」言語が登場し普及しました。また、コンピュータの能力が向上したことから、インタプリタ実行による柔軟性を持った**スクリプト言語** (scripting language) が多く使われるようになりました。

- **C++**、**Java**、**C#** — オブジェクト指向の汎用言語
- **Perl**、**Python**、**Ruby**、**JavaScript**、**PHP** — スクリプト言語
- **Haskell**、**OCaml** — 関数型言語の発展型

ここで挙げたのは著名なものだけでして、この他にも多数の言語が存在しています。このように言語が増えたのは、言語処理系の作り方が分かって来て誰でも簡単に作れるようになったことと、コンピュータの性能が向上したため高速で実行できるようにするための苦労をしなくても役に立つ言語処理系が作れるようになったことが大きな理由だと言えます。

2.3 プログラミング言語の種別

ここまでに見たように、プログラミング言語には多くの種別があります。ここではそれらの主要な「分類の基準」を取り上げて説明して行きます。

命令型と宣言型

命令型言語ないし**手続き型言語**は、コンピュータが行うべき動作を順に命令として記述するようなモデルの言語を言い、古くから現在に至るまで、プログラミング業界では主流を占めています。これらの言語はおなじみの「変数+代入+順次実行+制御構造」によって動作を記述します。

宣言型言語 (decralative language) とは、ものごとの性質や関係を記述し、その要件を満たすような解を処理系側で探ようなモデルの言語です。つまり、手続き型では処理の実行順序を人間が決めますが、宣言型では処理系が決めます。

これにはさらに次のような方式があります:

- 関数型 — 関数を記述し、その適用を通じて計算を行う。ML、Haskell、OCaml など。
- 論理型 — 変数を含んだ述語の群を記述し、その述語群を満たすような変数の値を求めることで計算結果が求まる。Prolog など。

[4] たとえばエディタ Emacs の中は大部分 Lisp で書かれています。

コンパイラ型とインタプリタ型

コンパイラにより機械語に翻訳する方式の言語は、実行速度が高くしやすいため、実用的なソフトウェア開発のための言語で中心的に使われて来ました (FORTRAN、C、C++などはすべてこれに当たります)。ただしコンパイラは CPU の機種が違うと (出力する機械語が異なるため) 作り直す必要があります (図 4 左)。

インタプリタにより解釈実行する方式は、実行時の柔軟性が高めやすく、教育用、記号処理、文字列処理など特殊目的言語で使われて来ました。BASIC、LOGO、Lisp ながこれに当たります。最近になって、CPU の実行速度が向上したので、実用的な言語でもインタプリタが採用されるようになっています。

インタプリタは普通高水準言語で書いてあるので、CPU が違っても単にその CPU 用にコンパイルし直せば済むのが普通です (図 4 中)。

また、1 つの言語について、コンパイラもインタプリタも存在するという場合もあります。だからこの区分は言語の特性というより、処理系の作り方ということになります。とはいえ、「この言語は主にこちら」という傾向はあります。

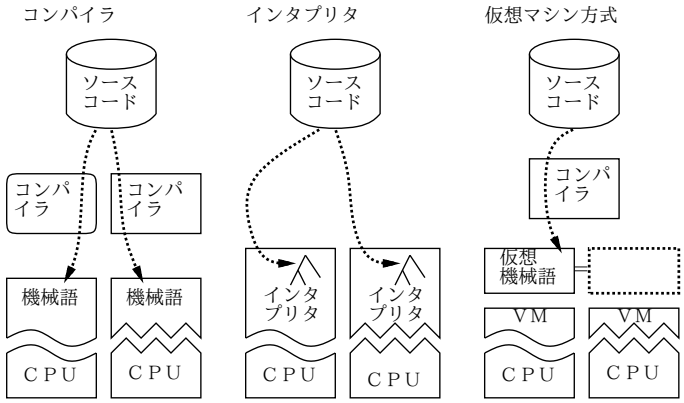


図 4: コンパイラとインタプリタ

さらに、両者を組み合わせた**仮想マシン** (virtual machine — VM) 方式と呼ばれるものもあります (図 4 右)。これは、コンパイルした結果が特定 CPU の機械語ではなく**仮想機械語** (virtual machine language) であり、それを仮想マシンと呼ばれる一種のインタプリタで実行するものです。この方式は、コンパイラと仮想マシンというインタプリタを組み合わせることから、**コンパイラインタプリタ**方式に分類できます。

仮想機械語は通常の機械語のように 0/1 の列で表現されますが、さまざまな CPU 上で仮想マシンが効率よく解釈実行できるように設計されています。

この方式はコンパイラを CPU に合わせて書き直さなくて済むわりに速度が向上させやすいという利点があります。近年では仮想マシンの実装技術が向上したため、機械語を出力するコンパイラにさほど遜色ない速度で実行できる場合も増えて来ました。Java と C#が代表的です。Ruby も 1.9.x から内部実行が仮想マシン方式になりました。

オブジェクト指向言語

既に学んで来たとおり、オブジェクト指向とは「もの」を中心としてプログラムを構成するという考え方で、それをサポートする言語機構を持った言語がオブジェクト指向言語です。クラス、メソッド、インスタンス変数などの概念を持つのが普通ですが、クラスを持たない方式もあります。

多様なライブラリ部品を用意する場合、それらをオブジェクト (ないしクラス) として用意するのが自然なので、今日の言語は多くがオブジェクト指向機能を提供するようになっています。

スクリプト言語

スクリプト (script) とは「台本」つまり「やりたいことを書いたもの」という意味です。これはもともとは、Unix の**シェルスクリプト** (shell script) がはじまりです。シェルスクリプトとは、コマンド (シェルコマンド) をファイルに入れておいて、そのファイルを実行すると書いてあるコマンドが順番に実行される、という仕組みで、これでそれなりにプログラムが書けます。その後、Larry Wall による Perl 言語が普及し、スクリプト言語という分野が認知されました。

Perl にはさまざまなシステム機能やファイル処理が自在に呼び出せ、またパターンマッチなどの機能が充実していてやりたいことが簡潔に書けるという利点があり、これにより支持者を増やしました。シェルスクリプトと Perl の間に **AWK** という言語があり、なお、Unix に標準で搭載されています。これもスクリプト言語ですが、プログラムの構造が独特で Perl ほど自由に書ける言語ではなかったため、Perl が普及しはじめるとあまり顧みられなくなってしまいした。

Perl 自体はオブジェクト指向の実現において弱点があったことから、これを改良した新世代のスクリプト言語として Python、Ruby が普及しています。このほか、ブラウザに搭載されるスクリプト言語として JavaScript、Web サーバのサーバ側プログラミングに使われるスクリプト言語として PHP など、各種用途ごとにそれむけの言語が作られるという面もあります。これらの言語を総称して**軽量言語** (light language) と呼ぶこともあります。つまり「取り回しが楽で、軽く書ける」言語、という意味ですね。

2.4 プログラミング言語と型

プログラミング言語において、**型** (type) とは、データの種別のことを言います。たとえば、同じ「+」であっても、CPU の命令では整数の足し算と浮動小数点の足し算は違う命令になりますし、整数は 32 ビット、浮動小数点は 64 ビットで扱うことが多いので確保する領域も違います。つまり、型によって扱いを変える必要があるわけです。また、オブジェクト指向言語では、「オブジェクトの種類=所属するクラス」なので、クラスと型が対応していると考えることができます。

プログラミング言語における型の扱いとして、次の 2 種類があります:

- **強い型** (strongly typed) ないし**静的な型** (statically typed) — 変数、式、メソッドの引数/返値などにおいて、すべて型が決まっているような言語
- **弱い型** (weakely typed) ないし**動的な型** (dynamically typed) — 変数や式やメソッドの引数・返値などにおいて、型が決まっていなくて、実行時にどのような型でも受け入れられる言語

strong/weak という用語は、**型安全** (type safe) である/ないという意味で用いることもあります。型安全とは、ある型の値に別の型のための操作を適用することがないと保証されていることで、これによってプログラムが予想外の動作をしたり、悪意あるデータによって動作を変更させられたりすることを防げます。ただし、OS など記述対象のソフトウェアによっては、型安全な言語だけでは記述できない部分があることもあります。

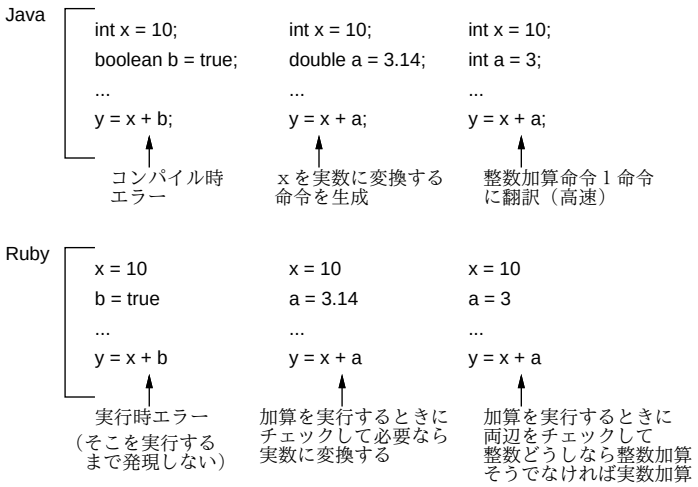


図 5: 強い型と弱い型

弱い型の言語では変数などに型を指定しませんが、実行時にはデータ本体に「これはこういう型」という情報が付属していて、それに基づいて適切な演算や処理を行っています。弱い型の言語では、どの変数にどんな種類のデータを入れてもよいのは便利ではありますが、その代わりに種類の間違ったものを入れると実行時にエラーになり、それを解決するのに苦労することがあります。さらに、複雑なプログラムでは全部の箇所を実行してみるのは難しいので、そのようなエラーがいつまでも発見されないで残ってしまうという問題もあります。また、実行時に型情報をチェックするため実行速度が遅くなる傾向にあります (図 5 下)。

強い型の言語では、すべての変数やメソッドの引数・返値について「これはこの型」ということを予め書く必要があります (ただし、ML、Haskell、OCaml などの言語では**型推論** (type inference) という機能により式の型を推定してくれるので、明示的に型を書く箇所はかなり少なくできます)。

これは慣れないと煩わしいのですが、反面、型が間違っているという状況はコンパイル時に検出されるので、実行している途中で型のエラーが出ることはなくなります。[5]また、弱い型の言語と異なり、実行時に型の情報を保持していてチェックしつつ実行する必要がなくなるため、実行が高速にしやすいという利点もあります (図 5 上)。

このため、大規模なソフトウェアを整合性を持って組み立てるなどの用途には強い型の言語 (C++、Java) が主に使われます。とはいえ、ソフトウェア開発者の好みによっては大規模なソフトウェアでも弱い型の言語を採用する場合も少なからず存在します。

3 Java 言語入門

3.1 Java のプログラムを動かす

では、ここまでに学んで来た Ruby とは別の、強い型の言語を体験していただくということで、以下では Java 言語を学んで行きます。Ruby のような弱い型の言語と何が違って何は違わないかを、自分で感じとってください。

とりあえず、最初の Java のプログラムを示します。これは、窓を作り出してその窓に「円」を 1 つ描くだけのものです。

```
import java.awt.*;       // おまじない、ライブラリ
import javax.swing.*;    // のクラスを使用するのに必要

public class Sample21 extends JPanel {      // クラス定義 (パネルのサブクラス)
  public void paintComponent(Graphics g) {  // 窓の中身を描画
    g.setColor(new Color(255, 180, 99));    // ペンの色を設定
    g.fillOval(100, 50, 100, 100);          // 指定位置に楕円を塗る
  }
  public static void main(String[] args) {  // ここから実行開始
    JFrame app = new JFrame();              // 窓を作る
    app.add(new Sample21());                // パネルをはめる
    app.setSize(400, 300);                  // サイズ設定
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  // 閉じたら終了
    app.setVisible(true);                   // 見えるようにする
  }
}
```

以下当分の間、`main()` の部分は「窓を作り、描画するクラスをはめこみ、表示」という同じ動作であり変更しません (例題ごとにクラス名は変えますので、そこだけ変更してください)。ですから、本当にやっていることは次の 3 点だけです。

- `Sample21` というクラスを、`JPanel`(窓の中にはめ込める領域) の一種 (サブクラス) として作る。なお、Java ではオブジェクトの生成は「`new` クラス名 `(…)`」という形で指定し (Ruby では「クラス名`.new`」でした)、その際呼び出される初期化メソッドのことを「コンストラクタ」と呼ぶ (Ruby では `initialize()` でした)。
- 領域の中身を描くメソッド `paintComponent()` を差し替える。
- その中で、`Graphics`(内容を描くためのメソッドを多数持っています) のメソッドを呼んで、色をオレンジぽい色に設定し、楕円 (この場合には縦横が同じなので円) を塗る。

これにより、窓の中に円が描けるのです。つまり、四角い領域というクラスのサブクラスを作り、描画部分だけ差し替えることで、自分が描きたい絵が描けるわけです。

では、これを打ち込んで動かして頂くための最低限の説明を示します:

[5]オブジェクト指向言語では動的束縛に関わる部分は実行時の処理になるので、そこに型のエラーが残ることはあります。

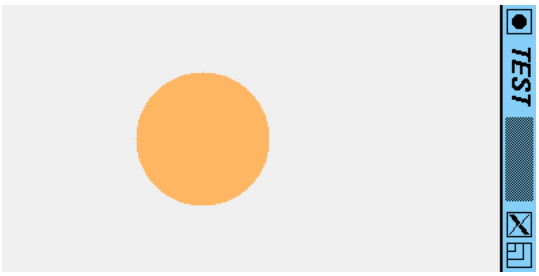


図 6: 円を描く

- Java のプログラムはすべてクラスが単位となる。そして、ファイル名と最外側の (public な) クラス名を一致させないといけない。たとえば「public class Sample21 ...」なプログラムであれば、か･な･ら･ず「Sample21.java」というファイルに入れないと、コンパイルしてもらえません。
- 実行はクラス内の「public static void main(String[] args)」というメソッドから開始される。この意味は「クラスの外部から呼べる、(インスタンスメソッドではなく) クラスに直接付随するメソッドで、名前は main で、パラメタとして文字列の配列 args を受け取る」ということです。
- プログラムを作ったらまずコンパイルする。強い型の言語の場合、ここで Ruby よりずっと厳しくチェックするため、さまざまなエラーが出ることがあります。そのときはプログラムを直して再度コンパイルしてください。「何も表示なしに終わった時」は OK です。

```
javac Sample21.java   ←コンパイルにはファイル名を指定
```

- エラーなしにコンパイルできたら実行に進む。

```
java Sample21         ←実行時にはクラス名を指定
```

実行させている様子を図 6 に示します。

それで、Ruby でもそうでしたが、どのオブジェクトにどんな機能があるかが分からないと様々なことができませんね。Java の場合は、とても豊富な標準ライブラリがあり、そのドキュメントも充実しています。本クラスのページから「JDK 1.6 API ドキュメント」というリンクをたどると、Java の標準クラス群の API ドキュメントが見られます (図 7)。[6]

このページをあけて、パッケージ java.awt を選び、その中のクラス Graphics を選んでください。その中にさまざまなメソッドが説明されていますが、fillOval() を見ると「指定された矩形の中の楕円を現在の色で塗りつぶします」とあります。つまりこのメソッドは、点 (x, y) を左上隅とし幅 W、高さ H の長方形を指定し、それに内接する楕円を塗りつぶすわけです。

このように、API ドキュメントで必要なクラスを調べることで、いちいち教えてもらわなくてもさまざまなクラスを活用したプログラムを作ることができるわけです。

演習 1 API ドキュメントでクラス java.awt.Graphics のところを開き、使えそうなメソッドをメモしてみよ。また、java.awt.Color のところも開き、コンストラクタとしてどのようなものがあるかも調べなさい。

演習 2 先の例題 Sample21.java をそのまま打ち込んで動かせ (クラス名とファイル名を同じにしないといけないので注意)。動いたら次のように手直ししてみよ。

a. 円の色や位置や大きさを変更してみなさい。
b. 円以外の図形を描くメソッドを使ってみなさい。多角形の場合は配列が必要なので、下にある配列の説明も読むこと。
c. 円を複数並べてみなさい。もし Ruby の times が欲しいと思ったら代わりに次のような for 文を使うとよいでしょう。

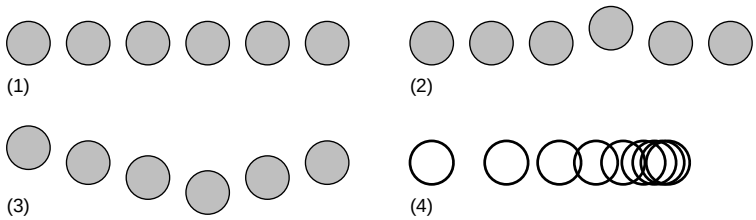
```
for(int i = 0; i < 20; ++i) {  ← 0〜19 の計数ループ
  ... ループ本体 ...
}
```

[6] API は「Application Programming Interface」つまり、プログラムを書く時に使うインタフェース、という意味です。

図 7: API ドキュメント

d. 円を下の図のように配置してみなさい。もし枝分かれが必要なら if 文を使うとよいでしょう。

```
if(条件) {
  ... 条件が成り立った場合 ...
} else {
  ... 条件が成り立なかった場合 ...
}
```



3.2 Java 言語の由来と特徴

ここで Java の歴史について、少しだけお話しておきましょう。Java 言語は、1990 年代に Sun Microsystems 社によって開発されました。当初の目的は、ネットワーク経由でコンパイル済みの仮想機械コードをさまざまなマシンに転送して実行させることができるようなシステムのための言語を開発することでした。その後 WWW が爆発的に普及した際、Sun によって**アプレット** (applet) (Web ページ内の長方形の領域内で Java のプログラムを動かし、アニメーションなどを可能にするものです) と呼ばれる技術が公開され、世の中はこれを熱狂的に歓迎して「Java ブーム」になり、Java はまたたく間にメジャーな言語となりました。当時は HTML による「止まった」ページしか作れなかった (まだ Flash も JavaScript もなかった) ので、内容が動くページには大きなインパクトがありました。Java 自体も (当時普及していた)C++よりも「安全で」「簡潔で」「スマートな」設計になっていた、ということも理由と言えるでしょう。

今日ではアプレットについては Flash などに駆逐されてしまっていますが、Java は「汎用のオブジェクト指向言語で」「仮想機械方式によりさまざまなプラットフォーム上で動作する」という特徴により、一定の支持を得てソフトウェア開発に広く使われているわけです。

3.3 Java 言語のデータモデル

Java の構文は C や C++に似せてありますが、そのデータモデルはどちらとも違います。Java では、データは次の 2 種類に明確に区分されています:

- **値** (value) — 数値などの基本型のデータ。整数 (`int`、`byte`、`short`、`long`)、実数 (`double`、`float`)、文字 (`char`)、論理値 (`boolean`) の 8 種類だけがある。加減乗除、大小比較などの演算はすべて値に対してだけ行える。
- オブジェクト — 基本型以外のあらゆるデータはオブジェクトであり、クラスによって定義されている。オブジェクトの値に対しては演算は行えず、すべてメソッド呼び出しによって操作する。例外として、`==`/`!=`により等しいオブジェクトか否かを調べることができます。また、文字列に限って、`+`により連結することができます。

このように画然と分かれていると不便なこともあります。たとえば整数をオブジェクトとして渡そうとするとそのままではできません。そこで、`int` に対して `Integer`、`double` に対して `Double` のような対応関係にあるクラスが用意されています。これを**包囲クラス** (wrapper class) と呼びます。そして、値と対応する包囲クラスのインスタンスとの間での変換は自動的に行ってくれるようになっています。

静的な型の言語では「この値の型は何か」を常に意識する必要がありますが、Java ではその型が基本型かオブジェクトかも注意する必要があります。

3.4 Java 言語の制御構造

Java の制御構造は C や C++から引き継いだ伝統的なもので、Ruby よりは「古典的」だと言えます。簡単に説明しておきましょう:

```
while(条件) {  ← while 文
  文…
}

if(条件) {     ← then だけの if 文
  文…
}

if(条件) {     ← 2 方向の if 文
  文…
} else {
  文…
}

if(条件) {     ← if-else if の連鎖 (Ruby の elsif)
  文…
} else if(条件) {
  文…
} else if(条件) {
  文…
} else {
  文…
}

for(初期化; 条件; ステップ) {
  文…
}
```

while、if、if-else if の連鎖は Ruby と書き方が違うだけです。最後の **for 文**は Ruby と全く違うので説明しましょう。Java の for 文はまず「初期化」を実行し、それから「条件」が成り立つ間繰り返し while 文と同様に反復しますが、その際、1 回本体を実行するごとに「ステップ」を実行します。たとえば、0 から 100 まで出力するというのは次のようになります:

```
for(int i = 0; i <= 100; ++i) { System.out.println(i); }
```

++i というのは変数 i を 1 増やす**インクリメント演算子** (increment operator) です。1 減らす**デクリメント演算子** (decrement operator) の--も同様に使えます。

なお、for 分にはもう 1 つの形があり、これは Ruby の配列などの each に相当します。

```
for(変数宣言 : 式) {
  文…
}
```

「式」は配列などのオブジェクトを結果としている必要があります。たとえば a が整数の配列だとすると、その要素を全部打ち出すのは次のようになります。

```
for(int i: a) { System.out.println(i); }
```

3.5 Java 言語の配列

Java の配列は、次の点が Ruby とは大きく異なっています:

- 強い型の言語なので、「型名 []」という書き方で「何型の配列か」を必ず指定する。たとえば「int[] a;」ならば変数 a は整数の配列ですし、「double[][] m」ならば m は実数の配列の配列になります。
- 配列の大きさは最初に作ったときに決まり、後から変更できない。これはすごく不便ですが、実行時の効率は優れています。一方で便利さのため、大きさが変化する配列を実現するオブジェクトが別にあり、そちらを使うことも多いですが、長くなるので説明しません。

配列 a に対して a.length で大きさ (要素数) が参照できるのは Ruby と同じです。

プログラム中で実際に配列を作る時、大きさを指定して作る方法と中身を指定して作る方法の 2 種類があるのは Ruby と同様です:

```
int[] a = new int[100];  ← 100 要素の整数の配列
double[][] m = new double[]{{1.0,0.0},{0.0,1.0}}; ← 2 次元配列
```

大きさだけ指定する場合、配列要素の初期値は数値型なら 0、論理型なら false、その他のオブジェクト型なら nil に固定されています。

上の例では配列を変数に入れていましたが、Ruby と同様、値として直接パラメタに渡したりしても構いません。たとえば、クラス Graphics の多角形を描くメソッド fillPolygon() は X 座標と Y 座標をそれぞれ 1 つの配列として渡します。そこで、次のように配列を直接渡して三角形を描かせることができます:

```
g.fillPolygon(new int[]{100, 150, 200}, new int[]{90, 20, 90}, 3);
```

3.6 アニメーションの原理

では絵が描けたところで、次は動かしてみましょう。アニメーション (動画) の原理はよく知られている通り、「人間に少しずつ違った絵を短い時間間隔で次々に見せると動いて見える」というものです。#4 では「沢山の絵を生成しておいて後で連続して見る」方法を紹介しましたが (図 8 上)、コンピュータは高速なので、「短い時間間隔」ごとにその場で「少しずつ違った絵」を生成して表示することもできます (図 8 下)。これを「実時間アニメーション」と呼びます。

具体的には、実時間アニメーションを行うコードは次のような形になります。

```
while(アニメーションの間) {
  短い時間だけ待つ
  現在の「時刻」に対応する絵を表示
}
```

簡単そうに見えますね? しかし、メインプログラムでこの動作を直接記述することは望ましくありません。というのは、メインプログラムがアニメーションに専念してしまうと、ユーザからの入力イベントに応答するなどの動作が行われなくなってしまうためです。

このため、通常の処理とは「別に」一定の時間間隔で実行する動作を「登録」するようにします。それには具体的には次のようにします。

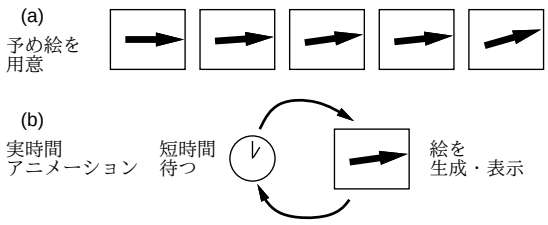


図 8: アニメーションの原理

```
new Timer(50, new ActionListener() {  // 50 ミリ秒間隔実行でタイマー生成
  public void actionPerformed(ActionEvent evt) {
    ... 一定間隔で実行させる動作本体
  }
}.start();                                  // 生成したタイマーを開始
```

このコードはタイマーオブジェクトを時間間隔 (ミリ秒単位) と「動作」を指定して生成し、ただちに実行開始させます。「動作」の書き方がややこしいですが、説明し始めると大変なので今回はこう書くものということにしておいてください。

3.7 アニメーションの例題

ではいよいよ、さっきの円を動かしてみましょう (図 8)。そのためには、先程は無かった「クラスのインスタンスを初期化するメソッド」(コンストラクタ) を作り、そこでタイマを使って定期動作を起動します。Java ではコンストラクタは、「クラス名と同名の、返す型の指定を持たないメソッド」として書きます。つまり次のような感じですね。

```
public class Sample22 {
  ... インスタンス変数定義 ...
  public Sample22() { ←コンストラクタ
    ... // 初期化の動作 (タイマー設定を含む)
  }
  public void paintComponent(...) {
    ... xpos, ypos 参照
  }
}
```

円の位置を示すインスタンス変数 xpos、ypos を追加していますが、これはスレッドの中でこれを書き換え (つまり位置を動かし)、paintComponent() の中ではそれに対応した位置に円を描くことで、時間とともに円が動くようにしているからです。では、コードを掲載します:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Sample22 extends JPanel {
  long baset = System.currentTimeMillis();
  double xpos = 100.0, ypos = 100.0;
  public Sample22() {
    setOpaque(false);
    new Timer(50, new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        double tm = 0.001*(System.currentTimeMillis()-baset);
        xpos = 100 + (tm*100) % 400;
        repaint();
```

```
      }
    }).start();
  }
  public void paintComponent(Graphics g) {
    g.setColor(new Color(255, 180, 99));
    g.fillOval((int)xpos-20, (int)ypos-20, 40, 40);
  }
  public static void main(String[] args) {
    JFrame app = new JFrame();
    app.add(new Sample22());
    app.setSize(400, 300);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
  }
}
```

メソッド System.currentTimeMillis() はある規準日時からの経過時間をミリ秒単位で返してくれます。この値を baset に保持し、後で再度読んだ時に baset を差し引くことで、プログラム実行開始からの時間を計算するようにしています。

コンストラクタの先頭で、まず「setOpaque(false);」を読んでいますが、画面が時間とともに変化するプログラムではこの設定をしておく必要があります。

その後は先に説明したタイマーの動作だけです。その先頭では上記の方法で経過秒数を変数 tm に入れます。続いて、xpos の値を時間とともに大きくなる (ただし 500 を超えると元に戻る) ように設定します。最後にある repaint() は「画面を描き直せ」という指示で、これがないと中の変数だけ変化し画面は変化しません。

このように、Java では API ドキュメントに掲載されている豊富な既存クラス群の機能を組み合わせることで、色々なことができるようになることがお分かり頂けると思います。そしてその際、型検査があるため、機能の呼び出し方が間違っていた場合も、コンパイル時にそのことを教えてもらえて、すぐ直すことができるわけです。

演習 3 Sample22.java を打ち込んで動かしなさい (Sample21.java を土台にコードを追加するのがよい)。コメントアウト部分は後から打ちこむことを薦めます。完成したら、次のような変更を試してみなさい。

a. 円の横位置も変化するようにする (どのようにするかは任意)。
b. 開始から 3 秒たったら円の色が別の色になるようにする。
c. 円の大きさも時間とともに変化するようにする。
d. 先の演習で追加した別の円や別の図形も動くようにする。
e. 動く美しい絵を表示するプログラムを作る。

3.8 マウスイベント入力とゲーム

円が動くようになりましたが、これをさらに発展させて、マウスクリックで多数の円を 1 つずつ「捕まえる」ゲームにしてみましょう。しかし先のコードの形のままで多数の円を描こうとすると大変ですね。何がいけないかというと、円がプログラム上でも「円」というオブジェクトになっていないからやりにくいのです。そこで、「円」を 1 つのクラス (今回の場合は本体プログラムの下請けとなる入れ子のクラス) にして、そのオブジェクトを多数生成する、というふうにします。円クラスから見てみましょう。

```
class Circle {
  double xpos, ypos, vx, vy, lastt = 0.0;
  boolean clicked = false;
  public Circle(double x, double y, double vx1, double vy1) {
    xpos = x; ypos = y; vx = vx1; vy = vy1;
  }
  public void draw(Graphics g) {
    g.setColor(clicked ? Color.BLUE : Color.RED);
```

```
      g.fillOval((int)xpos-20, (int)ypos-20, 40, 40);
    }
    public void setTime(double t) {
      double dt = t - lastt; lastt = t;
      xpos += vx*dt; ypos += vy*dt;
      if(xpos < 0 && vx < 0 || xpos > getWidth() && vx > 0) { vx = -vx; }
      if(ypos < 0 && vy < 0 || ypos > getHeight() && vy > 0) { vy = -vy; }
    }
    public void click(double x, double y) {
      if((xpos-x)*(xpos-x) + (ypos-y)*(ypos-y) <= 20*20) { clicked = true; }
    }
  }
```

円は等速直線運動で動き、窓の淵にぶつかるとはね返るということにするので、インスタンス変数として中心の XY 座標と、XY 方向の速度を持たせることにします。また、最後に位置を更新した時刻 `lastt` も持たせます。これらは `double` 型の変数です。あと、クリックされたかどうかを表す `boolean` 型の変数も用意します。

コンストラクタでは、XY 座標と速度を与えてインスタンス変数を初期化します。`draw()` は窓に描くための `Graphics` オブジェクトを受け取ってあるべき位置に円を描きます。円の色はクリックされたか否かで青か赤を選ぶようにしています。

`setTime()` は現在時刻を更新するものです。最後に更新した時刻 `lastt` と現在時刻の差を `dt` とし、`lastt` は現在時刻に更新します。その後、XY 座標を $V_x + dt$、$V_y + dt$ ぶんだけ増やすことで位置を更新します。そして、窓の端から出て行きそうになっていたら XY 方向の速度それぞれを反転させることで「はね返り」させます。

最後に `click()` は、マウスの XY 座標を渡されて、それがこの円内に入っていたらクリックされた状態にします。

では、本体を見てみましょう。円は `circles` という配列に入れることにして、コンストラクタで配列を用意して乱数で位置と速度を指定した円を 20 個生成し配列に格納します。

次にタイマーの定期動作ですが、これは各時刻ごとにすべての円の現在時刻を更新するだけです。

その後にあるのがマウスクリックに対応した動作の指定ですが、これも説明すると長くなるので、これでマウスクリック時の動作が指定できるのだと思っておいてください。ここでもやることは、全ての円に対してマウスの座標を指定して `click()` を呼ぶだけです。

`paintComponent()` は全ての円を `draw()` で表示させるだけ、`main()` はこれまでと同じです。

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Sample23 extends JPanel {
  long baset = System.currentTimeMillis();
  Circle[] circles;
  public Sample23() {
    setOpaque(false);
    circles = new Circle[20];
    for(int i = 0; i < 20; ++i) {
      circles[i] = new Circle(400*Math.random(), 300*Math.random(),
                        400*Math.random()-200, 400*Math.random()-200);
    }
    new Timer(50, new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        double tm = 0.001*(System.currentTimeMillis()-baset);
        for(Circle c: circles) { c.setTime(tm); }
        repaint();
      }
    }).start();
```

```
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent evt) {
        for(Circle c: circles) { c.click(evt.getX(), evt.getY()); }
      }
    });
  }
  public void paintComponent(Graphics g) {
    for(Circle c: circles) { c.draw(g); }
  }
  public static void main(String[] args) {
    JFrame app = new JFrame();
    app.add(new Sample23());
    app.setSize(400, 300);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
  }
  //ここにクラス Circle を挿入
}
```

このように、オブジェクト指向言語ではクラスをうまく分けることでプログラムの見通しがよくなる、という点は Ruby とまったく変わらないわけです。

演習 4 Sample23.java を打ち込んで動かしなさい。動いたら次のように直してみなさい。

a. 円の速度が乱数によって徐々に変化するようにする。
b. 20 秒たったら円を黒にして、黒くなったらクリックできなくする。
c. 1 回クリックするごとに赤→緑→黄色→青と変化するようにする。
d. 円の色を点滅させるようにして、色が特定の色のときだけクリックが有効になるようにする。
e. その他、やって楽しいゲームになるように工夫する。

A　本日の課題ではない 12A

課題はすべて終了しましたが、いつものように小レポートを送っていただければ拝見します。期限もとくにありません。

1. Subject: は「Report 12A」とする。
2. 学籍番号、氏名、投稿日時を書く。
3. 動かしたプログラムどれか 1 つのソース。
4. 以下のアンケートの回答。

Q1. 「さまざまなプログラミング言語」の内容についてどれくらい知っていましたか?
Q2. Java を動かして見て、Ruby との違いについてどうでしたか? Ruby で学んだことがらはどのくらいそのまま使える/使えないと感じましたか?
Q3. その他、感想、要望等どうぞ。