プログラミング基礎'00　＃3

久野　靖*

2000.6.10

0　はじめに

前回の皆様の感想もまあ「ゆとり」な人が多いようで大変結構でした。演習問題の解説は好評のようですから、今回もなるべく多く採り入れるようにします。

今回の内容ですが、最大の目標は、これまでは「既にあるクラスを使う」だけだったのに対し、いよいよ「自分独自のクラスを作る」ようになることです。といっても、簡単なクラスは書くのも簡単ですからそう心配することはありません。

それと関連して、これまで「なんとなく」済ませてきた Java プログラムの「書き方の規則」(構文) について、クラスの構造を中心に解説し、また前回も説明しましたが「クラス」と「インスタンス」の違い、「クラスメソッド」と「インスタンスメソッド」、「クラス変数」と「インスタンス変数」の違いについても説明します。

そして後半ですが、皆様絵が描けるようになったらアニメーションをやりたいでしょうから、アニメーションに不可欠な「スレッド」の説明を簡単にやって、それから前半で作った例題を拡張してアニメーションを行うように直してみます。

7 月 1 日の授業について

7 月 1 日は GSSM のオープンスクールですが、この授業は開始が 1 週間遅れたためこの週が最終回となり、実施します。ただし、オープンスクールに駆り出されている人がいるようですので、時間帯を相談しようと思います。午前とかがいいですか?

1　前回の演習問題の解説

1.1　演習 4

4a と 4b はとても簡単なので省略。ただし「色の作り方が分からない」という人がいたようですが。計算機上での色指定の標準的な方法は「赤/ 緑/ 青の光の強さを 0〜255 の数値で表す」ことで、たとえば HTML や X の各種ツールなどでもこの 0〜255 を 16 進数で表して、$(0, 32, 255)$ →「`#0020FF`」のような書き方で指定する。だから、Java でコンパイルして実行する前にどんな色か試してみたければ

```
xclock -bg '#0020FF'
```

みたいなコマンドを実行してみれば、背景が指定した色の xclock が表示される。

なお、「無限ループになった時の止め方が分からない」という人もいたが、それは「`^C`」(Control キーを押しながら「C」のキーを打つ) で止めらる。上記の `xclock` も同様。こんなの計算機科学基礎の話題だよなあ…

さて、課題 4c については例えば次のような感じ。

*筑波大学大学院経営システム科学専攻

```
import java.applet.Applet;
import java.awt.*;

public class r2ex4c extends Applet {
  Font fn = new Font("Helvetica", Font.BOLD, 36);
  public void paint(Graphics g) {
    g.setFont(fn);
    for(int i = 0; i < 10; ++i) {
      g.setColor(new Color(i*25, 100+i*10, 255-i*20));
      g.drawString("Hello, World", 30+i*3, 30+i*10);
    }
  }
}
```

ループが分かっていれば簡単!ですね。

1.2 課題 8b

8a は例題の「8」を「5」にするだけであまりに簡単なので略。8b であるが、要するに「1 つおきに長い半径と短い半径を使う」ようにすればよい。

```
import java.applet.Applet;
import java.awt.*;

public class r2ex8b extends Applet {
  int[] x = new int[100];
  int[] y = new int[100];
  public void paint(Graphics g) {
    g.setColor(new Color(100, 0, 255));
    double cx = 150;
    double cy = 100;
    double longradius = 100.0;
    double shortradius = 35.0;
    int num = 5;
    for(int i = 0; i < 2*num; ++i) {
      double rad = (2.0 * Math.PI) * i / (2*num);
      if(i%2 != 0) {
        x[i] = (int)(cx + longradius * Math.cos(rad));
        y[i] = (int)(cy + longradius * Math.sin(rad));
      } else {
        x[i] = (int)(cx + shortradius * Math.cos(rad));
        y[i] = (int)(cy + shortradius * Math.sin(rad));
      }
    }
    g.fillPolygon(x, y, 2*num);
  }
}
```

このプログラムでは「5」とか「100.0」などと直接書く代わりにわざわざ変数に入れている。このように「パラメタを変数に入れておく」方が後で調整しやすいから。そこで num は全部 2 倍して使っているけど、だったら最初から 2 倍を入れるようにした方がよかったと思えないか? そういう考えもあるけど、「5 角星」を描くのだから「5」を入れたいと思ったわけ。

ところでついでにもう 1 つ。上のプログラムでも前回のでも、「double 型の値を int 型の値で割り算」していた。このように、複数の異なる型を演算に混ぜた場合、Java では (C や C++も同じ)「一番一般的な型」に変換して揃えてから計算してくれる。それが気持ち悪ければ「(double)(2*num)」などと書いてキャストしてもよい。

1.3 課題 8d

8c については「円と三角形を重ね合わせてハートにする」という戦略と、「きれいなハートを紙でデザインしてその輪郭をデータとして打ち込む」という戦略とがある。前者は 8d の例解が参考になる。後者はただやるだけですし…(後者の方が美しくできるとは思う)。

8d については、結構悩んだ人がいたのは意外。計算機の画面は特別なことをしない限り「最後に描いたものが見える」(オーバライト)。だって、計算機では普通の変数とかも全部そうでしょう? だから、まず赤い正方形を描き、その上に白い長方形を 2 回描くと完成。これが一番簡単。十字型を fillPolygon() で描いた人もいるが、それでも別に構わない。

```
import java.applet.Applet;
import java.awt.*;

public class r2ex8d extends Applet {
  public void paint(Graphics g) {
    g.setColor(Color.red);
    g.fillRect(70, 20, 160, 160);
    g.setColor(Color.white);
    g.fillRect(130, 40, 40, 120);
    g.fillRect(90, 80, 120, 40);
  }
}
```

ところで、この Color.red とかいうのは…? これは、Color クラスのクラス変数 (後述) であり、よく使う色は予めこのように変数として用意されている。Math.PI も同様。

2 Java の構文

プログラムの正確な書き方は、その言語の「文法」によって定められている。一応、クラスからはじめてひととおり書き方の規則を示しておこう (まだ説明していないものもある)。なお「[...]」は「あってもなくてもよい」、「|」は「または」、「…」は「並んだもの」、「,…」は「カンマで区切って並んだもの」を表す。

```
プログラム ::= 単位…
単位       ::= クラス | インタフェース
クラス     ::= [public] class クラス名 継承指定 { 変数… メソッド… 単位… }
継承指定 ::= [extends クラス指定] [implements クラス指定,…]
変数       ::= [public] [static] 型指定 変数名 [= 式];
メソッド ::= [public] [static] [型指定] メソッド名 (宣言,…) [例外] { 文… }
```

ここまでのところは「クラスの中には変数やメソッドが入る」「メソッドの中には文が入る」という構造を表している。ところで、クラスとよく似た書き方をするもう 1 つの「かたまり」としてインタフェースというのがあるので、一応その構文も載せておく (意味は後で説明)。

```
インタフェース ::= [public] interface インタフェース名 { シグニチャ… }
シグニチャ ::= [public] [static] 型指定 メソッド名 (宣言,…);
```

ここまでで出てくる宣言や型の規則は次の通り。

```
宣言      ::= 型指定 変数名
型指定    ::= 型名 | 型名 [] | void
型名      ::= クラス名 | インタフェース名
            | boolean | byte | char | int | long | float | double
例外      ::= throws クラス名 [, クラス名…]
```

さて、文はこれまでにやったように、宣言、式 (代入文も式のうち)、if、while、for などがある。まだ説明していない文もあるがそれはいずれ。

```
文      ::= 宣言 [= 式]; | 式; | if(式) 文 [else 文] | while(式) 文
          | for([宣言 =] 式; 式; 式) 文 | { 文…}
          | break; |  continue; | return [式];
```

式とは「値を計算する指定」のことで、演算子、オブジェクト生成、メソッド呼び出し、リテラルながある。

```
式      ::= 式 演算子 式 | 演算子 式 | 式 [式] | new クラス名 (式,…)
          | new 型名 [式] | new 型名 []{式,…}
          | クラス名. メソッド名 (式,…) | 式. メソッド名 (式,…)
          | リテラル | 変数名 | クラス名. 変数名 | 式. 変数名 | ( 式 )
リテラル::= 整定数 [l] | 実定数 [f] | "..." | '...'
```

整数のリテラルは「l」がついたのは long、そうでないのは int。実数のリテラルは「f」がついたのは float、そうでないのは double。なお、演算子については先に挙げた。「a = b;」の「=」はあくまでも代入演算子であることに注意。演算子の一覧も再掲しておく。

```
四則演算      +   -   *   /   %  ←剰余
ビット演算    &   |   ^   ~
シフト演算    <<  >>
論理演算      &&  ||  !
比較演算      ==  !=  <   >   <=   >=
代入演算      =   +=  -=  *=  …
増減演算      ++  --
```

3　クラス、メソッド、変数

次に、クラスやオブジェクトはどうやって使うかについて説明しておこう。まず、クラスは「オブジェクトの種類」に対応していることを上で述べた。あるクラスに属する個々のオブジェクトをそのクラスの「インスタンス」と呼ぶ。インスタンスを作るには「new クラス名 (...)」という形の式を使う (配列オブジェクトだけは例外で、「new 型名 [式]」「new 型名 []{ 式,..} 」のいずれかを使う)。

クラスのさまざまな機能を利用するには「メソッド」を呼ぶ。メソッドは次の 2 種類がある。

クラス名.メソッド名(...)　←クラスメソッド
式.メソッド名(...)　←インスタンスメソッド

「式」は何らかのインスタンスを計算するものでなければいけない。クラスメソッドとインスタンスメソッドの違いは、前者がクラスに所属していてインスタンスと関係を持たないのに対し、後者はインスタンスに所属している (たとえば「私」オブジェクトの「名前を返す」メソッドの結果は「あなた」オブジェクトの「名前を返す」メソッドの結果とは違う) ことである。

これらはいずれもクラスの内側に書くことで定義されるが、クラスメソッドの場合は static というキーワードをつけるところが違う。

```
class クラス名 ... {
  public static void method1(...) { ... }
  public void method2(...) { ... }
}
```

ここで method1 はクラスメソッド、method2 はインスタンスメソッドになる。public というのはこのメソッドがクラスの外から呼べることを指定している。void というのはこのメソッドが値を返さない (手順を実行するだけである) ことを指定している。

なお、クラスの中には変数も書けるが、これもクラス変数とインスタンス変数がある。さらに 3 番目として、メソッドの中でだけ使う局所変数と、その特別な場合として値を受け取るパラメタとがある。

```
class クラス名 ... {
  static int var1;
  int var2;
  public static void method1(int x, ...) { int y; ... }
  public void method2(int z, ...) { int t; ... }
}
```

ここで変数 var1 はクラス変数、変数 var2 はインスタンス変数、y と t は局所変数、x と z は局所変数でありかつパラメタである。

4　インタフェース

オブジェクトはメソッドを呼び出すことで使うというのは繰り返し説明したが、複数の種類のオブジェクトでも呼び出し方は共通、ということがあり得る。たとえば「犬」も「猫」も「馬」も「歩け」といえば (それぞれ歩き方は違うにせよ) 歩くとか…

このような、複数の種類 (クラス) のオブジェクトにまたがって共通する呼び出し方をまとめる機能を「インタフェース」と呼ぶ。

```
interface Animal {
  public void walk(double speed);   // 歩かせる
  public void tell(String message); // 話し掛ける
}
```

もっと具体的な例として、前回やったアプレットで使えるような「画面上の図形」を考えてみる (時間設定があるのは「時とともに形が変わる図形」を想定してあるから)。

```
interface Figure {
  public void addTime(double dt); // 時間を経過させる
  public void draw(Graphics g);   // 画面に描く
}
```

インタフェースも型なので変数を作ることができる。

```
...
Figure f = new Triangle(...);
...
f.addTime(1.00); f.draw(g);
```

なお、変数 f には「Figure の一種であるような」オブジェクトなら何でも入れられる。「一種である」というのはどうやって指定するか…それはすぐ次で。

5 クラスを作る

ではいよいよ、図形のクラスを作ろう! まずは一番簡単そうな「円」を取り上げる。

```
class Circle implements Figure {
  Color col;
  double cx, cy, rad;

  public Circle(Color c, double x, double y, double r) {
    col = c; cx = x; cy = y; rad = r;
  }
  public void addTime(double dt) {
  }
  public void draw(Graphics g) {
    g.setColor(col);
    g.fillOval((int)(cx-rad), (int)(cy-rad), (int)(rad*2), (int)(rad*2));
  }
}
```

この意味は次の通り。

(1) 先頭の行でクラス名に加えて、このクラスのインスタンスは「Figure の一種である」ことを宣言している。その場合、Figure インタフェースで規定している 2 つのメソッドは必ず用意しなければならない (それ以外のメソッドもあってよいが)。

(2) インスタンス変数として、col、cx、cy、rad を持つことを宣言している。円には色、中心の座標、半径が必要だろうから、これは理にかなっていると言える。

(3) クラスと同名で型指定を持たないメソッドは「コンストラクタ」と言い、「new Circle(...)」のように new で呼び出される。その役割は、new で指定したパラメタに応じてインスタンス変数を初期設定すること。

(4) この円は時間とともに変化しないことにしたので、メソッド addTime() では特に何もしない。

(5) メソッド draw() では Graphics オブジェクトをパラメタとして受け取り、そのメソッドを使って色を設定し、円を塗りつぶす。

ではいよいよ、アプレットにして動かそう。この場合、クラス Figure や Circle は、このアプレットのためだけに使うので、アプレットクラスの内側に入れてしまうことにする。

```
import java.applet.Applet;
import java.awt.*;

public class R3Sample1 extends Applet {
  Figure f1 = new Circle(Color.blue, 100.0, 100.0, 30.0);
  Figure f2 = new Circle(Color.red, 120.0, 80.0, 20.0);

  public void paint(Graphics g) {
    f1.addTime(10.0); f1.draw(g);
    f2.addTime(10.0); f2.draw(g);
  }

  interface Figure {
    public void addTime(double time);
    public void draw(Graphics g);
  }

  class Circle implements Figure {
    Color col;
    double cx, cy, rad;

    public Circle(Color c, double x, double y, double r) {
      col = c; cx = x; cy = y; rad = r;
    }
    public void addTime(double t) {
    }
    public void draw(Graphics g) {
      g.setColor(col);
      g.fillOval((int)(cx-rad),(int)(cy-rad),(int)(rad*2),(int)(rad*2));
    }
  }
}
```

図形をオブジェクトにすることで、クラス定義は余計に書くことになるが、その代わりアプレットの `paint()` はずっと簡単になっていることが分かると思う。なお、このアプレットでは「1 回再読み込みするごとに 10 秒くらい経っている」

演習 1 上のアプレットをそのまま打ち込んで動かせ。念のため HTML ファイルも掲載する (親切だなあ!)。

```
<html><head><title>Sample</title></head><body>
<h1>R3Sample1</h1>
<applet code="R3Sample1.class" width=300 height=200></applet>
</body></html>
```

演習 2 上のアプレットに次の図形のクラスを追加し、それぞれを 2 個ずつ表示してみよ。どの図形にはどのような情報が必要かを考え、それらをインスタンス変数として用意すること。

a. 長方形。
b. 正多角形。
c. 日の丸の旗。ヒント: 白い長方形を描き、赤い円を描く。
d. スイスの国旗。ヒント: 赤い正方形を描き、白い長方形を 2 回描く。
e. Helvatica フォント 24 ポイントで表示される文字列。

演習 3 演習 2 で作った図形クラスに「速度」の機能を入れてみよ。うまく行けば「再読み込み」ごとに図形の位置が移動するはず。ヒント: インスタンス変数として vx、vy(X 方向/Y 方向の速度) を追加し、`addTime()` の中で $x' = x + V_x \triangle t$ および $y' = y + V_y \triangle t$ により新しい位置を計算する。

6 アニメーションとスレッド

さて、せっかく図形がオブジェクトになったのだから、それぞれの図形が時間とともにそれぞれ勝手に (?) 動くように、つまりアニメーションを入れてみよう。アニメーションを作るには、よく誤解されているように「ループで時間待ちしてちょっとずつ絵を描けばよい」わけでは全然! ない。まず、ループによる時間待ちというのは CPU の速さによってまったくタイミングが変わってしまう。第 2 に、その問題がないとしても、CPU を無駄に浪費するのはよくない。第 3 に、アプレットの場合、`paint()` の中でそういうことを始めると「永遠に描画が終らない」のでブラウザが困惑する。

ではどうするかというと、ブラウザのメインの実行とは別に「タイミングを取る」処理を**並列に**実行開始させ、その処理が一定時間ごとに描画をやり直すように信号を送る、というふうにする。このような「並列に実行する処理」のことを一般にスレッドと呼ぶ。

Java ではスレッドを作るにはクラス `Thread` を使い、そのコンストラクタに「並列に実行したい動作を持つオブジェクト」を渡すようにする。この渡すオブジェクトは次のインタフェース `Runnable` の一種でなければならない。

```
public interface Runnable {
  public void run();
}
```

つまり、このインタフェースを持つオブジェクトは `run()` というメソッドを必ず持つから、その `run()` というメソッドがスレッドとして (他の処理と並行に) 実行されるというわけである。

7 アニメーションを行うアプレット

では実例として、アニメーションを行うアプレットを示そう。

```
import java.applet.Applet;
import java.awt.*;

public class R3Sample2 extends Applet {
  Figure f1 = new RotPolygon(Color.blue, 100.0, 100.0, 50.0, 5, -0.5);
  Figure f2 = new RotPolygon(Color.red, 150.0, 80.0, 70.0, 3, 0.8);
  boolean running;
  long basetime;
```

```
public void paint(Graphics g) {
  long time = System.currentTimeMillis();
  double dt = 0.001*(time-basetime);
  basetime = time;
  f1.addTime(dt); f1.draw(g);
  f2.addTime(dt); f2.draw(g);
}
public void start() {
  running = true;
  basetime = System.currentTimeMillis();
  (new Thread(new Timer())).start();
}
public void stop() {
  running = false;
}
```

まだ途中だがここまで一応説明しよう。まず、変数 running はアニメーション中 (スレッドが動いていてよい状態) なら true、そうでなければ false であるような論理型 (真偽値) の変数、basetime は最後に画面更新した時刻を覚えておくための変数である。

paint() では現在時刻をミリ秒単位で取得し、最後に更新した時刻との差を求めて、秒単位に換算し、それぞれの図形に知らせる。最後に更新した時刻は新しいものに書き換える。

start() と stop() はアプレットのフレームワークに含まれているメソッドで (前回もちょっと説明したと思うが)、アプレットが表示される時と表示されなくなる時に呼ばれる。表示される時にはアニメーションをスタートしたいので、running を true にし、基準時間を覚えて、時間管理スレッドを起動する。表示されなくなる時は running を false にするだけである。では時間管理スレッドの中身は…それは次のクラスに置かれている。

```
class Timer implements Runnable {
  public void run() {
    while(running) {
      try { Thread.sleep(100); } catch(Exception e) { }
      repaint();
    }
  }
}
```

このクラス Timer は Runnable を implements しており、その動作 run() では running が true である間、「100 ミリ秒だけ休んでから、repaint()(アプレットの画面を書き直すように信号を送るメソッド) を呼ぶ」動作を繰り返す。「try { A } catch(Exception e) { B } 」というのは始めて出て来るが、A の範囲で例外 (エラーその他を通知する信号のようなもの) が発生したらその処理として B を実行せよ、という意味で、ここでは例外が起きても単に無視するために書かれている。

ところで、変数 running やメソッド repaint() はこのクラスではなく、外側のアプレットにある変数やメソッドである。このように、クラスの中に書いたクラスでは、そのインスタンスメソッドから外側のクラスのインスタンス変数やインスタンスメソッドが参照できる。逆に言えばこのクラス Timer はアプレットクラスの外に出すことはできない (他のクラスは別にそうしても構わない)。

さて、あとはこれまで見たのと同様だが、ただし時間の経過につれて変化するような図形でないと面白くない。ここでは正多角形で、各頂点の計算の起点となる角度を時間とともに進めるようにしている。

```
  interface Figure {
    public void addTime(double time);
    public void draw(Graphics g);
  }

  class RotPolygon implements Figure {
    int[] px, py;
    Color col;
    int num;
    double cx, cy, rad, theta, vtheta;

    public RotPolygon(Color c, double x, double y, double r,
                      int n, double v) {
      col = c; num = n; cx = x; cy = y; rad = r; theta = 0.0; vtheta = v;
      px = new int[num]; py = new int[num];
    }
    public void addTime(double dt) {
      theta = theta + dt*vtheta;
    }
    public void draw(Graphics g) {
      for(int i = 0; i < num; ++i) {
        double t = theta + (2.0 * Math.PI) * i / num;
        px[i] = (int)(cx + rad * Math.cos(t));
        py[i] = (int)(cx + rad * Math.sin(t));
      }
      g.setColor(col); g.fillPolygon(px, py, num);
    }
  }
}
```

ここまでで 1 つのアプレット (だいぶ長くなりましたね!) で、これで「回転する 2 つの正多角形」が現れるようになる。

演習 4 上のアプレットをそのまま動かせ (先に打ち込んだアプレットのコードを適当に利用するとよい)。

演習 5 さらに次のような動きを持つクラスを追加して一緒に動かせ。

a. 図形の位置が円や楕円に沿って動く (または直線状に振動する)
b. 文字や図形の色が時刻とともに変化していく
c. 図形の形が時刻とともに変形していく

8 実時間シミュレーション

ところで、この「一定時間ごとに状態を更新する」方法を使って、なおかつ状態 (というのは各変数に格納されている値) を一定の法則 (例: 物理法則など) に従って更新して行くことで、その法則に従った世界の「まねごと」が計算機内部で行える。これを「シミュレーション」(simulation) と呼ぶ。

たとえば、「無重力空間を運動している円」のシミュレーションを行うアプレットの例を示そう。途中までほとんど先の例題と同じなので、全部まとめて示す。

```
import java.applet.Applet;
import java.awt.*;

public class R3Sample2 extends Applet {
  Figure f1 = new BounceCircle(Color.blue, 100.0, 100.0, 30.0, 30.0, 95.0);
  Figure f2 = new BounceCircle(Color.red, 120.0, 80.0, 20.0, -85.0, 50.0);
  boolean running;
  long basetime;

  public void paint(Graphics g) {
    long time = System.currentTimeMillis();
    double dt = 0.001*(time-basetime);
    basetime = time;
    f1.addTime(dt); f1.draw(g);
    f2.addTime(dt); f2.draw(g);
  }
  public void start() {
    running = true;
    basetime = System.currentTimeMillis();
    (new Thread(new Timer())).start();
  }
  public void stop() {
    running = false;
  }

  class Timer implements Runnable {
    public void run() {
      while(running) {
        try { Thread.sleep(100); } catch(Exception e) { }
        repaint();
      }
    }
  }

  interface Figure {
    public void addTime(double time);
    public void draw(Graphics g);
  }

  class BounceCircle implements Figure {
    double width = 300.0;
    double height = 200.0;
    Color col;
    double cx, cy, rad, vx, vy;
```

```
    public BounceCircle(Color c, double x, double y, double r,
                  double vx1, double vy1) {
      col = c; cx = x; cy = y; rad = r; vx = vx1; vy = vy1;
    }
    public void addTime(double dt) {
      cx = cx + vx*dt; cy = cy + vy*dt;
      if(cx < 0 && vx < 0.0) { vx = -vx; }
      if(cx > width && vx > 0.0) { vx = -vx; }
      if(cy < 0 && vy < 0.0) { vy = -vy; }
      if(cy > height && vy > 0.0) { vy = -vy; }
    }
    public void draw(Graphics g) {
      g.setColor(col);
      g.fillOval((int)(cx-rad),(int)(cy-rad),(int)(rad*2),(int)(rad*2));
    }
  }
}
```

先の例題と違うのは「運動する円」クラスの部分 (とそのインスタンスを生成している部分) だけ。「運動する円」は当然、X 方向/Y 方向の速度をインスタンス変数として持つ。またコンストラクタではその初期値を設定する。

`addTime()` では円の中心位置を計算し直すが、それは「$\triangle t$ 時間における変移は速度と $\triangle t$ を掛けたもの」という物理法則に従うだけ。あと、それだけでは円が運動してアプレット領域の外へ行ってしまうので、「X 座標/Y 座標ともアプレットの範囲を飛び出したら X 方向/Y 方向の速度の符合を反転する」ようにした。これで「はね返って戻って来る」ようになる。

演習 6 上のアプレットをそのまま動かせ (先に打ち込んだアプレットのコードを適当に利用するとよい)。

演習 7 上のアプレットを次のように改良せよ。`BounceCirle` クラスを直接修正してもいいし、改良版のクラスを追加してその物体を新たに加えるようにしてもよい。

a. 円が壁に食い込んで跳ね返っているのを、食い込まないで跳ね返るように直す。
b. 無重力ではなく、重力が働く (画面下方向への加速度が働く) ようにする (上方向や横方向にしてもよい)。
c. 跳ね返る時にエネルギーが失われるようにする (またはエネルギーが増加するようにする)。
d. 円どうしに引力または反発力が働くようにする。または円どうしが接触した時は「ぶつかって跳ね返る」ようにする。(注意!: これは非常に難しいので腕に覚えがある人だけにしておいてください。)

演習 8 ここまでに学んだことを利用して、何でも自分の好きなアニメーションを行なうプログラムを設計・製作せよ。