計算機プログラミングI 2005　久野クラス　＃7

久野　靖*

2005.12.2

はじめに

クラスは作れるようになりましたか。クラスを活用することで、今回いよいよアニメーションに入りますので、頑張りましょう。なお、アニメーションの例題に入るためには関連して (1) 入れ子クラス、(2) インタフェース、(3) スレッドと新しいことが 3 つも出てくるのでちょっとハードです。

1　前回の練習問題の回答例

前回の演習で自由課題でないのは演習 2 だけなのでそれをまとめて示す。

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

public class r6ex2 extends JApplet {
  Rect r1 = new Rect(Color.red, 100, 100, 80, 40);
  Rect r2 = new Rect(Color.green, 150, 120, 60, 30);
  Circle c1 = new Circle(Color.pink, 120, 80, 40);
  Triangle t1 = new Triangle(Color.yellow, 100, 180, 140, 160, 180, 180);
  Flag f1 = new Flag(Color.blue, Color.green, 200, 50, 30);
  public void paint(Graphics g) {
    Graphics2D g2 = (Graphics2D)g;
    r1.draw(g2); r2.draw(g2); c1.draw(g2); t1.draw(g2); f1.draw(g2);
  }
}

class Rect {
  Paint col;
  int xpos, ypos, width, height;
  public Rect(Paint c, int x, int y, int w, int h) {
    col = c; xpos = x; ypos = y; width = w; height = h;
  }
  public void draw(Graphics2D g) {
    g.setPaint(col);
    g.fillRect(xpos-width/2, ypos-height/2, width, height);
  }
}
class Circle {
  Paint col;
  int xpos, ypos, rad;
```

*筑波大学大学院経営システム科学専攻

```
  public Circle(Paint c, int x, int y, int r) {
    col = c; xpos = x; ypos = y; rad = r;
  }
  public void draw(Graphics2D g) {
    g.setPaint(col); g.fillOval(xpos-rad, ypos-rad, 2*rad, 2*rad);
  }
}
class Triangle {
  Paint col;
  int[] xpts, ypts;
  public Triangle(Paint c, int x0, int y0, int x1, int y1, int x2, int  y2) {
    col = c; xpts = new int[]{x0,x1,x2}; ypts = new int[]{y0,y1,y2};
  }
  public void draw(Graphics2D g) {
    g.setPaint(col); g.fillPolygon(xpts, ypts, 3);
  }
}
class Flag {
  Circle c1;
  Rect r1;
  public Flag(Paint cl1, Paint cl2, int x, int y, int r) {
    c1 = new Circle(cl1, x, y, r); r1 = new Rect(cl2, x, y, r*4, r*3);
  }
  public void draw(Graphics2D g) {
    r1.draw(g); c1.draw(g);
  }
}
```

ほとんど説明する余地もないが、三角形は `fillPolygon()` で使う配列を保持しておくのが簡単。また「旗」については全部自前でやるより、既に作った `Rect` と `Circle` を利用するのが吉。

2　入れ子クラスと内部クラス

2.1　クラスの入れ子とスコープ

前回のように複数のクラスを含んだプログラムを作っていて、心配になった人はいないだろうか? つまり、あるプログラムで A という補助クラスを使って、別のプログラムでも A という補助クラスを使ったら、両者がまざって混乱しないだろうか? 実際、そういう問題はよく起きる。それを避けるために、今回は「クラスの中にクラスを入れる」技を学ぼう。

整理すると、前回のプログラムはこういう形になっていた。

```
import ...

public class Sample6x extends JApplet {
  ...
}
class Circle {
  ...
}
```

なお「`public`」というのは「外に公開されている」という意味で、「ファイル名とクラス名は必ず一致しないといけない」という規則は `public` 指定のクラスにあてはまる。言い替えれば `public` のついていないクラスなら 1 つのファイル

に一緒にいくつでも入れられる。しかし、これらのクラスは「一緒に入れてある」だけで、どこのクラスからでも参照でき、従って混同も起こり得る (「外に公開」の事情についてはもっと後の方で説明するが、ここで一緒に入れてあるものはすべて「身内」に相当)。

これを避けるためには、次のように補助クラスをアプレットクラスの内側に入れてしまう。

```
import ...

public class Sample6x extends JApplet {
  ...
  static class Circle {  // 「static」については後述
    ...
  }
}
```

このように、あるクラスの中に別のクラスを入れたものを「入れ子クラス」(nested class) と呼ぶ。こうすれば、クラス Circle は Sample6x の中からしか参照できないので、混同も起きない。一般に、「ある単位の中に入れたものはその単位の中からしか参照できない」という規則は「スコープ規則」などと呼ばれ、プログラミング言語で多く使われる (厳密には言語によってスコープ規則の詳細はいろいろ違っている)。

実は既にスコープ規則には沢山お目に掛かっている。まず局所変数とパラメタもスコープになっている。

```
 public int compute(int x) {
   int y = ...     // x, y 参照可
   if(...) {       // x, y 参照可
     int z = ...   // x, y, z 参照可
     ...           // x, y, z 参照可
   }
   return x + y;   // x, y 参照可
}
```

この例で、x や y はメソッド compute() の中全体がスコープだが、変数 z は if 文に続くブロック ({...}) の中で宣言されているので、このブロックの内側だけがスコープになる。

次に、インスタンス変数をアプレットのメソッド内からアクセスできるのもやはりスコープ規則によっている。

```
public class Sample6x {
  double a;
  public Sample6x(double a1) {
    a = a1; // a1, a 参照可
  }
  public void add(double d) {
    a += d; // a1, d 参照可
  }
}
```

2.2 static の謎

ところで、先の内側に入れたクラス Circle に static 指定がついていたがこれは何だろう? これまで、static がついている変数やメソッドはクラスに付属する「クラス変数」「クラスメソッド」、そうでないものはインスタンス (オブジェクト) に付属する「インスタンス変数」「インスタンスメソッド」と説明してきたが、それについてまず復習しよう。たとえば、正方形 (Square) クラスを作ったとして、いくつ正方形を作ったかを数えておきたいものとする。

```
class Square {
  static int count = 0;
  double gx, gy, len;
```

```
  public Square(double x, double y, double l) {
    gx = x; gy = y; len = l; // x,y,l,gx,gy,len,count
    count += 1;              // x,y,l,gx,gy,len,count
  }
  ...
  public void moveBy(double dx, double dy) {
    gx += dx; gy += dy;     // dx,dy,gx,gy,len,count
  }
  public static int getCount() {
    return count;            // count
  }
}
```

クラス変数 count は Square クラスに付属して 1 つだけ存在し、そこに「いくつ作ったか」の数を数えておく。一方、インスタンス変数 gx、gy、len は Square オブジェクトごとに存在する (でないと個々の正方形の位置や大きさを記憶しておく役に立たない)。

コンストラクタやインスタンスメソッド moveBy() の中からは、クラス変数とインスタンス変数の両方にアクセスできる。なぜなら、コンストラクタではちょうど今そのインスタンスを作って初期設定しているところであり、インスタンスメソッドは

```
Square s1 = new Square(100, 80, 40);
...
s1.moveBy(10, 10);
```

のように必ず「インスタンス (オブジェクト) を指定して呼び出される」からである。一方、メソッド getCount() は「いくつ作ったか」を調べるメソッドであり、static のついたクラスメソッドになっている (呼ぶ時は Square.getCount() のようにクラス名をつけて呼ぶ)。クラスメソッドの中からは、インスタンス変数はアクセスできず (なぜなら「インスタンスごとに存在しているどのインスタンス変数か」が決められないから)、パラメタや局所変数以外にはクラス変数だけにアクセスできる。このように、Java では「static」というキーワードがないと「いずれかのインスタンスに対応している」、あると「特定のインスタンスに対応していない」という意味になる。

さて、ここでようやく先の例で Circle クラスについていた static の説明に戻ることができる。Circle クラスのインスタンスは単に円を表しているだけで、特定の Sample6x アプレットのインスタンスに付属しているわけではない (たとえば、1 つの画面に 2 つ Sample6x アプレットを入れて、1 つのアプレットで作った円を途中でもう 1 つのアプレットに移すようなこともできる…説明が面倒だからやらないけど)。このために、static をつけていた。

ここでもし、static をつけていない入れ子クラス (Java では「内部クラス」と呼ぶ) を作ったとすると、そのインスタンスメソッドの中からは外側クラスのインスタンス変数にアクセスすることができるし、外側クラスのインスタンスメソッドを (インスタンスをわざわざ指定せずに) 呼び出すこともできる。

```
public class Sample6x {
  Circle c1 = new Circle(...);
  ...
  class MyInnerClass {
    ...
    public void someAction(...) {
      ... // c1 参照可能
    }
  }
  static class Circle {
    ...
    public void moveTo(...) {
      ... // c1 参照不可
    }
  }
```

```
}
```

その代わり、内部クラスのインスタンスをおなじみの「new クラス名 (...)」で作れるのは、外側クラスのインスタンス変数初期設定式かコンストラクタかインスタンスメソッドの中だけに限定される (そうでないと、外側クラスの「どの」インスタンスに対応づけるべきかが分からないため)。[1]

3 インタフェースとスレッド

ここまで学んできたことで大体お分かりと思うが、オブジェクトとは要するに「いちど作ってしまえば (既に作られていて API ドキュメントに載っているものも含む)、使い方さえ合わせれば使える」ものなわけである。

上の説明では、合わせるのはあくまでも「プログラムを作る人が、ドキュメントなりを読んで合わせる」ことを想定していたが、実は Java ではさらに進んで「プログラムの中で自動的に合わせる」ことも合わせるうちに入っている。ただしそれにはプログラマがそういう風にプログラムを作ることは当然ながら必要だが。

何がなんだか分からない? ではもっと具体的に説明しよう。たとえば、今回はアニメーションを行うために、本流の実行の流れとは並行した「新しい実行の流れを作り出すオブジェクト」を生成し利用する (Thread クラスは java.lang パッケージに含まれている)。

```
Thread t1 = new Thread(...);
t1.start();  // 並列実行を開始
```

しかし、その「新しい実行の流れ」で実際に何を実行するかはどうやって指定すればいいだろう? それは「...」のところで Thread クラスのコンストラクタに渡して指定するわけだが…

実はそこのところはこうなっている。

- Thread オブジェクトを利用する側では、

  ```
  public void run() {
    ...
  }
  ```

  というメソッドを持つクラスのインスタンスを用意してそれを渡す。

- Thread オブジェクトの側では、その渡されたインスタンスのメソッド run() を並列実行する内容として呼び出す。

ただし、このようなことを実際に行う上では、Thread クラス側では「コンストラクタで受け取るのはこのような run() を持っているオブジェクトですよ」ということを明記する必要があるし、渡すオブジェクトのクラスを作る側でも「このクラスはこのような run() を持つクラスですよ」ということを明言する必要がある。これらの約束ごとを明示するための機構が「インタフェース」である。いわば、インタフェースは「コンセントの形を決める約束」であり、その約束に合わせてある器具ならば、どの電気器具でもどこのコンセントにでも指して使える、そういう感じである。

では、実際にインタフェース定義の形を見てみよう。上の例にあげたメソッド run() を持つインタフェースは Runnable という名前で、既に標準 API の java.lang パッケージに含まれているが、これを自前で定義するとしたら次のようになる。

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

今まで作って来たクラスと違うのは、(1)class の代わりに interface と書くことと、(2) メソッドの冒頭部分だけが定義されていて、メソッド本体 (コード) や変数定義がないことである。(2) については、インタフェースが「呼び出し方の約束」を定めるためのものなので、考えてみれば当然だと言える。

そして、Thread クラスの API ドキュメントを見てみると、そのコンストラクタは次のようになっている。

```
public Thread(Runnable run);
```

[1] 上記以外の場所でもインスタンスを表す式を指定して「式.new クラス名 (...)」とすれば作れるが、この書き方を使ったプログラムを久野は今だかって見たことがない。

つまりインタフェースもクラスと同様に「型」(オブジェクトの種別) を表すものであり、Runnable インタフェースに従うようなクラスのインスタンスなら**何でも**この型にあてはまるものとして扱える。

次に、自分でインタフェースに従う (Java の用語では「実装する」) クラスを作る場合には、クラスの冒頭部分で implements というキーワードに続いてインタフェース名を指定する。

```
class MyRun implements Runnable {
  ...
  public void run() {
    ... ☆
  }
  ...
}
```

そして、このクラスのインスタンスを指定して Thread オブジェクトを作ってやれば、メソッド start() を実行すると、上記☆の部分がプログラム本体の流れと並行して実行されることになる。

4 アプレットでアニメーションを行う

では準備ができたのでいよいよ、アニメーションを行うアプレットの例を示そう。40 行ほどあるが、そのうち半分は前回やったような図形のクラス (ここでは Circle である。

もう 1 つ複雑になっているのは、アニメーションの場合、「絵を描く→クリア→絵を描く→クリア→…」の繰り返しになるので、そのまま作ると画面がチラチラつくのでそれを防ぐ仕組みを入れたためである。具体的には 1 つ画像を用意し、「画像をクリア→画像に描く→画像を画面に描く→画像をクリア→画像に描く→画像を画面に描く→…」のようにする。そのため次のような変更がある。

- アプレットの変数として Image を 1 つ用意する。その名前を buf としている。
- update() というメソッドを用意し、この中で上記の作業を行う。具体的には、まず buf が空なら画面と同じ幅、高さを持つ画像を用意する。次に画像をクリアし、画像用のペンを取り出して paint() にはそれを渡して絵を描かせる。描き終わったらその画像を実際の画面に描く。

この部分は今後アニメーションを行うアプレットではすべて共通になる。ではコードを見てみる。

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

public class R7Sample1 extends JApplet {
  Image buf;
  Circle c1 = new Circle(Color.blue, 0, 100, 20);
  public void update(Graphics g) {
    if(buf == null) { buf = createImage(getWidth(), getHeight()); }
    Graphics2D g2 = (Graphics2D)buf.getGraphics();
    g2.clearRect(0, 0, getWidth(), getHeight()); paint(g2);
    g.drawImage(buf, 0, 0, this);
  }
  public void paint(Graphics g) {
    Graphics2D g2 = (Graphics2D)g; c1.draw(g2);
  }
  public void start() { (new Thread(new MyRun())).start(); }
  class MyRun implements Runnable {
    public void run() {
      for(int i = 0; i < 200; ++i) {
        try { Thread.sleep(50); } catch(Exception e) { }
        c1.moveTo(c1.getX()+1, c1.getY());
```

```
        repaint();
      }
    }
  }
  static class Circle {
    Paint col;
    double gx, gy, rad;
    public Circle(Paint c, double x, double y, double r) {
      col = c; gx = x; gy = y; rad = r;
    }
    public void draw(Graphics2D g) {
      g.setPaint(col);
      g.fillOval((int)(gx-rad),(int)(gy-rad),(int)rad*2,(int)rad*2);
    }
    public void moveTo(double x, double y) { gx = x; gy = y; }
    double getX() { return gx; }
    double getY() { return gy; }
  }
}
```

まずアプレットクラスの冒頭で、Circle のインスタンスを作り変数 c1 に入れる。paint() では円を表示させるだけ。メソッド start()(表示開始時に呼ばれる) で、スレッドを作って直ちに実行開始させる。

ちょっと順序を変えて、Circle クラスを先に説明しよう。これは前回作ったものと同様だが、座標類を double つまり小数点以下まである数値として扱うように直してあり、また現在の X/Y 座標を取得するメソッド getX()、getY()、および X/Y 座標を設定するメソッド moveTo() が用意されている (これらは円を動かすところで使う)。

さていよいよ、クラス MyRun だが、これはメソッド run() つまりスレッドにより並行実行される動作だけを定義している。その中身は次の通り。

- 200 回繰り返し実行。
- 50 ミリ秒時間待ち。
- 円 c1 をこれまでの位置より 1 ピクセル右へ動かす。
- 画面の再描画を指示。
- ここまで繰り返し。

Thread クラスのクラスメソッド sleep() は、指定した時間、実行を休止して待ち合わせる。この行にある「try...catch」については今回は説明し切れないので次回ということで。ところで、このメソッドの中から外側クラスの変数 c1 やメソッド repaint() を参照していることに注意。このため、クラス MyRun は内部クラス (static でない入れ子クラス) にしてある。

メソッド repaint() はアプレットクラスが持っているメソッドで、「画面の再描画を指示」つまりちょっと暇になったらすぐ画面を描きなおしてね、という意味。たとえば沢山の箇所で絵が変更されて何回も repaint() が呼ばれたとしても、描き直しは 1 回だけで済ませるといった感じになっている。ともかくこれで、「円が横に動いて行く」だけだがアニメーションができた。

演習 1 上の例題をそのまま打ち込んで動かせ。

演習 2 動いたら次のように変更してみよ。

a. X 軸方向だけでなく Y 軸方向にも動くようにする。

b. 右へ動いて止まったあと、続けて別の動きをする。

c. なんでもいいからランダムな動きをさせる。

d. 円を 2 つとか 3 つに増やしてそれぞれ動かす。

e. 動くだけでなく円の色を徐々に変化させる。(ヒント: クラス Circle にメソッド setPaint() を追加する必要がある。)

5 より「オブジェクト指向らしい」アニメーション

先の方法で一応アニメーションができるようになったが、ある意味ではこれはちょっと「オブジェクト指向っぽくない」と思う。というのは、メソッド run() が「神様」になってすべてのメソッドに「こう動け」とすべて指図していたから。それが駄目ならどうするかって? その説明をしよう。

基本的な方針として、「画面に描けて、動くもの」を表すインタフェースを定義し、プログラム全体をこのインタフェースを通じてものを表示させたり動かす部分と、このインタフェースを実装する各種のものとに分離する。インタフェースをとりあえず示そう。

```
interface Animation {
  public void addTime(double dt);
  public void draw(Graphics2D g);
}
```

つまり、Animation インタフェースを実装するオブジェクトは、メソッド addTime() を呼ぶことで「時間を進める」ことができ、またメソッド draw() を呼ぶことで「画面に表示」できる。

次に、このインタフェースを前提としたアプレット部分を示す。このアプレットはアニメーション実行中か否かを表す変数 go と現在の時刻を秒単位で保持する変数 time を持つ。また、Animation インタフェースを実装したオブジェクトを 20 個まで、配列 a に保持しておいて、それらを順次動かしながら画面に描くことができる (変数 count に実際の個数を保持する)。初期設定メソッド init() で、後で出て来る「飛ぶ円」2 個と「ランダムに動く正方形」を生成し、配列に入れている。paint() では配列に保持しているものを順次描く。

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

public class R7Sample2 extends JApplet {
  Image buf;
  boolean go;
  double time;
  Animation[] a = new Animation[20];
  int count = 0;
  public void init() {
    a[count++] = new FlyingCircle(Color.red, 100, 100, 40, 30, -40);
    a[count++] = new FlyingCircle(Color.blue, 100, 100, 30, 40, 50);
    a[count++] = new ResizingSquare(Color.green, 100, 100, 100);
  }
  public void update(Graphics g) {
    if(buf == null) { buf = createImage(getWidth(), getHeight()); }
    Graphics2D g2 = (Graphics2D)buf.getGraphics();
    g2.clearRect(0, 0, getWidth(), getHeight()); paint(g2);
    g.drawImage(buf, 0, 0, this);
  }
  public void paint(Graphics g) {
    Graphics2D g2 = (Graphics2D)g;
    for(int i = 0; i < count; ++i) a[i].draw(g2);
  }
  public void start() { go = true; (new Thread(new MyRun())).start(); }
  public void stop() { go = false; }
  class MyRun implements Runnable {
    public void run() {
      time = 0.001 * System.currentTimeMillis();
      while(go) {
        try { Thread.sleep(100); } catch(Exception e) { }
```

```
        double dt = 0.001 * System.currentTimeMillis() - time;
        for(int i = 0; i < count; ++i) a[i].addTime(dt);
        time += dt; repaint();
      }
    }
  }
  interface Animation {
    public void addTime(double dt);
    public void draw(Graphics2D g);
  }
  static class FlyingCircle implements Animation {
    Paint col;
    double gx, gy, rad, vx, vy;
    public FlyingCircle(Paint c, double x, double y, double r,
                        double vx1, double vy1) {
      col = c; gx = x; gy = y; rad = r; vx = vx1; vy = vy1;
    }
    public void addTime(double dt) {
      gx += vx*dt; gy += vy*dt;
      if(gx < 0 && vx < 0 || gx > 300 && vx > 0) vx = -vx;
      if(gy < 0 && vy < 0 || gy > 200 && vy > 0) vy = -vy;
    }
    public void draw(Graphics2D g) {
      g.setPaint(col);
      g.fillOval((int)(gx-rad), (int)(gy-rad), (int)rad*2, (int)rad*2);
    }
  }
  static class ResizingSquare implements Animation {
    Paint col;
    double gx, gy, len, time;
    public ResizingSquare(Paint c, double x, double y, double l) {
      col = c; gx = x; gy = y; len = l; time = 0.0;
    }
    public void addTime(double dt) { time += dt; }
    public void draw(Graphics2D g) {
      int l = (int)(2.0*len + len*Math.sin(time)) / 4;
      g.setPaint(col);
      g.fillRect((int)(gx-l), (int)(gy-l), (int)(l*2), (int)(l*2));
    }
  }
}
```

表示開始のメソッド `start()` では変数 go を `true` にしてからスレッドを生成・実行開始する。表示停止時のメソッド `stop()` では go を `false` にする。

次がスレッドで実行するメソッド `run()` を定義するための内部クラス `MyRun` で、この中の疑似コードを示しておく。

- time ← 現在時刻を秒単位で表したもの。
- 変数 go が true である間繰り返し。
- 100 ミリ秒時間待ち。
- dt ← さっきの時刻と現在の時刻の差。
- すべてのモノの時刻を dt だけ進める。
- time ← time + dt。

- 画面再描画を指示。
- ここまで繰り返し。

つまり変数 go が true である間無限に時刻を進めながら画面を再描画することを繰り返している。この先にさっきのインタフェース Animation とものに対応するクラスを定義する。

```
interface Animation {
  public void addTime(double dt);
  public void draw(Graphics2D g);
}
static class FlyingCircle implements Animation {
  static final int MAX_X = 300, MAX_Y = 200;
  Paint col;
  double gx, gy, rad, vx, vy;
  public FlyingCircle(Paint c, double x, double y, double r,
                      double vx1, double vy1) {
    col = c; gx = x; gy = y; rad = r; vx = vx1; vy = vy1;
  }
  public void addTime(double dt) {
    gx += vx*dt; gy += vy*dt;
    if(gx < 0 && vx < 0 || gx > MAX_X && vx > 0) vx = -vx;
    if(gy < 0 && vy < 0 || gy > MAX_Y && vy > 0) vy = -vy;
  }
  public void draw(Graphics2D g) {
    g.setPaint(col);
    g.fillOval((int)(gx-rad), (int)(gy-rad), (int)rad*2, (int)rad*2);
  }
}
```

FlyingCircle は速度を持っていて等速直線運動で飛び続ける円。等速直線運動の場合、ある時刻に位置が (x, y) だったとして、微小時間 dt 後の位置 (x', y') は次の式で求まる (物理でスマソ)。

$$x' = x + v_x * dt, y' = y + v_y * dt.$$

ただしこのままだとあっという間に画面から出て行ってしまうので、画面の範囲外に出そうになったら X 方向/Y 方向の速度を反転することによって「反射」して帰って来るようにしている。このため、X 座標と Y 座標の最大値を MAX_X と MAX_Y という名前で定義している。[2]これらは static がついているためこのクラスの全オブジェクトに共通で、かつ final という指定がついているのでいちど設定したら値が変更されない「定数」となる。なぜこんなものを定義しているかというと、画面の大きさを変えたくなった時にここだけ変えれば済むように。

```
static class ResizingSquare implements Animation {
  Paint col;
  double gx, gy, len, time;
  public ResizingSquare(Paint c, double x, double y, double l) {
    col = c; gx = x; gy = y; len = l; time = 0.0;
  }
  public void addTime(double dt) { time += dt; }
  public void draw(Graphics2D g) {
    int l = (int)(2.0*len + len*Math.sin(time)) / 4;
    g.setPaint(col);
    g.fillRect((int)(gx-l), (int)(gy-l), (int)(l*2), (int)(l*2));
  }
```

[2]もちろん、これを大きくしてもいよいが、HTML 側の applet タグのパラメタも対応して変更しないと無意味なので注意。なお、大した理由なしにやたら巨大なアプレットを作るのはダサいと久野は考えているのでそのつもりで。

```
    }
  }
```

ResizingSquare の方は addTime() の中では変数 time に時間を足し込んで累計している。つまりここに経過時間が入ることになる。そして、paint() の中では経過時間の sin 関数に応じて 1 辺の大きさを計算しているので、正方形が大きくなったり小さくなったりする。

このように、アプレットクラス側では「時間を進める」だけで、「時間の経過につれてどのように動くか」はそれぞれの物体側でそれぞれ計算することで、いくつどんな動きをするオブジェクトが増えてもアプレット側は複雑にならない。このような分離がきれいにできるところがオブジェクト指向の利点だと言える。

演習 3 上の例題を打ち込んでそのまま動かせ。動いたら次のような物体を表すクラスを増やしてみよ。

a. 重力に従って下 (または上または右または左) に引っ張られながら飛ぶ円 (ヒント: 上の式に加えて $v'_y = v_y + \alpha dt$ により速度を増やす。α は適当に決めてみる)。

b. 円 (楕円) 軌道に従って動く何か (形は自由)。

c. 形が変化する何か (どういう形がどう変化するかは自由)。

d. 回転する N 角形または N 角星 (ヒント: 中心の座標を (x, y) として、頂点の座標を $(x + r\cos(\theta + \frac{2\pi i}{N}), y + r\sin(\theta + \frac{2\pi i}{N}))$ ただし $i = 0, 1, ..., N-1$ により決める。θ を時刻に比例して変化させる。星の場合は頂点数を倍にして長い半径と短い半径を交互に持たせる)。

演習 4 自分が美しい/かっこいいと思うアニメーションを行うアプレットを作成せよ。

A　本日の課題 7A

「演習 2」または「演習 3」で作成したアプレットを格納した WWW ページのための HTML ファイルを、自分の cp1 ディレクトリの下に report7a.html という名前で作成すること。また、そのプログラムのコードはいつも通り、「本日中に」久野までメールで送付してください。具体的な内容は次の通り。

1. Subject: は「Report 7A」とする。
2. 学籍番号、氏名、投稿日時を書く。
3. 選んだプログラム 1 つのソース。
4. その簡単な説明。
5. 下記のアンケートの回答。

   Q1. 絵が動かせるようになりましたか?
   Q2. 入れ子クラスとインタフェースについては分かりましたか?
   Q3. その他感想、要望等あればどうぞ。

B　次回までの課題 7B

次回までの課題は当然ながら「アニメーションを行うアプレットで美しい/かっこいいと思うものを作る」演習 4 をやって頂きます。作成したアプレットのための HTML ファイルを、自分の cp1 ディレクトリの下に report7b.html という名前で作成すること。また、そのプログラムのコードはいつも通り、久野までメールで送付してください。期限は次回授業開始の 10 分前 (10:30 まで) です。具体的な内容は次の通り。

1. Subject: は「Report 7B」とする。
2. 学籍番号、氏名、投稿日時を書く。
3. 作成したプログラムのソース。
4. その説明。どのような点を工夫したか書くこと。
5. 下記のアンケートの回答。

Q1. 自分で思ったように絵が動かせるようになりましたか?

Q2. プログラムを複数のオブジェクトに分けて、クラスとして記述していくというJavaの流儀に納得が行くようになりましたか?

Q3. その他感想、要望等あればどうぞ。