情報科学特殊講義'2000 ＃6

久野　靖*

2000.1.24-27

0　はじめに

今回は前回の課題 (全部だと大変なので一部) の解説をした後、「インタフェース」「スレッド」の話をしてから、いよいよアニメーションをやります (アニメーションをやるにはスレッド、スレッドをやるにはインタフェースが必要なのです)。今回は内容はそれだけにして、なるべく多く演習してもらいます。

1　前回の練習問題の解説

演習 2

演習 2 はどれも比較的関連しているので、まず「拡大縮小」「回転」「コピー」の機能 3 つをすべて入れた Sankaku クラスを示す。

```
class Sankaku {
  Color color; // 色
  double gx, gy; // 重心の X,Y 座標
  double dx0, dy0, dx1, dy1, dx2, dy2; // 変移ベクトル 3 組

  public Sankaku(Color c, double x0, double y0,
                 double x1, double y1, double x2, double y2) {
    gx = 0.333333 * (x0+x1+x2); gy = 0.333333 * (y0+y1+y2);
    dx0 = x0 - gx; dx1 = x1 - gx; dx2 = x2 - gx;
    dy0 = y0 - gy; dy1 = y1 - gy; dy2 = y2 - gy;
    color = c;
  }
  public void draw(Graphics g) {
    int[] x = new int[]{(int)(gx+dx0), (int)(gx+dx1), (int)(gx+dx2)};
    int[] y = new int[]{(int)(gy+dy0), (int)(gy+dy1), (int)(gy+dy2)};
    g.setColor(color); g.fillPolygon(x, y, 3);
  }
  public void moveTo(double x, double y) { gx = x; gy = y; }
  public double getX() { return gx; }
  public double getY() { return gy; }
  public void setColor(Color c) { color = c; }
  public Color getColor() { return color; }
  // ここまで変更なし。
```

*筑波大学大学院経営システム科学専攻

```
    public void scale(double r) {
      dx0 *= r; dy0 *= r; dx1 *= r; dy1 *= r; dx2 *= r; dy2 *= r;
    }
    public void rotate(double deg) {
      double t0 = Math.atan2(dy0, dx0); double r0 = Math.sqrt(dx0*dx0+dy0*dy0);
      double t1 = Math.atan2(dy1, dx1); double r1 = Math.sqrt(dx1*dx1+dy1*dy1);
      double t2 = Math.atan2(dy2, dx2); double r2 = Math.sqrt(dx2*dx2+dy2*dy2);
      double t = Math.PI * deg / 180.0;
      dx0 = r0*Math.cos(t0+t); dy0 = r0*Math.sin(t0+t);
      dx1 = r1*Math.cos(t1+t); dy1 = r1*Math.sin(t1+t);
      dx2 = r2*Math.cos(t2+t); dy2 = r1*Math.sin(t2+t);
    }
    public Sankaku copy() {
      return new Sankaku(color, gx+dx0, gy+dy0, gx+dx1, gy+dy1, gx+dx2, gy+dy2);
    }
  }
```

scale() は単に dx0、dy0、dx1、dy1、dx2、dy2 を r 倍するだけ。

rotate() はやっていることは簡単で、まず 3 頂点の位置ベクトルを極座標表現にするため角度とベクトルの長さを求め、次に指定された角度を足してから再度直交座標に戻す。

copy() は実は一番簡単で、色と 3 頂点の座標を指定して Sankaku のコンストラクタを呼べば新しい三角形オブジェクトができるので、これを返すだけ。

ではこれらを使ったアプレットクラスを見てみる。a と b は 1 つで兼ねることにしよう。

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

  public class r6ex2a extends Applet {
    Sankaku s1, s2;
    public void init() {
      Color c1 = new Color(100, 110, 50);
      Color c2 = new Color(10, 240, 80);
      s1 = new Sankaku(c1, 20.0, 150.0, 100.0,  150.0, 50.0, 30.0);
      s2 = new Sankaku(c2, 90.0, 100.0, 180.0, 90.0, 60.0, 30.0);
    }
    public void paint(Graphics g) {
      s1.moveTo(s1.getX()+5, s1.getY()-3);
      s2.moveTo(s2.getX()+3, s2.getY()+4);
      s1.scale(1.1); s2.scale(0.9);
      s1.rotate(5.0); s2.rotate(-3.0);
      s1.draw(g); s2.draw(g);
    }
  }
```

これは単に scale() と rotate() を呼んでみせるだけ。さてコピーの方だが:

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

  public class r6ex2c extends Applet {
```

```
  Sankaku a[] = new Sankaku[20];
  int count = 0;
  public void init() {
    Color c0 = new Color(10, 20, 0);
    a[count] = new Sankaku(c0, 90.0, 100.0, 180.0, 90.0, 60.0, 30.0);
    ++count;
  }
  public void paint(Graphics g) {
    for(int i = 0; i < count; ++i) a[i].rotate((double)i);
    if(count < 20) {
      a[count] = a[count-1].copy();
      a[count].setColor(a[count-1].getColor().brighter());
      a[count].moveTo(a[count].getX()+10, a[count].getY()+5);
      ++count;
    }
    for(int i = 0; i < count; ++i) a[i].draw(g);
  }
}
```

まずSankakuの入る配列を用意し、いくつ入っているかは変数countで覚えておく。init()で最初の1個を入れる。paint()ではまずそれぞれの三角形を適当な度数ずつ回転させ、次に配列の大きさ20を越えていない範囲で、一番新しい三角形のコピーを作って配列に追加格納し、色と位置をちょっとだけ前のものと変える。これで、再読み込みするたびにちょっとずつ回転しつつ数が増えるわけだ。

このように配列を使えば「同じものが並んでいる」というのがうまく扱えるのが分かりますか? まあまだ慣れないと思うので、この辺は後でまた出てきます。

演習 3a

以下の演習はすべて、その図形を描くのにどれだけ情報を持っていればよいかきちんと考えてインスタンス変数を用意するところがポイントかと思いますがどうでしょう?

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

public class r6ex3a extends Applet {
  Hinomaru s1, s2;
  public void init() {
    s1 = new Hinomaru(150.0, 100.0, 40.0, 80.0, 60.0);
    s2 = new Hinomaru(190.0, 120.0, 50.0, 60.0, 60.0);
    s2.setColors(Color.blue, Color.green);
  }
  public void paint(Graphics g) {
    s1.moveTo(s1.getX()+5, s1.getY()-3);
    s2.moveTo(s2.getX()+3, s2.getY()+4);
    s1.setRotation(s1.getRotation()+5.0);
    s2.setSize(s2.getDiam(), s2.getWidth()*1.1, s2.getHeight()*1.2);
    s2.draw(g); s1.draw(g);
  }
}
```

```
class Hinomaru {
  Color c0, c1; // 丸の色、地の色
  double gx, gy; // 中心の位置
  double r0, rx, ry; // 丸の直径、幅、高さ
  double deg; // 傾き

  public Hinomaru(double x, double y, double r, double w, double h) {
    gx = x; gy = y; r0 = r; rx = w; ry = h;
    c0 = Color.red; c1 = Color.white; deg = 0.0;
  }
  public void draw(Graphics g) {
    double t0 = Math.atan2(ry, rx);
    double t = Math.PI*deg/180.0;
    double l = Math.sqrt(rx*rx+ry*ry)*0.5;
    int[] x = new int[4]; int[] y = new int[4];
    x[0] = (int)(gx + l*Math.cos(t0+t));
    y[0] = (int)(gy + l*Math.sin(t0+t));
    x[1] = (int)(gx + l*Math.cos(Math.PI-t0+t));
    y[1] = (int)(gy + l*Math.sin(Math.PI-t0+t));
    x[2] = (int)(gx + l*Math.cos(Math.PI+t0+t));
    y[2] = (int)(gy + l*Math.sin(Math.PI+t0+t));
    x[3] = (int)(gx + l*Math.cos(-t0+t));
    y[3] = (int)(gy + l*Math.sin(-t0+t));
    g.setColor(c1);
    g.fillPolygon(x, y, 4);
    g.setColor(c0);
    g.fillOval((int)(gx-r0/2), (int)(gy-r0/2), (int)r0, (int)r0);
  }
  void moveTo(double x, double y) { gx = x; gy = y; }
  double getX() { return gx; }
  double getY() { return gy; }
  void setSize(double r, double w, double h) { r0 = r; rx = w; ry = h; }
  double getDiam() { return r0; }
  double getWidth() { return rx; }
  double getHeight() { return ry; }
  void setColors(Color c, Color b) { c0 = c; c1 = b; }
  Color getCircleColor() { return c0; }
  Color getBaseColor() { return c1; }
  void setRotation(double d) { deg = d; }
  double getRotation() { return deg; }
}
```

日の丸を回転させるには結局、三角形の時と同様に極座標にして回転させ、また描くのも `fillRect()` は使えないので `fillPolygon()` を使うことになりますね。

演習 3c

十字は面倒くさい割につまらないので略。N 角星ができれば N 角形にするのは簡単なので、星だけ示します。これは十字よりかえって簡単で、360 度を $2N$ 等分して長径、短径と交互に頂点を置けばよいので、ループで座標を設定できます。

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

public class r6ex3c extends Applet {
  NStar s1, s2;
  public void init() {
    s1 = new NStar(150.0, 90.0, Color.red, 5, 80.0, 30.0);
    s1.setRotation(-18.0);
    s2 = new NStar(200.0, 60.0, Color.blue, 8, 30.0, 10.0);
  }
  public void paint(Graphics g) {
    s1.moveTo(s1.getX()+5, s1.getY()-3);
    s2.moveTo(s2.getX()+3, s2.getY()+4);
    s1.setRotation(s1.getRotation()+5.0);
    s2.setSize(s2.getLongDiam()*1.1, s2.getSortDiam()*1.05);
    s1.draw(g); s2.draw(g);
  }
}

class NStar {
  Color c0; // 丸の色
  double gx, gy; // 中心の位置
  double r0, r1; // 中心から頂点までの距離 (長、短)
  double deg; // 傾き
  int n; // N 角形の N

  public NStar(double x, double y, Color c, int num, double l, double s) {
    gx = x; gy = y; c0 = c; n = num; r0 = l; r1 = s; deg = 0.0;
  }
  public void draw(Graphics g) {
    int[] x = new int[2*n]; int[] y = new int[2*n];
    double dt = 2.0*Math.PI / (2*n);
    double t = Math.PI*deg/180.0;
    for(int i = 0; i < 2*n; ++i) {
      if(i%2 == 0) {
        x[i] = (int)(gx + r0*Math.cos(t));
        y[i] = (int)(gy + r0*Math.sin(t));
      } else {
        x[i] = (int)(gx + r1*Math.cos(t));
        y[i] = (int)(gy + r1*Math.sin(t));
      }
      t += dt;
    }
```

```
    g.setColor(c0); g.fillPolygon(x, y, 2*n);
  }
  void moveTo(double x, double y) { gx = x; gy = y; }
  double getX() { return gx; }
  double getY() { return gy; }
  void setSize(double l, double s) { r0 = l; r1 = s; }
  double getLongDiam() { return r0; }
  double getSortDiam() { return r1; }
  void setColor(Color c) { c0 = c;  }
  Color getColor() { return c0; }
  void setRotation(double d) { deg = d; }
  double getRotation() { return deg; }
}
```

2 インタフェース

ところで、これまでのさまざまなお絵描きオブジェクトはどれも「画面に現れる」という点で共通していたが、それらを利用するアプレットクラスではそれぞれのオブジェクト用の変数を別に用意していた。このままではさまざまなものが増えたり減ったりする複雑な絵は難しい。もっと統一的に扱うには、たとえば、配列を用意して

```
a[0] = new Hinomaru(...);
a[1] = new Sankaku(...);
a[2] = new NStar(...);
...
for(int i = 0; i < count; ++i) a[i].draw(g);
```

のようなことがやりたいのだが、この場合配列 a の型は「何型の配列」にすればいいのだろう?

そこで、「こういう共通の呼び出しを持ったもの」を「ひとまとめにくくって」扱える仕組みである「インタフェース」について学ぼう。たとえば、壁についているコンセントもそこに差し込むプラグも色々な色や形をしているが、差し込む部分の規格は一緒で、どこのコンセントにどのプラグを差し込んでもちゃんと使える。インタフェースとはこのような「統一された規格」を決めるためのものである。

たとえば、「draw(g) で画面に描ける」という「規格」のみを 1 つのインタフェースとして定めたものを次に示す (Drawable という名前をつけた)。

```
interface Drawable {
  public void draw(Graphics g);
}
```

このように、インタフェースはクラスと似ているが (1)interface というキーワードで始まること、(2) 変数は定義しないこと、(3) メソッドは頭の部分 (パラメタ部分) までで、本体は書かないこと、が違いである。インタフェースはクラスと同様、型でもあるので、Drawable 型の変数を作ることができる。

次に、このようなインタフェースに「従っている」(Java の言葉では「実装している」という) クラスを作るには、次のことが必要である。

- クラスの冒頭部分 (最初の{の前) に「implements Drawable」という指定を書く。
- メソッドとして「public void draw(Graphics g) { ... }」なるメソッドを用意する。

このようなクラス X を作ったとすると、X のインスタンス (オブジェクト) は Drawable 型の変数に入れることができる。また、それに対してメソッド draw() を呼び出すこともできる。しかし、それ以外のメソッドは (もともとのクラス X にあったとしても) 呼ぶことはできない。

```
Drawable d = new X(...);  // d には X のインスタンスが格納可能
...
d.draw(g); // OK
```

では、Drawablle を実装する「円」のクラス DCircle と「正方形」のクラス DSquare を作り、それらをまとめて画面に描くようなアプレットの例題を示す。一度作ってしまえば、取り扱いはすべて一緒にできる、という点がポイントである。

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

public class R6Sample1 extends Applet {
  Drawable a[] = new Drawable[20];
  int count = 0;
  public void init() {
    a[count] = new DCircle(Color.red, 100, 100, 20); ++count;
    a[count] = new DCircle(Color.blue, 80, 150, 30); ++count;
    a[count] = new DSquare(Color.green, 200, 80, 40); ++count;
  }
  public void paint(Graphics g) {
    for(int i = 0; i < count; ++i) a[i].draw(g);
  }
}

interface Drawable {
  public void draw(Graphics g);
}

class DCircle implements Drawable {
  Color col;
  double gx, gy, rad;
  public DCircle(Color c, double x, double y, double r) {
    col = c; gx = x; gy = y; rad = r;
  }
  public void draw(Graphics g) {
    g.setColor(col);
    g.fillOval((int)(gx-rad), (int)(gy-rad), 2*(int)rad, 2*(int)rad);
  }
}

class DSquare implements Drawable {
  Color col;
  double gx, gy, len;
  public DSquare(Color c, double x, double y, double l) {
    col = c; gx = x; gy = y; len = l;
  }
  public void draw(Graphics g) {
    g.setColor(col);
    g.fillRect((int)(gx-0.5*len), (int)(gy-0.5*len), (int)len, (int)len);
```

```
    }
  }
```

実は! こういうことは前回学んだ「継承」でも行える。つまり、X のサブクラス Y を作った場合、Y のインスタンスは X 型の変数に入れられ、X のインスタンスと同じように扱える。ただし、「継承」の場合はインスタンス変数やメソッドも一緒に引き継いで来るので「構造が似たものを作る」機能がメインで、「共通にまとめて扱える」のはおまけだと考えるべきである。これに対し、インタフェースは純粋に「共通にまとめて扱う」ための機能である。

なお、継承機能の一環として、クラス X がインタフェース I を implements している場合、そのサブクラスも自動的にインタフェース I を implements しているものとして扱われる。

3 マルチスレッド

3.1 スレッドとは?

ここまで学んだプログラミングの方法では、様々なオブジェクトの様々なメソッドを呼び出すにしても、「現在実行している箇所」というのは常に特定の 1 箇所しかない。この「実行経路のひと筆書き」のことを糸になぞらえて「スレッド」と呼ぶ。

これまでのプログラムではすべて、スレッドは 1 つだけだった。既にアニメーションをやろうと思って `paint()` の中で次々と絵を描いたり時間待ちをしようとして敗れた人がいるようですが…なぜかわかります? それは、`paint()` の中でそれを始めるとブラウザにとっては「無限に絵が描き終らない」状態になってしまい、場合によっては他の操作を受け付けなくなるなど、いろいろな問題が起きる。これはすべて、実行が「ひと筆書き」であることに起因している。

ところが実は! Java はスレッドを複数同時に走らせることができる。このような機能を「マルチスレッド」と呼ぶ。マルチスレッドを使えば、これまでの通常の操作と「時間待ちをしてタイミングを取る」操作とを「並列に」実行できるのでうまくアニメーションができるようになる。

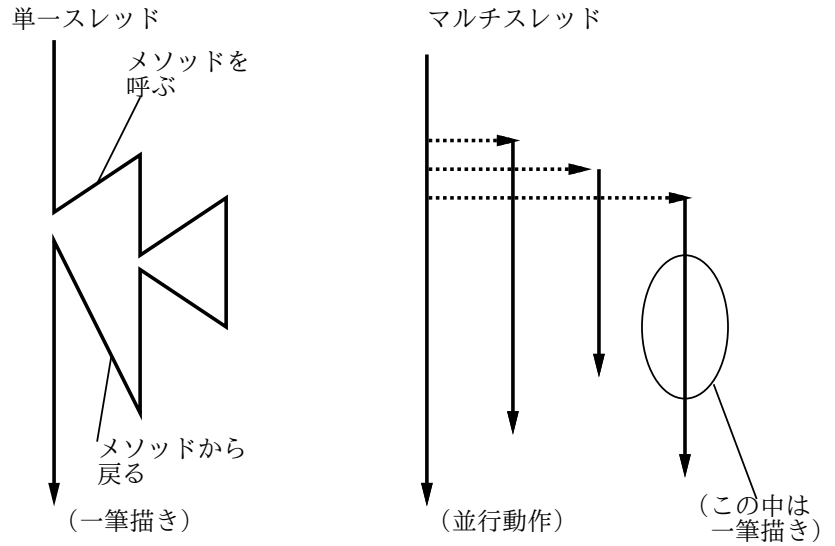

図 1: スレッドの概念

Java ではスレッドは `Thread` クラスのインスタンス (オブジェクト) として表される。つまり、新しい `Thread` オブジェクトを作って、それの `start()` メソッドを呼ぶと新しいスレッドができて動作を開始する…しかし何の動作を開始するのだろう? もちろん、スレッドを作る人がそれぞれやらせたい動作を指定できるのでなければ、ただ新しいスレッドを実行させても無意味である。

そこで、次のようなインタフェース Runnable が登場する (本当はこれは標準 API の java.lang パッケージに含まれているのだけど、あんまり簡単なのでここに再掲する)。

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

つまり Runnable というのは引数を持たず値も返さないメソッド run() を持つことだけを定めた「規格」である。そして Thread クラスのコンストラクタで Runnable インタフェースを実装したオブジェクトを 1 つ渡すと、そのスレッドが実行開始するとそのオブジェクトのメソッド run() が動くわけである。具体的には次のようにする。

```
Thread t = new Thread(...) // Runnable オブジェクト指定して作成
t.start();                 // スレッド実行開始
```

3.2 Thread クラスのメソッド

上の start() も含めて、Thread クラスのよく使うメソッドを以下に挙げておく。

```
public void start() --- スレッドを実行開始させる
public void stop()  --- スレッドの実行を強制終了させる
publid void destroy() --- スレッドを破棄する
public void join()  --- スレッドの実行が完了するまで待つ
public static void sleep(long ms) throws InterruptedException
    --- 現在実行中のスレッドを n ミリ秒停止させる
public static void yeald()
    --- 現在実行中のスレッドから他のスレッドに切り替える
```

最後の 2 つは static メソッドなので「Thread.sleep(...)」などのようにして使う。なお sleep() では停止中に割り込みが起きると InterruptedException という例外が発生するので、必ず try ... catch の中で使わないといけない。

yield() もちょっと説明が必要である。複数のスレッドといったが、実際には我々が使っているマシンの CPU は 1 つだけなので、実際には計算機は「あるスレッドをしばらく実行し、次に別のスレッドをしばらく実行し、…」という風にして小刻みにスレッドを切り替えながら実行していく。ここで「切り替えながら」と書いたがこれには 2 通りの流儀がある。

- 横取り可能なスレッド — 一定時間たつと自動的に切り替わる
- 横取り不可能なスレッド — あるスレッドが別のものに切り替わるのは、そのスレッドが終わるか、sleep() や yield() を実行した時だけ可能

以上は一般の話だったが、Java の実行系も上記 2 種類がある。つまり yield() というのは「やることはまだあるが、他にやりたいことがある人がいたらお先にどうぞ」ということ。

3.3 スレッドによるアニメーション

ではいよいよ、アプレットで「自動的に動く画面」つまりアニメーションを作成してみる。それには、「ずっと動いているスレッド」を作って、このスレッドから定期的にアプレットオブジェクト本体に対して「時刻を進め、画面を描き直せ」という指示を送るようにする。

例題が長いと打ち込むのがしんどいので、できるだけ簡単な例として、絵の部品をクラスにするのは一時棚上げして、単に「時刻」(正確には開始からの秒数) がどんどん変化しながら表示されるアプレットを作ってみた。

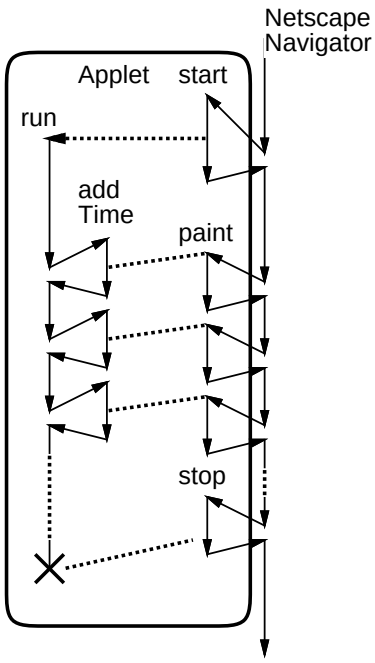

図 2: スレッドによるアニメーション

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

public class R6Sample2 extends Applet implements Runnable {
  boolean running = false;
  Font fn = new Font("Helvetica", Font.BOLD, 24);
  Color c0 = Color.green;
  String mesg = "???";
  int xpos = 100;
  int ypos = 100;
  double time = 0.0;
  public void start() { running = true; (new Thread(this)).start(); }
  public void stop() { running = false; }
  public void paint(Graphics g) {
    g.setFont(fn); g.setColor(c0); g.drawString(mesg, xpos, ypos);
  }
  public void addTime(double dt) {
    time = time + dt; mesg = "Time: "+time; repaint();
  }
  public void run() {
    long basetime = System.currentTimeMillis();
    while(running) {
      try { Thread.sleep(100); } catch(Exception e) { }
      long time = System.currentTimeMillis();
      addTime(0.001*(time-basetime)); basetime = time;
    }
  }
}
```

メソッド start() は、アプレットが「実行開始」したとき呼ばれるので、ここでスレッドを作ってすぐに start() する。なお、Thread のコンストラクタに this、つまりこのアプレットオブジェクト自身を渡していることに注意。これにより、スレッドを start() すると run() が実行される。

paint() はこれまでの例とさして変わらない。最後の run() がこのアプレットの「肝」ということになる。まず、現在の時刻をミリ秒単位で取得し変数 basetime に覚えておく。次に、runninig が true である間無限に次のことを繰り返す。

- 100 ミリ秒だけ待つ (待っている間に割り込まれた場合はそれを無視するように try...catch が使ってある)。
- 現在の時刻を改めて取得する。
- basetime との差分 dt を計算する (実数値で秒単位になるようにしている)。そして basetime は現在時刻に更新する。
- 各オブジェクトの addTime() を呼んで dt だけ時間を進めさせる。
- repaint() を呼んで画面を更新させる。

なお、ユーザがブラウザで別のページを表示させると、stop() が呼ばれてその中で running を false に変更するので、run() の実行が終り、従って作成したスレッドの実行も終わるわけである。なかなか大変でしたか?

演習 1 この例題アプレットを打ち込んでそのまま動かせ。

演習 2 例題アプレットが動いたら、次のように変更してみよ。どれから順にやってもよいが、最終的には a、b、c の 3 つとも実現できることが望ましい。

a. 時刻の文字が表示される位置がだんだん右に動いて行くが、ただしアプレット領域の右端まで来たら一気に左に戻る。(ヒント: 時刻を t として、t(に適当な値を掛けて整数に丸めた値) を領域の幅で割った余りを xpos にするとよい。剰余の演算子は「%」でしたね)

b. 時刻の文字が表示される位置が上下に振動する。(ヒント: 上と似ているが三角関数などに基づいて ypos を計算するとよい)

c. 時刻の文字の色がだんだん変化する。(ヒント: Color.getHSBColor(h, 0.8f, 0.8f) で h を時刻に応じて 0.0f〜1.0f まで変化させると美しい。このメソッドの意味は API ドキュメントでチェックすること。変化させるのは t に適当な値を掛けてから 1 で剰余を取るとよい。引数は double でなく float なのでリテラルにも f がついているのに注意)

これらの演習ができれば、「時間とともに絵が動く」ようする方法はマスターしたことになります。

3.4 動く図形のアニメーション

では次に、図形を動かすアニメーションをやってみよう。まず、さっきの Drawable インタフェースをちょっと拡張して次のような Animation インタフェースを定義する。

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

ここで、addTime() というのは「さっきから dt 時間だけ経過した状態に、形や位置を動かせ」という意味を持つことにする。次に、これを実装するクラスの例として「四角い領域の中を飛んでいる円」を作ることにする。

なお、「四角い領域」の情報は java.awt.Rectangle オブジェクトで表すことにする。このクラスは単に x、y、width、height という 4 つのインスタンス変数を持つだけで、これらの変数は外部から直接アクセスできる。

```
  Rectangle r = new Rectangle(0.0, 0.0, 200.0, 300.0);
  ...
  double w = r.width; // 200.0 のはず
```

このように、Rectangle はあんまり「もの」らしくなくて、どちらかというと「4 つの値の組」という感じである。では、飛ぶ円のクラス。

```
class ACircle implements Animation {
  Rectangle rect;
  Color col;
  double gx, gy, vx, vy, rad;
  double time = 0.0;
  public ACircle(Color c, double x, double y, double r,
                 double vx0, double vy0, Rectangle r0) {
    col = c; gx = x; gy = y; rad = r; vx = vx0; vy = vy0; rect = r0;
  }
  public void draw(Graphics g) {
    g.setColor(col);
    g.fillOval((int)(gx-rad), (int)(gy-rad), 2*(int)rad, 2*(int)rad);
  }
  public void addTime(double dt) {
    gx += vx*dt; gy += vy*dt; time += dt;
    if(gx < rect.x && vx < 0) vx = -vx;
    if(gx > rect.x+rect.width && vx > 0) vx = -vx;
    if(gy < rect.y && vy < 0) vy = - vy;
    if(gy > rect.y+rect.height && vy > 0) vy = -vy;
  }
}
```

今回は「動き」があるので、X 方向と Y 方向の「速度」を変数 vx、vy に入れるものとする。コンストラクタや draw() は前と変わらない (初期設定する変数は増えているが)。このクラスの「肝」は addTime() であるが、ここでやっていることは次の通り。

- まず、重心位置を変更する。もちろん、$X_{t'} = X_t + V_x dt$、$Y_{t'} = Y_t + V_y dt$ ですね? なお、ここでは使っていないが一応変数 time に時間の累計 (経過時間) を覚えている。
- 次に、位置が領域をはみだしていて、なおかつさらにはみだす方向に動いているなら、「はねかえす」つまり速度ベクトルの X 成分や Y 成分を符合反転する。

これだけで「はねかえり」が実現できるわけです。簡単でしょう? なお、このようなふるまいは「現実世界において、無重力空間の中を動いている弾性体」を「まねして」いる。このように、現実世界の現象を計算機の中で計算によって「まねる」ことをシミュレーション (simulation) と呼ぶ。(*シュミ*レーションじゃないよ!)

では、アプレット本体を見てみよう。アプレットは Runnable インタフェースを実装しているので、このアプレット本体のメソッド run() をスレッドとして実行させることができる。

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

public class R6Sample2 extends Applet implements Runnable {
  Animation a[] = new Animation[20];
  int count = 0;
```

```
  boolean running = false;

  public void init() {
    Rectangle r = new Rectangle(0, 0, 300, 200);
    a[count] = new ACircle(Color.red, 100, 100, 20, 35, 74, r); ++count;
    a[count] = new ACircle(Color.blue, 80, 150, 30, -28, 52, r); ++count;
  }
  public void start() { running = true; (new Thread(this)).start();  }
  public void stop() { running = false; }
  public void paint(Graphics g) {
    for(int i = 0; i < count; ++i) a[i].draw(g);
  }
  public void run() {
    long basetime = System.currentTimeMillis();
    while(running) {
      try { Thread.sleep(100); } catch(Exception e) { }
      long time = System.currentTimeMillis();
      double dt = 0.001*(time-basetime); basetime = time;
      for(int i = 0; i < count; ++i) a[i].addTime(dt);
      repaint();
    }
  }
}
// ここにインタフェース Animation を入れる
// ここにクラス ACircle を入れる
```

前半はさっきの配列を使った例題とほとんど変わらないが、ただし現在絵が「動いているかどうか」を変数 running に覚えておく。この変数は boolean 型なので true か false かの値を取る。

演習 3 上記のアニメーションの例題アプレットをそのまま動かせ。

演習 2 上記の例題アプレットにおける円の動き (つまりシミュレーションの内容) を次のように変更してみよ。

a. 上の例題では、無重力空間を「飛んで」いたが、「重力」が働くようにしてみよ。
b. はね返る代りに反対側からワープして出て来る (または領域の中央にワープして戻る) ようにしてみよ。
c. 上の例題では、はね返るときに壁にめりこんでいたが、めりこまずに壁のところではね返るようにしてみよ。
c. 「飛ぶ」のではなく、ある場所を中心に円運動したり振動しさせてみよ。
d. 円どうしが「衝突してはね返る」ようにする (インタフェース Animation にメソッドを追加する必要があり、極めて難しい。よほど精力の余っている人むけ)。

なお、できればクラスを直接直して全部の円の動きを変えるよりは、動きごとに新しいクラスを作って複数の異なる動きの円を共存させるほうがかっこいい。

演習 5 上記の例題アプレットに円以外の新しい図形を次のように追加してみよ。

a. 回転しながら飛ぶ三角形や矩形や十字型。
b. 回転したり振動しながら飛んだりするが、途中で突然飛び方や回転のしかたが変化するような何か。
c. ぱたぱたと羽ばたきながら飛ぶちょうちょ(?) もどき。

演習 6 何でもいいから、見て楽しく動くアニメーションアプレットを作成せよ。