計算機プログラミング'99　＃4

久野　靖*

1999.9.29

0　はじめに

前回でアプレットが作れるようになったので、今回はその続きとしてアニメーションを行うアプレットを作ります。Java 言語の内容としては、前回でクラスを作るようになったので、今回は「継承」「動的分配」を中心に取り上げることにしましょう。

1　クラスと継承

前回アプレットのところで出て来た「`extends`」の意味についてもっと具体的に説明しよう。Java のようなクラス方式のオブジェクト指向言語の多くは、「継承」(inheritance) と呼ばれる機能を持っている。この機能は「あるクラス A を土台として、新しいクラス B を定義する」のに使う。このとき、クラス A を「親クラス」「スーパークラス」、クラス B を「子クラス」「サブクラス」と呼ぶ。Java の構文ではクラス B を定義するところで「`class B extends A {`」という形で継承関係を指定する。

具体的には、このとき起こることは次の通りである。

- クラス B はクラス A の変数定義 (クラス変数、インスタンス変数とも) 一式をそっくりそのまま引き継ぐ。クラス B で新たに変数を定義することもでき、その場合はクラス B は A から引き継いだ変数群と新たに定義した変数群を併せ持つことになる。
- クラス B はクラス A のメソッド定義 (クラスメソッド、インスタンスメソッドとも) 一式をそっくりそのまま引き継ぐ。クラス B で新たにメソッドを定義することもでき、その場合はクラス B は A から引き継いだメソッド群と新たに定義したメソッド群を併せ持つことになる。
- また、クラス B ではクラス A から引き継いでいるメソッドと同じものを再定義 (override、オーバライド) することもできる。その場合はメソッドの動作は B で新たに定義したもので差し替えられる。

というわけで、前回アプレットを定義したときは、クラス `Applet` からアプレットの基本的な機能 (変数群、メソッド群) を継承し、固有の動作を行わせたいメソッド `paint()`、`init()` をオーバライドしてそこに行わせたい内容を記述したわけである。

演習 1 クラス `java.applet.Applet` の API ドキュメントを見て、どのようなメソッドがあるか確認してみよう。

注意: メソッド `paint()` は実は `Applet` クラスがさらにその親クラスから継承しているものである。

*筑波大学大学院経営システム科学専攻

2　マルチスレッドとアニメーション

2.1　スレッドとは?

さて、前回「キーを押すと動く絵」を作ることができたが、やはり「何もしないでも動いている絵」(アニメーション) が作って見たいですね? その方法を学ぼう。

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。そして実は! 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。

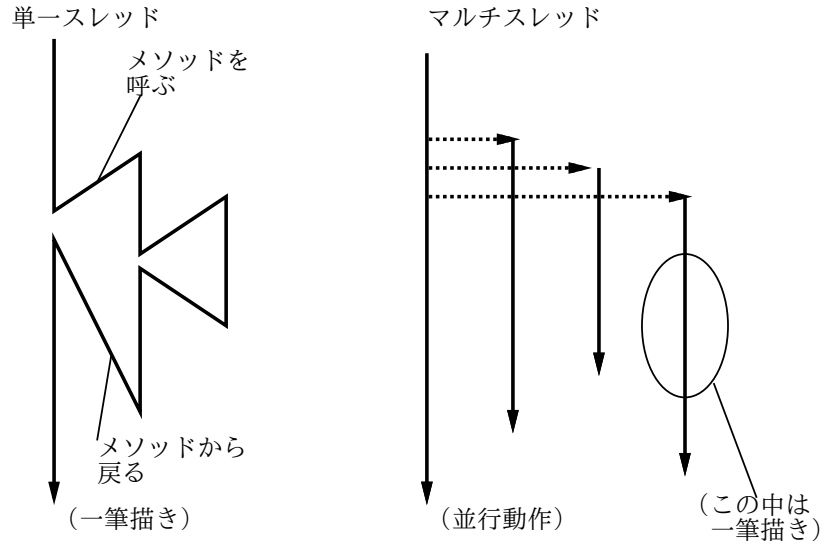

図 1: スレッドの概念

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。

スレッドの中で実際に実行される動作は run() というメソッドに記されている内容である。Thread の run() は手が出せないが、先に説明したようにそのサブクラスを作れば自分独自の run() の定義で差し替えることができる (オーバライド) ので、これを利用すればよい。それ以外のメソッドは、Thread で定義されたものをそのまま使うので特に何もしないでよい。

2.2　Thread クラスのメソッド

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 (エラー) が発生するので、必ず `try ... catch` の中で使わないといけない (つまりいつも main() で書いてるのと同じようにする)。

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。

2.3 簡単な例題

ごたくが多すぎたのでそろそろ例題に行こう。

```
public class R4Sample1 {
  public static void main(String args[]) {
    try {
      Thread t1 = new MyThread("Vow!", 300, 15); t1.start();
      Thread t2 = new MyThread("Meaow!", 200, 20); t2.start();
      Thread t3 = new MyThread("Moo!", 250, 20); t3.start();
      t1.join(); t2.join(); t3.join();
    } catch(Exception e) { System.err.println("!!"+e); }
  }
}
class MyThread extends Thread {
  String mesg;
  int ms, count;
  public MyThread(String m, int t, int c) {
    mesg = m; ms = t; count = c;
  }
  public void run() {
    for(int i = 0; i < count; ++i) {
      try {
        Thread.sleep(ms);
      } catch(Exception e) { System.err.println("!!"+e); }
      System.out.println(mesg);
    }
  }
}
```

`main()` は単に 3 つの MyThread を作って実行開始させ、終了を待つ。MyThread は Thread のサブクラスで、インスタンス変数としてメッセージ、時間間隔、回数を保持し、指定された回数だけ指定された時間間隔でメッセージを打ち出す。

```
% javac R6Sample1.java
% kaffe R6Sample1
Meaow!
Moo!
Vow!
Meaow!
Moo!
Vow!
...
```

それぞれのスレッドでは連続してメッセージを出力しているのに、互いに混ざり合っているところが不思議でしょう?

3 アプレットでのアニメーション

ではいよいよ、アプレットでどうやってアニメーションを行うかを説明しよう。まず、init() や paint() などのメソッドはブラウザ側の「メインスレッド」によって実行されるので、この中で時間待ちなどを行うとブラウザ内部の動作が「いつまでも終わらない」状態になってまずい、という話は前にした。

そこで、start() メソッド (というのは、アプレットがページに現われる時にメインスレッドによって呼び出される) の中でいわば「時間管理用」の新しいスレッドを作り、そのスレッドが定期的にアプレットのメソッドを呼び出すことで「今何時何分ですよ」という情報をアプレットに教えるようにする。教えられたアプレットでは、教えられた時刻に基づいてあるべき絵を作成し、これまで通り repaint() と paint() で画面に描画していけばよい。

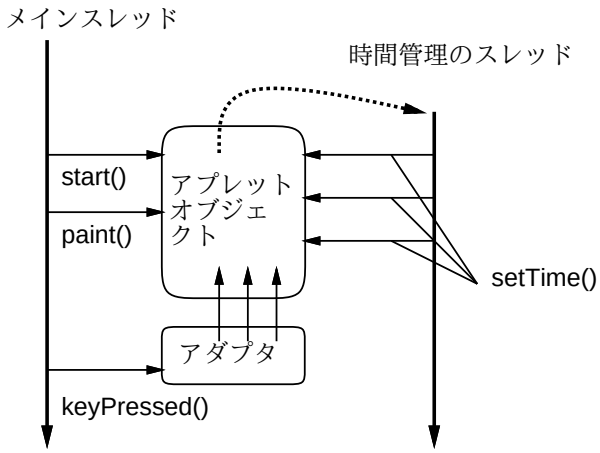

図 2: スレッドによるアニメーション

ではこれを実際に行うコードを見てみよう。まずアプレットクラスから。

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

public class R4Sample2 extends Applet {
  AnimCircle c0, c1;
  boolean running = false;
  public void init() {
    this.setBackground(Color.white);
```

```
    c0 = new AnimCircle(300, 200, Color.blue, 100, 100, 20, 33.0f, 15.0f);
    c1 = new AnimCircle(300, 200, Color.green, 130, 110, 25, -23.0f, 45.0f);
    addKeyListener(new KeyAdapter() {
      public void keyPressed(KeyEvent e) {
        char c = e.getKeyChar();
        if(c == '+') { c0.changeSpeed(1.1f); }
        if(c == '-') { c0.changeSpeed(0.9f); }
      }
    });
  }
  public void start() { running = true; (new MyThread()).start(); }
  public void stop() { running = false; }
  public void paint(Graphics g) { c0.draw(g); c1.draw(g); }
  public void initTime(float t) { c0.initTime(t); c1.initTime(t); }
  public void setTime(float t) { c0.setTime(t); c1.setTime(t); repaint(); }
  // ここに MyThread クラスを入れる
  // ここに AnimCircle クラスを入れる
}
```

このクラスでは、「動く円」を表すのに AnimCircle クラス、時間管理スレッドを表すのに MyThread クラスを用いる。変数 c0、c1 は 2 つの動く円を格納する。また変数 running は「現在円が動いている」かどうかを表すのに使う。

init() の中では 2 つの円を生成している。また、キーが押されたときには、押されたキーが「+」なら片方の円の速度を 1.1 倍にし、「-」なら 0.9 倍にするよう設定する。start() の中では running に true を設定してから、時間管理スレッドを作成して起動する。stop() では running に false を入れるだけである (そうすると時間管理スレッドが勝手に終了する)。paint() は前回までと同じで、それぞれの円の draw() を呼んで画面に表示される。initTime() と setTime() はそれぞれ、「最初の時計合わせ」と「今何時ですよ」に相当するが、時間の管理は AnimCircle クラスの機能として用意してあるので、これらのメソッドでは単に AnimCircle クラスのメソッドを呼び出すだけである (setTime() ではその後 repaint() を呼んで画面を再描画する)。

では時間管理スレッド用クラスを見てみよう。

```
class MyThread extends Thread {
  public void run() {
    long basetime = System.currentTimeMillis();
    initTime(0.0f);
    while(running) {
      try { Thread.sleep(100); } catch(Exception e) { }
      setTime(0.001f * (System.currentTimeMillis() - basetime));
    }
    destroy();
  }
}
```

このクラスは Thread の子クラスで run() だけをオーバライドしている。そこではまずアプレットの時計を 0 に合わせ、またシステムの「ミリ秒単位の時計」を読んでその時間を basetime に覚える。その後は、100 ミリ秒だけ寝て待ち、起きたらまた「ミリ秒単位の時計」を読んで最初の時刻との差を秒単位に換算し、applet の setTime() を呼んで知らせることを running が true である間繰り返す。running が false になったら動作を止めて自分自身を破棄する。

最後に、「動く円」を表すクラス AnimCircle を見てみよう。図形そのものは円だから簡単だが、「動く」ので「速度」を持っていて、時間経過とともにその速度に応じて位置が変化するという点がこれまでと違う。

```
class AnimCircle {
  int width, height;
  Color cl;
  float vx, vy, px, py, rad;
  float basetime = 0.0f;
  AnimCircle(int w, int h,
             Color c, int x, int y, int r, float vx1, float vy1) {
    width = w; height = h;
    cl = c; px = x; py = y; rad = r; vx = vx1; vy = vy1;
  }
  public void changeSpeed(float ratio) { vx *= ratio; vy *= ratio; }
  public void initTime(float time) { basetime = time; }
  public void setTime(float time) {
    px += (time - basetime)*vx;
    py += (time - basetime)*vy;
    if(px-rad < 0.0f) { vx = -vx; px = 2*rad - px; }
    if(px+rad > width) { vx = -vx; px = width*2 - (px+2*rad); }
    if(py-rad < 0.0f) { vy = -vy; py = 2*rad - py; }
    if(py+rad > height) { vy = -vy; py = height*2 - (py+2*rad); }
    basetime = time;
  }
  public void draw(Graphics g) {
    g.setColor(cl);
    g.fillOval((int)(px-rad), (int)(py-rad), (int)(2*rad), (int)(2*rad));
  }
}
```

一番の肝は時刻に応じて場所を変化させるメソッド setTime() で、その先頭では「距離=時間×速さ」で先の位置からの変移ベクトルを求めて前の位置に足し込んでいる。しかしそれだけだと無限に直線運動して画面の外へ出て行ってしまうため、画面のふちではね返るようにしたい。その計算が続く 4 行になっているが、これの仕組みは図 3 をよく見ていただければ分かると思う。

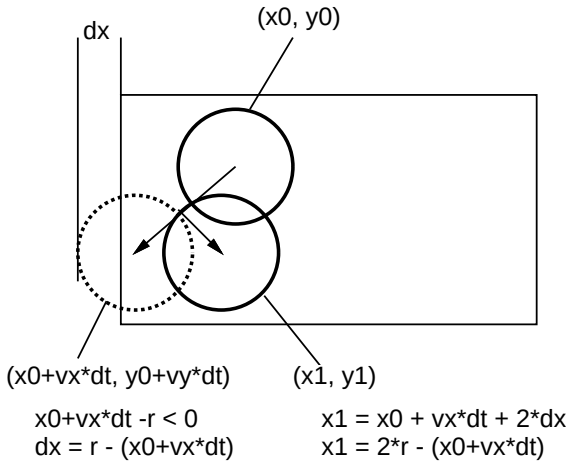

図 3: はね返り後の位置の計算

演習 2 上の「はね返り」の例題を打ち込んでそのまま動かせ。

演習 3 動いたら、次のような機能を追加してみよ。

a. 「!」キーを打つと片方の円がこれまでと逆の方向に同じ速度で動くようにしてみよ (ヒント: 速度を-1.0 倍にする)。または、垂直方向に反射するキーと水平方向に反射するキーの 2 つを用意してみてもよい (その場合 AnimCircle のメソッドも適宜追加する必要がある)。
b. 上の例は無重力空間だったが、重力を入れてみよ。つまり Y が増える方向に一定加速度が働くようにしてみよ。それでつまらなければ、上方向や横方向に加速度が働くようにしたらどうか?
c. 2 つの円の間に見えない力 (吸引力でも反発力でもいい) が働くようにしてみよ。もしできたら円を 3 つ、4 つにしてみよ。
d. 2 つの円の一方をたとえば a のようなやり方でユーザが制御して、他の円にぶつからないように逃げるゲームにしてみよ。開始から何秒間経過したかを絶えず画面に表示し続けるとかっこいい (アプレットの中でも時間をインスタンス変数に保持し、paint() の中で画面内の決まった場所にその時間を drawString() すればよい)。

4 継承とオブジェクトの階層

4.1 変数へのサブクラス値の格納

継承に関連して、もう 1 つ重要なことを説明しておこう。これまでは基本的に、X 型の変数には X 型のオブジェクトや値だけを格納していた。しかし実は、X がクラスの場合「入れられるかどうか」は次の規則によって決まる。

> 変数 x の型がクラス C 型である場合、変数 x にはクラス C のインスタンスだけでなく、C の任意のサブクラスクラス D のインスタンスも格納することができる。

たとえば、「のりもの」というクラスを作って、そのサブクラスとして「自転車」「バイク」「自動車」を作り、「自動車」のサブクラスとして「乗用車」「トラック」「バス」を作るようなことを考えるわけである。このとき、「のりもの」型の変数 x には「のりもの」なら何でも入られるのが自然なので、「バス」も「自転車」も入れることができるわけである。

4.2 動的分配

ここでさらに重要なことがあって、この変数 x に対して「`x.setSpeed(50);`」のようなメソッド呼び出しを行うと、

> 実際に呼び出されるメソッドは x に格納されているインスタンスがクラス C のインスタンスなら C で定義されているメソッド、サブクラス D のインスタンスなら D で定義されているメソッドというように、実際に格納されているインスタンスのクラスで定義されているものになる

という点である。たとえば `x.setSpeed(50)` を実行するとき x にトラックのインスタンスが格納されていれば (たとえばアクセルを踏み込んだりして) トラックのスピードを 50 にするという動作が行われるし、x に自転車のインスタンスが格納されていれば必死でこいで 50 にしようとするか、またはもしかしたら自転車では 50 も出ないというエラーになるかも知れない。

要は実際に x に入っているインスタンスの種類 (クラス) に応じて動作も変わるということである。メソッド呼び出し時のこの機能を動的分配 (dynamic dispatch) と呼び、オブジェクト指向言語の重要な機能である。

4.3 例題: 継承を用いたアニメーションの構造化

では、上で説明した 2 つの機能を用いてさまざまな図形を描くアニメーションプログラムを「きれいに」構造化して作ってみよう。まず、すべての「描けるもの」を表すクラスとして GrObject なるものを用意し、アプレットではそれを 1 つだけ保持していて描く、という形にする。

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

public class R4Sample3 extends Applet {
  GrObject obj;
  boolean running = false;
  public void init() {
    obj = new GrObject();
  }
  public void start() { running = true; (new MyThread()).start(); }
  public void stop() { running = false; }
  public void paint(Graphics g) { obj.draw(g, 150.0, 100.0); }
  public void initTime(float t) { obj.initTime(t); }
  public void setTime(float t) { obj.setTime(t); repaint(); }
  class MyThread extends Thread {
    public void run() {
      long basetime = System.currentTimeMillis();
      initTime(0.0f);
      while(running) {
        try { Thread.sleep(100); } catch(Exception e) { }
        setTime(0.001f * (System.currentTimeMillis() - basetime));
      }
      destroy();
    }
  }
}
```

GrObject の定義は後で出てくるが、とにかく「時間を設定」と「この場所に描く」というメソッドはあることにする。スレッドオブジェクトの方は先の例題とまったく同じ。

4.4 抽象的な描画オブジェクト

さて、では GrObject の定義を示そう。描ける「もの」は、インスタンス変数として位置、回転角度、スケール (拡大率)、色、基準時間を持ち、これらを定義設定したり参照するメソッドが用意されている。コンストラクタでは位置だけを引数として渡す (引数なしのコンストラクタでは位置を (0,0) にする。「this(...)」というのは自分のクラスの他のコンストラクタを呼び出すことを意味する)。

```
class GrObject {
  double gx, gy;
  double deg = 0.0, sc = 1.0;
  Color cl = Color.blue;
  double basetime = 0.0;
  public GrObject() { this(0.0, 0.0); }
  public GrObject(double x, double y) { gx = x; gy = y; }
```

```
  public void moveTo(double x, double y) { gx = x; gy = y; }
  public double getX() { return gx; }
  public double getY() { return gy; }
  public void setRotation(double d) { deg = d; }
  public double getRotation() { return deg; }
  public void setScale(double s) { sc = s; }
  public double getScale() { return sc; }
  public void setColor(Color c) { cl = c; }
  public Color getColor() { return cl; }
  public void initTime(double t) { basetime = t; }
  public void setTime(double t) { } // do nothing
  public void draw(Graphics g, double x, double y) { } // do nothing
}
```

ところで、最後の 2 つのメソッドは実際には「何もしない」。というのは、時間に応じて形を変えたり、形を描いたりするのは具体的な形が決まっていないとできないから。ということは、GrObject というクラスはそれだけでは何も役に立たないことになる。

4.5 具体的な形を持つオブジェクト

では実際の形を持つオブジェクトはどうやって作るかというと、GrObject のサブクラスを作って、その中で setTime() や draw() を適宜差し替えて必要な処理を記述する。まず「円」から見てみよう。円では半径が必要なのでそれを覚えるインスタンス変数 rad を追加している。コンストラクタでは位置のほかに半径も指定する (「super(...)」というのは親クラスのコンストラクタを呼び出す指定である)。

```
class GrCircle extends GrObject {
  double rad;
  public GrCircle(double x, double y, double r) { super(x, y); rad = r; }
  public void draw(Graphics g, double x, double y) {
    int px = (int)(gx+x); int py = (int)(gy+y); int r = (int)(sc*rad);
    g.setColor(cl); g.fillOval(px-r, py-r, 2*r, 2*r);
  }
}
```

三角形や矩形もこれまでにやった通り。三角形では draw() の処理を速くするため、3 頂点の変移ベクトルを極座標形式にして保持するようにした。

```
class GrTriangle extends GrObject {
  double r0, r1, r2, t0, t1, t2;
  public GrTriangle(double x0, double y0,
              double x1, double y1, double x2, double y2) {
    gx = 0.33333*(x0+x1+x2); gy = 0.33333*(y0+y1+y2);
    r0 = Math.sqrt((x0-gx)*(x0-gx)+(y0-gy)*(y0-gy));
    r1 = Math.sqrt((x1-gx)*(x1-gx)+(y1-gy)*(y1-gy));
    r2 = Math.sqrt((x2-gx)*(x2-gx)+(y2-gy)*(y2-gy));
    t0 = Math.atan2(y0-gy, x0-gx);
    t1 = Math.atan2(y1-gy, x1-gx);
    t2 = Math.atan2(y2-gy, x2-gx);
  }
  public void draw(Graphics g, double x, double y) {
```

```
    int[] px = new int[3]; int[] py = new int[3];
    px[0] = (int)(x + gx + sc*r0*Math.cos(t0+Math.PI*deg/180));
    py[0] = (int)(y + gy + sc*r0*Math.sin(t0+Math.PI*deg/180));
    px[1] = (int)(x + gx + sc*r1*Math.cos(t1+Math.PI*deg/180));
    py[1] = (int)(y + gy + sc*r1*Math.sin(t1+Math.PI*deg/180));
    px[2] = (int)(x + gx + sc*r2*Math.cos(t2+Math.PI*deg/180));
    py[2] = (int)(y + gy + sc*r2*Math.sin(t2+Math.PI*deg/180));
    g.setColor(cl); g.fillPolygon(px, py, 3);
  }
}
```

矩形もだいたい同様だが、三角形よりも覚える情報は少なくてすむ。

```
class GrRectangle extends GrObject {
  double r0, t0;
  public GrRectangle(double x, double y, double w, double h) {
    super(x, y); r0 = 0.5*Math.sqrt(w*w+h*h); t0 = Math.atan2(h, w);
  }
  public void draw(Graphics g, double x, double y) {
    int[] px = new int[4]; int[] py = new int[4];
    px[0] = (int)(x + gx + sc*r0*Math.cos(t0+Math.PI*deg/180));
    py[0] = (int)(y + gy + sc*r0*Math.sin(t0+Math.PI*deg/180));
    px[1] = (int)(x + gx + sc*r0*Math.cos(Math.PI-t0+Math.PI*deg/180));
    py[1] = (int)(y + gy + sc*r0*Math.sin(Math.PI-t0+Math.PI*deg/180));
    px[2] = (int)(x + gx + sc*r0*Math.cos(Math.PI+t0+Math.PI*deg/180));
    py[2] = (int)(y + gy + sc*r0*Math.sin(Math.PI+t0+Math.PI*deg/180));
    px[3] = (int)(x + gx + sc*r0*Math.cos(-t0+Math.PI*deg/180));
    py[3] = (int)(y + gy + sc*r0*Math.sin(-t0+Math.PI*deg/180));
    g.setColor(cl); g.fillPolygon(px, py, 4);
  }
}
```

たとえばここまでで、「***」の行を

```
  obj = new GrTriangle(80.0, 0.0, 0.0, 80.0, 0.0, 0.0);
```

に差し替えると「動かない三角形をずっと表示する」アプレットになる。

4.6 ものを中に格納するもの

さて、ここでものを動かすには、たとえば先にやった AnimCircle みたいに速度のインスタンス変数を追加して `setTime()` で位置を計算し直すなどすればできる。しかし、「動かし方」は色々あるわけなのに、そうしてしまうと動かし方を変えるたびにその部分のコードを全部書き直すことになってしまう。

今回はそうする代りに、「ノコギリ状に動く」とか「サイン曲線に従って大きさが変化する」というふうに、特定の形とは結びつかない「動きだけ」をあらわすクラスを用意して、それと各種の形を組み合わせることで、任意の形を任意の動き方で動かせるようにしよう。

その下準備として、単に「描くものを格納するような描くもの」のクラス GrContainer を作成する。

```
class GrContainer extends GrObject {
  GrObject obj;
  public GrContainer(GrObject o) { super(o.getX(), o.getY()); obj = o; }
```

```
  public void moveTo(double x, double y) { obj.moveTo(x, y); }
  public double getX() { return obj.getX(); }
  public double getY() { return obj.getY(); }
  public void setRotation(double d) { obj.setRotation(d); }
  public double getRotation() { return obj.getRotation(); }
  public void setScale(double s) { obj.setScale(s); }
  public double getScale() { return obj.getScale(); }
  public void setColor(Color c) { obj.setColor(c); }
  public Color getColor() { return obj.getColor(); }
  public void initTime(double t) { basetime = t; obj.initTime(t); }
  public void setTime(double t) { obj.setTime(t); }
  public void draw(Graphics g, double x, double y) { obj.draw(g, x, y); }
}
```

見てのとおり、GrContainer はインスタンス変数 obj に「描くもの」を格納しておいて、ほとんどのメソッドをその obj のメソッド呼び出しに中継するようになっている。なんでこんな面倒なことをするのか? それは次を見ていただければ分かる。

4.7 回転する動き

では、一定の角速度で回転する動きを作ってみよう。それにはまず GrContainer を継承したクラスを作り、その中で setTime() が呼ばれたら時刻に応じて内側に保持している図形の角度を変化させる。変数 ddeg は 1 秒あたりの回転角度を表している。

```
class GrRotation extends GrContainer {
  double ddeg;
  public GrRotation(GrObject o, double dd) { super(o); ddeg = dd; }
  public void setRotation(double d) { deg = d; }
  public double getRotation() { return deg; }
  public void setTime(double t) {
    obj.setRotation(deg + ddeg*(t-basetime)); obj.setTime(t);
  }
}
```

この状態で先の「***」の箇所を次のように直すと、回転する三角形のアニメーションが作成できる。

```
  obj = new GrTriangle(80.0, 0.0, 0.0, 80.0, 0.0, 0.0);
  obj = new GrRotation(obj, 30.0);
```

つまり 1 行目で先と同じ三角形を作り、2 行目でそれを「回転する動き」の中に入れるわけである。

4.8 動きの合成

上の例だけだとなんでこんな面倒なことをと思うかも知れないが、動きと形を分離するることで、動きを合成できるようになっている。たとえば「サイン曲線に従ってスケールを変化させる」という動きを作ってみよう。

```
class GrSinScale extends GrContainer {
  double d0, d1, sc1;
  public GrSinScale(GrObject o, double t, double u, double v, double w) {
```

```
    super(o); d0 = t; d1 = u; sc = v; sc1 = w;
  }
  public void setScale(double s) { sc = s; }
  public double getScale() { return sc; }
  public void setTime(double t) {
    double t0 = Math.PI*(d0 + t*d1)/180.0;
    obj.setScale(sc + sc1*Math.sin(t0)); obj.setTime(t);
  }
}
```

これを先の回転と組み合わせるには「***」のところを次のようにする。

```
obj = new GrTriangle(80.0, 0.0, 0.0, 80.0, 0.0, 0.0);
obj = new GrRotation(obj, 30.0);
obj = new GrSinScale(obj, 0.0, 360.0, 1.0, 0.5);
```

これで「回転しながらドキドキする三角形」ができる。もちろん、大きさではなく位置をサイン曲線に応じて変化させることもできる。

```
class GrSinMove extends GrContainer {
  double d0, d1, rx, ry;
  public GrSinMove(GrObject o, double t, double u, double x, double y) {
    super(o); d0 = t; d1 = u; rx = x; ry = y;
  }
  public void moveTo(double x, double y) { gx = x; gy = y; }
  public double getX() { return gx; }
  public double getY() { return gy; }
  public void setTime(double t) {
    double t0 = Math.PI*(d0 + t*d1)/180.0;
    obj.moveTo(gx+rx*Math.sin(t0), gy+ry*Math.sin(t0)); obj.setTime(t);
  }
}
```

また、サイン曲線ではなくノコギリ状に変化してもよい。このように、さまざまな動きもほとんどのコードを継承して少数のメソッドだけを定義しなおすことで簡単に作成できる。

```
class GrSawMove extends GrContainer {
  double vx, vy, lim;
  public GrSawMove(GrObject o, double vx0, double vy0, double l) {
    super(o); vx = vx0; vy = vy0; lim = l;
  }
  public void moveTo(double x, double y) { gx = x; gy = y; }
  public double getX() { return gx; }
  public double getY() { return gy; }
  public void setTime(double t) {
    obj.moveTo(gx+vx*(t%lim), gy+vy*(t%lim)); obj.setTime(t);
  }
}
```

なお「%」は「剰余」演算子でしたね。

4.9 複合図形

さて、ここまででは画面に現われる「もの」は 1 つだけだったが、複数の「もの」が現われて欲しいですね? このため、「複数のものが集まったもの」を表すクラス GrSet を作る。これはある意味では GrContainer に似ているが、ただし内部では GrObject の配列を使って複数の「もの」を収納する (最大いくつまで入れられるかはコンストラクタで指定)。

```
class GrSet extends GrObject {
  GrObject[] a;
  int count;
  public GrSet(double x, double y, int s) {
    super(x, y); a = new GrObject[s]; count = 0;
  }
  public void add(GrObject o) {
    if(count < a.length) { a[count] = o; ++count; }
  }
  public GrObject get(int i) {
    if(i < 0 || i >= a.length) return null; else return a[i];
  }
  public int getCount() {
    return a.length;
  }
  public void setRotation(double d) {
    deg = d; for(int i = 0; i < count; ++i) a[i].setRotation(d);
  }
  public void setScale(double s) {
    sc = s; for(int i = 0; i < count; ++i) a[i].setScale(s);
  }
  public void setColor(Color c) {
    cl = c; for(int i = 0; i < count; ++i) a[i].setColor(c);
  }
  public void initTime(double t) {
    basetime = t; for(int i = 0; i < count; ++i) a[i].initTime(t);
  }
  public void setTime(double t) {
    for(int i = 0; i < count; ++i) a[i].setTime(t);
  }
  public void draw(Graphics g, double x, double y) {
    for(int i = 0; i < count; ++i) a[i].draw(g, x+gx, y+gy);
  }
}
```

add()、get()、getCount() 以外はほとんど、配列 a に保持している各 GrObject にそのまま呼び出しを橋渡しするだけである。

では、これを使って 2 つの「もの」が現われるアプレットにしてみよう。まず、アプレットのインスタンス変数

```
GrObject obj;
```

を GrSet にとりかえる。

```
  GrSet obj;
```

そして「***」のところを次のように直す。

```
    obj = new GrSet(0.0, 0.0, 10);
    GrObject o1 = new GrTriangle(80.0, 0.0, 0.0, 80.0, 0.0, -40.0);
    o1 = new GrSawMove(o1, 30.0, -50.0, 3.0); obj.add(o1);
    GrSet o2 = new GrSet(-80.0, 0.0, 5);
    o1 = new GrTriangle(30.0, 0.0, -30.0, 0.0, 0.0, -30.0);
    o1.setColor(Color.red); o2.add(o1);
    o1 = new GrRectangle(0.0, 20.0, 60.0, 40.0);
    o1.setColor(Color.green); o2.add(o1);
    o1 = new GrSinScale(o2, 0.0, 90.0, 1.0, 0.5);
    o1 = new GrRotation(o1, 30); obj.add(o1);
```

つまり 1 つ目は動く三角形、2 つ目はまた GrSet で、その中には三角形と矩形が入っていて (家の形のつもり)、全体が大きさを変えながら回転している。

しかし実際に動かしてみると、家の箱と屋根がそれぞれ「その場で」動くため、家がバラバラになる。つまり、GrSet はあくまでも「それぞれのものは独立している」という集まりなわけである。

4.10 一緒に回転/拡大縮小するグループ

そこで、GrSet を少し手直しして、回転/拡大を行うとそれぞれのものの位置も回転/拡大される版を作る。それには draw() を差し替えるだけでよい (描くときに、それぞれのものの X/Y 座標を持って来てその回転/拡大位置をおなじみのやり方で計算し、その場所に描画する)。

```
  class GrGroup extends GrSet {
    public GrGroup(double x, double y, int s) { super(x, y, s); }
    public void draw(Graphics g, double x, double y) {
      for(int i = 0; i < count; ++i) {
        double x0 = a[i].getX(); double y0 = a[i].getY();
        double r0 = Math.sqrt(x0*x0 + y0*y0);
        double t0 = Math.atan2(y0, x0) + Math.PI*deg/180;
        a[i].draw(g, x+gx-x0+sc*r0*Math.cos(t0), y+gy-y0+sc*r0*Math.sin(t0));
      }
    }
  }
```

先の例の内側の GrSet を使っている

```
    GrSet o2 = new GrSet(-80.0, 0.0, 5);
```

のところを次のように GrGroup に変更すれば、

```
    GrSet o2 = new GrGroup(-80.0, 0.0, 5);
```

家がバラバラにならずに動くようになる。

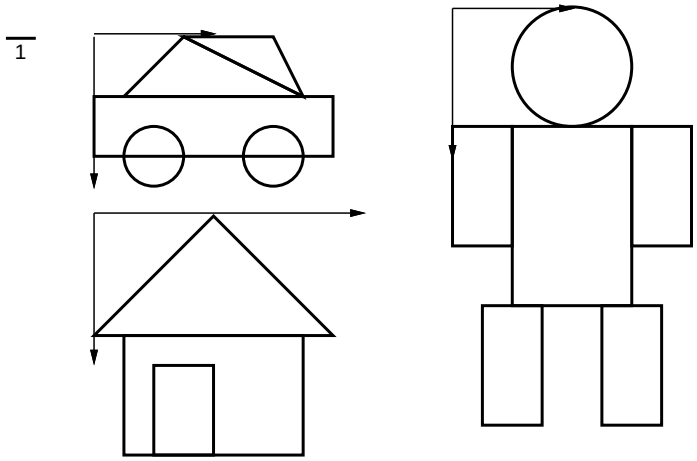

図 4: ごく簡単な家、人、車のかたち

4.11 複合図形をクラスにする

上のやり方でいろいろな絵はできるが、これではいつか通った道? で絵を組み立てるのがとっても面倒である。この問題に対処するには、前と同じく「家」とか「人」とかをクラスにしてしまえばよい。今回の場合は、せっかく GrSet を作ったのだから、そのような複合図形は GrSet のサブクラスにしてコンストラクタで絵を「組み立てる」だけでよい。ただし絵柄の各所で色が同じだといまいちなので、色も適宜設定するようにした。

図 4 に簡単な「家」「人」「車」のデザインを示す。この 1 単位の長さを指定してそれぞれの絵を作ることにしよう。まずは「家」から。

```
class GrHouse extends GrGroup {
  public GrHouse(double x, double y, double u) {
    super(x, y, 3);
    this.add(new GrTriangle(-4*u, 4*u, 4*u, 4*u, 0.0, 0.0));
    this.add(new GrRectangle(0.0, 6*u, 6*u, 4*u));
    this.add(new GrRectangle(-1*u, 6.5*u, 2*u, 3*u));
    this.setColor(new Color(0, 200, 100));
  }
  public void setColor(Color c) {
    this.get(0).setColor(c.brighter());
    this.get(1).setColor(c);
    this.get(2).setColor(c.darker()); cl = c;
  }
}
```

部品の個数は絵柄ごとに決まっているので、コンストラクタで親クラスのコンストラクタを呼び出すときその数を指定する。そしてあとは自分自身 (this) の add() で部品を追加すればよい。setColor() では 3 つの部品にそれぞれ適切な色を配置している。もう 1 つ、「人」も見てみよう。

```
class GrHuman extends GrGroup {
  public GrHuman(double x, double y, double u) {
    super(x, y, 6);
    this.add(new GrCircle(4*u, 2*u, 2*u));
    this.add(new GrRectangle(4*u, 7*u, 4*u, 6*u));
    this.add(new GrRectangle(1*u, 6*u, 2*u, 4*u));
    this.add(new GrRectangle(7*u, 6*u, 2*u, 4*u));
```

```
      this.add(new GrRectangle(2*u, 12*u, 2*u, 4*u));
      this.add(new GrRectangle(6*u, 12*u, 2*u, 4*u));
      this.setColor(new Color(200, 100, 0));
    }
    public void setColor(Color c) {
      this.get(0).setColor(c.brighter());
      this.get(1).setColor(c.darker()); cl = c;
      for(int i = 2; i < 6; ++i) this.get(i).setColor(c);
    }
  }
```

ちょっと部品が多いだけでほとんど同じですね? では「***」の部分を次のように差し替えて動かしてみよう。

```
      obj = new GrSet(0.0, 0.0, 10);
      GrObject o1 = new GrHuman(0.0, 0.0, 5.0);
      o1 = new GrSawMove(o1, 30.0, -30.0, 3.0); obj.add(o1);
      GrObject o2 = new GrHouse(-80.0, 0.0, 7.0);
      o2 = new GrSinScale(o2, 0.0, 90.0, 1.0, 0.5);
      o2 = new GrRotation(o2, 30); obj.add(o2);
```

「斜めに動く人と回転する家」ができあがる。

4.12 複合図形のカタチも動かす

しかし、いかに回転や移動や拡大ができても、カタチが変わらないといまいちである。実はカタチを変えるには、図形を組み立てるときに「動き」で包んでやるだけでよい。たとえば「タイヤが上下に動く自動車」を作ってみよう。

```
  class GrAuto extends GrGroup {
    public GrAuto(double x, double y, double u) {
      super(x, y, 5);
      this.add(new GrTriangle(-1*u, -2*u, 2*u, -2*u, 3*u, 0.0));
      this.add(new GrTriangle(-3*u, 0.0, 3*u, 0.0, -1*u, -2*u));
      this.add(new GrRectangle(0.0, 1*u, 8*u, 2*u));
      this.add(new GrSinMove(new GrCircle(-2*u,2*u,u),0.0,90.0,0.0,0.5*u));
      this.add(new GrSinMove(new GrCircle(2*u,2*u,u),0.0,90.0,0.0,-0.5*u));
      this.setColor(new Color(150, 0, 150));
    }
    public void setColor(Color c) {
      this.get(0).setColor(c.brighter());
      this.get(1).setColor(c.brighter());
      this.get(2).setColor(c);
      this.get(3).setColor(c.darker());
      this.get(4).setColor(c.darker());
    }
  }
```

これでさっきの後に次のようなコードを追加すると「でこぼこ道を走る車 (?)」が画面を横切るようになる。

```
      GrObject o3 = new GrAuto(150.0, 0.0, 10.0);
      o3 = new GrSawMove(o3, -25.0, -10.0, 10.0); obj.add(o3);
```

どうでしたか?

なお、この先はたとえば「ある時刻からある時刻までだけ画面に現われる絵」とか「ある時刻からこういう動き、次の時刻からこういう動き、…で動く絵」といった部品を追加していけば、最後はアプレットの画面で「桃太郎の鬼退治」とかまで作れるかも知れない (まさか?!)。というわけで、「最終レポート」の課題 (複数選択) のうちの 1 つは「ストーリのあるアニメーション紙芝居アプレットを作れ」ということになる予定である。

演習 4 上記のソースコード一式を持って来て動かせ。ただし `init()` の中は「何もしない」状態なので適宜直してみること。

演習 5 上記のソースを拡張して、新しい動き方を追加してみよ。または新しい (複合) 図形を追加してみよ。