ヒューマンインタフェース'99 ＃5

久野　靖*

1999.5.14

はじめに

今回で最終回ということで、お約束の Java アプレットを使ったグラフィクスとアニメーションを取り上げることにする。1 回しかないのでちょっと駆け足になると思うけどよろしく…

1　再掲: アプレット入門

「計算機科学基礎」でも Java とアプレットについて説明したが、はじめての人むけにごく簡単に説明しておこう。まず、Java というのはプログラミング言語の中でも「オブジェクト指向言語」と呼ばれる範疇に入る言語である。

オブジェクト指向とはソフトウェアを「もの」に注目して設計し製作するような流儀であり、オブジェクト指向言語はそのような流儀をサポートするようなプログラミング言語ということになる。オブジェクト指向言語にもいろいろな流派があるが、Java は最も一般的な「クラス方式のオブジェクト指向言語」流派の中の「強い型を持つ」言語の代表の 1 つ (もう 1 つの代表が C++)。

クラス方式の言語では、「クラス」が「ものの種類」を表し、ものの分類 (階層) 構造をクラスの継承関係 (親子関係) で表す。たとえば「動物」というクラスの子クラスとして「人間」とか「犬」とかを作る。このとき、親クラスから子クラスへ変数定義やメソッド (手続き) 定義が継承されるので、そこで足りない部分を追加したり、継承したものでは不十分なメソッドを差し替えることで子クラスを完成させる。

アプレットはすべて、親クラスとして Applet クラスを持ち、アプレットとして振る舞うのに必要なことがらはここで用意されている。だから自分独自のアプレットを作るには Applet の子クラスを作り、その中に変数を追加したりメソッド (典型的には絵を描くメソッド paint()) を差し替え (オーバライド) したりする。このように、作りたいものの大枠はあらかじめ用意しておいて、プログラム固有の部分をちょっとだけ差し替えて自分独自のプログラムに仕立てるという考え方を「アプリケーションフレームワーク」と呼ぶ。

あんまりごたくを並べていてもうんざりするので、まず簡単なアプレットを作って動かして見ていただこう。アプレットのプログラムを示す。

*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 AppSam1 extends Applet {
  Color c0 = Color.getHSBColor(0.7f, 0.5f, 1.0f);
  public void paint(Graphics g) {
    g.setColor(c0);
    g.fillOval(20, 20, 100, 100);
  }
}
```

まず、アプレットは上述のようにjava.lang.Appletのサブクラスになるので、これをimport(いちいち長い名前で指定しなくても使えるように準備) する。また、グラフィクスその他の画面機能はすべてjava.awt. ○○というクラスにあるので、これらもまとめてimport する。

クラスそのものは、アプレットとして使う場合はpublic(どこからでも参照可能) でなければならない。その中でColor オブジェクト (もの) を格納する変数c0 を定義し、Color クラスに付随するメソッド (クラスメソッド) を呼んで指定した色相/彩度/明度を指定してそのような色オブジェクトを作ってもらい、その色を格納する。

演習 Color クラスのAPI を眺めておこう。

さて、アプレットが動き出して画面表示する必要が生じると、アプレットオブジェクトのメソッドpaint() が呼び出される。その引数g にはGraphics オブジェクトが渡されるが、このオブジェクトに対するメソッドを呼ぶことでさまざまな形が描画できる。ここではfillOval() で円を描いている。paint() やfillOval() などは「オブジェクトに付随する」メソッドでありインスタンスメソッドと呼ばれる。

演習 Graphics クラスのAPI を眺めておこう。

ではこのアプレットを動かしてみよう。

(1) ディレクトリWWW(またはその下でアプレット群を置いてもいいなと思うディレクトリ) に移動する。以下では作成するファイルが全部「誰にでも読める」モードであって欲しいので、

```
% umask 022   ← 1 回だけでよい
```

を実行しておく。また、javaのコンパイラとしてJDK 1.2 のものを使うため

```
% export PATH=/usr/local/libproj/jdk12/bin:$PATH   ← 1 回だけでよい
```

も実行しておく

(2) アプレットを入れるHTML ファイルを作る。ここでは最低限の次のようなものにしておこう。

```
<HTM><HEAD><TITLE>AppSam1</TITLE></HEAD><BODY>
<H1>AppSam1</H1>
<APPLET CODE="AppSam1.class" WIDTH=300 HEIGHT=200>
</APPLET>
</BODY></HTML>
```

(4) 上記の AppSam1.java を打ち込み、次の手順でコンパイルする

```
% javac AppSam1.java
```

エラーが出たら直して再度。

これですべて完了、ブラウザで眺めるとアプレットが動くはずだ (おめでとう!)。

演習 1 上記のアプレットをそのまま動かせ。

演習 2 円の色を変えてみよ。

演習 3 円以外の図形も描いてみよ (Graphics クラスのメソッドを調べる)

2 アプレットのオーバライド用メソッド群

さて、アプレットの外から呼ばれる (フレームワークの一部をなす) メソッドは paint() の他にも次のものがある。

```
public void init();  --- アプレットの初期化用
public void start(); --- アプレットの実行開始時用
public void stop();  --- アプレットの停止用
public void destroy(); --- アプレットの破棄用
```

とくに init() は、インスタンス変数の初期化が式 1 つでは書けない場合に初期化動作を記述するのに使う。残りについては当面は考えなくてよい (そのうちまた出てくる)。

これを使って、単色ではなくいくつかの色調で重なった円を描くようにしてみよう。

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

public class AppSam2 extends Applet {
  Color[] colors = new Color[10];
  public void init() {
    for(int i = 0; i < 10; ++i) {
      float v = (float)i/9.0f;
      colors[i] = new Color(v, v, 1.0f-v);
    }
  }
  public void paint(Graphics g) {
```

```
    for(int i = 0; i < 10; ++i) {
      g.setColor(colors[i]);
      g.fillOval(200-i*20, 20+i*10, 40, 40);
    }
  }
}
```

なお、Java では配列は 1 つの型で、いくつの要素を持つかは new でパラメタとして指定する。配列以外のオブジェクトも、多くはクラスメソッドではなく new で生成する (どちらでもよいが)。new で生成するときは、初期設定のためにコンストラクタと呼ばれる特別なメソッドが呼ばれる。

ここで呼んでいる「new Color(float, float, float)」は赤、緑、青の 3 原色の値を 0.0f ～1.0f の float の値で指定し、その色調の色オブジェクトを生成するコンストラクタである。そして、それらの色を配列 colors に蓄えておき、paint() の中ではこれらの色に切替えてから順次同じ文字列をずらして描いている。

別に配列を使う必要はなかったのだけど、init() と配列を使う例題として入れてみました。

3 イベント処理

では、前回 X Window でやったイベント処理を Java ではどうするかについて簡単に見ておこう (計算機科学基礎に載っているのは JDK 1.0.2 の例なので古い)。

イベントを処理するには、イベントリスナオブジェクトを生成して、イベントが発生する窓なり部品なりにそれを登録する。イベントが発生すると、イベントごとに決まった名前のイベントリスナオブジェクトのメソッドが呼ばれるので、そのメソッド内で処理を行えばよい。

昔はこうではなく、窓自身の決まった名前のオブジェクトが呼ばれていたが、それだと違った動作をさせたければ窓そのものをサブクラス化するなどの必要があり大変だった。たとえばボタンの動作が違うごとに別のボタンのサブクラスを作るのでは面倒すぎる。新しい方法では、ボタンクラスは 1 つのまま、ボタンごとに別のリスナオブジェクトを作ればよい。

ここで、イベントリスナオブジェクトを毎回別のファイルに書くのでは大変なので、これまた新しい「無名クラス」という機構を使うことで簡単に書く。例を見ていただこう。

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

public class AppSam3 extends Applet {
  Point pts[] = new Point[100];
  int count = 0;
  public void init() {
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent e) {
        addPoint(e.getX(), e.getY());
      }
    });
```

```
  }
  public void addPoint(int x, int y) {
    if(count+1 < 100) pts[count++] = new Point(x, y);
    repaint();
  }
  public void paint(Graphics g) {
    g.setColor(Color.blue);
    for(int i = 0; i < count; ++i) {
      g.fillOval(pts[i].x-20, pts[i].y-20, 40, 40);
    }
  }
}
```

「new MouseAdapter() ... 」というのが、MouseAdapter クラス (マウス用イベントリスナのひな型クラス) の無名のサブクラスを定義して、そのインスタンスを生成する動作になっている。この無名クラスの中では、mousePressed() だけに関心があるので、このメソッドだけをオーバライドしている (他のメソッドは何もしない)。

演習 4 この例題を手直しして、円は 1 つだけで、マウスボタンが押されるたびに円がその位置に動くようにしてみよ。また、マウスカーソルと関係なく、クリックごとに円がちょっとずつ一定方向に移動するようにしてみよ。

演習 5 さらに前問の結果を手直しして、1 つの円ではなく、複数の図形がそれぞれ勝手な方向に移動するようにしてみよ (ヒント: クリックした回数を変数に数えておき、それをもとに各物体の位置を計算する)。

4 オブジェクト指向による構造化グラフィクス

さて、上のアプレットでは図形をいきなり描いていたが、図形も「画面上に現われるもの」であり、これもクラスとして扱う方がより美しい/ 後で拡張しやすいプログラムになる (大きさは大きくなるが…)

そこでクラス階層を設計することにして、まず「描くことのできるもの全般」を表す抽象クラス GrObject を作る。

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

abstract class GrObject {
  Color c0 = Color.black;
  int x0, y0;
  public GrObject(int x, int y) { x0 = x; y0 = y; }
  public void setColor(Color c) { c0 = c; }
  abstract void drawBody(Graphics g);
  public void draw(Graphics g) { g.setColor(c0); this.drawBody(g); }
}
```

すべての図形に共通する機能として「色」と「位置 (X 座標、Y 座標)」があるものと考え、これをインスタンス変数として持たせる。コンストラクタでは座標のみ指定すればよく、色は最初は黒で、setColor() で自由に変更できる。

図形を描くには draw() を呼ぶが、この中では色をこの図形の色に設定してから drawBody() という抽象メソッドを呼ぶ。こちらが実際に各サブクラスで個別の図形を描くようにオーバライドするメソッドである。たとえば矩形について見てみよう。

```
class GrRect extends GrObject {
  int w0, h0;
  public GrRect(int x, int y, int w, int h) {
    super(x, y); w0 = w; h0 = h;
  }
  public void drawBody(Graphics g) {
    g.fillRect(x0, y0, w0, h0);
  }
}
```

矩形では幅と高さも必要なので、それらを覚えるインスタンス変数が追加される (親クラス GrObject のインスタンス変数もそのまま継承されていることに注意)。コンストラクタでは、X 座標と Y 座標については親クラスのコンストラクタを呼んで設定したい。そのために「super(...)」という書き方ができるようになっている。あとの変数は自分で初期設定する。drawBody() は簡単で、要するに Graphics クラスの描画メソッドを呼ぶだけである。

その他の図形も同様なのでまあ見ておこう。

```
class GrOval extends GrObject {
  int w0, h0;
  public GrOval(int x, int y, int w, int h) {
    super(x, y); w0 = w; h0 = h;
  }
  public void drawBody(Graphics g) {
    g.fillOval(x0, y0, w0, h0);
  }
}

class GrLine extends GrObject {
  int dx, dy;
  public GrLine(int x0, int y0, int x1, int y1) {
    super(x0, y0); dx = x1-x0; dy = y1-y0;
  }
  public void drawBody(Graphics g) {
    g.drawLine(x0, y0, x0+dx, y0+dy);
  }
}

class GrTriangle extends GrObject {
```

```
  int dx1, dy1, dx2, dy2;
  public GrTriangle(int x0, int y0, int x1, int y1, int x2, int y2) {
    super(x0, y0); dx1 = x1-x0; dy1 = y1-y0; dx2 = x2-x0; dy2 = y2-y0;
  }
  public void drawBody(Graphics g) {
    int[] x = new int[3]; x[0] = x0; x[1] = x0+dx1; x[2] = x0+dx2;
    int[] y = new int[3]; y[0] = y0; y[1] = y0+dy1; y[2] = y0+dy2;
    g.fillPolygon(x, y, 3);
  }
}
```

最後にアプレットのクラスであるが、4つの図形オブジェクトを生成し、それらの色を適宜設定している。初期設定を行いたい場合は、前回やったように init() をオーバライドしてその中で行う。あとは paint() の中で各図形に「描け!」と言えばそれらが画面に現われる。なお、setBackgroundColor() は背景の色を設定するためのメソッドである。

```
public class AppSam7c extends Applet {
  GrObject g0 = new GrRect(10, 10, 100, 40);
  GrObject g1 = new GrOval(40, 80, 100, 40);
  GrObject g2 = new GrTriangle(80, 180, 180, 180, 120, 40);
  GrObject g3 = new GrLine(0, 0, 300, 200);
  public void init() {
    this.setBackground(Color.black);
    g0.setColor(Color.getHSBColor(0.7f, 0.5f, 1.0f));
    g2.setColor(Color.getHSBColor(0.3f, 0.5f, 1.0f));
    g1.setColor(Color.getHSBColor(0.9f, 0.5f, 1.0f));
    g3.setColor(Color.getHSBColor(0.1f, 0.5f, 1.0f));
  }
  public void paint(Graphics g) {
    g0.draw(g); g1.draw(g); g2.draw(g); g3.draw(g);
  }
}
```

5　複合図形オブジェクト

さて、このような単純な図形オブジェクトだけを大量に使って絵を組み立てるのはあんまり嬉しくない。もっと高レベルな、「家」とか「車」といった図形オブジェクトが欲しいですね?

実はこれは簡単で、そのような図形オブジェクトは単純な図形の組合せなのだから、中にそれぞれの部品オブジェクトをインスタンス変数として持てばよい。たとえば図1のような感じのものを作とすると、「原点位置と単位となる大きさ」を渡してもらって、必要な部品オブジェクトを作っていけばよい。

たとえば図1の「家」に相当するクラスを作ってみよう。

```
class GrHouse extends GrObject {
```

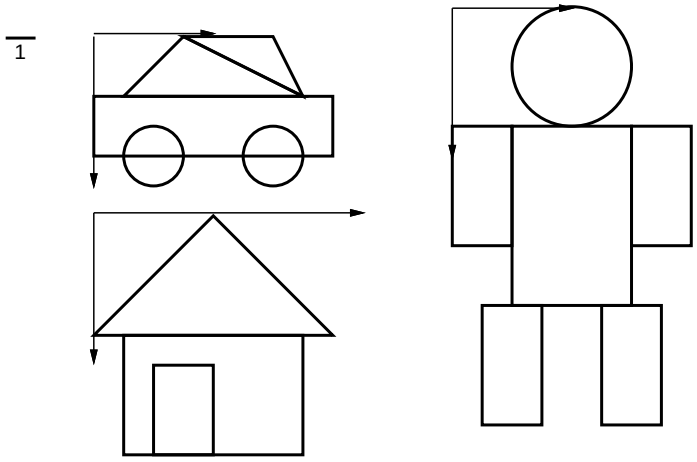

図 1: 車、人、家のデザイン

```
  GrObject roof, body, door;
  int unit = 10;
  public GrHouse(int x0, int y0, int u0) {
    super(x0, y0); unit = u0;
    roof = new GrTriangle(x0, y0+u0*4, x0+u0*8, y0+u0*4, x0+u0*4, y0);
    body = new GrRect(x0+u0, y0+u0*4, u0*6, u0*4);
    door = new GrRect(x0+u0*2, y0+u0*5, u0*2, u0*3);
  }
  public void setColor(Color c0) {
    roof.setColor(c0);
    body.setColor(c0.brighter()); door.setColor(c0.darker());
  }
  public void drawBody(Graphics g) {
    roof.draw(g); body.draw(g); door.draw(g);
  }
}
```

なお、色の設定のときに、Color オブジェクトのメソッドに「より明るい色」「より暗い色」というのがあるので、これらを使って屋根と本体とドアの色をちょっとずつ変えている。アプレットも示そう。

```
public class AppSam7d extends Applet {
  GrObject g0 = new GrHouse(50, 50, 10);
  GrObject g1 = new GrHouse(150, 60, 14);
  public void init() {
    this.setBackground(Color.black);
    g0.setColor(Color.getHSBColor(0.7f, 0.5f, 1.0f));
    g1.setColor(Color.getHSBColor(0.3f, 0.5f, 1.0f));
  }
  public void paint(Graphics g) {
```

```
    g0.draw(g); g1.draw(g);
  }
}
```

演習 6 この例題のコードをコピーして来てそのまま動かせ。動いたら、今度は新しい複合オブジェクトのクラスを追加してみよ。アイデアがでなければ図 1 の「車」や「人」でもよい。

6 移動する図形

ここまでの絵はアプレットなのに表示されたらそれきりで「動き」がなかった。これではつまらないので、マウスでクリックするごとに動く絵を作ってみよう。それにはまず、各図形に「速度」を与えることにする。といっても、X 方向と Y 方向の移動量を持たせ、move() というとその分だけ移動する、というだけである。

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

abstract class GrObject {
  Color c0 = Color.black;
  int x0, y0, vx = 0, vy = 0;
  public GrObject(int x, int y) { x0 = x; y0 = y; }
  public void setColor(Color c) { c0 = c; }
  public void setSpeed(int vx0, int vy0) { vx = vx0; vy = vy0; }
  public void move() { x0 += vx; y0 += vy; }
  abstract void drawBody(Graphics g);
  public void draw(Graphics g) { g.setColor(c0); this.drawBody(g); }
}
```

基本図形についてはこれまでと全く変わっていないが再掲しておく。

```
class GrRect extends GrObject {
  int w0, h0;
  public GrRect(int x, int y, int w, int h) {
    super(x, y); w0 = w; h0 = h;
  }
  public void drawBody(Graphics g) {
    g.fillRect(x0, y0, w0, h0);
  }
}

class GrOval extends GrObject {
  int w0, h0;
  public GrOval(int x, int y, int w, int h) {
```

```
    super(x, y); w0 = w; h0 = h;
  }
  public void drawBody(Graphics g) {
    g.fillOval(x0, y0, w0, h0);
  }
}

class GrLine extends GrObject {
  int dx, dy;
  public GrLine(int x0, int y0, int x1, int y1) {
    super(x0, y0); dx = x1-x0; dy = y1-y0;
  }
  public void drawBody(Graphics g) {
    g.drawLine(x0, y0, x0+dx, y0+dy);
  }
}

class GrTriangle extends GrObject {
  int dx1, dy1, dx2, dy2;
  public GrTriangle(int x0, int y0, int x1, int y1, int x2, int y2) {
    super(x0, y0); dx1 = x1-x0; dy1 = y1-y0; dx2 = x2-x0; dy2 = y2-y0;
  }
  public void drawBody(Graphics g) {
    int[] x = new int[3]; x[0] = x0; x[1] = x0+dx1; x[2] = x0+dx2;
    int[] y = new int[3]; y[0] = y0; y[1] = y0+dy1; y[2] = y0+dy2;
    g.fillPolygon(x, y, 3);
  }
}
```

問題は複合図形で、速度を設定したり動かしたりは個々の部品全部に対して伝播させないといけない。ここではちょっと「イタズラ」してドアの速度だけ変えてみた。

```
class GrHouse extends GrObject {
  GrObject roof, body, door;
  int unit = 10;
  public GrHouse(int x0, int y0, int u0) {
    super(x0, y0); unit = u0;
    roof = new GrTriangle(x0, y0+u0*4, x0+u0*8, y0+u0*4, x0+u0*4, y0);
    body = new GrRect(x0+u0, y0+u0*4, u0*6, u0*4);
    door = new GrRect(x0+u0*2, y0+u0*5, u0*2, u0*3);
  }
  public void setColor(Color c0) {
    roof.setColor(c0);
    body.setColor(c0.brighter()); door.setColor(c0.darker());
```

```
  }
  public void setSpeed(int vx, int vy) {
    roof.setSpeed(vx, vy); body.setSpeed(vx, vy); door.setSpeed(-vx, -vy);
  }
  public void move() {
    roof.move(); body.move(); door.move();
  }
  public void drawBody(Graphics g) {
    roof.draw(g); body.draw(g); door.draw(g);
  }
}
```

最後にアプレットクラスだが、いちいち g0、g1、…という変数を作るのもしんどくなったので配列にしよう。

```
public class AppSam7e extends Applet {
  GrObject[] obj = new GrObject[5];
  public void init() {
    this.setBackground(Color.black);
    obj[0] = new GrRect(10, 10, 100, 40);
    obj[1] = new GrOval(40, 80, 100, 40);
    obj[2] = new GrTriangle(80, 180, 180, 180, 120, 40);
    obj[3] = new GrLine(0, 0, 300, 200);
    obj[4] = new GrHouse(50, 50, 10);
    obj[0].setColor(Color.getHSBColor(0.7f, 0.5f, 1.0f));
    obj[1].setColor(Color.getHSBColor(0.9f, 0.5f, 1.0f));
    obj[2].setColor(Color.getHSBColor(0.3f, 0.5f, 1.0f));
    obj[3].setColor(Color.getHSBColor(0.1f, 0.5f, 1.0f));
    obj[4].setColor(Color.getHSBColor(0.5f, 0.4f, 1.0f));
    obj[0].setSpeed(3, 5);
    obj[1].setSpeed(7, 1);
    obj[2].setSpeed(-2, 4);
    obj[3].setSpeed(-4, -5);
    obj[4].setSpeed(2, 5);
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent e) { doMove(); }
    });
  }
  public void doMove() {
    for(int i = 0; i < 5; ++i) obj[i].move();
    repaint();
  }
  public void paint(Graphics g) {
    for(int i = 0; i < 5; ++i) obj[i].draw(g);
```

```
    }
  }
```

マウスがクリックされると、すべての図形オブジェクトに「動け」と言ってから画面を再描画する。そうすると、各図形が新しい位置に描かれるわけである。なかなか楽しいでしょう?

演習 7 実は上のプログラム構造には重複がある。もっと美しいプログラム構造にならないものか検討せよ。

7 Vector クラスと Enumeration

先の演習問題の答えは何でしょう? つまり、家も人も車も、そして最後にアプレットが描く画面もすべて、「いくつかの GrObject が集まったもの」という点では同じなのだから、そのような共通機能はくくり出した方がよい、という趣旨だったのですが…

ただ、それを実現しようとすると問題がちょっとあって、「何個か分からない GrObject を格納する」という機能が必要になる。配列は「最初に個数を決めて作る」ので、このような目的にはあまり向いていない (多めに領域を取っておいたり、思ったより多くなりすぎたら新しく大きいものを取りなおすという手はあるが面倒である)。

実はこのような目的のためには java.util.Vector クラスが適している。このクラスのオブジェクトは配列のように複数の値 (ただしオブジェクトの値に限られる) を保持するが、その個数は決まっていない。主な使い方を示しておこう。

```
new Vector()        --- Vector オブジェクトを作る
addElement(Object) --- 要素を 1 つ追加する
elementAt(int)      --- N 番目の要素を取り出す
size()              --- 現在何個の要素が入っているかを返す
elements() --- 全要素を順に取り出す Emumeration オブジェクトを返す
```

最後のがちょっと分かりにくいが、順番に説明しよう。まず、いくつかの要素を持つ Vector オブジェクトは次のようにして作れる。

```
Vector v = new Vector();
v.add(...); v.add(...); ...
```

では次に、その要素を順番に取り出して来るには?

```
for(int i = 0; i < v.size(); ++i) {
  Object o = v.elementAt(i); ...
}
```

これでもいいのだが、Java では次のように書く方がかっこいい。

```
for(Enumeration e = v.elements(); e.hasMoreElements(); ) {
  Object o = e.nextElement(); ...
}
```

つまり、Enumerationオブジェクトというのは「順番に値を取り出してくる機能を持つオブジェクト」であり、Vectorに頼んで各要素を順に取り出して来るEnumerationを作ってもらい、それに対して「まだあればその要素を取って処理する」というのを繰り返すわけである。

演習 java.utlパッケージにあるVectorとEnumerationのAPIを調べてみよう。

では以上の知識に基づいて、「複合図形」クラスをまず用意し、家はそのサブクラスに、そしてアプレットの方もこのクラスを利用するように直してみよう。まず先頭部分はVectorのためにjava.util.*をインポートしているのが違うだけであとは同じ。

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

abstract class GrObject {
  Color c0 = Color.black;
  int x0, y0, vx = 0, vy = 0;
  public GrObject(int x, int y) { x0 = x; y0 = y; }
  public void setColor(Color c) { c0 = c; }
  public void setSpeed(int vx0, int vy0) { vx = vx0; vy = vy0; }
  public void move() { x0 += vx; y0 += vy; }
  abstract void drawBody(Graphics g);
  public void draw(Graphics g) { g.setColor(c0); this.drawBody(g); }
}

class GrRect extends GrObject {
  int w0, h0;
  public GrRect(int x, int y, int w, int h) {
    super(x, y); w0 = w; h0 = h;
  }
  public void drawBody(Graphics g) {
    g.fillRect(x0, y0, w0, h0);
  }
}

class GrOval extends GrObject {
  int w0, h0;
  public GrOval(int x, int y, int w, int h) {
    super(x, y); w0 = w; h0 = h;
  }
  public void drawBody(Graphics g) {
    g.fillOval(x0, y0, w0, h0);
  }
}
```

```
class GrLine extends GrObject {
  int dx, dy;
  public GrLine(int x0, int y0, int x1, int y1) {
    super(x0, y0); dx = x1-x0; dy = y1-y0;
  }
  public void drawBody(Graphics g) {
    g.drawLine(x0, y0, x0+dx, y0+dy);
  }
}

class GrTriangle extends GrObject {
  int dx1, dy1, dx2, dy2;
  public GrTriangle(int x0, int y0, int x1, int y1, int x2, int y2) {
    super(x0, y0); dx1 = x1-x0; dy1 = y1-y0; dx2 = x2-x0; dy2 = y2-y0;
  }
  public void drawBody(Graphics g) {
    int[] x = new int[3]; x[0] = x0; x[1] = x0+dx1; x[2] = x0+dx2;
    int[] y = new int[3]; y[0] = y0; y[1] = y0+dy1; y[2] = y0+dy2;
    g.fillPolygon(x, y, 3);
  }
}
```

では、「複数の図形を組み合わせた図形オブジェクト」を定義する。基本的には、要素を「追加する」のと、あとは各設定メソッド等はその要素図形に対してそれぞれ呼ぶ、というだけである。

```
class GrSet extends GrObject {
  Vector vec = new Vector();
  public GrSet(int x0, int y0) { super(x0, y0); }
  public void add(GrObject o) { vec.addElement(o); }
  public void setColor(Color c0) {
    for(Enumeration e = vec.elements(); e.hasMoreElements(); ) {
      GrObject o = (GrObject)e.nextElement(); o.setColor(c0);
    }
  }
  public void setSpeed(int vx, int vy) {
    for(Enumeration e = vec.elements(); e.hasMoreElements(); ) {
      GrObject o = (GrObject)e.nextElement(); o.setSpeed(vx, vy);
    }
  }
  public void move() {
    for(Enumeration e = vec.elements(); e.hasMoreElements(); ) {
      GrObject o = (GrObject)e.nextElement(); o.move();
```

```
    }
  }
  public void drawBody(Graphics g) {
    for(Enumeration e = vec.elements(); e.hasMoreElements(); ) {
      GrObject o = (GrObject)e.nextElement(); o.draw(g);
    }
  }
}
```

ではこれを使って「家クラス」を書き直してみよう。単にコンストラクタで要素を追加して行くだけでいいのだが、色が全部同じだとつまらないので setColor は屋根、壁、ドアが別の色になるようにオーバライドしている。

```
class GrHouse extends GrSet {
  public GrHouse(int x0, int y0, int u0) {
    super(x0, y0);
    this.add(new GrTriangle(x0, y0+u0*4, x0+u0*8, y0+u0*4, x0+u0*4, y0));
    this.add(new GrRect(x0+u0, y0+u0*4, u0*6, u0*4));
    this.add(new GrRect(x0+u0*2, y0+u0*5, u0*2, u0*3));
  }
  public void setColor(Color c0) {
    ((GrObject)vec.elementAt(0)).setColor(c0);
    ((GrObject)vec.elementAt(1)).setColor(c0.brighter());
    ((GrObject)vec.elementAt(2)).setColor(c0.darker());
  }
}
```

ではこれを呼び出すアプレットクラスを示そう。アプレットクラスもその中の図形群は「図形の集まり」なので、GrSet オブジェクトを利用している。GrSet オブジェクトは一見抽象クラスで良さそうだが、実はこのような使い方もできるように普通のクラスにしてあったわけである。

```
public class AppSam8a extends Applet {
  GrSet set = new GrSet(0, 0);
  public void init() {
    this.setBackground(Color.black);
    GrObject g = new GrRect(10, 10, 100, 40); g.setSpeed(3, 5);
    g.setColor(Color.getHSBColor(0.7f, 0.5f, 1.0f)); set.add(g);
    g = new GrOval(40, 80, 100, 40); g.setSpeed(7, 1);
    g.setColor(Color.getHSBColor(0.9f, 0.5f, 1.0f)); set.add(g);
    g = new GrTriangle(80, 180, 180, 180, 120, 40); g.setSpeed(-2, 4);
    g.setColor(Color.getHSBColor(0.3f, 0.5f, 1.0f)); set.add(g);
    g = new GrLine(0, 0, 300, 200); g.setSpeed(-4, -5);
    g.setColor(Color.getHSBColor(0.1f, 0.5f, 1.0f)); set.add(g);
    g = new GrHouse(50, 50, 10); g.setSpeed(2, 5);
    g.setColor(Color.getHSBColor(0.5f, 0.4f, 1.0f)); set.add(g);
```

```
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent e) { doMove(); }
    });
  }
  public void doMove() {
    set.move(); repaint();
  }
  public void paint(Graphics g) {
    set.draw(g);
  }
}
```

8 イメージ

さて、ここまでで直線、楕円、多角形 (三角形)、矩形を組み合わせた図形なら作れるようになったが、これで十分という気はあまりしない。たとえば「もやもやと周囲が霞んだ雲」とか、「左から右へ行くほど暗い色になる円」とかはこれまでの方法では全然無理そうでしょう? また、単純な形でも「ドーナツ」はお手上げですね。

そこで、これまでとはやり方を変えて、「画面上の個々の点 (ピクセル) を制御する」絵の作り方を学ぶことにしよう。これを計算機の世界では「図形」と対照させて「イメージ (画像)」と呼ぶ。

イメージの基本的な考え方は、矩形の領域をまず用意し、それを縦横に並んだ点 (ピクセル --- pixel) に分解して、それぞれの色を自由に設定することにある。Java では 1 つのピクセルを int の値として表現するようなイメージ機構があるので、これを使うことにする。すると、たとえば幅が W ピクセル、高さが H ピクセルの画像であれば $W \times H$ 個の int の要素を持つ配列で表せるわけである。

たとえば、10 × 5 の画像を pix という名前の整数配列に入れると、まず最初の 1 列目が pix[0] 〜pix[9]、2 列目が pix[10] 〜pix[19]、…となり、最後の列は pix[40] 〜pix[49] ということになる (図 2)。

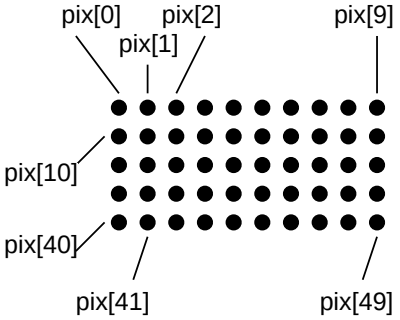

図 2: イメージの整数配列への割り当て

次に個々のピクセルの表現だが、1 つの整数は 32 ビットのビット列として表現されているので、これを 8 ビットずつに分けて赤、緑、青の成分を図 3 のように入れている。前回、各色の

明るさを整数の 0〜255 で表す方法もあると延べたが、これは実は各色の明るさをこのように 8 ビットで表すためだったのだ ($2^8 = 256$)。

図 3: 1 つのピクセルの表現

さらに alpha と記した部分があるが、これは「透明度」を表す。つまり、イメージを描画するとき、前回までの説明ではすべて「重ね塗り」すると後の色だけが有効になっていたが、これを「前の色と後の色を適当な比率で混ぜた結果にする」ことができる。つまり、α=255 であれば前回まで同様後の色だけが有効、α=0 であれば前の色だけが有効だからその部分については「完全に透明」な色になるわけである。

ではこれを使った絵の要素をこれまでのクラス群に付け加えてみよう。まず、このために使うクラス群は java.awt.image パッケージの中にあるので import を追加する。

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

GrObject から GrHouse までは先のリストとまったく同じなので略す。そして、GrObject のサブクラスとしてイメージを使った描画を行うためのクラス GrImage を用意する。

```
class GrImage extends GrObject {
  int w0, h0;
  int[] pix;
  Image img = null;
  Component win;
  public GrImage(int x, int y, int w, int h, Component c) {
    super(x, y); w0 = w; h0 = h; pix = new int[w*h]; win = c;
    for(int i = 0; i < w*h; ++i) { pix[i] = 0; }
  }
  public void setPixel(int x, int y, Color c) {
    if(x >= 0 && x <= w0 && y >= 0 && y <= h0) {
      pix[y*w0+x] = c.getRGB(); img = null;
    }
  }
  public void setPixel(int x, int y, float alpha, Color c) {
    int mask = (int)(alpha*255.0) << 24 | 0x00ffffff;
    if(x >= 0 && x <= w0 && y >= 0 && y <= h0) {
      pix[y*w0+x] = c.getRGB() & mask; img = null;
    }
  }
  public void drawBody(Graphics g) {
```

```
      if(img == null) {
        img = win.createImage(new MemoryImageSource(w0, h0, pix, 0, w0));
      }
      g.drawImage(img, x0, y0, win);
    }
  }
```

イメージは形としては矩形と同じく、幅と高さを持つ。しかしその内部では上述のように配列 pix にイメージの情報を保持する。なお、実際に描画するにはこの配列のデータを Image オブジェクトに一旦変換するが、描くたびに変換するのは計算量が増えるので「描く直前に変換し、次に描こうとしたとき変化がなかったら前の Image を再利用する」方針をとる。null という値は「変数にオブジェクトが入っていない」ことを表すので、これをもって「まだ Image を用意していない」ことを表させることにする。

また、イメージを描画するときにはこれまでの Graphics オブジェクトに加えて Component オブジェクト (ウィンドウの部分領域を表すオブジェクト) を使う必要があるため、これを内部で保持しておくことにする (実は Applet は Component のサブクラスなのでアプレット自身を渡せばよい)。

さて、init での初期設定としては、幅や高さや Component を覚えておくのに加えて、配列 pix の領域を確保して初期化することが必要である。初期値では「すべてのピクセルは色は黒だが完全に透明」にしておく。

次に、setPixel は座標 X,Y のピクセルを好きな色に設定するメソッドである。実は、Color オブジェクトのメソッド getRGB を使うとちょうど図 3 と同じ値 (ただし α の部分は常に 255 になる) が取り出せるのでほぼこれを利用する。なお、X,Y 座標が矩形の範囲外だったら何もしないことと、ピクセルを設定するときはこれまでの Image は常に無効にすることにも注意。

さて、もう 1 つ setPixel がありますね! 実は、上で説明したものは常に α が 255(不透明) なので、これとは別に自由に α が設定できるのも用意した。Java では、同じ名前のメソッドでも引数で区別ができるなら複数用意してよいことになっている (これを「オーバロード」と呼ぶ)。さて、こちらの版では α の値としては 0.0f〜1.0f の float 値を使うものとし、これを 255 倍して整数に変換したものを 24 ビット左にシフトすることで α の位置に置き、その右側 24 ビットはすべて「1」が来るような値を変数 mask に入れておく。あとでこの変数と getRGB の返値を and 演算すると引数の α の値と getRGB の値が組み合わされたピクセル値が完成する (難しいですか? 分からなければ、とにかくこのメソッドにお願いすれば α と色をともに設定できると思っておいてください)。

最後に drawBody だが、これは Image が無効ならば配列データから MemoryImageSource オブジェクトを作り、さらに Component に備わっている CreateImage メソッドで Image オブジェクトを生成する。そして最後に、Graphics オブジェクトの drawImage でイメージを画面に描画する。

なかなか大変だったが、これで準備ができたので、あとは好きなイメージを作るだけである。ここでは一番簡単そうなものとして、「円」をやってみよう。円は中心の座標と半径、α、そして HSB 値を指定するが、ただし B については 2 つ値を用意し、左から右へ行くにしたがって両者の間で値が連続的に変化するようにした。

```
  class ImgCircle extends GrImage {
    public ImgCircle(int x, int y, int r, float alpha,
```

```
                    float h, float s, float b0, float b1, Component c) {
    super(x-r, y-r, r*2+1, r*2+1, c);
    for(int i = 0; i <= r*2; ++i) {
      for(int j = 0; j <= r*2; ++j) {
        float b = b0 + (b1-b0)*(float)i/(float)(r*2);
        if((i-r)*(i-r)+(j-r)*(j-r)<=r*r) {
          this.setPixel(i, j, alpha, Color.getHSBColor(h, s, b));
        }
      }
    }
  }
}
```

これであとは、先のアプレットの init に次の 2 行を追加するだけで「グラデーションつきの円」が絵に加わるわけである。

```
g = new ImgCircle(100, 100, 30, 0.6f, 0.3f, 1.0f, 0.0f, 1.0f, this);
g.setSpeed(1, 1); set.add(g);
```

なかなかかっこいいでしょう?

演習 8 このプログラム (AppSam8b.java) をコピーしてきて動かせ。

演習 9 次のような図形を新しいクラスとして追加してみよ。

a. ドーナツ型。
b. 周囲に行くほど透明になる円。
c. 水平方向/垂直方向に色調が変化する矩形。
d. 何でもいいからグラデーションを持つ三角形。

なお、d はかなり難しい。以下のおまけでヒントをさしあげます。

9 おまけ 1: static メソッド

さて、これまで作ってきたメソッドはすべて、「あるオブジェクトについてこういう操作を行う」というものだった。しかし、計算の種類によっては、Math.tan のようにクラスに付随したメソッド (static メソッド) の方が適している。

たとえば、「2 つの整数を受け取り大きい方を返す」メソッドを定義してみる。

```
public static int max2(int a, int b) {
  if(a > b) {
    return a;
  } else {
    return b;
  }
}
```

これを使いたいクラスの中に入れておけばあとはどこでも式の中で「max2(x, y)」というのが自由に使えるようになる。また、複数のクラスの中から使いたいならこのようなメソッドを集めた「自分用の Math クラス」のようなものを作って、「MyMath.max2(x, y)」のように使う方がよいかも知れない。

ところで、上の例は「2 つの数の最大」だったが、「3 つの数の最大」だとどうだろう? もちろん、if の中で if を使えばできるが…次のはどうですか?

```
public static int max3(int a, int b, int c) {
  return max2(a, max2(b, c));
}
```

つまり、「3 つのうち最大のものは、b と c の大きい方と a とを比べてその大きい方」というわけ。このように、一度作った関数は徹底して利用し倒そう。

10 おまけ 2: 塗りつぶした三角形

塗りつぶした三角形というのは、円と同様に考えて「ある条件を満たした点の集まり」として扱うとよい。まず、点 (x_0, y_0) と (x_1, y_1) を通る直線の方程式は

$$y = y_0 + (x - x_0)\frac{y_1 - y_0}{x_1 - x_0}$$

ですね? (高校の数学だぜ。) だから、この「=」を「>」に置き換えると、その不等式を満たす点の集まりは直線より上の「A」の領域、また「<」に置き換えると直線より下の「B」の領域を表す。ここまでいい?

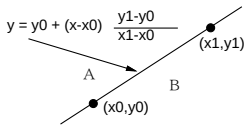

図 4: 直線の式

言い替えれば、次の式が正か負かによって点 (x, y) が直線のどちら側かが判定できる (0 ならば直線上にある):

$$(y - y_0)(x_1 - x_0) - (x - x_0)(y_1 - y_0)$$

ただしこの式は、さっきの式の右辺を左辺に移項して、それから垂直な線のときに x_0 と x_1 が等しくなって 0 で割ってしまうのを防ぐため、両辺に $(x_1 - x_0)$ を掛けたものである。この値を計算するメソッドを作っておこう。

```
static double ptv(double x0, double y0, double x1, double y1,
                  double x double y) {
    retrn (y-y0)*(x1-x0) - (x-x0)*(y1-y0);
}
```

さて、三角形では直線が 3 つあるから、これを組み合わせる。つまり、(x_0, y_0)、(x_1, y_1)、(x_2, y_2) から 2 つずつ選んで組み合わせて上の式にあてはめ、それの積を計算する。つまり次の計算をする。

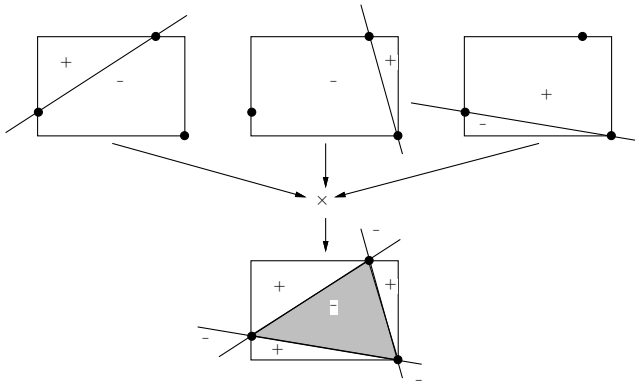

図 5: 三角形の領域

```
static double ptv3(double x0, double y0, double x1, double y1,
                   double x2, double y2, double x, double y) {
  return ptv(x0,y0,x1,y1,x,y)*ptv(x1,y1,x2,y2,x,y)*ptv(x2,y2,x0,y0,x,y);
}
```

ptv3 の符号は、図 5 のように三角形の内側の符号がマイナスで、そのすぐ外側がプラスであるか、または (点の順序によってはその反対に) あるいは内側がプラス、そのすぐ外側がマイナスになるはず。で、内側の符号がどちらかは、三角形の重心 $(\frac{x_0+x_1+x_2}{3}, \frac{y_0+y_1+y_2}{3})$ について ptv を計算してその値 (v としよう) の符号を調べればわかる (それと同じになる)。なお、交差部の外側も中と同じ符号になるが、そこは調べないので関係ない。

で、どの範囲の (x, y) について調べればいいか? それは、図 5 の長方形の中について調べればよい。この長方形の左端の X 座標は $min3(x_0, x_1, x_2)$、右端の X 座標は $max3(x_0, x_1, x_2)$、上端の Y 座標は $max3(y_0, y_1, y_2)$、下端の Y 座標は $min3(y_0, y_1, y_2)$ ですね。

なお、三角形の形が図 6 のように「平べったい」とうまく行かないので、その場合は 2 つの「平べったくない」三角形に分割して使うように。

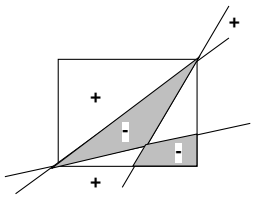

図 6: うまく行かない三角形の例

11 マルチスレッド

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。

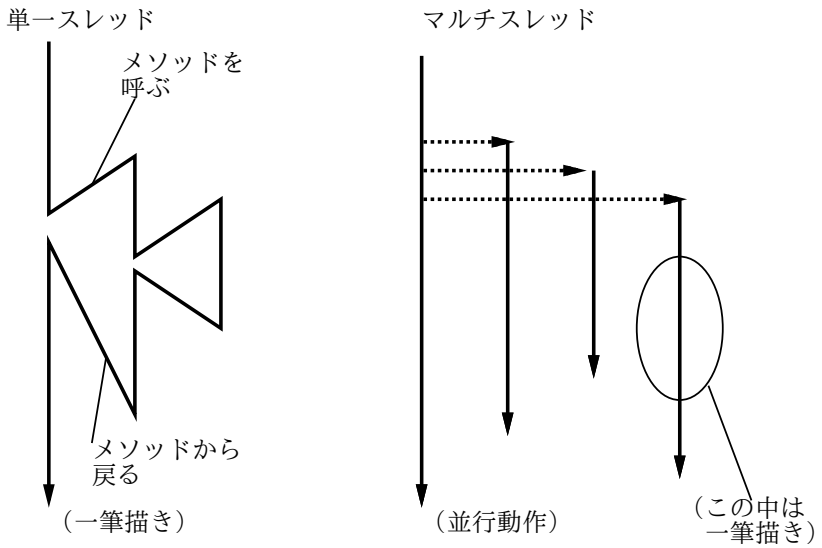

図 7: スレッドの概念

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。

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 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 yield()
    --- 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。

12　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。

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。

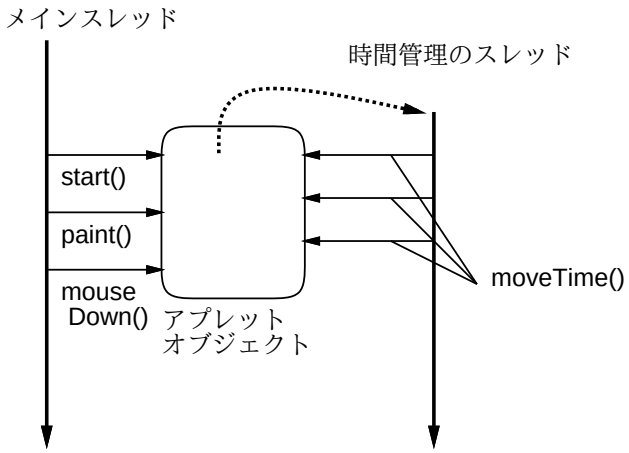

図 8: スレッドによるアニメーション

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 AppSam9a extends Applet {
  AnimCircle c0, c1;
  boolean running;
  public void init() {
    this.setBackground(Color.white);
    c0 = new AnimCircle(Color.blue, 100, 100, 20, 33.0, 15.0);
    c1 = new AnimCircle(Color.green, 130, 110, 25, -23.0, 45.0);
    addKeyListener(new KeyAdapter() {
      public void keyPressed(KeyEvent e) { setKey(e.getKeyChar()); }
    });
```

```
  }
  public void start() {
    running = true;
    new Thread() {
      public void run() {
        long basetime = System.currentTimeMillis();
        initTime(0.0);
        while(getRunning()) {
          try { Thread.sleep(100); } catch(Exception e) { }
          setTime(0.001 * (System.currentTimeMillis() - basetime));
        }
      }
    }.start();
  }
  public void stop() { running = false; }
  public boolean getRunning() { return running; }
  public void paint(Graphics g) { c0.draw(g); c1.draw(g); }
  public void initTime(double t) { c0.initTime(t); c1.initTime(t); }
  public void setTime(double t) { c0.setTime(t); c1.setTime(t); repaint(); }
  public void setKey(char ch) {
    if(ch == '+') { c0.changeSpeed(1.1); }
    if(ch == '-') { c0.changeSpeed(0.9); }
  }
}
```

このクラスでは、描かれる「もの」を表すのに AnimCircle クラスを用いる。また時間管理スレッドは内部クラスから生成する Thread オブジェクトによっている。init() の中では AnimCircle オブジェクトを用意することと、キーが押された時のリスナを登録する。そのリスナは押されたキーを引数として setKey() を呼ぶ) ようになっているが、setKey() の中では押されたキーが「+」なら片方の円の速度を 1.1 倍にし、「-」なら 0.9 倍にしている。

start() の中では時間管理スレッドを作成して起動するが、その時間管理スレッドのクラスを内部クラスとして定義している。時間管理スレッドの実行内容はメソッド run() の中に書くが、ここではまず最初に時間を初期設定するメソッド、続いて一定時間ごとに現在の時間を通知するメソッドを呼ぶようになっている。スレッドはアプレットが stop() されたら終わりたいので、実行中は変数 running に true が入っていて、これが false になると終わるという仕組みにする。

次に「動く円」を表すクラス AnimCircle を見てみよう。図形そのものはこれまでやったもののうちで最も簡単だが、「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 {
  Color cl;
  double vx, vy;
  double px, py;
```

```
    int rad;
    double basetime = 0.0;
    AnimCircle(Color c, int x, int y, int r, double vx1, double vy1) {
      cl = c; px = x; py = y; rad = r; vx = vx1; vy = vy1;
    }
    public void changeSpeed(double ratio) { vx *= ratio; vy *= ratio; }
    public void initTime(double time) { basetime = time; }
    public void setTime(double time) {
      px += (time - basetime)*vx;
      py += (time - basetime)*vy;
      if(px-rad < 0.0) { vx = -vx; px = 2*rad - px; }
      if(px+rad > 300.0) { vx = -vx; px = 600.0 - (px+2*rad); }
      if(py-rad < 0.0) { vy = -vy; py = 2*rad - py; }
      if(py+rad > 200.0) { vy = -vy; py = 400.0 - (py+2*rad); }
      basetime = time;
    }
    public void draw(Graphics g) {
      g.setColor(cl); g.fillOval((int)px-rad, (int)py-rad, 2*rad, 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 9 をよく見ていただければ分かると思う。

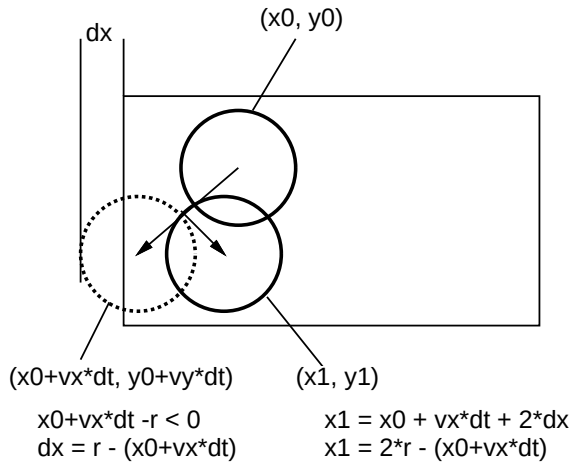

図 9: はね返り後の位置の計算

なお、このプログラムは無重力空間中でのボールの運動を「まね」していると言える。つまり無重力は宇宙へ行かないと体験できないにも関わらず、計算機内部の計算によってそれと同じことを計算し、その情報を利用者に見せてくれるわけである。これを「シミュレーション」

といい、計算機の重要な用途の1つである。たとえば、これに「重力」や「引力」などを入れて、実際には存在し得ない世界のモデルをシミュレーションさせることも簡単である。

演習 10 上の「はね返り」の例題をコピーしてきてそのまま動かせ。

演習 11 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)。

演習 12 上では円だけだったが、もっと別の形のものが「飛ぶ」ようにしてみよ。回転しながら飛んだり、飛んでいるものどうしが「ぶつかってはね返る」となおかっこよい。物理現象を正確に再現することには特にこだわらない。

13 インタフェース

さて、ここまでクラスの機能を整理して来たが、クラスが定義しているものは大きく次の2種類に分けることができる。

- インタフェース --- そのクラスのインスタンス (やサブクラスのインスタンス) が、どのようなメソッドを持っているか、という「外向きの」側面。
- 実現 --- そのクラスのインスタンスが内部的にはどのようなデータ (インスタンス変数) を持っていて、どのように動作するか (メソッドの動作記述)、という「内輪だけの」側面。

サブクラスを作ると、サブクラスは親クラスからインタフェースと実現の双方を引き継ぐことになる。

しかし場合によっては、実現を引き継ぎたくはないこともあるかも知れない (使い方は同じでも内部では全く違ったデータ表現を持つなど)。そのために、Java など新しい言語では「インタフェースだけを定義して実現は定義しない」機能が用意されるようになった。これをインタフェース機能と呼ぶ。

Java のインタフェースは、書き方はクラスとよく似ているが、次の点が違う。

- class の代りに interface と書く。
- 変数は final 変数 (つまり定数) だけ定義できる。

- メソッド定義は動作を書かない (abstract メソッドと同じ)。

たとえば次のような感じ。

```
interface Barkable {
  public void bark();
  public void barkStrong();
}
```

また、これを使う方 (クラスで言えばサブクラス) は次のようになる。

- extends ではなく implements と書く。
- インタフェースはいくつ implements してもよい。
- インタフェースで定義しているメソッドはすべてオーバライド (実現) しなければならない。

たとえば次のような感じ。

```
class Dog implements Barkable {
  ...
  public void bark() { System.out.println("Vow!"); }
  public void barkStrong() { System.out.println("VOW!"); }
}
class Airplane implements Barkable, Runnable {
  ...
  public void bark() { System.out.println("Boom!"); }
  public void barkStrong() { System.out.println("BOOM!"); }
}
```

そして、インタフェース型の変数にはそのインタフェースを実現 (implements) しているクラスのインスタンスが自由に入れられる。

```
Barkable b = new Dog(...);
...
b = new Airplane(...);
...
b.bark(); b.barkStrong(); b.bark();
```

さっきとちっとも変わらないって? 一見そうですが、「犬」と「飛行機」はたぶん中の構造は全然違って、共通の親クラスを持たないでしょうけれど、にも関わらずインタフェースを通じて「区別なく」「動的分配しながら」使えるわけです。

14 Runnableインタフェースを利用したスレッド

これまで、アニメーション等のためスレッドを作る場合、そのスレッドの動作は次の方法で指定していた。

- java.lang.Thread のサブクラスを作って、メソッド run() をオーバライド (定義) して、そこにスレッドの動作内容を書く)。

しかし実はこの他に、インタフェースを利用したもっと簡便なやり方があるのでそれを説明しよう。Thread クラスのコンストラクタを API ドキュメントで調べてみると、

```
public Thread(Runnable r);
```

というものがある。これを使うと、java.lang.Runnable インタフェースを実装したオブジェクトなら何でもそれをスレッドとして動かせる。Runnable インタフェースは次のようなものである。

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

つまりこのインタフェースは引数を持たないメソッド run() を持つ、ということだけを定めていて、Thread のコンストラクタに Runnable オブジェクト (Runnable インタフェースを実装したクラスのインスタンス) を渡すと、メソッド run() がスレッドとして実行されるような Thread オブジェクトが生成できるわけである。

これを利用すれば、これまでのようにアプレットのクラスとスレッドのクラスを分けなくても 1 つのクラスで済ませることができる。つまり次のようにするのである。

```
public class RxSample extends Applet implements Runnable {
  Thread tr;
  public void init() { ... }// 初期設定
  public void start() { (new Thread(this)).start(); }
  public void stop() { ... }
  public void paint(Graphics g) { ... } // 描画
  public void run() { ... } // この中の動作がスレッドとして実行
  ...
}
```

ここで一番の肝は「new Thread(this)」で Thread オブジェクトを生成しているところで、this はこのアプレットのインスタンスだから Runnable オブジェクトであり、そしてスレッドを実行開始させるとこのアプレットの run() メソッドが実行されるわけである。

では例題として、先の「飛ぶ円」をこの方式に改造してみよう。なお、さっきは円は 2 つだけだったが、今度は「動くものの集合」を用意し、そこにいくつでも入れられるようになっている。

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

class AnimCircle {
  Color cl;
```

```
  float vx, vy;
  float px, py;
  int radius;
  float basetime;
  AnimCircle(Color c, int x, int y, int r, float vx1, float vy1) {
    cl = c; px = x; py = y; radius = r; vx = vx1; vy = vy1;
  }
  public void setTime(float time) {
    basetime = time;
  }
  public void changeSpeed(float ratio) {
    vx *= ratio; vy *= ratio;
  }
  public void moveTime(float time) {
    px += (time - basetime)*vx;
    py += (time - basetime)*vy;
    if(px-radius < 0.0f) { vx = -vx; px = 2*radius - px; }
    if(px+radius > 300.0f) { vx = -vx; px = 600.0f - (px+2*radius); }
    if(py-radius < 0.0f) { vy = -vy; py = 2*radius - py; }
    if(py+radius > 200.0f) { vy = -vy; py = 400.0f - (py+2*radius); }
    basetime = time;
  }
  public void draw(Graphics g) {
    g.setColor(cl);
    g.fillOval((int)px-radius, (int)py-radius, 2*radius, 2*radius);
  }
}

class AnimSet {
  Vector vec = new Vector();
  public void add(AnimCircle a) {
    vec.addElement(a);
  }
  public void changeSpeed(float t) {
    for(Enumeration e = vec.elements(); e.hasMoreElements(); ) {
      AnimCircle a = (AnimCircle)e.nextElement(); a.changeSpeed(t);
    }
  }
  public void setTime(float t) {
    for(Enumeration e = vec.elements(); e.hasMoreElements(); ) {
      AnimCircle a = (AnimCircle)e.nextElement(); a.setTime(t);
    }
  }
```

```
  public void moveTime(float t) {
    for(Enumeration e = vec.elements(); e.hasMoreElements(); ) {
      AnimCircle a = (AnimCircle)e.nextElement(); a.moveTime(t);
    }
  }
  public void draw(Graphics g) {
    for(Enumeration e = vec.elements(); e.hasMoreElements(); ) {
      AnimCircle a = (AnimCircle)e.nextElement(); a.draw(g);
    }
  }
}

public class AppSam10a extends Applet implements Runnable {
  AnimSet set = new AnimSet();
  long basetime = System.currentTimeMillis();
  boolean running;
  public void init() {
    this.setBackground(Color.white);
    set.add(new AnimCircle(Color.blue, 100, 100, 20, 33.0f, 15.0f));
    set.add(new AnimCircle(Color.green, 130, 110, 25, -23.0f, 45.0f));
    addKeyListener(new KeyAdapter() {
      public void keyPressed(KeyEvent e) { setKey(e.getKeyChar()); }
    });
  }
  public void start() {
    running = true;
    (new Thread(this)).start();
  }
  public void stop() {
    running = false;
  }
  public void paint(Graphics g) {
    set.draw(g);
  }
  public void setKey(int ch) {
    if(ch == '+') set.changeSpeed(1.1f);
    if(ch == '-') set.changeSpeed(0.9f);
  }
  public void run() {
    float t = 0.001f * (System.currentTimeMillis() - basetime);
    set.setTime(t);
    while(running) {
      try { Thread.sleep(100); } catch(Exception e) { }
```

```
        t = 0.001f * (System.currentTimeMillis() - basetime);
        set.moveTime(t); repaint();
      }
    }
  }
```

演習 13 上の例題をそのまま動かせ。OK なら AnimCircle の「さまざまな飛び方をするサブクラス」を作り、それらを同時に飛ばせてみよ。

15　3 次元グラフィクス

15.1　3 次元グラフィクス

これまでの例題で出て来る絵はすべて 2 次元的なものだった。これに対し、立体的な (奥行きのある) 画像を生成することを「3 次元グラフィクス」と呼ぶ。ではこれらの相違は何だろう?

3 次元グラフィクスでは当然、「円」や「直線」や「多角形」の代りに「球」や「平面」や「多面体」を扱わなければならない。そして、それらに関わる点の座標もすべて 3 次元の座標ということになる。ではさっそく、そのためのクラスから用意しよう。

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

class Point3D {
  float x, y, z;
  public Point3D(float x0, float y0, float z0) {
    x = x0; y = y0; z = z0;
  }
  public float getX() { return x; }
  public float getY() { return y; }
  public float getZ() { return z; }
  public float perspX(float ze, float zs) { return x * (ze-zs)/(ze-z); }
  public float perspY(float ze, float zs) { return y * (ze-zs)/(ze-z); }
}
```

単なる「3 次元空間の点」だからその値を保持したり成分を取り出せるようになっているだけ、ということで簡単ですね。ところで、空間座標を 3 次元で計算したとしても、最後に表示するときはアプレット画面の 2 次元座標に変換しなければならない。ここでは図 10 のようにする。

すなわち、3 次元平面のうち、Z 軸の手前側の位置 $(0, 0, Z_e)$ の場所に目を起き、XY 平面に並行な平面 $Z = Z_s$ に「画面」があるものと考え、表示したい点 (X, Y, Z) と目の位置を結ぶ直線がこの「画面」と交わる箇所の画面上にこの点を置く。話を聞くと難しそうだが、これは単に

$$X_s \;=\; X\frac{Z_e - Z_s}{Z_e - Z}$$

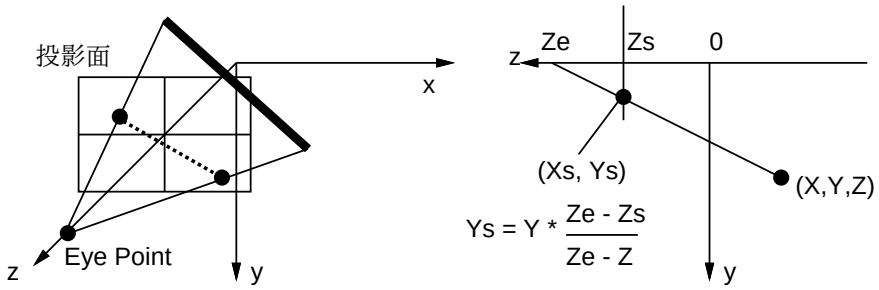

図 10: 視点変換

$$Y_s = Y\frac{Z_e - Z_s}{Z_e - Z}$$

という変換で画面座標 (X_s, Y_s) を求めればよいということに過ぎない。これを行うのがメソッド perspX() 等なわけである。

ついでに、ある点から別の点への変移はベクトルとして表せるので、ベクトルクラスも作ってしまおう。最初の方はほとんど点と変わらないが、内積と外積とノルムと単位ベクトルができるようにしてある。

```
class Vector3D {
  float x, y, z;
  public Vector3D(float x0, float y0, float z0) {
    x = x0; y = y0; z = z0;
  }
  public Vector3D(Point3D p0, Point3D p1) {
    this(p1.getX()-p0.getX(), p1.getY()-p0.getY(), p1.getZ()-p0.getZ());
  }
  public float getX() { return x; }
  public float getY() { return y; }
  public float getZ() { return z; }
  public float iprod(Vector3D v1) {
    float x1 = v1.getX(); float y1 = v1.getY(); float z1 = v1.getZ();
    return x*x1 + y*y1 + z*z1;
  }
  public Vector3D oprod(Vector3D v1) {
    float x1 = v1.getX(); float y1 = v1.getY(); float z1 = v1.getZ();
    return new Vector3D(y*z1-y1*z, z*x1-z1*x, x*y1-x1*y);
  }
  public float norm2() { return this.iprod(this); }
  public float norm() { return (float)Math.sqrt(this.norm2()); }
  public Vector3D unit() {
    float n = this.norm();
    if(n > 0.0f) { return new Vector3D(x/n, y/n, z/n); }
```

```
    else          { return this; }
  }
  public String toString() {
    return "vec:x:"+x+" y:"+y+" z:"+z;
  }
}
```

15.2 3次元の座標変換

さて、3次元グラフィクスでは空間内で物体が移動したり回転させられないと面白くない。このような座標変換は行列演算で表すのが便利である。たとえば、ある点 (X, Y, Z) の座標を (T_x, T_y, T_z) だけ移動させるのは

$$[X \quad Y \quad Z \quad 1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_Z & 1 \end{bmatrix} = [X + T_x \quad Y + T_y \quad Z + T_z \quad 1]$$

という行列演算で行える。また、同様にして原点を中心とした拡大は

$$\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

で表せるし、X軸、Y軸、Z軸廻りの回転はそれぞれ

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

を掛けることで表せる。こうしておくと便利なのは、たとえば「移動して、回転して、拡大」といった一連の変換を予め計算しておけることである。つまり、

$$[X \quad Y \quad Z \quad 1] \quad M_1 \quad M_2 \quad M_3 = [X \quad Y \quad Z \quad 1](M_1 \quad M_2 \quad M_3)$$

だから行列積を先に計算しておけばよい。ということは、任意の座標変換は1つの4×4行列で表せるわけである。

というわけで、このような行列(座標変換行列)とその計算をクラスにしてみた。この内部では「float の配列の配列」を保持していて、これを使って行列を表している。なお、その初期化には配列初期化指定と呼ばれるものを使ってみた(意味は見れば分かりますね)。

```
class Trans3D {
  float[][] a = { { 1.0f, 0.0f, 0.0f, 0.0f },
                  { 0.0f, 1.0f, 0.0f, 0.0f },
                  { 0.0f, 0.0f, 1.0f, 0.0f },
```

```
                { 0.0f, 0.0f, 0.0f, 1.0f } };
private Trans3D setTranslate(float tx, float ty, float tz) {
  a[3][0] = tx; a[3][1] = ty; a[3][2] = tz; return this;
}
private Trans3D setScale(float sx, float sy, float sz) {
  a[0][0] = sx; a[1][1] = sy; a[2][2] = sz; return this;
}
private Trans3D setRotX(float t) {
  float sint = (float)Math.sin(t);
  float cost = (float)Math.cos(t);
  a[1][1] = a[2][2] = cost; a[2][1] = sint; a[1][2] = -sint; return this;
}
private Trans3D setRotY(float t) {
  float sint = (float)Math.sin(t);
  float cost = (float)Math.cos(t);
  a[0][0] = a[2][2] = cost; a[2][0] = sint; a[0][2] = -sint; return this;
}
private Trans3D setRotZ(float t) {
  float sint = (float)Math.sin(t);
  float cost = (float)Math.cos(t);
  a[0][0] = a[1][1] = cost; a[0][1] = -sint; a[1][0] = sint; return this;
}
public static Trans3D newTranslate(float tx, float ty, float tz) {
  return (new Trans3D()).setTranslate(tx, ty, tz);
}
public static Trans3D newScale(float sx, float sy, float sz) {
  return (new Trans3D()).setScale(sx, sy, sz);
}
public static Trans3D newRotX(float theta) {
  return (new Trans3D()).setRotX(theta);
}
public static Trans3D newRotY(float theta) {
  return (new Trans3D()).setRotY(theta);
}
public static Trans3D newRotZ(float theta) {
  return (new Trans3D()).setRotZ(theta);
}
private float[][] getArray() {
  return a;
}
public Trans3D multiply(Trans3D bt) {
  Trans3D ct = new Trans3D();
  float[][] c = ct.getArray();
```

```
    float[][] b = bt.getArray();
    for(int i = 0; i < 4; ++i) {
      for(int j = 0; j < 4; ++j) {
        c[i][j] = 0.0f;
        for(int k = 0; k < 4; ++k) { c[i][j] += a[i][k]*b[k][j]; }
      }
    }
    return ct;
  }
  public Point3D transform(Point3D pt) {
    float x = pt.getX(); float y = pt.getY(); float z = pt.getZ();
    return new Point3D(x*a[0][0]+y*a[1][0]+z*a[2][0]+a[3][0],
                       x*a[0][1]+y*a[1][1]+z*a[2][1]+a[3][1],
                       x*a[0][2]+y*a[1][2]+z*a[2][2]+a[3][2]);
  }
}
```

いくつか注意すべき点について説明しておこう。

- コンストラクタでは「単位行列」に相当する変換 (何も変換しない) が生成される。その他の移動、拡大、回転はそのような行列を生成する static メソッドを使って作り出す。これらの static メソッドは中で単位行列を作った後、適切に書き換えるための private メソッドを呼ぶようになっている。
- 乗算演算では中の配列を直接参照した方が早いので、中身の配列を取り出すメソッドをやはり private メソッドで用意した。

さて、次にこれらの座標変換のパラメタを「時間とともに」変化させることでアニメーションを行うので、そのためのクラス群を作る。これらのクラス群は Move3D インタフェースを共通に持つものとして、生成した後はこのインタフェースに従って「ある時刻での変換」を取り出して利用するようにする。具体的な変化する変換としては「変化しない」(これがないと位置等を固定的に設定するのに不便)、「2 点間で直線にそって往復運動」「各軸の廻りに回転」を用意した。

```
interface Move3D {
  public Trans3D getTrans3D(float time);
}

class StaticMove3D implements Move3D {
  Trans3D t;
  public StaticMove3D(Trans3D t0) { t = t0; }
  public Trans3D getTrans3D(float time) { return t; }
}

class SlideMove3D implements Move3D {
  float dx, dy, dz, dt, t0;
```

```
  public SlideMove3D(float x, float y, float z, float t, float b) {
    dx = x; dy = y; dz = z; dt = t; t0 = b;
  }
  public Trans3D getTrans3D(float time) {
    float c = (float)Math.cos(t0 + time*dt);
    return Trans3D.newTranslate(dx*c, dy*c, dz*c);
  }
}

class RotateXMove3D implements Move3D {
  float dt, t0;
  public RotateXMove3D(float t, float b) { dt = t; t0 = b; }
  public Trans3D getTrans3D(float t) { return Trans3D.newRotX(t0+t*dt); }
}

class RotateYMove3D implements Move3D {
  float dt, t0;
  public RotateYMove3D(float t, float b) { dt = t; t0 = b; }
  public Trans3D getTrans3D(float t) { return Trans3D.newRotY(t0+t*dt); }
}

class RotateZMove3D implements Move3D {
  float dt, t0;
  public RotateZMove3D(float t, float b) { dt = t; t0 = b; }
  public Trans3D getTrans3D(float t) { return Trans3D.newRotZ(t0+t*dt); }
}
```

次に画面上のオブジェクトであるが、ここでは基本的な図形としては一番簡単な「三角形」のみ用意し、あと例によって「複数の動くオブジェクトを組み合わせた世界」との 2 種類のみを使って構成する。これらはいずれも Object3D インタフェースを実装し、時刻の設定と描画を共通に扱える。描画時には Graphics オブジェクト、変換行列、視点と画面の Z 座標、そして「光線ベクトル」をパラメタに持つ。光線ベクトルというのは光が当っている方向を表すもので、光に向いた面は明るく、光に垂直な面は暗く表示するために使う。

```
interface Object3D {
  public void setTime(float time);
  public void draw(Graphics g, Trans3D t, Vector3D l, float ze, float zs);
}

class Triangle3D implements Object3D {
  Point3D[] a = new Point3D[3];
  float hue, sat;
  public Triangle3D(Point3D p, Point3D q, Point3D r, float h, float s) {
    a[0] = p; a[1] = q; a[2] = r; hue = h; sat = s;
```

```
  }
  public void setTime(float time) { }
  public void draw(Graphics g, Trans3D t, Vector3D l, float ze, float zs) {
    int[] xp = new int[3]; int[] yp = new int[3];
    Point3D[] b = new Point3D[3];
    for(int i = 0; i < 3; ++i) {
      b[i] = t.transform(a[i]);
      xp[i] = (int)b[i].perspX(ze, zs); yp[i] = (int)b[i].perspY(ze, zs);
    }
    Vector3D v1 = new Vector3D(b[0], b[1]);
    Vector3D v2 = new Vector3D(b[0], b[2]);
    float x = 0.3f + 0.7f*(float)Math.abs(l.iprod(v1.oprod(v2).unit()));
    g.setColor(Color.getHSBColor(hue, sat, x));
    g.fillPolygon(xp, yp, 3);
  }
}

class World3D implements Object3D {
  Vector objs = new Vector();
  Vector moves = new Vector();
  Trans3D trans;
  public void addObj3D(Object3D o) { objs.addElement(o); }
  public void addMove3D(Move3D m) { moves.addElement(m); }
  public void setTime(float t) {
    for(Enumeration e = objs.elements(); e.hasMoreElements(); ) {
      Object3D o = (Object3D)e.nextElement();
      o.setTime(t);
    }
    trans = new Trans3D();
    for(Enumeration e = moves.elements(); e.hasMoreElements(); ) {
      Move3D m = (Move3D)e.nextElement();
      trans = trans.multiply(m.getTrans3D(t));
    }
  }
  public void draw(Graphics g, Trans3D t, Vector3D l, float ze, float zs) {
    for(Enumeration e = objs.elements(); e.hasMoreElements(); ) {
      Object3D o = (Object3D)e.nextElement();
      o.draw(g, trans.multiply(t), l, ze, zs);
    }
  }
}
```

World3D の方は基本的に複数の物体と座標変換を保持するだけで、時刻に応じた座標変換を計算してそれに基づいて各オブジェクトの draw() メソッドを呼び出す。

では最後に、簡単な立体アニメーションを作ってみよう。簡単といいつつ、立体だとデータが大変ですね。あと、見る方向をキー操作で変更できるようにkeyDown()でキーコマンドを受け付け、適宜変換を変更するようにしてある。

```
public class AppSam10b extends Applet implements Runnable {
  World3D world = new World3D();
  Trans3D trans = Trans3D.newTranslate(150f, 100f, 0f);
  Vector3D light = new Vector3D(3.0f, 2.0f, 1.0f).unit();
  float ze = 200.0f;
  float zs = 100.0f;
  long basetime = System.currentTimeMillis();
  boolean running;
  public void init() {
    this.setBackground(Color.white);
    Point3D p1 = new Point3D(50.0f, 0.0f, 0.0f);
    Point3D p2 = new Point3D(0.0f, 50.0f, 0.0f);
    Point3D p3 = new Point3D(0.0f, 0.0f, 0.0f);
    Point3D p4 = new Point3D(-50.0f, 0.0f, 0.0f);
    Point3D p5 = new Point3D(0.0f, -50.0f, 0.0f);
    World3D wing = new World3D();
    wing.addObj3D(new Triangle3D(p1, p2, p3, 0.7f, 0.7f));
    wing.addObj3D(new Triangle3D(p3, p4, p5, 0.2f, 0.8f));
    wing.addMove3D(new RotateZMove3D(0.3f, 0.0f));
    wing.addMove3D(new RotateXMove3D(1.0f, 0.0f));
    wing.addMove3D(new StaticMove3D(Trans3D.newTranslate(20f, 10f, 0f)));
    wing.addMove3D(new SlideMove3D(30.0f, 30.0f, 0.0f, 0.3f, 0.0f));
    p1 = new Point3D(-100f, -100f, -70f);
    p2 = new Point3D(100f, -100f, -70f);
    p3 = new Point3D(100f, 100f, -70f);
    p4 = new Point3D(-100f, 100f, -70f);
    p5 = new Point3D(0f, 0f, -50f);
    world.addObj3D(new Triangle3D(p1, p2, p5, 0.4f, 0.3f));
    world.addObj3D(new Triangle3D(p3, p4, p5, 0.4f, 0.3f));
    world.addObj3D(new Triangle3D(p2, p3, p5, 0.1f, 0.3f));
    world.addObj3D(new Triangle3D(p4, p1, p5, 0.1f, 0.3f));
    world.addObj3D(wing);
    addKeyListener(new KeyAdapter() {
      public void keyPressed(KeyEvent e) { setKey(e.getKeyChar()); }
    });
  }
  public void start() {
    running = true;
    (new Thread(this)).start();
  }
```

```
  public void stop() {
    running = false;
  }
  public void paint(Graphics g) {
    world.draw(g, trans, light, ze, zs);
  }
  public void setKey(char ch) {
    if(ch == 'd') {
      trans = trans.multiply(Trans3D.newTranslate(0f, -10f, 0f));
    } else if(ch == 'u') {
      trans = trans.multiply(Trans3D.newTranslate(0f, +10f, 0f));
    } else if(ch == 'r') {
      trans = trans.multiply(Trans3D.newTranslate(-10f, 0f, 0f));
    } else if(ch == 'l') {
      trans = trans.multiply(Trans3D.newTranslate(+10f, 0f, 0f));
    } else if(ch == '1') {
      trans = trans.multiply(Trans3D.newRotX(0.2f));
    } else if(ch == '2') {
      trans = trans.multiply(Trans3D.newRotX(-0.2f));
    } else if(ch == '3') {
      trans = trans.multiply(Trans3D.newRotY(0.2f));
    } else if(ch == '4') {
      trans = trans.multiply(Trans3D.newRotY(-0.2f));
    } else if(ch == '+') {
      zs -= 10f; ze -= 10f;
    } else if(ch == '-') {
      zs += 10f; ze += 10f;
    }
  }
  public void run() {
    float t = 0.001f * (System.currentTimeMillis() - basetime);
    world.setTime(t);
    while(running) {
      try { Thread.sleep(100); } catch(Exception e) { }
      t = 0.001f * (System.currentTimeMillis() - basetime);
      world.setTime(t); repaint();
    }
  }
}
```

で、最後の run() をスレッドとして実行させることでアニメーションを実現するわけである。さすがに最後の例題だけあって長かったですね!

演習 14 上のアプレットをそのまま動かせ。OK なら「好きな形のものを好きなように飛ばして」みよ。