情報科学特殊講義'2000 ＃5

久野　靖*

2000.1.24-27

0　はじめに

「メソッドを作る」というのは、やっぱりはじめての人には結構大変だと思います。そのため、今回は前回の課題の解説をきちんとやります。

続いて、いよいよ「クラスを作る」ことにします。実はクラスはほとんどメソッドの集まりなので、メソッドが作れればクラスを作るのも簡単です。そして、ちょっとハードかも知れませんが、あるクラスを「下敷き」として別のクラスを作る「継承」の機能についても説明します。

1　前回の練習問題の解説

演習 1a

決まった大きさの「日の丸」。これは中心の座標をもとに矩形や円の位置を指定するパラメタを計算してやればよい。

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

public class r5ex1a extends Applet {
  public void paint(Graphics g) {
    hinomaru(g, 100,80); hinomaru(g, 180, 130); hinomaru(g, 200, 60);
  }
  void hinomaru(Graphics g, int x0, int y0) {
    g.setColor(Color.white);  g.fillRect(x0-40, y0-30, 80, 60);
    g.setColor(Color.red); g.fillOval(x0-20, y0-20, 40, 40);
  }
}
```

演習 1b

上と同じく。ただし円の直径が指定されるので、計算が厄介に。実数で計算した場合は最後に int に変換しないといけない。

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

public class r5ex1b extends Applet {
```

*筑波大学大学院経営システム科学専攻

```
  public void paint(Graphics g) {
    hinomaru(g,100,80,20); hinomaru(g,180,130,25); hinomaru(g,200,60,30);
  }
  void hinomaru(Graphics g, int x0, int y0, int r0) {
    g.setColor(Color.white);
    g.fillRect(x0-r0, y0-(int)(0.75*r0), 2*r0, (int)(1.5*r0));
    g.setColor(Color.red);
    g.fillOval(x0-r0/2, y0-r0/2,r0, r0);
  }
}
```

演習 1c

色を指定できるだけで前とほぼ同じ。

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

public class r5ex1c extends Applet {
  public void paint(Graphics g) {
    hinomaru(g,100,80,20, Color.blue, Color.red);
    hinomaru(g,180,130,25, Color.green, new Color(100, 130, 250));
    hinomaru(g,200,60,30, Color.black, Color.white);
  }
  void hinomaru(Graphics g, int x0, int y0, int r0, Color c0, Color c1) {
    g.setColor(c0);
    g.fillRect(x0-r0, y0-(int)(0.75*r0), 2*r0, (int)(1.5*r0));
    g.setColor(c1);
    g.fillOval(x0-r0/2, y0-r0/2,r0, r0);
  }
}
```

演習 2a

極座標ベクトルから普通の 2 次元ベクトルに変換できれば簡単。Math.sin 等の結果は double なので int に変換することと、度数とラジアンの換算ができることが必要。

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

public class r5ex2a extends Applet {
  public void paint(Graphics g) {
    for(int i = 0; i < 100; ++i) {
      polar(g, Color.red, 150, 100, (double)(i*3), 100.0);
    }
  }
  void polar(Graphics g, Color c, int x0, int y0, double t, double l) {
    int x1 = x0 + (int)(Math.cos(Math.PI*t/180.0)*l);
    int y1 = y0 + (int)(Math.sin(Math.PI*t/180.0)*l);
```

```
    g.setColor(c); g.drawLine(x0, y0, x1, y1);
  }
}
```

演習 2b

ほとんど解説を要しないくらい簡単。日の丸より簡単。

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

public class r5ex2b extends Applet {
  public void paint(Graphics g) {
    for(int i = 0; i < 20; ++i) {
      square(g, new Color(i*10, 255-i*5, 100+i*5), 150, 100, 180-i*6);
    }
  }
  void square(Graphics g, Color c, int x0, int y0, int l) {
    g.setColor(c); g.fillRect(x0-l/2, y0-l/2, l, l);
  }
}
```

演習 2c

「描かない辺」を指定するということは、それぞれの辺について「その指定された辺でなければ描く」ように if 文をかぶせればよい。

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

public class r5ex2c extends Applet {
  public void paint(Graphics g) {
    square1(g, Color.red, 100, 100, 50, 0);
    square1(g, Color.red, 150, 150, 50, 1);
    square1(g, Color.red, 200, 100, 50, 2);
    square1(g, Color.red, 150, 50, 50, 3);
  }
  void square1(Graphics g, Color c, int x0, int y0, int l, int d) {
    g.setColor(c);
    int u = l/2;
    if(d != 2) g.drawLine(x0-u, y0-u, x0-u, y0+u);
    if(d != 1) g.drawLine(x0-u, y0-u, x0+u, y0-u);
    if(d != 0) g.drawLine(x0+u, y0-u, x0+u, y0+u);
    if(d != 3) g.drawLine(x0+u, y0+u, x0-u, y0+u);
  }
}
```

演習 4a

これは単に「1 方向に曲る」だけでなく「逆にも曲る」ように自分自身への呼び出しを追加すればよい。

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

public class r5ex4a extends Applet {
  public void paint(Graphics g) {
    tree(g, Color.blue, 150, 200, -90, 20, 45, 5, 0.8);
  }
  void tree(Graphics g, Color c, int x0, int y0, int d0, int d1,
      int l0, int l1, double r) {
    if(l0 <= l1) return;
    int x1 = x0 + (int)(Math.cos(Math.PI*d0/180.0)*l0);
    int y1 = y0 + (int)(Math.sin(Math.PI*d0/180.0)*l0);
    g.setColor(c); g.drawLine(x0, y0, x1, y1);
    tree(g, c, x1, y1, d0+d1, d1, (int)(r*l0), l1, r);
    tree(g, c, x1, y1, d0-d1, d1, (int)(r*l0), l1, r);
  }
}
```

演習 4b

これは単に「終わる」という条件のところで「円を描く」動作を追加すればよい。円を描いたあと return; でもいいけど if-else にした方が美しいと私は思う (どうかな?)。

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

public class r5ex4b extends Applet {
  public void paint(Graphics g) {
    tree(g, new Color(150, 100, 0), 150,200,-90,20,45,10,0.8,Color.pink,10);
  }
  void tree(Graphics g, Color c, int x0, int y0, int d0, int d1,
      int l0, int l1, double r, Color c1, int l2) {
    if(l0 <= l1) {
      g.setColor(c1); g.fillOval(x0-l2/2, y0-l2/2, l2, l2);
    } else {
      int x1 = x0 + (int)(Math.cos(Math.PI*d0/180.0)*l0);
      int y1 = y0 + (int)(Math.sin(Math.PI*d0/180.0)*l0);
      g.setColor(c); g.drawLine(x0, y0, x1, y1);
      tree(g, c, x1, y1, d0+d1, d1, (int)(r*l0), l1, r, c1, l2);
      tree(g, c, x1, y1, d0-d1, d1, (int)(r*l0), l1, r, c1, l2);
    }
  }
}
```

演習 4c の「花見」はこれを複数配置するだけだから略。

演習 5

まずプログラムは疑似コードをそのまま直すだけだから…

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

public class r5ex5a extends Applet {
  public void paint(Graphics g) {
    dragon(g, Color.black, 150-64, 100, 150+64, 100, 8);
  }
  void dragon(Graphics g, Color c, int x0, int y0, int x1, int y1, int n) {
    int dx = (x1 - x0)/2;
    int dy = (y1 - y0)/2;
    int x2 = x0 + dx - dy;
    int y2 = y0 + dx + dy;
    if(n > 1) {
      dragon(g, c, x0, y0, x2, y2, n-1);
      dragon(g, c, x1, y1, x2, y2, n-1);
    } else {
      g.setColor(c);
      g.drawLine(x0, y0, x2, y2); g.drawLine(x1, y1, x2, y2);
    }
  }
}
```

さて、問題はこれが何を意味するか。上のは n が 8 だったが、これが 1〜4 の場合は図 1 のような形になる。

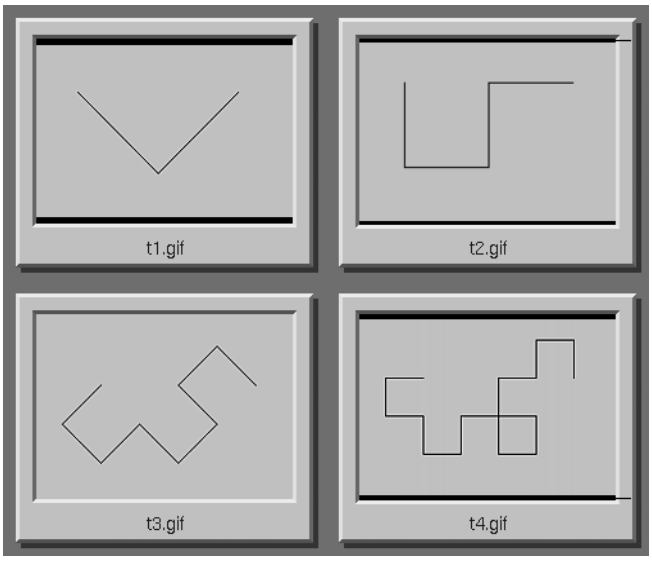

図 1: dragon の形 (n=1〜4)

ではこれはどのようにして作られているのだろう? まず、n=0 のときは単なる両端を結ぶ線分になる。そうでないときは、その線分を直角 2 等辺三角形の長辺として、残り 2 辺を通る折れ線に書き換える (折れ線を元の線のどちら側に置くかは「交互に」置くようにする)。これを図 2 のように繰り返して行くと、折れ線の線分数が倍々になって、だんだんごちゃごちゃとした形になっていく。その形が「龍」に似ているため、これは「ドラゴン曲線」と呼ばれている (英語で curve というのは折れ線も含むらしい)。

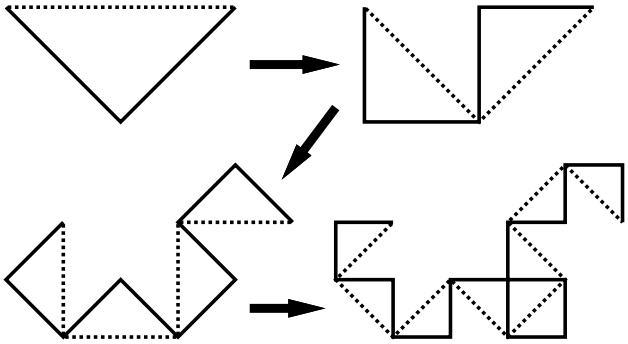

図 2: dragon の生成規則

2　クラスとオブジェクト

2.1　クラスを作ることの意味

これまでに「PrintStream オブジェクト」「Color オブジェクト」「Graphics オブジェクト」などさまざまなオブジェクトを利用してきたが、これらはすべて「既に定義されているオブジェクトの種類」(簡単に言えば API ドキュメントに載っているもの) だった。しかし今回からは、「自分で新しいオブジェクトの種類を作り出す」ことを学ぶ。まず最初に、なぜそういうことが必要なのか説明しよう。

これまで、プログラムを書くときは主に「動作」を中心に考えて来た。たとえば「三角を描く」とか「日の丸を描く」といったひとまとまりの動作をメソッドにすれば、プログラムは分かりやすくなる、ということも学んだ。

しかし、実際の世の中では我々は「動作」の前にまず「もの」について考える。たとえばここに「石ころ」という「もの」があって、それに「けとばす」という「動作」を施すとそれは「ころがる」という「動作」をする、というふうに。だからプログラムでもまず「もの」について考えるほうが普段と同じ考え方が使えてわかりやすいものができる。

具体的にはどういうことだろう。それは、たとえば上の例だとまず「三角形」という「もの」を考えることに相当する。画面には三角形をいくつも描けるが、それぞれの三角形には「色」とか「各頂点の座標」といったそれぞれに固有の「属性」があると考える。そして、「描く」という動作はそれぞれの三角形に付随している。

たとえば三角形 A は青くて大きい。三角形 B は赤くて小さい。A に「描く」という動作をさせれば、青くて大きい三角形が画面に描かれ、B に「描く」という動作をさせれば赤くて小さい三角形が画面に描かれる。こういうふうにしておけば、2 つの三角形の各頂点の座標を別に覚えておいて「三角形を描く」というメソッドを毎回 7 つも 8 つもパラメタをつけて呼び出すよりずっと分かりやすくなる。また、三角形に「色を変える」という動作を実行させてから再度「描く」ことで形はそのままで色を変えたりも簡単にできる。逆に三角形の色を知りたい場合も、色は三角形自身が属性として持っているので、「色を教えてもらう」という動作が必要になる。

2.2　Java でのクラスの作り方

…抽象的で分かりにくいですか? ではこれらの事柄を Java でどういうふうに表すか説明しよう。まず、A も B も「三角形」という「種類」は同じだった。Java ではものの「種類」はクラスで表す。クラスの名前は自由につけてよい。たとえば Sankaku という名前にしよう。

```
class Sankaku {
  ...
}
```

次に、それぞれの三角形には固有の「属性」があるのだった。属性で、その三角形を表すのに必要な情報が一式揃っていなければならない。これらの情報は Java では当然、変数として表す。3 頂点の座標はいろいろな

表し方があるが、ここでは後で扱うのに便利なように「重心の座標 (gx, gy) と、そこを起点とする 3 つの変移ベクトル」を使うことにする。あと、色も必要ですね。

```
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
  ...
}
```

さて、当然これらの情報を「初期設定」する必要がある。それには、「コンストラクタ」を定義する。コンストラクタはメソッドとよく似ているが、ただしクラスと同じ名前を持っている。たとえば「new Color(...)」というのは Color クラスのコンストラクタを呼んでいるのだった。ここではクラス名が Sankaku だからコンストラクタの名前も Sankaku になる。そしてその引数として色と 3 頂点の座標を渡すことにしよう。3 頂点の座標から上の情報を求めるのは少々計算してやればよい。

```
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;
  }
  ※
}
```

なお、public というのはこのコンストラクタがクラスの外から自由に呼べることを指定している。

さて、あとは三角形のさまざまな「動作」を Java のメソッドとして※のところに追加して行けば良い。で最初は「描く」から。

```
  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);
  }
```

色や位置は三角形の中に覚えてあるので、パラメタで指定しなくてよい。ずっとすっきりしたでしょう? なお、座標類は double で覚えているけれど、fillPolygon には int を渡す必要があるため、型変換が必要なのに注意。

次に、位置を変更するメソッドを用意しよう。

```
  public void moveTo(double x, double y) { gx = x; gy = y; }
```

すごく簡単ですね。ところで、位置を変更するには現在の位置も知りたいでしょう? そのため、重心の X 座標と Y 座標を調べるメソッドを用意する。これらのメソッドは重心の値を「返す」ので、(1) 返り値として double を指定する、(2)「return 式;」という文で値を実際に返す、ということが必要になる。

```
  public double getX() { return gx; }
  public double getY() { return gy; }
```

返す値は属性として覚えているので、引数がなにもないことに注意。ついでに色も設定や参照ができるようにしよう。

```
  public void setColor(Color c) { color = c; }
  public Color getColor() { return color; }
```

これで Sankaku クラスが完成した。これを使って画面に三角形を 2 つ描くアプレットを作ってみよう。アプレットのクラスは単に Sankaku オブジェクトを 2 つ作って変数に入れておき、paint() ではそれらの draw() メソッドを呼ぶだけだである。ソースファイルにはアプレットクラスとそれが使う補助クラスを一緒に入れておけばよい。全体をまとめて掲載する。

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

public class R5Sample1 extends Applet {
  Sankaku s1 = new Sankaku(Color.red, 20.0, 150.0, 100.0, 150.0, 50.0, 30.0);
  Sankaku s2 = new Sankaku(Color.blue, 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);
    s2.setColor(s2.getColor().brighter());
    s1.draw(g); s2.draw(g);
  }
}

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; }
}
```

ここでは、アニメーションはまだちょっと無理なので、とりあえず「再表示ごとにちょっとずつ絵が動く」というのをやってみよう。それには、paint() の中でそれぞれの三角形の位置をちょっとずらしてやるようにした。

つまり、paint() は 2 つの三角形を今までよりちょっとだけずれた位置に動かしてから描画を行う。これをブラウザで表示させると、「再読み込み」ボタンを押すごとにちょっとずつ三角形が移動していくのを見ることができる。[1]

演習 1 上記の例題アプレットを打ち込んでそのまま動かせ。

演習 2 上記の例題アプレットのクラス Sankaku を改良して、アプレット全体次のように変更してみよ。

a. Sankaku クラスに全体の大きさを r 倍に拡大/縮小するメソッド scale() を追加し、再読み込みごとに 2 つの三角形の大きさが変化していくようにしてみよ。

b. Sankaku クラスに全体を t 度回転させるメソッド rotate() を追加し、再読み込みごとに 2 つの三角形が互いに逆方向に回転するようにしてみよ。

(ヒント: 3 つの変移ベクトルをそれぞれ極座標表現に変換し、角度を加算してからまた直交座標に戻す。極座標にするのには $arctan$ を使うが、$Math.atan(dy/dx)$ とすると dx が 0 のとき商が無限大になってしまってうまく行かないので、$Math.atan2(dx, dy)$ を使うとよい。$atan2$ は内部で割り算を使わずに dx が 0 ならただちに $\frac{\pi}{2}$ を返してくれるようになっている。)

c. Sankaku クラスに自分と同じ「コピーの」三角形を作って返すメソッド copy() を追加し、最初は三角形が 1 つだけだが、再読み込みごとに新しい場所に最初の三角形のコピーが現われるようにしてみよ。

(ヒント: 三角形を新しく作るのは「new Sankaku(...)」を使えばよい。三角形の個数がいくつまで増えるか分からないので、Sankaku を入れる配列と今いくつ Sankaku オブジェクトがあるか覚えておく変数を用意する。たとえば

```
Sankaku[] a = new Sankaku[20]; // 20 個まで
int count = 0; // 個数
```

という感じかな。)

3 クラスと継承

さて、ここでクラスの新しい側面として「継承」という機能について説明しよう。「継承」というのは、あるクラスを下敷き (土台) にして新しいクラスを作ることを言う。なぜそういう機能があるのかというと、既にできている「土台」を元にそこにちょっとだけ継ぎ足すことで、少しずつ機能を増やして行けるとプログラムが作りやすい場合があるから。継承について以下にまとめておく。

- 下敷きになるクラスを「親クラス」「スーパークラス」、新しく作るクラスを「子クラス」「サブクラス」と呼ぶ。
- 継承を指定するには、クラス定義の先頭部で「`class ... extends 親クラス ...`」と指定する。
- 親クラスの変数、メソッドは子クラスにそのまま引き継がれる (継承される)。
- 子クラスでは、変数やメソッドを新たに追加できる。
- さらに子クラスでは、メソッドを別のものに差し替えることができる (これをオーバライドという)。
- 子クラスのコンストラクタやメソッドの中からは「`super`」という特別な名前を指定することで親クラスのコンストラクタやメソッドを呼び出すことができる。

ちんぷんかんぷんですか? でも「extends」というのは見たことがあるでしょう? 実は、皆様が作って来たアプレットも、Applet というクラスを「土台」にして、そのうちの「初期設定の処理」(init() のことですね)、「画面を描く処理」(paint() のことですね) などのメソッドを差し替えて作ったものなのである。

[1] appletviewer の場合は「restart」ないし「再実行」するか、窓の上に別の窓をかぶせてからどけるとちょっとずつ動く。後者方法は Netscape でも使える。実は paint() は窓が隠されてその後見えるようになったとき、窓の中身を描き直すために呼ばれる。ただし部分的に隠してからどけると隠されなかった部分は描き直されないので絵がくずれてしまうが。

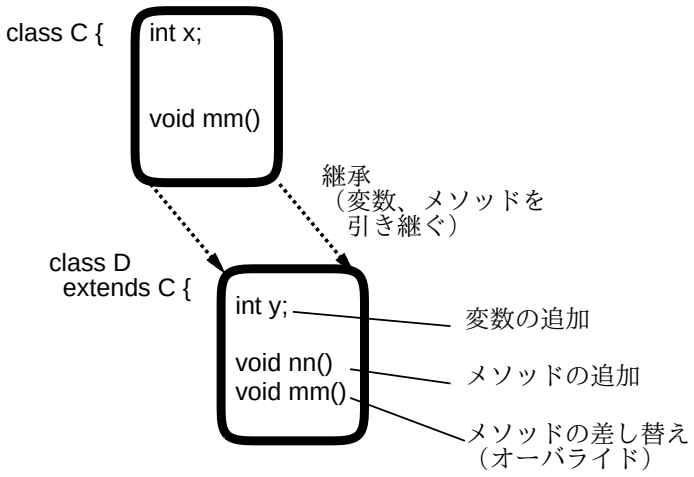

図 3: クラスと継承

4 アプレットクラスのさまざまなメソッド

上述のように、アプレットはクラス Applet のサブクラスであり、そのメソッド群を継承している。これまで paint() だけを定義していたのは、継承してきた paint() は「何も描かない」メソッドなので、これを差し替えて (オーバライド) 自分の好きな絵を描くように変更していたわけ。これ以外にもアプレットクラスには次のようなメソッドがあり、必要に応じてオーバライドすることができる。

- `public void init()` — アプレットが最初にロードされた後 1 回だけ呼ばれる。初期設定に使われる。
- `public void start()` — アプレットの実行開始時 (というのはそのアプレットを含むページが表示される時そのつど) 呼ばれる。
- `public void stop()` — アプレットの実行停止時 (というのはそのアプレットを含むページの表示が別のものに切り替わる時そのつど) 呼ばれる。
- `public void destroy()` — アプレットが破棄される時 (というのはブラウザのメモリ等が不足して使っていないアプレットのコードなどを捨ててしまう時) 呼ばれる。
- `public void paint(Graphics g)` — アプレットの画面を描く必要があるときに呼ばれる。

特に init() は込み入った初期設定があるときには必ずお世話になるので覚えておくとよい。

5 継承を利用したクラスの構造化

さて、次は継承を実際に自分のプログラムで活用する話題に進もう。継承を使うと、土台となるクラスに揃っている機能はそのまま「借りて来れば」済むのでわざわざ書かなくて済む。たとえば、先の Sankaku クラスの機能を「借りて来て」円のクラスを作り、これを利用して「再描画するごとにちょっとずつ動く三角と円」を作成した。

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

public class R5Sample2 extends Applet {
  Sankaku s1 = new Sankaku(Color.red, 20.0, 150.0, 100.0, 150.0, 50.0, 30.0);
  Circle s2 = new Circle(Color.blue, 90.0, 100.0, 40.0);
  public void paint(Graphics g) {
    s1.moveTo(s1.getX()+5, s1.getY()-3);
```

```
    s2.moveTo(s2.getX()+3, s2.getY()+4);
    s2.setColor(s2.getColor().brighter());
    s1.draw(g); s2.draw(g);
  }
}

class Sankaku {
  // まったく同じなので略
}

class Circle extends Sankaku {
  double rad;
  public Circle(Color c, double x, double y, double r) {
    super(c, x, y, x, y, x, y); rad = r; //☆
  }
  public void draw(Graphics g) {
    g.setColor(color);
    g.fillOval((int)(gx-rad), (int)(gy-rad), (int)(rad*2), (int)(rad*2));
  }
}
```

つまり、新しくクラス Circle を作っているが、moveTo() とか setColor() とかは Sankaku クラスのものを利用しているので、用意するのはコンストラクタと draw() だけで済む。その代り、ちょっとヘンなところがある。

- Circle のコンストラクタの中にある「`super(c, x, y, x, y, x, y)`」とは? — これは、Circle をあくまでも Sankaku の特別なものとして作るので、Circle を作る時には Sankaku のコンストラクタを (初期設定のために) 呼ばなければいけない。「`super(...)`」というのはサブクラスの中で親クラスのコンストラクタを呼び出すための記法である。

このように、ドロナワ的に Sankaku を利用した Circle を作ることはできるのだが、プログラムとして美しくない。これをちゃんとするには、「図形一般」というクラスを用意して、そのサブクラスとして Sankaku や Circle を用意するようにすればよい。

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

public class R5Sample3 extends Applet {
  Sankaku s1 = new Sankaku(Color.red, 20.0, 150.0, 100.0, 150.0, 50.0, 30.0);
  Circle s2 = new Circle(Color.blue, 90.0, 100.0, 40.0);
  public void paint(Graphics g) {
    s1.moveTo(s1.getX()+5, s1.getY()-3);
    s2.moveTo(s2.getX()+3, s2.getY()+4);
    s2.setColor(s2.getColor().brighter());
    s1.draw(g); s2.draw(g);
  }
}
class Figure {
  Color color; // 色
  double gx, gy; // 重心の X,Y 座標
  public Figure(Color c, double x, double y) {
```

```
    color = c; gx = x; gy = y;
  }
  public void draw(Graphics g) { } // 何もしない
  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; }
}
class Sankaku extends Figure {
  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) {
    super(c, 0.333333*(x0+x1+x2), 0.333333*(y0+y1+y2));
    dx0 = x0 - gx; dx1 = x1 - gx; dx2 = x2 - gx;
    dy0 = y0 - gy; dy1 = y1 - gy; dy2 = y2 - gy;
  }
  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);
  }
}
class Circle extends Figure {
  double rad;
  public Circle(Color c, double x, double y, double r) {
    super(c, x, y); rad = r;
  }
  public void draw(Graphics g) {
    g.setColor(color);
    g.fillOval((int)(gx-rad), (int)(gy-rad), (int)(rad*2), (int)(rad*2));
  }
}
```

こんどは「図形一般」を表すクラス Figure に共通の機能が納められ、Sankaku や Circle はそれぞれ固有の部分だけを持つのでずっとすっきりする。このように、継承を活用するときにはクラス構造をきちんと考えることが必要である。

演習 3 上の例題を打ち込んでそのまま動かせ。

演習 4 その例題に次のような図形を表すクラスを追加して、その図形も画面に表示するように直してみよ。どのクラスのサブクラスにするか、よく考えること。

a. 正方形 (回転しなくてもよい)。
b. 二重丸型 (大きい丸のなかにやや明るい色の小さい丸がはまっている)
c. 正 N 角形
d. その他自分の好きな形