プログラミング基礎'00　＃4

久野　靖*

2000.6.24

0　はじめに

前回の皆様の感想ですが、そろそろ「難しい」という人が増えて来たようです。プログラミング始めての人も多いわけですし、始めてでない人でもだいたいはオブジェクト指向は始めてでしょうから…自前のクラスを作る、というのは結構つらいだろうと思います。

まあそれは予想されていたことでして、今回はクラスを作りまくるのは少し押えて、GUI と GUI 部品の話を中心に説明します。前半では GUI 部品を配置するのを中心に扱いますから、「既にあるクラスを利用する」だけで済みます。

後半では GUI 部品に動作をつけますが、それにはクラスを作る必要があります。ただし「無名クラス」機能を使えばあんまりクラスらしくない書き方で済ませることができます。

しかし「クラスを作る」のはオブジェクト指向言語を活用するキーポイントには違いないので、できるだけマスターして頂きたいです。前回の演習問題の解説では当然クラスがたくさん出ますので、後でよく復習しておいてください。

1　前回の演習問題の解説

1.1　演習 3、演習 2

この辺の演習はまとめて 1 つのプログラムで示す。このため、例題プログラムの冒頭部分を手直しして、配列にさまざまな図形をまとめて入れておき、ループで一斉に処理することにする。

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

public class r3ex2 extends Applet {
  Figure[] figs = new Figure[]{
    new Circle(Color.blue, 100.0, 100.0, 30.0, 1.1, -3.0),
    new Circle(Color.red, 120.0, 80.0, 20.0, -0.8, 0.5),
    new Rect(new Color(200, 100, 250), 10.0, 10.0, 80.0, 50.0),
    new RegularPolygon(Color.yellow, 180.0, 120.0, 80.0, 3),
    new RegularPolygon(Color.green, 50.0, 120.0, 30.0, 7),
    new Hinomaru(220.0, 50.0, 20.0),
    new Hinomaru(200.0, 140.0, 15.0),
    new ShowText(Color.black, 100, 180, "Hello, World", 0.0, -3.0),
```

*筑波大学大学院経営システム科学専攻

```
};

public void paint(Graphics g) {
  for(int i = 0; i < figs.length; ++i) {
    figs[i].addTime(10.0); figs[i].draw(g);
  }
}
```

インタフェース Figure は同じ。

```
interface Figure {
  public void addTime(double time);
  public void draw(Graphics g);
}
```

クラス Circle はほとんど同じだが、「速度」を持たせるようにしてその速度に応じて中心位置が移動するようにした (演習 3)。

```
class Circle implements Figure {
  Color col;
  double cx, cy, rad, vx, vy;

  public Circle(Color c, double x, double y, double r,
                double vx1, double vy1) {
    col = c; cx = x; cy = y; rad = r; vx = vx1; vy = vy1;
  }
  public void addTime(double dt) {
    cx = cx + vx*dt; cy = cy + vy*dt;
  }
  public void draw(Graphics g) {
    g.setColor(col);
    g.fillOval((int)(cx-rad),(int)(cy-rad),(int)(rad*2),(int)(rad*2));
  }
}
```

さて、演習 2 に戻って「矩形」のクラス。

```
class Rect implements Figure {
  Color col;
  double x0, y0, width, height;

  public Rect(Color c, double x, double y, double w, double h) {
    col = c; x0 = x; y0 = y; width = w; height = h;
  }
  public void addTime(double dt) {
  }
  public void draw(Graphics g) {
    g.setColor(col);
    g.fillRect((int)(x0), (int)(y0), (int)(width), (int)(height));
```

```
  }
}
```

「正多角形」のクラス。これは動く版を前回例題でやったけど。

```
class RegularPolygon implements Figure {
  Color col;
  double cx, cy, rad;
  int num;
  int[] px, py;

  public RegularPolygon(Color c, double x, double y, double r, int n) {
    col = c; cx = x; cy = y; num = n; rad = r;
    px = new int[n]; py = new int[n];
  }
  public void addTime(double dt) {
  }
  public void draw(Graphics g) {
    for(int i = 0; i < num; ++i) {
      double theta = (2.0 * Math.PI) * i / num;
      px[i] = (int)(cx + rad * Math.cos(theta));
      py[i] = (int)(cy + rad * Math.sin(theta));
    }
    g.setColor(col); g.fillPolygon(px, py, num);
  }
}
```

「日の丸」。なお、「矩形」と「円」のクラスを同じ場所に置くことで日の丸にしようとしていた人がいるが、題意ではあくまでも「1 つのクラスで」日の丸を作るものとしている。

```
class Hinomaru implements Figure {
  double cx, cy, rad;

  public Hinomaru(double x, double y, double r) {
    cx = x; cy = y; rad = r;
  }
  public void addTime(double dt) {
  }
  public void draw(Graphics g) {
    g.setColor(Color.white);
    g.fillRect((int)(cx-rad*3.0), (int)(cy-rad*2.0),
               (int)(rad*6.0), (int)(rad*4.0));
    g.setColor(Color.red);
    g.fillOval((int)(cx-rad),(int)(cy-rad),(int)(rad*2.0),(int)(rad*2.0));
 }
}
```

最後に「文字列を表示」。これも速度を持たせるようにした。

```
  class ShowText implements Figure {
    Font fn = new Font("Helvetica", Font.BOLD, 24);
    Color col;
    double px, py, vx, vy;
    String text;

    public ShowText(Color c, double x, double y, String t,
                    double vx1, double vy1) {
      col = c; px = x; py = y; text = t; vx = vx1; vy = vy1;
    }
    public void addTime(double dt) {
      px = px + vx*dt; py = py + vy*dt;
    }
    public void draw(Graphics g) {
      g.setFont(fn); g.setColor(col);
      g.drawString(text, (int)(px), (int)(py));
    }
  }
}
```

1.2 演習 5

これも図形の扱いを配列に入れるように直した。

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

public class r3ex5 extends Applet {
  Figure[] figs = new Figure[]{
    new RotPolygon(Color.blue, 100.0, 100.0, 50.0, 5, -0.5),
    new RotPolygon(Color.red, 150.0, 80.0, 70.0, 3, 0.8),
    new MovingSquare(Color.green, 120.0, 80.0, 20.0, 30.0, 0.5),
    new ChangeColorText("Hello, World", 130.0, 40.0, 0.2),
  };
  boolean running;
  long basetime;

  public void paint(Graphics g) {
    long time = System.currentTimeMillis();
    double dt = 0.001*(time-basetime);
    basetime = time;
    for(int i = 0; i < figs.length; ++i) {
      figs[i].addTime(dt); figs[i].draw(g);
    }
  }
  public void start() {
```

```
  running = true;
  basetime = System.currentTimeMillis();
  (new Thread(new Timer())).start();
}
public void stop() {
  running = false;
}

class Timer implements Runnable {
  public void run() {
    while(running) {
      try { Thread.sleep(100); } catch(Exception e) { }
      repaint();
    }
  }
}

interface Figure {
  public void addTime(double time);
  public void draw(Graphics g);
}

class RotPolygon implements Figure {
  int[] px, py;
  Color col;
  int num;
  double cx, cy, rad, theta, vtheta;

  public RotPolygon(Color c, double x, double y, double r,
                    int n, double v) {
    col = c; num = n; cx = x; cy = y; rad = r; theta = 0.0; vtheta = v;
    px = new int[num]; py = new int[num];
  }
  public void addTime(double dt) {
    theta = theta + dt*vtheta;
  }
  public void draw(Graphics g) {
    for(int i = 0; i < num; ++i) {
      double t = theta + (2.0 * Math.PI) * i / num;
      px[i] = (int)(cx + rad * Math.cos(t));
      py[i] = (int)(cx + rad * Math.sin(t));
    }
    g.setColor(col); g.fillPolygon(px, py, num);
  }
}
```

ここまでは例題と同じ。さて、演習 5a は「動く図形」なので、正方形を円周運動させることにして、運動の半径と角速度を持たせるようにした。

```
    public class MovingSquare implements Figure {
      Color col;
      double px, py, len, rad, vtheta;
      double theta = 0.0;

      public MovingSquare(Color c, double x, double y, double l,
                          double r, double vt) {
        col = c; px = x; py = y; len = l; rad = r; vtheta = vt;
      }
      public void addTime(double dt) {
        theta = theta + vtheta*dt;
      }
      public void draw(Graphics g) {
        int x = (int)(px + rad*Math.cos(theta));
        int y = (int)(py + rad*Math.sin(theta));
        g.setColor(col); g.fillRect(x, y, (int)(len), (int)(len));
      }
    }
```

色の変化はいろいろなやり方があるが、Color.getHSBColor() を使うと色相、採度、明度の 3 要素から色が作れるので、色相だけを一定率で変化させるようにしてみた。

```
    class ChangeColorText implements Figure {
      Font fn = new Font("Helvetica", Font.BOLD, 24);
      String text;
      double px, py, vtheta;
      double theta = 0.0;

      public ChangeColorText(String t, double x, double y, double v) {
        text = t; px = x; py = y; vtheta = v;
      }
      public void addTime(double dt) {
        theta = theta + vtheta*dt;
      }
      public void draw(Graphics g) {
        g.setFont(fn);
        g.setColor(Color.getHSBColor((float)(theta % 1.0), 0.8f, 0.8f));
        g.drawString(text, (int)(px), (int)(py));
      }
    }
  }
```

演習 5c の「形の変化」はまあ回転や移動と同様なので (多角形にしておいて特定の頂点だけ動かすとか)、省略。

1.3 演習 7

これはほとんど例題と同じで、BounceCircle の中を変えて 7a〜7c をまとめて入れてみよう。

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

public class r3ex7 extends Applet {
  Figure[] figs = new Figure[]{
    new BounceCircle(Color.blue, 100.0, 100.0, 30.0, 30.0, 95.0),
    new BounceCircle(Color.red, 120.0, 80.0, 20.0, -85.0, 50.0),
  };
  boolean running;
  long basetime;

  public void paint(Graphics g) {
    long time = System.currentTimeMillis();
    double dt = 0.001*(time-basetime);
    basetime = time;
    for(int i = 0; i < figs.length; ++i) {
      figs[i].addTime(dt); figs[i].draw(g);
    }
  }
  public void start() {
    running = true;
    basetime = System.currentTimeMillis();
    (new Thread(new Timer())).start();
  }
  public void stop() {
    running = false;
  }

  class Timer implements Runnable {
    public void run() {
      while(running) {
        try { Thread.sleep(100); } catch(Exception e) { }
        repaint();
      }
    }
  }

  interface Figure {
    public void addTime(double time);
    public void draw(Graphics g);
  }
```

重力を入れるため「縦方向の加速度」を用意する。跳ね返りの時の速度の減衰率も入れる。

```
  class BounceCircle implements Figure {
    double alpha = 100.0;
    double ratio = 0.85;
    double width = 300.0;
    double height = 200.0;
    Color col;
    double cx, cy, rad, vx, vy;

    public BounceCircle(Color c, double x, double y, double r,
                 double vx1, double vy1) {
      col = c; cx = x; cy = y; rad = r; vx = vx1; vy = vy1;
    }
    public void addTime(double dt) {
      cx = cx + vx*dt; cy = cy + vy*dt;
      vy = vy + alpha*dt;
      if(cx-rad < 0 && vx < 0.0) { vx = -vx*ratio; }
      if(cx+rad > width && vx > 0.0) { vx = -vx*ratio; }
      if(cy-rad < 0 && vy < 0.0) { vy = -vy*ratio; }
      if(cy+rad > height && vy > 0.0) { vy = -vy*ratio; }
    }
    public void draw(Graphics g) {
      g.setColor(col);
      g.fillOval((int)(cx-rad),(int)(cy-rad),(int)(rad*2),(int)(rad*2));
    }
  }
}
```

まず加速度は速度に対して $V_y' = V_y + \alpha \triangle t$ の形で作用させる。はね返りが壁に食い込まないようにするためには、半径ぶんだけずらして判定すればよい (実はこれだとまだ不完全…わかりますか?)。そして、速度を反転するとき、減衰率を乗じる。

2 implements と extends

前回、インタフェースとは「複数のクラスをまとめて扱う」ためのものである、という話をした。実は、そのような働きは、クラスを使っても行える。つまり、前回の説明だと

```
interface Animal {
  public void forward(double speed); // 前進
  public void tell(String mesg);     // 話しかける
}
```

というのがあって、それに対して犬や猫のクラスを

```
class Dog implements Animal {
  double speed;
  public void forward(double speed) { 犬の動作… }
  public void tell(String mesg) { 犬の動作… }
}
```

```
class Cat implements Animal {
  double speed;
  public void forward(double speed) { 猫の動作… }
  public void tell(String mesg) { 猫の動作… }
}
```

のように定義し、それらを使う方では

```
Animal a = ... 犬または猫を入れる
...
a.tell("Come on!");  犬または猫に応じた動作
```

のように使うのだった。ところで、ここで「犬でも猫でも前進の動作方法はまったく同じ」だとすると、それをクラス Dog と Cat の両方に重複して書くのは無駄で嫌である。このような場合、Animal をクラスとしてそこに共通の動作を入れてしまうことができる。さらに、インスタンス変数やクラス変数も Animal で定義し、Dog や Cat ではそれをそのまま利用することができる。このようにした場合でも Animal 型はインタフェースだった時と同じような使い方ができる。

```
class Animal {
  double speed;
  public void forward(double speed) { 標準的な前進… }
  public void tell(String mesg) { 標準的な話しかけ… }
}
class Dog extends Animal {
  public void tell(String mesg) { 犬の動作… }
}
class Cat extends Animal {
  public void tell(String mesg) { 猫の動作… }
}
```

すると、Dog や Cat では定義していないメソッド forward() の動作は Animal の動作を引き継いでそのまま使うことになるし、変数 speed も Animal での定義を引き継いでそのまま利用する。このような機能を「継承」と呼ぶ。継承関係があるとき、動作等を引き継ぐ方のクラス (この場合は Animal) を「親クラス」「スーパークラス」、引き継ぐ方の (この場合は Dog や Cat) を「子クラス」「サブクラス」と呼ぶ。

なお、tell() についても子クラスで定義しなければ親クラスの動作をそのまま継承するが、ここでは自前のものを定義しているので、そちらが有効になる。これを「差し替え」(オーバライド、override) と呼ぶ。継承やオーバライドを活用するのは高度な技になるので、ここでは実習しない。ただ、オブジェクト指向言語ではこのような機能も用意されている、ということは一応知っておいていただければと思う。

3 GUI と GUI 部品

GUI(Graphical User Interface) とは、最も広い意味でいえば「計算機のグラフィクス能力を活用したユーザインタフェース」ということになり、これには、ほとんど無限の多様性がある。しかし現実には GUI をもっと限定的に、「画面上にいろいろな『部品』(GUI 部品) が配置され、マウス等でそれを操作することで計算機とやりとりするようなインタフェース」程度の意味で捉えることが多い。部品の例としては「ボタン」「入力欄」「メニュー」などがある。

このようなスタイルの利点は、利用者にとってはさまざまなプログラムで使われている部品に共通性があり、その操作方法をいちいち覚えなくても済むこと、またソフトウェア作成者にとっては部品の実現部分は誰かが

書いたものを持って来て再利用するだけで済んだり、さらに進んで部品を「見たまま方式」で直接配置しながらユーザインタフェースを設計/構築するツールが利用できたりして生産性が上がるということがある (しかしその半面、無批判に GUI 部品によるインタフェースばかり使うことは、人間の能力を殺しているのではないかという気もする)。

まあ疑問はさておき、Java で GUI 部品を使う方法について学ぼう。なお、この部分は JDK 1.1 と 1.2 で変化があったところで、1.2 では「Swing」と呼ばれる新しい部品群が追加されている。ここでは多くのブラウザでサポートされている、1.1 から存在する部品だけを扱う。これらの部品は `java.awt` パッケージに含まれている。

どのような GUI 部品でも、画面上に配置し、色やフォントを指定して表示を行わせるというところは共通なので、そのためのメソッド (共通の親クラス Component に定義されている) の主なものを挙げておこう。

- `setBounds(int, int, int, int)` — 画面上の位置 (x,y) と幅と高さを設定
- `setForeground(Color)` — 前景色を設定
- `setBackground(Color)` — 背景色を設定
- `setFont(Font)` — フォントを設定

では早速、「ラベルとボタンを 2 つ持ったアプレット」という例題を見ていただこう。

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

public class R4Sample1 extends Applet {
  Font fn = new Font("Helvetica", Font.BOLD, 24);
  TextField t1 = new TextField("text...");
  Button b1 = new Button("B1");
  Button b2 = new Button("B2");
  Label l1 = new Label("label...");
  public void init() {
    setLayout(null);
    add(t1); t1.setBounds(20, 20, 200, 40);
    add(b1); b1.setFont(fn); b1.setBounds(20, 80, 60, 40);
    add(b2); b2.setFont(fn); b2.setBounds(120, 80, 60, 40);
    add(l1); l1.setForeground(Color.red);
    l1.setBackground(Color.yellow); l1.setBounds(20, 140, 200, 40);
  }
}
```

このアプレットは初期設定しかしない (あとの動作は GUI 部品が勝手にやってくれる) ので、メソッド `init()` だけを持つ。アプレットが持っているメソッド `add()` を呼ぶことで、部品をアプレット画面に「追加」できる。最初の `setLayout(null)` は自動配置機能を off にしている (そうしないと場所指定が効かない)。

部品を追加したら、そのあと部品のメソッドを使って位置、フォントなどを設定している。この例題では部品としてテキスト入力欄 (文字列を入力する部品)、ボタン (押す」ことができる部品)、ラベル (文字列を表示する部品) を取り上げた。これらを含め、たとえば次のような部品がある。

- `Label` — 文字を表示するだけの部品
- `Button` — 押しボタン
- `Choice` — 選択メニュー

- `Checkbox` — チェックボックス
- `CheckboxGroup` — チェックボックスをグループ化するための部品
- `TextField` — テキスト入力欄
- `TextArea` — 複数行テキスト入力欄
- `List` — 複数項目の並んだリスト
- `Frame` — 独立した窓*
- `MenuBar` — メニューバー*
- `Menu` — プルダウンメニュー*
- `PopupMenu` — ポップアップメニュー*

演習 1 上の例題を打ち込んでそのまま動かせ。動いたら色やフォントや配置を調整してみよ。

演習 2 上の例題に出てこなかった面白そうな部品を追加してみよ (API ドキュメントでメソッド等を調べられる)。ただし*がついているのはこれまでの説明だけだと難しいのでやめておいた方がよい。

演習 3 次のような GUI プログラムのインタフェース部分だけを設計し (必ず紙にラフスケッチを描くこと)、その GUI 部分だけを Java で作ってみよ。動作本体は作らなくてよい。

a. 華氏の温度を摂氏の温度に変換する。
b. 2 つの数値を入力し、その合計を求める。
c. 簡単な電卓 (機能は適当に設計してよい)。
d. 住宅ローンの計算 (〃)。
e. その他自分の好きなもの。

4 部品の動作を指定するには

これまでのところ、部品は「勝手に動作」するけれど、たとえばボタンを押してもそれ以上何も起きなかった (あたりまえだけど)。GUI 部品を役に立てるためには、「ボタンが押されたらこれこれの動作をする」といったコードが必要である。そのためにどのような仕掛けが使われているかを説明しよう。

- GUI 部品は `addActionListener(ActionListner)` というメソッドを持ち、これをもちいて ActionListener オブジェクトを設定できる。
- ActionListener は実はインタフェースであり、`actionPerformed(ActionEvent)` というメソッドのみを定義している。
- そこで、ActionListener インタフェースを implements したクラスを用意し、そのメソッド `actionPerformed()` で GUI 部品が「押された」時の動作を指定した上、このクラスのインスタンスを `addActionListener()` で GUI 部品に設定する。

では実際にこの方法で先のアプレットに「ボタン B1 を押すと入力欄の文字列をひっくり返して表示する」というのを実現してみよう。ActionListener 関係のクラスはパッケージ `java.awt.event` にあるので `import` が増えているのに注意。

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

```
public class R4Sample2 extends Applet {
  Font fn = new Font("Helvetica", Font.BOLD, 24);
  TextField t1 = new TextField("text...");
  Button b1 = new Button("B1");
  Button b2 = new Button("B2");
  Label l1 = new Label("label...");
  public void init() {
    setLayout(null);
    add(t1); t1.setBounds(20, 20, 200, 40);
    add(b1); b1.setFont(fn); b1.setBounds(20, 80, 60, 40);
    add(b2); b2.setFont(fn); b2.setBounds(120, 80, 60, 40);
    add(l1); l1.setForeground(Color.red);
    l1.setBackground(Color.yellow); l1.setBounds(20, 140, 200, 40);
    b1.addActionListener(new MyAdapter());
  }

  class MyAdapter implements ActionListener {
    public void actionPerformed(ActionEvent evt) {
      String str = t1.getText();
      String res = "";
      for(int i = str.length()-1; i >= 0; --i) {
        res = res + str.charAt(i);
      }
      l1.setText(res); t1.setText("");
    }
  }
}
```

すなわち、ボタン b1 の動作として MyAdapter クラスのインスタンスを生成して設定しているので、ボタンが押されるとこのクラスの actionPerformed() が呼び出される。このメソッドの中では入力欄 t1 のテキスト (String 型) を取り出し、左右反転して、ラベル l1 に設定している。

5 無名の内部クラス

ところで、クラス MyAdapter というのは 1 箇所でしか参照されていない。そのようなもののために名前を考えてクラスを作るのはわずらわしい (特に動作が違うボタンがいくつもあったら、そのボタンごとに別のクラスが必要になるので名前を考えるのも大変である)。

そこで実は!「無名の内部クラス」という、このような「ちょっとした用途の」クラスのためにいちいち名前を考えたり覚えたりしなくて済むような機能が用意されている。無名の内部クラス機能は、次の条件を満たすときに使う。

- 内部クラスは何らかのインタフェースを implements している。または何らかのクラスを extends している。
- その内部クラスのインスタンスを 1 回だけ生成する。

このとき、上の例題の書き方だと次のようになる。

```
class MyXXXClass extends/implements YYY {
  // クラス定義本体
}
..... new MyXXXClass(引数) ...
```

ただし new する箇所と MyXXXClass の定義の順序関係は上の通りでなくても (離れていても) よい。これを無名内部クラスにする場合は次のようにする。

```
..... new YYY(引数) {
  // クラス定義本体
} ...
```

つまり、new するところにクラス定義を「そっくり」埋め込んでしまうわけである。上の例題をこの書き方に直したものを示す。

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

public class R4Sample3 extends Applet {
  Font fn = new Font("Helvetica", Font.BOLD, 24);
  TextField t1 = new TextField("text...");
  Button b1 = new Button("B1");
  Button b2 = new Button("B2");
  Label l1 = new Label("label...");
  public void init() {
    setLayout(null);
    add(t1); t1.setBounds(20, 20, 200, 40);
    add(b1); b1.setFont(fn); b1.setBounds(20, 80, 60, 40);
    add(b2); b2.setFont(fn); b2.setBounds(120, 80, 60, 40);
    add(l1); l1.setForeground(Color.red);
    l1.setBackground(Color.yellow); l1.setBounds(20, 140, 200, 40);
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        String str = t1.getText();
        String res = "";
        for(int i = str.length()-1; i >= 0; --i) {
          res = res + str.charAt(i);
        }
        l1.setText(res); t1.setText("");
      }
    });
  }
}
```

演習 3 R4Sample3 の例題そのまま打ち込んで動かせ。

演習 4 動いたらボタン B2 に次のような動作をつけてみよ (`java.lang.String` クラスの API ドキュメントをよく探すこと)。

a. B2 が押されたら入力欄の文字列中の小文字を大文字にしたものを表示する。
b. B2 が押されたら入力欄の文字列中の「a」を「*」に置き換えたものを表示する。
c. B2 が押されたら入力欄の文字列中の「a、i、u、e、o」すべてを「*」に置き換えたものを表示する。

ヒント: B2 が押された場合の処理を無名内部クラスで書く場合、次のような形になる (これをどこに入れるかくらいは自分で考えること)。

```
b2.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent evt) {
    処理の内容…
  }
});
```

6 例外とエラー処理

ところで、actionPerformed() の中でさまざまな処理をしていてエラーが起きたときはどうすればいいだろう? アプレットがエラーで止まってしまうのは不親切なので避けたい。実はこのような場合に、前回の例題でちょっと出て来た「例外をうけとめる」機能を使うとよい。例外 (各種のエラー) を受け止めるのは次のような構文による。

```
try {
  ... 各種の動作 ... ←この中で起きた例外を
} catch(Exception ex) {
  ... ←受け止めてここで処理できる。
}
```

これを書かないと起きた例外は呼び出し元を遡って伝播していき、最後は `main()` を起動したところやブラウザの中でエラーとして処理される (つまりアプレットやプログラムは止まってしまう)。受け止めるようにしてあれば、処理した後は普通に実行が続けられる。

GUI を使ったアプレットであれば、例外はとにかく上記の方法で受け止めて、例外情報 (上記では変数 `ex` に入る) を文字列に変換して表示してやれば、大変親切ではないにせよ、何とか許されるかと思う。そういう処理を入れた例を見てみよう。

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

public class R4Sample4 extends Applet {
  Font fn = new Font("Helvetica", Font.BOLD, 24);
  TextField t1 = new TextField("1");
  Button b1 = new Button("+1");
  Button b2 = new Button("-1");
  Label l1 = new Label("");
  public void init() {
    setLayout(null);
    add(t1); t1.setBounds(20, 20, 200, 40);
    add(b1); b1.setFont(fn); b1.setBounds(20, 80, 60, 40);
```

```
    add(b2); b2.setFont(fn); b2.setBounds(120, 80, 60, 40);
    add(l1); l1.setForeground(Color.red);
    l1.setBackground(Color.yellow); l1.setBounds(20, 140, 300, 40);
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        try {
          int i = (new Integer(t1.getText())).intValue();
          t1.setText("" + (i+1));
        } catch(Exception ex) { l1.setText("" + ex); }
      }
    });
    b2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        try {
          int i = (new Integer(t1.getText())).intValue();
          t1.setText("" + (i-1));
        } catch(Exception ex) { l1.setText("" + ex); }
      }
    });
  }
}
```

この例ではボタンの名称やラベルの幅などは適宜調整してある。また、さっきと違って計算した結果はテキスト入力欄の方に設定される。そして、例外が起きたらそれを受け止め、文字列に変換してラベル l1 に表示している。たとえば入力欄にある文字列が数値の形をしていないとちゃんとエラーが表示される。

演習 5 演習 3 で作ったプログラムに動作をつけてみよ。

7 おまけ: GUI で絵を制御する

では次に、前回までやってきた絵のパラメタを GUI 経由で変更する、というのを試してみよう。

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

public class R4Sample5 extends Applet {
  Circle c = new Circle(Color.red, 100.0, 150.0, 30.0);
  Label l1 = new Label("");
  Button b1 = new Button("SET!");
  TextField t_x = new TextField("100");
  TextField t_y = new TextField("150");
  TextField t_rad = new TextField("30");
  TextField t_r = new TextField("255");
  TextField t_g = new TextField("0");
  TextField t_b = new TextField("0");
  public void init() {
```

```
    setLayout(null); l1.setBackground(Color.white);
    add(t_x); t_x.setBounds(20, 20, 40, 30);
    add(t_y); t_y.setBounds(80, 20, 40, 30);
    add(t_rad); t_rad.setBounds(140, 20, 40, 30);
    add(t_r); t_r.setBounds(20, 60, 40, 30);
    add(t_g); t_g.setBounds(80, 60, 40, 30);
    add(t_b); t_b.setBounds(140, 60, 40, 30);
    add(b1); b1.setBounds(200, 60, 40, 30);
    add(l1); l1.setBounds(20, 100, 300, 30);
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        try {
          c.moveTo(new Double(t_x.getText()).doubleValue(),
                   new Double(t_y.getText()).doubleValue());
          c.setRadius(new Double(t_rad.getText()).doubleValue());
          c.setColor(new Color(new Integer(t_r.getText()).intValue(),
                               new Integer(t_g.getText()).intValue(),
                               new Integer(t_b.getText()).intValue()));
          repaint();
        } catch(Exception ex) { l1.setText(ex.toString()); }
      }
    });
  }
  public void paint(Graphics g) { c.draw(g); }
  class Circle {
    Color c0;
    double cx, cy, rad;
    public Circle(Color c, double x, double y, double r) {
      c0 = c; cx = x; cy = y; rad = r;
    }
    public void draw(Graphics g) {
      g.setColor(c0);
      g.fillOval((int)(cx-rad), (int)(cy-rad), (int)(rad*2), (int)(rad*2));
    }
    double getX() { return cx; }
    double getY() { return cy; }
    double getRadius() { return rad; }
    Color getColor() { return c0; }
    void moveTo(double x, double y) { cx = x; cy = y; }
    void setRadius(double r) { rad = r; }
    void setColor(Color c) { c0 = c; }
  }
}
```

すなわち、円の X 座標、Y 座標、半径、および色の R、G、B 値を TextField から打ち込んで自由に設定できるようにしてみた。結構くらくらしますか? これまでに学んだことを組み合わせればこれくらいできる、ということで。