情報科学特殊講義'2000 ＃7

久野　靖*

2000.1.24-27

0　はじめに

今回は前回の演習の解説はなるべく簡単に済ませて、しばらく続いていた「お絵描き」をちょっと離れて「GUI 部品」をやります。GUI 部品を使えば、いろいろと「訳に立ちそうっぽい」プログラムが作れるようになります。

1　前回の練習問題の解説

前回の問題は色々遊べるが、全部解説すると大変なのでいくつかの項目だけ解説する。まずそれを全部盛り込んだアプレットを見ていただく。ここでは「飛ぶ円 (の改造版)」と「自転しつつ公転する星」をまとめて扱っている。

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

public class r6ex4a extends Applet implements Runnable {
  Animation a[] = new Animation[20];
  int count = 0;
  boolean running = false;

  public void init() {
    Rectangle r = new Rectangle(0, 0, 400, 300);
    a[count] = new ACircle(Color.red, 100, 100, 20, 35, 74, r); ++count;
    a[count] = new ACircle(Color.blue, 80, 150, 30, -28, 52, r); ++count;
    a[count] = new AMovingStar(Color.green, 5, 20, 50, 70, 30, r);
    ++count;
    setBackground(Color.yellow);
  }
  public void start() { running = true; (new Thread(this)).start();  }
  public void stop() { running = false; }
  public void paint(Graphics g) {
    for(int i = 0; i < count; ++i) a[i].draw(g);
  }
  public void run() {
    long basetime = System.currentTimeMillis();
    while(running) {
```

*筑波大学大学院経営システム科学専攻

```
      try { Thread.sleep(100); } catch(Exception e) { }
      long time = System.currentTimeMillis();
      double dt = 0.001*(time-basetime); basetime = time;
      for(int i = 0; i < count; ++i) a[i].addTime(dt);
      repaint();
    }
  }
}

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

class ACircle implements Animation {
  Rectangle rect;
  Color col;
  double gx, gy, vx, vy, rad;
  double time = 0.0;
  public ACircle(Color c, double x, double y, double r,
                 double vx0, double vy0, Rectangle r0) {
    col = c; gx = x; gy = y; rad = r; vx = vx0; vy = vy0; rect = r0;
  }
  public void draw(Graphics g) {
    g.setColor(col);
    g.fillOval((int)(gx-rad), (int)(gy-rad), 2*(int)rad, 2*(int)rad);
  }
  public void addTime(double dt) {
    vy += 40.0*dt; gx += vx*dt; gy += vy*dt; time += dt;
    if(gx < rect.x && vx < 0) vx = -vx;
    if(gx > rect.x+rect.width && vx > 0) vx = -vx;
    if(gy < rect.y && vy < 0) gy += rect.height;
    if(gy+rad > rect.y+rect.height && vy > 0) {
      vy = -1.1*vy; gy -= 2;
     }
  }
}

class AMovingStar implements Animation {
  Rectangle rect;
  Color col;
  int num;
  double gx, gy, cx, cy, rad1, rad2;
  double theta = 0.0; double vtheta = 1.0;
  double gamma = 0.0; double vgamma = 1.7; double rgamma = 200.0;
  double time = 0.0;
  public AMovingStar(Color c, int n, double r1, double r2,  double x, double y,
                     Rectangle r) {
```

```
    col = c; num = n; rad1 = r1; rad2 = r2; cx = x; cy = y;
    rect = r;
  }
  public void draw(Graphics g) {
    int[] x = new int[num*2]; int[] y = new int[num*2];
    double dt = 2.0*Math.PI / (2*num); double t = theta;
    for(int i = 0; i < num*2; ++i) {
      if(i % 2 == 0) {
        x[i] = (int)(gx + rad1*Math.cos(t));
        y[i] = (int)(gy + rad1*Math.sin(t));
      } else {
        x[i] = (int)(gx + rad2*Math.cos(t));
        y[i] = (int)(gy + rad2*Math.sin(t));
      }
      t += dt;
    }
    g.setColor(col); g.fillPolygon(x, y, 2*num);
  }
  public void addTime(double dt) {
    time += dt; theta += vtheta;
    gx = cx + rgamma*Math.cos(time);
    gy = cy + rgamma*Math.sin(time);
  }
}
```

では項目ごとに解説。まず次の 3 つは ACircle を改造してある。

- 重力を入れる方法 — これは簡単で「addTime()」の中で $V_y' = V_y + Gdt$ のようなことをすればよい。G の値は画面で見てみて適当に選ぶ。
- ワープする方法 — これは、ある条件 (たとえば上側の壁に当たった等) の成り立った時点で座標の値を変更すればよい。
- めりこまなくする方法 — これは、判定条件を「重心の位置」ではなく「当たった場所の位置」にすればよい。

自転しつつ公転する星は AMovingStar で実現されているが、星を描くところは前回の解説と同様。

- 自転させるには、回転角度を時刻に応じて変わるようにすればよい。
- 公転させるには、座標を $(x + rcos(t), y + rsin(t))$ のようにすればよい。ただし t は時刻に応じて変化させる。

2 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 (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 R7Sample1 extends Applet {
  Font fn = new Font("Helvetica", Font.BOLD, 24);
  TextField txf = new TextField("text...");
  Button b1 = new Button("B1");
  Button b2 = new Button("B2");
  public void init() {
    setLayout(null);
    add(txf); txf.setForeground(Color.red); txf.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);
  }
}
```

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) とボタン (押す」ことができる部品) 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. 簡単な電卓 (機能は適当に設計してよい)。
c. 住宅ローンの計算 (〃)。
d. その他自分の好きなもの。

3 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「ボタンが押されるごとにラベルに『*』が増える」というのを実現してみよう。

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

public class R7Sample2 extends Applet {
  Font fn = new Font("Helvetica", Font.BOLD, 16);
  Label l1 = new Label("*");
  Button b1 = new Button("Press Me!");
  public void init() {
    setLayout(null); l1.setBackground(Color.white);
    add(l1); l1.setFont(fn); l1.setBounds(20, 20, 160, 40);
    add(b1); b1.setFont(fn); b1.setBounds(20, 80, 100, 40);
```

```
    b1.addActionListener(new R7Sample2Adapter(this));
  }
  public void press() {
    l1.setText(l1.getText() + "*"); repaint();
  }
}

class R7Sample2Adapter implements ActionListener {
  R7Sample2 app;
  public R7Sample2Adapter(R7Sample2 a) { app = a; }
  public void actionPerformed(ActionEvent e) { app.press(); }
}
```

すなわち、R7Sample2Adapter というアダプタクラスはコンストラクタの他にはメソッド actionPerformed() だけを持ち、その中でアプレットオブジェクトの press() というメソッドを呼んでいる。このクラスのインスタンスをボタンの addActionListener() で設定してあるので、ボタンを押すとアダプタを介して press() が呼び出され、ラベルの「*」の数が増える。

しかし、こんなことをしなくても R7Sample2 クラス自体に ActionListener インタフェースを implements させたらもっと簡単になる、と思いませんでしたか? この場合はそうなのだけど、ボタンが複数あったりしたら面倒なことになるでしょう?

4 内部クラス

しかし、もっと別の方法で上のコードを簡単にすることができる。それは「内部クラス」と呼ばれる機能を使うことである。内部クラスとは、クラスの中で「下請け用の」クラスを定義することであり、たとえば次のような形になる。

```
public class X {  // 外側のクラス
  ...
  public void m1() { // クラス X のインスタンスメソッド
    ...
  }
  public void m2() { // クラス X のインスタンスメソッド
    ...
    new X1(...);     // 内部クラス X1 のインスタンス生成
  }
  class X1 { // X の内部クラス X1
    ...
    public void m3() { // 内部クラス X1 のメソッド
      ...
      ... m1(...) ...  // X のメソッドも呼べる
    }
  }
  class X2 { // X の内部クラス X2
    ...
  }
}
```

内部クラスには次のような性質がある。

- 内部クラスは public と指定されていない限り、それを含むクラスの中でだけその名前が使える。ただし、そのインスタンスを外側のクラスからさらに外側に渡すことはできる。
- 内部クラスは static と指定されていない限り、そのインスタンスはそれを含むクラスのインスタンスメソッドの中でだけ生成できそして、内部クラスのインスタンスは、外側の (内部クラスのインスタンスを生成したときの) インスタンスを「覚えて」いて、外側のインスタンスのインスタンス変数を参照しだりメソッドを呼んだりできる。
- 内部クラスが static と指定されていた場合は、上のような制約はないが、その代り外側のインスタンスを「覚えておく」という機能も持たない。

これを利用すれば、上のコードは次のようにできる。

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

public class R7Sample3 extends Applet {
  Font fn = new Font("Helvetica", Font.BOLD, 16);
  Label l1 = new Label("*");
  Button b1 = new Button("Press Me!");
  public void init() {
    setLayout(null); l1.setBackground(Color.white);
    add(l1); l1.setFont(fn); l1.setBounds(20, 20, 160, 40);
    add(b1); b1.setFont(fn); b1.setBounds(20, 80, 100, 40);
    b1.addActionListener(new MyAdapter());
  }
  class MyAdapter implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      l1.setText(l1.getText() + "*"); repaint();
    }
  }
}
```

だいぶ簡単になったでしょう? しかし実は! もっと短くする方法が提供されている。それには、「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を使う。

無名の内部クラス機能は、次の条件を満たすときに使う。

- 内部クラスが何らかのインタフェースを implements しているか、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 R7Sample4 extends Applet {
  Font fn = new Font("Helvetica", Font.BOLD, 16);
  Label l1 = new Label("*");
  Button b1 = new Button("Press Me!");
  public void init() {
    setLayout(null); l1.setBackground(Color.white);
    add(l1); l1.setFont(fn); l1.setBounds(20, 20, 160, 40);
    add(b1); b1.setFont(fn); b1.setBounds(20, 80, 100, 40);
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        l1.setText(l1.getText() + "*"); repaint();
      }
    });
  }
}
```

5 例外を活用したエラー処理

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。それには、前にやった try...catch で例外を受け止めて、どんな例外があったよ、ということだけとりあえずどこかに表示すれば、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 R7Sample5 extends Applet {
  Font fn = new Font("Helvetica", Font.BOLD, 16);
  TextField t1 = new TextField("1");
  Label l1 = new Label("");
  Button b1 = new Button("+1");
  Button b2 = new Button("-1");
  public void init() {
    setLayout(null); l1.setBackground(Color.white);
    add(t1); t1.setFont(fn); t1.setBounds(20, 20, 160, 40);
    add(b1); b1.setFont(fn); b1.setBounds(20, 80, 60, 40);
    add(b2); b2.setFont(fn); b2.setBounds(100, 80, 60, 40);
```

```
    add(l1); l1.setBounds(20, 140, 300, 40);
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        try {
          t1.setText("" + (new Integer(t1.getText()).intValue() + 1));
        } catch(Exception ex) { l1.setText(ex.toString()); }
      }
    });
    b2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        try {
          t1.setText("" + (new Integer(t1.getText()).intValue() - 1));
        } catch(Exception ex) { l1.setText(ex.toString()); }
      }
    });
  }
}
```

すなわち、例外が起きたらそれを受け止め、toString() で文字列に変換してラベル l1 に setText() することで表示している。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 4 R7Sample2-5 の例題から好きなものを 1 つ打ち込んで動かしてみよ。

演習 5 演習 3 で作ったプログラムに動作をつけてみよ。内部クラスを使うかどうかなどは好きに決めてよい。