計算機プログラミング'99　＃6

久野　靖*

1999.10.13

0　はじめに

前回の予告通り、今回はGUIとGUI部品を取り上げます。また、JDK 1.1以降ではGUI部品の動作を実装するときにインタフェースを利用することがほぼ必須なので、その辺の思想についても取り上げ、最後に「フレームワーク」の考え方を学んでおきましょう。

1　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、その共通部分をくくり出す親クラスがあると当然思われますね? それがクラスComponentである。そこに定義されている、各種設定用メソッド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。

*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 R6Sample1 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。実はアプレット自体が Component のサブクラスであり、その `add()` メソッド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. 電卓 (機能は適当に設計してよい)。
c. ローンの計算 (〃)。
d. その他自分の好きなもの。

2 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。

それにはどうしたらいいだろう? たとえば「ボタンのサブクラスを作って『動作』メソッドをオーバライドする」???

実は JDK 1.0.2 までの Java では実際その方法が使われていたのだけれど、その方法だと動作の内容ごとにどんどん別のクラスを作るはめになり、大変わずらわしい。このため、JDK 1.1 以降ではそれに代って次のような方法が取られている。

- 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 R6Sample2 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 R6Sample2Adapter(this));
  }
  public void press() {
    l1.setText(l1.getText() + "*"); repaint();
  }
}

class R6Sample2Adapter implements ActionListener {
  R6Sample2 app;
  public R6Sample2Adapter(R6Sample2 a) { app = a; }
  public void actionPerformed(ActionEvent e) { app.press(); }
}
```

すなわち、R6Sample2Adapter 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 R6Sample2 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?

3 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、内部クラスを使うことである。これまできちんと説明してこなかったけれど、内部クラスは次のような性質がある。

> 内部クラスを生成するのは、インスタンスメソッドの中でなければいけない。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。

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 R6Sample3 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。

- 内部クラスは、何らかのクラスのサブクラスか、何らかのインタフェースを implements 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 R6Sample4 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();
      }
    });
  }
}
```

演習 4 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。

4 モデルとインタフェースの分離

さて、これまでのところは「GUI を作って、その動作部分でアプリケーショのロジックを実現する」という形でやってきた。しかし、よく考えると、「アプリケーションのロジック」(つまりアプリケーションの「モデル」とそのインタフェースは本来独立している、つまり 1 つのアプリケーションにさまざまなロジックがつけられるようなものであるはずである。

これを実現するには、次のようにすればよい (1)。

- まず、モデルが実装するインタフェースを設計する。
- アプリケーションロジックを、そのインタフェースを implements するクラスとして作成する。
- GUI の部分は、そのインタフェースを利用するクラスとして作成する。

たとえば、次のようなモデルを考える

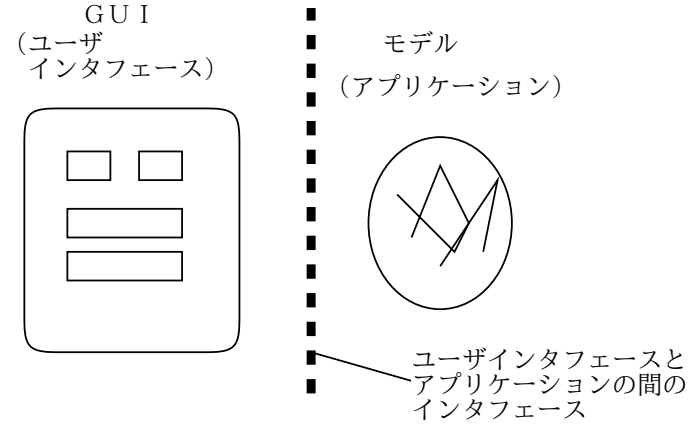

図 1: ユーザインタフェースとアプリケーションの分離

```
interface R6Model {
  public String getTitle();
  public String getButton1Label();
  public String getButton2Label();
  public String button1Action(String s);
  public String button2Action(String s);
}
```

これは、「タイトルがあり、ボタンが 2 つあり、入力欄が 1 つあり、表示欄が 1 つあり、2 つのボタンそれぞれについて入力欄に基づいて動作を行い、結果を表示欄に表示する」というアプリケーションのモデルになっている。次にこれを implements する「摂氏華氏/華氏摂氏変換クラス」を示そう。

```
class R6Sample5Model implements R6Model {
  public String getTitle() { return "Temperature Calc."; }
  public String getButton1Label() { return "F to C"; }
  public String getButton2Label() { return "C to F"; }
  public String button1Action(String s) {
    try {
      double f = new Double(s).doubleValue();
      double c = 5.0*(f - 32.0)/9.0;
      return (new Double(c)).toString();
    } catch(Exception e) { return e.toString(); }
  }
  public String button2Action(String s) {
    try {
      double c = new Double(s).doubleValue();
      double f = 9.0*c/5.0 + 32.0;
      return (new Double(f)).toString();
    } catch(Exception e) { return e.toString(); }
  }
}
```

つまり、それぞれのボタンが押された時に変換を行うというだけである。では最後に、アプレットクラスを含めた全体を示そう。

```
import java.applet.Applet;
```

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

public class R6Sample5 extends Applet {
  R6Model model = new R6Sample5Model();
  Font fn = new Font("Helvetica", Font.BOLD, 16);
  Label l1 = new Label(model.getTitle());
  Label l2 = new Label("");
  TextField tf = new TextField("");
  Button b1 = new Button(model.getButton1Label());
  Button b2 = new Button(model.getButton2Label());
  public void init() {
    setLayout(null); l2.setBackground(Color.white);
    add(l1); l1.setFont(fn); l1.setBounds(20, 10, 160, 40);
    add(l2); l2.setFont(fn); l2.setBounds(20, 60, 160, 40);
    add(tf); tf.setFont(fn); tf.setBounds(20, 110, 160, 40);
    add(b1); b1.setFont(fn); b1.setBounds(20, 160, 80, 40);
    add(b2); b2.setFont(fn); b2.setBounds(120, 160, 80, 40);
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        l2.setText(model.button1Action(tf.getText()));
        tf.setText(""); repaint();
      }
    });
    b2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        l2.setText(model.button2Action(tf.getText()));
        tf.setText(""); repaint();
      }
    });
  }
}

interface R6Model {
  public String getTitle();
  public String getButton1Label();
  public String getButton2Label();
  public String button1Action(String s);
  public String button2Action(String s);
}

class R6Sample5Model implements R6Model {
  public String getTitle() { return "Temperature Calc."; }
  public String getButton1Label() { return "F to C"; }
  public String getButton2Label() { return "C to F"; }
  public String button1Action(String s) {
    try {
      double f = new Double(s).doubleValue();
```

```
      double c = 5.0*(f - 32.0)/9.0;
      return (new Double(c)).toString();
    } catch(Exception e) { return e.toString(); }
  }
  public String button2Action(String s) {
    try {
      double c = new Double(s).doubleValue();
      double f = 9.0*c/5.0 + 32.0;
      return (new Double(f)).toString();
    } catch(Exception e) { return e.toString(); }
  }
}
```

明らかに、このアプレットはモデル (R6Model インタフェースを実現するクラス) をいろいろ差し替えることでさまざまなアプリケーションになる。また逆に、アプレットクラスの側を差し替えることで同じ温度変換でもいろいろなインタフェースが作れる。

演習 5 上記のプログラムをそのまま動かせ。

演習 6 モデルクラスを別のものに差し替えて、同じインタフェースのままで別のアプリケーションにしてみよ。特に、上の例ではモデルの中に状態を持たなかったが、持つようにしてみるとよい (例: 積算電卓など)。

演習 7 アプレットクラスを別のものに差し替えて、同じアプリケーショのままで別のインタフェースにしてみよ。

演習 8 演習 4 でやったものを、モデルとインタフェースを分離するように改造してみよ。