情報科学 2007　久野クラス　＃13

久野　靖*

2008.1.25

はじめに

今回はいよいよ、Java を用いて「グラフィクスと GUI」をやりましょう。既存のクラスを沢山使うので、ライブラリクラスのドキュメント (API ドキュメント) の見かたも身につけて頂きたいと思っています。その前に、あと課題 10B の解説を少しだけやります (ゲームとか状態空間はいいですよね)。

1　課題 10B の解説

1.1　演習 1、演習 2

スタックとキューを単リストで実装するのは慣れれば簡単。

```
Cell = Struct.new(:data, :next)

class Stack2
  def initialize
    @top = nil
  end
  def isempty
    return @top == nil
  end
  def push(x)
    @top = Cell.new(x, @top)
  end
  def pop
    x = @top.data; @top = @top.next; return x
  end
end

class Queue2
  def initialize
    @top = @last = nil
  end
  def isempty
    return @top == nil
  end
  def enq(x)
```

*筑波大学大学院経営システム科学専攻

```
    if isempty
      @top = @last = Cell.new(x, nil)
    else
      @last.next = Cell.new(x, nil); @last = @last.next
    end
  end
  def deq
    if @top == @last
      x = @top.data; @top = @last = nil; return x
    else
      x = @top.data; @top = @top.next; return x
    end
  end
end
```

キューの場合は「空っぽ」を特別扱いするのでちょっとだけ長くなる。

1.2 演習 3

路線図のデータをどのように表現するかをまず考える。ここでは、1 つの駅 (ノード) を次の 3 つのフィールドを持つレコードとする。

- name — 駅の名前
- arr — 隣接する駅 (ノード) を並べたもの
- dist — 出発駅からの距離 (不明なら-1)

そして、駅名からノード検索するハッシュ表をグローバル変数として用意する。最初にこのデータ構造を構築するメソッドを示そう。

```
def prepare
  $graph = Hash.new()
  cn("赤羽", "池袋"); cn("赤羽", "田端"); cn("池袋", "田端")
  cn("八王子", "立川"); cn("立川", "新宿"); cn("池袋", "新宿")
  cn("新宿", "お茶の水"); cn("お茶の水", "秋葉原"); cn("田端", "秋葉原")
  cn("お茶の水", "東京"); cn("秋葉原", "東京"); cn("東京", "品川")
  cn("新宿", "大崎"); cn("大崎", "品川"); cn("品川", "川崎")
  cn("立川", "川崎"); cn("大崎", "横浜"); cn("川崎", "横浜")
  cn("八王子", "横浜")
end


Node = Struct.new(:name, :arr, :dist)
def cn(name1, name2)
  if $graph[name1] == nil then $graph[name1] = Node.new(name1, [], -1) end
  if $graph[name2] == nil then $graph[name2] = Node.new(name2, [], -1) end
  $graph[name1].arr.push($graph[name2])
  $graph[name2].arr.push($graph[name1])
end
```

スタックを使う (深さ優先の) グラフの「たどり」は次の通り。

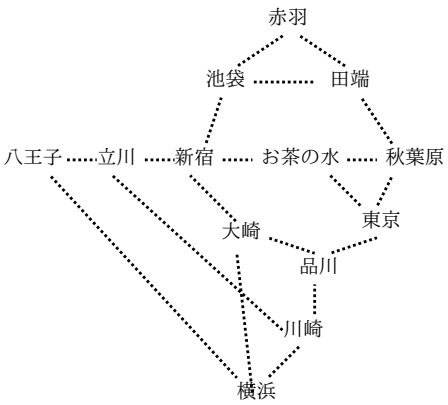

図 1: 鉄道路線図

```
def traverse1(start, goal)
  s = Stack1.new; n = $graph[start]; n.dist = 0; s.push(n)
  puts("START: #{start}")
  while !s.isempty do
    n = s.pop
    n.arr.length.times do |i|
      n1 = n.arr[i]
      if n1.dist < 0
        n1.dist = n.dist + 1
        puts("#{n1.dist}: #{n1.name}")
        if n1.name == goal then return else s.push(n1) end
      end
    end
  end
end
```

これを幅優先にするには、キューを使うように取り替えるだけ (push は enq、pop は deq になる)。これらを動かすメソッドも用意した。

```
def test1
  prepare; traverse1('横浜', '池袋')
  puts('------')
  prepare; traverse2('横浜', '池袋')
end
```

実行例を示そう。

```
irb(main):022:0> test1
START: 横浜
1: 大崎
1: 川崎
1: 八王子
2: 立川
3: 新宿
4: 池袋
------
```

```
START: 横浜
1: 大崎
1: 川崎
1: 八王子
2: 新宿
2: 品川
2: 立川
3: 池袋
=> nil
```

確かに、深さ優先の方が「さっさと」池袋に到達するが、幅優先の方が一番短い経路を発見できる。

1.3 演習 5

これは解答だけ。

a. 「a が 0 個以上あり、b が 1 回あり、その後 a が 0 個以上ある」

b. 「まず a があり、その後 aa と ba が任意個、任意の順序で続いたもの」

c. 「a、abb、abbaa、abbaabb、abbaabbaa、…」言葉で言うなら、「まず a があり、その後 b2 個と a2 個が交互に現れる。a2 個や b2 個の途中以外ならどこで終わってもいい」

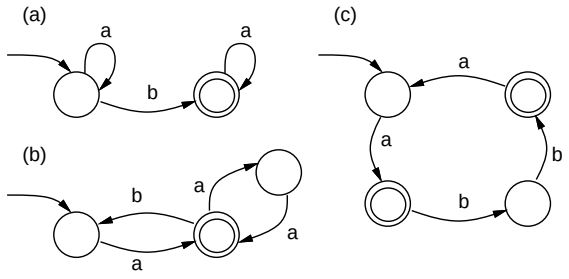

図 2: 演習 5 の有限オートマトン

1.4 演習 6

これは図 3 にオートマトンを示し、対応する$atm の値を以下に示す。

a. 「a が 1 個以上連続し」なので、初期状態からまず a が 1 個来た時の状態に進む必要がある。ここからは次は b でもいいし、a が何個あってもこの状態を続ければいい。b が来たら最終状態で、そこから先は遷移はない。

```
$atm = [{'a' => 1},
        {'a' => 1, 'b' => 2},
        {:final => true}]
```

b. 「a の連続は 2 個まで」なので、初期状態 (a がない状態)、a が 1 個の状態、a が 2 個連続の状態の 3 つを区別する必要がある。どこでも b が来たら初期状態 (a がない状態) に戻る。a が 2 個来た状態ではさらに a があってはいけないので a の遷移がない。すべての状態は最終状態であってよい。

```
$atm = [{'a' => 1, 'b' => 0},
        {'a' => 2, 'b' => 0},
        {'b' => 0, :final => true}]
```

c. これが実は一番簡単で、a が偶数個 (0 も含む) と奇数個の 2 つの状態だけあればいい。もちろん偶数個の状態 (初期状態) が最終状態。

```
$atm = [{'a' => 1, 'b' => 0, :final => true},
        {'a' => 0, 'b' => 1}]
```

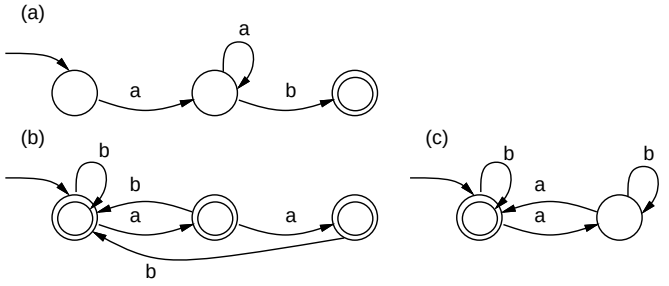

図 3: 演習 6 の有限オートマトン

2 オブジェクト指向と再利用

2.1 クラスライブラリと API ドキュメント

前回も少し出て来たが、オブジェクト指向言語の利点の 1 つとして、クラスという単位はデータと手続きが 1 つにパッケージされて完結しているので、クラス単位で「役に立つ部品」を用意しておきやすいことがあげられる。つまり、そのような部品の集まりを提供しておけば、プログラムを組む側はこれらを持って来て組み合わせ活用することで複雑なことでも簡単にできるわけである。これを (クラスの集まりだから) **クラスライブラリ**と呼ぶ。

一般にソフトウェア開発において、既に誰かが書いたもの (自分が過去に書いたものでもよい) を活用してラクをすることを**再利用**と呼ぶ。クラスライブラリは (そこに含まれるクラスは誰かが過去に書いたものだから) 再利用の有力な 1 形態である。

Java は豊富なクラスライブラリで知られている。Java では、ライブラリに含まれるクラスやそのメソッドの呼び出し方などをまとめたドキュメントを **API ドキュメント**と呼んでいる (API は application programming interface の略)。たとえば、皆様が使っている JDK 1.5 の標準ライブラリ群の API ドキュメントは

```
http://java.sun.com/j2se/1.5.0/ja/docs/ja/api/
```

で見ることができる。ここには、パッケージ単位でそのパッケージに含まれるクラスのドキュメントが集積されている。このすべてを、皆様は Java プログラムから利用できるわけだ。

あるクラスの API ドキュメントは、おおむね次のように構成されている。

- そのクラスがどんなクラスであるかの概要
- フィールドの概要。フィールドとは、インスタンス変数 (Java では、クラス内で定義されている、static のつかない変数) とクラス変数 (クラス内で定義されている、static のついた変数) それぞれについて、その型と概要が説明されている。
- コンストラクタの概要。Ruby では初期化は `initialize` というメソッドによっていたが、Java ではコンストラクタと呼ばれる、オブジェクト生成後に初期化を行うための特別なメソッドがある。これは必ず「`new` クラス名 (引数…)」で呼び出される。Java ではクラス名と同じ名前を持つ返値のないメソッドがコンストラクタとなる。
- メソッドの概要。つまりインスタンスメソッド (Java では、クラス内で定義されている、static のつかないメソッド) とクラスメソッド (クラス内で定義されている、static のついたメソッド) それぞれについて、その引数の個数と型、返値の型と機能の概要が説明されている。

- 上記それぞれの詳細説明。

なお、クラス外部からは変数の参照方法は「クラス名. 変数名」、インスタンス変数の参照方法は「そのオブジェクトを表す式. 変数名」だが、クラス内部からは「変数名」だけで参照できる。メソッドについても同様。

なお、Java ではコンストラクタやメソッドを (引数の個数や型によって区別できる限りにおいて) 複数回定義してもよい。これを多重定義 (**オーバロード**) と呼ぶ。オーバライド (サブクラスにおける親クラスのメソッドの差し替え) と言葉が似ているので注意。このため API ドキュメントでも同名のメソッドが何回も出ていることがある。

演習 1 API ドキュメントから `java.awt` パッケージ中の `Color` クラスのページを開き、色オブジェクトにはどのような生成方法やどのようなメソッドがあるかを調べよ。

2.2 フレームワーク

では次に、たとえば自分で窓を作るプログラムをどうやって書くかを考えてみよう。`javax.swing` パッケージに `JFrame` というクラスがあるが、これが「窓」のクラスである。だから、このクラスのインスタンス (オブジェクト) を生成して適当な大きさにして見えるようにしてやれば窓はできる (注意! このプログラムは何の役にも立たないので打ち込んで動かしても無駄です):

```
import java.awt.*;      ← import が
import javax.swing.*;  ←増えている
public class Sample30 {
  public static void main(String[] args) {
    JFrame frame = new JFrame();   ←窓を作る
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  ←「X」で終了
    frame.setPreferredSize(new Dimension(400, 400));       ←大きさ設定
    frame.pack();                                          ←設定適用
    frame.setVisible(true);                                ←表示する
  }
}
```

しかしこのプログラムでは、窓はできるが中身は真っ白なだけである。では、窓の中に自分が表示したい内容を表示させるのにはどうしたらいいだろうか? それには、`JFrame` が持っているメソッド `paint()`(窓の内容を描くためのメソッド) を取り替えて、自分の表示したいものを表示する動作をさせればよい。しかし、既にできてしまっているクラスのメソッドを書き換えるにはどうしたらいいだろう? 実はそれは簡単で、`JFrame` のサブクラスを定義して、そこで `paint()` を差し替え (オーバライド) すればよい。

このように、オブジェクト指向言語では、ある程度カタチが出来ているもの (**フレームワーク**) を持って来て、その一部を自分の都合に合わせて変更する、という形の再利用も多くつかわれる。フレームワークとクラスライブラリでは、同じ再利用でも自分のプログラムの立つ位置が違うことに注意 (図 4)。なお、実際には 1 つのプログラムで両方のやり方が組み合わされて使われることも多い。

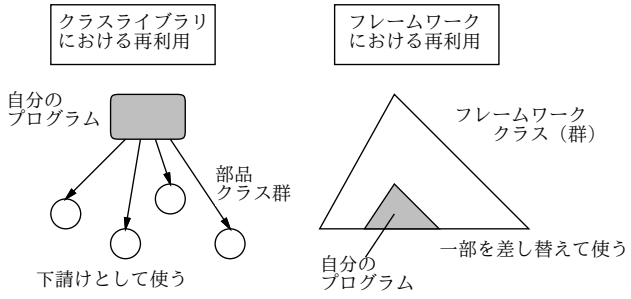

図 4: クラスライブラリとフレームワーク

2.3 窓を作って簡単な絵を描く

ではいよいよ、窓の中に簡単な絵を描くプログラムを作ってみよう。上述のように、今度のプログラムは JFrame のサブクラスとして作り、メソッド paint() をオーバライドして絵を描く処理を書く。あと、大きさ指定などの初期設定をコンストラクタで行う (コンストラクタはクラスと同じ名前なので決してオーバライドできない)。そして、main() は依然として必要なのだが、単にこのクラスのインスタンスを作って見えるようにする 1 行のコードだけにしてしまう。なお、extends は Ruby の「<」に相当し、親クラスからの継承を表す。

```
import java.awt.*;
import javax.swing.*;

public class Sample31 extends JFrame {
  Font fn = new Font("SansSerif", Font.BOLD, 48);
  public Sample31() {
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setPreferredSize(new Dimension(400, 400)); pack();
  }
  public void paint(Graphics g) {
    g.setColor(getBackground()); g.fillRect(0, 0, getWidth(), getHeight());
    g.setColor(Color.red); g.fillOval(200, 150, 100, 100);
    g.setFont(fn); g.setColor(Color.green); g.drawString("Hello", 200, 80);
  }
  public static void main(String[] args) { new Sample31().setVisible(true); }
}
```

main() はクラスメソッド (static のついたメソッド) で、インスタンスと無関係に呼び出せるのに対し、paint() はインスタンスメソッド (static のつかないメソッド) であることに注意。main() は一番最後にあるが、単にこのクラスのインスタンスを作って見えるようにしているだけ。あと、クラス内の 1 行目でインスタンス変数 fn を定義し初期化しているが、これは本体内で使うフォントオブジェクトを入れておくのに好都合だから (画面を描くたびにフォントオブジェクトを生成してたら重そうですね)。コンストラクタ内は前の例にあった初期化を入れているが、今度は「自分のクラスの変数やメソッドの参照」なのでオブジェクトやクラスを指定しなくてよくなっているのに注意。

さて paint() だが、このメソッドはパラメタとして Graphics オブジェクトを 1 つ渡される。これが画面に描くための「ペン」に相当し、そのメソッドをあれこれ呼び出すことでさまざまなものが画面に描ける。ここでは、1 行目でペンを背景色に設定して窓内全体を塗り、次にペンを赤にして楕円 (実際には縦横が同じなので円) を塗、次にペンを緑にして先に用意した 48 ポイントフォントで Hello と描いている (図 5)。

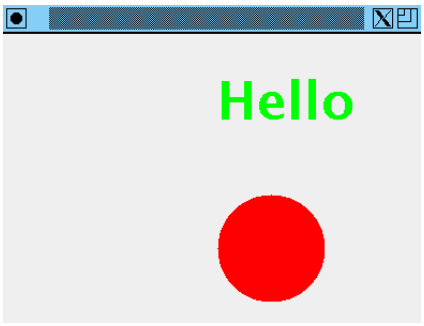

図 5: 窓に Hello と描くプログラム

演習 2 java.awt パッケージの Graphics クラスの API ドキュメントで、setColor()、fillRect()、fillOval()、setFont()、drawString() などのメソッドの呼び出し方と機能を確認せよ。またその他使いたそうなメソッドがあればメモしておくこと。

演習 3 上の例題プログラムをそのまま打ち込んで動かせ。動いたら次のような変更を施してみよ。

a. 文字や円 (楕円) の位置や大きさ等を変更してみる。
b. 先に調べた `Color` の指定方法を用いて色をさまざまに変更してみる。
c. もっと図形の数や種類を増やしてみる。
d. ループを使って同じ図形を繰り返し、色と位置を少しずつ変更しながら描いてみる。
e. その他自分が美しいと思う絵を描いてみる。

3 オブジェクト指向と GUI プログラミング

3.1 GUI 部品

絵が描けるようになったところで、今度は GUI 部品を取り上げよう。オブジェクト指向ではもちろん、GUI 部品の 1 つひとつがオブジェクトになっている。そして、それらは共通の機能 (窓に貼り付けられるとか、大きさが指定できるとか) を持つので、そのような機能を実装している共通の親クラスの子孫になっているのが普通である。

ここから具体例だが、今回扱う部品群はいずれも、`java.awt.Component` のサブクラスになっていて、この親クラスが持つメソッド `setBounds(`X`,` Y`,` 幅`,` 高さ`)` を継承している。このメソッドを使うことで画面上の位置と大きさが指定できるわけだ。次の例題は、画面上にボタンとテキスト欄を 1 つずつ配置するだけのものである。配置した部品は勝手に自分を描画するようになるので、今回は初期化 (コンストラクタ) だけで `paint()` の定義は不要である。

```
import java.awt.*;
import javax.swing.*;

public class Sample32 extends JFrame {
  JButton b0 = new JButton("Push Me");
  JTextField f0 = new JTextField();
  public Sample32() {
    Container c = getContentPane(); c.setLayout(null);
    c.add(b0); b0.setBounds(20, 80, 90, 30);
    c.add(f0); f0.setBounds(20, 120, 90, 30);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setPreferredSize(new Dimension(400, 400)); pack();
  }
  public static void main(String[] args) { new Sample32().setVisible(true); }
}
```

`getContentPane()` は部品を貼り付ける領域を取り出すのに使う。続いて自動レイアウト機能をオフにしているが、こうしないと部品の配置や大きさを勝手に変更されてしまうので。その後、各部品を貼り付け、位置と大きさを設定している。その他の初期設定等は前と同じ。これを動かした様子を図 6 に示す。もちろん、部品を配置しただけなのでボタンを押したり入力欄に打ち込んだりはできるが、それ以上の動作は何もない。

演習 4 上の例題を打ち込んでそのまま動かせ。動いたら、ボタンや入力欄の数を増やしてみよ。さらに、`javax.swing` パッケージにある次のような GUI 部品も、API ドキュメントで使い方を確認して、窓に入れてみよ。`JLabel`、`JSlider`、`JCheckBox`、`JToggleButton`、`JComboBox`、`JTextArea`、`JRadioButton`

演習 5 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. 数を入力すると素数かどうか教えてくれる。
b. 華氏の温度を摂氏の温度に変換する。

図 6: ボタンと入力欄

c. 3 つの数値 (0〜255) を入力してボタンを押すとそれを RGB 値とする色が窓全体の背景色になる。
d. 簡単な電卓 (機能は適当に設計してよい)。
e. 入力も表示も 2 進数で行う簡単な電卓 (機能は適当に設計してよい)。
f. 数値を入力するとその数値までの素数一覧がスクロールリストに現れる。
g. 円や星型その他の図形のパラメタを入力してボタンを押すとそのパラメタに従った図形が現れる。
h. その他自分の好きなもの。

3.2 コンポジションとインタフェース

先の例題ではボタンを押しても何も起きなかったが、ここにちゃんと動作をつけてみよう。それにはどうしたらいいだろうか。上で学んだフレームワークの考えによるなら、たとえばボタンに `action()` というメソッドがあり、動作をつけたいときはボタンのサブクラスを作ってこのメソッドをオーバライドする… ということになるだろう。しかしその方法はよくない。というのは、ボタン以外にメニューやキーボードショートカット等さまざまなものからその「動作」を呼び出したいかも知れないのに、ボタンのサブクラスにしてしまうとそれはボタンとしてしか使えないから。

このため、Java では次のような方法で動作を指定している。

- ボタンに `addActionListener()` というメソッドがあり、これを使って「動作オブジェクト」を設定する。
- ボタンは押されるとこの動作オブジェクトの「`actionPerformed()`」というメソッドを呼び出す。
- 動作オブジェクトはメソッド `actionPerformed()` の中に押された時の動作を記述。

これならば、メニューやキーボードショートカットにも同じ動作オブジェクトを設定すれば同じ動作がさまざまな方法で呼び出せるようになる。このように、継承を使ってオブジェクトを組み合わせる代わりに、オブジェクトに他のオブジェクトを参照 (結合) させる方法で組み合わせることを**コンポジション**と呼ぶ。

Ruby ならこれだけでいいのだが、Java の場合は強い型なので、この「動作オブジェクト」をどういう型にするかという問題がある。様々なボタンにつける動作オブジェクトは構造や機能はバラバラであり、共通なのは「actionPerformed()」を持つ、というインタフェースだけである。このようなものを指定するために、Java ではその名もインタフェースという機能がある。上の「`actionPerformed()` を持つ」ということは標準ライブラリ API 中で次のようなインタフェースとして定義されている。

```
interface ActionListener {
  public void actionPerformed(ActionEvent e);
}
```

そして、あるクラスがインタフェースを実装する (そこに指定したメソッドを持つ) ことは、クラス定義の冒頭に「`implements` インタフェース名」と指定して宣言する。これをまとめると、上の記述は次のように書き直せる。

- ボタンに `addActionListener()` というメソッドがあり、これを使って `ActionLisnter` オブジェクトを設定する。

- ボタンは押されるとActionListenerオブジェクトの「actionPerformed()」というメソッドを呼び出す。
- 「ActionListnerオブジェクトはメソッドactionPerformed()の中に押された時の動作を記述。

では実際にこれを使って、先のGUIに動作をつけてみよう。ActionListenerオブジェクトを定義するクラスMyAdapterは、クラスの内側で定義する**内部クラス**にしている。内部クラスは、外側クラスのインスタンス変数を自由に参照/変更できるのでこのような場合に便利である。

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

public class Sample33 extends JFrame {
  JButton b0 = new JButton("Push Me");
  JTextField f0 = new JTextField("0");
  public Sample33() {
    Container c = getContentPane(); c.setLayout(null);
    c.add(b0); b0.setBounds(20, 80, 90, 30);
    c.add(f0); f0.setBounds(20, 120, 90, 30);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setPreferredSize(new Dimension(400, 300)); pack();
    b0.addActionListener(new MyAdapter());
  }
  public static void main(String[] args) { new Sample33().setVisible(true); }
  class MyAdapter implements ActionListener {
    public void actionPerformed(ActionEvent evt) {
      int i = Integer.parseInt(f0.getText());
      f0.setText("" + (i+1));
    }
  }
}
```

見ての通り、本体で違うのはボタンにMyAdapterオブジェクトを設定している点だけ。そしてクラスMyAdapterはActionListnerインタフェースを実装するクラスであり、そのメソッドActionPerformed()の中ではテキスト欄の内容を取り出し、それを整数に変換し、1増やして文字列に戻してテキスト欄の内容として設定し直している。

3.3 無名内部クラス

上のような方法でGUI部品に動作をつけることができるようになったが、動作1つ毎に新しくクラスを作るのが面倒である。そこでJavaでは、(1)1つのクラスをextendsしているか、または1つのインタフェースだけをimplementsしていて、(2)1箇所でしか参照されないような内部クラスを定義する時に

```
... new 内部クラス名 () ...

class 内部クラス名 extends/implements XXX {
  (内部クラス定義本体)
}
```

を次のように「参照するその場に埋め込んで」書いてもよいようにしている。

```
... new XXX() {
  (内部クラス定義本体)
} ...
```

つまりこの場合、内部クラスには名前をつけなくてもよくなる (XXX は親クラスまたはインタフェースの名前)。これを**無名内部クラス**と呼ぶ。先の例題を無名内部クラスで書き換えたものを示す。

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

public class Sample34 extends JFrame {
  JButton b0 = new JButton("Push Me");
  JTextField f0 = new JTextField("0");
  public Sample34() {
    Container c = getContentPane(); c.setLayout(null);
    c.add(b0); b0.setBounds(20, 80, 90, 30);
    c.add(f0); f0.setBounds(20, 120, 90, 30);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setPreferredSize(new Dimension(400, 300)); pack();
    b0.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        int i = Integer.parseInt(f0.getText());
        f0.setText("" + (i+1));
      }
    });
  }
  public static void main(String[] args) { new Sample34().setVisible(true); }
}
```

書き方は気持ち悪いが、ちょっとだけ、短くなっていますね。

演習 6 で作成したインタフェースだけのプログラムに動作をつけて使えるようにしてみよ (無名内部クラスを使っても使わなくてもよい)。

3.4 GUI 部品とその配置

前節までの方法で GUI 部品を配置したプログラムは、窓の大きさを変えても部品の配置が変わらないのでちょっと情けないところがあった。そのため Java では、部品の配置を調整する機能である**レイアウトマネージャ**とばれるクラスが複数用意されている。ここでは簡単なものとして次の 2 つを使って見る。

- `BorderLayout` — 領域や窓の「中央」「上端」「下端」「右端」「左端」に部品を置くことができる。
- `FlowLayout` — 領域に左から順に部品を詰めて行ける。

`JFrame` や `JPanel`(ただの領域) の内側の領域には特にに指定しない場合は `BorderLayout` が設定されているが、変更したければ `setLayout()` で使いたいレイアウトマネージャのインスタンスを設定する。ここでは細かい説明は大変なので略すが、とにかく次のように配置する (図 7)。

- 外側の窓の中央には絵を描く場所として `JPanel` のサブクラスを配置する (好きなように絵を描くために `paint()` をオーバライド
- 外側の窓の下端に GUI 部品用 `JPanel` を配置してその中の配置には `FlowLayout` を使う。

これを使ったプログラムを次に示す (図 8)。

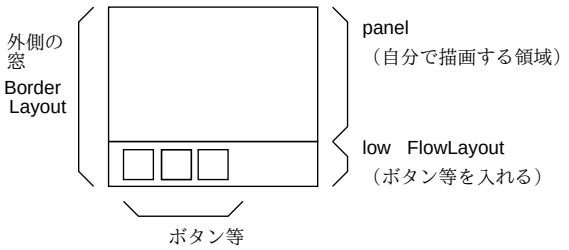

図 7: レイアウトマネージャ

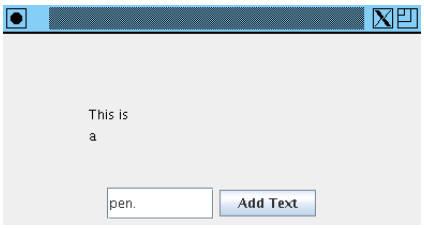

図 8: 入力テキストを描画する例題

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

public class Sample35 extends JFrame {
  String[] lines = new String[100];
  int count = 0;
  JTextField f0 = new JTextField();
  JButton b0 = new JButton("Add Text");
  JPanel panel = new JPanel() {
    public void paint(Graphics g) {
      g.setColor(Color.black);
      for(int i = 0; i < count; ++i) { g.drawString(lines[i], 80, 80+i*20); }
    }
  };
  JPanel low = new JPanel();
  public Sample35() {
    Container c = getContentPane();
    c.add(panel); c.add(low, BorderLayout.SOUTH);
    low.setLayout(new FlowLayout()); low.add(f0); low.add(b0);
    f0.setPreferredSize(new Dimension(100, 30));
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setPreferredSize(new Dimension(400, 300)); pack();
    b0.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        lines[count] = f0.getText(); f0.setText(""); ++count; repaint();
      }
    });
  }
```

```
  public static void main(String[] args) { new Sample35().setVisible(true); }
}
```

すなわち low はただの JPanel(空っぽの領域) だが、panel はそれを無名内部クラスで継承したもので、paint() をオーバライドして配列 lines の内容を描画している。窓全体の中には panel と low を追加するが、後者は「南」に置く (何も指定しないと中央になる)。low の中にはボタンを 2 つ順に追加するが、FlowLayout を設定してあるのでボタンは追加した順に並ぶ。ボタンの動作の中では入力テキストを lines に保存し、個数を増やし、最後に repaint() を呼ぶ。repaint() は「窓の内容が変化したので描画し直してください」という通知を行うもので、これによって窓全体が再描画され、その一環として先に定義した paint() も呼び出される。

演習 7 何でも好きな GUI プログラムを作りなさい。

A　本日の課題 13A

プログラム作成を内容とする演習問題から 1 つ選び、動かしたプログラムを含む小レポートを今日中に久野までメールで送ってください。

1. Subject: は「Report 13A」とする。
2. 学籍番号、氏名、投稿日時を書く。
3. 動かしたプログラムどれか 1 つのソース。
4. 以下のアンケートの回答。

   Q1. 窓を作るプログラムで絵が描けそうですか?

   Q2. GUI 部品を使ったプログラムが作れそうですか?

   Q3. その他、感想、要望等どうぞ。

次回までの課題はありません。