計算機プログラミング'99　＃5

久野　靖*

1999.10.6

0　はじめに

前回はスレッドの話と継承の話を両方やったため、相当ハードでした。それで今回は少し反省して、前半ではなるべく短いプログラムを使って継承の「練習問題」をやってみようと思います。その後で後半にインターフェースの話をして、それと関係のありそうな話題をいくつか拾い集めてみます。本当はイベントや GUI の話が関係深いのですが、それをいちどにやるとまた消化不良になりそうなので次回に回します。

1　例題+演習: 関数のグラフ

では早速、「関数のグラフを描くアプレット」という例題を見ていただこう。

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

public class R5Sample1 extends Applet {
  Function fn = new LinearFunc(1.5, -0.3); // ***
  public void paint(Graphics g) {
    g.setColor(Color.black);
    g.drawLine(0, 100, 200, 100); g.drawLine(100, 0, 100, 200);
    g.setColor(Color.blue);
    drawFunc(g, fn);
  }
  public void drawFunc(Graphics g, Function f) {
    double x0 = -1.0;
    double y0 = f.calculate(x0);
    for(int i = 1; i <= 100; ++i) {
      double x = 0.02*i - 1.0;
      double y = f.calculate(x);
      g.drawLine(100+(int)(100*x0), 100-(int)(100*y0),
                 100+(int)(100*x), 100-(int)(100*y));
      x0 = x; y0 = y;
    }
  }
}
```

*筑波大学大学院経営システム科学専攻

```
class Function {
  public double calculate(double x) { return 0.0; }
}

class LinearFunc extends Function {
  double a, b;
  public LinearFunc(double a0, double b0) { a = a0; b = b0; }
  public double calculate(double x) { return a*x + b; }
}
```

すなわち、「関数」をオブジェクトとして扱うことにし、クラス Function がすべての「関数」の元締めとなる親クラスとする。その子クラスとして 1 次関数に対応する LinearFunc というクラスを作り、このアプレットでは $f(x) = 1.5x - 0.3$ の関数を描いている (描く範囲は x が -1.0 から 1.0 まで)。

では、これをもとに次の演習をしてみて頂こう。

演習 1 このアプレットのソースコードを打ち込んでそのまま動かせ。

演習 2 適宜クラスを追加して、次のような関数のグラフを描くようにしてみよ。アプレットクラスでは「`***`」の行の「`new`」より右側だけ直せば済むようにすること。追加するクラスをどのクラスのサブクラスにするか考慮すること。「美しさを損なわない範囲でできるかぎり」コードの再利用を行うことが望ましい。

a. $f(x) = 2x^2 + 1.5x - 0.3$
b. $f(x) = 2x^3 + 1.5x - 0.3$
c. $f(x) = \frac{0.3}{(x-1.01)} + 1$
d. $f(x) = 0.7sin(4x)$
e. $f(x) = 0.7sin(\frac{0.3}{(x-1.01)} + 1)$

2 インタフェース

さて、先の例題のクラス Function は明らかに、「`public double calculate(double x)`」というメソッドを持つということを共通に規定することだけを目的としていた。つまり、このクラスの子孫はすべて同じ呼び方で `calculate()` が使えるわけだが、実際にどのようなインスタンス変数を持ち、どのようなコードで計算を実現するかは個々のクラスごとにバラバラであった (もちろん、ある子クラスとその下にある孫クラスとでは再利用が行われている場合があるだろうけど)。

このように、クラスが持つインタフェースのみを規定し、インスタンス変数やメソッドなどの実装は規定しない、というやり方は、従来の継承 (実装とインタフェースが一緒に引き継がれる) に比べてよりよい方法である、という考えが主流になりつつある。そして、そのための言語機能を持つような言語が作られ始めている。実は Java はそのような機能を持ち実用的に使われている言語では最初のものである。

先の例だいをインタフェース機能を使うように直したものを次に示す。

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

public class R5Sample2 extends Applet {
  Function fn = new LinearFunc(-0.3, 1.5); // ***
  public void paint(Graphics g) {
    g.setColor(Color.black);
    g.drawLine(0, 100, 200, 100); g.drawLine(100, 0, 100, 200);
```

```
    g.setColor(Color.blue);
    drawFunc(g, fn);
  }
  public void drawFunc(Graphics g, Function f) {
    double x0 = -1.0;
    double y0 = f.calculate(x0);
    for(int i = 1; i <= 100; ++i) {
      double x = 0.02*i - 1.0;
      double y = f.calculate(x);
      g.drawLine(100+(int)(100*x0), 100-(int)(100*y0),
                 100+(int)(100*x), 100-(int)(100*y));
      x0 = x; y0 = y;
    }
  }
}

interface Function {
  public double calculate(double x);
}

class LinearFunc implements Function {
  double a, b;
  public LinearFunc(double a0, double b0) { a = a0; b = b0; }
  public double calculate(double x) { return b*x + a; }
}
```

アプレットクラスはまったく同じである。一方、Function はここではクラスからインタフェースに変更されている。クラスとインタフェースの具体的な違いは次の通り:

- 「class」の代りに「interface」と書く。
- インスタンス変数は定義できない。[1]
- メソッドは引数部分までで、コード本体は書かずに「;」でおしまいにする。

また、インタフェースを「利用する」側ではクラスの継承と比べて次のような違いがある。

- 「extends」の代りに「implements」と書く。
- 「extends」では 1 個だけしか親クラスを指定できないが、「implements」ではいくつでも親インタフェースを指定できる。さらに「extends」と併用してもよい。
- implements で指定したインタフェースで定義されているメソッドは必ずコードを実装しなければならない (よその親クラスから継承してきてもよいが)。

インタフェースの場合は継承と異なり、実現 (インスタンス変数、メソッドのコード) を引き継がないので、内部的にまったく異なるデータ構造を持ったものであっても同じインタフェースという形でくくることができる。また、多重継承のような干渉の問題もないため、1 つのクラスが多数のインタフェースを implements していてもよい。

[1] 例外として、「定数」の役割をする「書き換えられない」変数 — 「public final int 変数名 = 初期値;」のようなもの — は定義してもよい。

3 Enumeration インタフェース

インタフェースが具体的にどのような形で使われるかの例として、Enumeration インタフェースを取り上げよう。Enumeration インタフェースは java.util パッケージに含まれているもので、次の 2 つのメソッドだけを定義している。

- `public boolean hasMoreElements()` — 「まだ要素があるかどうか」を返す
- `public Object nextElement()` — まだ要素があるとき、「次の要素」を取り出して返す。

これは何に使うかというと、プログラムで頻繁に行われる「何かを次々に処理する」という場合にこのインタフェースを implements したオブジェクトを利用することを意図している。つまり、次のような使い方になる。

```
for(Enumeration e = ...; e.hasMoreElements(); ) {
  Object o = e.nextElement();
  「o」の値を使用する処理...
}
```

これの何が嬉しいか? それは、たとえば従来のやり方だと「あるデータ構造の全要素を順に処理する」場合、そのデータ構造が何であるかによって全部ループのし方が変わって来るのが当然だった。

```
for(int i = 0; i < limit; ++i) {
  elem = a[i];  // 配列の要素を順に処理
  ...
}

for(link* p = top; p != NULL; p = p->next) {
  elem = p->info; // 連結リストの要素を順に処理
  ...
}
```

しかし、ということは、単に「要素を順に処理したい」だけのコードなのに、データ構造の細部を直接操作してしまっている。あとでデータ構造を変更すると面倒なことに…これに対し、Enumeration を使うようにしておけば、データ構造を変更しても (それらのクラスが Enumeration を提供するようにさえなっていれば) ほとんど変更なしで済む。

4 Vector クラス

Enumeration オブジェクトを提供してくれるような標準クラスの例として、Vector クラスを見てみよう。Vector は簡単に言えば「自動的に伸び縮みしてくれる配列」である。Vector を使って、「空行が来るまで行を読み込み、空行が来たらそれまでに読み込んだ行を表示する」プログラムを見てみよう。

```
import java.io.*;
import java.util.*;

public class R5Sample3 {
  public static void main(String args[]) {
    try {
      BufferedReader in =
        new BufferedReader(new InputStreamReader(System.in));
      Vector vec = new Vector();
      while(true) {
```

```
        System.out.print("String> "); System.out.flush();
        String str = in.readLine();
        if(str.equals("")) break;
        vec.add(str);
      }
      for(Enumeration e = vec.elements(); e.hasMoreElements(); ) {
        Object obj = (String)e.nextElement();
        System.out.println(obj);
      }
    } catch(Exception e) { System.err.println("!!"+e); }
  }
}
```

つまり、Vector オブジェクトにはいくつでも値が `add()` でき、メソッド `elements()` を呼ぶと現在入っている値を順に返すような Enumeration オブジェクトを返してくれる。なお、Vector に格納できるものは Object 型 (ということは任意のクラスのインスタンス) であることに注意。また、これと対応して、Enumeraiton オブジェクトのメソッド `nextElement()` は Object 型を返すことにも注意。

演習 3 上のプログラムをそのまま打ち込んで動かせ。

5 自分で Enumeration オブジェクトを作ってみる

では、使うだけだとつまらないのでもっと別な Enumeration オブジェクトを自前で作ってみよう。次のは、「i から j まで s ずつ増えていく」ような Integer オブジェクトを次々に返す Enumeration オブジェクトのクラスとそれを動かしてみるメインクラス。

```
import java.io.*;
import java.util.*;

public class R5Sample4 {
  public static void main(String args[]) {
    try {
      for(Enumeration e = new IntEnum(1, 10, 1); e.hasMoreElements(); ) {
        Object obj = e.nextElement();
        System.out.println(obj);
      }
    } catch(Exception e) { System.err.println("!!"+e); }
  }
}

class IntEnum implements Enumeration {
  int val, to, by;
  public IntEnum(int f, int t, int b) { val = f; to = t; by = b; }
  public boolean hasMoreElements() { return val <= to; }
  public Object nextElement() { return new Integer(val++); }
}
```

どうです、なかなか面白いでしょう? なお、Enumeration が返す値は Object でないといけないので Integer クラスを使っていることに注意。

演習 4 上のプログラムを打ち込んでそのまま動かせ。またコンストラクタの引数を変えてちゃんと動くことを確認せよ。

演習 5 これを参考に、次のような Enumearation オブジェクト (Enumeration インタフェースを実装したクラス) を作って上のプログラムの IntEnum と差し替えて動かせ。

a. x から始めて y より小さい範囲で値が倍々に増えていく。
b. 1 から始めて y より小さいフィボナッチ数を次々に返す。
c. 文字列 (String オブジェクト)s を渡すとその文字列の文字を 1 つずつ返す。

6 Enumeration 用の内部クラスの利用

さて、実際には Enumeration オブジェクトは Vector など「要素を複数集めて保持する」クラスが別途あって、その各要素を順次取り出すのに使うのが普通である。そのような例として、たとえば自前で Vector のようなものを作ってみよう (ただし簡単のため最大容量は固定し、あふれたものは捨てられる)。

```
import java.io.*;
import java.util.*;

public class R5Sample5 {
  public static void main(String args[]) {
    try {
      BufferedReader in =
        new BufferedReader(new InputStreamReader(System.in));
      MyVector vec = new MyVector(10);
      while(true) {
        System.out.print("String> "); System.out.flush();
        String str = in.readLine();
        if(str.equals("")) break;
        vec.add(str);
      }
      for(Enumeration e = vec.elements(); e.hasMoreElements(); ) {
        Object obj = e.nextElement();
        System.out.println(obj);
      }
    } catch(Exception e) { System.err.println("!!"+e); }
  }
}

class MyVector {
  Object[] a;
  int count = 0;
  public MyVector(int size) { a = new Object[size]; }
  public void add(Object o) { if(count < a.length) a[count++] = o; }
  public Enumeration elements() { return new MyEnumeration(); }
  class MyEnumeration implements Enumeration {
    int i = 0;
    public boolean hasMoreElements() { return i < count; }
```

```
    public Object nextElement() { return a[i++]; }
  }
}
```

このように、Enumeration オブジェクトを内部クラスによって定義すると、その中では外側のクラスのインスタンス変数も参照できるため作業がやりやすい。

演習 6 上のコードを打ち込んでそのまま動かせ。

演習 7 上で出て来た elements() は「入れた順に」要素を返す Enumeration オブジェクトを作り出していた。その反対に、「入れたのとは逆順に」返すような Enumeration オブジェクトを作り出すようなメソッドも提供してみよ。このようにすることで、要素を「さまざまな方法で」たどれるようになる。

演習 8 上の例では配列を用いていたが、連結リストを用いたバージョンや木 (2 分木、多分木、B 木) を用いたバージョンも作ってみよ。Enumeration が要素を返す順番は好きに決めてよい。

演習 9 Enumeration オブジェクトを使うことの利点として、複数の Vector(など) を「並行して」辿るようなことがやりやすいことも挙げられる。これを利用して、まず単語を好きなだけ読ませるが、大文字で始まる単語と小文字で始まる単語を対にして打ち出すというプログラムを作ってみよ (数が同じでないときどうするかはお任せする)。たとえば次のような感じ。

```
% java XXXXX
String> This
String> is
String> a
String> Book.
String>
This, is
Book, a
%
```

ヒント: Vector を 2 つ用意して、大文字か小文字かでどっちかに入れてやる。取り出すときは…考えてみよう。