計算機プログラミングI 2005　久野クラス　＃12

久野　靖*

2006.1.27

はじめに

半年間に渡っておつき合い頂いた「計算機プログラミングI」も今回で最終回となりました。今回は「計算機プログラミングI」公式掲示に書かれていてまだやっていない話題である、計算量とデータ量の議論が納得できるような内容ということで、Iterator インタフェースと ArrayList と呼ばれるクラスを取り上げます。いずれも、標準 API の java.util パッケージに入っているものです。また、それに引続き、この ArrayList のようなクラスの実現方法について考えるのを兼ねて、計算量の話題も取り上げます。

あと、情報図形教室から「最終回に学生アンケートを実施するように」というお達しが来ていまして、アンケート用紙とマークカードを配付しますので、これは時間中に鉛筆でマークして提出してください。このクラスでは皆様のご意見はさんざん伺っていますが、統一して調査するということですので。今回の当日レポートはそれとはまったく関係なく、いつも通り「今日中」でお願いします。

1　前回の演習問題の回答例

いちおう、前回のメイン演習 3 の (b)、(c)、(d) を合わせた回答例を示しておこう。MyTableModel の手前までは当然ながらまったく一緒。

```
import javax.swing.*;
import javax.swing.table.*;

public class r11ex3 extends JFrame {
  public r11ex3() {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    getContentPane().add(new JScrollPane(new JTable(new MyTableModel())));
    pack(); setSize(400, 150);
  }
  public static void main(String[] args) {
    new r11ex3().setVisible(true);
  }
```

そして MyTableModel も冒頭部分は一緒。なお、isCellEditable はすべてのセルが変更可能としたので常に true を返す。

```
  class MyTableModel extends AbstractTableModel {
    double[] rows = new double[]{1, 3, 5, 7, 11, 13, 17, 19, 23};
    double[] cols = new double[]{1, 2, 3, 4, 5, 6, 7, 8};
    public int getRowCount() { return rows.length; }
    public int getColumnCount() { return cols.length; }
    public boolean isCellEditable(int r, int c) { return true; }
```

*筑波大学大学院経営システム科学専攻

```
public Object getValueAt(int r, int c) {
  return new Double(rows[r] * cols[c]);
}
```

そして、setValueAt は、一番上の行の場合、一番右側の列の場合、そのなかで一番下のセルの場合、それ以外 (中央のセル) の 4 通りで処理が異るので枝分かれ。条件の順序は書きやすいように入れ換えている。

```
public void setValueAt(Object o, int r, int c) {
  if(r == 0) {
    cols[c] = new Double(o.toString()).doubleValue();
  } else if(c == 0 && r == rows.length-1) {
    int n = new Integer(o.toString()).intValue();
    if(n > rows[rows.length-1]) rows = primes(n);
  } else if(c == 0) {
    rows[r] = new Double(o.toString()).doubleValue();
  } else {
    cols[c] = new Double(o.toString()).doubleValue() / rows[r];
  }
  fireTableDataChanged();
}
```

枝分かれの中は次の通り。

- 一番上の行 — cols[c] に入力数値を入れる。
- 一番右の列 — rows[r] に入力数値を入れる。
- ただし一番下のセル — 入力数値が現在の rows に入っている最大より大きいなら、rows を作りなおす (これはまとまった仕事なので下請けメソッドを読んでいる)。
- 中央のセルなら、その値を x として、cols[c] = x / rows[r] とする (こうすれば逆に x = cols[c]*rows[r] になるから)。

あとは素数の配列を入れた double の配列を返す下請けメソッド。

```
  private double[] primes(int n) {
    boolean[] a = new boolean[n+1]; // initally all false
    int prime = 2, count = 0;
    while(prime <= Math.sqrt(n)) {
      for(int i = prime*2; i <= n; i = i + prime) a[i] = true;
      ++prime;
      while(a[prime] && prime <= n) ++prime;
    }
    for(int i = 1; i <= n; ++i) if(!a[i]) ++count;
    double[] b = new double[count]; count = 0;
    for(int i = 1; i <= n; ++i) if(!a[i]) b[count++] = i;
    return b;
  }
 }
}
```

素数を増やすときは一瞬のうちに列が増えるのでなかなか楽しい。

2 Iterator インタフェース

これまでにも Runnable など標準 API 中のインタフェースをいくつか見てきたが、今回はインタフェースの特徴を活かしたものとして Iterator インタフェースを取り上げよう。Iterator 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 3 つのメソッドだけを定義している。

- `public boolean hasNext()` — 「まだ要素があるかどうか」を返す
- `public Object next()` — まだ要素があるとき、「次の要素」を取り出して返す。
- `public void remove()` — 「その要素を削除する」時に使う (少し後で説明)。

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(Iterator i = ...; i.hasNext(); ) {
  Object o = i.next();
  「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
  ...
}
```

以下は Java ではなく C か C++の書き方だけど:

```
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、`Iterator` 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 `Iterator` 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。

なお、上では「データ構造の要素の処理」を挙げたが、`Iterator` はそれに限らず「順番に出して来る」ものなら何にでも使える。たとえば、「数値を順番に出して来る」例を示そう。

```
import java.util.*;

public class R12Sample1 {
  public static void main(String[] args) {
    for(Iterator i = new MyIter1(1, 10, 2); i.hasNext(); ) {
      System.out.println(i.next());
    }
  }
  static class MyIter1 implements Iterator {
    int val, limit, step;
    public MyIter1(int v, int l, int s) { val = v; limit = l; step = s; }
    public boolean hasNext() { return val < limit; }
    public Object next() { int v = val; val += step; return new Integer(v); }
    public void remove() { throw new UnsupportedOperationException(); }
  }
}
```

なお、この Iterator は `remove()` は意味がないので実装しないこととし、もし呼ばれた場合は `UnsupportedOperationException` を投げることとする (例外とその投げ方を忘れた人は過去資料を復習しておいてください)。ではこれを実行してみる。

```
% javac R12Sample1.java
% java R12Sample1
1
3
5
7
9
%
```

つまり「new MyIter1(x, y, z) は x から始まって y を越えない範囲で z ずつ増えていく整数の並びを返すような Iterator オブジェクト (Iterator インタフェースを実装するクラスのインスタンス)」なわけである (next() が返すものは Object でなければいけないので、Integer オブジェクトを作って返している点にも注意)。

「これだったらこんなことをしないで直接 for 文で計算した方が楽だ」と思いましたか? これに限定すればそうだが、たとえば次のような場合はどうだろう?

- 上記のような数列から最初 30 個取り出して使い、しばらく後で 20 個取り出して使い…というふうにバラバラに使う場合。
- 上記のような数列や別の計算方法 (下記演習参照) を複数「組にして」使う場合。
- 数列のパラメタや種類を選ぶ場所とその数列を使う場所が離れているような場合。

などは Iterator オブジェクトを使う方が楽である。それはつまり、「使いかけの数列」を「もの」として取り扱えるからであり、これこそ「オブジェクト指向」の良さなわけである。

演習 1 上の例題を打ち込んでそのまま動かせ。動いたら Iterator クラスを改造して、次のような並びを出力させてみよ。

a. 1 つ数値を指定し、その数を越えない範囲で 1、2、4、…のように倍々になっていく値を返す。
b. 1 つ数値を指定し、その数を越えない範囲で 1、1、2、3、5、8、…のようにフィボナッチ数列を返す。[1]
c. 文字列を 1 つ指定し、その文字列の各文字を 1 つずつ返す。(注意: 文字を返す時は「new Character(文字)」のようにして Character クラスのインスタンスにする必要があるだろう。)
d. 文字列を 1 つ指定し、その文字列を循環させた文字列 ("abcd"であれば"bcda"、"cdab"、"dabc"、"abcd" のように) を返す。
e. 文字列を 1 つ指定し、その文字列のあらゆる並べ替え (順列) を返す。(注意: これはかなり難しいのでそのつもりで。)

3 ArrayList クラス

これまで、数値やオブジェクトを複数まとめて格納するのには配列を使って来たが、配列は「最初に要素の最大数を指定しなければならない」「途中に要素を挿入したり、途中の要素を削除したりできない」という制約があった。実は、java.util パッケージに含まれている ArrayList クラスは配列によく似ているがそのような制約がない。つまり、最初に new ArrayList() で作って、add(o) でいくつでも要素を増やして行くことができる。これはオブジェクトなので i 番目の要素を取り出したり変更するのにはメソッドを呼び出す。また、add(i, o) や remove(i) を使って任意の位置で要素を追加したり削除できる (2 つの add() のどちらであるかは、引数の数や型を見れば区別できる。このように、同じ名前のメソッドを複数回定義することを**オーバロード**とよぶ。オーバロードするときには当然、「呼び方でどの場合か区別できる」ことが必要)。そしてもちろん、要素を順番に取り出すには Iterator オブジェクトを使う。簡単な例を見てみよう。[2]

```
import java.util.*;

public class R12Sample2 {
```

[1] フィボナッチ数列とは、$x_0 = x_1 = 1$、$x_n = x_{n-1} + x_{n-2}$(ただし $n > 1$ のとき) で定義されるような数列。
[2] ここでは変数 l は List となっているが、これはインタフェースであり、add() 等のメソッドはこちらで定義している。List を実装するクラスは ArrayList 以外にもあってよい (実際標準 API では LinkedList というのがそうである)。

```
    public static void main(String[] args) {
      List l = new ArrayList();
      l.add("a"); l.add("b"); l.add("c"); l.add("d");
      l.add(2, "X"); l.add(3, "Y"); l.remove(4);
      for(Iterator i = l.iterator(); i.hasNext(); ) {
        System.out.println(i.next());
      }
    }
  }
```

これを動かした様子を見てみよう。

```
% javac R12Sample2.java
% java R12Sample2
a
b
X
Y
d
%
```

確かに、2 つ要素が挿入され、1 つ削除されている。

4 ダウンキャスト

ところで、ArrayList に入れられるのは Object であり、Object はすべてのオブジェクトの親クラスなので任意のオブジェクトが格納できる (「値」を入れたいときはその値の包囲クラスのインスタンスにして入れる)。たとえば「1」を入れたければ Integer オブジェクトを使う。String などはもともとオブジェクトだからそのままで入れることができる。

ところが、それを取り出して来た時には Object として取り出されて来るので、それを元の Integer や String に戻さないといけないだろう。その「戻す」というのはつまり「キャストする」ということ。具体的には次のようにする。

```
List l = new ArrayList();
...
l.set(0, new Integer(100));  // 0 番目の位置に格納
l.set(1, "ABCD");            // 1 番目の位置に格納
...
Integer i = (Integer)(l.get(0)); // Integer にキャスト
String s = (String)(l.get(1));   // String にキャスト
```

つまり、親クラスにあたる Object に変換するときはキャストが要らないが、子クラスにあたる Integer 等に戻すときはキャストが必要。これを「ダウンキャスト」と呼ぶ。ちょうど、int を double にするときはキャストが要らないが double を int にするときはキャストが要るのに似ている。しかし、違うところもある。

つまり、double を int にするのは (値を切り捨てるだけだから) 常に成功するが、Integer や String へのダウンキャストは「そこに入れたものが Integer や String だから」成功するのであって、全然別のオブジェクトにはキャストできない。そのような間違ったキャストを行なおうとすると、ClassCastException という例外が発生してしまう。逆に言えば、ダウンキャストは常にそのようなチェックを (Java 処理系の内部で) 伴うため、配列のアクセスに比べて効率が悪くなる。この辺は Java の弱点と言える。

ところで、さっきの例題はなぜダウンキャストが不要だったのだろう? それは、System.out.println() では Object が渡された時は Object クラスに定義されている toString() を呼んで文字列に変換してくれるから。
toString() はオブジェクトが実は Integer だったり String だったりした場合でももちろんちゃんと動作してくれる。

5 コンテナクラスを作る

しかし、ArrayList はいったいどうやって「自動的に延びる配列」を実現しているのだろう? それを理解するために、自前で ArrayList のようなものを作ってみよう。次の例題のプログラムは ArrayList を MyList に取り換えただけである (ちなみに「コンテナクラス」というのは ArrayList のように「オブジェクトを格納するためのオブジェクト」を定義するクラスのことである)。

```
import java.util.*;

public class R12Sample3 {
  public static void main(String[] args) {
    MyList l = new MyList(); // ここだけ変更
    l.add("a"); l.add("b"); l.add("c"); l.add("d");
    l.add(2, "X"); l.add(3, "Y"); l.remove(4);
    for(Iterator i = l.iterator(); i.hasNext(); ) {
      System.out.println(i.next());
    }
  }
}
```

で、MyList の方を示そう。クラス MyList は、以下でずっと使うので public クラスとして**別の Java ソースファイル**「MyList.java」として用意する。その中では、Object の配列を持っていて、そこに要素を格納している。ただし、要素は add() するたびに増えるので、「現在いくつ要素を格納しているか」を保持する変数 count を別に持つようにした。

```
import java.util.*;

class MyList {
  Object[] a = new Object[10];
  int count = 0;
```

では、ArrayList と同様のメソッドで主要なものを用意しよう。まず大きさを調べる size() は count を返すだけ、あと要素を取り出したり書き換えたりするメソッド set() と get() は配列 a の対応する場所をアクセスするだけだから簡単である。

```
  public int size() { return count; }
  public Object get(int i) { return a[i]; }
  public Object set(int i, Object e) { Object v=a[i]; a[i]=e; return v; }
  public void add(Object e) { add(count, e); }
```

add(o) は「最後の場所に挿入すればよい」わけだから、add(i, o) を呼ぶだけになっている。ではそちらの方を見てみよう。

```
  public void add(int i, Object e) {
    if(count >= a.length) increaseCapacity();
    for(int k = count-1; k >= i; --k) a[k+1] = a[k];
    a[i] = e; ++count;
  }
  public void remove(int i) {
    for(int k = i; k < count-1; ++k) a[k] = a[k+1];
    --count;
  }
```

もし配列 a が満杯だったらそれ以上要素が入れられない…ので、そのときは increaseCapacity() を呼び出して配列を大きくする。で、配列に余裕ができたところで、指定位置より後ろの要素をすべて 1 ずつつ後ろにずらす。そして、空

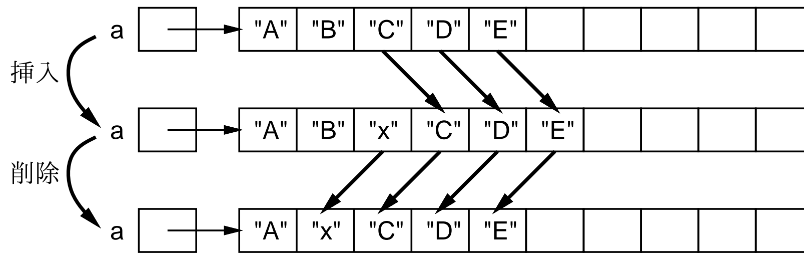

図 1: 挿入と削除

いたところに挿入すべき要素を格納して count を 1 ふやす。逆に要素を削るのは逆に削る場所から後ろをずらして詰めてしまえばよい (図 1)。

さて、increaseCapacity() は実は次に示すメソッドである (private と指定してあるので、このクラスの中からだけしか呼び出せない)。

```
private void increaseCapacity() {
  Object[] a1 = new Object[a.length + 10];
  for(int i = 0; i < count; ++i) a1[i] = a[i];
  a = a1;
}
```

つまり、新しい配列 a1 をこれまのでの配列 a より 10 だけ大きいサイズで用意し、要素をすべてコピーしてから、最後に a をこちらに「取り換えて」しまう。これでちゃんと配列 a の容量が増えたわけだ (図 2)。魔法のようでしょう?

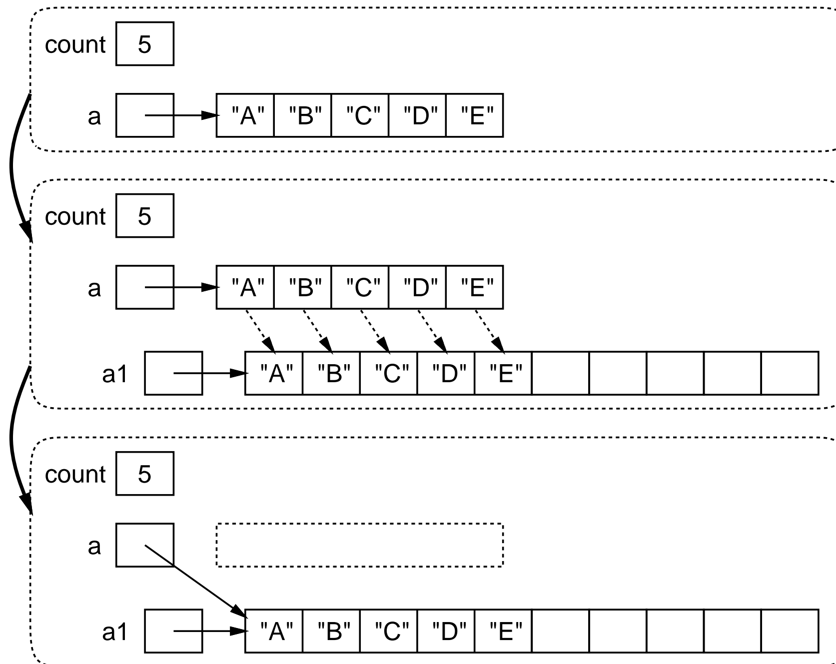

図 2: 内部で使う配列の容量を増やす

最後に、iterator() を示そう。これは Iterator インタフェースを implements する内部クラス MyIter のインスタンスを作って返す。MyIter は「今どこまで返したか」を自分のインスタンス変数として持ち、これと外側の (MyList の) インスタンス変数を参照して値を順番に返して行くだけである。

```
public Iterator iterator() { return new MyIter(); }
class MyIter implements Iterator {
  int i = 0;
  public boolean hasNext() { return i < count; }
```

```
      public Object next() { Object v = a[i]; ++i; return v; }
    }
  } //←クラスの終わりの「}」
```

なお、MyIter のインスタンスを作るのはここ 1 個所だけなので、無名内部クラスにして次のようにしてもよい (わずかに短いけど分かりにくいかも?)。

```
    public Iterator iterator() {
      return new Iterator() {
        int i = 0;
        public boolean hasNext() { return i < count; }
        public Object next() { Object v = a[i]; ++i; return v; }
      }; //← return の終わりの「;」
    }
  } //←クラスの終わりの「}」
```

このように、Iterator 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。バラバラに説明したので、MyList のソースをまとめて掲載しておく。

```
import java.util.*;

class MyList {
  Object[] a = new Object[10];
  int count = 0;
  public int size() { return count; }
  public Object get(int i) { return a[i]; }
  public Object set(int i, Object e) { Object v=a[i]; a[i]=e; return v; }
  public void add(Object e) { add(count, e); }
  public void add(int i, Object e) {
    if(count >= a.length) increaseCapacity();
    for(int k = count-1; k >= i; --k) a[k+1] = a[k];
    a[i] = e; ++count;
  }
  private void increaseCapacity() {
    Object[] a1 = new Object[a.length + 10];
    for(int i = 0; i < count; ++i) a1[i] = a[i];
    a = a1;
  }
  public void remove(int i) {
    for(int k = i; k < count-1; ++k) a[k] = a[k+1];
    --count;
  }
  public Iterator iterator() {
    return new Iterator() {
      int i = 0;
      public boolean hasNext() { return i < count; }
      public Object next() { Object v = a[i]; ++i; return v; }
      public void remove() { throw new UnsupportedOperationException(); }
    };
  }
}
```

演習 2 上 2 つの例題のコードを打ち込んでそのまま動かせ。ただし MyList は長いので次のようにしてコピーしてきてもよい。

```
cp  /home01/kuno/work/MyList.java  MyList.java
```

演習 3 上で出て来た iterator() は「入れた順に」要素を返す Iterator オブジェクトを作っていた。「入れたのとは逆順に」返すような Iterator オブジェクトを作るメソッド reversedIterator() も提供し、それを呼び出してみよ。また、その他の好きな順番 (ランダム順とか) で返すものもやってみよ。

演習 4 MyIter はメソッド remove() を実装していない (例外を返す) が、これを実装して、「呼ばれた時は最後に next() で返した要素を削除する」「まだ next() が呼ばれていない時や既に 1 回 remove() してしまったや要素の終りまで来てしまっているときなど、削除するものがない場合は例外 IllegalStateException を投げる」ようにしてみよ。(この演習をやれば remove() の用途や動作内容が分かりますね。)

というわけで、この方法を進めていけば要素を「さまざまな方法で」たどれるようになることが分かる。

6 データと計算量

さて、上で作った MyList の「性能」はどんなものだろうか? これを計るために次のような例題を作った。

```
import java.util.*;

public class R12Sample4 {
  public static void main(String[] args) {
    MyList l = new MyList();
    long t0 = System.currentTimeMillis();
    for(int i = 0; i < 10000; ++i) l.add("a");
    long t1 = System.currentTimeMillis();
    System.out.println("time = " + (t1-t0));
  }
}
```

これはつまり、MyList に「"a"」という文字列を 1 万回追加したときに要する時間を (ミリ秒単位で) 計っている。動かしてみよう。

```
% javac MyList.java
% javac R12Sample4.java
% java R12Sample4
time = 629
%
```

つまり 1 万個の add() には 629 ミリ秒掛かった。さて、では 2 万個だとどうなると思いますか? また 3 万個だと? やってみたところ 2 万個だと 2360 ミリ秒、3 万個だと 5453 ミリ秒となった (私の手元のマシンで)。つまり、個数を N とすると、処理に要する時間は N^2 に比例するわけだ。なぜだか分かりますか?

上の MyList のコードでは、配列が満杯になるごとに配列の大きさを 10 個増やして内容をコピーしていた。ということは、最初に満杯になったときは 10 個の要素をコピーし、次に満杯になった時は 20 個の要素をコピーし、次は 30 個をコピーし…と繰り返されるわけである。この調子で $10M$ 個を挿入すると、挿入に伴うコピー量の総和は

$$10 + 20 + 30 + \ldots + 10M = 10\frac{M(M+1)}{2}$$

ここで個数を簡単のため 10 の倍数だとして $N = 10M$ と置くとコピー量の総和は

$$\frac{1}{10}N^2 + 10N$$

となり、N が大きくなればほとんど N^2 に比例するわけだ。

このように、プログラムの計算量は多くの場合、「データの量 N に比例する」のではなく、それより大きい「$N^{1.5}$」「$NlogN$」「N^2」などに比例する。ある仕事をするのに計算時間が 10 分で済んだから、10 倍のデータを食べさせて昼飯でも食ってこよう、というのがいかに無謀だか分かりますね?

演習 5 上の例題を打ち込み、自分が使っているマシンでの処理時間を計測してみよ。マシンがのろくて 1 万で多すぎるようなら適宜減らしてもよいが、数個の値で計測してグラフを描き、どのような曲線に乗っているか確認すること。

演習 6 MyList を改良して、`add()` に掛かる「平均時間を」もっと速くしてみよ。演習 5 と同様に計測し、どのようになったか報告せよ。

ヒント: うまくやると、処理に要する時間が N^2 ではなく N に (ほぼ) 比例するようにできるので工夫してみよう。うまく行ったら、なぜそうなるのかも考えてみること。

演習 7 `add("a")` の代りに `add(0, "a")` で先頭に追加する場合についても同様に計測し、改良してみよ。

演習 8 途中の場所に add(i, o) で挿入する場合でも遅くならないような工夫ができないかどうか、考えてみよ。ヒント: 同じ位置 (ないしその近辺) に連続して、なら速くする方法がある。ランダムにどの位置でも、というのは難しいと思う。

演習 9 `add()` だけでなく `remove()` についても同様に計測し高速化を試みよ。

7 おまけ: 計算量を減らす工夫と適応的なデータ構造

上の演習、いかがでしたか。実は、挿入の時のコピー量の総和を減らすのはそんなに難しくない。たとえば、配列を「10 ずつ増やす」代りに「100 ずつ増やす」と、コピーの量はおよそ $\frac{1}{10}$ になる。しかしこれでもまだ甘い! というのは、N^2 に比例することに変わりはないから。そうではなく「2 倍ずつに増やす」とどうだろう?

$$1 + 2 + 4 + 8 + \ldots + N = 2N - 1$$

だから、たとえ配列の大きさが「1」から始まったとしても、N に比例する計算量で済むようになってしまう。ちょっとすごいでしょう?

さて、末尾に追加するときはこれでいいとして、先頭に追加するときはどうだろう? 先頭に追加するときは「必ず全部の要素を後ろへずらさなければならない」からそのコピー量だけ考えても

$$1 + 2 + 3 + 4 + \ldots + N = \frac{N(N-1)}{2}$$

で、また N^2 に比例してしまいそうに思えますか? それは「配列の先頭からデータを詰めている」からであって、図 3 の (B) のようにデータを配列の終わり側に詰めておけば先頭への挿入はすぐできるようになる。その代り末尾への挿入がのろいって? そうなのだけれど、先頭への挿入が連続し、それから末尾への挿入が連続し、のような場合はその切り替わりのところでデータを (A) と (B) の間で移動すればよい。だったら、真中付近での挿入/削除も (C) のようにその位置に「ギャップ」が来るようにしておけばいい。このように、操作に応じてデータの入れ方を変えて計算量を有利に保つやり方を「適応的なデータ構造」と呼ぶ。配列 1 個だとランダムにあちこちに挿入されたらどうしようもないが、(D) のように木構造にして行くことでバラバラな挿入や削除に対応することができる。まあお話だけということで。

A 本日の課題 12A

本日は「演習 1(の小課題)」から 1 つ、そして「演習 4〜9 のどれか」から 1 つ選び、作成したプログラム 2 つのソースコードをいつも通り、「本日中に」久野までメールで送付してください。具体的な内容は次の通り。

1. Subject: は「Report 12A」とする。
2. 学籍番号、氏名、投稿日時を書く。
3. 選んだプログラム 1 つのソース。
4. その簡単な説明。
5. 選んだもう 1 つのプログラムのソース。
6. その簡単な説明 (計測内容等があればそれも含む)。
7. 下記のアンケートの回答。

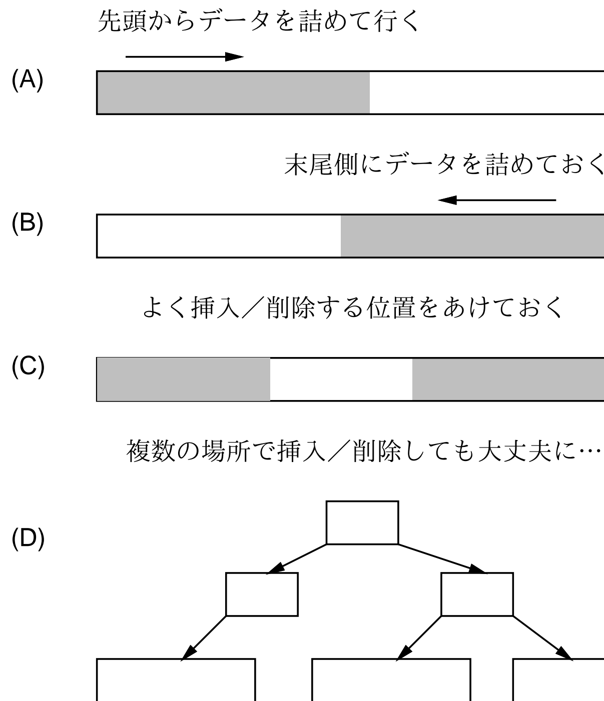

図 3: ギャップの位置を適応的に変化させる

Q1. Iterator インタフェースについて納得しましたか?

Q2. ArrayList のような「いれもの」の作り方が分かりましたか? また「計算量」についてはどうですか?

Q3. ここまで来て「プログラミング」についてどのように考えるようになりましたか。

B　おまけ: パッケージ機能と保護設定

これまで標準 API のクラス群を使うために import は多用してきたし、その関係でパッケージの参照方法についても学んで来たが、まだ自分でパッケージを作ることはなかった。しかし、そろそろ「プログラムがごちゃごちゃになる」ことを防いだり、他人と共同でプログラムを作ることを考えて、パッケージについていちおう説明しておこう。そして、パッケージを学んではじめて意味がわかる、public 等の保護設定の話題についても説明する。

B.1　パッケージ機能

まず、パッケージとはどういうものかについて改めて整理しておく。

- java.awt.*、java.awt.event.*などのように、階層状にクラス名をつけることができ、
- クラス名が同じでもパッケージが違っていれば混同することなく両方のクラスが使い分けられるような機能、のことをパッケージ機能という。

この「階層状」というのは Unix などのディレクトリ構造とよく似ていますね? 実は、パッケージ内のクラスはちょうどそのパッケージ名と同じディレクトリ構造に格納することになっている (図 4 左)。つまり、パッケージはファイルを整理する手段とクラスを整理する手段とを「兼ねて」いるわけだ。なお、これと併せて、パッケージに入れるクラスのソースファイルは 1 行目に必ず

```
package パッケージ名;
```

という指定を入れなければならない。

ところで、この「階層構造」の先頭位置はどこなのだろう? もちろん、システムのルートディレクトリではない (あなたが使っているシステムに「/java」というディレクトリはないでしょう?)。実は、「先頭位置」は複数持つことができる。そうしないと、あなたが mypkg というパッケージを作ったとして、それを java.lang.*などの標準パッケージとを

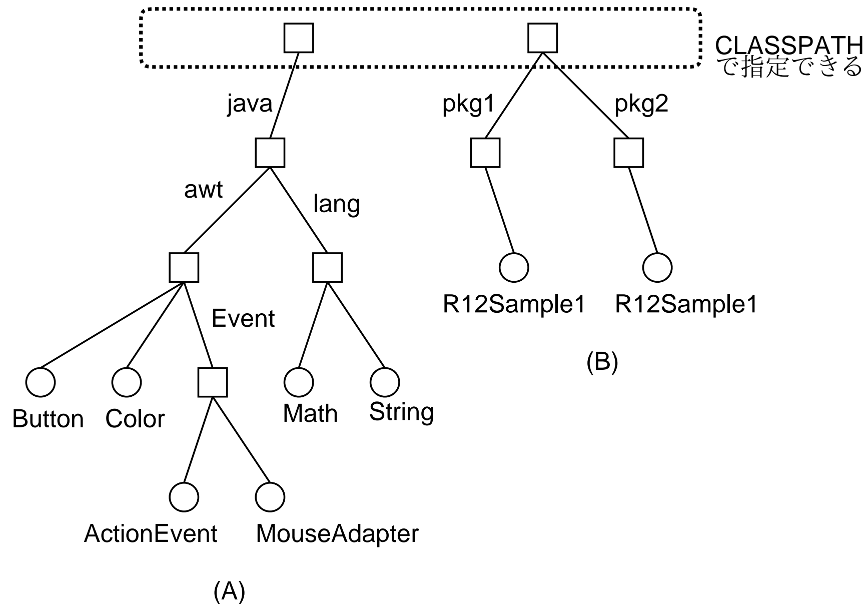

図 4: パッケージとディレクトリ/ファイルの対応

同じ場所に入れることになり、大変困る (と言うのは、あなたの個人用パッケージをシステム標準の場所に入れたくないから)。

その複数の先頭は、環境変数 CLASSPATH で指定することができる。つまり

```
setenv  CLASSPATH  Dir1:Dir2:···:DirN  ← Unix、Mac OS-X
set CLASSPATH = Dir1;Dir2;···;DirN     ← Windows
```

のように複数のディレクトリを「:」(Windows では「;」で区切ってならべた値を CLASSPATH に指定しておくと、これらのディレクトリ以下に置かれたパッケージがすべて利用できるように (つまり import で指定できるように) なる。では、CLASSPATH を設定しなかったら…? そのときは、カレントディレクトリ「.」だけが指定されているものとみなされる。なお、標準の `java.*`パッケージ群はこれらとは別に常に指定されているものとみなされる。ではまず、この状態で試してみよう。まず、パッケージ用のディレクトリを 2 つ作る (以下「%」はプロンプトなので打ち込む必要はない)。

```
% cd          ←後の演習の都合上、ホームディレクトリで作る
% umask 022   ←以下で作るファイルは誰でも読めるように
% mkdir pkg1 pkg2        ←ディレクトリを 2 つ作る
```

では Java ソースを打ち込もう。

```
package pkg1;

public class R15Sample1 {
  public static void main(String[] args) {
    System.out.println("Kuno's R15Sample1 in pkg1.");
  }
}
```

このファイルは当然、ディレクトリ pkg1 に置くこと。同様に、

```
package pkg2;

public class R15Sample1 {
  public static void main(String[] args) {
    System.out.println("Kuno's R15Sample1 in pkg2.");
  }
}
```

こちらのファイルは pkg2 に置くこと。両方ともコンパイルするが、このときそのディレクトリへは移動せず、ホームディレクトリにいる状態で次のようにファイルを指定してコンパイルする。

```
% javac pkg1/R15Sample1.java
% javac pkg2/R15Sample1.java
```

そして、そのまま pkg1 と pkg2 のあるディレクトリ (つまりホームディレクトリ) で次のように実行してみる。

```
% java pkg1.R15Sample1
...
% java pkg2.R15Sample1
...
```

クラス名が同じでも、パッケージが違うのできちんと区別できる。

なお、外部に公開するパッケージ名は、その組織/個人のフルドメイン名に基づいて、その左右をひっくり返してつけることになっている。たとえばここのドメインは ecc.u-tokyo.ac.jp だからパッケージは

```
package jp.ac.u-tokyo.ecc.なんとか;
```

などとするわけである (が、「-は許されない文字なので東大の人は皆困っているそうな)。こうすることで、複数組織が公開するパッケージにおいて名前が衝突しないようにできる。今は単なる演習だからこの規則には従わない。では次に、CLASSPATH を設定してみよう。

```
% setenv  CLASSPATH  /
```

ルートディレクトリを指定したので、Unix のクラス階層とパッケージ階層が同じになる (普通はこういうことはやらない! あくまで演習のため)。では、先の 2 つのファイルのパッケージ指定を

```
import homeXX.だれそれ.pkg1;
```

(「だれそれ」のところは自分のユーザ名、また XX のところはそれぞれの人のホームディレクトリの番号) に修正して、再度コンパイルする。今度は動かすとき次のように指定して動かす必要がある。

```
% java homeXX.だれそれ.pkg1.R15Sample1
```

演習 1 ここまでの演習を各自行え。うまく行ったら、そばに座っているほかの人のパッケージも指定してやってみること (その人の保護設定が適切でないとアクセスできないので注意)。

B.2 クラスやメソッドの修飾子

さて、これまで public とか private とかについてきちんと説明していなかったが、パッケージの説明が終ったのでようやく説明できるようになった。これらの修飾子は、クラスやインタフェースにつくものとメソッドや変数につくものがある。

```
public class XXX ... {    ←この public はクラスの保護指定
  putlic static double value;  ←こちらは変数の保護指定
  public XXX() { ... }    ←これはメソッド/コンストラクタに指定
  ...
}
```

まず、トップレベルの (入れ子でない) クラスやインタフェースにつく保護指定の修飾子は次の 2 通りがある。

- public — このクラスやインタフェースはパッケージ外部から参照できる。
- 無指定 — このクラスやインタフェースは同じパッケージの中からだけ参照できる。

言い忘れていたが、package 指定のないクラスはすべて「空っぽの」名前を持つ無名パッケージ (1 つだけ存在する) に含まれているものと見なされる。なので、これまで作ったプログラムでは public と指定していないクラスでも自由に参照できたわけである。

ところで、保護設定ではないがクラスにつけられる修飾子があと 2 つあるので説明しておく。

- abstract — このクラスは「抽象クラス」(抽象メソッドを持つクラス) であり、直接インスタンスを作ることができない。サブクラスを作ってそこで抽象メソッドをすべてオーバライドして実際に定義したら、そのインスタンスは作ることができる。
- final — このクラスはサブクラスを作ることができない。つまり、サブクラスを作って機能を拡張/変更されたくない場合に指定する。

次に、クラス内の各変数/メソッド/入れ子クラスにつく保護指定のための修飾子は次の 4 通りがある。

- public — この変数やメソッドやクラスはパッケージ外からアクセスできる。
- 無指定 — この変数やメソッドやクラスは同じパッケージの中からだけアクセスできる。
- private — この変数やメソッドやクラスは同じクラスの中からだけアクセスできる。
- protected — この変数やメソッドやクラスは基本的には private と同じだが、ただしこのクラスのサブクラスを作った場合には、そのサブクラス内でもアクセスできる。

変数やメソッドやクラスにつく保護指定ではない修飾子には次のものがある。

- static — おなじみ、クラス変数やクラスメソッドを意味する。クラスの場合は「インスタンスに付属していないクラス」を意味する。
- abstract — クラスに指定した場合は抽象クラス。メソッドに指定した場合はそのメソッドは抽象メソッドであり、ここではインタフェースだけ規定してコードは書かない (インタフェースでのメソッド指定の書き方)。サブクラスでこのメソッドをオーバライドすることを前提としている。
- final — クラスに指定した場合はこのクラスはサブクラスを作れない。メソッドの場合は、このメソッドはオーバライドできない。動作を変更させたくない場合に使う。また変数の場合は初期設定はできるが、コード中で値を変更できない。つまり「定数」を意味する。

C 最後に

皆様、短い間でしたがご静聴ありがとうございました。ここまで来た人は結構プログラミングができる人になっていると思います。これからも機会があったらプログラミングと親しまれて、特にプログラマにならなかったとしても「必要ならプログラムを作る人生」を歩んで頂ければ、私としては嬉しく思います。ではまた機会があればお会いしましょう。