計算機プログラミング'99　＃7

久野　靖*

1999.10.20

0　はじめに

今回は前回の GUI のような華々しいことは全くないけど、実際に大きなシステムを開発しようとした場合に重要な問題になる、パッケージと保護の話をさせて頂きます。また、後半では「どのようにクラスやインスタンスを配置してプログラム構造を設計するか」という観点からデザインパターン等の話をさせて頂きます。

そのほかの案としては: リフレクション

1　パッケージ機能

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学んでおこう。

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。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、Unix 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

```
export CLASSPATH=Dir1:Dir2:…:DirN
```

*筑波大学大学院経営システム科学専攻

のように複数のディレクトリを「:」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。

```
% cd        ←後の演習の都合上、ホームディレクトリで作る
% mkdir pkg1 pkg2        ←ディレクトリを 2 つ作る
% chmod a+rx pkg1 pkg2   ←〃、保護設定を「誰でも読める」に
```

では Java ソースを打ち込もう。

```
package pkg1;

public class R7Sample1 {
  public static void main(String[] args) {
    System.out.println("Kuno's R7Sample1 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 R7Sample1 {
  public static void main(String[] args) {
    System.out.println("Kuno's R7Sample1 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した上、クラスファイルをいつものように「誰でも読める」保護設定にしておく。そして、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.R7Sample1
...
% java pkg2.R7Sample1
...
```

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 GSSM のドメインは gssm.otsuka.tsukuba.ac.jp だからパッケージは

```
package jp.ac.tsukuba.otsuka.gssm. なんとか;
```

などとするわけである。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。

```
% export 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 u2.だれそれ.pkg1;
```

等とする (「だれそれ」のところは自分のユーザ名)。そして、動かすときは

```
% java u2.だれそれ.pkg1.R7Sample1
```

で動かす。

演習 1 ここまでの演習を各自行え。うまく行ったら、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)。

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、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、各変数/メソッド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。
- abstract — メソッドのみに指定。これを指定した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 — 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。

3　ダウンキャストとクラスの動的指定

既に何回も見てきたように、Java では親クラスの変数に子クラスのインスタンスを入れることができる。インタフェースとそれを実装しているクラスのインスタンスについても同様である。

```
class B extends A { ...
  public void m() { ... }  ←追加のメソッド
}

A x = new B();  // OK
```

この例で A 型の変数 x には B のインスタンスが入っている。しかし、この状態では B に固有のメソッド m() を呼ぶことはできない。

```
x.m();   // NG ... コンパイルエラーになる
```

なぜなら、変数 x に対してはあくまで A で定義しているメソッドだけが使えるようにコンパイラが検査するから (そうでないと x に A のインスタンスが入っているときに困る)。では m() を呼びたいときはどうするか? それには、B 型の変数に入れ直せばいい。

```
B y = (B)x;  // ダウンキャスト
y.m();       // OK
```

ここで、「(B)x」という式は、x の値が実際にはクラス B(ないしそのサブクラス以下) のインスタンスであることをチェックし、そのチェックが OK だった場合は値をクラス B の方として扱うことを意味する。これを「ダウンキャスト」と呼ぶ。では、実は x には A のインスタンスが入っていたら…? その場合は、チェックが失敗して ClassCastException という例外が発生する。

しかし、なんでわざわざ B 型の値を A 型の変数 x に入れて、また元の B 型に戻すなどという面倒くさいことをするのだろう? それは、変数代入に限らず引数渡しや返値の型も同様の規則が適用され、汎用的な動作を行うクラスやインタフェースはより「広い」クラスを扱うように定義されているからである。

たとえば、Vector などの「いれもの」のオブジェクトや Enumeration などの「汎用のインタフェース」はオブジェクトならば何でも扱えるようにするため、Object 型の値をやりとりするように定義してある。しかし実際には特定の型の値を入れて使うわけだが、取り出したところではあくまで Object 型として扱われてしまうので、それを元の型に戻す必要がある。

```
Vector vec = new Vector();
...
vec.add("ABC"); vec.add("DEF"); ... // String オブジェクトを入れる
...
for(Enumeration e = vec.elements(); e.hasMoreElements(); ) {
  Object o = e.nextElement();  // e.nextElement() は Object を返す
  String s = (String)o; // ここで元の型に戻す
  ...
}
```

実際には上の部分はもっと簡略に 1 行で

```
  String s = (String)(e.nextElement());
```

と書いてもいい (その方が普通である)。ところで、上の場合は Vector に String しか入れなかったので必ず String が帰って来るが、もっと「さまざまな」ものを入れる可能性がある場合には、必ず String にキャストできるとは限らない。ここで、String である場合だけ何らかの処理をする、という動作を行わせたいとすると、次の 2 つの方法がある。

- 常に String にキャストして、ClassCastException が発生した場合にそれを受け止めて別の処理をする。
- 特別な演算子 instanceof を使って、値が実際に String なのかどうかを予め判定する。

どちらでもよいが、通常は例外は例外的な場合だけに使いたいだろうから、後者の方法が普通である。たとえば、次のようになるわけだ。

```
Vector vec = new Vector();
...
vec.add("ABC"); vec.add("DEF"); ... // String オブジェクトを入れる
vec.add(new Integer(10)); ...       // 別のオブジェクトも入れる
...
for(Enumeration e = vec.elements(); e.hasMoreElements(); ) {
  Object o = e.nextElement();
  if(o instanceof String) {
    String s = (String)o; // 必ず成功するキャスト
    ...
  } else {
    ... // String でない場合の処理
  }
}
```

さて、この機能と関連の深いものに、文字列によるクラスの動的指定があるのでついでに取り上げておこう。

- Class.forName(文字列) — 「文字列」で指定したクラスを表す、「クラスオブジェクト」を返す。もちろん、存在しないクラスを指定すると例外 ClassNotFoundException が投げられる。
- クラスオブジェクト.newInstance() — そのクラスのインスタンスを生成する。ただし、引数なしのコンストラクタによってインスタンスが生成できるようなクラスでなければならない。これも、正しくない場合は InstantiationException などの例外が投げられる。

普通はこれらを組み合わせて

```
Object o = Class.forName(文字列).newInstance();
```

というふうに使うわけである。

では、この辺の機能を使って「皆で協力して何かを動かす」というのをやってみよう。まず、u1.kuno.work.R7StrFilte
というインタフェースを見ていただく。

```
package u1.kuno.work;

public interface R7StrFilter {
  public void putStr(String s);
  public String getStr();
}
```

このインタフェースは、文字列を putStr() で与え、次に getStr() で取り出すと何らかの「加工」がなされている (つまり Unix のフィルタの文字列版のようなもの)、という機能を持ったクラスを表すつもり。さて、これを実装するクラスを 2 つ示してみる。まず、「1 つ前の値を覚えていて順送りする」。

```
package u1.kuno.work;

public class R7SF1 implements u1.kuno.work.R7StrFilter {
  String s1, s2;
```

```
  public R7SF1() { s1 = "abc"; s2 = "def"; }
  public void putStr(String s) { s2 = s1; s1 = s; }
  public String getStr() { return s2; }
}
```

簡単ですね? 次は前にやった「左右ひっくり返し」。

```
package u1.kuno.work;

public class R7SF2 implements u1.kuno.work.R7StrFilter {
  String str = "";
  public void putStr(String s) {
    str = "";
    for(int i = s.length()-1; i >= 0; --i) { str += s.charAt(i); }
  }
  public String getStr() { return str; }
}
```

さて、こういうフィルタを自由に組み合わせて試すプログラムを示す。

```
package u1.kuno.work;
import java.io.*;
import java.util.*;

class R7Sample2 {
  public static void main(String[] args) {
    try {
      Vector vec = new Vector();
      for(int i = 0; i < args.length; ++i) {
        Object o = Class.forName(args[i]).newInstance();
        if(o instanceof u1.kuno.work.R7StrFilter) {
          vec.add(o);
        } else {
          System.out.println("Not a filter: " + args[i]);
        }
      }
      BufferedReader in =
        new BufferedReader(new InputStreamReader(System.in));
      while(true) {
        System.out.print("String> "); System.out.flush();
        String str = in.readLine();
        if(str.equals("")) break;
        for(Enumeration e = vec.elements(); e.hasMoreElements(); ) {
          Object o = e.nextElement();
          u1.kuno.work.R7StrFilter f = (u1.kuno.work.R7StrFilter)o;
          f.putStr(str); str = f.getStr();
        }
        System.out.println(str);
      }
    } catch(Exception e) { System.err.println("!!"+e); }
```

```
    }
  }
```

すなわち、配列 args に渡された引数の各文字列をフルクラス名だと思って Class.forName().newInstance() でインスタンスを作り、それらを Vector に集める。その際、予め u1.kuno.work.R7StrFilter インタフェースにキャストできるものだけを集める。フィルタ群が完成したら、前にやったように 1 行ずつ文字列を打ち込み、受け取った文字列について Vector に格納されているフィルタを 1 個ずつ適用していく。たとえば次のような具合である。

```
% java u1.kuno.work.R7Sampel2 u1.kuno.work.R7SF1
String> a
abc
String> b
a
String> c
b
String>
% java u1.kuno.work.R7Sampel2 u1.kuno.work.R7SF1 u1.kuno.work.R7SF2
String> this
cba
String> is
siht
String> a
si
String>
%
```

演習 2 上のインタフェース u1.kuno.work.R7StrFilter を implements するクラスを 1 つ以上作れ。例えば次のような機能を持っているもの。

a. 母音 aeiou を全部「*」にする。
b. 1 文字巡回させる。abcde → bcdea のように。
c. 2 つくっつける。abc → abcabc のように。
d. その他、自分で思いつく文字列の加工なら何でも。

クラスは演習 1 で作ったパッケージ「u2. だれそれ.pkg1」に入れること。コンパイルも動かすのも CLASSPATH が「/」になっている状態でないとうまく行かない。動かすのは R7Sample2 をわざわざ入力する必要はなく、次のようにして久野のディレクトリに置かれたものをそのまま実行すればよい。

```
% java u1.kuno.work.R7Sample2 u2. だれそれ.pkg1. クラス名
```

うまく行ったら、他人が作ったものも動かしてみて、どんなフィルタであるかあてっこしてみよ。

4 イベント処理の補遺: マウスイベント

さて、この後の例題で使うのでちょっと脱線して、これまでに触れる機会がなかったマウスイベントの処理の話をしておく。基本的には、マウスイベントもボタン等が生成するイベントと類似した形で通知される。ただし次のような違いがある。

- マウスの場合は、ボタン等の特定の部品と違って、ウィンドウ領域のどこからでもイベントが発生する。たとえばアプレット領域全体に対してマウスイベントの受け取りを行うことなどもできる。

- ボタンではイベントは ActionEvent だったが、マウスでは MouseEvent と MouseMotionEvent の 2 種類のイベントがある。後者はマウスが「動くたび」に発生するのですごくイベント数が多くなるため、必要ない場合は受け取る処理をまったく行わない方が効率がよい。このために前者と分けてある。
- ActionEvent の受け取りは actionPerformed() というメソッド 1 つに決まっていたが、マウスの場合は mousePressed()、mouseReleased() などいくつもの種類がある。

ではとりあえず、マウスをドラグすると矩形が描けるアプレットという例題を見てみよう。

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

public class R7Sample3 extends Applet {
  int x1, y1, x2, y2, x0, y0, width, height;
  public void init() {
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent e) {
        x1 = x2 = e.getX(); y1 = y2 = e.getY(); calcbox(); repaint();
      }
      public void mouseReleased(MouseEvent e) {
        x2 = e.getX(); y2 = e.getY(); calcbox(); repaint();
      }
    });
    addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent e) {
        x2 = e.getX(); y2 = e.getY(); calcbox(); repaint();
      }
    });
  }
  private void calcbox() {
    x0 = Math.min(x1, x2); y0 = Math.min(y1, y2);
    width = Math.abs(x1-x2); height = Math.abs(y1-y2);
  }
  public void paint(Graphics g) {
    g.drawRect(x0, y0, width, height);
  }
}
```

なお、MouseAdapter や MouseMotionAdapter というのは、上で説明したような複数のメソッド (ただし何もしない) を予め用意したクラスで、無名内部クラスをこれらのクラスのサブクラスとして作成し、必要なメソッドだけをオーバライドして処理を記述している。

演習 3 上のアプレットをそのまま動かせ。

演習 4 もっと別な図形を描くようにしてみよ。ドロップダウンメニューで選択できるとよい。また、一度描いたものは消えないで次々に追加されていくようにできるとなおよい。

5 GUI 部品の動的生成

さて、今日の本題に戻って、今度はクラスの動的指定を利用して GUI 部品を「その場で」生成し配置するアプレットを作ってみよう。実は先の例題アプレットは「どの大きさでどこに部品を作るか」矩形を指定する

のに利用しようというつもりだった。

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

public class R7Sample4 extends Applet {
  int x1, y1, x2, y2, x0, y0, width, height;
  Label l0 = new Label("Component:");
  TextField t0 = new TextField();
  Button b0 = new Button("Create");
  Button b1 = new Button("Move");
  Label l1 = new Label();
  Component c0 = null;
  public void init() {
    setLayout(null);
    add(l0); l0.setBounds(10, 10, 60, 30);
    add(t0); t0.setBounds(80, 10, 200, 30);
    add(b0); b0.setBounds(290, 10, 60, 30);
    add(b1); b1.setBounds(360, 10, 40, 30);
    add(l1); l1.setBounds(10, 50, 380, 30);
    b0.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        try {
          c0 = (Component)Class.forName(t0.getText()).newInstance();
          R7Sample4.this.add(c0); c0.setBounds(x0, y0, width, height);
          t0.setText("");
        } catch(Exception ex) { l1.setText(ex.toString()); }
      }
    });
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        if(c0 != null) { c0.setBounds(x0, y0, width, height); }
      }
    });
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent e) {
        x1 = x2 = e.getX(); y1 = y2 = e.getY(); calcbox(); repaint();
      }
      public void mouseReleased(MouseEvent e) {
        x2 = e.getX(); y2 = e.getY(); calcbox(); repaint();
      }
    });
    addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent e) {
        x2 = e.getX(); y2 = e.getY(); calcbox(); repaint();
      }
    });
```

```
  }
  private void calcbox() {
    x0 = Math.min(x1, x2); y0 = Math.min(y1, y2);
    width = Math.abs(x1-x2); height = Math.abs(y1-y2);
  }
  public void paint(Graphics g) {
    g.drawRect(x0, y0, width, height);
  }
}
```

演習 5 上のアプレットをそのまま動かせ。

演習 6 上のアプレットを改良して、部品の前景色と背景色が自由に指定できるようにしてみよ。色の指定方法は自分で設計する (一番簡単には、0〜255 の値を 3 つ指定させ、「new Color(値, 値, 値)」で Color オブジェクトを生成させる。

演習 7 上のアプレットを改良して、新しい部品を描いた後でも前の部品に戻って位置等が変更できるようにしてみよ。

6 自己反映機能

プログラミング言語における自己反映 (reflection) 機能とは、プログラムの中でそのプログラム自身 (ないしその一部分) に関する情報を取得したり、その動作を変更させたりするような機能を言う。つまり、普通ならプログラマしか知ったり変更したりできないようなことを、プログラムの中で自動的に行わせるような機能なわけである。

実は、今回学んだ機能のうち、インスタンスの動的生成、instanceof 演算子などは自己反映機能の一種だと言ってよい。実際、先の例題では普通だったら「new Button()」とか「new TextField()」とかプログラム中に書かなければならないところを、その名前文字列をプログラムで扱って任意の部品を画面に貼りつけていましたね? このように、自己反映機能を使うことでより柔軟性の高いソフトウェアを作ることができるようになる (ただしその代償として「分かりにくい」コードになる危険性も多々ある)。

ところで、先の例題ではあくまでも生成した部品が Component のサブクラスであることを前提として、そのメソッドを呼ぶだけだった。これをもっと進めて、「そのオブジェクトがどんなメソッドを持っているか調べて」「その任意のメソッドを呼ぶ」ことができるともっと柔軟にいろいろできる。

そのためには、次のような機構が利用できる。

- 任意のオブジェクト (インスタンス) に対して、メソッド getClass() を呼ぶことでそのオブジェクトが所属しているクラスの情報を保持するクラスオブジェクト (クラス Class のインスタンス) を取得できる。
- クラスオブジェクトのメソッド getFields()、getMethods()、getConstructors() などを呼ぶことで、このクラスが持っている変数、メソッド、コンストラクタの情報を取り出すことができる。具体的には、これらの情報はクラス Field、Method、Constructor の配列型の値として返される。
- たとえば Method クラスのインスタンスに対しては、その getName()、getParameterType()、getReturnType() などのメソッドを使って、そのメソッドの名前、引数の型、返値の型などを調べることができる。さらに、メソッド invoke() を使って、実際にそのメソッドを呼び出す (!!!) ことができる。

つまり、プログラムの中で、何だかわからないクラスの名前からインスタンスを作り、そのインスタンスの性質を調べて、さらにそのインスタンスを操作するメソッドを呼び出すことができるわけである。

では、これを利用して先のアプレットを改良し、生成した GUI 部品について「文字列を 1 つだけ引数に持つ」ようなメソッドを自由に呼び出せる (その引数文字列はテキスト入力欄から入力できる) ようにしてみよう。

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.lang.reflect.*;

public class R7Sample5 extends Applet {
  int x1, y1, x2, y2, x0, y0, width, height;
  Component c0 = null;
  Label l0 = new Label("Component:");
  TextField t0 = new TextField();
  Button b0 = new Button("Create");
  Button b1 = new Button("Move");
  Label l1 = new Label();
  Choice c2 = new Choice();
  Label l2 = new Label("String:");
  TextField t2 = new TextField();
  Button b2 = new Button("Call");
  Method[] meths; int[] maps;
  public void init() {
    setLayout(null);
    add(l0); l0.setBounds(10, 10, 60, 30);
    add(t0); t0.setBounds(80, 10, 200, 30);
    add(b0); b0.setBounds(290, 10, 60, 30);
    add(b1); b1.setBounds(360, 10, 40, 30);
    add(l1); l1.setBounds(10, 40, 380, 30);
    add(c2); c2.setBounds(10, 70, 200, 30);
    add(l2); l2.setBounds(10, 110, 60, 30);
    add(t2); t2.setBounds(80, 110, 200, 30);
    add(b2); b2.setBounds(290, 110, 40, 30);
    b0.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        try {
          c0 = (Component)Class.forName(t0.getText()).newInstance();
          R7Sample5.this.add(c0); c0.setBounds(x0, y0, width, height);
          t0.setText("");
          meths = c0.getClass().getMethods();
          maps = new int[meths.length];
          c2.removeAll();
          for(int i = 0, k = 0; i < meths.length; ++i) {
            Class[] arg = meths[i].getParameterTypes();
            if(arg.length == 1 && arg[0] == String.class) {
              c2.add(meths[i].toString()); maps[k++] = i;
            }
          }
        } catch(Exception ex) { l1.setText(ex.toString()); }
      }
    });
    b1.addActionListener(new ActionListener() {
```

```
      public void actionPerformed(ActionEvent e) {
        if(c0 != null) { c0.setBounds(x0, y0, width, height); }
      }
    });
    b2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        try {
          Method m = meths[maps[c2.getSelectedIndex()]];
          m.invoke(c0, new Object[]{t2.getText()}); t2.setText("");
        } catch(Exception ex) { l1.setText(ex.toString()); }
      }
    });
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent e) {
        x1 = x2 = e.getX(); y1 = y2 = e.getY(); calcbox(); repaint();
      }
      public void mouseReleased(MouseEvent e) {
        x2 = e.getX(); y2 = e.getY(); calcbox(); repaint();
      }
    });
    addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent e) {
        x2 = e.getX(); y2 = e.getY(); calcbox(); repaint();
      }
    });
  }
  private void calcbox() {
    x0 = Math.min(x1, x2); y0 = Math.min(y1, y2);
    width = Math.abs(x1-x2); height = Math.abs(y1-y2);
  }
  public void paint(Graphics g) {
    g.drawRect(x0, y0, width, height);
  }
}
```

演習 8 上のアプレットをそのまま動かせ。

演習 9 「文字列 1 個を引数とする」ものだけでなく、もっとさまざまなメソッドが呼べるようにしてみよ。

演習 10 ここまでは GUI 部品を配置するだけだったが、ボタンが押されたときの ActionListener オブジェクトを設定して何らかの動作を行わせるようにしてみよ。たとえば 2 つの GUI 部品から getText() で文字列を取り出して来てそれを連結するとか数値として足し算するとか。その結果は当然、別の GUI 部品に setText() で表示する。

演習 10 まで来るともはや、これは一種の「ユーザインタフェース構築ツール」になってくる。なかなか「すごい」と思うのだけどどうですか?