情報科学 2006　久野クラス　＃ 12

久野　靖*

2007.1.30

はじめに

半年間に渡っておつき合い頂いた「情報科学」も今回が最終回ということで、最後の 1 回くらいは Java とオブジェクト指向を活かした内容を取り上げようと思います。Java プログラミングにはこういう内容もあるんだなと思って頂ければと思います。

1　継承とウィンドウプログラム

1.1　クラスと継承

ここまでクラスについて「データ構造とメソッドを一体にしたもの」と説明し、何回も例題で活用してきたが、実はクラスについてまだ説明していない重要なことがらがいくつかある。今回はこれらの機能と、それをどうやって使うのかという話を観光旅行的に (詳しくやる時間はないので要点をつまみ食いする形で) 説明していく。全部理解しなくても、ここではとりあえずこういう機能もあるのかと思って頂ければ十分である。

これまでに何回かクラスを作ってきて、「あのクラスとこのクラスは類似している」と思ったことはないだろうか。そのような場合に、同じことを繰り返し書くのではなく、既に作ったクラスを「土台にして」新しい部分を足すだけで次のクラスを作れるとよいかも知れない。Java を含む多くのオブジェクト指向言語には、実際そのようなことを可能にする機能が備わっていて、**継承** (inheritance) と呼ばれている。その説明をしよう。

繰り返すと「継承」というのは、あるクラスを下敷き (土台) にして新しいクラスを作ることを言う。なぜそういう機能があるのかというと、既にできている「土台」を元にそこにちょっとだけ継ぎ足すことで、少しずつ機能を増やして行けるとプログラムが作りやすい場合があるから。継承について以下にまとめておく。

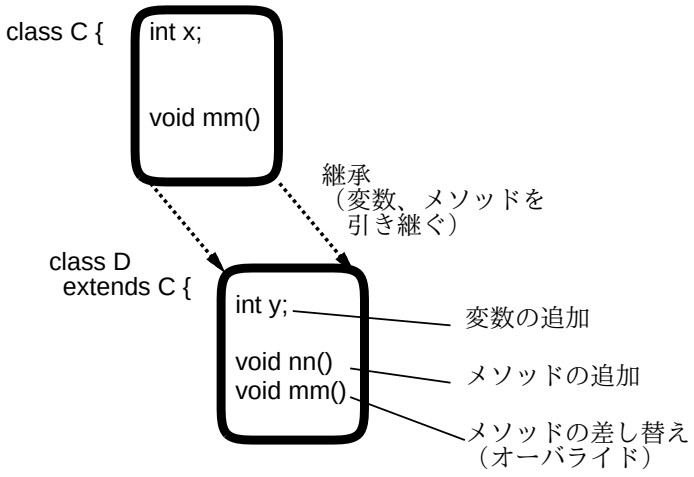

図 1: クラスと継承

*筑波大学大学院経営システム科学専攻

- 下敷きになるクラスを**親クラス**ないし**スーパークラス** (superclass)、新しく作るクラスを**子クラス**ないし**サブクラス** (subclass) と呼ぶ。
- Java では継承を指定するには、クラス定義の先頭部で「`class ... extends 親クラス ...`」と指定する。
- 親クラスの変数、メソッドは子クラスにそのまま引き継がれる (継承される)。
- 子クラスでは、変数やメソッドを新たに追加できる。
- さらに子クラスでは、メソッドを別のものに差し替えることができる。これを**上書き**ないし**オーバライド** (override) という。
- 子クラスのコンストラクタやメソッドの中からは **super** という特別な名前を指定することで親クラスのコンストラクタやメソッドを呼び出すことができる。

これだけでは何のことか分からないとは思うが、以下でその使い方について説明するので少々お待ちいただきたい。

その前にもう 1 つ重要なこととして、「親クラスの型の変数には子クラスのオブジェクトが入れられる」ということがある。さらにまた重要なこととして、「まったく `extends` を指定していないクラスは `Object` というクラスのサブクラスになる」という規則がある。つまりクラス `Object` はすべてのクラスの直接または間接の親クラスである。前に、「`Object` 型の変数には任意のオブジェクトをを入れることができる」と説明したのはそういうわけだったのだ。

そういうわけで、`Object` 型に限らずクラス型の変数にはそのクラスのインスタンスだけでなく、その任意のサブクラス (孫、ひ孫など間接的なサブクラスも含む) のインスタンスが入る可能性がある。これに関連して、このとき、この変数に対してメソッド M を呼び出すと、「実際にその変数に入っているオブジェクトのクラスで定義されている (継承されているものを含む)M が呼び出される」ことになる (もちろんそうでないと不都合ですよね?)。これを、実行時にどのメソッドになるかが決まることから**動的分配** (dynamic dispatch) と呼ぶ。

1.2 簡単な継承の例

簡単な例を見てみよう。次の例では、`Animal` というクラスのサブクラスとして `Dog`、`Cat` というクラスを作っている。

```
public class R12Sample1 {
  public static void main(String[] args) {
    Animal a1 = new Dog("Pochi", 10.0);
    if(Math.random() > 0.5) { a1 = new Cat("Tama", "stripe"); }
    System.out.println(a1.getName());
    a1.bark(); System.out.println("---"); a1.barkTwice();
    System.out.println(a1.getInfo());
  }
  static class Animal {
    String name;
    public Animal(String n) { name = n; }
    public String getName() { return name; }
    public void bark() { System.out.println("?"); }
    public void barkTwice() { bark(); bark(); }
    public String getInfo() { return getName(); }
  }
  static class Dog extends Animal {
    double weight;
    public Dog(String n, double w) { super(n); weight = w; }
    public void bark() { System.out.println("Vow!"); }
    public String getInfo() { return super.getInfo()+" weight = "+weight; }
    public double getWeight() { return weight; }
  }
  static class Cat extends Animal {
    String color;
    public Cat(String n, String c) { super(n); color = c; }
    public void bark() { System.out.println("Meaow!"); }
    public String getInfo() { return super.getInfo()+" color = "+color; }
    public String getColor() { return color; }
  }
}
```

親クラスとなる `Animal` では、インスタンス変数 `name`、`String` を 1 個受け取るコンストラクタ、そしてメソッド `getName()`、`bark()`、`barkTwice()`、`getInfo()` を定義している。

サブクラス Dog では、インスタンス変数 weight を追加し、コンストラクタからは「super(...)」で親クラスのコンストラクタを呼び出し (親クラスがコンストラクタを定義しているなら必ずそれを呼ばなければならない)、加えて自分のところで追加したインスタンス変数を初期化している (これがサブクラスのコンストラクタの標準的な形)。メソッド bark() は差し替え (オーバライド) により、自分の「ほえ方」で出力している。メソッド getInfo() もオーバライドしているが、その中で親クラスの getInfo() を呼ぶ必要があり、「super.getInfo()」により呼び出している (「super.」がないと自分自身を呼んでしまうので堂々めぐりになる)。getWeight() はこのクラスで追加したメソッドになる。それ以外のメソッド getName()、barkTwice() は親クラスから継承する。サブクラス Cat もだいたい同様である。

main() では変数 a1 にとりあえず Dog のインスタンスを入れるが、すぐ後で確率 0.5 で Cat に入れ換える。その後このオブジェクトに対して getName()、bark()、barkTwice()、getInfo() を呼び出す。これらのメソッドは、a1 に Dog オブジェクト、Cat オブジェクトのどちらが入っているかによって、そのクラスのメソッドを呼び出す。具体的には次の通り。

- getName() — いずれのクラスでも Animal から継承しているので、Animal で定義されているものが使われる。
- bark() — いずれのクラスでも差し替え (オーバライド) しているので、犬猫どちらかに応じそちらの bark() が呼ばれる。
- barkTwice() — いずれのクラスでも Animal から継承しているので、Animal で定義されているものが使われるが、その中で bark() を呼び出していることに注意。この bark() は犬猫どちらであるかに応じてそちらのものが呼ばれる。
- getInfo() — いずれのクラスでも差し替え (オーバライド) しているので、そちらのものが呼ばれる。どちらでも、getName() を呼んで名前を取り出し、その後ろに犬なら重さ、猫なら毛の色をくっつけた文字列を返す。

実際に動かしてみよう。

```
% java R12Sample1
Tama
Meaow!
---
Meaow!
Meaow!
Tama color = stripe
%
```

実行内容はちっとも面白くないが、上で説明してきたような原理の確認ということで。

1.3 抽象クラスと抽象メソッド

ところで、上の例題では Animal クラスのサブクラスとして Dog クラスと Cat クラスを作成したが、Animal のインスタンスも作成はできるようになっていた。しかし実際には「犬」「猫」という動物はあっても「動物」という動物はない。つまり Animal は複数の動物に共通の部分を定義する「起き場所」のようなクラスなわけであり、そのオブジェクト自体は作り出さない方が普通である。そのようなクラスを一般に**抽象クラス** (abstract class) と呼び、クラスの先頭に abstract というキーワードをつけて表す。抽象クラスに指定するとそのインスタンスを生成することが禁止される。

また、抽象クラスのメソッドの一部は「サブクラスで具体的な動作を決めるため、その名前と引数だけを定義する」のにとどめる方が適切なものがある (たとえば上の例では bark() がこれに相当する)。このようなメソッドは**抽象メソッド** (abstract method) と呼び、これも先頭に abstract というキーワードをつけて表し、なおかつ動作本体を省略する。

クラス Animal をこれらの変更をほどこした抽象クラスに書き直すと次のようになるだろう。

```
static abstract class Animal {          // 抽象クラスにした
  String name;
  public Animal(String n) { name = n; }
  public String getName() { return name; }
  public abstract void bark();          // 抽象メソッドにした
  public void barkTwice() { bark(); bark(); }
  public String getInfo() { return getName(); }
}
```

いちどに多くのことを覚えるのも大変なので、とりあえずこういう書き方もできる程度に思っておいてもらえればよい。

1.4 窓を作り出すプログラム

ここまでのプログラムはどれも、コマンド窓の中で動作し、入出力もコマンド窓の入出力かファイル入出力のみを利用していた。しかし今日では、ウィンドウシステムの機能を利用して自分独自の窓を開いて動作するプログラムの方が一般的である。ここでは Java でそのようなプログラムを作ってみよう。

基本的にはそれは難しいことではなく、Java の標準ライブラリに含まれている「窓」オブジェクト (JFrame クラスのインスタンス) を活用すればよいだけのことである。ただし、JFrame のインスタンスを生成しただけでは、真っ白な内容のない窓ができるだけである。というのは、「窓の中をどのように表示するか」はそれぞれのプログラムを作る人が決めることであり、JFrame 側では決められないからである。

では自分が窓の中身を決めるとして、それにはどうしたらいいのだろうか? それには「自分が作るクラスを JFrame のサブクラスにして、そこでメソッド paint() を差し替える」ようにする。なぜそれでいいかというと、paint() はその窓の中身を描くという分担のメソッドであり、それを差し替えて自分独自の内容にすることで、自分独自の内容が描けるようになるわけである。paint() は引数として Graphics オブジェクトを 1 個受け取るが、このオブジェクトは窓の中身を描画する「ペン」に相当し、そのメソッドを呼び出すことで窓の内容を色々に描くことができる。また、描画以外のさまざまな窓の機能については継承によって JFrame が持つ機能が引き継がれて使われることになる。

このように、オブジェクト指向言語では、ある程度込み入ったプログラムについてはその「枠組み」を用意しておき、特定のクラスのサブクラスを作って特定のメソッドを差し替え (オーバライド) することで自分の必要に応じて作り変える、という形で作成することが多い。このような方法を一般に**アプリケーションフレームワーク** (application framework) と呼ぶ。アプリケーションフレームワークでは少量のコードで高度な機能を作り出すことができるが、その一方でプログラムの作り手は「大部分が他人が書いたものの中に」「少量の自分のコードを埋め込む」ことになるので、何が起きているのか分かりにくく、思い通りに行かない時にもその原因が分かりづらいという弱点がある。

図 2: 簡単な絵

では前置きはこれくらいにして、図 2 のような簡単な絵を表示するプログラムを見てみよう。

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

public class R12Sample2 extends JFrame {
  Font fn = new Font("Helvetica", Font.BOLD, 20);
  public R12Sample2() {
    setDefaultCloseOperation(EXIT_ON_CLOSE); setSize(400, 400);
  }
  public void paint(Graphics g) {
    g.setColor(Color.blue); g.setFont(fn); g.drawString("Hello", 100, 100);
    g.setColor(new Color(0.9f, 0.1f, 0.6f, 0.3f)); g.fillOval(120,50,120,80);
  }
  public static void main(String[] args) {new R12Sample2().setVisible(true);}
}
```

まず、グラフィクス関係のオブジェクトを使うため、import がこれまでと違うのに注意。そして、作成するクラスは上で説明したように「extends JFrame」と指定して JFrame のサブクラスとしている。

main() メソッドは一番最後にあるが、そこではこのクラス (R12Sample2 のインスタンスを生成し、setVisible() を呼んで見える状態にするだけである。

R12Sampel2 のインスタンスを生成すると、その最初にコンストラクタが呼び出されるので次にそこを見てみよう。コンストラクタの中では、「窓を閉じるとプログラムも終わる」ように設定し、なおかつ窓の大きさを 400x400 ピクセルに設定している。

あとメソッド paint() であるが、先に述べたようにこのメソッドが窓の中身をすべて描くことになる。そのために、引数として渡されて来る Graphics オブジェクト g の必要なメソッドを呼び出す。ここではまず色を青に、フォントを(予め用意してあった)Helvetica 20 ポイントに設定し、「Hello」という文字を描き、また色を半透明の桃色に設定し、楕円を描いている。

演習 1 API ドキュメント (クラスのページからたどれる) を開き、java.awt パッケージのクラス Color を見てさまざまな色の指定方法を確認せよ。またクラス Graphics を見て描画のためにどのような機能が提供されているか確認せよ。

演習 2 上の例題をそのまま打ち込んで動かせ。動いたら、窓の中に描く図形や文字を変更してみよ。[1]

1.5 スレッドとアニメーション

さて、上の例題では絵が動かないものだったので、わざわざ Java で描く理由はあんまりないようなものだった。それではつまらないので、絵を動かすようにしてみる。具体的には、文字の表示される y 座標を時間につれて上下にサイン曲線に従って動かしてみよう。

そのためには、プログラム自体の流れとは「別に」次のような動作をする「実行の流れ」を動かしておけばよい。

- 無限に繰り返し、
- 　0.1 秒待つ。
- 　time ← time + 0.1。
- 　ypos ← sin(time)。
- 　画面を再表示。
- 以上を繰り返す。

このような「独立した実行の流れ」のことを**スレッド** (thread) と呼び、Java では Thread クラスで作り出すことができる。具体的なやり方はいくつかあるが、ここでは次の方法による。

- Thread クラスのサブクラスを作る。
- その中に public void run() { ... } というメソッドを定義し、その中に「独立した実行の流れ」の内容を記述。

つまり、サブクラスを作って必要な部分を差し替え、という方法をここでも使っているわけだ。

あと 1 つ、画面に描くところも今度は JPanel というクラスのサブクラスを作って、その pain() を差し替えるように直した。JPanel というのは窓の中にはめ込むことのできる領域をあらわし、たとえば「絵を描く領域」「GUI 部品の領域」などを区分したいときに使う。1 つの絵しかないのになぜこれを使うかというと、アニメーションで絵を動かすときにちらちらしないための機能を JPanel が提供してくれるから。ではプログラムコードを示す。

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

public class R12Sample3 extends JFrame {
  Font fn = new Font("Helvetica", Font.BOLD, 20);
  double time = 0.0, xpos = 100.0, ypos = 100.0;
  JPanel panel = new MyPanel();
  public R12Sample3() {
    setDefaultCloseOperation(EXIT_ON_CLOSE); setSize(400, 400);
    getContentPane().add(panel); new MyThread().start();
  }
  class MyPanel extends JPanel {
    public void paint(Graphics g) {
      g.setColor(getBackground()); g.fillRect(0, 0, getWidth(), getHeight());
      g.setColor(Color.blue); g.setFont(fn);
      g.drawString("Hello", (int)xpos, (int)ypos);
      g.setColor(new Color(0.9f, 0.1f, 0.6f, 0.3f));
```

[1] たとえばループを使って徐々に位置と色を変えながら図形を描くとちょっときれいかも知れない。

```
      g.fillOval(120, 50, 120, 80);
    }
  }
  class MyThread extends Thread {
    public void run() {
      while(true) {
        try { sleep(100); } catch(Exception ex) { }
        time += 0.1; ypos = 100 + 100*Math.sin(time); panel.repaint();
      }
    }
  }
  public static void main(String[] args) {new R12Sample3().setVisible(true);}
}
```

今度は先の例と次の点が違っている。

- 時刻、XY 座標を変数として持つ。
- JPanel のサブクラス MyPanel を用意しそのインスタンスを作っている。
- コンストラクタの初期設定で、(1) 窓の中に MyPanel をはめ込み、(2)MyThread のインスタンスを作って実行を開始させている (これにより独立した実行の流れが動きはじめる)。
- 先の例題の paint() と同様のものを今度は MyPanel の中で定義しているが、何回も絵を描くので最初に背景色で全体を塗るようにしている。また文字を描く座標を変数によって指定している。
- MyThread は Thread のサブクラスで、上に示した疑似コードの内容を独立した実行の流れとして実行し続ける。panel の内容を再描画させるにはそのメソッド repaint() を呼べばよい。

ところで、クラス MyThread や MyPanel には static が指定されていないことに気がついただろうか? これらのクラスのメソッドの中からは、外側クラス (R12Sample3) のインスタンス変数 panel、time、ypos などを参照している。そのため、これらのクラスのインスタンスは外側クラスのオブジェクトに「くっついた」オブジェクトである必要がある。そのような場合は static を指定しない (クラスメソッドとインスタンスメソッドの違いと類似している)。このような static でない内側クラスのことを Java では**内部クラス** (inner class) と呼んでいる。

1.6 無名内部クラス

Java でウィンドウものを扱うときには内部クラスを多数使うので、それをなるべく簡単に書きたい。このため、内部クラスが (1)1 個の extends または implements(後述) のみを指定し、(2) コンストラクタを持たず、(3)1 箇所だけで使うようであれば、いちいち名前をつけずに短く書く方法を用意している。具体的には、

```
... new XXX() ...

class XXX extends YYY {
  // 中身
}
```

のように書くのがこれまで学んだ方法だが、それをくっつけて、

```
... new YYY() {
      //中身
    } ...
```

のように使う箇所に「そのまま」クラス定義を埋め込んで書く。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に書き直したものを示しておく。

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

public class R12Sample4 extends JFrame {
  Font fn = new Font("Helvetica", Font.BOLD, 20);
  double time = 0.0, xpos = 100.0, ypos = 100.0;
  JPanel panel = new JPanel() {
    public void paint(Graphics g) {
      g.setColor(getBackground()); g.fillRect(0, 0, getWidth(), getHeight());
      g.setColor(Color.blue); g.setFont(fn);
      g.drawString("Hello", (int)xpos, (int)ypos);
```

```
      g.setColor(new Color(0.9f, 0.1f, 0.6f, 0.3f));
      g.fillOval(120, 50, 120, 80);
    }
  };
  public R12Sample4() {
    setDefaultCloseOperation(EXIT_ON_CLOSE); setSize(400, 400);
    getContentPane().add(panel);
    new Thread() {
      public void run() {
        while(true) {
          try { sleep(100); } catch(Exception ex) { }
          time += 0.1; ypos = 100 + 100*Math.sin(time); panel.repaint();
        }
      }
    }.start();
  }
  public static void main(String[] args) {new R12Sample4().setVisible(true);}
}
```

演習 3 アニメーションの例題 (どちらのバージョンでもよい) をそのまま打ち込んで動かせ。動いたら動きの内容をもっと別のものにしてみよ。たとえば文字を横方向にも動かす、楕円の色や大きさを変える、図形を増やすなど。

注意! 今回の例題では「クラス名を指定するところ」が数箇所ある。そのどれかを直し忘れると動かなかったり、さらには別の例題のクラスが作られて動いてしまったりするので注意。

2 インタフェースと GUI 部品

2.1 オブジェクト指向グラフィクス

先の例題では文字が動いていたが、この調子で色々なものを動かすのは大変ですよね? それは作るプログラムが「すべての物体を動かしつつ表示する唯一の神様」をやらなければならないから。日本は多神教なので (?)、そうではなくて「それぞれの物体が生命を持って動く」ようにした方が分かりやすい。

次のプログラムはその考え方によって、複数の動く円を表示する。それぞれの円はオブジェクトであり、配列 a にまとめて入っている (ちなみに「count++」というのは「++count」と同様に変数を 1 増やすが、ただし増やす前の値を取るという式で、配列に順番に入れるにはこれが便利)。そして、paint() ではそれぞれの円に対して draw() を呼び出して画面に自分を描かせ、またスレッドではそれぞれの円に対して addTime() を呼び出して時刻を進める。このようにすることで、プログラムの構造が「物体」と「それを含んだ空間 (?)」にきれいに分けられる。

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

public class R12Sample5 extends JFrame {
  FlyingCircle[] a = new FlyingCircle[10];
  int count = 0;
  JPanel panel = new JPanel() {
    public void paint(Graphics g) {
      g.setColor(getBackground()); g.fillRect(0, 0, getWidth(), getHeight());
      for(int i = 0; i < count; ++i) { a[i].draw(g); }
    }
  };
  public R12Sample5() {
    setDefaultCloseOperation(EXIT_ON_CLOSE); setSize(400, 400);
    getContentPane().add(panel);
    a[count++] = new FlyingCircle(Color.red, 100, 100, 40, 25, 40, panel);
    a[count++] = new FlyingCircle(Color.blue, 100, 100, 30, 60, -90, panel);
    new Thread() {
      public void run() {
        while(true) {
          try { sleep(20); } catch(Exception ex) { }
          for(int i = 0; i < count; ++i) { a[i].addTime(0.02); }
          panel.repaint();
        }
      }
    }.start();
```

```
  }
  static class FlyingCircle {
    Color col;
    double xpos, ypos, rad, vx, vy;
    JPanel panel;
    public FlyingCircle(Color c, double x, double y, double r,
                        double vx1, double vy1, JPanel p) {
      col = c; xpos = x; ypos = y; rad = r; vx = vx1; vy = vy1; panel = p;
    }
    public void draw(Graphics g) {
      g.setColor(col);
      g.fillOval((int)(xpos-rad), (int)(ypos-rad), (int)rad*2, (int)rad*2);
    }
    public void addTime(double dt) {
      xpos += vx * dt; ypos += vy * dt;
      if(xpos<0 && vx<0 || xpos>panel.getWidth() && vx>0) { vx = -vx; }
      if(ypos<0 && vy<0 || ypos>panel.getHeight() && vy>0) { vy = -vy; }
    }
  }
  public static void main(String[] args) {new R12Sample5().setVisible(true);}
}
```

飛ぶ円のクラス内では、XY 座標、半径、XY 方向の速度を保持していて、時間につれての位置変化を (手抜きな Euler 法で) 計算している。ただの直線運動だとすぐ画面の外に消えてしまうので、panel を持たせておいてその範囲を超えたら速度を反転してはね返って来るようにしている。

このように、オブジェクト指向言語では「画面上に見えるもの」や (見えなくても)「プログラムが扱う何らかのもの」をそれぞれオブジェクトに対応させて扱うことで、込み入ったプログラムでも見通しよく構築できる。そして、これらのオブジェクトの定義 (クラス) については、継承やインタフェース (後述) を活用して構造化していくわけである (図 3)。

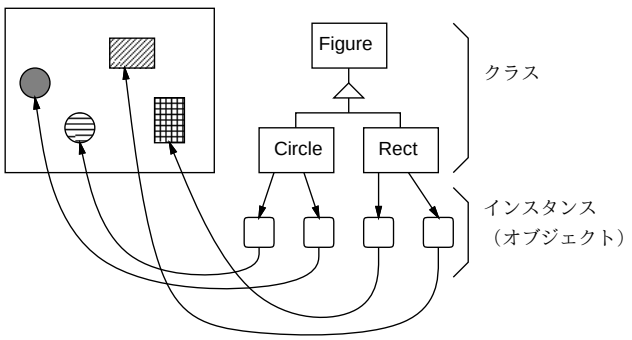

図 3: オブジェクト指向グラフィクス

演習 4 このプログラムを打ち込んで動かせ。動いたら円の数を増やしてみよ。また、円に「重力」を働かせて下に向かって放物運動をするようにしてみよ。はね返る時にいくらかエネルギーを失う (または逆に獲得する) ようにしてみてもよい。

2.2 さまざまな図形とインタフェース

前の例題では、配列に入れられるのは「飛ぶ円」だけだったが、もっとさまざまな図形を扱いたい場合はどうすればいいだろうか。

1 つの方法として、「図形」という抽象クラスを用意し、さまざまな図形をそのサブクラスとして作ることが考えられる。そうすれば、「図形」の配列にはこれらのさまざまな図形を入れておける。

確かにそれでもいいのだが、これらの「図形」について共通の部分がほとんどないので、抽象クラスにくくり出すべきものが見つからないという問題がある。実際に欲しいのは、機能をくくり出すことではなく、単に「これこれの共通のメソッドがある」というインタフェース (外からの使い方) を統一して扱いたい、ということである。

Java ではこのような場合に、**インタフェース**という機能が使える。インタフェースとは一群のメソッド名と引数に名前をつけたものである。たとえば、動く図形のために次のようなインタフェースを使うことを考える。

```
interface Animation {
  public void draw(Graphics g);
  public void addTime(double dt);
}
```

このインタフェースを持ったオブジェクトであれば、どれでも先に用意したような枠組みで利用することができる。

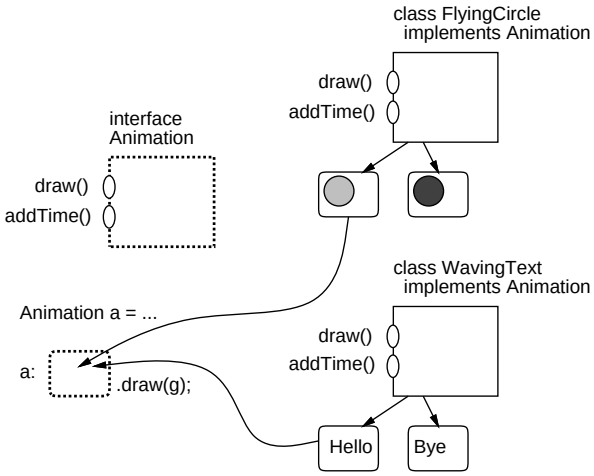

図 4: インタフェース

具体的には、インタフェースはクラスと同様に型であり、インタフェース型の変数にはそのインタフェースに従うクラスのインスタンスをどれでも入れることができる。そして、その変数に対するメソッド呼び出しを行うと、(動的分配により) 現在その変数に入っているオブジェクトに対応するメソッドが呼び出される (図 4)。

次の例題では、インタフェース型の配列を用意し、そこに複数の種類の動く図形を入れるようにしている。

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

public class R12Sample6 extends JFrame {
  Animation[] a = new Animation[20];
  int count = 0;
  JPanel panel = new JPanel() {
    public void paint(Graphics g) {
      g.setColor(getBackground()); g.fillRect(0, 0, getWidth(), getHeight());
      for(int i = 0; i < count; ++i) { a[i].draw(g); }
    }
  };
  public R12Sample6() {
    setDefaultCloseOperation(EXIT_ON_CLOSE); setSize(400, 400);
    getContentPane().add(panel);
    a[count++] = new FlyingCircle(Color.red, 100, 100, 40, 25, 40, panel);
    a[count++] = new FlyingCircle(Color.blue, 100, 100, 30, 60, -90, panel);
    a[count++] = new WavingText(Color.green, "Hello", 80, 200, 100, 5);
    new Thread() {
      public void run() {
        while(true) {
          try { sleep(20); } catch(Exception ex) { }
          for(int i = 0; i < count; ++i) { a[i].addTime(0.02); }
          panel.repaint();
        }
      }
    }.start();
  }
  interface Animation {
    public void draw(Graphics g);
    public void addTime(double dt);
  }
```

draw() とか addTime() とかは実際にそのとき a[i] に入っているオブジェクトに対応するものが呼ばれて実行されることに注意 (動的分配)。

さて、インタフェースに従うクラスを定義するときは継承の「extends クラス名」の代わりに「implements インタフェース名」を指定する。

```
  static class WavingText implements Animation {
    static Font fn = new Font("Helvetica", Font.BOLD, 20);
    Color col;
    String text;
    double xpos, ypos, yrad, theta = 0.0, vtheta;
    public WavingText(Color c, String s,
                      double x, double y, double r, double v) {
      col = c; text = s; xpos = x; ypos = y; yrad = r; vtheta = v;
    }
    public void draw(Graphics g) {
      int x = (int)xpos, y = (int)(ypos + yrad*Math.sin(theta));
      g.setColor(col); g.setFont(fn); g.drawString(text, x, y);
    }
    public void addTime(double dt) { theta += vtheta*dt; }
  }
  static class FlyingCircle implements Animation {
    Color col;
    double xpos, ypos, rad, vx, vy;
    JPanel panel;
    public FlyingCircle(Color c, double x, double y, double r,
                        double vx1, double vy1, JPanel p) {
      col = c; xpos = x; ypos = y; rad = r; vx = vx1; vy = vy1; panel = p;
    }
    public void draw(Graphics g) {
      g.setColor(col);
      g.fillOval((int)(xpos-rad), (int)(ypos-rad), (int)rad*2, (int)rad*2);
    }
    public void addTime(double dt) {
      xpos += vx * dt; ypos += vy * dt;
      if(xpos<0 && vx<0 || xpos>panel.getWidth() && vx>0) { vx = -vx; }
      if(ypos<0 && vy<0 || ypos>panel.getHeight() && vy>0) { vy = -vy; }
    }
  }
  public static void main(String[] args) {new R12Sample6().setVisible(true);}
}
```

このようにして、さまざまな図形を作りながら、それぞれの図形の記述は各クラスだけで扱い、全体部分は手をつけない、という分離が可能になる。

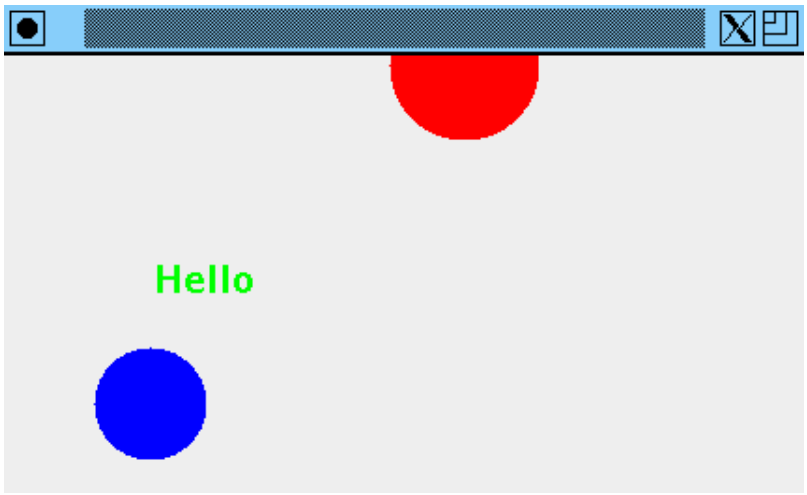

図 5: インタフェースを用いて複数図形を描く

演習 5 この例題 (コピーできるようにした) をそのまま動かせ。動いたらなにか自分独自の動き方をする図形を追加してみよ。

2.3 マウスイベントの受け取り

図形が動かせるようになったところで、入力について説明しよう。これまでの CUI アプリケーションでは入力命令を実行したところでプログラムは待ち状態になり、入力が終わらないと実行が先に進まなかった。

ウィンドウアプリケーションの場合、入力は**イベント** (外部事象) として受け取られる。イベントはプログラムの実行とは非同期に発生し、その情報がプログラムに送られて来る。たとえば「キーボード押し」も「マウスボタン押し」もいつでもユーザによって行えることなので、これらがプログラムと非同期に発生して送られて来る、というモデルは実態に合っている。

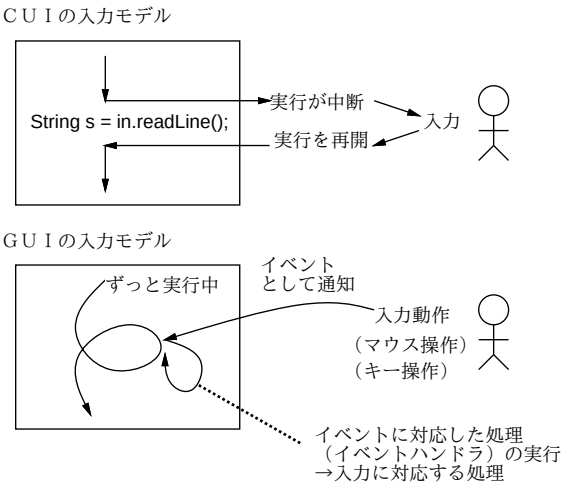

図 6: イベントによる入力の受け取り

それを受け取るためには、窓や窓の中の領域に**イベントハンドラ** (イベント受け取りのしくみ) を設定する必要がある。一番基本的なマウスイベントについて見てみよう。

- `JFrame` や `JPanel` などは `addMouseListener()` というメソッドを持つ。
- このメソッドは `MouseListener` というインタフェースを `implements` したクラスのインスタンスを引数として受け取る。
- このインタフェースにはマウスのイベントを受け取るためのメソッドが定義されている。たとえば `mousePressed()` というメソッドはマウスボタンが押されたというイベントを受け取る。

言い替えると、`MouseListener` インタフェースを `implements` し、`mousePressed()` というメソッドを定義したオブジェクト (アダプタオブジェクト) を用意し、それを `addMouseListenr()` で登録しておくと、窓やパネル上でマウスがクリックされたときこのメソッド (イベントハンドラ) が呼ばれる。なので、このオブジェクトにマウスボタンが押された時の動作を記述しておけばよいわけだ。なお、このメソッドには `MouseEvent` クラスのインスタンスが渡され、そのメソッド `getX()`、`getY()` によりボタンが押された時のマウスの XY 座標が取得できる。

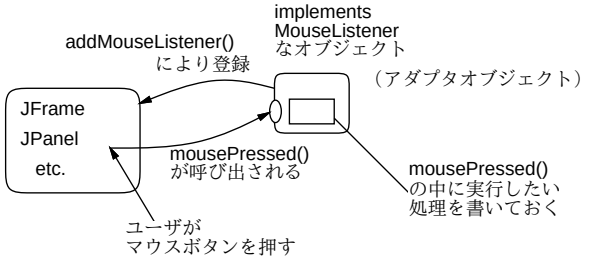

図 7: イベントハンドラの登録と処理

実際には、`MouseListener` インタフェースを直接 `implements` して色々なメソッドを用意すると長くなるので、その代わりに、`MouseAdapter` というクラス (これが `MouseListener` インタフェースを直接 `implements` している) のサブクラ

スを作って、必要なメソッドだけ差し替えるのがよい。この方針で、マウスボタンが押された時次の動作をするような例題を示す。

- マウスポインタが図形の上にあれば、その図形が重なりの一番上に出る。
- マウスポインタがどの図形の上にもなければ、新しい「飛ぶ円」をその位置に生成する。

「上にある」かどうかが問題になるため、Animation インタフェースに hit() というメソッドを追加した。このメソッドは指定した XY 座標について、その座標が図形の範囲内かどうかを論理値として返す。
例題であるが、冒頭部分は先の例題と変わらない (ただし import が 1 つ増えているので注意)。

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

public class R12Sample7 extends JFrame {
  Animation hit = null;
  Animation[] a = new Animation[20];
  int count = 0;
  JPanel panel = new JPanel() {
    public void paint(Graphics g) {
      g.setColor(getBackground()); g.fillRect(0, 0, getWidth(), getHeight());
      for(int i = 0; i < count; ++i) { a[i].draw(g); }
    }
  };
```

コンストラクタでの初期化時に、panel に対してイベント受け取りのアダプタを設定する。ここでは短くするため MouseAdapter のサブクラスとなる無名内部クラスとして用意し、mousePressed() のみ差し替えている。そこではマウスの XY 座標を取得してメソッド press() を呼ぶ。あとはこれまでと変わらない。

```
  public R12Sample7() {
    setDefaultCloseOperation(EXIT_ON_CLOSE); setSize(400, 400);
    getContentPane().add(panel);
    panel.addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent e) { press(e.getX(), e.getY()); }
    });
    a[count++] = new FlyingCircle(Color.red, 100, 100, 40, 25, 40, panel);
    a[count++] = new FlyingCircle(Color.blue, 100, 100, 30, 60, -90, panel);
    a[count++] = new WavingText(Color.green, "Hello", 80, 200, 100, 0.5);
    new Thread() {
      public void run() {
        while(true) {
          try { sleep(20); } catch(Exception ex) { }
          for(int i = 0; i < count; ++i) { a[i].addTime(0.02); }
          panel.repaint();
        }
      }
    }.start();
  }
```

press() では一番後ろ (上) から順に図形を調べて、hit するものがあればそれが一番上になるように配列の格納順序を入れ替える。1 つもなければ、代わりに新しい FlyingCircle オブジェクトを乱数パラメタで作り、それを最後に入れる。hit という変数は、当たったオブジェクトまたは新しく生成したオブジェクトを記憶する (次の例題で使う)。

```
  public void press(int x, int y) {
    hit = null;
    for(int i = count-1; i >= 0; --i) {
      if(a[i].hit(x, y)) {
        hit = a[i];
        for(int j = i; j < count-1; ++j) { a[j] = a[j+1]; }
        a[count-1] = hit; return;
      }
    }
    if(count+1 >= a.length) { return; }
    hit = a[count++] = new FlyingCircle(
      Color.getHSBColor((float)Math.random(), 1f, 1f), x, y,
      10+20*Math.random(), 10+40*Math.random(), 10+40*Math.random(), panel);
  }
```

Animation インタフェース以下は hit() を追加した点以外は同じ。

```
  interface Animation {
    public void draw(Graphics g);
    public void addTime(double dt);
    public boolean hit(double x, double y);
  }
  static class WavingText implements Animation {
    static Font fn = new Font("Helvetica", Font.BOLD, 20);
    Color col;
    String text;
    double xpos, ypos, yrad, theta = 0.0, vtheta;
    public WavingText(Color c, String s,
                      double x, double y, double r, double v) {
      col = c; text = s; xpos = x; ypos = y; yrad = r; vtheta = v;
    }
    public void draw(Graphics g) {
      int x = (int)xpos, y = (int)(ypos + yrad*Math.sin(theta));
      g.setColor(col); g.setFont(fn); g.drawString(text, x, y);
    }
    public void addTime(double dt) { theta += vtheta*dt; }
    public boolean hit(double x, double y) {
      int xp = (int)xpos, yp = (int)(ypos + yrad*Math.sin(theta));
      return xp < x && x < xp+15*text.length() && yp-20 < y && y < yp;
    }
  }
  static class FlyingCircle implements Animation {
    Color col;
    double xpos, ypos, rad, vx, vy;
    JPanel panel;
    public FlyingCircle(Color c, double x, double y, double r,
                        double vx1, double vy1, JPanel p) {
      col = c; xpos = x; ypos = y; rad = r; vx = vx1; vy = vy1; panel = p;
    }
    public void draw(Graphics g) {
      g.setColor(col);
      g.fillOval((int)(xpos-rad), (int)(ypos-rad), (int)rad*2, (int)rad*2);
    }
    public void addTime(double dt) {
      xpos += vx * dt; ypos += vy * dt;
      if(xpos<0 && vx<0 || xpos>panel.getWidth() && vx>0) { vx = -vx; }
      if(ypos<0 && vy<0 || ypos>panel.getHeight() && vy>0) { vy = -vy; }
    }
    public boolean hit(double x, double y) {
      return (xpos-x)*(xpos-x) + (ypos-y)*(ypos-y) <= rad*rad;
    }
  }
  public static void main(String[] args) {new R12Sample7().setVisible(true);}
}
```

演習 6 この例題 (コピーできるようにした) をそのまま動かせ。動いたらマウスボタンを押した時の動作を次のように改造してみよ。

a. クリックするとその図形が最前面に出るかわりに一番後ろに行くようにする。

b. 図形以外のところでクリックすると最後にクリックした図形がその場所にワープする。(ヒント: これを実現するには、Animation インタフェースに setPosition(x, y) のようなメソッドを追加し、各図形クラスもそれを追加実装する必要がある。)

c. インタフェース MouseListener とアダプタクラスの土台 MouseAdapter を API ドキュメントで確認し、マウスボタン押し以外のイベントとして何が受け取れるか調べよ。それに基づき、ボタン押し以外の動作でも何か画面に変化が起きるように直してみよ。

d. マウスの移動やドラグを受け取るには、JPanel の addMouseMortionListener() メソッドによってインタフェース MouseMortionListener に従うオブジェクトを登録する (アダプタクラスの土台として MouseMortionAdapter が利用できる)。これらを活用して、図形をドラグできるようにしてみよ。(これも上記 setPosition() がやはり必要になることに注意。)

2.4 GUI 部品とその配置

前節までの方法で確かにマウスボタンの情報は受け取れるようになるが、それで **GUI 部品** (ボタンとかスライドレバーとかメニューとか) を作るのは結構大変そうである。そしてもちろん、Java ではこれらの GUI 部品は既にクラスとして用意してあって、そのインスタンスを生成して画面に配置するだけで動作してくれる。

ところで画面への配置方法はどうすればいいだろう。XY 座標と幅と高さを指定する? それでも「窓の大きさが変わらないならば」いいが、窓の大きさはもちろんユーザが自由に変えてしまえる。そのとき、間抜けな空白ができたり必要なボタンが画面から無くなってしまっては困るので、部品の配置を調整する機能も必要である。Java ではこのような機能を実現するオブジェクトを**レイアウトマネージャ**と呼ぶ。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` 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 8)。

- 外側の窓の中央にはこれまで通りアニメーション用 `JPanel` を配置する。
- 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。

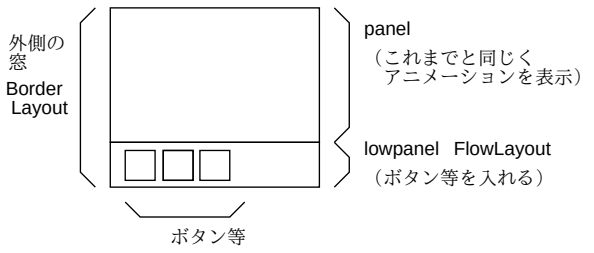

図 8: GUI 部品の例題のレイアウト

さて、次に GUI 部品の動作をプログラムからどうやって使うかを考えてみよう。たとえばテキスト入力欄 (`JTextField`) については、`getText()` でその中に入っている (ユーザが入力した) 文字列が取り出せるというだけで、特に変わったことはない。一方、押しボタン (`JButton`) については、「ボタンが押された」というイベントを受け取って処理する必要がある。このためには、先のマウスイベントと同様だが名前が全部ちがっていて、メソッド `addActionListener()` によりイベントハンドラを設定し、そのイベントハンドラは `ActionListener` インタフェースに従い、ボタンが押されると `actionPerformed()` というメソッドが呼び出される形で用意する。メソッドはこれ 1 つだけなので、土台となるアダプタクラスは不要である。

では実際に例題を見てみよう。冒頭部分はこれまでと同様だが、GUI 部品として押しボタンを 3 つ、テキスト入力欄を 1 つ用意、また部品を入れる `JPanel` を 1 つ用意してこれらも変数に保持する。

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

public class R12Sample8 extends JFrame {
  Animation hit = null;
  Animation[] a = new Animation[20];
  int count = 0;
  JPanel panel = new JPanel() {
    public void paint(Graphics g) {
      g.setColor(getBackground()); g.fillRect(0, 0, getWidth(), getHeight());
      for(int i = 0; i < count; ++i) { a[i].draw(g); }
    }
  };
  JButton b1 = new JButton("Fast");
  JButton b2 = new JButton("Slow");
  JButton b3 = new JButton("Text");
  JTextField t1 = new JTextField();
  JPanel lowpanel = new JPanel();
```

さて、コンストラクタによる初期化であるが、まず窓の中には「中心に」panel、「南 (下端) に」lowpanel を入れる。lowpanel のレイアウトマネージャは FlowLayout に変更する。その後、lowpanel の中に部品を 1 つずつ詰めていくが、ボタンについてはそのボタンが押された時の動作を指定したイベントハンドラを設定していく。

```
  public R12Sample8() {
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    getContentPane().add(panel, BorderLayout.CENTER);
    getContentPane().add(lowpanel, BorderLayout.SOUTH);
    lowpanel.setLayout(new FlowLayout());
    lowpanel.add(b1);
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        if(hit != null) { hit.changeSpeed(1.1); }
      }
    });
    lowpanel.add(b2);
    b2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        if(hit != null) { hit.changeSpeed(0.9); }
      }
    });
    lowpanel.add(b3);
    b3.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        String s = t1.getText();
        if(count+1>=a.length || s.equals("")) { return; }
        a[count++] = new WavingText(
          Color.getHSBColor((float)Math.random(), 1f, 1f), s,
          400*Math.random(), 200*Math.random(), 100*Math.random(), 1);
      }
    });
```

最初の 2 つのボタンは現在選ばれている図形の速度を変更する (このためにインタフェースに changeSpeed() の定義を追加し、各図形クラスでもメソッドを増やしている)。3 番目のボタンは入力欄に打ち込まれた文字列を持った WavingText を生成して追加する。JTextField については、イベントハンドラは必要ないが、どれくらいの大きさにするかを setPreferredSize() で指定する必要がある (何文字くらい打ち込むのに使うかで適正な大きさが違ってくるから)。それ以後はこれまでの例題と同様である。

```
    lowpanel.add(t1); t1.setPreferredSize(new Dimension(60, 25));
    pack(); setSize(400, 400);
    panel.addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent e) { press(e.getX(), e.getY()); }
    });
    a[count++] = new FlyingCircle(Color.red, 100, 100, 40, 25, 40, panel);
    a[count++] = new FlyingCircle(Color.blue, 100, 100, 30, 60, -90, panel);
    a[count++] = new WavingText(Color.green, "Hello", 80, 200, 100, 0.5);
    new Thread() {
      public void run() {
        while(true) {
          try { sleep(20); } catch(Exception ex) { }
          for(int i = 0; i < count; ++i) { a[i].addTime(0.02); }
          panel.repaint();
        }
      }
    }.start();
  }
```

ここ以下はこれまでとほぼ同じである。

```
  public void press(int x, int y) {
    hit = null;
    for(int i = count-1; i >= 0; --i) {
      if(a[i].hit(x, y)) {
        hit = a[i];
        for(int j = i; j < count-1; ++j) { a[j] = a[j+1]; }
        a[count-1] = hit; return;
      }
    }
    if(count+1 >= a.length) { return; }
    hit = a[count++] = new FlyingCircle(
```

```
      Color.getHSBColor((float)Math.random(), 1f, 1f), x, y,
      10+20*Math.random(), 10+40*Math.random(), 10+40*Math.random(), panel);
  }
  interface Animation {
    public void draw(Graphics g);
    public void addTime(double dt);
    public boolean hit(double x, double y);
    public void changeSpeed(double r);
  }
  static class WavingText implements Animation {
    static Font fn = new Font("Helvetica", Font.BOLD, 20);
    Color col; String text;
    double xpos, ypos, yrad, theta = 0.0, vtheta;
    public WavingText(Color c, String s,
                      double x, double y, double r, double v) {
      col = c; text = s; xpos = x; ypos = y; yrad = r; vtheta = v;
    }
    public void draw(Graphics g) {
      int x = (int)xpos, y = (int)(ypos + yrad*Math.sin(theta));
      g.setColor(col); g.setFont(fn); g.drawString(text, x, y);
      g.setColor(Color.black);
    }
    public void addTime(double dt) { theta += vtheta*dt; }
    public boolean hit(double x, double y) {
      int xp = (int)xpos, yp = (int)(ypos + yrad*Math.sin(theta));
      return xp < x && x < xp+15*text.length() && yp-20 < y && y < yp;
    }
    public void changeSpeed(double r) { vtheta *= r; }
  }
  static class FlyingCircle implements Animation {
    Color col; double xpos, ypos, rad, vx, vy; JPanel panel;
    public FlyingCircle(Color c, double x, double y, double r,
                        double vx1, double vy1, JPanel p) {
      col = c; xpos = x; ypos = y; rad = r; vx = vx1; vy = vy1; panel = p;
    }
    public void draw(Graphics g) {
      g.setColor(col);
      g.fillOval((int)(xpos-rad), (int)(ypos-rad), (int)rad*2, (int)rad*2);
    }
    public void addTime(double dt) {
      xpos += vx * dt; ypos += vy * dt;
      if(xpos<0 && vx<0 || xpos>panel.getWidth() && vx>0) { vx = -vx; }
      if(ypos<0 && vy<0 || ypos>panel.getHeight() && vy>0) { vy = -vy; }
    }
    public boolean hit(double x, double y) {
      return (xpos-x)*(xpos-x) + (ypos-y)*(ypos-y) <= rad*rad;
    }
    public void changeSpeed(double r) { vx *= r; vy *= r; }
  }
  public static void main(String[] args) {new R12Sample8().setVisible(true);}
}
```

演習 7 この例題 (コピーできるようにした) をそのまま動かせ。動いたら好きなように改造してみよ。

3 さいごに

これで「情報科学」久野クラスの講義はすべておしまいです。あとは冬休み課題のレポートよろしくお願いします。最後にひとこと。皆様は理 2・3 類なのでソフトウェア系の専門に進まれるかたはほとんどいないと思いますが、たとえそうだとしても、情報科学の基本部分を見聞し、また実際にプログラミングができるようになったことには多くの価値があると思います。たとえば、できあいのソフトではやりにくいことでも、プログラムを書けばすぐにできたり楽に仕事ができることは多数あります。皆様が今後ともそういう形で、情報科学やプログラミングの知識を活用して行って頂けるようになれば、私としては大変幸せです。では半年間、ありがとうございました。

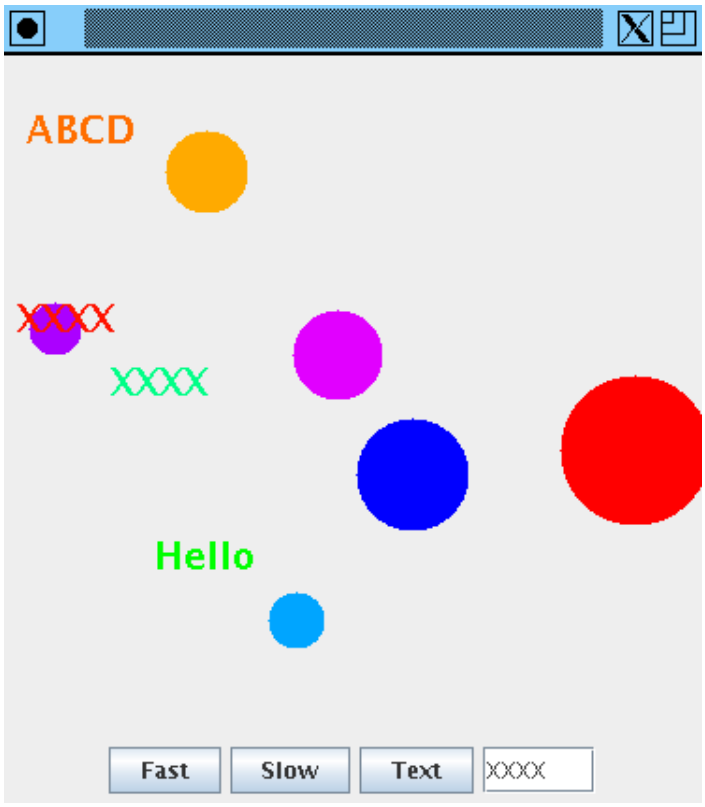

図 9: GUI 部品とアニメーションを持った例題

A　本日の課題 13A

講義は 12 回目ですが前回休講時も当日レポートを出したので今回を 13A としています。

「演習 2」「演習 3」のいずれか 1 つについて、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。演習 4 以降のはどう見ても巨大になるので遠慮します (もちろん趣味として各自やっていただくのは歓迎します)。

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. 今回の内容についていかがでしたか。

   Q2. 半年間のこの講義の総括を自由に書いてください。

B　おまけのお願い 14A

お聞き及びと思いますが、「情報科学」の他の多くのクラスでは共通の筆記試験 (プログラミング言語は Ruby) を実施します。私としては、皆様の実力が他のクラスに遜色ないくらいついたことを確認したいので、共通試験実施後に (実施翌日の 2/16 に)、言語を Ruby から Java に直したバージョンを本クラスのページ以下に用意したいと思います。ご協力頂ける方は、この問題を「資料や参考書等を見ないで、90 分以内で」解答し、14A として私あて送ってください (Subject: を Report14A とする以外は書式自由、感想も書いてください)。なお、これはあくまでも調査のための参考データとして用いるだけであり、成績には関係しません。ただし、皆様にとっても腕試しとして面白いのではないかと思います。