オブジェクト指向プログラミング'10 ＃2

久野　靖*

2010.9.14

前回は「既にあるクラスを API ドキュメントで使い方を調べて利用する」という部分を扱いましたが、今回からは自分でクラスを作ってもらいます。これによって、オブジェクト指向の利点を具体的に感じて頂けるものと思います。

1　図形のクラス定義

今回の最初の例題は「円を 2 つ描く」でして、描く絵のレベルは前回までの絵とまったく変わりません。ただし、プログラムの構造は大幅に (後で拡張しやすいように) 変わっています。

このように、「プログラムの機能は変えないで構造を改良する」ことを**リファクタリング**と呼びます (図 1)。[1)] プログラムに機能を追加し終わった時には、リファクタリングを行っておくことで、さらなる機能の拡張がスムーズに行えるようになるのです。

1)プログラムの機能と構造を同時に変更すると、問題が発生したときに追加した機能の問題なのか変更した構造の問題なのかが分からずに難儀しやすいので、構造を改良するときは機能を変更しないまま行うのがよいのです。

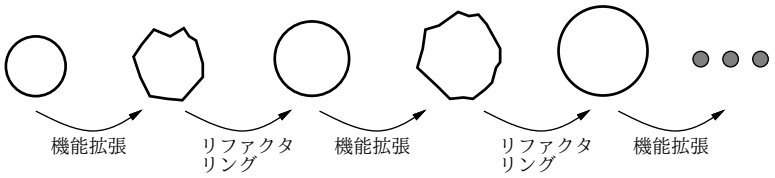


図 1: リファクタリング

例題: クラスを使って円を 2 つ描く

ではさっそく、構造を改良した「円を 2 つ描く」プログラムを見て頂きましょう。クラス `Circle` は名前どおり円を現すクラスであり、「色、中心の X 座標、Y 座標、半径」を指定して作成します。そして、`Graphics` オブジェクトを指定したメソッド `draw()` によって、その `Graphics` オブジェクトに対応する画面に円を描きます。

プログラム本体では、2 つの `Circle` オブジェクトを生成して変数 `c1`、`c2` に入れておき、`paintComponent()` の中でこれらの円を順次 `draw()` で表示するだけです。`main()` はこれまでと同じですが、あとの部分は非常に簡潔になったと言えるでしょう。

*経営システム科学専攻

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

public class Sample41 extends JPanel {
  Circle c1 = new Circle(Color.MAGENTA, 100, 50, 30);
  Circle c2 = new Circle(Color.BLUE, 190, 90, 40);

  public void paintComponent(Graphics g) {
    c1.draw(g); c2.draw(g);
  }
  public static void main(String[] args) {
    JFrame app = new JFrame();
    app.add(new Sample41());
    app.setSize(400, 300);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
  }
}
```

そしてもちろん、クラス Circle は自分で用意します。[2] 見ての通り、Circle は色と XY 座標と半径をインスタンス変数として持ち、コンストラクタでこれらを初期化し、draw() では渡された Graphics オブジェクに対してこれまでと同様に fillOval() を呼んで円を描いています。

2)一般的なクラス記述の構成や書き方については、前回に説明したので、読み返してみてください。

```
class Circle {
  Color col;
  int xpos, ypos, rad;
  public Circle(Color c, int x, int y, int r) {
    col = c; xpos = x; ypos = y; rad = r;
  }
  public void draw(Graphics g) {
    g.setColor(col);
    g.fillOval(xpos-rad, ypos-rad, rad*2, rad*2);
  }
}
```

実際にプログラムを打ち込むときは、これら全体を 1 つのファイルに入れてコンパイルしてください。[3] 動かしたようすを図 2 に示します。

3)main() を含んだクラスについては public と指定し、ファイル名をクラス名に合わせる必要がありましたが、Circle のような下請けクラス (public をつけません) は、いくつでも一緒にファイルに入れておくことができます。この状態でコンパイルすると、.class ファイルやクラスごとに別のものが生成されます。

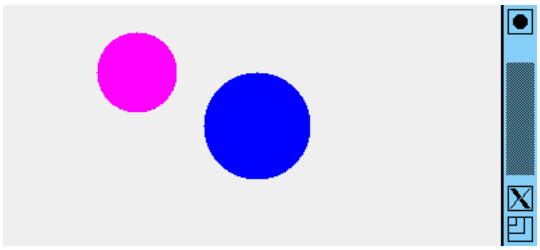


図 2: 2 つの円

それで、プログラムの構造はどのように改善されたでしょうか? Circle というクラスを独立させて作ったことで、本体部分が簡潔になり、また本体と本体が使う部品のクラス Circle を分けて考えられるので、プログラムを検討するときに一時に見なければならない範囲が小さくなりました。[3]

また、インスタンス変数は原則としてそのクラスのメソッドからだけアクセスされるので、それらの変数を書き換える箇所が限定でき、その範囲だけをよく検討することで間違いのないプログラムが作りやすくなります。このように、内部のデータ構造を**カプセル化**して外部からの干渉を排除することを**情報隠蔽**と呼び、オブジェクト指向の重要な特徴となっています。[5]

演習 4-1 例題 Sample41.java をそのまま打ち込んで動かしなさい (クラス Circle も一緒のファイルに打ち込んでかまいません)。動いたら、次のように手直ししてみなさい。

a. 円の色や位置や大きさを変更してみなさい。たとえば、半透明な色を持つ円にしてみる、同心円状に配置するなど。
b. 円の数を 3 つ、4 つに増やしてみなさい。
c. 「長方形」を現すクラス Rect を追加し、円と一緒に表示させてみなさい。作成時のパラメタとしては「色、中心の XY 座標、幅、高さ」を指定するものとします。
d. 「三角形」を現すクラス Triangle を追加し、円と一緒に表示させてみなさい。作成時のパラメタとしては「色と 3 頂点の XY 座標」を指定するものとします。

例解 4-1-cd

これらの問題はいずれも、(1) 新しいクラスを作り、(2) それらのインスタンスを変数に保持し、(3)paintComponent() の中でそれらのオブジェクトを描く、という形で実現できます。長方形も三角形も作成時に渡したパラメタをそのまま保持し、描画するときに適宜必要な形に変換しています。[6]

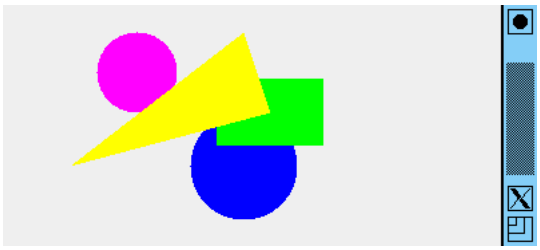
```
import java.awt.*;
import javax.swing.*;

public class ex41cd extends JPanel {
  Circle c1 = new Circle(Color.MAGENTA, 100, 50, 30);
  Circle c2 = new Circle(Color.BLUE, 180, 120, 40);
  Rect r1 = new Rect(Color.GREEN, 200, 80, 80, 50);
  Triangle t1 = new Triangle(Color.YELLOW,
    50, 120, 200, 80, 180, 20);

  public void paintComponent(Graphics g) {
    c1.draw(g); c2.draw(g); r1.draw(g); t1.draw(g);
  }
  // (main 省略)
}
// (クラス Circle 省略)
class Rect {
  Color col;
  int xpos, ypos, width, height;
  public Rect(Color c, int x, int y, int w, int h) {
    col = c; xpos = x; ypos = y; width = w; height = h;
  }
```

4)一般に、プログラムは「大きさが 2 倍になると作成する手間は 4 倍になる」とも言われます。それは、作成している部分ごとの相互作用が複雑になり、その把握や調整に手間が掛かるからです。これに対し、そのプログラムをうまく 2 つの部分に分けられれば、もとの大きさのプログラムを 2 つ作るのと同程度の手間、つまり 2 倍程度の手間で済ませられるわけです。上の例でプログラムをクラス Circle とそれ以外の部分に分けたことは、そのような効果をもたらすわけです。

5)より正確には、**抽象データ型** (Abstract Data Types、ADT) と呼ばれる機能の性質ですが。オブジェクト指向言語のクラスは抽象データ型の機能を併せ持っているわけです。

6)三角形の方は、描画するときに用いるメソッド fillPolygon() が X 座標の配列と Y 座標の配列を受け取るようにできているため、インスタンス変数でも X 座標の配列と Y 座標の配列を保持するようにしています。コンストラクタの中で配列生成式「new int[]{…}」を使って初期値を持つ配列を作っています。

```
  public void draw(Graphics g) {
    g.setColor(col);
    g.fillRect(xpos-width/2, ypos-height/2, width, height);
  }
}
class Triangle {
  Color col;
  int[] xs, ys;
  public Triangle(Color c, int x0, int y0, int x1, int y1,
                  int x2, int y2) {
    col = c; xs = new int[]{x0,x1,x2}; ys = new int[]{y0,y1,y2};
  }
  public void draw(Graphics g) {
    g.setColor(col); g.fillPolygon(xs, ys, 3);
  }
}
```



2　複合図形を組み立てる

クラスを作ることでそれが部品になる、という話をしましたが、それを実感できる例題として「部品どうしを組み合わせた図形を作る」ことを考えましょう。たとえば、図 4 のような図柄の旗を作るのに、7)「長方形」という部品と「円」という部品を組み合わせて作るわけです。

7)この図柄のような旗が国旗になっている国もいくつかありますね。

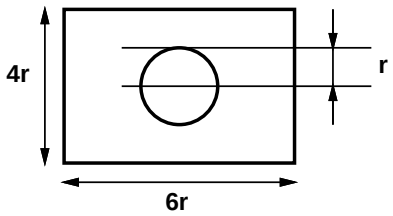


図 3: 旗の図柄の設計図

例題: 旗のクラスを定義する

では、上で設計した旗をクラスとして定義して描くという例題を見ていただきます。まず、クラスができたものとして、それを使う部分はこれまでとほとんど変わりません。8)

8)Flag のコンストラクタは「new Flag(地の色, 丸の色, 丸の半径, X 座標, Y 座標)」という呼び出し方にするものとしました。

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

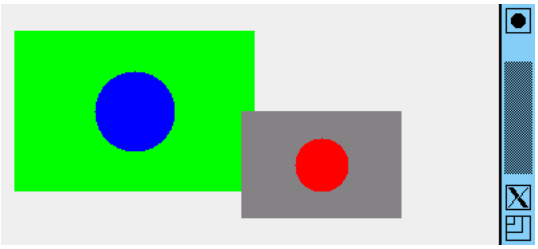


図 4: 2 つの旗

```
public class Sample42 extends JPanel {
  Flag f1 = new Flag(Color.GREEN, Color.BLUE, 30, 100, 80);
  Flag f2 = new Flag(Color.GRAY, Color.RED, 20, 240, 120);

  public void paintComponent(Graphics g) {
    f1.draw(g); f2.draw(g);
  }
  public static void main(String[] args) {
    JFrame app = new JFrame();
    app.add(new Sample42());
    app.setSize(400, 300);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
  }
}
```

では、肝心の旗のクラスを見ていただきましょう。

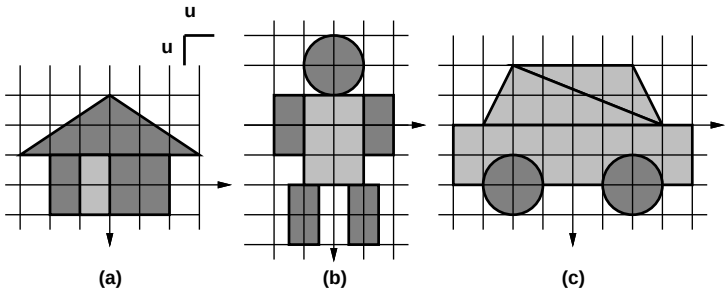
```
class Flag {
  Rect r1;
  Circle c1;
  public Flag(Color cl1, Color cl2, int r, int x, int y) {
    r1 = new Rect(cl1, x, y, r*6, r*4);
    c1 = new Circle(cl2, x, y, r);
  }
  public void draw(Graphics g) {
    r1.draw(g); c1.draw(g);
  }
}
```

これだけ! です。つまり、旗というのは長方形と円から成っているので、その 2 つのオブジェクトをコンストラクタで生成してインスタンス変数として保持し、表示時もこれらを表示すればいいわけです。[9]そして、長方形と円のクラスは先に作成した通りです。このように、既に作ったクラスは部品として、それらを組み合わせて複合図形を作ることで、次々に複雑な絵を作ることができます。

[9]描く時には、まず長方形を描いてから円を描くという順序が大切です。後から描いた絵が上書きになるので、長方形を後にすると円が消されてしまいます。

演習 4-2 既に作った長方形、三角形、円のクラスを利用して下の図のような複合図形のクラスを作成しなさい。大きさは図の u の長さを指定し、

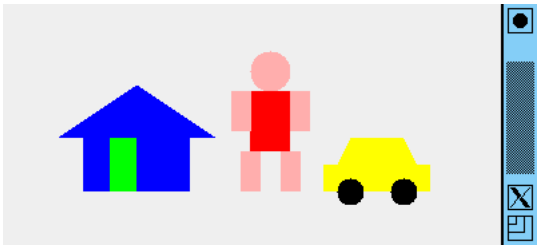
色の塗り分け方は適宜判断してください。完成したら、さらに自分で設計した独自の図形も作ってみるとよいでしょう。



例解 4-2-abc

3 つの絵のどれも、設計図の通りに既に作った円、長方形、三角形の各オブジェクトを組み合わせればできます。紙面の都合により、「家」のクラスまでだけ示します。

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

public class ex42abc extends JPanel {
  House h1 = new House(Color.BLUE, Color.GREEN, 20, 100, 120);
  Human m1 = new Human(Color.PINK, Color.RED, 15, 200, 80);
  Car c1 = new Car(Color.YELLOW, Color.BLACK, 10, 280, 120);

  public void paintComponent(Graphics g) {
    h1.draw(g); m1.draw(g); c1.draw(g);
  }
  public static void main(String[] args) {
    JFrame app = new JFrame();
    app.add(new ex42abc());
    app.setSize(400, 300);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
  }
}
class House {
  Rect r1, r2;
  Triangle t1;
  public House(Color cl1, Color cl2, int u, int x, int y) {
    r1 = new Rect(cl1, x, y, u*4, u*2);
    r2 = new Rect(cl2, x-u/2, y, u, u*2);
    t1 = new Triangle(cl1, x, y-u*3, x-u*3, y-u, x+u*3, y-u);
  }
  public void draw(Graphics g) {
    r1.draw(g); r2.draw(g); t1.draw(g);
  }
}
(以下略)
```



3　マウスで図形を移動させる

ぶじにさまざまな図形がオブジェクトになったところで、マウスを使って図形をドラグして移動させてみましょう。[10)]

まずそのために、入力の取り扱い方から説明しましょう。マウスボタンが押される、マウスが移動する、キーボードのキーが押されるなどのことがらを一般に**入力イベント**と呼びます。[11)] そして、イベントが発生した「時点で」それを受け取って処理するコード (**イベンドハンドラ**) を予め用意しておき、このコードが実際に入力を処理します。

マウスやキーボードのイベントは「窓」単位で (つまりある窓が選択された状態でマウスやキーが操作されると) 発生するので、イベントの受け取りも窓単位で行います。これには、具体的には次のようにします (図 5)。

- イベントを受け取るための**アダプタオブジェクト**を用意する。このオブジェクトは、イベントの種類ごとに特定のインタフェースを実装している必要がある。
- アダプタオブジェクトを、イベントが発生するオブジェクト (今の場合は窓オブジェクト) に対して登録する。[12)]
- 実際にイベントが発生すると、そのイベントに対応してアダプタオブジェクトの決まったメソッド (上述のインタフェース定めているもの) が呼び出されるので、そこで処理を記述する。

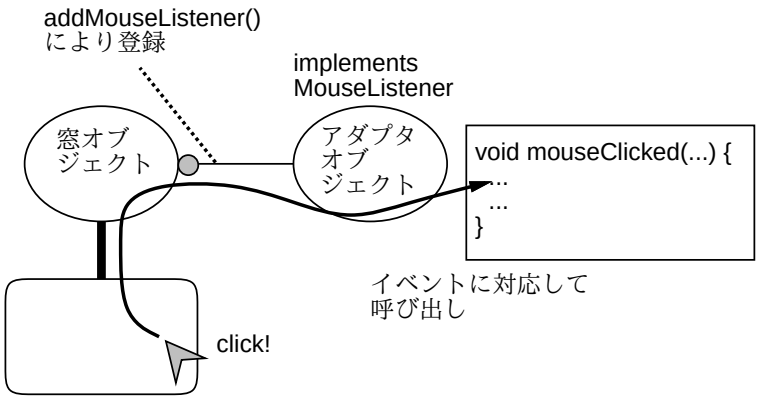


図 5: 入力イベントの受け取り

具体的にインタフェースとそれが規定しているメソッドについて説明しましょう。[13)]マウスの場合、「動きを伴わないイベント」と「伴うイベント」でインタフェースが分かれています。キーボードでは 1 つです。

10)これが、はじめての入力を使ったプログラムということになりますね!

11)イベントという用語は、プログラムの実行とは独立したタイミングで (つまりユーザが操作を行った時点で) これらのことがらが発生する、ということから来ています。

12)ここでのプログラムは主となるクラスが `JFrame` のサブクラス、つまり窓のクラスですから、そのインスタンスメソッド中で単に `addMouseLisner()` 等を呼び出せば、自分自身つまり窓に対する登録がおこなえます。この点については付録の説明も見てください。

13)以下のインタフェースやクラスはすべて、パッケージ `java.awt.event` の中で定義されているので、使う時にはファイルの先頭に「`import java.awt.event.*;`」の指定を入れます。

- MouseListener — 動きを伴わないマウスイベントのインタフェース
  - mousePressed(MouseEvent e) — マウスボタンが押された
  - mouseReleased(MouseEvent e) — マウスボタンが離された
  - mouseClicked(MouseEvent e) — ボタンがクリックされた
  - mouseEntered(MouseEvent e) — マウスが窓領域に入った
  - mouseExited(MouseEvent e) — マウスが窓領域から出た
- MouseMotionListener — 動きを伴うマウスイベントのインタフェース
  - mosuseMoved(MouseEvent e) — マウスが移動した
  - mouseDragged(MouseEvent e) — マウスがドラグされた
- KeyListener — キーボードイベントのインタフェース
  - keyPressed(KeyEvent e) — キーが押された
  - keyReleased(KeyEvent e) — キーがが離された
  - keyTyped(KeyEvent e) — キーが打鍵された

これらの中から受け取りたいイベントに応じて適切なインタフェースを選び、それを実装したアダプタオブジェクトを作って窓に対して登録します。[14][15] そうすれば、イベントが発生した時に対応するメソッドが呼び出されるので、その中で処理を行います。このとき、引数として渡されて来るイベントオブジェクト (MouseEvent、KeyEvent のインスタンス) には、イベントに関する情報 (マウスの XY 座標や押されたキーの情報) が含まれていて、それらを取り出して使うことができます。

上記 3 つのインタフェースはいずれも複数のメソッドを規定していますが、実際にはその一部だけを使いたいのが普通で、その時に他のメソッドすべて用意するのは面倒です。そこで、全部のイベントに何もしないメソッドを定義したアダプタクラス MouseAdapter、MouseMotionAdapter、KeyAdapter が用意されています。自分が定義するアダプタクラスはこれらのサブクラスにして、使いたいメソッドだけオーバライドすればよいのです。たとえば自分が扱いたいのがマウスドラグなら、次のようにするわけです。

```
class MyAdapter1 extends MouseMotionListener {
  public void mouseDragged(MouseEvent e) {
    ... ここにドラグに対応する動作を書く ...
  }
}
```

14)登録にはそれぞれメソッド addMouseListener()、addMouseMotionListener()、addKeyListener() を用います。

15)さらに、キーイベントを取るためには、どれかのマウスイベントの処理の中で窓のメソッド requestFocus() を呼び出してください。このメソッドを呼ぶことで、窓が「キー入力を受け取る」モードになります。

例題 4-3: 円をドラグする

ではいよいよ、円をドラグする例題を見てみましょう (図 4[16])。まず本体部分から。

16)ただし印刷ではドラグしているかどうかは分かりませんね。

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

public class Sample42 extends JPanel {
```

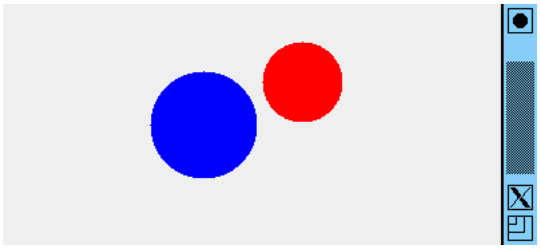


図 6: 円をドラグ

```
  Circle c1 = new Circle(Color.RED, 100, 50, 30);
  Circle c2 = new Circle(Color.BLUE, 150, 90, 40);

  public Sample42() {
    setOpaque(false);
    addMouseMotionListener(new MyAdapter1());
  }
  public void paintComponent(Graphics g) {
    c1.draw(g); c2.draw(g);
  }
  public static void main(String[] args) {
    JFrame app = new JFrame();
    app.add(new Sample42());
    app.setSize(400, 300);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
  }
  class MyAdapter1 extends MouseMotionAdapter {
    public void mouseDragged(MouseEvent evt) {
      c1.moveTo(evt.getX(), evt.getY()); repaint();
    }
  }
}
```

クラス Sample42 はコンストラクタを持っていますが、これは (1) 画面が変化するモードに設定する、[17] (2) アダプタを登録する、という 2 つの初期設定が必要だからです。そして、クラス MyAdapter1 がクラス Sample42 の「中に」書かれていますが、これを**内部クラス**と呼びます。[18] アダプタの中では、マウスの座標を取得してきて、円 c2 の位置を動かしています。さらにその後 repaint() というメソッドを呼び出しますが、これは窓に対して画面を描き直すことを依頼しています。クラス Circle の方も見てみましょう。

```
class Circle {
  Color col;
  int xpos, ypos, rad;
  public Circle(Color c, int x, int y, int r) {
    col = c; xpos = x; ypos = y; rad = r;
  }
```

17)正確に言うと、setOpaque(false) を呼ぶことで「このパネル部分は毎回全部を描き直すことはしない」ことを上位の要素に通知します。そうすると、上位では領域をクリアしてから paintComponent() を呼んでくれます。これを呼ばないと「前の画面の残像」が残ったまま次々に描かれる」状態になってしまいます。

18)内部クラスの説明は付録にありますが、このようにすることで Sample42 のインスタンス変数である c2 をアクセスできるようになります。アダプタの中で円 c2 を動かすためにこれにアクセスする必要があるわけです。

```
  public void moveTo(int x, int y) {
    xpos = x; ypos = y;
  }
  public void draw(Graphics g) {
    g.setColor(col);
    g.fillOval(xpos-rad, ypos-rad, rad*2, rad*2);
  }
}
```

これまでとほとんど変わらず、ただし位置の変更が必要になったのでメソッド `moveTo()` が追加されています。[19] このように、円をオブジェクトとしておくことで、本体側もオブジェクトを定義するクラス側も素直に書くことができるわけです。

19)その中身は渡された XY 座標を円の XY 座標として設定しているだけです。

無名内部クラス

先の例ではアダプタクラス `MyAdapter1` を (本体クラスの内側ではありましたが) 普通のクラスとして書き、`addMouseMotionListener()` の呼び出しのところでそのインスタンスを生成していましたが、これを 1 箇所にまとめて書くことができます。具体的には、Java では次のような書き換えができます。この書き方を**無名内部クラス**と呼びます。[20]

20)よく見ると分かるように、`YYY` というのは親クラスまたは実装するインタフェースの名前であり、書き換え前のクラスの名前 `XXX` は書き換え後は無くなっています。つまり無名のクラスなわけです。

- 内部クラス `XXX` が次の形をしている:

  ```
  class XXX implements/extends YYY {
    ... クラス本体 ...
  }
  ```

- その内部クラスのインスタンスを生成する箇所が 1 箇所だけであり、そこが次 の形をしている (コンストラクタが引数を持たない):

  ☆☆☆ `new XXX()` ☆☆☆

- 上の 2 条件が成り立つ場合、この 2 箇所をまとめて、インスタンスを生成する箇所を次のように書くことができる:

  ```
  ☆☆☆ new YYY() {
    ... クラス本体 ...
  } ☆☆☆
  ```

先の例題 4-2 を無名内部クラスに書き換えるには、コンストラクタを次のようにして、`MyAdapter1` の定義は削除すればよいのです。

```
public Sample42() {
  setOpaque(false);
  addMouseMotionListener(new MouseAdapter() {
    public void mouseDragged(MouseEvent evt) {
      c2.moveTo(evt.getX(), evt.getY()); repaint();
    }
  });  //←この「)」は「addMouseMotionListener(」に対応
}
```

ちょっと慣れないと奇異に感じますが、プログラムが短くできるので本書ではこれからこちらを使うことにします。皆様はどちらでも好きな方法で書いて構いません。

演習 4-2 例題 Smaple42.java をそのまま打ち込んで動かしなさい。[21] 動いたら、次のように手直ししてみなさい。

a. 例題では赤い円がドラグされますが、青い円と重なった時にも後ろに隠れたままドラグされていきます。赤い円が手前になるようにしてみなさい。[22]

b. 例題では赤い円は窓の範囲内であればどこまででもドラグできますが、窓の中央より右側には行かないようにしてみなさい。その他、もっとヘンな動き (例: 右にドラグすると左に動くなど) を実現してみなさい。

c. シフトキーを押しながらドラグしたら青い円、押さずにドラグしたら赤い円がドラグされるようにしてみなさい。[23]

d. 円ばかり動かしてもつまらないので、「家」ないし任意の複合図形をドラグできるようにしてみなさい。[24]

e. キーボードから「>」「<」のキーを打つとどれかの図形が大きくなったり小さくなるようにしてみなさい。[25]

21) クラス Circle は前に打ち込んだものを利用してかまいませんが、新たなメソッドを追加するのを忘れないように。

22) 後から描いたものが「手前に」あるように見えます。

23) ヒント: MouseEvent オブジェクトに対してメソッド isShiftDown() を呼ぶことで、シフトキーが押されているかどうかが判定できます。

24) もちろん、そのためにはその複合図形のクラスに位置を変更するメソッド moveTo() を定義する必要があるでしょう。

25) addKeyListener() でキーボードのアダプタも追加し、キー押し下げ時に KeyEvent オブジェクトのメソッド getKeyChar() でキーの文字を取得して判断するのがよいでしょう。なお、キーイベントを取るためにはマウスイベントの処理時に requestFocus() を呼ぶ必要があります。

26) **return** というのは「このメソッドの処理をここで終わる」という命令です。値を返すメソッドの場合は「return 式;」により返す値を指定しますが、ここでは値は返さないでただメソッドの処理を終わるだけでです。

27) さらに、キー入力が取れるように requestFocus() も呼んでいます。

例解 4-2-b

これは簡単で、X 座標がある値以上ならばドラグイベントの処理をしないように直せばよいのです。mouseDragged() だけを示します。[26]

```
public void mouseDragged(MouseEvent evt) {
  if(evt.getX() > 200) { return; }
  c1.moveTo(evt.getX(), evt.getY()); repaint();
}
```

例解 4-2-c

これも簡単で、ヒントにあるように isShiftDown() で枝分かれしてどちらかを動かすようにすればよいわけです。

```
public void mouseDragged(MouseEvent evt) {
  if(evt.isShiftDown()) {
    c2.moveTo(evt.getX(), evt.getY()); repaint();
  } else {
    c1.moveTo(evt.getX(), evt.getY()); repaint();
  }
}
```

例解 4-2-de

これはやや複雑になるので、全体像を示します。ドラグしたり拡大/縮小する図形は「家」ということにして、これを 2 つ保持します (変化させるのは片方)。そして大きさを変化させるということで、絵の「単位」を入れておく変数 unit を用意しました。動かすのは例題と同様ドラグのイベントに対応して moveTo() を呼びます。[27][28]

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


public class ex43de extends JPanel {
  int unit = 20;
  House h1 = new House(Color.BLUE, Color.GREEN, 20, 100, 120);
  House h2 = new House(Color.RED, Color.PINK, 15, 200, 80);

  public ex43de() {
    setOpaque(false);
    addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent evt) {
        requestFocus();
        h1.moveTo(evt.getX(), evt.getY()); repaint();
      }
    });
    addKeyListener(new KeyAdapter() {
      public void keyPressed(KeyEvent evt) {
        if(evt.getKeyChar() == '>') { unit += 5; }
        if(evt.getKeyChar() == '<' && unit > 5) { unit -= 5; }
        h1.resize(unit); repaint();
      }
    });
  }
  public void paintComponent(Graphics g) {
    h1.draw(g); h2.draw(g);
  }
  public static void main(String[] args) {
    JFrame app = new JFrame();
    app.add(new ex43de());
    app.setSize(400, 300);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
  }
}
```

家を動かすメソッド moveTo() と大きさを設定するメソッド resize() があるかのように作りましたが、実際にはこれらを自前で作らなければなりません。ここでは簡単のため、位置や大きさが変わったら家の部品である三角形や長方形を作り直すことにしました。[29] そうすると、最初に作るのも後で作り直すのもやることは同じなので、作る処理を init() というメソッドに分けて、必要なところでこれを呼ぶようにしました。実はこのメソッドの中身はこれまでのコンストラクタと同じです。[30][31]

作り直すときに必要なので、2 つの色と大きさ、XY 座標もインスタンス変数として持つようにしました。こうしておけば、最初に作るときも位置や大きさを変更するときも、これらの変数を変更して init() を呼び出すだけです。Triangle や Rect などのクラスはこれまでに出て来たものと同じです。[32]

```
class House {
  Rect r1, r2;
  Triangle t1;
  Color col1, col2;
  int unit, xpos, ypos;

  public House(Color cl1, Color cl2, int u, int x, int y) {
```

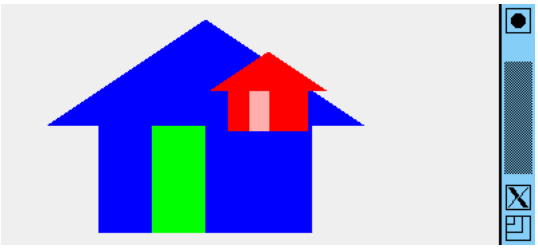
28)大きさを小さくしすぎないように、「文字が'<' であり、なおかつ unit が 5 より大きいなら」小さくする、というふうにしています。「&&」は複合条件を書くときに「かつ」を指定する演算子でしたね。

29)このようにした場合、作り直す前に使っていたオブジェクトはどうなるのか、という疑問があるかと思います。その答えは「使わなくなったオブジェクトは Java 実行形が自動的に回収して領域を再利用してくれます」というものです。この自動回収機能を**ガベージコレクション** (garbage collection、**ごみ集め**) と言います。これがあるおかげで、Java ではオブジェクトを気軽に作って使うことができるのです。

30)メソッド init() はこのクラスの中からだけ呼ぶので **private**(クラス外からは呼べない) と指定してあります。

31)このように、同じ動作を複数の箇所から使う場合に、それをメソッド (手続き) にしておいて使うところから呼び出す、というのは手続き型プログラミングにおいてコードを整える主要な方法です。オブジェクト指向言語ではさらに大きな構造としてオブジェクトを定義して構造を整理しますが、メソッドの中は手続きなので、手続き的な方法も必要に応じて使うわけです。

```
    col1 = cl1; col2 = cl2; unit = u; xpos = x; ypos = y;
    init(col1, col2, unit, xpos, ypos);
  }
  private void init(Color cl1, Color cl2, int u, int x, int y) {
    r1 = new Rect(cl1, x, y, u*4, u*2);
    r2 = new Rect(cl2, x-u/2, y, u, u*2);
    t1 = new Triangle(cl1, x, y-u*3, x-u*3, y-u, x+u*3, y-u);
  }
  public void moveTo(int x, int y) {
    xpos = x; ypos = y; init(col1, col2, unit, xpos, ypos);
  }
  public void resize(int u) {
    unit = u; init(col1, col2, unit, xpos, ypos);
  }
  public void draw(Graphics g) {
    r1.draw(g); r2.draw(g); t1.draw(g);
  }
}
```



32)毎回作り直すので moveTo() を追加する必要もありません。ただ、毎回作り直すと処理が重たくなるので、あまり重いようだったら moveTo() を用意して移動のときはこちらを呼ぶように変更した方がよいでしょう。このような改良もリファクタリングの一種だと言えます。

4　付録: 名前のスコープと入れ子クラス

今回からは 1 つのプログラムに複数のクラスが出て来るようになり、またあるクラスの中に別のクラスを作ることも行いました。ここで、改めてこれらの機能を含んだことがらについて整理しておきます。

スコープの概念

一般にプログラミング言語では、名前 (クラス名、メソッド名、変数名など) には**有効範囲** (**スコープ**) があります。スコープの広い名前は多くの箇所から参照できますが、スコープの狭い名前は小さい範囲からしか参照できません。スコープは図 7 のように、あるスコープの中に別のスコープが「そっくり入った」構造 (**入れ子構造**、nest) を構成しています。そして原則として「内側からは外側にある名前が参照できるが、外側から内側の名前は参照できない (入れ子の外から内には入れない)」という規則が成り立ちます。33)

パッケージのスコープ

以上は一般の話でしたが、Java ではパッケージ、クラス、メソッド、ブロックがスコープの単位になっています (図 8)。最初にパッケージについて説明してしまいましょう。

Java では各ソースファイルの先頭に「package パッケージ名;」という宣言を入れることで、そのファイル中のクラスをどれかのパッケージに所属さ

33)参照できないというのは不便なようですが、このおかげで内部の変数を外部からアクセスされたり壊されたりする心配がなくなり、また内部の変数どうしが干渉する心配がなくなるので、このような規則はとても大切です。なお、参照できないというのはあくまでも「原則」でして、一部の名前については (1)「どの単位の中の」という**接頭語** (プレフィクス) をつけ、(2) アクセス制御で「外から参照可能」と指定することで参照できるようにもなります。

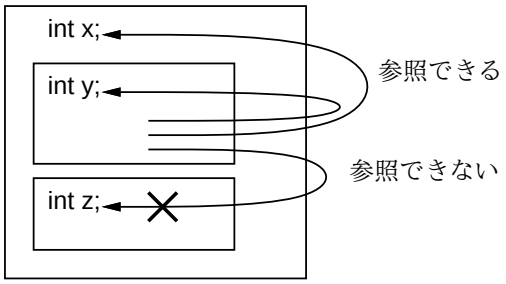


図 7: スコープの入れ子

せることができます。この宣言が無い場合は、そのファイル中のクラスは 1 つだけある「無名の」パッケージに所属します。本書ではこの状態でプログラムを書いているわけです。[34)]

34)図 8 ではあるパッケージのクラスが並んでいるように描いてありますが、複数のファイルに分かれていても、同じパッケージのクラス群はこのように「一緒にまとめられている」ものとして扱うわけです。

あるパッケージのクラスを他のパッケージから使うには、まずそのクラスが public 指定になっている必要があります。その上で、次の 2 つのやり方のどちらかでそのクラスを指定します。

- 「パッケージ名 . クラス名」という書き方で常にパッケージ名を先頭につけて指定する。
- 「import パッケージ名 . クラス名;」という指定をファイル先頭に置き、以後ファイル内ではクラス名だけで指定する。

毎回パッケージ名をつけるのは煩わしいので、そのために import 文を使うわけです。そして、クラス名の変わりに「*」と指定することで「そのパッケージ中にある public 指定のクラス全部を import する」ことができます。これまでライブラリクラスを利用するために「import java.awt.*;」などと書いて来たのは、実はこの機能を使っていたわけです。

クラスのスコープ

パッケージの内側のスコープはクラスのスコープです。クラス内で定義するものとしては、変数 (インスタンス変数とクラス変数)、メソッド (インスタンスメソッドとクラスメソッド)、そしてクラスの内側で定義したクラス (**入れ子クラス**) があります。

あるクラス内からは、スコープの一般規則どおり、そのクラスの変数、メソッド、入れ子クラスが参照できます。そして、これらに public 指定がなされていれば、クラスの外側からでもプレフィクスつきで参照できます。[35)]

35)さらに、同じパッケージ内からであれば、何も指定していない場合でも同様に参照できます。参照できなくしたい場合は private や protected という指定をつけます。

ところでこれらにおいて、static の有無に注意を払う必要があります。static のついた変数やメソッドはクラス変数やクラスメソッドですから、特定のインスタンスには付属していません。このため、クラス内部のどこからでも名前だけで参照できますし、外部からは「クラス名 . 名前」で参照できます。実は入れ子クラスについても static のついたものは同様です。

一方、static のついていない変数、メソッド、入れ子クラス[36)] は特定のインスタンスに付属しています。このため、同じクラスの他の static のついていないメソッドや入れ子クラス内からは名前だけで参照できますが、static のついたメソッドや入れ子クラス、そしてクラスの外部からは付属

36)static のついていない入れ子クラスのことを**内部クラス**と呼びます。無名内部クラスもこの特別な場合ということになります。

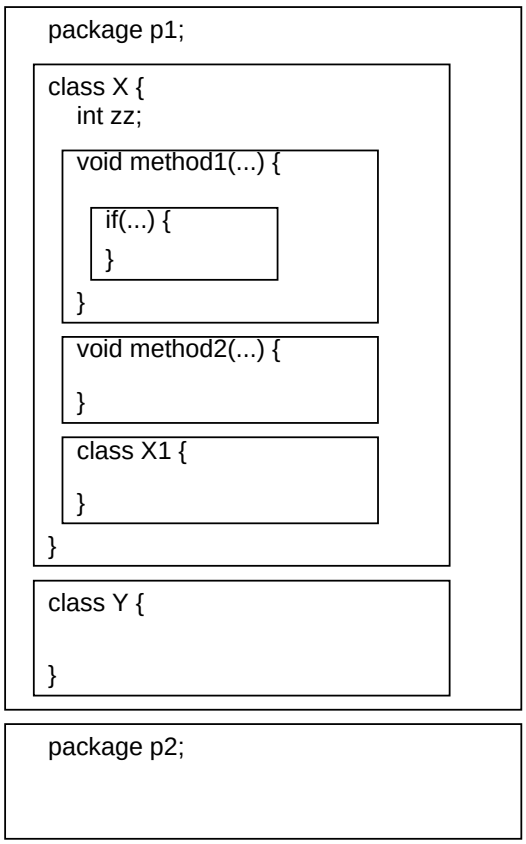


図 8: Java のスコープ単位

するオブジェクトが無いので名前だけでは呼び出せず、「オブジェクト . 名前」の形で参照する必要があります。実際、インスタンスメソッドの呼び出しはずっとそのようにしてきましたね。

関連して、`this` について説明しておきましょう。`static` のついていないメソッドでは `this` は「現在のオブジェクト」つまりメソッドが付随しているオブジェクトを意味しています。では、`static` のついていない入れ子クラス (つまり内部クラス) ではどうでしょうか。内部クラスのインスタンスメソッドでは、単に `this` と書いた場合、その内部クラスの「現在のオブジェクト」を参照します。そのオブジェクトが付属している外側のクラスのオブジェクトを参照したい場合は、「クラス名`.this`」と書くことになっています。

局所変数とパラメタ

Java で一番スコープの狭い名前はローカル変数 (局所変数) とパラメタです。たとえば、図 9 を見てください。パラメタ `x` や局所変数 `y` の有効範囲はこのメソッド全体です。[37] これに対し、局所変数 `z` は内側のブロック内で定義されていますから、そのブロックの範囲でだけ有効です。

このような規則になっているため、図 10(a) のようなプログラムはうまく動きません。2 つの `max` の定義はどちらもブロックの内側にあるので、ブロックが終わったところでその有効範囲を出てしまい、最後の `return` のところからは参照できないためです。図 10(b) のように、if 文の前に `max` の定義を入れておく必要があります。こうしておけば、`return` のところで `max` が参

[37] 正確には、局所変数は定義より前では使えませんから、定義のある場所から後ろだけが有効範囲になります。

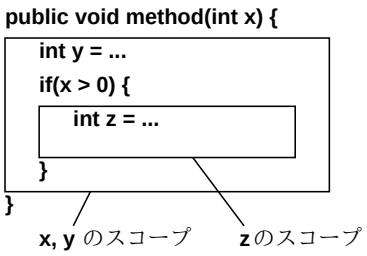


図 9: 局所変数のスコープ

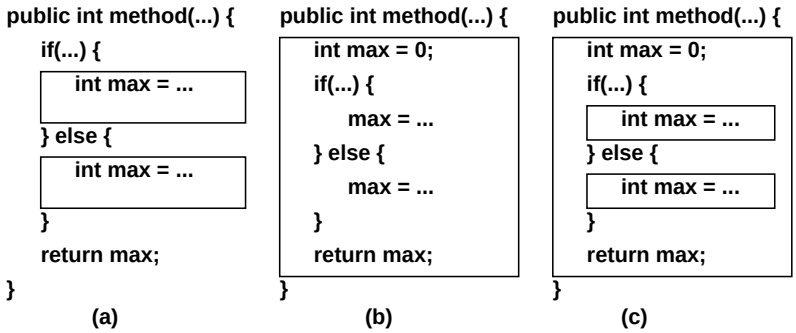


図 10: スコープが問題になりやすいコード

照できます。

ところで (a) から (b) に直すときに、図 10(c) のように変数宣言の `int` を消し忘れたらどうなるでしょうか? こうなると、内側のブロックでは「別の `max`」が定義されて使われていますから、そこで入れた値は外側の `max` には何の効果ももたらさず、`return` のところで参照される `max` の値は常に 0 になります。このように、局所変数やパラメタは「内側の変数が外側の同名の変数を隠す」ようになっていて、これが分かりにくい間違いの原因になることがあります。注意しましょう。

5 インタフェースと多相性

インタフェースとは前回にも説明しましたが、複数のクラスを「まとめて」扱うことを可能にする仕組みです。[38] ここまでの例題ではマウス操作で動かせるオブジェクトは「決め打ちで 1 個」でしたが、今度は「複数種類の図形が画面にあり、そのどれでも動かせる」ようにしてみます。そのためには、各図形が「画面に表示でき、選択でき、動かせる」という共通の切り口を持つ必要がありますが、それを表すのに次のインタフェースを定義します。

38)継承にも同じ働きがありますが、継承の場合はクラスの実装つまりインスタンス変数やメソッドも共通になります。中身には関わらず、外からの扱い方だけを共通にしたい場合はインタフェースを使う方がよいでしょう。

```
interface Figure {
  public void draw(Graphics g);
  public void moveTo(int x, int y);
  public boolean hit(int x, int y);
}
```

`draw()`、`moveTo()` はこれまで同様、オブジェクトを画面に表示させたり動かします。`hit()` は XY 座標を受け取り、その図形が XY 座標を内部に含

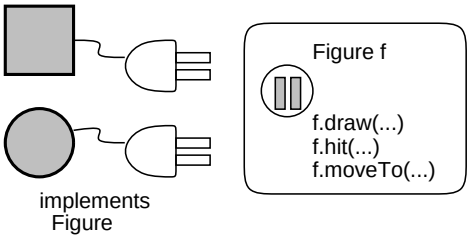


図 11: インタフェースの考え方

んでいるか[39] を判別します。どれも、各クラスで普通に実装することができます。では、これまでと何が違うのでしょう？ それは、インタフェースとして定義することで、このインタフェースに従うオブジェクトをまとめて扱えるようになるのです。図 11 のように、プログラム中の図形を扱うところで `Figure` 型の変数を使用します。そしてこのインタフェースに従う図形であればどれでも、`Figure` 型のソケットに「差し込んで」同じように表示したり動かしたりできるのです。

具体的には、`Figure` 型の変数 `f` に対して `f.draw()` を実行すると、そこを「実行する時点で」どんなオブジェクトが入っているかによって、円が描けたり長方形が描けたりします。このように、あるコードが「実際に扱う対象に応じて異なる動作を行う」ことを**多相性**ないし**ポリモルフィズム** (polymorphism) と呼びます。多相性をうまく使うと、「図形が何であれ、ここで表示」のように書けるので、コードが簡潔で読みやすくなります。[40]

39)つまり、マウスなどで特定位置をクリックしたときにその位置にその図形があるかどうかということですね。

40)オブジェクト指向以前は、「図形が X なら X の描画を実行し、Y なら Y の描画を実行し、Z なら…」のような枝分かれが必要だったので、プログラムが長く繁雑になりがちでした。

例題 5-1: ドラグできる円と長方形

では例題として、おなじみの円と長方形が 2 つずつ画面に現れ、それらのどれでもマウスで掴んでドラグすることで動かせる、というプログラムを見て頂きましょう (図 12)。[41]

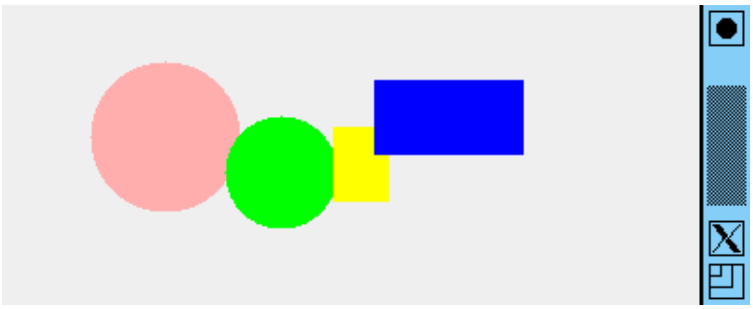


図 12: 複数種類の図形をドラグする

41)先にで入れ子クラスを説明したので、ここからは各例題の中で使う下請けのクラス群 (やインタフェース群) は本体クラスの中に入れるようにしました。この方が各例題のクラスファイルが干渉しなくて済みます。さらに、各クラスから本体クラスのインスタンス変数を参照する必要が無い場合には、干渉が起きにくい `static` 指定の入れ子クラスを使用します。`static` な入れ子クラスの説明は、付録を読み返してみてください。

このプログラムでは、任意個数の図形 (`Figure` インタフェースに従うオブジェクト) を画面上で扱うため、これらのオブジェクト群を `ArrayList<Figure>` 型のコンテナオブジェクトに入れて保持します。[42]

あと、現在選択している (掴んでいる) 図形を保持する変数も必要です。[43]では、プログラムの先頭から初期設定を行うコンストラクタまでを見てみましょう。

42)コンテナクラスについては、順番が前後してすみませんが、末尾の付録を見てください。これまでは要素を並べて扱うのに配列を使って来ましたが、コンテナクラスを使うと、大きさが自由に変化させられる、要素の挿入や削除が簡単にできるなどの柔軟性が得られます。

43)この変数は何も掴んでいないときは「オブジェクトが入っていない」ことを表す印の値 **null** が入っています。

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

public class Sample51 extends JPanel {
  ArrayList<Figure> figs = new ArrayList<Figure>();
  Figure sel = null;

  public Sample51() {
    setOpaque(false);
    figs.add(new Circle(Color.PINK, 200, 100, 40));
    figs.add(new Circle(Color.GREEN, 220, 80, 30));
    figs.add(new Rect(Color.YELLOW, 240, 60, 30, 40));
    figs.add(new Rect(Color.BLUE, 260, 40, 80, 40));
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent evt) {
        sel = pick(evt.getX(), evt.getY());
      }
    });
    addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent evt) {
        if(sel == null) { return; }
        sel.moveTo(evt.getX(), evt.getY()); repaint();
      }
    });
  }
```

まず、コンストラクタの先頭で画面を変化するモードに設定し、コンテナ figs に 2 つの円と 2 つの長方形を追加し、その後マウスボタン押しとマウスドラグを受け取るアダプタクラスを設定しています。

2 つのアダプタの中身ですが、まず、マウスボタンが押されたら、その時点のマウス位置に当たっている図形を取って来て sel に入れます (そのために下請けのメソッド pick() を呼びますが、それはすぐ次に読みます)。ドラグされたら、選択中の図形を moveTo() でマウス位置に移動して画面を描き直します。選択中の図形が無い場合は、return でメソッドをすぐ抜けることで、何もしないようにしています。

```
  private Figure pick(int x, int y) {
    Figure p = null;
    for(Figure f: figs) {
      if(f.hit(x, y)) { p = f; }
    }
    return p;
  }
  public void paintComponent(Graphics g) {
    for(Figure f: figs) { f.draw(g); }
```

```
  }
  public static void main(String[] args) {
    JFrame app = new JFrame();
    app.add(new Sample51());
    app.setSize(400, 300);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
  }
```

マウス位置に当たる図形を取り出すメソッド pick() [44] では、変数 p をまず null にしておき、figs に入っているすべての図形を順次取り出しながら [45] hit() でマウス位置に当たるか調べ、当たるものがあればそれを p に入れます。最後に p の値を返すことで、マウス位置に当たるものがあればそれが、なければ最初に入れた null が返されるわけです。paintComponent() は figs に入っているすべての図形を draw() で描くので、これまでより簡単です。main() はこれまでと変わっていません。

44)このクラス内からだけ使うので、private 指定にしてあります。

45)「for(型名 変数: 式) …」という for 文は **foreach** ループで、「式」として配列やコンテナを指定することで、その中に入っている値を順次取り出して変数に入れながらループ本体を実行します。

```
  interface Figure {
    public void draw(Graphics g);
    public boolean hit(int x, int y);
    public void moveTo(int x, int y);
  }
  static class Circle implements Figure {
    Color col;
    int xpos, ypos, rad;
    public Circle(Color c, int x, int y, int r) {
      col = c; xpos = x; ypos = y; rad = r;
    }
    public boolean hit(int x, int y) {
      return (xpos-x)*(xpos-x) + (ypos-y)*(ypos-y) <= rad*rad;
    }
    public void moveTo(int x, int y) {
      xpos = x; ypos = y;
    }
    public void draw(Graphics g) {
      g.setColor(col);
      g.fillOval(xpos-rad, ypos-rad, rad*2, rad*2);
    }
  }
  static class Rect implements Figure {
    Color col;
    int xpos, ypos, width, height;
    public Rect(Color c, int x, int y, int w, int h) {
      col = c; xpos = x; ypos = y; width = w; height = h;
    }
    public boolean hit(int x, int y) {
      return xpos-width/2 <= x && x <= xpos+width/2 &&
```

```
            ypos-height/2 <= y && y <= ypos+height/2;
      }
      public void moveTo(int x, int y) {
        xpos = x; ypos = y;
      }
      public void draw(Graphics g) {
        g.setColor(col);
        g.fillRect(xpos-width/2, ypos-height/2, width, height);
      }
    }
  }
```

インタフェース Figure は既に説明した通り、そして図形のクラス Circle と Rect についてはおなじみのものですが、Figure インタフェースに従うことを示すために implements Figure の指定が入れてあります。[46)]

46)当たり判定の hit() ですが、円については (x, y) と円の中心との距離の 2 乗 $(= (x - x_0)^2 + (y - y_0)^2)$ が半径の 2 乗以下であるかどうかを調べて返します。長方形については、X 座標、Y 座標がともに長方形の範囲内かどうかを調べて返します。

演習 5-1 例題 Sample51.java をそのまま打ち込んで動かしなさい。動いたら、次のような手直しをおこなってみなさい。

a. 例題のプログラムでは、図形を選んでも選んだ図形が重なりの手前に出てくることはありません。これは気持ちが悪いので、選んだ図形が手前に出るようにしてみなさい。

b. 例題のプログラムでは、図形でないところをクリックしても何も起きませんでしが、図形でないところをクリックした場合は新たな図形 (たとえば円) がそこに現れるようにしてみなさい。

c. 別の図形 (三角形や、できれば先にやった人や家などの複合図形) も画面に現れるようにしてみなさい。ただし、ドラグはできなくてよい。

d. やっぱりドラグできないとつまらないので、別の図形もドラグできるようにしてみなさい。[47)]

47)点 (x, y) が三角形の内側にあるかどうかを判断するのがちょっと難しいかも知れませんが、次のことをヒントにしてみてください。点 (x, y) が線分 $(x_1, y_1) - (x_2, y_2)$ の (x_1, y_1) を起点として見て左側にある/線上にある/右側にあるのいずれであるかは、式 $(x_2 - x_1) \times (y - y_1) - (y_2 - y_1) \times (x - x_1)$ が負/ゼロ/正であることに対応します (この式はベクトル $(x_1, y_1) - (x_2, y_2)$ とベクトル $(x_1, y_1) - (x, y)$ の外積を計算するもので、外積の値によって 2 つのベクトルの方向関係が分かるのですが、こういう数学な話はあまり聞きたくないですよね…とにかく、計算した式の符号を調べればいいとだけ思ってください)。

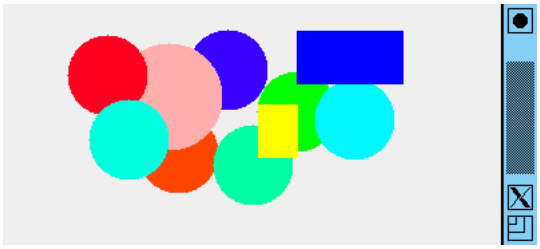
例解 5-2-ab

図形の重なり順はどのようにして決っているのでしょうか。それは、コンテナ figs に「後から」入れたものが後から描かれるので、より手前にあるように見えます。ですから、マウスボタン押しで図形が選択できたときに、その図形をコンテナからいったん取り除き、改めて追加すればよいのです。

```
  public void mousePressed(MouseEvent evt) {
    sel = pick(evt.getX(), evt.getY());
    if(sel != null) {
      figs.remove(sel); figs.add(sel); repaint();
```

次に、図形が選択できなかった場合に図形を追加する方ですが、これは適当な図形オブジェクトを作ってコンテナに追加し、さらにそのままドラグもできるように sel にも入れればいいでしょう。とりあえず円を追加することにして、色はランダムに選ぶようにしてみました。[48)]

```
    } else {
      Color c = Color.getHSBColor((float)Math.random(), 1f, 1f);
      sel = new Circle(c, evt.getX(), evt.getY(), 30);
      figs.add(sel); repaint();
    }
  }
```



例解 5-2-cd

図形をドラグできない状態で追加するのは簡単そうですが、図形クラスに `implements Figure` を指定するところにちょっと面倒があります。つまりインタフェースに従うということは、`draw()`、`moveTo()`、`hit()` の 3 つのメソッドが必ず必要なわけです。なので、`hit()` をとにかく作って、ドラグできなくていいのだから、本体に「`return false;`」とだけ書いておけばいいわけです。`moveTo()` もまだ作ってなければ「何もしない」(本体が空っぽの) ものを作れば済みます。あとは、図形オブジェクトを `figs` に追加するだけです。ここでは三角形と家を追加してみます。

```
figs.add(new Triangle(Color.RED, 200, 100, 280, 100, 220, 50));
figs.add(new Triangle(Color.YELLOW, 220, 80, 290, 90, 220, 30));
figs.add(new House(Color.BLUE, Color.GREEN, 20, 100, 120));
```

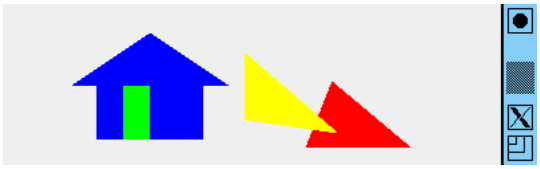
ドラグできるようにするとなると、`hit()` と、(まだ作っていなければ)`moveTo()` とを作る必要があります。まず、三角形から示しましょう。

```
static class Triangle implements Figure {
  ...
  public boolean hit(int x, int y) {
    int a = (xs[1]-xs[0])*(y-ys[0]) - (ys[1]-ys[0])*(x-xs[0]);
    int b = (xs[2]-xs[1])*(y-ys[1]) - (ys[2]-ys[1])*(x-xs[1]);
    int c = (xs[0]-xs[2])*(y-ys[2]) - (ys[0]-ys[2])*(x-xs[2]);
    return a <= 0 && b <= 0 && c <= 0;
  }
  public void moveTo(int x, int y) {
    xs[1] += x-xs[0]; ys[1] += y-ys[0];
    xs[2] += x-xs[0]; ys[2] += y-ys[0];
    xs[0] = x; ys[0] = y;
  }
```

`hit()` は問題のところに書いたヒントの通りやっています。ただしこれが使えるためには、三角形の 3 頂点を「左回り順に」指定する必要があります。[49] `moveTo()` ですが、最初の頂点の位置ががその三角形の位置だということにして、新しい XY 座標を指定したときは、2 番目と 3 番目の頂点の XY 座標を「1 番目の頂点がずれるのと同じだけずらす」ことにします。[50]

次は家ですが、`moveTo()` は前に作ったので、`hit()` だけ追加すれば済みます。それは非常に簡単で、「3 つの部分図形のどれかに当たっていれば全体として当たっている」というふうにしました。[51]

```
static class House implements Figure {
  ...
  public boolean hit(int x, int y) {
    return r1.hit(x,y) || r2.hit(x,y) || t1.hit(x,y);
  }
```



48) `Color.getHSBColor()` は、**HSB カラーモデル**のパラメタである色相 (Hue)、彩度 (Saturaiton)、明度 (Blightness) の 3 つの値を (`0.0f`〜`1.0f` の範囲の `float` 型の値として) 指定すると対応する色を返してくれます。ここでは彩度と明度は最大で、色相を**乱数** `Math.random()` で選択してみました。`Math.random()` は `double` 型の $[0.0, 1.0)$ の一様乱数を返すので、`float` 型にキャストして渡しています。

49) 実はこれまでに出て来た例はそのように指定してありました。

50) 演算子「`+=`」は、左辺の変数に右辺の計算結果を「足し込む」ので、これを使ってずらす量を右辺で計算して足し込んでいます。

51) 「扉」の範囲は「壁」の範囲に全部含まれているので、本当は壁だけ見ればいいのですが、分かりやすさのために全部見ています。

6　継承によるくくり出しと抽象クラス

インタフェースの使い方が分かったところで、また少しプログラムのリファクタリングをしましょう。例題 5-1 は十分分かりやすいですが、少し**重複**があります。具体的には、円でも長方形でも色と XY 座標があるところと、`moveTo()` で XY 座標を変更できるところは一緒でした。プログラムのコードに重複があることは、次のような理由から一般に良くないとされています。

- コードの量が多くなる。
- コードを手直しするとき、片方だけ直し忘れたり違った直し方になったりして、矛盾や不整合が起きやすい。

クラス方式のオブジェクト指向言語では、複数のクラスが持つ重複部分を 1 つの親クラスにまとめて、元のクラスをこのクラスの子クラスにすることで重複をなくすという方法が使えます。円と長方形について、これをやってみましょう。まず、両方の重複部分をくくり出す親クラスとして、`SimpleFigure` というクラスを作成します。

```
static abstract class SimpleFigure implements Figure {
  Color col;
  int xpos, ypos;
  public SimpleFigure(Color c, int x, int y) {
    col = c; xpos = x; ypos = y;
  }
  public void moveTo(int x, int y) {
    xpos = x; ypos = y;
  }
  public abstract boolean hit(int x, int y);
  public abstract void draw(Graphics g);
}
```

確かに、コンストラクタで色と XY 座標を初期化するところと、メソッド `moveTo()` があります。しかしこの **abstract** というのは、何でしょうか?

実は `SimpleFigure` というクラスは、インスタンスを生成できません。円とも長方形とも分からない (正確にはこれらの共通部分だけ取り出した) ものなわけですから… このように、複数のクラスの土台となることだけが目的で、インスタンスを生成しないクラスのことを**抽象クラス** (abctract class) と呼びます。`abstract` というのは抽象クラスを表すキーワードでした。

メソッド `hit()` と `moveTo()` についても、インタフェース `Figure` に従うために定義しますが、その中身は (円とか長方形とか具体的な形が決まらないと書きようがないため) ありません。[52] このような名前だけ定義して本体を書かないメソッドを**抽象メソッド** (abstract method) と呼びます。抽象メソッドは抽象クラスにだけ定義でき、抽象クラスを土台として作る**具象クラス** (concrete class) [53] でオーバライドして定義する必要があります。[54]

共通の親クラスができたところで、円と長方形のクラスの改良版を見てみましょう (`hit()`、`draw()` は前と同じなので省略します)。

52) 正確には、これらの抽象メソッドはこのクラスが実装している `Figure` インタフェースにあるものなので、ここに書かなくてもインタフェースから定義が持って来られます。しかし、これらが必要なことはっきりさせるために、本書では実装しているインタフェースにあるものでも抽象メソッドをきちんと定義するようにします。

53) 抽象クラスでない、インスタンスを生成できるクラスのことを、具象クラスと呼びます。

```
static class Circle extends SimpleFigure {
  int rad;
  public Circle(Color c, int x, int y, int r) {
    super(c, x, y); rad = r;
  }
  ...

static class Rect extends SimpleFigure {
  int width, height;
  public Rect(Color c, int x, int y, int w, int h) {
    super(c, x, y); width = w; height = h;
  }
  ...
```

54)抽象クラスのサブクラスとして抽象クラスを定義することもでき、その中で一部の抽象メソッドにだけ本体を与えることもできます。しかし、下の方のどこかでは全部の抽象メソッドを定義した具象クラスにする必要があります。そうしないと、インスタンスが作れなくて、クラスとして訳に立たないからです。

いずれも、先に定義した SimpleFigure のサブクラスとすることで、col、xpos、ypos を継承し、また moveTo() の定義を継承します。[55][56] しかし、このコンストラクタの中の **super** というのは何でしょう? これは、親クラス (この場合は SimpleFigure) のコンストラクタを呼び出すための指定で、そのパラメタとして渡した色と XY 座標は、SimpleFigure のコンストラクタの中で col、xpos、ypos を初期設定するのに使われます。[57] このように、クラスを継承してサブクラスを作る時には、その先頭で「super(...)」を書いて親クラスのコンストラクタのどれかを呼び出す、ということは覚えておいてください。[58]

55)親クラスが実装しているインタフェースは子クラスに引き継がれるので、implements Figure は指定しなくても指定されたのと同じことになります。

56)独自に必要な変数 rad、width と height については、それぞれ追加しています。

57)なぜこうなっているのかというと、子クラスのインスタンスの中には親クラスのインスタンスが「埋め込まれて」いて、それを初期設定するためには親クラスのコンストラクタを「呼ばなければならない」、というのが Java の設計方針だからです。親クラスで定義したインスタンス変数も子クラスのコンストラクタで直接初期設定した方が簡単では、と思うかも知れませんが、それは許されていません。

58)しかしずっと JFrame のサブクラスを作って来たけれど super(...) は書いていない、と思ったかも知れません。それは、JFrame は引数なしのコンストラクタを持っているからで、親クラスに引数なしのコンストラクタがあれば、子クラスのコンストラクタで「super(...)」を書かなかった場合はその先頭で親クラスの引数なしのコンストラクタを自動的に呼び出すようになっています。すべてのコンストラクタが引数ありの場合は、自動的というわけには行かないので、自分で「super(...)」を書く必要があります。

演習 5-2 これまでに作成した複数のクラスを持つ演習問題の解に対して、複数のクラスに共通部分があったらそれを抽象クラスとしてくくり出してみなさい。リファクタリングなので動作は変更しないこと。[59]

59)これはあんまり楽しい演習ではないかも知れませんが、継承のしくみについて慣れておくためには有用だと思いますよ。

7 型の判定と行き来

ここまでで、インタフェース型の変数にはそのインタフェースを実装しているオブジェクトが入れられること沢山見てきました。このほか、前回説明しましたが、親クラスの型の変数に子クラスのオブジェクトを入れることもできます。では逆に、インタフェース型や親クラス型の変数に入っているオブジェクトを、元の型に戻すには…それには、**キャスト**を使うのでしたね。[60] このようなキャストを、子クラスなど「下の方の」型に変換することから**ダウンキャスト**と呼びます。

しかし、インタフェース型や親クラス型の変数には、さまざまなクラスのインスタンスが入っています。それを実行時に「元の型」に戻そうとしても、別の「元の型」のオブジェクトかも知れません。その点を判定するために、**instanceof** 演算子が使えます。たとえば、Figure 型の変数 f1 にどれかの図形が入っているとして、それが円である時だけ何かしたければ、次のような if 文を書くわけです。

```
if(f1 instanceof Circle) {
  Cicle c1 = (Cicle)f1; // ダウンキャスト
  円に対する処理 ...
```

チェックして OK の場合だけダウンキャストをしているので、このダウンキャストは失敗しません。 [61] 多相性を活用していくということは、場合によってはこのような処理をとりまぜて行くことも必要なのです。

60)なぜ元の型に戻したいかというと、元の型の変数に入れたい場合や、元の型が持っているメソッドの呼び出しを行いたい場合があるためです。線の太さを変えるために、Graphics 型のオブジェクトを Graphics2D にキャストして setStroke() を呼んでいたのが、まさに後者に相当します。

61)ダウンキャストが失敗した場合は、例外が発生します。例外については後で扱います。

62)共通部分の抽象クラスへのくくり出しは、やってもあまりすっきりしないので、ここではやっていません。

例題 5-3: マルバツ

型の判定を含む例題として、マルバツ (三目並べ、tic-tac-toe) のプログラムを作ります (図 12)。このプログラムでは画面に「ます目」を表す四角形と○と×が現れますが、これらはいずれも Figure インタフェースに従うオブジェクトですが、ただし今度は Figure は draw() だけを持つことにします。これまでと順番を変えて、図形のクラス群から見てみましょう。[62]

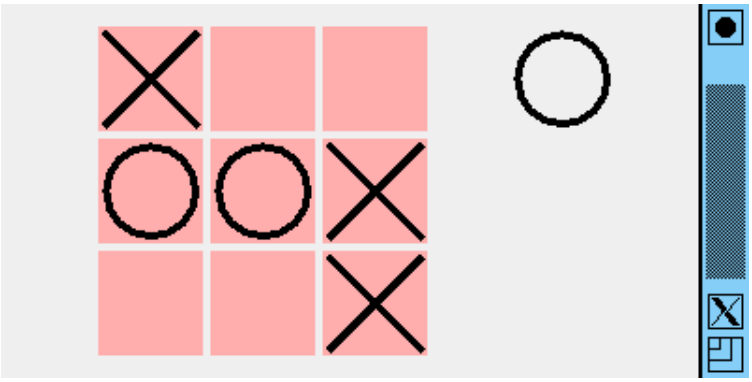


図 13: マルバツの画面

```
interface Figure {
  public void draw(Graphics g);
}
static class Maru implements Figure {
  int xpos, ypos, sz;
  public Maru(int x, int y, int s) {
   xpos = x; ypos = y; sz = s;
  }
  public void draw(Graphics g) {
    g.setColor(Color.BLACK);
    ((Graphics2D)g).setStroke(new BasicStroke(4));
    g.drawOval(xpos-sz, ypos-sz, 2*sz, 2*sz);
  }
}
static class Batsu implements Figure {
  int xpos, ypos, sz;
  public Batsu(int x, int y, int s) {
    xpos = x; ypos = y; sz = s;
  }
  public void draw(Graphics g) {
    g.setColor(Color.BLACK);
```

```
      ((Graphics2D)g).setStroke(new BasicStroke(4));
      g.drawLine(xpos-sz, ypos-sz, xpos+sz, ypos+sz);
      g.drawLine(xpos-sz, ypos+sz, xpos+sz, ypos-sz);
    }
  }
  static class Rect implements Figure {
    Color col;
    int xpos, ypos, width, height;
    public Rect(Color c, int x, int y, int w, int h) {
      col = c; xpos = x; ypos = y; width = w; height = h;
    }
    public boolean hit(int x, int y) {
      return xpos-width/2 <= x && x <= xpos+width/2 &&
             ypos-height/2 <= y && y <= ypos+height/2;
    }
    public int getX() { return xpos; }
    public int getY() { return ypos; }
    public void draw(Graphics g) {
      g.setColor(col);
      g.fillRect(xpos-width/2, ypos-height/2, width, height);
    }
  }
```

では、本体側を見てみましょう。このプログラムではインスタンス変数として、図形を入れるコンテナ figs の他に、次が○の番か×の番かを覚えておくための boolean 型変数 turn を持ちます。コンストラクタでは、ます目となる 9 個の長方形と、最初はバツの手であることを表す Batsu オブジェクトを 1 個、figs に入れます。[63)]

63)演算子「/」は整数の切捨て除算、%は「割った余り」なので、i が 0、1、2、3、4、5、6、7、8 と変化するとき、r は 0、0、0、1、1、1、2、2、2、c は 0、1、2、0、1、2、0、1、2 と変化します。これを利用して、1 重の for ループで 9 個のます目を縦横に並べているわけです。

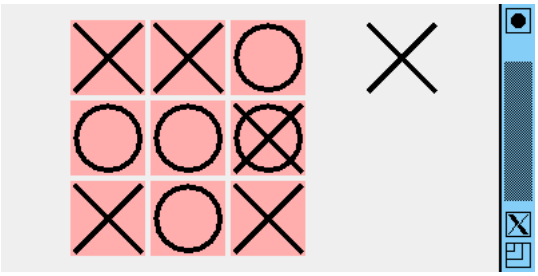
```
  import java.awt.*;
  import java.awt.event.*;
  import javax.swing.*;
  import java.util.*;

  public class Sample51 extends JPanel {
    ArrayList<Figure> figs = new ArrayList<Figure>();
    boolean turn = true;

    public Sample51() {
      setOpaque(false);
      for(int i = 0; i < 9; ++i) {
        int r = i / 3, c = i % 3;
        figs.add(new Rect(Color.PINK,80+r*60,40+c*60,56,56));
      }
      figs.add(new Batsu(300, 40, 24));
      addMouseListener(new MouseAdapter() {
        public void mouseClicked(MouseEvent evt) {
```

```
        Rect r = pick(evt.getX(), evt.getY());
        if(r == null) { return; }
        figs.remove(figs.size()-1);
        if(turn) {
          figs.add(new Batsu(r.getX(), r.getY(), 24));
          figs.add(new Maru(300, 40, 24));
        } else {
          figs.add(new Maru(r.getX(), r.getY(), 24));
          figs.add(new Batsu(300, 40, 24));
        }
        turn = !turn; repaint();
      }
    });
  }
  public Rect pick(int x, int y) {
    Rect r = null;
    for(Figure f: figs) {
      if(f instanceof Rect && ((Rect)f).hit(x, y)) {
        r = (Rect)f;
      }
    }
    return r;
  }
  public void paintComponent(Graphics g) {
    for(Figure f: figs) { f.draw(g); }
  }
  public static void main(String[] args) {
    JFrame app = new JFrame();
    app.add(new Sample51());
    app.setSize(400, 300);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
  }
  // Figure, Maru, Batsu, Rect の定義をここに入れる
}
```

マウスクリックがあったら、メソッド pick() でその位置に当たる四角を取り出します。どの四角にも当たらなければ、それで終わりです。当たっていれば、figs に最後に入れた図形 (○×どちらの手かの表示) を消して、それと同じ手を四角の位置に生成し、次の手を右側に置きます。そして、「次の手」を反転し、[64] 画面を描き直します。

pick は例題 5-1 と同様ですが、ただし hit() を持っているのは Rect だけですから、まず instanceof で Rect かどうかを調べ、OK の場合だけ hit() で当たりをチェックします。この部分で型の判定とダウンキャストが使われるわけです。paintComponent() と main() はこれまでと同様です。これでぶじマルバツができましたが、ただしどこに打ったかのチェックはしていないので、既に打った場所に重ねて打つこともできてしまいます。

64)「!」は論理値の反転つまり true を false、false を true に変換する演算子です。



演習 5-1 例題 Sample51.java をそのまま打ち込んで動かしなさい。動いたら、次のような手直しをおこなってみなさい。

a. 例題のます目は 3 × 3 だが、もっとます目を多くして、5 目並べができるようにしてみなさい。

b. 同じ場所に重ねて打たないことをチェックするように直してみなさい。

c. 3 目並べでも 5 目並べでもよいので、勝負がついたら「おめでとう」のメッセージが出るようにしてみなさい。[65] 正しくない場所に打とうとしたらその旨警告するとなおよいでしょう。[66]

d. 16 × 16 のます目が表示され、クリックすると色が変わり、再度クリックすると戻る (または N 色が循環で変化する) ようにして、タイル絵のようなものがデザインできるプログラムを作ってみなさい。

e. ます目の盤面を持った、自分の好きなゲームを作ってみなさい。[67] 自動対戦機能もつけられるとなおよいでしょう。

例解 5-1-abc

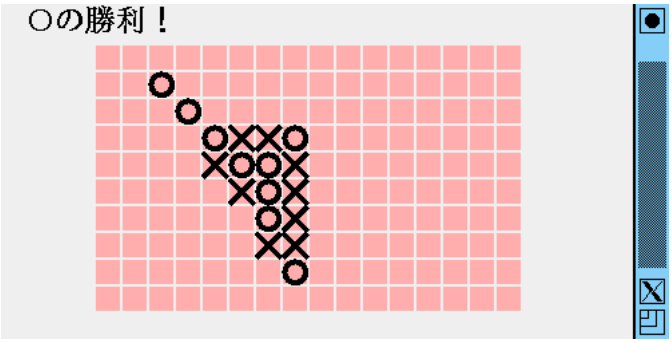
例題を拡張して、勝敗判定つきの 5 目並べにしてみます。まず、メッセージを出す部分を考えましょう。そのために、メッセージ表示用のオブジェクトを作ります。文字列を表示するには、Graphics オブジェクトのメソッド drawString() を使いますが、その前に setFont() でフォントを設定して大き目の文字にする方がよいでしょう。そこで、XY 座標、文字列、フォントを持つクラス Text をに用意しました。

```
static class Text implements Figure {
  int xpos, ypos;
  String txt;
  Font fn;
  public Text(int x, int y, String t, Font f) {
    xpos = x; ypos = y; txt = t; fn = f;
  }
  public void setText(String t) { txt = t; }
  public void draw(Graphics g) {
    g.setColor(Color.BLACK); g.setFont(fn);
    g.drawString(txt, xpos, ypos);
  }
}
```

65) 文字列を表示するためには、Graphics オブジェクトのメソッド setFont() を呼んで表示に使うフォントを設定し、おなじみ setColor() で文字の色を設定し、その後 drawString() で指定した文字列を指定した位置に描きます。また、フォントはクラス Font によって表しますが、Font のコンストラクタは、フォント種別、文字飾り種別、文字サイズの 3 つを指定します。とりあえず、「new Font("Serif", Font.BOLD, 20)」などとしてみてください。はじめてのメソッドやオブジェクトについては、付録か API ドキュメントで使い方を確認してください。

66) 5 目並べだと「33」が禁止なので、それも組み込めるとさらによいのですが、けっこう大変です。

67) オセロ、チェッカー、はさみ将棋などが考えられます。



あとは下請けクラスの変更はなく、本体部分だけを直しますが、いろいろと新しい概念が必要なのでここでまとめて説明します。まず、ます目の空きを調べたりいくつ並んでいるかを調べるには、これまでのように画面にものが見えるだけでは済まず、ゲームの状態をプログラム内でもデータとして保持する必要があります。5 目並べはます目が縦横に並んでいますから、プログラム内のデータ構造も値が縦横に並んだ構造、具体的には **2 次元配列**を使います。

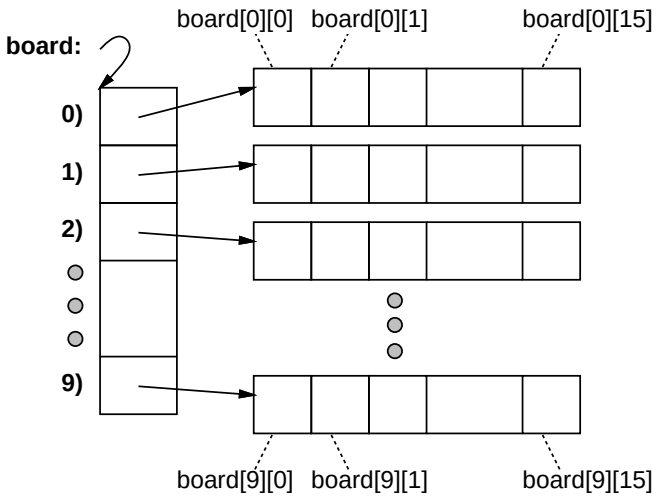


図 14: 2 次元配列

Java では 2 次元配列は「配列の配列」つまり、配列オブジェクトのそれぞれの要素として配列が保持されているものです (図 14)。今回は `board` という名前の変数に縦 10 行、横 16 列の整数の 2 次元配列を入れるために、次のようなコードを書きます。

```
int[][] board = new int[10][16];
```

変数の型は「配列の配列」なので「`int[][]`」、そして生成するときも縦と横の大きさを `new` において 2 つのかっこ内の整数で指定します。2 次元配列の要素は、最初の添字でどの行か、2 番目の添字でどの列かを指定します。今回の場合は、一番上の行が `board[0][0]`、`board[0][1]`、…、`board[0][15]`、次の行が `board[1][0]`、`board[1][1]`、…、`board[1][15]`、一番下の行が `board[9][0]`、`board[9][1]`、…、`board[9][15]` ということになります。

ゲームの状態ということは、これらの各ます目が「あき/○/×」のどれであるかを記録する必要があります。ここではこの 3 つの値を 0、1、2 の整数

で表すことにして、名前をつけるようにしました。

```
static final int EMPTY = 0, BATSU = 1, MARU = 2;
```

final と指定された変数は、いちど値を入れたら変更できないので、このような定数を保持するためによく使います。[68][69] つまり、2 次元配列 board には最初すべてのます目に EMPTY が入っていて、[70] ゲームの進行につれて EMPTY のます目に BATSU や MARU が入っていく、ということになります。そして、この内容をチェックすることで、「ある場所が既に打った場所かどうか」「どちらかが勝ったか」などのチェックができるわけです。

インスタンス変数としては、例題 5-3 からある figs、turn、上述の board、メッセージ用の Text オブジェクトを入れておく t1 に加えて、winner という整数の変数があります。これは文字通り「勝者」を記録するためのもので、最初は EMPTY を入れておき、どちらかが勝ったらその値が入るようにしました。この変数をチェックすることで、いちど勝負がついたらそれ以上手を打たないようにしています。

説明が長くなりましたが、いよいよコードを見てみてください。[71]

```
public class ex51abc extends JPanel {
  static final int EMPTY = 0, BATSU = 1, MARU = 2;
  static final int YMAX = 10, XMAX = 16;
  ArrayList<Figure> figs = new ArrayList<Figure>();
  boolean turn = true;
  int winner = EMPTY;
  int[][] board = new int[YMAX][XMAX];
  Text t1 = new Text(20, 20, "五目並べ、次の手番：×",
              new Font("Serif", Font.BOLD, 22));

  public ex51abc() {
    figs.add(t1);
    for(int i = 0; i < 160; ++i) {
      int r = i / YMAX, c = i % YMAX;
      figs.add(new Rect(Color.PINK, 80+r*20, 40+c*20, 18, 18));
    }
    setOpaque(false);
    addMouseListener(new MouseAdapter() {
      public void mouseClicked(MouseEvent evt) {
        Rect r = pick(evt.getX(), evt.getY());
        if(r == null || winner != EMPTY) { return; }
        int x = (r.getX()-80)/20, y = (r.getY()-40)/20;
        if(board[y][x] != EMPTY) {
          t1.setText("空いてません"); repaint(); return;
        }
        if(turn) {
          figs.add(new Batsu(r.getX(), r.getY(), 8));
          board[y][x] = BATSU;
        } else {
          figs.add(new Maru(r.getX(), r.getY(), 8));
          board[y][x] = MARU;
        }
        int s = board[y][x], a = ck(1,1,s), b = ck(1,-1,s);
        int c = ck(1,0,s), d = ck(0,1,s);
        if(a > 4 || b > 4 || c > 4 || d > 4) {
          t1.setText((turn?"×":"○")+"の勝利！");
          winner = turn ? BATSU : MARU;
        } else {
          turn = !turn;
```

68) ます目の縦横の数も後で一括して変更しやすくするため、同様に定数として持つようにしました。

69) 「複数の場合のどれか」を表す値のことを一般に**列挙値**と呼びます。多くのプログラミング言語は、列挙値を扱うための専用の機能を持っていて、実は Java もそうなのですが、ここでは新しいことがあまり沢山出てこない方がよいので、整数の定数で代用しています。

70) Java では初期値を入れないままの整数の変数や配列要素には自動的に 0 が入るので、ちょうど EMPTY になっているわけです。

```
            t1.setText("次の手番：" + (turn?"×":"○"));
          }
          repaint();
        }
      });
    }
```

71)この例題では文字列に日本語を使っています。日本語の箇所でエラーが出るなどしてコンパイルできない場合は、「javac -encoding JISAutoDetect ファイル名」でコンパイルしてみてください。日本語の扱いについては後でもっと詳しく説明します。

ほとんどの処理はマウスクリックのイベントハンドラで行います。まずクリック時に当たる四角が無かったり勝負が済んでいる場合は何もせずに戻ります。次に当たった四角の座標から逆算して箱の縦横の番号を計算し、その位置が空いていない場合はメッセージを「空いていません」に取り替えます。これら以外の場合は、実際に手を打つことになるので、これまでと同様に○か×を増やしますが、そのとき配列 board にもその情報を記入します。次に、今打った手によって 5 並びができたかどうかを調べますが、それには下請けメソッド ck() に対して、並んでいる手の種類 (MARU、BATSU) と並び方向 (右下がりの斜め、右上がりの斜め、水平、垂直) を渡して、その方向にその手が最大いくつ並んでいるかを調べます。そしで、どれかの向きで 5 以上並んでいたら終わりなので、メッセージを変更して winner を設定します。そうでなければ、手番を反転してその旨のメッセージを出します。

最後に下請けのメソッド ck() を示します。これは、盤面のすべてのますからはじめて、指定方向に指定した手が並んでいる数を変数 c に数え、その最大値を変数 max として更新していくことで、最大の並び数を数えます。

```
private int ck(int dx, int dy, int s) {
  int max = 1;
  for(int y = 0; y < YMAX; ++y) {
    for(int x = 0; x < XMAX; ++x) {
      int c = 0;
      for(int k = 0; k < 5; ++k) {
        int x1 = x + dx*k, y1 = y + dy*k;
        if(y1 < 0 || y1 >= YMAX || x1 < 0 || x1 >= XMAX ||
           board[y1][x1] != s) { break; }
        ++c;
      }
      max = Math.max(max, c);
    }
  }
  return max;
}
```

8 付録: コンテナクラスとパラメタつきクラス

これまでは、一連の値やオブジェクトの並びをまとめて扱うためには、配列を使って来ました。配列は Java 言語に基本として備わっている機能ですが、次のような弱点があります。

- 作る時に要素数 (大きさ) を決める必要があり、その値は作った後は変更できない。
- 各要素のアクセスは常に「何番目」を指定して格納したり取り出す必要がある。72)

72)ただし、foreach ループを使えば「全部順番に取り出して処理」だけは簡単に書けます。

これらの制約なら逃れたい場合には、Java では配列の変わりに標準ライブラリにある各種のコンテナクラスを使うことができます (値を「入れてお

く」ことからこのような名前で呼ばれています)。例として、配列の代わりによく使われるクラス ArrayList を見てみましょう。このクラスはパッケージ java.util に含まれていますから、使う時にはファイル冒頭に「import java.util.*;」の指定を入れてください。

配列を使う時に「何の型を並べた配列」という指定をするのと同様、ArrayList も「何の型を並べた ArrayList」という指定が必要です。この「何の型」の部分を「<...>」の中に書くので、たとえば Circle オブジェクトを入れる場合は「ArrayList<Circle>」という型指定になります。

このような、「<...>」のついたクラスのことを**パラメタつきクラス**ないし**ジェネリッククラス**と呼びます。[73][74]

ジェネリッククラス ArrayList<E>のコンストラクタとメソッドの代表的なものとして、次のものがあります (「何の型」つまりパラメタ型を E で表しています)。

- new ArrayList<E>() — 空の並びを生成する (コンストラクタ)。
- void add(E) — 要素を末尾に追加。
- void set(int, E) — 位置を指定して要素を格納。
- E get(int) — 位置を指定して要素を取り出す。
- void remove(int) — 位置を指定して要素を削除。
- void remove(Object) — 要素を指定してその要素を削除。
- int size() — 現在の要素数を返す。
- Iterator<E> iterator() — 各要素を返すイテレータを返す。

最後の Iterator<E>というのは、「E 型の要素を次々に返すオブジェクト」を表す型で、これを返すメソッド iterator() があるおかげでこのクラスも foreach ループで使うことができます。[75] その実例は例題で繰り返し出て来ました。

なお、ジェネリッククラスのパラメタとして書けるのはクラス名なので、整数などの基本型をコンテナクラスで扱う際には注意が必要です。つまり、int を扱いたければパラメタとしては対応する包囲クラス Integer を指定してください。int 値と Integer オブジェクトの間の行き来は自動ボクシング/アンボクシングで処理されますから、あとはあまり手間なしに基本型が扱えます。

```
ArrayList<Integer> a = new ArrayList<Integer>();
...
a.add(1); a.add(2); a.add(3);  // 自動ボクシング
...
for(int i: a) {                // 自動アンボクシング
  ... i の値を使用 ...
}
```

73)このパラメタつきクラスの機能は JDK 1.5 から入ったもので、古い Java 言語にはありませんでした。

74)Java 言語では制約として、パラメタには基本型が書けません。このため、int や double などの基本型をコンテナクラスに入れたい場合は、代わりに Integer や Double などの包囲クラスをパラメタに指定するようにします。値の格納や取り出し時には、自動ボクシング/アンボクシングによる変換が行えるので、あまり意識しなくても基本型の値を出し入れしているかのように使えます。

75)正確には、クラスがメソッド Iterator<E> iterator() を定義したインタフェース Iterable<E>を実装している場合に、そのクラスのインスタンスを foreach ループで使うことができるようになります。