オブジェクト指向プログラミング'10 ＃3

久野　靖*

2010.9.21

今回は前回説明したけれど難しかった「無名内部クラス」について理解していただくために、少し前にもどってフレームワークの話から復習していきます。その後は、前回の続きとして「複数の図形を動かす」題材を取り上げ、多相性について学びます。

1　フレームワークと無名内部クラス

オブジェクト指向でクラスやインタフェースを活用するやり方の 1 つに、**アプリケーションフレームワーク**と呼ばれるものがあります。これは、プログラムの大きな枠組みはすでに作られたものがあって、その一部分として自分が書いたものを「はめ込む」ことで自分が作りたいものを完成させる、というものです (図 1)。

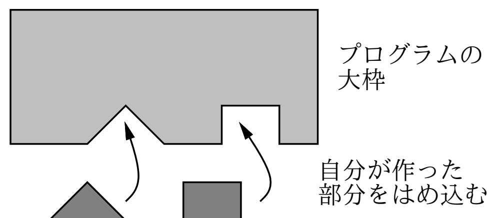


図 1: アプリケーションフレームワークの考え方

そんなうまいことができるのか、と言われそうですが、できるのです。オブジェクト指向では、あるクラスやインタフェースのオブジェクトを使うように書かれているコードに、そのクラスのサブクラスや、そのインタフェースに従うさまざまなクラスを渡して使わせることができます。ですから、サブクラスやインタフェースに従うクラスを作り、その中でメソッドをオーバライドしたり定義することで、「自分が書いたもの」を残りの部分にはめ込めるわけです。この考え方は、これからもずっと使うことになりますから、ぜひ覚えておいてください。[1)]

1)正確には「アプリケーション全体に対する」はめ込みを行う場合にアプリケーションフレームワークと呼ぶので、もっと小さい部分に対するはめ込みについてはとくに名前がないか、または単に「フレームワーク」と呼びます。

1.0.1　復習: これまでの例題の構造を再考する

ここで、最初の例題「画面に絵を描く」に戻って、上記のフレームワークがどのように Java で表現されているかを見てみます。

*経営システム科学専攻

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

public class Sample21 extends JPanel {
  public void paintComponent(Graphics g) {
    g.setColor(new Color(255, 180, 99));
    g.fillOval(100, 50, 100, 100);
  }
  public static void main(String[] args) {
    JFrame app = new JFrame();
    app.add(new Sample21());
    app.setSize(400, 300);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
  }
}
```

このコードで、main() は実行を始めるところであり、クラス本体とはあまり関係がありません。そしてこのクラスは、JPanel のサブクラスで、paintComponent() を差し替えることで「自分の好きな中身を描く」パネルにしてたわけです。この部分がフレームワークの形になっているわけですね。

こうして整理してみると、この例題で肝心なところは「paintComponent() を自分で定義したパネルをはめる」というところであり、そのためには必ずしも Sample21 を JPanel のサブクラスにする必要はありません。次のように、本体とは別のクラスを作ってはめてもいいのです。

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

public class Sample21b {
  public static void main(String[] args) {
    JFrame app = new JFrame();
    app.add(new MyPanel());
    app.setSize(400, 300);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
  }
}
class MyPanel extends JPanel {
  public void paintComponent(Graphics g) {
    g.setColor(new Color(255, 180, 99));
    g.fillOval(100, 50, 100, 100);
  }
}
```

さらに、これらのプログラムでは、パネルのオブジェクトを生成する箇所は 1 箇所だけです。Java では「あるクラスのサブクラスを作ってそのインスタンスを使う」「あるインタフェースを実装したクラスを作ってそのイン

スタンスを使う」ことは頻繁にあるので、(1) 指定する親クラスまたはインタフェースが 1 個である、(2) コンストラクタを定義しない、(3) そのクラスを参照する場所[2)]が 1 箇所である、という条件が満たされるなら、図 2 のようにして、その場所に直接クラス定義を書くことができます。この場合、書き換えたところで指定するのは親クラス (またはインタフェース) の名前であることに注意してください。つまり、定義するクラスには名前をつけません。このため、この書き方を**無名内部クラス**と呼んでいます。

2)new でオブジェクトを生成する場所のことです。

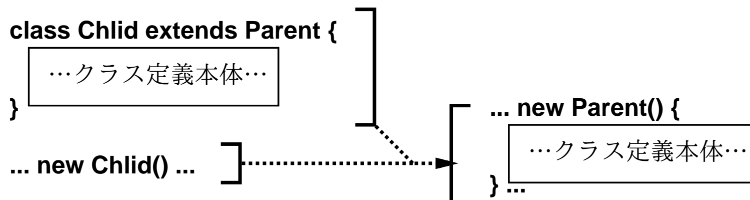


図 2: 無名内部クラス

例題をこの書き方に直すと、次のようになります。[3)]

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

public class Sample21c {
  public static void main(String[] args) {
    JFrame app = new JFrame();
    app.add(new JPanel() {     // 無名クラスのはじまり
      public void paintComponent(Graphics g) {
        g.setColor(new Color(255, 180, 99));
        g.fillOval(100, 50, 100, 100);
      }
    });  // 無名クラス終わりの「}」と app.add の終わりの「);」
    app.setSize(400, 300);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
  }
}
```

3)「app.add(…)」の「…」の部分に無名クラスの定義がはさまっているので、慣れるまで読みにくいと思います。

実際には、インスタンス変数を共通のデータの置き場所として利用したり、コンストラクタを定義したりしたいので、プログラム本体のクラスについては JPanel のサブクラスにするという最初の書き方を使って来ました。しかし、微小な部分でフレームワークのためにサブクラスをたくさん作るような場合には、いちいち別の箇所にクラス定義を書くと分かりにくくなるので、その場合は無名クラスの書き方を使うようにします。

復習: イベントの受け取り

ここで、入力イベントの受け取りについて復習しておきます。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 3)。

- 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。4)
- 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。

4)この章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。

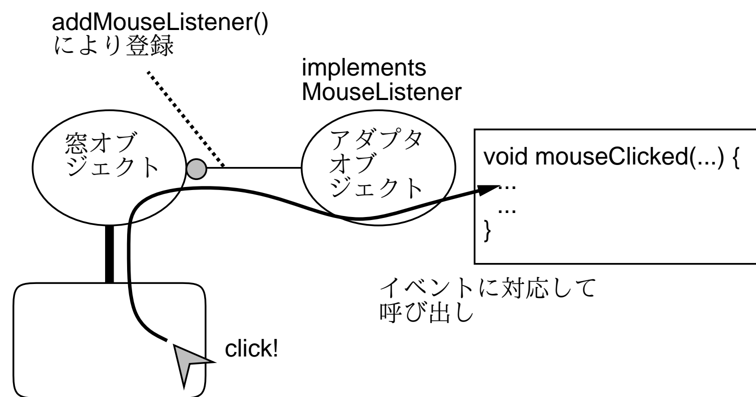


図 3: 入力イベントの受け取り

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は、次のようになっていました。5) 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。

5)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

実際には、たとえばMouseListenerインタフェースであれば、メソッドが5つあるので、5つのメソッドを定義したクラスを作成して、それにimplements MouseListenerと指定するわけです。しかし、実際に使いたいのはイベントのうちの一部 (たとえばmousePressed()だけ) だったりするわけです。そこで、これらのインタフェースを実装し、実際には何もしないメソッドをひととおり定義したクラスMouseAdapter、MouseMotionAdapter、KeyAdapterが提供されていて、我々はこれらのサブクラスを作成して、必要なメソッドだけ差し替えるわけです。この形もまた、これまでに見て来たフレームワークの形になっていることに注意してください。

そういうわけで、そのようなアダプタクラス定義+オーバライドの記述は次のようになります。

```
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 ...
  }
}
```

そして、このクラスを参照してアダプタを設定する部分は次のようになります。

```
  addMouseMotionListener(new MyAdapter1());
```

なのですが、この5行くらいしかないクラスを毎回定義して、それを参照する、というのが沢山出て来るとどれがどれだか分からなくなります。そこで、無名内部クラスを使って、設定する箇所に直接アダプタクラスの定義+オーバライドの記述を書きます。

```
  addMouseMotionListener(new MouseMotionAdapter() {
    public void mouseDragged(MouseEvent evt) {
      ... ここにドラグに対応する動作を書く ...
    }
  });
```

これが前回やっていたイベントの受け取り部分のコードなわけです。

2 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。[6] 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。

6)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);
  }
```

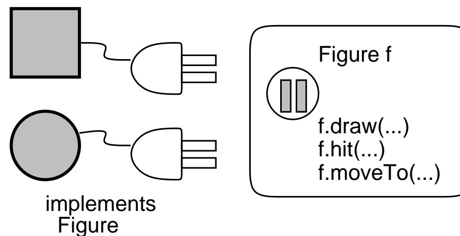


図 4: インタフェースの考え方

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 座標を内部に含んでいるか[7] 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 4 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。

[7] 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。

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。[8]

[8] 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 5)。[9]

[9] 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。

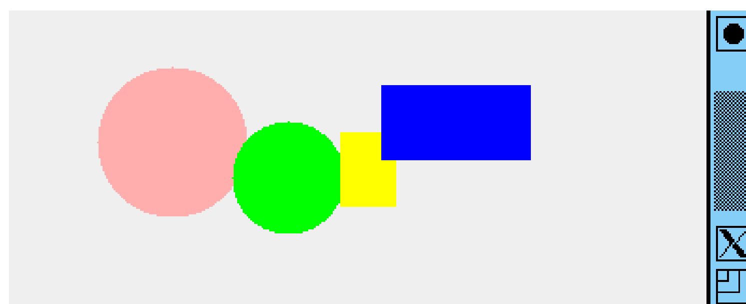


図 5: 複数種類の図形をドラグする

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。[10]

[10] 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。

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。[11]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。

[11]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() [12] 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 [13] 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。

12) このクラス内からだけ使うので、private 指定にしてあります。

13) 「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。[14]

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。[15]

14) 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。

15) 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。[16]

```
    } 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。[17] moveTo() ですが、最初の頂点の位置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。[18]

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。[19]

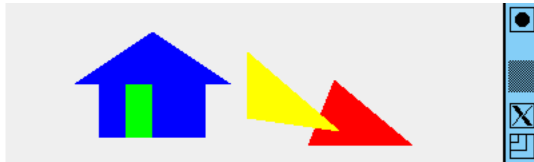
16) Color.getHSBColor() は、HSB カラーモデルのパラメタである色相 (Hue)、彩度 (Saturaiton)、明度 (Blightness) の 3 つの値を (0.0f〜1.0f の範囲の float 型の値として) 指定すると対応する色を返してくれます。ここでは彩度と明度は最大で、色相を乱数 Math.random() で選択してみました。Math.random() は double 型の $[0.0, 1.0)$ の一様乱数を返すので、float 型にキャストして渡しています。

17) 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。

18) 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。

19) 「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。

```
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);
  }
```



3　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

20)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。

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。[20] 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) [21] でオーバライドして定義する必要があります。[22]

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)。

```
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;
  }
  ...
}
```

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。[23][24] 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。[25] 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。

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。[26]

21) 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。

22) 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。

23) 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。

24) 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。

25) 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。

26) 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。

4　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。[27] 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。[28] 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。

27) 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。

28) 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。

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 5)。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。[29]

29) 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。

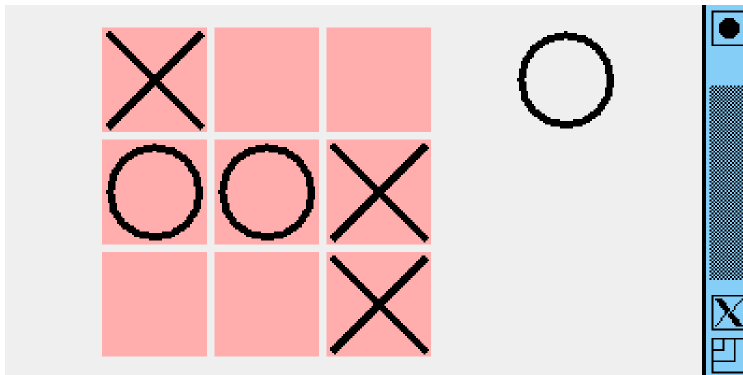


図 6: マルバツの画面

```
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。[30)]

```
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); }
```

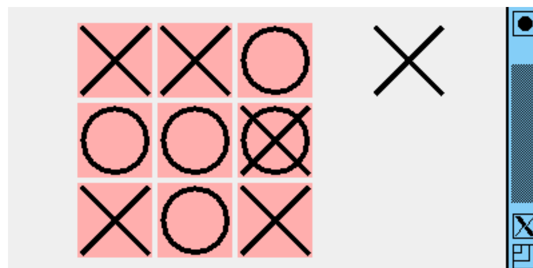
30)演算子「/」は整数の切捨て除算、%は「割った余り」なので、i が 0、1、2、3、4、5、6、7、8 と変化するとき、r は 0、0、0、1、1、1、2、2、2、c は 0、1、2、0、1、2、0、1、2 と変化します。これを利用して、1 重の for ループで 9 個のます目を縦横に並べているわけです。

```
  }
  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、[31] 画面を描き直します。

31)「!」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。

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。



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。[32]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。[33]

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。[34] 自動対戦機能もつけられるとなおよいでしょう。

32)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。

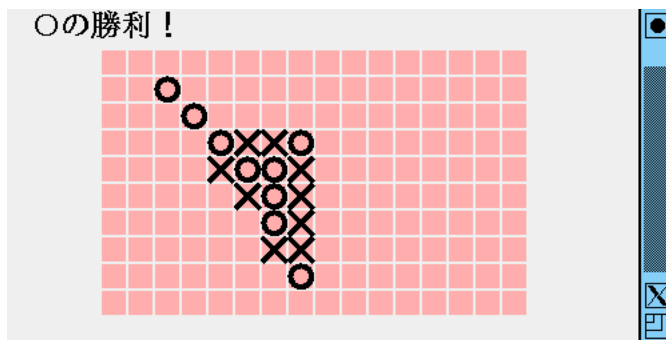
33)5 目並べだと「33」が禁止なので、それも組み込めるとさらによいのですが、けっこう大変です。

34)オセロ、チェッカー、はさみ将棋などが考えられます。

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);
  }
}
```



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。

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 7)。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。

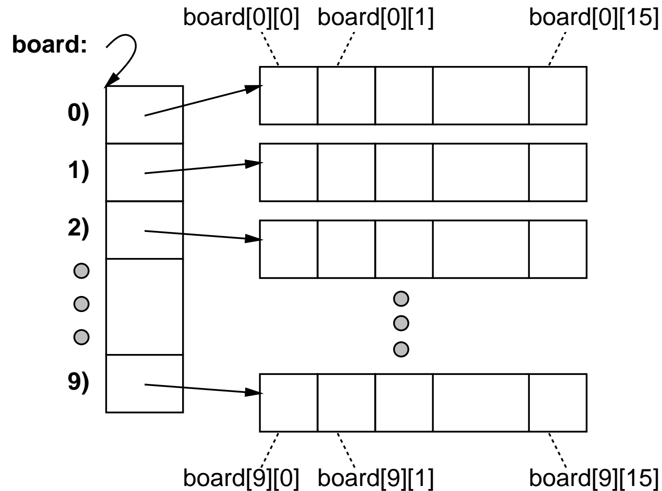


図 7: 2 次元配列

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。[35][36] 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、[37] 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。[38]

```
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));
```

35) 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。

36)「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。

37) 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。

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

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。

38)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。

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;
}
```

5　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。[39]

39)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。[40][41]

40)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。

41)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。

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。その実例は例題で繰り返し出て

来ました。[42] 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 ...
}
```

[42] 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。