Javaによる UI プログラミング: MICS 実験 2-19-1c
＃02 ユーザインタフェースイベントの扱い

久野　靖 *

2019.10.9

1　UI プログラミングとイベント

1.1　イベントの意味とその扱い方

前回までで窓に絵を表示することができるようになりましたが、それらの絵は一方的に表示されるだけで、こちらからマウスやキーボードなどを通じてそれらを操作することはできませんでした。今回はこの話題を扱います。

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** (input event) と呼びます。[1] 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**、event handler) 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 1)。

- イベントを受け取るための**アダプタオブジェクト** (adapter object) 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。[2]
- 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を定めているもの) が呼び出されるので、そこで処理を記述する。

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。[3] 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。

- **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`) — ボタンが押された
  - **mouseReleased**(`MouseEvent e`) — ボタンが離された

*電気通信大学

[1] 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。

[2] この章のプログラムは主となるクラスが `JPanel` のサブクラス、つまり窓の中の領域を表すクラスですから、そのコンストラクタやインスタンスメソッド中で単に `addMouseLisner()` 等を呼び出せば、自分自身つまりその領域内でのイベント受け取りの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。

[3] 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。

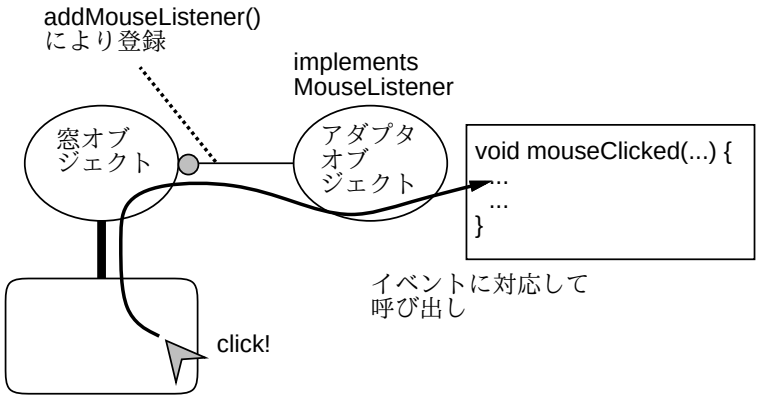

図 1: 入力イベントの受け取り

- **mouseClicked**(`MouseEvent e`) — クリックされた [4]
- **mouseEntered**(`MouseEvent e`) — ポインタが窓に入った
- **mouseExited**(`MouseEvent e`) — ポインタが窓から出た

- **MouseMotionListener** — 動きを伴うマウスイベントのインタフェース
  - **mouseMoved**(`MouseEvent e`) — 移動した
  - **mouseDragged**(`MouseEvent e`) — 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

[5]

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。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、窓にキーイベントが送られるようになります。[6] 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。

---

[4] 「押し」「離し」が対になって起こったという意味です。

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

[6] 実はキーイベントの取得にはきちんとした**フォーカス制御** (focus control) が必要なのですが、その説明をする紙面はとてもないので、本書ではこのようにしておきます。本来ならもっと「礼儀正しい」方法があることだけは覚えておいてください)。

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

この 5 行しかない、1 箇所でしか使わないクラスを別の場所で定義するのは繁雑なので、**無名クラス**にして addMouseMotionListener() 等の呼び出し位置に次のように書いてしまう方が簡単です (以下の例題でも基本的にそのようにしています)。[7]

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

1.2 例題: 円をドラグする

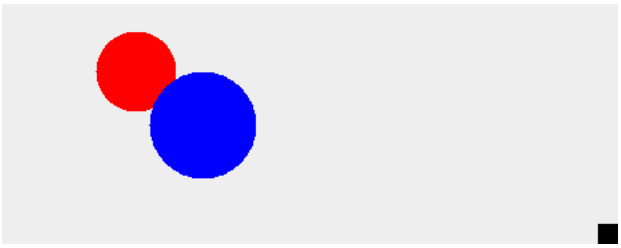

図 2: 円をドラグ

基本的な例として、円をドラグするという例題を見てみましょう (図 2[8])。まず本体部分から。

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

public class Sam21 extends JPanel {
  Circle c1 = new Circle(Color.RED, 100, 50, 30);
  Circle c2 = new Circle(Color.BLUE, 150, 90, 40);

  public Sam21() {
    setOpaque(false);
    addMouseMotionListener(new MouseMotionAdapter() {
      @Override
      public void mouseDragged(MouseEvent evt) {
        c1.moveTo(evt.getX(), evt.getY()); repaint();
      }
    });
  }
```

[7] このような書き方を**無名クラス** (anonymous class) と呼び、Java 言語の特徴的な構文の 1 つです。
[8] 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。

```
  public void paintComponent(Graphics g) {
    c1.draw(g); c2.draw(g);
  }
  public static void main(String[] args) {
    JFrame app = new JFrame();
    app.add(new Sam21());
    app.setSize(400, 300);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
  }
```

クラス Sam21 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、[9] (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、その登録のところで、MouseMotionAdapter のサブクラスであるような無名クラスを定義してインスタンスを生成しています。[10] 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。

```
  static 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 moveTo(int x, int y) { xpos = x; ypos = y; }
    public void setRad(int r) { rad = r; }
    public int getX() { return xpos; }
    public int getY() { return ypos; }
    public int getRad() { return rad; }
    public void draw(Graphics g) {
      g.setColor(col);
      g.fillOval(xpos-rad, ypos-rad, rad*2, rad*2);
    }
  }
} // 本体の終わりの「}」
```

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。[11] 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。

演習 1 例題 Sam21.java をそのまま打ち込んで動かしなさい。動いたら、次のように手直ししてみなさい。

[9] 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。

[10] 内部クラスの説明は本章の付録にありますが、このようにすることで Sam21 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。

[11] 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。また、後で演習のとき使うと思うので、半径の設定、XY 座標や半径の取得もできるようにしました。

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。
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、もっと変わった動きも実現してみるとなおよいです。
c. 例題では 1 ピクセル単位でドラグできますが、図形を扱うときはよく「グリッド単位で」位置決めをします。XY とも座標が必ず 20 の倍数 (もっと大きくしてみてもよい) になるようにしてみなさい。
d. 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動くようにしてみなさい。[12]
e. 円の位置を動かすのではなく、円の「ふち」をドラグして大きさを変化させるようにしてみなさい。できたら、中心付近を持ったら移動、ふち付近を持ったら大きさ変更にできるとなおよい。
f. 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。[13]
g. 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。[14]

2 人間の認知とインタフェースの動作

そもそもドラグは「引っ張る」という意味であり、人間の手の動作 (=マウスの動作) と画面上のオブジェクト (この場合は円) が「結びついて」感じられるから引っ張っているという感覚になるわけです。この部分はソフトウェアによって色々に変えられ、変えることによって「人間にとっての感じられ方」が変化します。実際に演習で体験してみましょう。

演習 2 先の円をドラグする例題をさらに次のように変更してみなさい。それを操作したときどのように感じるかも検討すること。

a. 例題では動き始めた瞬間に「中心が」ドラグ位置になりますが、これは「気持ち悪い」です (なぜか?)。そこで、「最初に掴んだ位置を保ったまま」ドラグできるように直してみなさい。
b. 例題では「マウスの X が増えると円の X 座標が増える」ようになっていたが、これを「マウスの X が増えると円の X 座標が減る」ようにしてみよ。Y についても同様にしてみよ。さらに両方同時にしてみよ。感じ方はどうか。
c. 今度は「マウスの X が増えると円の Y 座標が増える」「マウスの Y が増えると円の X 座標が増える」ではどうか。
d. 「X の 2 倍 (3 倍、等…) が円の X 座標になる」ではどうか。
e. 今まではマウスの座標が変化したら必ず円の座標も変化していた。そうではなく、「9 割の場合は変化するが残り 1 割はそのまま」にしたらどう感じるか。乱数が必要なら「`Math.random()`」で 0 以上 1 未満の一様乱数が得られる。

さまざまな「変化」でこれまでに出て来ていないものに「遅れ」が考えられます。遅れを実現するためには、「データを一定の間溜めておいてから有効にする」ため、キューを使うのが自然です。その例を示しておきましょう。

[12] 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。
[13] 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。
[14] `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。

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

public class Sam22 extends JPanel {
  Circle c1 = new Circle(Color.RED, 100, 50, 30);
  Circle c2 = new Circle(Color.BLUE, 150, 90, 40);
  Queue<Integer> qx = new ArrayDeque<Integer>();
  Queue<Integer> qy = new ArrayDeque<Integer>();

  public Sam22() {
    setOpaque(false);
    for(int i = 0; i < 10; ++i) { qx.add(100); qy.add(50); }
    addMouseMotionListener(new MouseMotionAdapter() {
      @Override
      public void mouseDragged(MouseEvent evt) {
        qx.add(evt.getX()); qy.add(evt.getY());
        c1.moveTo(qx.remove(), qy.remove()); repaint();
      }
    });
  }
  public void paintComponent(Graphics g) {
    c1.draw(g); c2.draw(g);
  }
  // main 同じ
  // Circle 同じ
}
```

すなわち、X 座標用と Y 座標用のキューを作って 10 個ずつ円の初期位置を押し込み、あとはマウスイベントごとに現在の位置をキューに入れてキューから取り出した XY 座標を円の位置としている。動かしてみると、確かに「遅れてついてくる」感じになる。

演習 3 キューによる遅れの例題をそのまま動かして観察せよ。遅れ (最初に入れておく数) がどれくらいまで操作できるか? 納得したら次のことをやってみよ。

a. 先にやった「左右反対」「XY 入れ換え」などと遅延を組み合わせるとどうなるか試してみる。
b. 遅延の量を一定でなく、徐々に大きくなるようにしてみる (同じ XY 座標を 2 回追加するなどする)。大きくなるだけでなく、小さくもなるようにしてみるとなおよい。
c. 座標の変換や遅延を使った面白い操作ができるインタフェースを実装してみよ。

3 インタフェースを用いた汎用のドラグ機能

ここまで見て来て「円を 1 つドラグするだけで大変なのに、沢山のものをドラグできるようにするとなるとひどく面倒そうだ」と思った人もいるかと思います。しかし実は、「ドラグできるもの」をうまく抽象化して扱うことで、どんな図形かとかに関知せず一括して扱うようにプログラムを書き、すべてを扱うことができます。

具体的には、次の例題ではプログラムが扱う図形をすべて、次のインタフェースを実装するクラスとして定義します。

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

つまり、「XY 座標を指定したとき、その座標が図形と当たっているかを調べる」メソッドと、あとは「位置 XY に動かす」「画面に描画する」メソッドを持つ、ということです。これだけ?と思うでしょうが、ドラグするだけならこれで十分なのです。実際に見てみましょう。

こんどは、図形 (上記の Figure インタフェースに従うオブジェクト) を多数扱えるようにするため、fset(Figure の集まり) という変数を用意し、ここに全ての図形を登録します。並びの後にあるものほど「上」に重なっているものとします。例題のコンストラクタではとりあえず円と長方形を 1 つずつ登録しました。

次に、マウスボタン押しの時の動作ですが、fset 中でマウスポインタの XY 座標に当たっているものを探し、target に入れます (複数あった場合は後のもの — 上に重なっているもの — が残ることに注意)。そして、target に入ったものがあれば、それを「一番上」にするため、いちど fset から削除して追加し直します (これで最後になる)。

マウスドラグの時は、target に入っているものがあるなら、その位置をマウスポインタの XY 座標にするだけです。これでドラグができます。以上です (簡単でしょう?)。

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

public class Sam23 extends JPanel {
  ArrayList<Figure> fset = new ArrayList<Figure>();
  Figure target = null;

  public Sam23() {
    setOpaque(false);
    fset.add(new Circle(Color.RED, 100, 50, 30));
    fset.add(new Rect(Color.GREEN, 150, 80, 40, 30));
    addMouseListener(new MouseAdapter() {
      @Override
      public void mousePressed(MouseEvent evt) {
        target = null;
        for(Figure f: fset) {
          if(f.hit(evt.getX(), evt.getY())) { target = f; }
        }
        if(target == null) { return; }
        fset.remove(target); fset.add(target); repaint();
      }
    });
```

```
    addMouseMotionListener(new MouseMotionAdapter() {
      @Override
      public void mouseDragged(MouseEvent evt) {
        if(target == null) { return; }
        target.moveTo(evt.getX(), evt.getY()); repaint();
      }
    });
  }
  public void paintComponent(Graphics g) {
    for(Figure f: fset) { f.draw(g); }
  }
```

main はこれまでと変わりません。Figure は先に説明した通り。

```
  public static void main(String[] args) {
    JFrame app = new JFrame();
    app.add(new Sam23());
    app.setSize(600, 600);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
  }
  interface Figure {
    public boolean hit(int x, int y);
    public void moveTo(int x, int y);
    public void draw(Graphics g);
  }
```

円ですが、さっきまでの円のクラスよりむしろ簡単で、ただし Figure インタフェースを実装することとし、そのために hit を実装しています。当たっているかどうかは、中心から渡された XY 座標までの距離が半径より小さいかどうかで分かります。

```
  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 (x-xpos)*(x-xpos) + (y-ypos)*(y-ypos) <= 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);
    }
  }
```

長方形も同様です。当たっているかどうかは X と Y それぞれが範囲内かどうかを見ることで実現できます。

```
  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);
    }
  }
} // 外側クラスの「}」
```

演習 4 上の例題をそのまま動かし動作を確認しなさい。納得したら、次のことをやってみなさい。

a. マウスボタンを図形の上でないところで押した場合は、新しい図形が現れ (特定のものに決めてもよいし、図形の種類や色や大きさを乱数で決めることにしてもよい)、それをドラグした状態なるようにしてみなさい。

b. この実装だと、ドラグを開始したとたんに「中央を持った」状態になる。それは気持ち悪いので、端っこを持った時でもその持った位置のままドラグできるようにしてみなさい。

c. 円や長方形以外の図形を追加してみなさい。

d. その他、ここまでに出て来た「さまざまな機能」を組み込んでみなさい。

4 付録: 名前のスコープと入れ子クラス

4.1 スコープの概念

一般にプログラミング言語では、**名前** (identifier — クラス名、メソッド名、変数名など) には**スコープ** (scope、**有効範囲**) 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 3 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。[15]

4.2 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 4)。最初にパッケージについて説明してしまいましょう。

[15] 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。

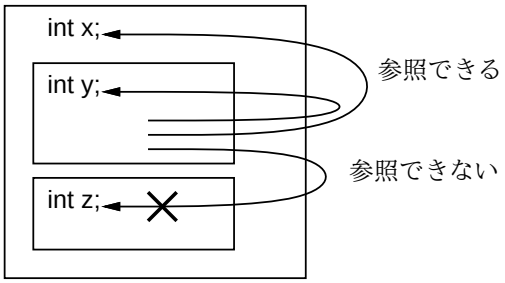

図 3: スコープの入れ子

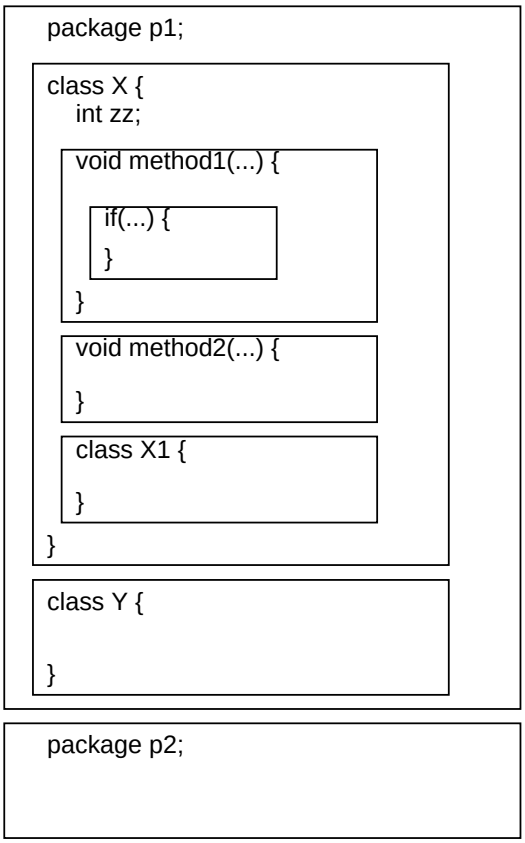

図 4: Java のスコープ単位

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。[16]

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 (フルクラス名で) 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** 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。

4.3 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**、nested class) 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。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。[17]

一方、static のついていない入れ子クラス [18] のオブジェクトは、外側クラスの特定のインスタンスに付属しています。このために、そのインスタンスメソッドやコンストラクタ中から外側クラスのインスタンス変数が参照できるのですが、その代わり、内部クラスのインスタンスを (これまでに知っているやり方で) 作ることは、外側クラスのインスタンスメソッドやコンストラクタの中からだけ可能です。[19]

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、「外側クラス名.this」と書くことになっています。

[16] 図 4 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。

[17] つまり、普通のクラスと同じものを作りたければ static をつけるべきなのです。

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

[19] 実はそれ以外からも可能ですが、外側クラスのオブジェクトを指定する必要があるため、「オブジェクト.new 内部クラス名 (…)」という見なれない書き方で new 演算子を使う必要があります。

4.4 局所変数とパラメタ

Java で一番スコープの狭い名前は局所数 (ローカル変数) とパラメタです。たとえば、図 5 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。これに対し、局所変数 z は内側のブロック [20] 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。

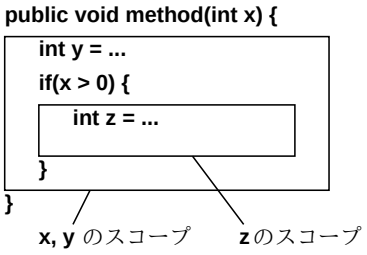

図 5: 局所変数のスコープ

このような規則になっているため、図 6(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 6(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 が参照できます。

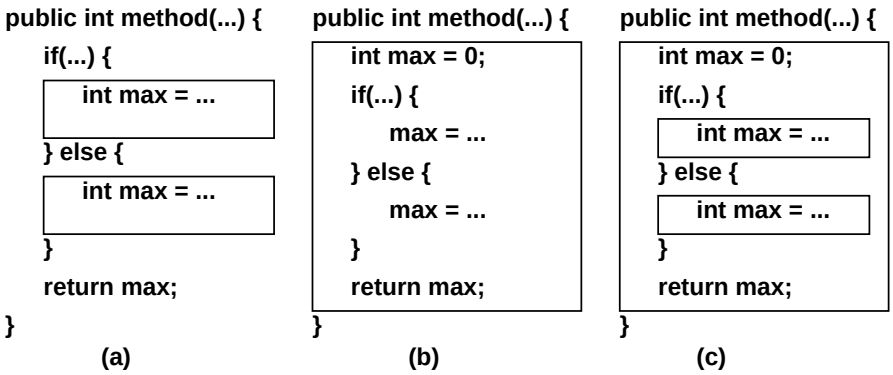

図 6: スコープが問題になりやすいコード

ところで (a) から (b) に直すときに、図 6(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。

[20] **ブロック** (block) とは、{ … } で囲まれたコードの範囲のことです。メソッドの本体も、while 文や if 文の本体も、ブロックになっています。