計算機プログラミングI 2005　久野クラス　＃11

久野　靖*

2006.1.13

はじめに

繰り返しお伝えしているように、次回 1/20 は授業休講、ただし TA さんがプログラム相談に来てくれますので、できるだけ次回までに冬休み課題プログラムの骨格を作成してきて、次回にまとめてバグ取りし、完成に近づけるようにしましょうね。

今回は前回予告したように、単独の窓を開く Java プログラムについてまず学び、関連して窓の大きさが変化したときに部品の位置を自動調整してくれる仕組みであるレイアウトマネージャについても説明します。その後が本題で、混み入った GUI 部品で有用な「モデルとインタフェースの分離」の考え方を説明し、具体的なプログラムで見てみましょう。

1　前回の演習問題の解説

前回の演習問題から、メロディをいろいろに加工する演算を定義する演習 3 をまとめてやってみる。メインクラスはその演算をいろいろ利用してみるという趣旨で、冒頭部分は元となるメロディの定義まで前回の例題と同じ。

```
import javax.sound.midi.*;

public class r10ex3 {
  static final int Q = 96;
  public static void main(String[] args) throws Exception {
    Sequencer seqr = MidiSystem.getSequencer(); seqr.open();
    Synthesizer synth = MidiSystem.getSynthesizer(); synth.open();
    seqr.getTransmitter().setReceiver(synth.getReceiver());
    Sequence seq = new Sequence(Sequence.PPQ, Q);
    Melody m1 = new Melody(100);
    m1.a(60,Q).a(62,Q).a(64,Q*2);
    m1.a(60,Q).a(62,Q).a(64,Q*2);
    m1.a(67,Q).a(64,Q).a(62,Q).a(60,Q).a(62,Q).a(64,Q).a(62,Q*2);
```

この後、いくつかの加工したメロディを配列に入れることにする。0 番目は元のまま、1 番目は倍の速さで 2 回演奏。2 番目はそれを前後逆にしたもの。3 番目はそれを「ミ」の音を中心として高低反転したもの。そして、メロディm2 は m1 の後ろに今の 0 番〜3 番からランダムに選んで 7 回連結したもの。これをピアノで演奏し、元のメロディを 2 オクターブ下げて 8 回反復したものをベースで演奏。あと、リズムセクションも加えてみた (チャネル 9 に設定すると打楽器になるのでしたね)。

```
    Melody[] m = new Melody[100];
    m[0] = m1;
    m[1] = m[0].scale(0.5).repeat(2);
    m[2] = m[1].reverse();
    m[3] = m[2].inverse(64);
```

*筑波大学大学院経営システム科学専攻

```
    Melody m2 = m1; // m[0], m[1], ...
    for(int i = 0; i < 7; ++i)
      m2 = m2.concat(m[(int)(Math.random()*4)]);
    m2.apply(seq.createTrack(), 1, 0, 95); //piano
    m1.raise(-24).repeat(8).apply(seq.createTrack(), 34, 1, 80); //bass
    Melody m3 = new Melody(100);
    m3.a(36,Q*3).a(36,Q).a(36,Q*4);
    m3.scale(0.5).repeat(32).apply(seq.createTrack(), 0, 9, 80);
    Melody m4 = new Melody(100);
    m4.a(44,Q).a(38,Q).a(38,Q).a(38,Q).a(44,Q).a(38,Q).a(38,Q).a(38,Q);
    m4.scale(0.5).repeat(32).apply(seq.createTrack(), 0, 9, 100);
    seqr.setTempoInBPM(160f); seqr.setSequence(seq); seqr.start();
    while(seqr.isRunning()) Thread.sleep(100);
    System.exit(0);
  }
```

ではメロディクラスの方を見てみよう。冒頭部分は前回と同じ。

```
  static class Melody {
    int note[], t0[], t1[], atime = 0, count = 0;
    public Melody(int n) { note=new int[n]; t0=new int[n]; t1=new int[n]; }
    private void add1(int n, int l) {
      if(count < note.length) {
        note[count] = n; t0[count] = atime; t1[count++] = atime = atime+l;
      }
    }
    public Melody a(int n, int l) { add1(n, l); return this; }
    public Melody p(int n, int l) { add1(n, l); atime -= l; return this; }
    public Melody s(int l) { add1(-9999, l); return this; }
    private void msg1(Track trk, int cmd, int ch, int v1, int v2, int t) {
      try {
        ShortMessage msg = new ShortMessage();
        msg.setMessage(cmd, ch, v1, v2); trk.add(new MidiEvent(msg, t));
      } catch(Exception e) { }
    }
    public void apply(Track trk, int pg, int ch, int v) {
      msg1(trk, ShortMessage.PROGRAM_CHANGE, ch, pg, 0, 0);
      for(int i = 0; i < count; ++i)
        if(note[i] >= 0) {
          msg1(trk, ShortMessage.NOTE_ON, ch, note[i], v, t0[i]);
          msg1(trk, ShortMessage.NOTE_OFF, ch, note[i], v, t1[i]);
        }
    }
```

音をずらすメソッド raise() は前回ちゃんと説明しなかったが、要するに新しいメロディを用意し、そこに自分のメロディを高さ d だけずらした状態で順次追加していくだけ。

```
    public Melody raise(int d) {
      Melody m = new Melody(count);
      for(int i=0; i<count; ++i) m.add1(note[i]+d, t1[i]-t0[i]);
      return m;
    }
```

2 つの音を連結する concat() は、raise() と同様だが、音の高さは変えず、新しいメロディに、まず自分のメロディ、続いて 2 番目のメロディを追加する。

```
public Melody concat(Melody n) {
  Melody m = new Melody(count + n.count);
  for(int i=0; i<count; ++i) m.add1(note[i], t1[i]-t0[i]);
  for(int i=0; i<n.count; ++i) m.add1(n.note[i], n.t1[i]-n.t0[i]);
  return m;
}
```

ここからが演習課題だが、まずメロディを N 回反復させる repeat() は、空のメロディに対し自分自身を N 回連結するだけ。

```
public Melody repeat(int n) {
  Melody m = new Melody(0);
  for(int i=0; i<n; ++i) m = m.concat(this);
  return m;
}
```

長さを変える scale() は、コピーするときに時間を r 倍するだけ。

```
public Melody scale(double r) {
  Melody m = new Melody(count);
  for(int i=0; i<count; ++i) m.add1(note[i], (int)(r*(t1[i]-t0[i])));
  return m;
}
```

前後逆にする reverse() は、コピーするときに末尾から順にするだけ。

```
public Melody reverse() {
  Melody m = new Melody(count);
  for(int i=count-1; i>=0; --i) m.add1(note[i], t1[i]-t0[i]);
  return m;
}
```

高低反転する inverse() は、単に計算で高さを変更するだけ、ただしマイナスの数値は「休符」なので別扱いする必要がある。

```
public Melody inverse(int b) {
  Melody m = new Melody(count);
  for(int i=0; i<count; ++i) {
    if(note[i] < 0) m.add1(-9999, t1[i]-t0[i]);
    else            m.add1(2*b-note[i], t1[i]-t0[i]);
  }
  return m;
}
}
}
```

なお、これらの演算はメロディが単音であることを前提にしているので、p() を使って作った和音はうまく扱えない (頑張ればできると思うので興味ある人はチャレンジしてみてはどうでしょうか)。

2　自前のウィンドウを開くプログラム

2.1　自前の窓の開き方と終わり方

これまでアプレットをやってきたのは、グラフィクスやGUI部品などを活用したいから、という理由が主だった (アプレットなら多くの人が互いに鑑賞できる、というのもあったけど)。しかしアプレットでなく普通のプログラムであっても、自分でウィンドウを開くようにすればそこでアプレットと同様にグラフィクスやGUI部品を利用することができる。また、アプレットにあるさまざまな制約がないので、次のことが可能になる。

- ファイルを読み書きできる。
- 任意の相手先にネットワーク接続を張ることができる。
- 手元のマシンのデバイス (サウンドボードとか) にアクセスできる。

というわけで、そのようなプログラムの作り方を説明しよう。

まず、「自前のウィンドウ」を作る機能はクラス JFrame が提供してくれている。ただし、JFrame は「何も中身がない」窓を作るようになっているので、そのサブクラスとして「自分なりの中身を持った窓」を用意する。つまり次のような感じになる。

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

public class XXX extends JFrame {
  public XXX() {
    // アプレットの場合なら init() に書くような初期設定...
  }
  // その他のメソッド (必要なら)...
}
```

なお、アプレットであれば初期設定は init() の中で行うが、これはアプレットではないので代りにコンストラクタの中で初期設定を行うようにする。

次に、このクラスのインスタンスを生成し、窓を開くための main() がどこかに必要である。main() はどこかのクラスに public static メソッドとして入れておけばよい。ここでは簡単のため、「どこかのクラス」としてこのウィンドウのクラスをそのまま利用してしまおう。

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

public class XXX extends JFrame {
  public XXX() {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // 自動終了
    setSize(幅, 高さ); // 窓の幅と高さはここで設定できる
    // アプレットの場合なら init() に書くような初期設定...
  }
  // その他のメソッド (必要なら)...
  public static void main(String[] args) { (new XXX()).setVisible(true); }
}
```

これで、「java XXX」のように実行開始するとまず main() が動き、XXX オブジェクト (というのは自分独自の窓) を生成して、それをメソッド setVisible() を呼び出して「見える状態にする」。

それはよいけど、このプログラムはいつ終るのだろう? これまでの普通のプログラムは main() が終ると終っていたが、今度は新しい窓を作ったため、main() が終っても窓が開いている限りプログラムは動き続ける。実は、このようなプログラムが終るためにはクラス System に備わっているクラスメソッド exit() を呼ぶ必要がある。あとそれ以外に、実はウィンドウに対してウィンドウマネージャが「閉じる」操作 (「×」アイコンのクリックとか、「窓の削除」機能に

対応) を実行した時に「自動的に終わる」ようにも設定できる。そのための指定が上の「自動終了」の行なのだった。では簡単な例題を見てみよう。

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

public class R11Sample1 extends JFrame {
  JScrollPane a1 = new JScrollPane(new JTextArea());
  JButton b1 = new JButton("Quit");

  public R11Sample1() {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); setSize(400, 300);
    Container c = getContentPane(); c.setLayout(null);
    c.add(a1); a1.setBounds(20, 20, 340, 200);
    c.add(b1); b1.setBounds(20, 240, 340, 40);
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) { System.exit(0); }
    });
  }
  public static void main(String[] args) {
    (new R11Sample1()).setVisible(true);
  }
}
```

このプログラムは、中央にスクロールバーつきのテキスト入力欄、下にボタンを配置し、ボタンが押されるとプログラムが終了するというだけものである (あと窓を閉じても終わる)。このようにしてこれまでのアプレットと同様の (ただしファイルの読み書き等ができる) プログラムは簡単に作成できるわけだ。

演習 1 これまでにアプレットとして作ったプログラムから好きなものを選んで、窓を開くアプリケーションに改造してみよ。

2.2 レイアウトマネージャ

ところで、上のプログラムは普通の窓を開くので、窓の大きさはマウス操作で自由に変更できる。しかしそうした場合、相当情けないことになる。つまり、アプレットでは領域の大きさは固定だったが、今度は窓の大きさに追従するようにしないと困るわけだ。

ここでいよいよ、これまで `setLayout(null)` でわざわざ無効にしてきた自動配置機能を使うことにしよう。Java ではこの自動配置機能をレイアウトマネージャと呼び、「縦横にます目に並べる」「一列に並べる」「上下左右と中央に置く」などさまざまな「並べ方」のレイアウトマネージャが用意されている。そして、特に指定しなければ (`setLayout(null)` をやらなければ)、`BorderLayout` という名前の「上下左右と中央に置く」レイアウトマネージャが動作するようになっている。

具体的には、`BorderLayout` では、ウィンドウ内に部品を配置するときに図 1 に示すように「東西南北または中央」と位置が指定でき、それぞれに指定された部品はその位置を占めるように自動的に配置してもらえる。

ではこれを利用して中央に `JTextArea`、その下にボタンを自動配置するように直したものを示す。

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

public class R11Sample2 extends JFrame {
  JScrollPane a1 = new JScrollPane(new JTextArea());
```

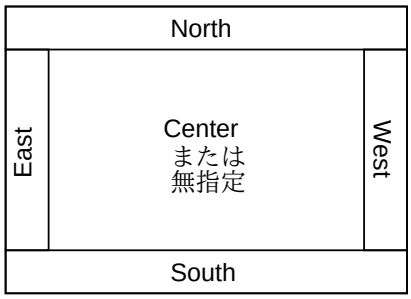

図 1: BorderLayout の指定方法

```
  JButton b1 = new JButton("Quit");

  public R11Sample2() {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    Container c = getContentPane();
    c.add(a1); c.add(b1, BorderLayout.SOUTH);
//  c.add(new JButton("Button 1"), BorderLayout.NORTH);
//  c.add(new JButton("Button 2"), BorderLayout.EAST);
//  c.add(new JButton("Button 3"), BorderLayout.WEST);
    pack(); setSize(400, 300);
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) { System.exit(0); }
    });
  }
  public static void main(String[] args) {
    (new R11Sample2()).setVisible(true);
  }
}
```

このプログラムでは、pack() で自動配置を開始した後、最後に窓の大きさを設定している (こうしないと各部品が入る最小の窓の大きさになってしまう)。また、コメントアウト部分を含めると上と左右にもボタンが現れ、BorderLayout の機能がよく分かる。

2.3 もうちょっと複雑な配置をしたいときは…

しかし、上の方法だとたとえば下側にボタンを 4 つ並べようと思ってもうまく行かない。そのような場合は、JPanel を「貼りつけ用部品」として利用する。すなわち、JPanel を作って、その上に「ボタンを順番に並べる」には今度は FlowLayout という単に順番にならべるだけのレイアウトマネージャを設定してからボタンを追加していく。JPanel 自体は外側のウィンドウの下に詰めればよい。

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

public class R11Sample3 extends JFrame {
  JScrollPane a1 = new JScrollPane(new JTextArea());
  JButton b1 = new JButton("Quit");
  JPanel p1 = new JPanel();

  public R11Sample3() {
```

```
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    Container c = getContentPane();
    c.add(a1); c.add(p1, BorderLayout.SOUTH);
    p1.setLayout(new FlowLayout());
    p1.add(b1);
    p1.add(new JButton("Button 1"));
    p1.add(new JButton("Button 2"));
    p1.add(new JButton("Button 3"));
    pack(); setSize(400, 300);
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) { System.exit(0); }
    });
  }
  public static void main(String[] args) {
    (new R11Sample3()).setVisible(true);
  }
}
```

このようにして、窓の大きさが変わっても配置が自動的に調整でき、しかもある程度自分で配置を制御できるようになったわけだ (一応は…)。なお、レイアウトマネージャはここで学んだ 2 つのほかにまだ数個ある。興味があったら API ドキュメントや、そこからリンクされているチュートリアルで調べてみるとよい。また、レイアウトマネージャ以外に JTabbedPane、JSplitPane などを使ってタブで切り替えたり画面を 2 分割するなどの部品も「いれものの配置」という点では似ているかも。

演習 2 これまでに作成したアプレットを単独窓に改造したものについて、部品配置をレイアウトマネージャで行うように手直ししてみよ。上に出て来た以外のレイアウトマネージャも使ってみるとよい。

3 モデルとビューの分離

3.1 JTable

JTree、JTable など Swing 部品群の多くで採り入れられている重要な概念として「モデルとビューの分離」がある。たとえば「表」というのは縦横のます目が並んだ「見た目」を持っているが、そのます目の中身というのはどのような形で保管されていても構わないはずである。もちろん、実際に M × N 個の要素が並んだ配列 (2 次元配列) を中に持っていてもいいのだが、用途によってはそうでない方が便利かも知れない。

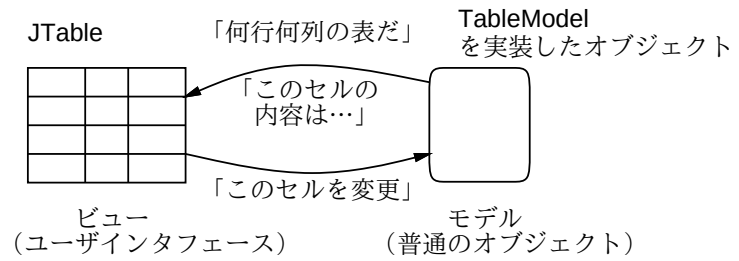

図 2: 表のビューとモデル

このため、JTable の場合、表の中身は TableModel というインタフェースを実装するオブジェクトとして表すことになっている (図 2)。そのオブジェクトは、たとえば次のようなメソッドを持つ。

- `int getRowCount()` — 表の行数を返す
- `int getColumnCount()` — 表の列数を返す
- `String getColumnName(`i`)` — i 列目の列名を文字列として返す

- boolean isCellEditable(r,c) — r 行 c 列目のセルをユーザが編集してもよいか否かを返す
- Object getValueAt(r,c) — r 行 c 列目のセルに対応するオブジェクトを返す
- void setValueAt(Object,r,c) — r 行 c 列目のセルの値として設定したいオブジェクトを渡して呼び出される

そのほかにも多数のメソッドがあって全部用意するのは大変なので、おなじみMouseAdapter同様、AbstractTableModelというクラスが用意されていて、そのサブクラスを作るようにすれば自分が再定義したいメソッドだけを書けば済む(getRowCount()、getColumnCount()、getValueAt()の3つは抽象メソッドなので必ず書く必要がある)。

以上をまとめると、表を使いたい時はAbstractTableModelのサブクラスとして自分の好きな情報を保持するクラスを作り、それをJTableのコンストラクタに渡して表を作ることで、データが表の形で表示され、またユーザが表に書き込む形でデータを更新することができる。具体例を見てみよう。

```
import javax.swing.*;
import javax.swing.table.*;

public class R11Sample4 extends JFrame {
  public R11Sample4() {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    getContentPane().add(new JScrollPane(new JTable(new MyTableModel())));
    pack(); setSize(400, 150);
  }
  public static void main(String[] args) {
    new R11Sample4().setVisible(true);
  }
  class MyTableModel extends AbstractTableModel {
    double[] rows = new double[]{1, 3, 5, 7, 11, 13, 17, 19, 23};
    double[] cols = new double[]{1, 2, 3, 4, 5, 6, 7, 8};
    public int getRowCount() { return rows.length; }
    public int getColumnCount() { return cols.length; }
    public boolean isCellEditable(int r, int c) {
      if(r == 0) return true; else return false;
    }
    public Object getValueAt(int r, int c) {
      return new Double(rows[r] * cols[c]);
    }
    public void setValueAt(Object o, int r, int c) {
      if(r == 0) {
        cols[c] = new Double(o.toString()).doubleValue();
      }
      fireTableDataChanged();
    }
  }
}
```

画面はJScrollPaneの内側に入れたJTableだけで、そのインスタンスを生成するときに後で定義するMyTableModelのインスタンスを渡しているので、このモデルを表示する表が画面に現れる。

さて、結局このプログラムの「中心」はMyTableModelクラスということになる。その中ではrowsとcolsという2つの配列があり、それぞれが縦方向と横方向の列に対応する数値を保持している。従って、getRowCount()、getColumCount()はそれぞれの配列のlengthを返せばよい。また、一番上の1行だけはユーザがセルを変更できることにしたので、isCellEditable()は行が0の時だけtrueを返す。そして、各セルに表示される値はrows[r]とcols[c]の積、つまりこの表は単なる「かけ算」の表。なお、getValueAt()は積の値を返すが、返すのは何らかのオブジェクトでなければならないので、Doubleクラスのインスタンスとして返す。

データが変更された時の処理は、変更されるのは 1 行目だけだが一応そのことをチェックし、渡されたものを文字列に変換し、double に変換し、そしてそれを cols[c] に入れる。つまりユーザが cols[c] の値を自由に変更できるわけだ。データを変更した場合は、表示全体が変わるので AbstractTableModel から継承している自分のメソッド fireTableDataChanged() というメソッドを呼ぶことで JTable 側で表示を変更してくれる (グラフィクスの repaint() のようなものと思えばよい)。

このように、「モデルとユーザインタフェース (表示部分) を分離する」という考え方は、プログラムの構造が整理される (半分ずつに分けると作成の労力はそれぞれが元の 1/4 くらいずつになると言われている)、1 つのモデル (計算処理) でユーザインタフェースだけ取り替えることができる、などの利点があるので、機会があれば活用してみることをお勧めする。具体的には次のように考えるとよい。

- 自分がしたい計算部分を「外からアクセスする」にはどのようなメソッドがあれば十分かを考え、それを Java のインタフェースとして定義してやる。
- 計算部分はそのインタフェースを implements するクラスとして作る。
- ユーザインタフェース部分はそのインタフェースを implements したオブジェクトを最初に受けとり、そのメソッドを呼び出しながら表示や入力を行うように作る。

演習 3 上の例題をそのまま動かせ。動いたら次のように変更してみよ。

(a) 上の例では「積」の表だったが、「和」や「差」など別の種類の演算の表に直してみよ。

(b) 上の例では一番上の列だけ変更可能だったが、加えて一番左の列も変更可能にしてみよ。ただし rows[0] は 1 のまま変更しないものとしてよい。

(c) 上に加えて、「任意の」位置が変更できるように直してみよ (その書き換えた位置の値から逆算して cols[i] または rows[i] を適切な値に直す)。

(d) 一番下左のセルを書き換えた場合、その書き換えた数値をもとに「その数値以下のすべての素数」が左の列に並ぶように表を変更するように直してみよ (もちろん、表の行数が変化することになる)。

(e) 住宅ローン計算プログラムを作れ。最初に借りた金額と、毎年の返済額と、年利をどこかのセルに入力させる。毎年、前年の債務残高から返済額を引き、債務残高に年利を書けた値を加えたものが翌年の債務残高になるわけ。

(f) 表を使って何か面白いプログラムを作れ。

3.2 DefaultTableModel

上のようにして自前でモデルを作るのはおおむねよい方法だが、単に値を入れておきたいだけの場合には「おまかせ」でただの 2 次元配列のようなものがあると便利である。そのために、javax.swing.table パッケージの中には DefaultTableModel というクラスが用意されている。これを利用して先の例題を手直しし、「九九の表」にしたものを見てみよう。

```
import javax.swing.*;
import javax.swing.table.*;

public class R11Sample5 extends JFrame {
  TableModel tab = new DefaultTableModel(9, 9);
  public R11Sample5() {
    for(int i = 1; i < 10; ++i) {
      for(int j = 1; j < 10; ++j) tab.setValueAt(new Integer(i*j), i-1, j-1);
    }
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    getContentPane().add(new JScrollPane(new JTable(tab)));
    pack(); setSize(400, 150);
  }
  public static void main(String[] args) {
```

```
    new R11Sample5().setVisible(true);
  }
}
```

非常に簡単だが、ただし表に記入するとその値が入ったままになる (当然)。なおこのクラスを extends して自分用のモデルを用意することもできる。

4 メニュー関係の機能

ところで、ここまでに出て来た例題は、窓はできるものの、普段使っているアプリケーションと比べてだいぶ「淋しい」ですよね? 普通のアプリケーションであれば、ボタンではなくメニューバーとメニューを使いますよね? 一応このあたりの機能も説明しておこう。

- JFrame のインスタンスメソッド setJMenuBar() で JMenuBar オブジェクトを設定することで窓にメニューバーがつけられる。
- JMenuBar に対してはインスタンスメソッド add() で JMenu オブジェクトを設定することで複数のプルダウンメニューが設定できる。
- JMenu に対してはインスタンスメソッド add() で JMenuItem オブジェクトを設定することでメニュー項目を追加して行ける。またインスタンスメソッド addSeparator() で「区切り線」を追加できる。
- JMenuItem オブジェクトに対しては押しボタンと同様、addActionListener() で動作を設定できる。
- JOptionPane というクラスのクラスメソッド showMessageDialog() を呼ぶことで簡単にダイアログボックスが表示させられる (その他の種類のダイアログも同様)。

なんだかよく分からないだろうが、ここまでの事柄を使った例題プログラムを見ていただこう。

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

public class R11Sample6 extends JFrame {
  JMenuBar mbar = new JMenuBar();
  JMenu menu0 = new JMenu("File");
  JMenuItem item0 = new JMenuItem("Red");
  JMenuItem item9 = new JMenuItem("Quit");
  JPanel panel = new JPanel();

  public R11Sample6() {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    menu0.add(item0); menu0.addSeparator(); menu0.add(item9);
    mbar.add(menu0); setJMenuBar(mbar); getContentPane().add(panel);
    pack(); setSize(400, 300);
    item0.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(R11Sample6.this, "paint red.");
        panel.setBackground(Color.red);
      }
    });
    item9.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) { System.exit(0); }
    });
  }
```

```
  public static void main(String[] args) {
    (new R11Sample6()).setVisible(true);
  }
}
```

上で説明したことが具体的にどう使われているか見ていこう。

- インスタンス変数ではメニューバー、その中のメニュー、そのまた中のメニュー項目2つ、およびパネル1つを保持している。
- メニューに項目1、区切り線、項目2を追加。
- メニューバーにメニューを追加。
- 窓にメニューバーを追加
- 「中身」にパネルを追加 (BorderLayout なので中央に)。
- 項目1が選ばれた場合の動作として、ダイアログウィンドウでメッセージを表示し、パネルの背景を赤に設定する。JOptionPane のクラスメソッド showMessageDialog() は第1引数として窓 (この場合は R12Sample1 のインスタンス) を渡す必要があるが、単に this と書くと一番内側のクラス、つまり無名内部クラスのインスタンスを意味してしまう。そこで外側クラスの this を指定したい場合は外側クラスの名前をつけて「R12Sample1.this」と指定する。
- 項目2が選ばれた場合の動作としてプログラムを終了するようにする。

演習 4 上の例題を打ち込んでそのまま動かせ。動いたら次のように直せ。面倒がらずに API ドキュメントも参照すること。

a. メニュー項目を増やし、もっと別の色も設定できるようにせよ。

b. a. と同様だが、File メニューとは違うメニューを増やし、そこで色が選べるようにする。(ヒント: `JMenu menu1 = new JMenu("Color");` などとしてインスタンス変数を増やし、これも mbar に追加する)。

c. 色を設定するとき「OK」ではなく「やめる」こともできるようにする。(ヒント: JOptionPane クラスの別のクラスメソッドを利用する。)

d. メニュー項目として JCheckBoxMenuItem を使い、確認ダイアログの出る/出ないを制御できるようにする。(ヒント: 前記のメニュー項目はメソッド getState() を呼ぶと ON か OFF か調べられる。)

A 本日の課題 11A

演習 3〜演習 4(の小課題1つ) から1つ以上選び、プログラムを作成すること。また、そのプログラムのコードはいつも通り、「本日中に」久野までメールで送付してください。具体的な内容は次の通り。

1. Subject: は「Report 11A」とする。
2. 学籍番号、氏名、投稿日時を書く。
3. 選んだプログラム1つのソース。
4. その簡単な説明。
5. 下記のアンケートの回答。

   Q1. レイアウトマネージャの機能、メニューバーの作り方など、だいたい分かりましたか。

   Q2. JTable のように部品 (画面) とその内部状態 (モデル) を分離するという考え方について納得しましたか。

   Q3. その他冬休み課題の状況、感想、要望などどうぞ。