Java による UI プログラミング: MICS 実験 2-19-1c
04 GUI 部品とその扱い

久野　靖 *

2019.10.16

1　GUI と GUI 部品

1.1　基本的な GUI 部品の配置

GUI(Graphical User Interface) とは、コンピュータの画面に図形や絵などが表示され、それらをマウス等で操作するようなユーザインタフェースを言います。[1]　今日のコンピュータでユーザが使うプログラムは大部分が GUI を持つプログラムですから、本書でもこちらを中心に扱っています。

多くの GUI プログラムでは、ボタン、メニュー、入力欄など決まった形の部品が画面に表示され、それらを通じてプログラムを操作します。このようなものを一般に **GUI 部品** (GUI controls) と呼びます。

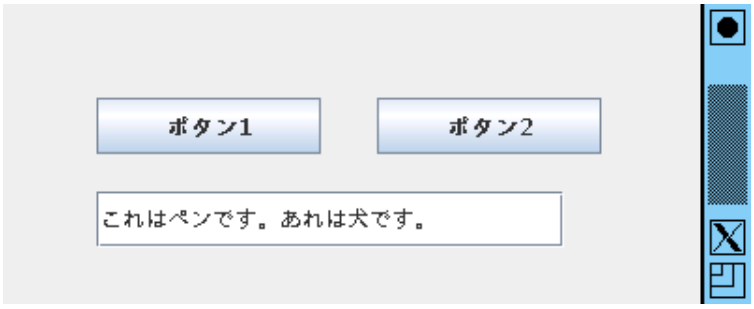

図 1: ボタンと入力欄を窓に配置

オブジェクト指向では、GUI 部品の 1 つひとつをオブジェクトとして扱うのが自然です。クラス方式の言語であれば、部品の種類ごとにクラスがあり、そのインスタンスを画面に配置します。[2]

さっそく、図 1 のように GUI 部品を配置する例を見てみましょう。

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

public class Sam41 extends JPanel {
  JButton b1 = new JButton("ボタン 1");
  JButton b2 = new JButton("ボタン 2");
  JTextField f1 = new JTextField();
```

*電気通信大学

[1]この対義語として、画面には文字だけが表示され、キーボードの文字入力で操作する **CUI** (Character User Interface) があります。ただし GUI でもキーボードは当然使います。

[2]以下では Java 標準ライブラリに含まれる **Swing** という部品群を使いますが、そこでは上で挙げた**ボタン** (push button) はクラス **JButton**、**入力欄** (input field) はクラス **JTextField** に対応しています。

```
  public Sam41() {
    setLayout(null);
    add(b1); b1.setBounds(50, 50, 120, 30);
    add(b2); b2.setBounds(200, 50, 120, 30);
    add(f1); f1.setBounds(50, 100, 250, 30);
  }
  public static void main(String[] args) {
    JFrame app = new JFrame();
    app.add(new Sam41());
    app.setSize(400, 300);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
  }
}
```

前回までと比較して、今回は短いプログラムです。これは、さまざまな GUI 部品はライブラリ中にあって、自分で定義しなくてもいいことと、その画面への表示もユーザからの入力も自動的に行われることによります。このようになっているため、`paintComponent()` や入力のイベントハンドラを書く必要がありません。[3]

では実際に見てみましょう。まずインスタンス変数ですが、この例題では GUI 部品のボタンを 2 つと入力欄を 1 つ生成し、インスタンス変数に入れています。その後はコンストラクタでの初期設定ですが、その内容は次の通りです。

- 部品の自動配置機能をオフにする
- それぞれの部品をメソッド **add()** で領域に追加する。
- それぞれの部品が持つメソッド **setBounds()** でそれぞれの部品の XY 座標、幅、高さを指定することで、部品の配置を定める。

`add()` は `JPanel` から継承してきたメソッドであり、[4]`setBounds()` は各 GUI 部品が共通に持つ親クラス **Component** から継承してきたメソッドです。ここに示した以外ものでも、Swing の GUI 部品はどれも同じようにして画面に追加し、配置できます。

演習 1 `Sam41.java` をそのまま打ち込んで動かしなさい。うまく動いたら、次のような改造をしてみなさい。

a. ボタンや入力欄の位置を変化させたり、もっと個数を増やす。

b. 各部品は親クラスから継承したメソッド `setForeground()` で文字の色、`setBackground()` で背景 (地) の色を設定できます (パラメタは `Color` オブジェクト)。適当な部品の文字や地の色を設定してみなさい。

c. API ドキュメントで `javax.swing` パッケージ中にある GUI 部品のクラスを確認し、興味を持ったものを画面に入れてみなさい。たとえば、`JLabel`、`JCheckBox`、`JRadioButton`、`JComboBox`、`JList`、`JTextArea`、`JSlider` などを試すといいでしょう。

c について少し補足しておきましょう。`JRadioButton` については、複数のボタンをグループに入れてどれか 1 つを選ぶと他のものが選択解除されるようにするのが普通なので、`ButtonGroup` を併せて

[3] さらに、ユーザの入力操作に対する応答も GUI 部品が自分で行います。たとえばボタンを押すと押されたことを表すように画面が変化しますし、入力欄を選択すると文字列を打ち込むことができます。

[4] 正確には `JPanel` がさらにその親クラスの **Container** から継承してきています。

使ってみてください。JComboBox はコンストラクタで生成するときに、選択する項目を文字列の配列で渡すとよいでしょう。JList と JTextArea は項目数が多くなったときにスクロールバーを出す使い方が普通です。このため、この部品をさらに「中身をスクロール可能にする」部品である JScrollPane と組み合わせ、「new JScrollPane(内側の部品)」のようにして使うといいでしょう (画面に配置するのは JScrollPane の方になります)。

1.2 GUI のデザイン

ここまでで GUI 部品を画面に配置できるようになりましたが、実際に「使いやすい」GUI をデザインするためには、以下のことがらに対する配慮が必要です。

- どのような機能を持つプログラムなのかが明確である。
- 提供されている各機能を使おうとした時に、どのように操作すればよいのかが明確である。
- 各機能を使う場合の操作が容易である。

最後の「容易である」というのにも、さまざまな基準があります。

- 操作に要する時間が短くて済む。
- 操作の方法が理解しやすい。
- 操作に慣れるのに掛かる時間が短い。
- 操作を行った時の疲労が少ない。

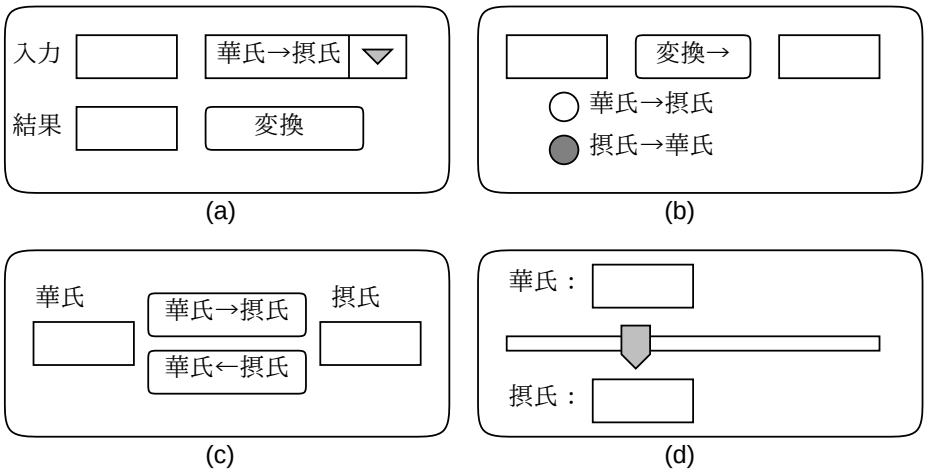

図 2: 温度変換プログラムの GUI の案

実際には、これらの基準の間には**トレードオフ** (trade-off) があります。たとえば、慣れた時に非常に高速に操作できるようなインタフェースは、操作方法を理解したり慣れるのに掛かる時間は長くなりがちです。逆に、1 ステップずつ導いてくれるようなインタフェースは、理解はしやすいけれども時間が掛かりがちです。ですから、GUI を設計する際には、これらの基準のどれを重視するかをそのつど考えた上で、どの設計を選ぶかを判断する必要があります。

たとえば、「摂氏の温度と華氏の温度を相互に変換するプログラムの GUI を設計する」という問題を考えてみましょう (図 2)。

まず、(a) 案のように「摂氏→華氏」「華氏→摂氏」のどちらかを選択メニューで選択してもらい、入力欄に数値を打ち込んでもらって「変換」ボタンを押す、という方法が考えられます。しかし、選

択メニューはメニューを出してみないとどのような選択肢があるか分からないので、2 つしか項目がない今回の場合はあまりよくありません。[5]

そこで、(b) 案のように 2 つの変換をラジオボタンで選択する案が考えられます。しかしこれでも、(1) 数値を入れて、(2) ラジオボタンを選択して、(3) 変換ボタン、という 3 ステップになります。そこで、(c) 案のように、変換ボタンを 2 つにして、どちらを押すかでどちら向きの変換かを選ぶようにすれば、2 ステップになります。さらに、左の入力欄は華氏、右の入力欄は摂氏のようにしておけば、数値を入れ終わった瞬間に変換がなされて他方の欄に対応する値が出るようにできるかも知れません。

さらに、(d) 案のように数値は入力せず、スライダーを動かすとその値に応じた摂氏と華氏の値が表示されるようにすることも考えられます。(d) 案は一見すばらしそうですが、実際に作ってみると正確な値をスライダーで設定するのは結構手間が掛かります。結局、変換の向きがしばしば変わる場合は (c) 案、いちど摂氏→華氏の変換をしたらしばらく続けて同じ変換をするなら (b) 案がよさそうです。

GUI のデザインは、単に動けばいいわけではなく、多くのを考慮し、よい設計を追求する必要があることが、お分かり頂けたでしょうか。

演習 2 図 2 のうち、気に入った案の画面 (ないし、自分で考案した別の案) を Java のプログラムで作ってみなさい。ちなみにまだ動作はしなくてよいが、摂氏と華氏の変換式は次の通り。

$$C = (F - 32) \times \frac{5}{9} \qquad F = C \times \frac{9}{5} + 32$$

演習 3 次のようなプログラムの GUI デザインを、方眼紙の上に描いてみなさい。最低でも 2 つの案を出して比較検討すること。

a. 整数を入力して、素数かどうかを判定してもらうプログラム。
b. 2 つの数値を入力して、その最大公約数と最小公倍数を計算させるプログラム。
c. 簡単な (四則程度の) 計算をおこなう電卓プログラム。
d. その他、自分が作ってみたいプログラム。

1.3 GUI 部品に動作をつける

ここまでで、画面に GUI 部品は配置しましたが、ボタン等を押しても何も起きませんでした。[6] そこで次は、GUI 部品に動作をつけてみましょう。GUI プログラムでユーザがボタンを押したりスライダを動かしたりするなどの動作は**入力イベント**と呼ぶのでしたね。[7] そして、イベント (たとえばボタン押し) に対して何か動作をつけたければ、そのイベントを受け取るための**イベントハンドラ**を持った**アダプタオブジェクト**を作り、予め登録しておく必要があるのでした。

実は、そのやり方は既に学んだもので、`javax.swing.Timer` に動作を渡すのに使ったのと同じ方法を使います。つまり、**ActionListener** インタフェースを実装するアダプタオブジェクトを作り、その中のメソッド **actionPerformed()** として動作を記述します。最後に、ボタンオブジェクトに対してメソッド **addActionListener()** を使ってアダプタを登録すれば、以後ボタンが押されるごとに記述した動作が実行されるようになります。

では「数が素数かどうか判定する」という例題を見てみましょう (図 3)。

[5] 項目数が多い場合は、常に選択肢が出ていると画面上で邪魔になるので、選択する時だけ項目が表示される選択メニューが適しています。

[6] 動作をプログラムしていないのですから、当り前ですね!

[7] ここまででは、マウスを動かしたりキーを打鍵することを入力イベントと説明しましたが、その結果として GUI 部品を操作することもそれに含まれるのでした。

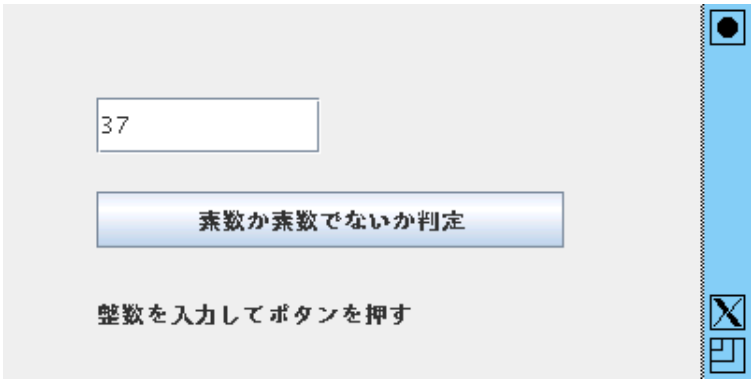

図 3: 素数判定プログラム

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

public class Sam42 extends JPanel {
  JTextField f1 = new JTextField();
  JButton b1 = new JButton("素数か素数でないか判定");
  JLabel l1 = new JLabel("整数を入力してボタンを押す");
  public Sam42() {
    setLayout(null);
    add(f1); f1.setBounds(50, 50, 120, 30);
    add(b1); b1.setBounds(50, 100, 250, 30);
    add(l1); l1.setBounds(50, 150, 250, 30);
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        int n = Integer.parseInt(f1.getText());
        boolean prime = true;
        for(int i = 2; i < n; ++i) {
          if(n % i == 0) { prime = false; }
        }
        l1.setText(n + (prime ? "は素数。" : "は合成数。"));
        f1.setText("");
      }
    });
  }
  public static void main(String[] args) {
    JFrame app = new JFrame();
    app.add(new Sam42());
    app.setSize(400, 300);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
  }
}
```

コンストラクタ中のでGUI部品を配置しているところは、これまでと同様です。[8] 続いて、ボタンb1のaddActionListener()を呼び出して、ボタンの動作を設定します。ここで設定するオブジェクトは、ActionListenerインタフェースを実装する無名クラスのインスタンスであり、その中に定義されたメソッドactionPerformed()が、ボタンが押された時に実行されます。

では、メソッドの中の動作を見てみましょう。

- 入力欄の文字列を**getText()**で取得し、Integerのクラスメソッド**parseInt()**を使って整数値に変換し、変数nに入れる。
- 論理型の変数primeを用意し、初期値として「はい」を入れておく。
- 整数iを2からnの手前まで変化させながら、nをiで割った剰余が0ならprimeを「いいえ」に変更する。[9]
- 最後に、primeの値に応じてラベルに「素数です」「素数ではありません」のどちらかの表示を出させ、入力欄は空に戻す。

primeのような使い方の変数のことを**フラグ** (flag、**旗**) と呼びます。つまり、最初は「その数は素数だ」という印 (旗) を立てておき、素数かどうかチェックし、どこかで割り切れて素数でないと分かったら旗を降ろします。最後まで来て旗が立ったままなら、どの数でも割り切れなかったわけなので、素数だと分かるわけです。

元の話題に戻ると、ボタン押しに対応する動作をつける場合は、アダプタオブジェクトが持つメソッドactionPerformed()の中に、必要な動作を記述すればいいわけです。

演習 4 上の例題を打ち込んで動かしなさい。動いたら、次のような簡単なGUIプログラム (動作つき) を作成しなさい。[10]

a. 2つの入力欄に数値を入力し、ボタンを押すとそれらの数値の和を表示する。

b. 1つの入力欄に「1」と表示されていて、「Add」ボタンを押すとその値が1増え、「Sub」ボタンを押すとその値が1減る。

c. ボタンを押すたびに表示欄にランダムに1から6までの整数が表示される。つまりサイコロの代わりをする。[11]

d. 10秒間に何回ボタンが押せるかを競うゲーム。

演習 5 これまでに画面を作成したGUIプログラムについて、ボタンを押した時の動作をつけてみなさい。

2 メニューバーとタブ

ここまで、1枚のパネルの中に色々なGUI部品を入れて来ましたが、皆さんが通常のアプリケーションを使っている場合は窓の上部にメニューバーがあって機能を起動できたり、窓の中にタブがあって画面を切替えられたりしますね? そのような「枠」の部分についても簡単に取り上げましょう。

まず、窓 (JFrame) にはsetMenuBar()というメソッドがあり、これを使ってMenuBarオブジェクトを設定できます。そして、1つのMenuBarオブジェクトにはいくつでもMenuオブジェクトを追加できます (その1つずつがプルダウンメニューになります)。Menuオブジェクトにはいくつでも MenuItem

[8] 入力欄とボタンと、あと結果表示用にラベルを配置しています。

[9] 素数とは、1とその数自身でしか割り切れないような整数のことですから、2〜n-1のどれかで割り切れたら素数ではないわけです。

[10] いずれも、数値を文字列に変換して表示させるためには、空文字列""と数値を「+」を使って連結させるとよいでしょう。

[11] 1〜6のランダムな値は、「1 + (int)(6*Math. random())」で得るとよいでしょう。これは「0以上1未満の一様乱数を6倍して整数に切捨て、それに1を足す」ものです。

オブジェクトが追加できます。これがメニュー項目です。そして、MenuItem オブジェクトに対して addActionListener() でイベントハンドラをつけることで、そのメニュー項目が選択された時の動作を指定できます。

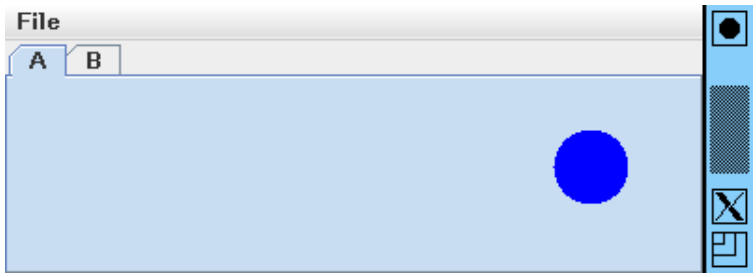

図 4: メニューとタブの例題

次の例題 (図 4 では、外側クラスは JFrame の子クラスになっていて、そのコンストラクタでメニューバーの設定、File メニューの設定、そしてその中の Quit 項目の設定、選ばれた時にプログラムを終了する動作の設定をおこなっています。

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

public class Sam43 extends JFrame {
  int xpos = 100, ypos = 100;
  public Sam43() {
    JMenuBar mbar = new JMenuBar(); setJMenuBar(mbar);
    JMenu m1 = new JMenu("File"); mbar.add(m1);
    JMenuItem i1 = new JMenuItem("Quit"); m1.add(i1);
    i1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        System.exit(0);
      }
    });
    JTabbedPane tabs = new JTabbedPane(); add(tabs);
    tabs.add("A", new Panel1());
    tabs.add("B", new Panel2());
  }
```

窓の中身が何もないとつまらないので、続いて窓の中にタブ切替え領域を入れ、そこに「A」「B」という 2 つのタブを設定しました。設定する内容は Compnent の任意のサブクラスですが、ここでは円を表示する JPanel のサブクラスを 2 つ作って入れています。

その内容ですが、Panel1 はインスタンス変数 xpos、ypos の位置に円を表示するだけです。Panel2 はそれに加えて、マウスドラグイベントを受け取り、円の位置を移動するようにしています。そうすると、「B」タブを選んでドラグすると円の位置が移動できますが、「A」に戻ったときはそちらの円も同様に (同じだとつまらないので少しずらして) 移動しています。

```
  class Panel1 extends JPanel {
    public void paintComponent(Graphics g) {
      g.setColor(Color.BLUE);
```

```
      g.fillOval(xpos-20, ypos, 40, 40);
    }
  }
  class Panel2 extends JPanel {
    public Panel2() {
      setOpaque(false);
      addMouseMotionListener(new MouseAdapter() {
        public void mouseDragged(MouseEvent evt) {
          xpos = evt.getX()-20; ypos = evt.getY()-20; repaint();
        }
      });
    }
    public void paintComponent(Graphics g) {
      g.setColor(Color.RED);
      g.fillOval(xpos, ypos, 40, 40);
    }
  }
  public static void main(String[] args) {
    JFrame app = new Sam43();
    app.setSize(400, 300);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
  }
} // 外側クラスの「}」
```

演習 6 例題をそのまま動かしなさい。動いたらループを使って「File メニューが 10 個ありその中に Quit が 10 個ずつあるが、本当に終了できるのは 1 つだけ」というイタズラアプリを作ってみなさい。

演習 7 タブの「空けてみるまで内容が見えない」という性質を利用してゲームを作ってみなさい。たとえば円をドラグして数値を-100〜100 の範囲で変更できるが、その数値は別のタブに行かないと表示されない、というふうにして「できるだけ速く 0 に設定する」ことを競うなど。

演習 8 タブは一時に 1 つの面しか見えないが、`JSplitPane` という部品を使うと 2 つの面が同時に見える (境界線を移動してサイズの調整もできる)、という機能になる。API ドキュメントで使い方を調べてこちらの方に変更してみよ。そのとき、赤い円をドラグしたら青い円もついてくるようにするにはどうしたらよいか検討し実装しなさい。

演習 9 GUI 部品を操作するのにはそれぞれ一定の法則にしたがって時間が掛かるが、その時間を検討してみなさい。いくつかの GUI 部品を決まった順番で操作し、完了すると時間が分かるようなアプリケーションを作るとなおよいでしょう。

3 付録: 例外処理で例外的な場合に対処する

3.1 例外処理について

たとえば例題の素数判定プログラムに対して「aa」を入れてボタンを押してみると、プログラムを起動した窓に次のようなものが表示されました。

```
Exception in thread "AWT-EventQueue-0" java.lang.NumberFormat
Exception: For input string: "aa"
    at java.lang.NumberFormatException.forInputString(Number
FormatException.java:48)
    at java.lang.Integer.parseInt(Integer.java:449)
    at java.lang.Integer.parseInt(Integer.java:499)
    at Sam42$1.actionPerformed(Sam42.java:16)
    ...                          ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑
```

冒頭を見ると「入力文字列"aa"に対する NumberFormatException」と書かれています。その少し下を見ると、`Integer.parseInt` と表示されています。そしてさらに少し下の「↑」をつけたところを見ると、プログラム中の `Integer.parseInt` を呼び出しているところです。ですから、文字列「`"aa"`」を整数に変換しようとしたところで駄目だと言われているらしいことが分かります。

実は Java では、データ等に対する操作が想定外のものであるときはその種類に応じた**例外** (exception) と呼ばれるものが発生します。そして、発生した例外をとくに取り扱わなければ、上の例のようにプログラムを起動した箇所にエラーが表示されます。[12]

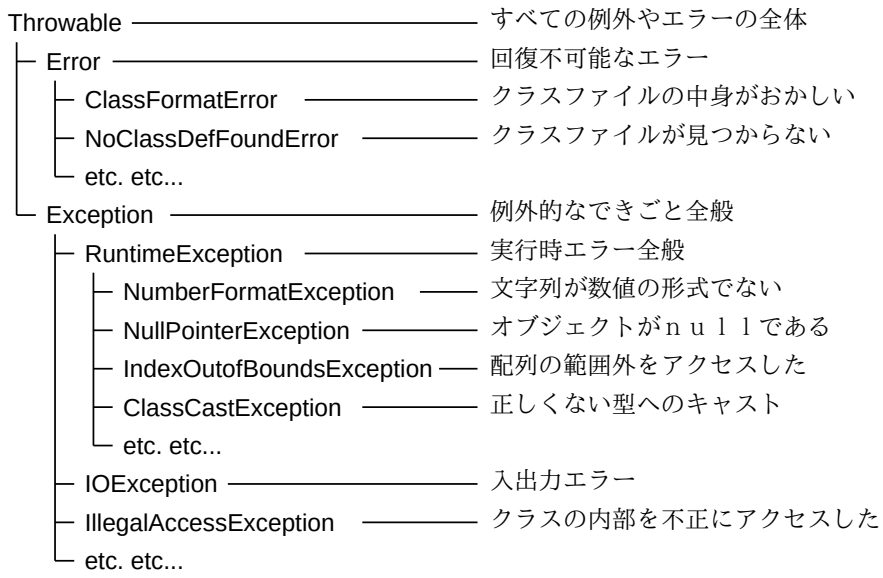

図 5: Java の例外クラスの階層

Java では例外は一群のクラスによって表現され、クラスの継承関係によって分類されています。代表的な例外クラスを図 5 に示します。すべての例外クラスは `Throwable` というクラスの子孫になっています。そして先の例ではこれらのうちの、`NumberFormatException`(文字列が数値の形式になってない) が発生しているわけです。

しかし、ユーザがついうっかり「aa」のようなものを打ち込むことはよくあるので、そのたびに長いエラーメッセージが表示されるのは考えものです。このような場合には、例外を受け止めて処理することで、自前で適切な処理を行うことができます。例外を受け止めて処理するためには、次の形をした **try 文** (try statement) を使います。

```
try {
  ... 例外が発生する可能性のある処理 ...
} catch(例外クラス名 変数) {
  ... 例外が発生した場合の処理 ...
```

[12] プログラムの実行は続けられる場合も停止される場合もあります。

```
}
```

これは、図 6 のように働きます。まず図の左に示してあるように、try の後にある部分で例外が発生したときそれが分類で catch 部に書かれたクラス XXX 以下のものであれば、その catch 部に実行が移り、そこで処理した後でその下に実行が続きます。

発生した例外が XXX 以下のものでなければ、図の右にあるように、その外側にある YYY の catch 部に (その例外が YYY 以下の分類ならば) 移ります。また、内側の catch 部の中で発生した例外も同様です。そして、どこでも catch されなかった例外については、これまで通りエラーメッセージが表示されるわけです。

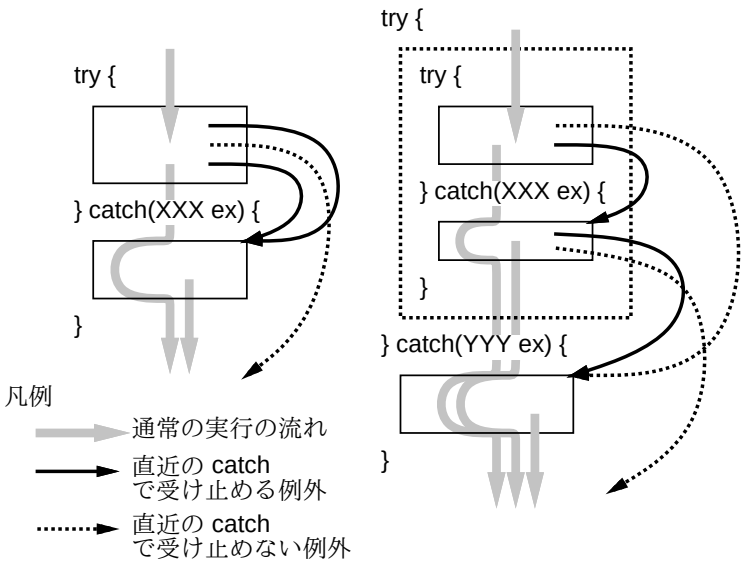

図 6: try-catch の構造

では、具体的には XXX や YYY はどのように指定したらいいでしょうか。大規模なプログラムでは例外の受け止めを細かく設計することもありますが、ここでは基本的に、分類 Exception 以下を受け止めて「何かまずいことがあった」という表示をしてから先に進む、という方針を採用しておきます。

3.2 例題に例外処理を追加する

では、先の素数判定の例題に例外処理を追加しましょう。基本的には、これまでの処理の中身全体を try-catch で囲み、Exception 以下全部を受け止めます。そして、受け止めたところの処理はラベルに「問題があったのでやり直してください」と表示するわけです (図 7)。

```
public class Sam42b extends JPanel {
  JTextField f1 = new JTextField();
  JButton b1 = new JButton("素数か素数でないか判定");
  JLabel l1 = new JLabel("整数を入力してボタンを押す");
  public Sam42b() {
    setLayout(null);
    add(f1); f1.setBounds(50, 50, 120, 30);
    add(b1); b1.setBounds(50, 100, 250, 30);
    add(l1); l1.setBounds(50, 150, 250, 30);
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
```

```
        try {
          int n = Integer.parseInt(f1.getText());
          boolean prime = true;
          for(int i = 2; i < n; ++i) {
            if(n % i == 0) { prime = false; }
          }
          l1.setText(n + (prime ? "は素数。" : "は合成数。"));
          f1.setText("");
        } catch(Exception ex) {
          l1.setText("問題がありました。再度どうぞ。");
        }
      }
    });
  }
```

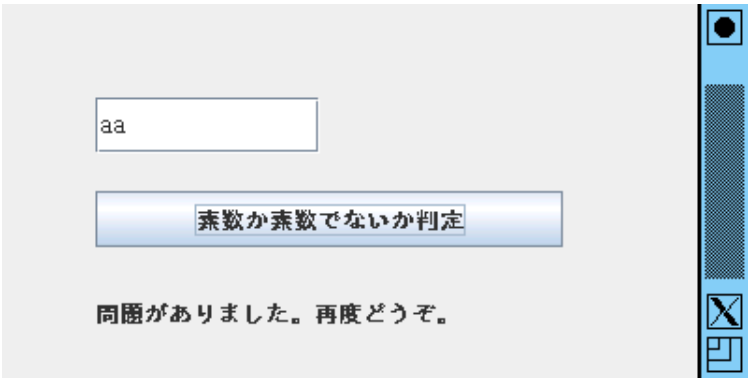

図 7: 素数判定プログラムで例外に対処

3.3 例外処理に関する補足説明

例外処理にはさまざまな考慮点があるので、本文で説明できなかったことを、ここでもう少し説明しておきます。

例外を投げる

まず、ここまでででは例外を受け止めて処理することだけ取り上げて来ましたが、自分で (何か処理を記述していて例外的な事態が起きたときに) 例外を発生させることもあります (「投げる」とも言います)。この場合には、**throw 文** (throw statement) を使用します。典型的な使い方は次のようになります。

```
throw new RuntimeException("xxx is wrong.");
```

throw 文では式を 1 つ指定しますが、その式の結果は必ず例外クラス (`Throwable` のサブクラス) のインスタンスでなければなりません。自前の例外クラスを定義してもよいのですが、既存の例外クラスに例外の種類をよく表すものがあれば、そのクラスを使っても構いません。

上の例では `new` 演算子で `RuntimeException` のインスタンスを生成していますが、だいたいの例外クラスは文字列を 1 つ受け取るコンストラクタを持っていて、ここに「何がまずいか」を表す説明を書くことができます。

そして、catch で受け止めた例外オブジェクトを (表示などのために)toString() で文字列に変換すると、例外の種別とこの説明文字列がつながった文字列が返されるので、それを表示しておくことでとりあえず「何がまずかったか」をユーザに示すことができるわけです。[13]

3.4 例外の宣言とコンパイラのチェック

Java では、各メソッドについて、それがどのような例外を発生するかを予め、メソッドの冒頭で宣言するようになっています。たとえば次のような具合です。

```
public void someMethod(...) throws IOException {
```

API ドキュメントを見ると、このような **throws 宣言**がついたメソッドが多数あることが分かります。自分が書くメソッドの場合も、その中で例外を投げる場合には、その例外を上記の形で予め宣言する必要があるわけです。

一方、このように例外を宣言したメソッドを呼ぶ側では、例外を処理する方法として次の 2 つがあります。

- そのメソッドを呼んでいるコードの範囲を try 文で囲んでその例外を受け止めて処理する。
- 自分のメソッドにも同じ throws 宣言をつけることで、発生した例外を自分を呼び出したところに伝播させて処理してもらう。

そして、この少なくとも片方を行う必要があります。そうしないと、発生した例外を責任を持って処理するところがないので困ることになるからです。Java プログラムをコンパイルしていて「例外○○が処理されていない」というメッセージを受け取ったら、このことを思い出してください。

このように説明してきましたが、さらにもう 1 つ「例外」があります。Java では、**RuntimeException** 以下の例外については、プログラムのあらゆる箇所で発生する可能性があるため、すべてのメソッドについて予め「`throws RuntimeExeception`」という throws 宣言がついているものとして扱われます (いちいち書いていると繁雑なため)。従って、RuntimeException 以下の例外については、いちいち受け止めて処理したり宣言しなくても大丈夫なのです。

[13]例外クラスは通常のオブジェクトですから、自分でクラスを定義した場合は説明文字列以外にさまざまなデータを保持させることもできます。