オブジェクト指向プログラミング'10 ＃5

久野　靖*

2010.10.5

今回は最終回ということで、GUI の話を中心に取り上げます。その前にレポート課題の説明をしておきます。

レポート課題

これまでに学んだどの内容でもいいので、それに基づいて 1 つ自力で Java プログラムを作成する (例題から手直ししたものでも構わない)。そのプログラムについて、次の内容から成るレポートを作成する。

- 表題「オブジェクト指向プログラミング 2010 レポート」、学籍番号、氏名、提出日付
- 課題の概要 — どのようなプログラムを作ろうと思ったか、なぜそれを選んだか
- 課題の方針 — どのような方針でプログラムを作ったか
- リスティング — 作成したプログラムのコード
- 解説 — プログラムコードのそれぞれの箇所についての解説。工夫したところなどは詳しく
- 結果 — 動作画面、動き方の説明など
- 考察 — やってみてどんなことが分かったか、オブジェクト指向がどのように役だったか
- 感想その他

レポートの締切は「11 月 9 日 (火) 一杯」とします。紙でもファイルでもいいですが、Word は使わないのでファイルの場合は PDF か単なるテキストでください。ということでよろしく。

1　GUI と GUI 部品

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。

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。

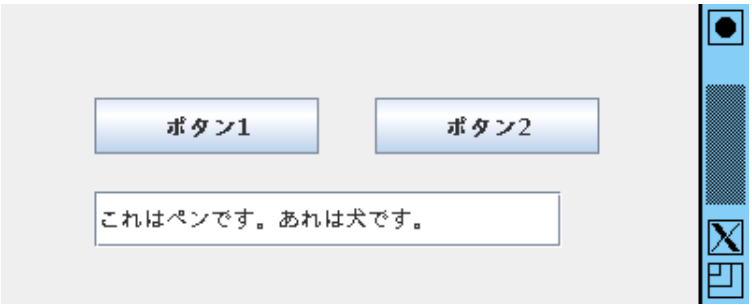


図 1: ボタンと入力欄を窓に配置

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。

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)]

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。

例題 7-1: 画面に GUI 部品を配置する

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 Sample71 extends JPanel {
  JButton b1 = new JButton("ボタン 1");
  JButton b2 = new JButton("ボタン 2");
  JTextField f1 = new JTextField();

  public Sample71() {
    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 Sample71());
    app.setSize(400, 300);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
  }
}
```

*経営システム科学専攻

前章までででプログラムがかなり長く大変になっていましたが、今回は短いプログラムです。これは、さまさまな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。[3][4]

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。[6]
- 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、[6] `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。[7]

演習 7-1 `Sample71.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GUI部品は `Component` 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。おすすめのものとしては、`JLabel`、`JCheckBox`、`JRadioButton`、`JComboBox`、`JList`、`JTextArea`、`JSlider` などがいいでしょう。[8]

例解 7-1-c

実際に、上に挙げた各部品を画面に入れて見せるだけのプログラムを例解として示します。

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

public class ex71c extends JPanel {
  JLabel l1 = new JLabel("ラベル1");
  JCheckBox c1 = new JCheckBox("チェック");
  JRadioButton r1 = new JRadioButton("赤", true);
  JRadioButton r2 = new JRadioButton("緑");
  ButtonGroup g1 = new ButtonGroup();
  JComboBox m1 = new JComboBox(new String[]{"赤","青","黄"});
  JList i1 = new JList(new String[]{"あ","い","う","え","お"});
  JScrollPane s1 = new JScrollPane(i1);
  JSlider d1 = new JSlider(0, 100, 25);
```

3)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。

4)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。

5)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。

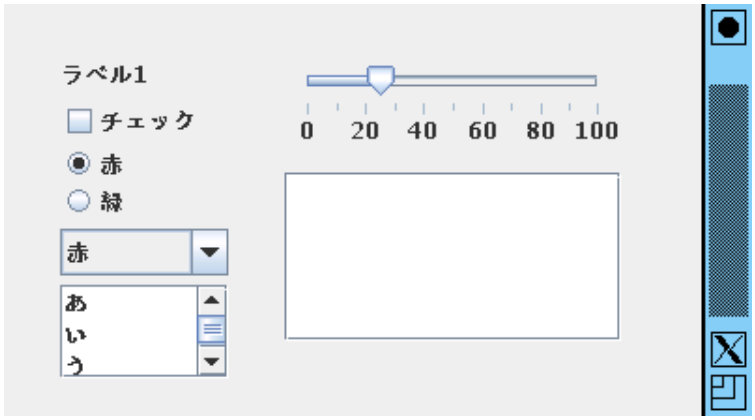
6)自動配置機能については10章で扱います。

7)選択メニューなど配置する以外の初期化(項目を設定するなど)が必要なものもありますが、それはそのためのメソッドを呼べばよいのです。

8)`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)`」のようにして使う方がいいでしょう(画面に配置するのは `JScrollPane` の方になります)。

```
  JTextArea a1 = new JTextArea();
  JScrollPane s2 = new JScrollPane(a1);

  public ex71c() {
    setLayout(null);
    add(l1); l1.setBounds(30, 25, 90, 25);
    add(c1); c1.setBounds(30, 50, 90, 25);
    g1.add(r1); g1.add(r2);
    add(r1); r1.setBounds(30, 75, 90, 20);
    add(r2); r2.setBounds(30, 95, 90, 20);
    add(m1); m1.setBounds(30, 120, 90, 25);
    add(s1); s1.setBounds(30, 150, 90, 50);
    d1.createStandardLabels(20);
    d1.setMajorTickSpacing(20); d1.setMinorTickSpacing(10);
    d1.setPaintLabels(true); d1.setPaintTicks(true);
    add(d1); d1.setBounds(150, 30, 180, 50);
    add(s2); s2.setBounds(150, 90, 180, 90);
  }
  public static void main(String[] args) {
    JFrame app = new JFrame();
    app.add(new ex71c());
    app.setSize(400, 300);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
  }
}
```



使用している部品ごとに説明しておきましょう。

- **JLabel** — 文字列を表示するだけの部品で、配置を指定するだけで使えます。
- **JCheckBox** — チェックを ON/OFF できるような部品で、配置を指定するだけで使えます。
- **JRadioButton** — 複数のボタン群の中で 1 つだけが ON になっていて、他のボタンを押して ON にすると今まで ON だったものは OFF になります。個別のボタンを配置するのに加えて、ボタン群を表すのに **ButtonGroup** オブジェクトを使います。このオブジェクトは生成した後、メソッド `add()` でグループに所属するボタンを追加するので、この追加の操作も初期化の一環として呼び出す必要があります。
- **JComboBox** — 複数の項目から 1 つを選択するメニューのような部品で、文字列の配列を指定して生成するだけで、あとは配置を指定すれば使えます。

- **JList** — 複数の項目 (ないし行) を縦に並べて表示する部品で、項目は `JComboBox` と同様に文字列の配列で指定します。画面上の領域よりも行数の方が多いのが普通なので、この部品は **JScrollPane** に入れて、そちらを画面に追加して位置指定します。
- **JSlider** — 連続した数値から値を選ぶスライドレバーの部品で、数値の範囲 (と必要なら初期値) をコンストラクタで指定して作ります (さらに方向を縦にすることもできます)。[9]
- **JTextArea** — 複数行のテキストを入力したり編集できるような部品で、これも `JScrollPane` の中に入れて使います。

[9] 目盛りやラベルをつけたい場合は、そのためのメソッドを呼んで目盛りの間隔やラベルの生成を指定します。

色々なものがあって大変だと思うかも知れませんが、実はこれでもまだ Swing の GUI 部品のごく一部を説明しただけです。必要があればまた追加して説明します。

演習 7-3 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。

2 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。[10] 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。[11] 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。

[10] 動作をプログラムしていないのですから、当り前ですね!

[11] 5 章では、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。

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。

例題 7-2: 素数判定

では「数が素数かどうか判定する」例題を見てみましょう (図 2)。

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

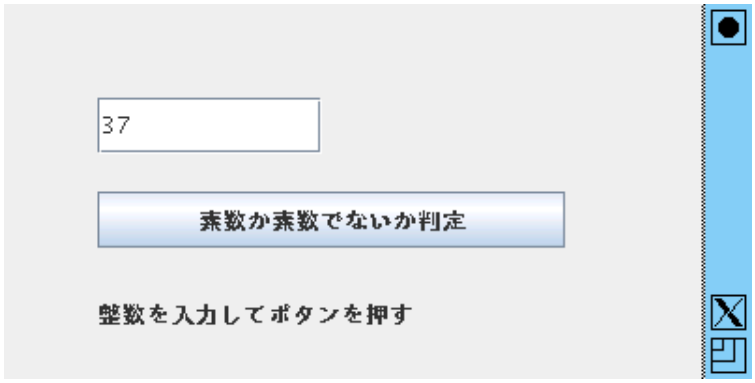


図 2: 素数判定プログラム

```
public class Sample72 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 Sample72() {
    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 Sample72());
    app.setSize(400, 300);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
  }
}
```

コンストラクタの中で GUI 部品を配置しているところは、これまでと同様です。[12]その後、ボタン b1 に対してメソッド addActionListener() により、アダプタオブジェクトを設定します。このオブジェクトは ActionListener

[12]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。

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、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。[13)]
- 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。

13)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。

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 7-4 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。[14)]

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。つまりサイコロのかわりをする。[15)]

14)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。

15)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。

演習 7-5 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。

演習 7-6 数値を入力するプログラムに対して「aa」みたいに数値でないものを入力するとどうなるか観察しなさい。

例解 7-5: 電卓

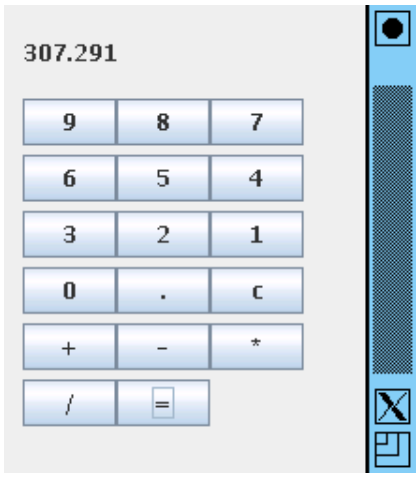
電卓を作ってみたい人は多そうなので、例解を示しましょう。電卓のように同種のボタンが沢山ある場合は、そのアダプタクラスは 1 つにしておいて、「どのボタンか」という情報を持たせ、それに応じた処理をさせるのが良さそうです。そうなると、コンストラクタでパラメタを渡すことになりますから、無名内部クラスではなく、名前のある内部クラスを使うことになります。実際に見てみましょう。

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

public class ex75a extends JPanel {
```

```
  JLabel l0 = new JLabel("0");
  String mem = "0";
  char op = '+';

  public ex75a() {
    setLayout(null);
    add(l0); l0.setBounds(10, 10, 180, 30);
    String s = "9876543210.c+-*/=";
    for(int i = 0; i < s.length(); ++i) {
      JButton b = new JButton("" + s.charAt(i));
      add(b); b.setBounds(10+(i%3)*50, 50+(i/3)*30, 50, 25);
      b.addActionListener(new MyAdapter(s.charAt(i)));
    }
  }
  class MyAdapter implements ActionListener {
    char ch;
    public MyAdapter(char c) { ch = c; }
    public void actionPerformed(ActionEvent evt) {
      if(ch == 'c') {
        l0.setText("0");
      } else if(ch=='+' || ch=='-' || ch=='*' || ch=='/') {
        mem = l0.getText(); l0.setText("0"); op = ch;
      } else if(ch == '=') {
        double x = Double.parseDouble(mem);
        double y = Double.parseDouble(l0.getText());
        if(op == '+')      l0.setText("" + (x+y));
        else if(op == '-') l0.setText("" + (x-y));
        else if(op == '*') l0.setText("" + (x*y));
        else if(op == '/') l0.setText("" + (x/y));
      } else {
        l0.setText(l0.getText() + ch);
      }
    }
  }
  public static void main(String[] args) {
    JFrame app = new JFrame();
    app.add(new ex75a());
    app.setSize(220, 250);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
  }
}
```



GUI 部品としては、表示窓 (電卓のボタンを押すと数値が入って行く窓) と、あとは押しボタンだけです。押しボタンは作った後から参照することはないので、インスタンス変数には保持しません。インスタンス変数は、表示窓、演算対象の文字列 mem と、どの演算ボタンを押したかを覚える変数 op の 3 つです。コンストラクタでは、表示窓をまず配置したあと、各ボタンをループを使って配置します。文字列 s にボタンの各文字を入れ、その各文字を左上から順に 3 列にならべて規則的に配置します。[16]その後、ボタン用のアダプタを設定しますが、その時にアダプタオブジェクトに「どの文字のボタンか」を渡して覚えさせます。

16)この、縦横に規則的に配置するやり方は、5 章のマルバツでも使いました。

ボタンのアダプタ MyAdapter は static ではない内部クラスなので、外側クラスのインスタンス変数を参照できます。コンストラクタで渡された自分の文字を変数 ch に覚えておいて、それに応じて次のようにボタン押し時の処理を枝分かれ処理しています。

- 「c」ボタン→表示窓に「0」を設定する (クリアする)。
- 演算ボタン→表示窓の内容を変数 mem に記録し、また変数 op に演算の文字を記録する。
- 「=」ボタン→ mem の内容を実数に変換したものと、表示窓の内容を実数に変換したものを演算し、[17] 結果を表示窓に入れる。
- それ以外 (つまり 0〜9 の数字) →表示窓の最後にその文字をくっつける。[18]

17)演算の種類は op に記録してあるものに応じて加減乗除のどれかを選びます。

18)このため、「0」が表示されているときに入力し始めると先頭に 0 がくっついたままになりますが、これを削除するのはよかったらやってみてください。

3 例外処理: 例外的な場合に対処する

一番最後の演習 7-6 をやってみたでしょうか? 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 Sample72$1.actionPerformed(Sample72.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。[19]

19)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。

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 3 に示します。すべての例外

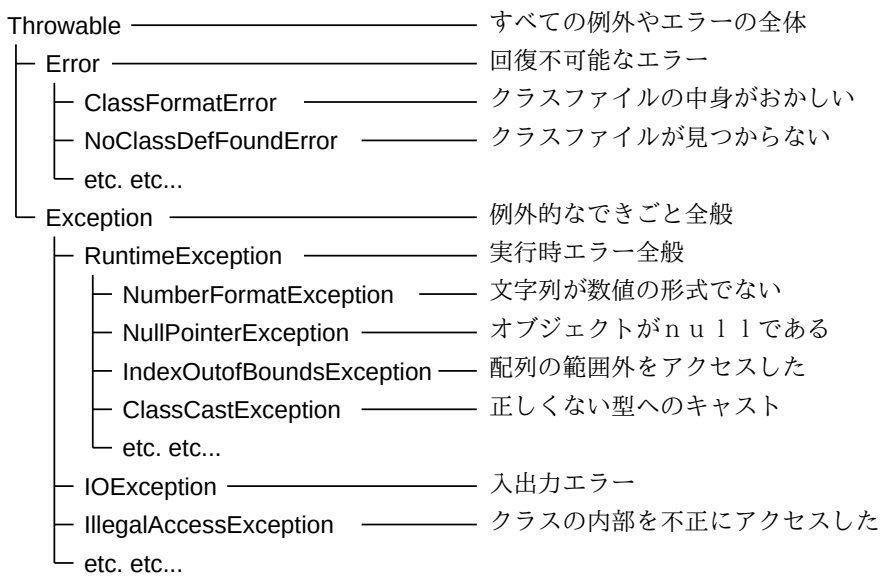


図 3: Java の例外クラスの階層

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、自前で適切な対処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 ...
}
```

これは、図 4 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、そこで対処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。

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。

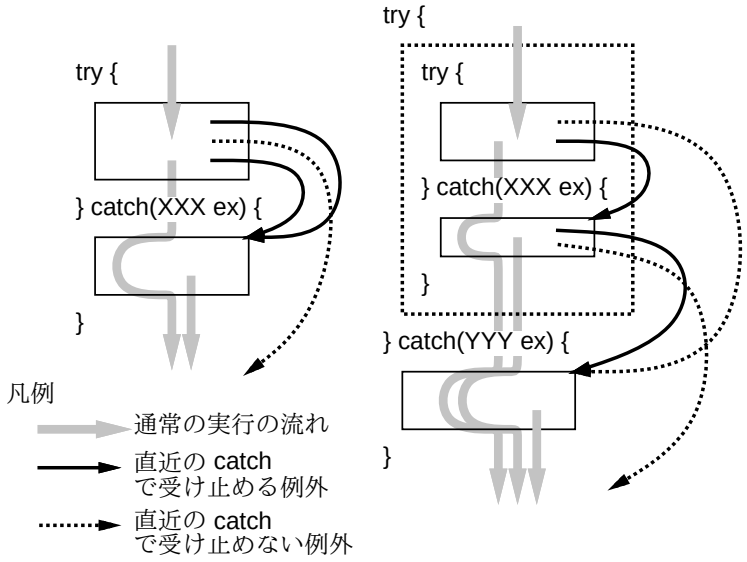


図 4: try-catch の構造

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 5)。

```
public class Sample72b 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 Sample72b() {
    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。");
        }
```

```
        }
      });
    }
```

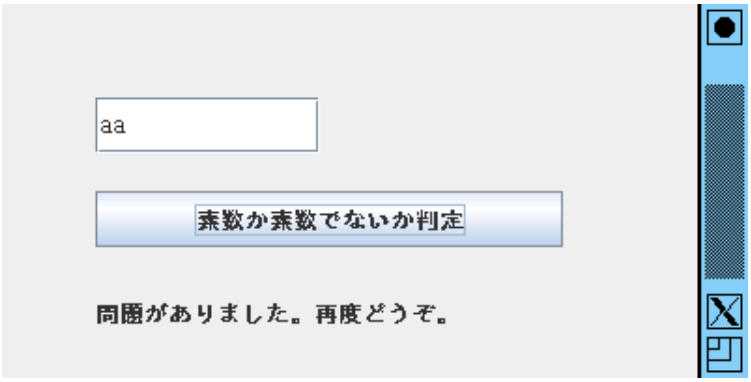


図 5: 素数判定プログラムで例外に対処

演習 7-7 これまでに作成した GUI プログラムについて、例外処理を追加して、正しくない入力でもきちんと対処されるようにしてみなさい。

ここでは、GUI 部品を使って値の入出力を行うやり方について基本的なことがらを学びました。また、GUI 部品をやみくもに使えばいいわけではなく、プログラムが分かりやすいインタフェースを持つようにするためには設計や工夫が必要であることも説明しました。これと関連して、例外処理により正しくないデータに対処する枠組みについても取り上げました。

4 付録: 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。[20)]

[20)]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。

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(...) thorws 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。

4 章で画像ファイル読み込みのところが次のようになっていたのを覚えているでしょうか?

```
try {
  img = ImageIO.read(new File(fname));
} catch(Exception ex) { }
```

これは、メソッド ImageIO.read() は throws IOException と指定されているので、この例外に対処する必要があるためです。そして上のコードでは、この例外が発生した場合は受け止めてそのまま無視するようになっているのでした。[21)]

[21)]本当は、「読めなかった」印となる画像 (バツ印など) を内部的に用意して動作を続行すると、プログラムがとりあえず動作し続けるという点では望ましかったはずです。

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「thorws 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。

5　ストリーム入出力

ここまでで、マウスクリックや打鍵や GUI 部品による入力、窓に図形など描く形での出力を扱って来ましたが、実用のプログラムで最も多いのは文字 (テキスト) の入出力でしょう。多くのプログラミング言語では、テキスト入出力を**ストリーム**と呼ばれる概念で統一的に扱います (図 6)。[22)]

22)ストリームとは要するに「文字が流れるパイプ」のようなもので、入力ストリームであればそこから次々に文字が読み出せますし、出力ストリームではそこに文字を書き込むことになります。

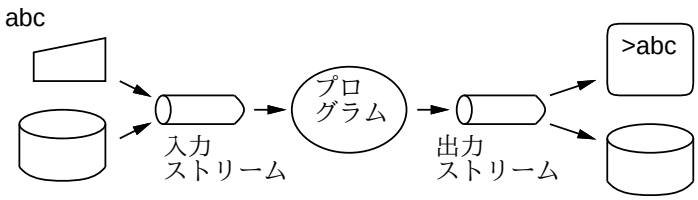


図 6: ストリームの概念

入力ストリームの入力側はキーボード、ファイル、ネットワーク接続などさまざまなものに接続される可能性があり、また出力ストリームの出力側も文字表示画面、ファイル、ネットワーク接続などさまざまなものに接続される可能性があります。重要なのは、プログラム中のストリームを使って読み書きしている部分は「文字を読む」「文字を書く」という操作に統一されているので、接続先がどこであってもプログラムは同じままでよい、ということです。これを**機器独立性**と言います。[23)]

23)機器独立性がないと、接続する相手先によって読み書きの方法を変えたり、悪くするとプログラムのアルゴリズムまで手直しする必要があったりして、ある機器向けに作ったプログラムを別の機器向けにそのまま使うことができなくなります。ストリーム入出力は、そのような先人の「失敗」を土台として作られた、よく考えられた入出力の抽象化なのです。

標準の入出力ストリーム

Java プログラムを実行開始したとき、次の 3 つのストリームオブジェクトが自動的に用意され、利用可能になります。[24)]

- **System.in**(**標準入力**、standard input) — `InputStream` オブジェクトであり、Java プログラムを起動した画面に対応するキーボード入力に接続されます。
- **System.out**(**標準出力**、standard output) — `PrintStream` オブジェクトであり、Java プログラムを起動した画面への文字表示に接続されています。
- `System.err`(**標準エラー出力**、standard error output) — 同じく画面への文字表示に接続された `PrintStream` オブジェクトであり、エラー表示などに使われます。[25)]

24)これらのオブジェクトはクラス `System` のクラス変数に予め格納されています。

25)`System.out` を別の出力先に切り替えた時に、エラー表示まで切り替えられて表示されなくなると不都合なので、そのような事態を避けるために別途 `System.err` があるわけです。

このため、キーボード入力と文字出力だけを使う簡単な CUI プログラムは、これらのストリームだけを使って (新たなストリームを用意したりせずに) 作ることができます。

例題 8-1: PrintStream による出力

標準で用意されているストリームのうち、**PrintStream** については、次のようなインスタンスメソッドが用意されていて、さまざまな出力にそのまま使うことができます。

- **print**(X)、**println**(X) — `int`、`double` など各種の基本型と `String` を引数とした同名のメソッドがあり、[26] それらを出力します。[27]
- `printf`(書式文字列，引数…) — `String.format()` と同様、第 1 引数が書式文字列で、その中に現れる「`%`」で始まる変換指定の箇所に第 2 引数以降を変換して埋め込み、完成した文字列を出力します。

`PrintStream` を使った簡単な例題を示しましょう。コマンド起動時に整数 N を指定すると、1 から N までの整数の 2 乗と平方根を表示するものです。実行例から示します。

```
% java Sample81 5    ←起動のための Java コマンド (値を指定)
 N  N*N   sqrt(N)
--------------------------
 1    1   1.00000000000000
 2    4   1.41421356237310
 3    9   1.73205080756888
 4   16   2.00000000000000
 5   25   2.23606797749979
%
```

起動時に「5」を指定していることに注意してください。このように起動時のコマンド行で指定した値は空白のところで区切られ、文字列の配列として `main()` に渡されます。これを**コマンド引数** (command line arguments) と言います。これまで `main()` は文字列の配列 `args` を受け取るように書いて来ましたが、その内容は利用していませんでした。この例題ではコマンド引数の先頭要素を整数に変換して N として使います。ではコードを見てみましょう。[28]

```
import java.util.*;

public class Sample81 {
  public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);
    System.out.println(" N  N*N   sqrt(N)");
    System.out.println("--------------------------");
    for(int x = 1; x <= n; ++x) {
      System.out.printf("%2d %4d %18.14f\n", x, x*x, Math.sqrt(x));
    }
  }
}
```

例題 8-2: Scanner による入力

表示で用意されているストリームのうち **InputStream** の方は、単に文字 (正確にはバイト値) を読み込むだけの機能しか持たないので、`java.util` クラスに含まれている **Scanner** オブジェクトと組み合わせることが普通です。`Scanner` オブジェクトは `InputStream` やその他の入力ストリームをコンストラクタで指定して生成し、数値や文字列を読み込むための次のようなメソッドを提供してくれます。

26) このように、同じ名前で複数のメソッドを定義することを**オーバロード** (overload、**多重定義**) と呼びます。Java では、引数の個数と型によって区別が可能であれば、いくつでもメソッド定義をオーバロードできます。

27) `println()` の方は出力したあと「改行」します。また、改行するためだけの引数なしの `println()` も用意されています。

28) 書式文字列を使った出力では、書式指定のときに幅を指定することで縦に揃った出力を生成することができます。ただし、文字ごとに幅が変化するような場合は、当然ながら文字数だけ揃えても縦に揃って見えるようにはなりません。

- **nextLine()** — 改行までを読み込み、改行文字は除いてその手前までを 1 つの文字列として返す。
- **next()** — 改行や空白文字で区切られた「かたまり」を読み込み、1 つの文字列として返す。
- **nextInt()**、**nextLong()**、**nextFloat()**、**nextDouble()** など — `int`、`long`、`float`、`double` などの基本型の値を読み込んで返す。
- **hasNextInt()**、**hasNextLong()**、**hasNextFloat()**、**hasNextDouble()** など — それぞれの入力に先立って、次に入力する値があるかどうかをチェックしする (ある場合は `true` を返す)。

では、`Scanner` と `PrintStream` を使って簡単な CUI プログラムを作ってみましょう。入力間違いに対処するため、例外処理も使っています。

```
import java.util.*;

public class Sample82 {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    while(true) {
      try {
        System.out.print("x> ");
        Double x = sc.nextDouble();
        System.out.print("y> ");
        Double y = sc.nextDouble();
        System.out.printf("x + y = %.10f\n", x+y);
        System.out.print("continue? ");
        String s = sc.next();
        if(!s.matches("^[yY]")) { break; }
      } catch(Exception ex) {
        sc.nextLine();
        System.out.println("? " + ex.toString());
      }
    }
  }
}
```

先頭で入力ストリーム `System.in` を渡して `Scanner` を作ります。続いてループに入り、さらに例外処理のための try 文に入ります。その中でプロンプトを出して 2 つの実数値 `x` と `y` を読み込み、その和を打ち出します。続いて、「続けますか?」と質問し、次の語を読み込んでその語の先頭が「Y」または「y」なら続行ですが、そうでないならループを抜けて終わります。[29] 実行例を示しておきます。

ここまでのどこかで例外があった場合はそれを全て受け止め、「?」に続いて例外を文字列に変換したものを打ち出し、ループの周回を続けます。[30]

```
% java Sample82
x> 1
```

29)文字列のインスタンスメソッド **matches()** は、**正規表現** (regular expression、ないし**パターン** — pattern) を受け取り、文字列がそのパターンにあてはまるかどうか判定するものです。ここでは「^」(先頭位置にある) と「[...]」(かぎ括弧内のどれかの文字に一致) を組み合わせたパターンを指定しています。

30)例外ハンドラの中で `Scanner` の **nextLine()** を呼んでいますが、これは例外が起きたところで入力が止まっていてストリーム中にその先の文字が残っているため、次の周回にそれが読まれてしまうと困るので、現在の行の内容を読みすてるためです。

```
y> 3
x + y = 4.0000000000
continue? y
x> a                                    ←数値でない入力
? java.util.InputMismatchException      ←例外ハンドラの出力
x> 5
y> 9
x + y = 14.0000000000
continue? n
%
```

演習 8-1 例題Sample81.java、Sample82.javaをそのまま打ち込んで動かしなさい。動いたら、そのどちらかを手直しする形で、次のような処理を行う CUI プログラムを作成しなさい。

a. 身長と体重を受け取り、**BMI** 指数を出力する。[31)]
b. **摂氏** (Celsius) の温度を**華氏** (Fahrenheit) に変換、華氏の温度を摂氏に変換する (どちらの変換かもそのつど指定する)。[32)]
c. 元金と年利を受け取り、20 年間にわたって複利で増やした場合の毎年の元利合計を出力する。
d. ローン金額、利息、毎年返済額を受け取り、元利均等返済で返済するまでの毎年のローン残額と返済額累計を出力する。
e. その他、自分でやってみたいと思う計算をおこなう。

31)BMI 指数は肥満の度合を示すのに使われます。その計算式は次の通りです。

$$BMI = \frac{\text{体重 } (kg)}{(\text{身長 } (m))^2}$$

32)摂氏と華氏間の変換式は次の通りです。

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

$$F = \frac{9}{5} \times C + 32$$

例解 8-1-b

この問題は、摂氏→華氏と華氏→摂氏のどちらの変換をするのか判断する必要があります。そこで、方針として、コマンド引数で変換する温度を指定し、その最後[33)] に「C」「c」がついていれば摂氏→華氏、「F」「f」がついているか何もついていなければ華氏→摂氏の変換をすることにします。次のような感じです。

33)正規表現では「$」が末尾へのあてはまりを表します。

```
% java ex81b 25c
25.00C = 77.00F
% java ex81b 32.5f
32.50F = 0.28C
% java ex81b 32.5
32.50F = 0.28C
%
```

このため、コマンド引数は末尾の「F」「C」を削除してから実数に変換します。[34)] 続いて、matches() を使って数字の後ろに C がついているか否かを判定し、それに応じて摂氏→華氏、華氏→摂氏のいずれかの計算を行います。

34)これには文字列のインスタンスメソッド **replaceFirst()** が利用できます。

```
public class ex81b {
  public static void main(String[] args) {
    double x = Double.parseDouble(args[0].replaceFirst("[FfCc]$", ""));
    try {
      if(args[0].matches("^[.0-9]+[Cc]")) {
        double c = x;
        double f = c * 9 / 5 + 32;
        System.out.printf("%.2fC = %.2fF\n", c, f);
```

```
      } else {
        double f = x;
        double c = (f - 32) * 5 / 9;
        System.out.printf("%.2fF = %.2fC\n", f, c);
      }
    } catch(Exception ex) {
      System.out.println("? " + ex.toString());
    }
  }
}
```

例解 8-1-d

この問題はユーザに項目ごとに入力してもらう形のプログラムにします。そのため、基本的な構造は例題 8-2 と同じです。データが揃ったら内側のループを使って返済までの経過を出力します。[35)]

35)返済額が少なすぎる場合は残額が増える一方になるのでその旨警告するだけで経過出力はしません。

```
import java.util.*;

public class ex81d {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    while(true) {
      try {
        System.out.print("loan amount? ");
        Double loan = sc.nextDouble();
        System.out.print("interest rate[%]? ");
        Double rate = sc.nextDouble();
        System.out.print("repayment amount? ");
        Double repay = sc.nextDouble();
        if(loan * (1.0+0.01*rate) - repay >= loan) {
          System.out.println("repayment amount too little.");
        } else {
          System.out.println("Year  Remaining  Repay      Total Repay");
          int year = 0;
          double total = 0.0;
          while(loan > 0.0) {
            loan = loan * (1.0+0.01*rate) - repay;
            if(loan < 0.0) { repay += loan; loan = 0.0; }
            total = total + repay; ++year;
            System.out.printf("%4d %10.2f %10.2f %10.2f\n",
                              year, loan, repay, total);
          }
        }
        System.out.print("continue? ");
        String s = sc.next();
        if(!s.matches("^[yY]")) { break; }
      } catch(Exception ex) {
        sc.nextLine();
        System.out.println("? " + ex.toString());
      }
    }
  }
}
```

実行のようすも示しておきます。

```
% java ex81d
loan amount? 1000000
```

```
interest rate[%]? 8
repayment amount? 50000
repayment amount too little.
continue? y
loan amount? 1000000
interest rate[%]? 8
repayment amount? 200000
Year  Remaining  Repay      Total Repay
   1  880000.00  200000.00  200000.00
   2  750400.00  200000.00  400000.00
   3  610432.00  200000.00  600000.00
   4  459266.56  200000.00  800000.00
   5  296007.88  200000.00 1000000.00
   6  119688.52  200000.00 1200000.00
   7       0.00  129263.60 1329263.60
continue? n
%
```

6 ファイルやネットワークの入出力

ここまでは、Java プログラムに対して最初から用意されるストリームを使って入出力を行ってきましたが、本章のはじめに説明したように、ファイルやネットワークに対する入出力も同じように扱うことができます。ただし、最初にストリームを作るところはそれぞれの接続先ごとに固有のやり方が必要です。これについて説明しましょう。

まずファイルの場合から説明しましょう。ファイルへの入出力のためには、対象となるファイルを指定する必要があります。Java ではファイルを表すための **File** オブジェクトが用意されていて、これを用いてファイルのさまざまな属性 (保護モードなど) を調べたりできます。そして、このオブジェクトを指定してストリームを作成することができます。

- `new File(`文字列`)` — ファイル名やファイルが置かれているディレクトリまで含めたパス名の文字列を指定して `File` オブジェクトを生成する。
- `new PrintStream(File)` — 指定したファイルに対して出力をおこなう `PrintStream` オブジェクトを生成する。
- `new Scanner(File)` — 指定したファイルから読み込む `Scanner` オブジェクトを生成する。

ネットワークについてはさまざまな場合があって複雑なので、一番簡単な「特定 URL からの読み込み」だけを説明しましょう。この場合はファイルの場合と同様、まず **URL** オブジェクトを生成して、そこからストリームを取得します。[36)]

- `new URL(`文字列`)` — URL の文字列を指定して URL オブジェクトを生成する。
- **openStream()** — URL オブジェクトのインスタンスメソッドであり、その URL の指す内容を読み込む `InputStream` オブジェクトを返す。

最後に注意すべきことですが、自分で新たにストリームを作成した場合は、使い終わったらインスタンスメソッド `close()` を呼ぶようにしてくだ

36)URL からは `InputStream` が得られるので、それを行単位などで読み込みたい場合には `new Scanner(new URL("...").openStream())` などのようにして `Scanner` を通じて読み込むようにします。

さい。通常の Java オブジェクトは「後始末」をしなくてもガベージコレクタが回収してくれますが、入出力が関係する場合には OS などの資源も使われるので、それを解放するために `close()` が必要なのです。[37]

また、ここで説明した各クラスは、`File` および `PrintStream` が `java.io`、`URL` が `java.net`、`Scanner` が `java.util` の各パッケージに含まれているので、必要な `import` 指定を忘れないようにしてください。[38]

37) Java プログラムが終了すると自動的に解放されるので、すぐ終わるプログラムではあまり問題ありませんが、長時間実行を続けるプログラムでは OS 側の資源が解放されないままになって新規にストリームが作れなくなるなどの問題が起きることがあります。

38) または、`import` するかわりに「`java.net.URL(...)` のようにフルクラス名で指定するのでも構いません。

7 日本語の扱いと文字コード

本書ではここまで、あまり説明せずに日本語の文字を含む例題を掲載してきました。それで何の問題もなく動作する環境が大部分だとは思います。しかしそのような場合でも、入力や出力のところで何らかの変換が行われていて、場合によってはそのことを意識した設定や調整が必要になることもあります。これについて説明しておきましょう。

Java は最初からさまざまな国で使われることを想定して設計されたため、さまざまな国の**文字コード** (character code) を扱うことが前提になっています。このとき、プログラム中で直接各国の文字に依存した処理をしてしまうと、そのプログラムを他の国の文字で動かすことが難しくなります。

そのため、Java ではプログラムが扱う文字集合とその表現方法を **UNICODE** の 16 ビット表現に統一しました。つまり、どの国で Java プログラムを動かしていても (たとえば 1 文字が 8 ビットで表せる英語圏の国でも)、1 文字は 16 ビットで表されています。これを**内部エンコーディング** (internal encoding) と呼びます。一方、プログラムの外側のファイルや入出力装置上では、文字はそれぞれの国や地域の慣習に従って表現されています。これを**外部エンコーディング** (external encoding) と言います。そして、図 7 のように、外部から文字をプログラムに読み込むところと、プログラムから外部に文字を書き出すところで、表現の変換を行います。

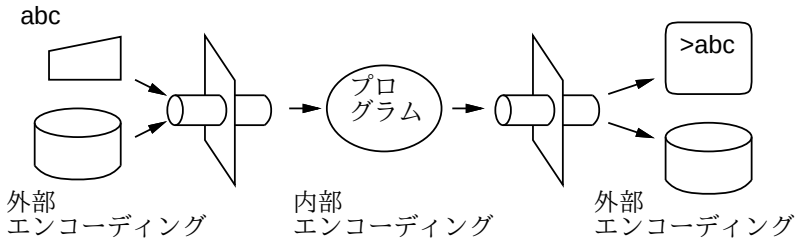


図 7: 内部エンコーディングと外部エンコーディング

エンコーディングの境界をまたがるところにあるのはストリームですから、この変換はストリームの機能の一部として行われます。そして、Java プログラム側では、読み込む時は「この外部エンコーディングから読み込む」、書き出す時は「この外部エンコーディングに書き出す」という形で変換を指定します。その指定をするのは次の箇所です。

- `new PrintStream(File, String)` — `PrintStream` を作る時に追加の文字列で指定する。

- `new Scanner(InputStream, String)`、`new Scanner(File, String)` — `Scanner` を作る時に追加の文字列で指定する。

指定する内容は**文字セット名** (character set name) と呼ばれますが、日本語に関係するものとしては以下のものがあります。

- `"JISAutoDetect"` — JIS 自動判別
- `"ISO-2022-JP"` — 7 ビット JIS コード
- `"Shilf_JIS"` — Shift JIS コード
- `"EUC_JP"` — 日本語 EUC コード
- `"UTF-8"` — UNICODE UTF-8 表現
- `"UTF-16"` — UNICODE UTF-16 表現

入力に関しては、`"JISAutoDetect"`を指定しておけば、7 ビット JIS、Shilft JIS、日本語 EUC のどれであるかを自動的に判別してもらえます。出力についてはそういうわけに行かないので、特定のものを指定します。

Java のソースプログラム中に日本語を書く場合も、実はこれと同じ問題があります。[39] 日本語を含んだソースプログラムがうまくコンパイルできない場合は、次のように `javac` コマンドで文字セットを指定します。[40]

```
javac -encoding Shift_JIS Test.java
```

最後に、これまで `javac` コマンドでも入出力ストリームでも何も指定しなかったけれど問題なく日本語が扱えていた、という場合の説明です。Java の実行系では、ストリームに文字セットが指定されていない場合、実行環境の文字コード情報を取得して、それに合わせて変換を行おうとします。これがうまく行っている場合は、とくに指定しなくてもきちんと日本語が扱えます。それはそれでいいのですが、「出力はうまく行くけれど入力が駄目」「これこれのファイルから読み込む時は駄目」など特定の場合に問題が生じたら、その場合には適切な文字セットを指定するような手直しを行ってください。

[39] そもそも `javac` コマンドは Java で書いてありますから、読み込む時に内部エンコーディングへの変換をするわけです。

[40] 5 章で「うまく行かない場合に」と紹介したのはこのやり方で JIS 自動判別を指定する方法だったわけです。

8 例題: 家計簿管理プログラム

材料が揃ったので、ここで少し大きめの例題として「家計簿管理プログラム」を作ってみます。家計簿のデータは各行に「日付」「金額」「支出内容」「分類」の 4 つの項目を並べて空白で区切ったファイルに入れておく (エディタなどで作成/追加する) ものとします。データの例を示しておきましょう。

```
2010.1.11  1000 食料品  生活費
2010.1.11  1500 文房具  学習費
2010.1.12   400 弁当代  生活費
2010.1.12  1500 教科書  学習費
2010.1.12  2500 コンパ  娯楽費
```

このようなデータの入ったファイルがあるとして、家計簿プログラムは次の機能 (コマンド) を持つようにします。

- read — データファイルを読み込む (複数回この機能を使うと複数のデータファイルの内容を併せて処理できるものとします)。
- print — データをファイルに書き出す。
- sum — 総合計と分類ごとの合計を書き出す。
- exit — 終了する。

print と sum では、ファイル名を指定すると画面の代わりにファイルに結果を書き出すものとします。このプログラムを動かしている様子を示します。

```
% java Sample83
command> read test.data
command> print
2010.1.11     1000 食料品 生活費
2010.1.11     1500 文房具 学習費
2010.1.12      400 弁当代 生活費
2010.1.12     1500 教科書 学習費
2010.1.12     2500 コンパ 娯楽費
command> sum
TOTAL = 6900
娯楽費 = 2500
学習費 = 3000
生活費 = 1400
command> sum t
command> exit
%
```

やや長いので、部分ごとに分けて概要を説明して行きます。家計簿データは read で読み込むとそのままプログラム内で保持することになるので、そのためのデータ構造を決める必要があります。ここでは、「日付」「金額」「項目」「分類」それぞれを ArrayList オブジェクトに保持するようにしました。金額は整数なので ArrayList<Integer>、あとの 3 つは ArrayList<String> です。

```
import java.util.*;
import java.io.*;

public class Sample83 {
  static ArrayList<String> date = new ArrayList<String>();
  static ArrayList<Integer> amt = new ArrayList<Integer>();
  static ArrayList<String> item = new ArrayList<String>();
  static ArrayList<String> kind = new ArrayList<String>();
```

次に、main() ではコマンドをキーボードから入力するための Scanner を用意して利用し、続いて無限ループでコマンドを処理します。ただしこのプログラムでは、キーボードからの入力は nextLine() で 1 行単位で読んでしまい、String のインスタンスメソッド **split()** を使って空白の箇所で区切って配列 cmd に入れます。[41] これはこの方がコマンドごとに個別に行内

[41] split() も matches() と同様に正規表現を使って、区切る箇所を指定します。ここでは「空白かタブ文字の 1 個以上の並び」のところで区切るようにしています。

容を解析するよりも便利ですし、パラメタがついているかどうか等も調べやすいからです。

コマンドについては、if-else の連鎖で順番に「どのコマンドであるか」を振り分けるようになっています。[42] そして、read などのコマンドはそれぞれが下請けのメソッドを呼び出し、その中で処理するようになっています。

[42]文字列が等しいかどうかを調べるには、メソッド **equals()** を使う必要があったことを思い出してください。

```
public static void main(String[] args) {
  Scanner sc = new Scanner(System.in);
  while(true) {
    try {
      System.out.print("command> ");
      String[] cmd = sc.nextLine().split("[ \t]+");
      if(cmd.length == 0) {
        // do nothing
      } else if(cmd[0].equals("exit")) {
        System.exit(0);
      } else if(cmd[0].equals("read")) {
        if(cmd.length < 2) {
          System.out.println("Usage: read <filename>");
          continue;
        }
        read(cmd[1]);
      } else if(cmd[0].equals("print")) {
        PrintStream pr = System.out;
        if(cmd.length > 1) {
          pr = new PrintStream(cmd[1], "ISO-2022-JP");
        }
        print(pr);
      } else if(cmd[0].equals("sum")) {
        PrintStream pr = System.out;
        if(cmd.length > 1) {
          pr = new PrintStream(cmd[1], "ISO-2022-JP");
        }
        sum(pr);
      } else {
        System.out.println("Unknown command: " + cmd[0]);
      }
    } catch(Exception ex) {
      System.out.println("? " + ex.toString());
    }
  }
}
```

read の場合はファイル名のパラメタが必要なので、それが無ければ (配列 cmd の長さが 1 以下なら) エラーメッセージを出して無限ループの次の周回に進みます。ファイル名があればそのファイル名をパラメタとして下請けメソッドを呼び出します。

print と sum の場合はどちらも類似していて、まず pr に System.out を入れておき、ファイル名のパラメタが指定されていれば pr にファイルに出力する PrintStream を入れ直すようになっています。それが終わったらそれぞれの下請けメソッドを呼びます。

では、read の下請けメソッドを見てみましょう。指定されたファイルから読み込む Scanner を用意し、こちらも main() と同様に行単位で読み込み、split() で分割します。そして、各フィールドを取り出しますが、金額については同時に整数に変換します。それが完了したら、各項目を ArrayList に追加します。[43)]

43)すべての取り出しが完了するまで追加を始めないので、どこかでエラーがあって例外ハンドラに飛んだ場合でも、ArrayList の長さが揃わなくなることはないようになっています。

```
public static void read(String f) throws Exception {
  Scanner sc = new Scanner(new File(f), "JISAutoDetect");
  int lineno = 0;
  while(sc.hasNextLine()) {
    String line = sc.nextLine(); ++lineno;
    String[] a = line.split("[ \t]+");
    try {
      String d = a[0], i = a[2], k = a[3];
      int m = Integer.parseInt(a[1]);
      date.add(d); amt.add(m); item.add(i); kind.add(k);
    } catch(Exception ex) {
      System.out.printf("line %d: splious line: '%s'\n",
                        lineno, line);
    }
  }
  sc.close();
}
```

コマンド print については、単に ArrayList に入っている各データを printf() で出力するだけです。

```
public static void print(PrintStream pr) throws Exception {
  for(int i = 0; i < date.size(); ++i) {
    pr.printf("%-8s %8d %s %s\n",
         date.get(i), amt.get(i), item.get(i), kind.get(i));
  }
}
```

コマンド sum は、やや複雑です。まず、**TreeMap** という新しいコンテナクラスが出て来ますが、これは 2 つの型パラメタを持っていて、1 つ目のパラメタ型の値を鍵として 2 つ目のパラメタ型の値を登録し、同じ鍵を指定して登録された値を検索できるようになっています。これは、図 8 のように、2 つのカラムから成る表で、1 カラム目 (鍵) の値でどの列かを検索し、対応する列の 2 カラム目の値を読み書きする、というふうにイメージすると分かりやすいと思います。[44)]

44)同様の機能を持つコンテナクラスとして **HashMap** があります。HashMap の方が高速ですが、TreeMap は foreach などで値を取り出す時に鍵の値の昇順に並んで取り出されるのに対し、HashMap では順番はランダムになります。ここでは順番に表示したいので TreeMap を使っています。

これを何に使うかというと、総合計は「TOTAL」、分類ごとの小計はその分類の名前文字列を鍵として、TreeMap<String,Integer>に登録しておきます。こうすれば、使用記録ごとに TOTAL と分類の登録値を取り出して

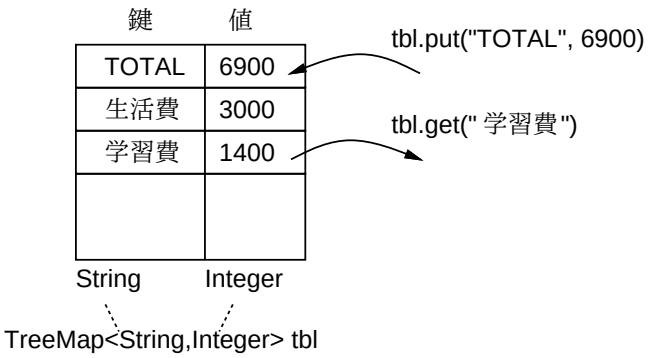


図 8: TreeMap コンテナクラス

それにその使用記録の金額を追加して登録し直すことで、総合計と分類ごとの合計が計算できるわけです。

```
  public static void sum(PrintStream pr) throws Exception {
    TreeMap<String,Integer> tbl =
        new TreeMap<String,Integer>();
    for(int i = 0; i < date.size(); ++i) {
      int a = amt.get(i);
      int t = 0;
      if(tbl.containsKey("TOTAL")) { t += tbl.get("TOTAL"); }
      tbl.put("TOTAL", t + a);
      String k = kind.get(i);
      t = 0;
      if(tbl.containsKey(k)) { t += tbl.get(k); }
      tbl.put(k, t + a);
    }
    for(String k: tbl.keySet()) {
      pr.format("%s = %d\n", k, tbl.get(k));
    }
  }
} // Sample83 の終わり
```

合計を計算し終わったら最後に **keySet()** で鍵の並びを取り出し、それぞれの鍵について対応する金額を並べて表示することで、総合計と分類毎の合計が表示されます。

演習 8-2 例題 Sample83.java を打ち込んでそのまま動かしなさい。動いたら、次のようなコマンドを追加してみなさい。

a. 指定した金額以上の買物のみを表示するコマンド higher。[45)]

b. 1 件あたりの平均金額を全体およびそれぞれの分類について表示するコマンド ave。

c. それぞれの日ごとの使用合計額を計算するコマンド daysum。

d. 金額の高い順に指定件数ぶんの買物を表示するコマンド high。

e. その他役に立ちそうな分析を行うコマンド。

45)たとえば 1 万円以上の買物だけを見たければ「higher 10000」とするわけです。

この項ではこれまでとうって変わって、窓を開かず、プログラムを起動した窓でそのまま入出力を行うタイプのプログラム (CUI プログラム) とファイル入出力を取り上げました。毎回決まった手順でデータを加工するなどの用途であれば、このタイプのプログラムの方が GUI プログラムよりも扱いやすいので、ここで学んだタイプのプログラムは業務用には広く使われています。皆様も「定型的な仕事」が必要になった時はこのことを思い出してください。

9 付録: 文字列の操作と正規表現

ここまで、あまり系統的に説明しないまま、文字列やパターンを利用して来ました。しかし、どのような機能があるかをひととおり知っておいた方が、適切な使い方が判断しやすいでしょうから、ここでまとめて整理しておきます。

文字列の基本的な操作

文字列は概念としては文字の並びであり、それらを扱うには次のインスタンスメソッドを知っておけばいちおうの用が足ります。

- `int` **length**`()` — 文字列の長さ (含まれる文字数) を返す。
- `char` **charAt**`(int)` — 指定した位置の文字を返す。
- `String` **substring**`(int,int)`、`substring(int)` — 最初のパラメタで指定した位置から 2 番目のパラメタで指定した位置の手前/末尾の位置までの部分文字列を抜き出して返す。
- `String` **concat**`(String)` — 指定した文字列を後ろに連結した文字列を返す。[46]
- `int` **indexOf**`(String)`、`int indexOf(String,int)` — 文字列先頭から/2 番目のパラメタで指定した位置から見て行き、最初にどの位置に 1 番目のパラメタで指定した文字列が部分文字列として現れるかを返す (現れない場合は-1 を返す)。
- `int` **lastIndexOf**`(String)`、`int lastIndexOf(String,int)` — おおむね `indexOf()` と同様だが、ただし一番最後の出現位置を返す (位置を指定する場合、その位置より手前で一番最後の出現位置を返す)。

46) 実際には演算子「+」は左辺か右辺どちらから文字列の場合は文字列の連結演算になりますから、これを使って連結することが多いでしょう。

正規表現を扱う操作

上では文字列の中にある別の文字列を見つける操作として `indexOf()` 等を挙げましたが、これらは「決まった文字列」しか見つけられません。しかし、実際には、「空白が 1 個以上あるところで区切る」「英小文字のどれかがあるところを見つける」など、もっと一般的なパターンを指定して見つけたいことが多いものです。そのようなパターンを指定する標準的な方法として、Unix ではじまった**正規表現**という記法があるのです。[47]

正規表現には、次のものが含まれることができます。

47) 正規表現についてはそれ単独で 1 冊の本があるくらい広範な話題ですが、ここではよく使うものだけに絞って説明します。

- c — 一般の文字は、その文字自体にマッチする正規表現を表します。
- $\backslash c$ — この行以下に現れる特別な意味を持つ文字は、$\backslash$ を前につけることでその文字自体にマッチする正規表現を表します。[48]
- `.` — 任意の 1 文字にマッチする正規表現を表します。
- `^`、`$` — 文字列の先頭位置/末尾位置にマッチすることを表します。
- `[`$c\cdots$`]`、`[^`$c\cdots$`]` — それぞれ、角かっこ内の文字のどれか 1 文字/どれか以外の 1 文字にマッチすることを表します。[49]
- X`+`、X`*` — 正規表現 X が 1 個以上/0 個以上並んだものを表します。

正規表現を扱う文字列のメソッドの代表例を挙げます。[50]

- `boolean matches(String)` — 文字列がパターンにマッチする否かを判断する。
- `String replaceFirst(String,String)`、`String replaceAll(String,String)` — パターンにマッチする最初の箇所/全箇所を 2 番目のパラメタで指定した文字列に置き換えた文字列を返す。
- `String[] split(String)` — パターンにマッチするすべての箇所で文字列を分割し、結果を文字列の配列として返す。

48)「\」も特別な文字ですから、1 個の「\」にマッチする正規表現は「\\」ということになります。

49)角かっこ内では「`A-Z`」で「A から Z までの範囲の全文字」を表すという形で「`-`」を使うことができます。「`-`」自体を含めたい場合は先頭に書く必要があります。

50)いずれも本章の例題や例解でおなじみのものです。正規表現そのものは文字列の形で表現して渡します。

10　レイアウトマネージャ

ここまでは、GUI 部品を「位置と大きさ」を指定する形で窓内に配置していました。このやり方は分かりやすいのですが、ユーザが窓の大きさを変更したときに対応できません。たとえば例題 8-1 のプログラムで、窓の大きさを小さくすると、GUI 部品が窓からはみ出して切れてしまいます (図 9)。

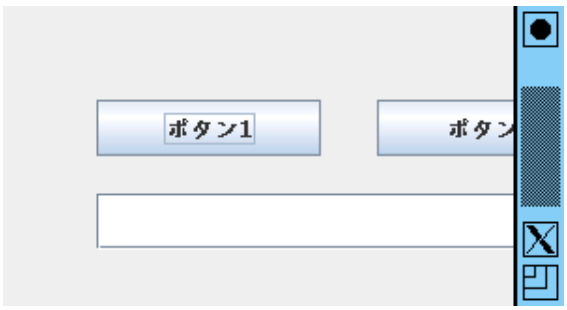


図 9: 例題 8-1 の窓の大きさを変更したところ

これに対処するには、窓の形に応じて部品を自動配置する機構である**レイアウトマネージャ**(layout manager) を利用します。その場合、部品を領域に追加するとレイアウトマネージャが自動的に部品を配置してくれるので、`setBounds()` で大きさや位置を指定する必要はありません。[51]

Java の標準ライブラリには多数のレイアウトマネージャが含まれていますが、機能が複雑なものも多いので、ここでは最低限知っておくとよいものとして、`FlowLayout` と `BorderLayout` について説明します (図 10)。

51)`setBounds()` を使っても指定が無視されます。

FlowLayout では図 10 左のように、add(部品) で窓に複数の部品を入れていくと、それらは左から順番につめられて行きます。窓の幅に入らないくらい沢山詰めた場合、あふれたものは表示されません。

BorderLayout では図 10 右のように、窓の中を上下左右および中央の 5 つの領域に分け、部品を追加する時には add(部品, 位置) という形でどこに入れるかを指定して追加します。位置の指定としては、BorderLayout.NORTH、BorderLayout.SOUTH、BorderLayout.EAST、BorderLayout.WEST、ないし BorderLayout.CENTER の 5 つの値のどれかを指定します (位置指定なしの add() を使うと中央になります)。そして、中央以外の領域はそこに部品が入ればその部品の幅や高さになりますが、部品を入れなかった場合は幅や高さが 0、つまり無いのと同じになります。

JFrame や JPanel などの領域ごとに、その領域をどのレイアウトマネージャに管理させるかを指定できます。具体的には、new FlowLayout() や new BorderLayout() でレイアウトマネージャのインスタンスを生成し、**setLayout()** でそのインスタンスを設定すればよいのです。[52]

52)ですが、JFrame であれば BorderLayout、JPanel であれば FlowLayout が初期状態で予め設定されていますから、そのレイアウトマネージャでよければ設定し直す必要はありません。それで 7 章では、わざわざ setLayout(null) によってレイアウトマネージャを外していたわけです。

例題 A1: 部品を自動配置した GUI プログラム

既によく知っている「円をドラグする」プログラムを GUI 部品と組み合わせ、配置をレイアウトマネージャに行わせてみます (図 11)。[53]

これまでのプログラムでは本体クラスを描画のための領域である JPanel のサブクラスにしていましたが、本章のでは領域が複数必要になるので、本体クラスは JFrame のサブクラスにしています。

窓本体の他に使用する GUI 部品オブジェクトには、JButton が 2 つ、JButton を入れるための JPanel、そして JPanel のサブクラスである無名内部クラスで paintComponent() を差し替えたもの (もちろん絵を表示するために) があります。[54] それほど長くないので、全体をまとめて見ていただきましょう。

53)円が描画領域のふちに掛かると当然そこで切れて見えます。

54)あとでマウスアダプタを設定するため変数 p2 に入れています。

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

public class SampleA1 extends JFrame {
  Color col = Color.RED;
```

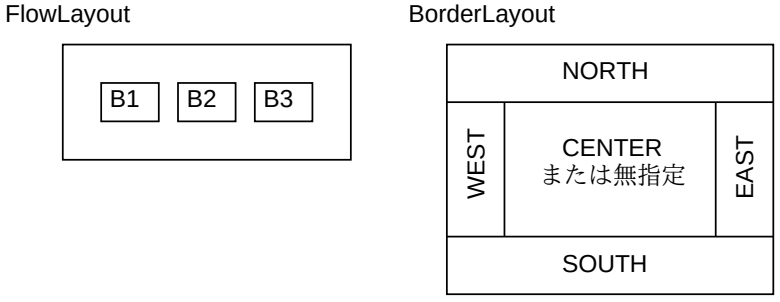


図 10: レイアウトマネージャによる配置

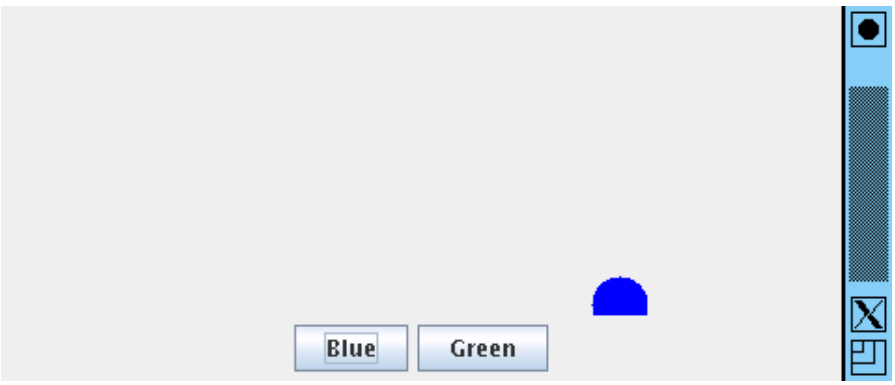


図 11: 部品を自動配置する「ドラグできる円」の画面

```
int xpos = 100, ypos = 100, rad = 15;

public SampleA1() {
  JButton b1 = new JButton("Blue");
  JButton b2 = new JButton("Green");
  JPanel p2, p1 = new JPanel(); p1.add(b1); p1.add(b2);
  add(p1, BorderLayout.SOUTH);
  add(p2 = new JPanel() {
    public void paintComponent(Graphics g) {
      setOpaque(false); g.setColor(col);
      g.fillOval(xpos-rad, ypos-rad, rad*2, rad*2);
    }
  });
  b1.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
      col = Color.BLUE; repaint();
    }
  });
  b2.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
      col = Color.GREEN; repaint();
    }
  });
  p2.addMouseMotionListener(new MouseMotionAdapter() {
    public void mouseDragged(MouseEvent evt) {
      xpos = evt.getX(); ypos = evt.getY(); repaint();
    }
  });
}
public static void main(String args[]) {
  JFrame app = new SampleA1();
  app.setDefaultCloseOperation(EXIT_ON_CLOSE);
  app.setPreferredSize(new Dimension(400, 300));
```

```
      app.pack(); app.setVisible(true);
    }
  }
```

インスタンス変数としては円の色と XY 座標と半径だけを用意しています。ボタンとボタンを入れる `JPanel` は後から参照する必要はないので、コンストラクタ中でローカル変数に入れて作業し、2 つのボタンをパネルに追加してそのパネルを窓の下側に追加しています。そして、円を入れるパネルも `paintCompnent()` を差し替えて円を描くようにした無名内部クラスで、これを窓の中央に追加してそれで終わりです。

あとはボタンのアダプタでボタンの色を変え、マウスアダプタでドラグできるようにしていますが、このあたりは既に学んだのと同じです。最後の `main()` はこれまでとだいたい同様ですが、クラス `SampleA1` のインスタンス (これが窓の一種ということになります) だけを生成することと、レイアウトマネージャを使う場合は窓の大きさは **setPreferredSize()** に **Dimension** オブジェクトを渡す形で設定しその後で **pack()** により自動配置を開始させるところが違っています。

演習 10-1 例題 `SampleA1.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変更してみなさい。

a. 内側の `JPanel` を使ってボタンを 2 つ並べていますが、そうする変わりに窓の上端 (ないし左端) に青ボタン、下端 (ないし右端) に緑ボタンが来るようにしてみなさい。

b. 窓の上側に「半径を大きくするボタン」「半径を小さくするボタン」を横に並べて追加してみなさい。

c. プログラムを終了させる「Quit」ボタンをどこかにつけてみなさい。プログラムを終了させるには「`System.exit(0)`」を実行します。

d. 上記の各種動作をボタンの代わりにメニュー項目で起動できるようにしてみなさい (メニューについては下記参照)。

e. その他ここまでに学んだ機能や API ドキュメントで探した機能を使ってプログラムのユーザインタフェースを工夫してみなさい。

11 メニューとタブ

部品の自動配置について分かったところで、今日の GUI プログラムで機能を機動するのに多く使われる**メニュー** (menu) と、複数画面を切り替えるのに使われる**タブ** (tab) について説明します。これらが使えると、かなり「普通の GUI プログラム」に近いものが作れると思います。

メニューとメニューバー

多数の機能を持つ GUI プログラムでは、画面上部に**メニューバー** (menu bar) を持ち、その中の**メニューボタン**を押すと**プルダウンメニュー** (pull-down menu) が降りて来て**メニュー項目** (menu item) が選択できるように

なっているものが標準的です。これは、「メニューバー→プルダウンメニュー→メニュー項目」のように階層化することで、多数の項目をコンパクトに収容できるからです。

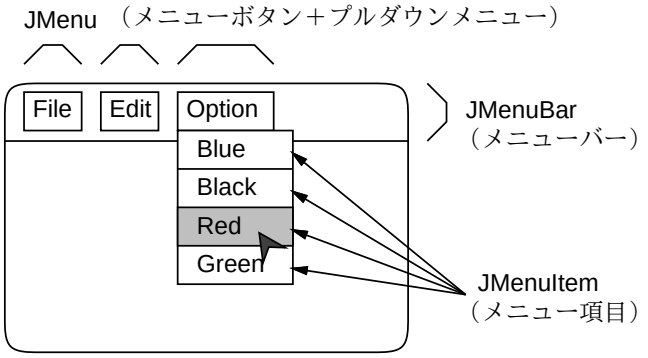


図 12: メニューバーの構成

Java でも窓に対してこのような階層化されたメニューバーが簡単につけられるようになっています (図 12)。具体的には次のようにします。[55)]

- 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。

55) JMenu や JMenuItem は new で作り出す時に JButton などと同様にラベル文字列を渡します。これらの文字列がメニューボタンやメニュー項目の中に表示されるわけです。

タブ

タブとは図 13 のように、1 つの窓内で複数の画面が切り替えられるような機能を言います。タブを使うと作業内容に応じた複数画面を持つアプリケーションが容易に作成できます。

タブを使うためには、部品 **JTabbedPane** を使います。この部品は JPanel などと同様に窓の中に配置できますが、メソッド add() を使ってその中に複数のタブを追加して行くことができます。[56)]

56) 引数としては、タブの名前となる文字列と中に入る部品 (典型的には JPanel など) を、この順で指定します。

例題: メニューとタブを使ってみる

では、メニューとタブを使った簡単な例題を示しましょう。この例題では、2 つのタブで赤い円と青い円が切り替えられますが、赤い円のときだけ円をドラグしたり「>」「<」で大きさを変更することができます。そして終了するには「File」メニューの「Quit」を選択します。まずメニューとタブを設定するコンストラクタの部分を示します。

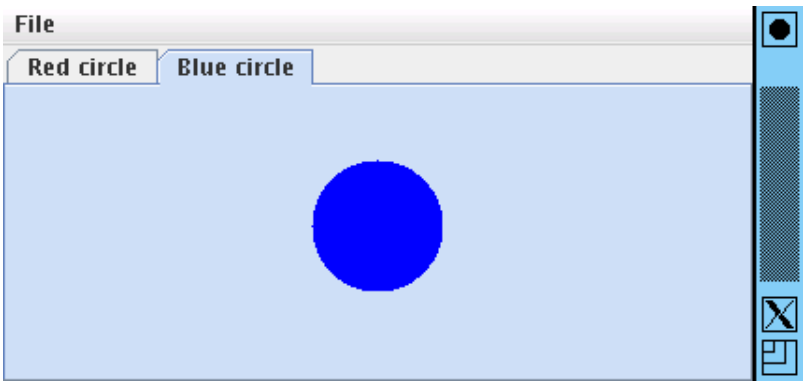


図 13: メニューとタブを持つ例題

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

public class SampleA2 extends JFrame {
  int xpos = 100, ypos = 100, rad = 15;

  public SampleA2() {
    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("Red circle", new MyPanel());
    tabs.add("Blue circle", new JPanel() {
      public void paintComponent(Graphics g) {
        setOpaque(false); g.setColor(Color.BLUE);
        g.fillOval(xpos-rad, ypos-rad, rad*2, rad*2);
      }
    });
  }
```

赤い円のタブ本体はドラグやキー操作があるので MyPanel という内部クラスで実現していますが、青い円のタブ本体は表示するだけなので JPanel のサブクラスである無名内部クラスを使って paintComponent() だけをオーバライドしています。MyPanel 以降は既に学んで来たものと同様です。

```
  class MyPanel extends JPanel {
    public MyPanel() {
      addMouseMotionListener(new MouseMotionAdapter() {
```

```
        public void mouseDragged(MouseEvent evt) {
          requestFocus();
          xpos = evt.getX(); ypos = evt.getY(); repaint();
        }
      });
      addKeyListener(new KeyAdapter() {
        public void keyPressed(KeyEvent evt) {
          if(evt.getKeyChar() == '>') { rad += 5; }
          if(evt.getKeyChar() == '<' && rad > 5) {rad-=5;}
          repaint();
        }
      });
    }
    public void paintComponent(Graphics g) {
      setOpaque(false); g.setColor(Color.RED);
      g.fillOval(xpos-rad, ypos-rad, rad*2, rad*2);
    }
  }
  public static void main(String args[]) {
    JFrame app = new SampleA2();
    app.setDefaultCloseOperation(EXIT_ON_CLOSE);
    app.setPreferredSize(new Dimension(400, 300));
    app.pack(); app.setVisible(true);
  }
}
```

12 総合制作: KWIC 表示つきエディタ

では最後の例題として、実際に役に立つ、ある程度大きなアプリケーションを作って見ましょう。文章やプログラムを書いていて、ある特定のキーワード (単語や名前) が、どのような文脈で現れているか (前後にどのような内容が接しているか) を知りたいと思うことがあります。このような表示のことを「**KWIC**(keyword in context) 表示」と呼びます (図 14)。[57] KWIC 表示があると、用語づかいを整理したり、特定の変数がどこで使われているかをチェックするのにとても役立ちます。

さらに、テキストやプログラムのファイルに対して特定語の KWIC 表示をチェックしていて、修正すべき箇所が見つかったらすぐに元のファイルの該当箇所が修正できると便利です。ここではまさにそのようなツールを作って見ましょう。まとめると、このツールの機能は次の通りです。

- テキストファイルを指定して読み込める。
- 単語を指定して、その単語の KWIC 表示が行える。
- KWIC 表示をクリックして元ファイルの当該箇所に移動できる。
- ファイルの編集と KWIC 表示の間で自由に行き来できる。
- 編集し終わったファイルを保存できる。

57) この画面は本章の LaTeX 原稿ファイルで「プログラム」を KWIC 表示しているところです。

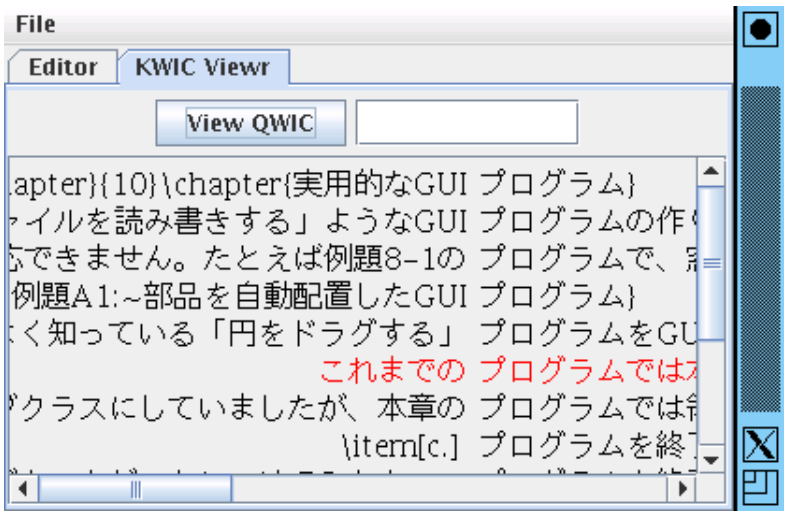


図 14: KWIC 表示つきエディタ (KWIC 画面)

図 15: KWIC 表示つきエディタ (編集画面)

図 15 は、図 14 の KWIC 画面で 1 行を選択してからその状態で元ファイルの編集タブに切り替えたようすを示しています。確かに先に選んだ行のところが表示されているのが分かります。

200 行弱のプログラムですが、これまでの例題よりはだいぶ長いですし、インスタンス変数もかなり多いので、順序立てて説明して行きましょう。

インスタンス変数群

インスタンス変数群の説明は次の通りです。

- `Font fn` — 表示に使うフォントオブジェクト。
- **FontMetrics** `fm` — フォントの各文字の大きさを保持するオブジェクト。実際に表示を行う環境にならないと取得できないので、最初は `null` を入れてあります。

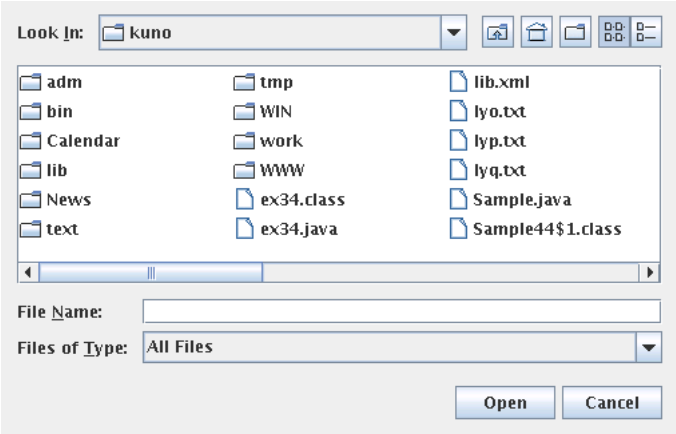


図 16: ファイル選択ダイアログ

- **JFileChooser** `fc` — 読み書きするファイルを選択する GUI 部品。これでファイルを選択しているところを図 16 に示します。
- `JLabel l1` — エラーメッセージなどを表示するラベル。窓の下端に配置します。
- `JTextField f1` — KWIC の対象となる語を入力するための入力覧。
- `KWICPanel k1` — KWIC 表示を行う部品。`JPanel` のサブクラスで、このプログラムの一部として作成します。
- `JTextArea a1` — テキストの表示と編集を行うための GUI 部品。
- `JScrollPane a1s` — `JTextArea` を中に入れることで上下/左右のスクロール機能を持たせる。
- `JTabbedPane tabs` — 複数タブを切り替え可能にする。
- `JFrame app` — `JFileChooser` を開く時に窓オブジェクトを渡す必要があるのでそれを保持するための変数。コンストラクタの先頭で `this`(つまり `SampleA3` のインスタンス) を格納しています。
- `ArrayList<String> lines` — ファイルの各行を格納。
- `ArrayList<String> left` — KWIC 表示の「当該語に隣接する左側」の文字列を当該語の出現毎に格納。
- `ArrayList<String> right` — KWIC 表示の「当該語以降の部分」の文字列を当該語の出現毎に格納。
- `ArrayList<Integer> lref` — KWIC 表示の当該語が元のファイルの何行目にあったか当該語の出現毎に格納。編集画面を選ばれた当該語の位置にスクロールさせるのに使用します。
- `ArrayList<Integer> cref` — KWIC 表示の当該語が元のファイルの何文字目にあったか当該語の出現毎に格納。編集画面の文字カーソル (キャレット) を当該語の箇所に設定するのに使用します。
- `int lnum` — KWIC 表示で現在何行目 (何番目の当該語の出現) が選ばれている (赤く表示されている) かを保持。どの語も選ばれていないときは `-1` が入っています。

- `boolean changed` — `true` なら編集画面で何らかの変更を行ったことを示す。タブ切り替え時にこの値が `true` ならファイル内容が変化しているので KWIC データを作り直します。
- `boolean selected` — `true` なら編集画面でどれかの行を選択したことを示す。タブ切り替え時にこの値が `true` なら編集画面を選択位置までスクロールしてカーソル位置を設定します。[58]
- `String word` — KWIC 表示に使う文字列。空文字列なら KWIC 表示は行いません。

58)単に確認のために KWIC に切り替えて戻した時には編集の邪魔にならないよう、スクロール位置やカーソル位置は変化させないようにしています。

`import`、クラスの冒頭、インスタンス変数の宣言部分を示します。

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

public class SampleA3 extends JFrame {
  Font fn = new Font("Serif", Font.PLAIN, 16);
  FontMetrics fm = null;
  JFileChooser fc = new JFileChooser();
  JLabel l1 = new JLabel("file viewer.");
  JTextField f1 = new JTextField();
  KWICPanel k1 = new KWICPanel();
  JTextArea a1 = new JTextArea();
  JScrollPane a1s = new JScrollPane(a1);
  JTabbedPane tabs = new JTabbedPane();
  JFrame app;
  ArrayList<String> lines = new ArrayList<String>();
  ArrayList<String> left = new ArrayList<String>();
  ArrayList<String> right = new ArrayList<String>();
  ArrayList<Integer> lref = new ArrayList<Integer>();
  ArrayList<Integer> cref = new ArrayList<Integer>();
  int lnum = -1;
  boolean changed = false, selected = false;
  String word = "";
```

メニューバーの設定

次に、コンストラクタの冒頭からはじめて、メニューバーの設定のところまでを示します。先頭で変数 `app` に `this` を入れ、また `JTextArea` が使用するフォントを `fn` に設定します。その後、File メニューを作り、Open、Save、Quit の 3 項目を追加します。前 2 者の動作は下請けメソッドを呼ぶだけ、[59] Quit については **System.exit(0)** を呼ぶだけです。[60]

```
  public SampleA3() {
```

59)`KWICPanel` の メ ソッド `changeSize()` は KWIC 表示の中身を作り直します。ファイルが変更されたら当然 KWIC 表示も別になりますから。

60)このクラスメソッドは、プログラムをすぐに終了させる働きを持ちます。

```
app = this; a1.setFont(fn);
JMenuBar mbar = new JMenuBar(); setJMenuBar(mbar);
JMenu m1 = new JMenu("File"); mbar.add(m1);
JMenuItem i1 = new JMenuItem("Open"); m1.add(i1);
i1.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent evt) {
    l1.setText(""); readFile(); k1.changeSize(); repaint();
  }
});
JMenuItem i2 = new JMenuItem("Save"); m1.add(i2);
i2.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent evt) {
    l1.setText(""); writeFile(); repaint();
  }
});
m1.addSeparator();
JMenuItem i3 = new JMenuItem("Quit"); m1.add(i3);
i3.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent evt) {
    System.exit(0);
  }
});
```

GUI 部品の配置

メニューの設定が終わったら、レイアウトマネージャ等を使って GUI 部品を配置します。まず「View KWIC」ボタンを作成し、先に作ってあった入力覧とともにパネルに入れます。入力覧については単語を入れるためにやや横長の大きさになるよう **setPreferredSize()** で設定します。

もう 1 つパネルを作り、こちらは `BorderLayout` を使うように設定して、上側に先のパネル、中央に `KWICPanel` を入れます。そして、`tabs` に対して 1 番目のタブとして `a1s` を入れた編集画面、2 番目のタブとしてここで作ったパネルを入れた KWIC 画面を追加し、`tabs` そのものは窓の中央、メッセージ表示ラベルは窓の下端に入れます。

続いて、ボタンが押された時の動作として入力覧から文字列を取り出して `word` に保存し、`makeKWIC()` と `changeSize()` を呼ぶようにします。また、`JTextArea` については何らかの内容変更が行われた場合はすべて `changed` を `true` にするように、`DocumentListener` を設定します。[61)]

```
JButton b1 = new JButton("View KWIC");
JPanel p1 = new JPanel(); p1.add(b1); p1.add(f1);
f1.setPreferredSize(new Dimension(120, 25));
JPanel p2 = new JPanel();
p2.setLayout(new BorderLayout());
p2.add(p1, BorderLayout.NORTH);
p2.add(new JScrollPane(k1));
tabs.addTab("Editor", a1s); tabs.add("KWIC Viewr", p2);
```

61) **DocumentListener** は文書の変更があったら通知を受け取るような機能に対応するアダプタのインタフェースです。`JTextArea` から **getDocument()** で中に入っている **Document** オブジェクトを取り出し、それに対して **addDocumentListener()** を呼んでアダプタを設定する必要があります。

```
add(tabs); add(l1, BorderLayout.SOUTH);
b1.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent evt) {
    if(!f1.getText().equals("")) { word = f1.getText(); }
    makeKWIC(word); f1.setText("");
    l1.setText(""); k1.changeSize(); repaint();
  }
});
a1.getDocument().addDocumentListener(new DocumentListener(){
  public void changedUpdate(DocumentEvent evt) {
    changed = true;
  }
  public void insertUpdate(DocumentEvent evt) {
    changed = true;
  }
  public void removeUpdate(DocumentEvent evt) {
    changed = true;
  }
});
```

タブ切り替え時の処理

タブが切り替わる時にも動作が必要なので、tabs に対して **ChangeListener** インタフェースを実装したアダプタを設定します。切り替わりがあった時は、JTabbedPane のインスタンスメソッド **getSelectedIndex()** を使って、今どのタブに切り替わったのかを調べます。

KWIC 画面になった場合でファイル内容が変更されている場合は、lines を一旦クリアして JTextArea の中身の文字列を改行箇所で区切って作った各行を入れ直します。[62][63]

逆に、編集画面になった場合で KWIC 画面で特定行が選択されていた場合は JTextArea のカーソル位置を設定し、また JScrollPane の見える位置を管理する **JViewport** オブジェクトを取り出し、指定位置が先頭になるようにスクロールします。[64] ここの処理では FontMetrics が取れていなかったり等の問題が起きた時まとめて受け止めて処理する (実際にはスクロールをあきらめれば済むのでエラーを無視する) ために try 文を使っています。

62)ちょっと編集しただけでも全部作り直すのは無駄みたいですが、この方が簡単です。この処理はタブの切り替わり時にしか行わないので、処理に時間が掛かって遅くなるという心配はあまりありません。

63)requestFocus() を呼んでいるのは、入力可能にしないとキャレットが表示されないためです。

64)位置は画面上のピクセル単位で指定するので、FontMetrics オブジェクトからフォントの高さを指定して行数を掛けて位置を求めています。2 行差し引いているのは、少し上の行まで見えた方が使いやすいからです。

```
tabs.addChangeListener(new ChangeListener() {
  public void stateChanged(ChangeEvent evt) {
    if(tabs.getSelectedIndex() == 1 && changed) {
      changed = false; lines.clear(); lnum = -1;
      String[] a = a1.getText().split("\n");
      for(String s: a) { lines.add(s); }
      if(!word.equals("")) { makeKWIC(word); }
    } else if(tabs.getSelectedIndex() == 0 && selected) {
      selected = false;
      try {
```

```
            a1.setCaretPosition(cref.get(lnum));
            a1.requestFocus();
            JViewport v = a1s.getViewport();
            int p = (lref.get(lnum)-2) * fm.getHeight();;
            v.setViewPosition(new Point(0, p));
          } catch(Exception ex) { }
        }
      }
    });
  }
```

ファイルの読み書き

ファイルの読み書きはどちらも原理的には同じで、JFileChooser のメソッド **showOpenDialog()**、**showSaveDialog()** を呼ぶことで読み込み/書き出し用のダイアログが開かれます。この時、パラメタで窓オブジェクトを渡す必要があるので、変数 app の値を渡します。いずれの場合も、OK なら JFileChooser.APPROVE_OPTION が返されるので、**getSelectedFile()** で File オブジェクトを取り出してストリームを開き、読み書きを行います。[65]

[65] JTextArea にはストリームを渡すと読み書きをすべてやってくれるメソッド read() と write() があるのですが、ここではサンプルとして自前でループにより読み書きをするようにしました。

```
  public void readFile() {
    try {
      int ret = fc.showOpenDialog(app); l1.setText("");
      if(ret != JFileChooser.APPROVE_OPTION) {
        throw new Exception();
      }
      Scanner sc = new Scanner(fc.getSelectedFile(),
                               "JISAutoDetect");
      lines.clear(); left.clear(); right.clear();
      while(sc.hasNextLine()) { lines.add(sc.nextLine()); }
      sc.close();
      a1.setText("");
      for(String s: lines) { a1.append(s + "\n"); }
    } catch(Exception ex) {
      l1.setText("erro in reading file.");
    }
  }

  public void writeFile() {
    try {
      int ret = fc.showSaveDialog(app); l1.setText("");
      if(ret != JFileChooser.APPROVE_OPTION) {
        throw new Exception();
      }
      PrintStream pr = new PrintStream(fc.getSelectedFile(),
                                       "EUC_JP");
      String[] a = a1.getText().split("\n");
```

```
      for(String s: a) { pr.println(s); }
      pr.close();
    } catch(Exception ex) {
      l1.setText("erro in reading file.");
    }
  }
```

KWIC 表示のデータ構造の計算

いよいよ、KWIC 表示のためのデータ構造である `left`、`right`、`lref`、`cref` を正しく計算する一番肝心なメソッド `makeKWIC()` を見ることにしましょう。単語は行をまたがらないので、行単位で処理をします (外側の for ループ)。その中で、処理する行を変数 `t` に入れます。文字列のメソッド `indexOf(`s`,` p`)` は、その文字列中で位置 p から初めて最初に文字列 s が (部分文字列として) 現れる位置を探してくれます (見つからない場合は-1 を返します)。

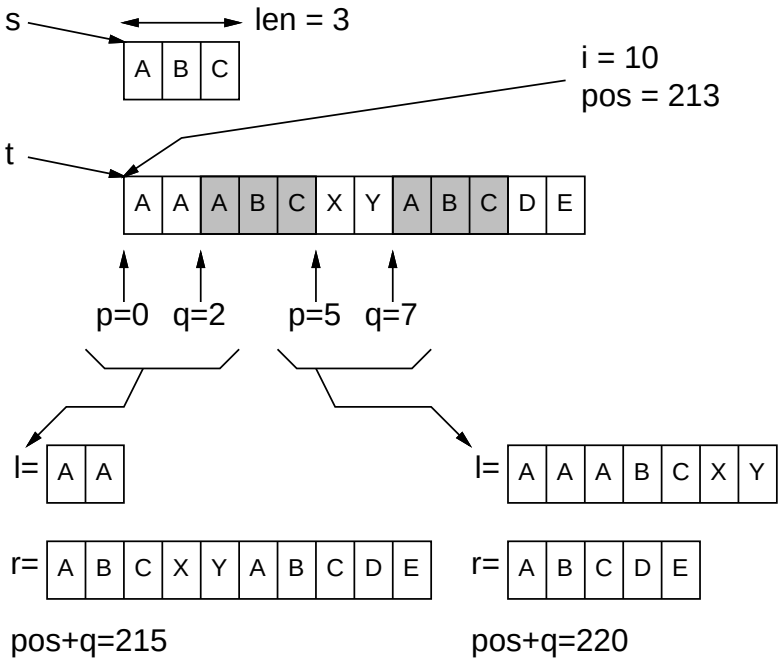


図 17: KWIC 情報の計算

図 17 は、`s="ABC"`に対する KWIC 表示のデータを作成していて、`i=10` 行目を処理している様子を示しています。先頭からこの行の手前までには合計 `pos=213` 文字があったものとします。

まず `p=0` から `indexOf()` で探索すると `q=2` のところに文字列があります。そこで、この位置で行を分割して左側を変数 `l`、右側を変数 `r` に入れます。[66]

ただし、それだけだとその位置が行頭や行末であった場合に前後の文脈が分からないので、`l` の左には 1 つ前の行 (`i-1` 行)、`r` の右には 1 つ後の行 (`i+1` 行) の文字列をくっつけ、それらを並び `left` と `right` に追加します。

66)文字列のインスタンスメソッド `substring()` は、文字列中の位置を 1 つ指定された場合はそれ以後の文字列、2 つ指定された場合はその 2 つの位置の間にある文字列を取り出します。

また、現在の行番号 i=10 と分割位置 pos+q=215 をそれぞれ lref、cref に追加します。

ここまででこの位置はいいのですが、この行にはまだ s に等しい文字列があります。そこで、位置 q+len=5 を新たに p として indexOf() で次の位置を探し、同様に処理します。これを、これ以上 indexOf() で次の位置が見つからなくなるまで反復するわけです。

```
public void makeKWIC(String s) {
  if(s.length() == 0) { return; }
  int n = lines.size(), pos = 0, len = s.length();
  left.clear(); right.clear(); lref.clear(); cref.clear();
  for(int i = 0; i < n; ++i) {
    String l, r, t = lines.get(i);
    int p = 0;
    while(true) {
      int q = t.indexOf(s, p); if(q < 0) break;
      l = t.substring(0, q); r = t.substring(q);
      if(i > 0) { l = lines.get(i-1) + l; }
      if(i < n-1) { r = r + lines.get(i+1); }
      left.add(l); right.add(r); lref.add(i); cref.add(pos+q);
      p = q + len;
    }
    pos += t.length() + 1;
  }
}
```

このようにしてデータ構造ができていれば、KWIC 表示を行うのは難しくありません。KWIC 表示とそれに対するマウスクリックを扱うクラス KWICPanel を見てみましょう。

まず、コンストラクタではマウスアダプタを設定しています。その中では、クリック時に何行目をクリックしたかをマウス Y 座標をフォントの高さで割ることで求めて変数 lnum に記録し、クリックがあったことを示すフラグ selected を立てます。

```
class KWICPanel extends JPanel {
  public KWICPanel() {
    setOpaque(false);
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent evt) {
        if(fm == null) { return; }
        lnum = (evt.getY() / fm.getHeight());
        if(lnum<0 || lnum>left.size()-1) { lnum = -1; }
        selected = true; repaint();
      }
    });
  }
```

次は paintComponent() による表示ですが、まず最初に fm が null であれば Graphics オブジェクトのメソッド g.getFontMetrics() でフォントの

サイズ情報を取得しておきます。[67] 続いてフォントや色を設定し、1 行の高さを h に取り出します。

次は、left が空っぽなら KWIC 表示の準備はできていないので (まだ単語が指定されていない場合などが相当します)、とりあえず元の行をそのまま表示するようにします。

そうでない場合は、単語の各出現ごとに left、right、lref、cref が用意されているわけですから、left、right から取り出した文字列を左右に並べて表示します。このとき、右側の文字列が X 座標 250 の位置に縦に揃うようにするため、左側の文字列は FontMetrics オブジェクトのインスタンスメソッド **getStringWidth()** を用いて幅を調べ、右端が (少しあけて)X 座標 245 の位置に来るように表示しています。また、この際この出現が前にクリックした位置であれば、色を赤に切り替えて表示します。

```
public void paintComponent(Graphics g) {
  if(fm == null) { g.setFont(fn); fm = g.getFontMetrics(); }
  g.setFont(fn); g.setColor(Color.BLACK);
  int h = fm.getHeight();
  if(left.size() == 0) {
    for(int i = 0; i < lines.size(); ++i) {
      g.drawString(lines.get(i), 3, (i+1)*h);
    }
  } else {
    for(int i = 0; i < left.size(); ++i) {
      if(i == lnum) { g.setColor(Color.RED); }
      String l = left.get(i);
      g.drawString(l, 245 - fm.stringWidth(l), (i+1)*h);
      g.drawString(right.get(i), 250, (i+1)*h);
      if(i == lnum) { g.setColor(Color.BLACK); }
    }
  }
}
```

KWICPanel の最後のメソッド changeSize() は、KWIC 表示の縦横の大きさを計算して setPreferredSize() で自身の大きさを設定するためのものです。[68] こちらも left が空かどうかで枝分かれし、高さは表示する行数にフォントの高さを掛けた値、幅は stringWidth() で調べた幅の最大値によって計算します。

```
public void changeSize() {
  if(fm == null) return;
  if(left.size() == 0) {
    int w = 0, h = fm.getHeight()*(1+lines.size());
    for(String s:lines) w = Math.max(w, fm.stringWidth(s));
    setPreferredSize(new Dimension(w+5, h));
  } else {
    int w = 0, h = fm.getHeight()*(1+left.size());
    for(String s:right) w = Math.max(w, fm.stringWidth(s));
```

67) フォントのサイズ情報は Graphics オブジェクトから取るしかないので、予め用意しておくことはできません。このため、最初に描画するときに取るようにしているわけです。

68) JScrollPane はスクロールバーの位置や大きさを調節してスクロール機能を提供してくれますが、そのためには中に入っているものの正しい大きさが必要なわけです。

```
        setPreferredSize(new Dimension(w+255, h));
      }
      revalidate();
    }
  }

  public static void main(String[] args) {
    JFrame app = new SampleA3();
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setPreferredSize(new Dimension(400, 300));
    app.pack(); app.setVisible(true);
  }
}
```

そして最後に main() がありますが、これはこれまでと変わっていません。どうでしたか、大変だったでしょうか? でもこれくらい書けば本当に役に立つ (あると実際にありがたい) アプリケーションが作れるわけです。

演習 A-2 例題 SampleA3.java をそのまま動かしなさい。動いたら次のような改良を行ってみなさい。

a. 現在は KWIC 表示の単語 (文字列) は 1 つだけですが、複数の語について検討したいこともあります。そこで、単語を複数指定しておき、「どの単語」と指定したらその単語の KWIC 表示に切り替わるようにしてみなさい。[69)]

b. 別の方法として、複数の単語を指定すると、それらの単語がまざって「どれか 1 つ」が出現するごとに 1 行ずつの KWIC 表示が行われるようにしてみなさい。

c. 文字列を探索した後「置換」するのはよくやる操作です。このとき、全部を一括置換するのではなく、選んだところだけ置換することもよくあります。そこで、KWIC 表示の上で置換したいところをクリックで選択し (たとえば青く表示)、OK だと思ったらそれらの出現箇所について指定した文字列に一括置換する、という機能をつけてみなさい。

d. その他、自分があったら便利だと思う機能を考え、追加してみなさい。

69) 単語をメニューやボタンで選択できるようにする、単語 1 つにつき 1 つずつ KWIC 表示のタブができる、などの設計が考えられます。

13 さいごに

最初は円を 1 つ表示するだけのところから始めて、さまざまな機能や、それらの機能を使いこなすための「考え方」を、自分のレパートリーに追加していくことで、かなり複雑なプログラムでも作れるようになりました。最後の例題はとても手が出ないと思うかも知れませんが、これもずっと短くて簡潔なバージョンから徐々に改良していってこのようになったものです。皆様も同様にして、「こういうツールが必要だ」と思ったらそのエッセンスの部分だけ少し作ってみる、ということから始めれば、きっと自分の実力の範囲で役にたつプログラムが作れると思います。