応用Javaプログラミング
第4回

1. GUIプログラミング1
2. 例外処理
3. Template Methodデザインパターン

GUIの簡単な例

次のような貯金残高を計算できる電卓を作ってみよう.

アプリケーション

貯金電卓の部品

1枚のパネルに5つのラベルと3つのテキスト・フィールドと
2つのボタンを，5行2列に貼り付けたことになっている．

●ラベル：文字列の表示に使われる部品．
●テキスト・フィールド：キーボードからの入力に使われる部品．
●ボタン：クリックするとイベントが発生する．それに応じた処理が可能．

前頁のアプリケーションはこのパネルをフレームに貼ったもの．

よく使われるGUI部品に対応するライブラリ・クラス

部品名	クラス名	
フレーム	javax.swing.JFrame	java.awt.Frame
ラベル	javax.swing.JLabel	java.awt.Label
ボタン	javax.swing.JButton	java.awt.Button
テキスト・フィールド	javax.swing.JTextField	java.awt.TextField
テキスト・エリア	javax.swing.JTextArea	java.awt.TextArea
パネル	javax.swing.JPanel	java.awt.Panel
ラジオ・ボタン	javax.swing.JRadioButton	java.awt.RadioButton

Swingライブラリ

AWTライブラリ

Swing ライブラリの利点

- 豊富で便利なユーザインターフェース
- プラットフォームの依存度が少ない
- 一貫したフィーリング

Abstract Window Toolkit の略

貯金電卓の部品

JLabel
tfLabel[0],
tfLabel[1],
tfLabel[2],
tNameLabel;

JTextField
textField[0] ,
textField[1] ,
textField[2] ;

JLabel totalLabel;

JButton
button[0];

JButton button[1];

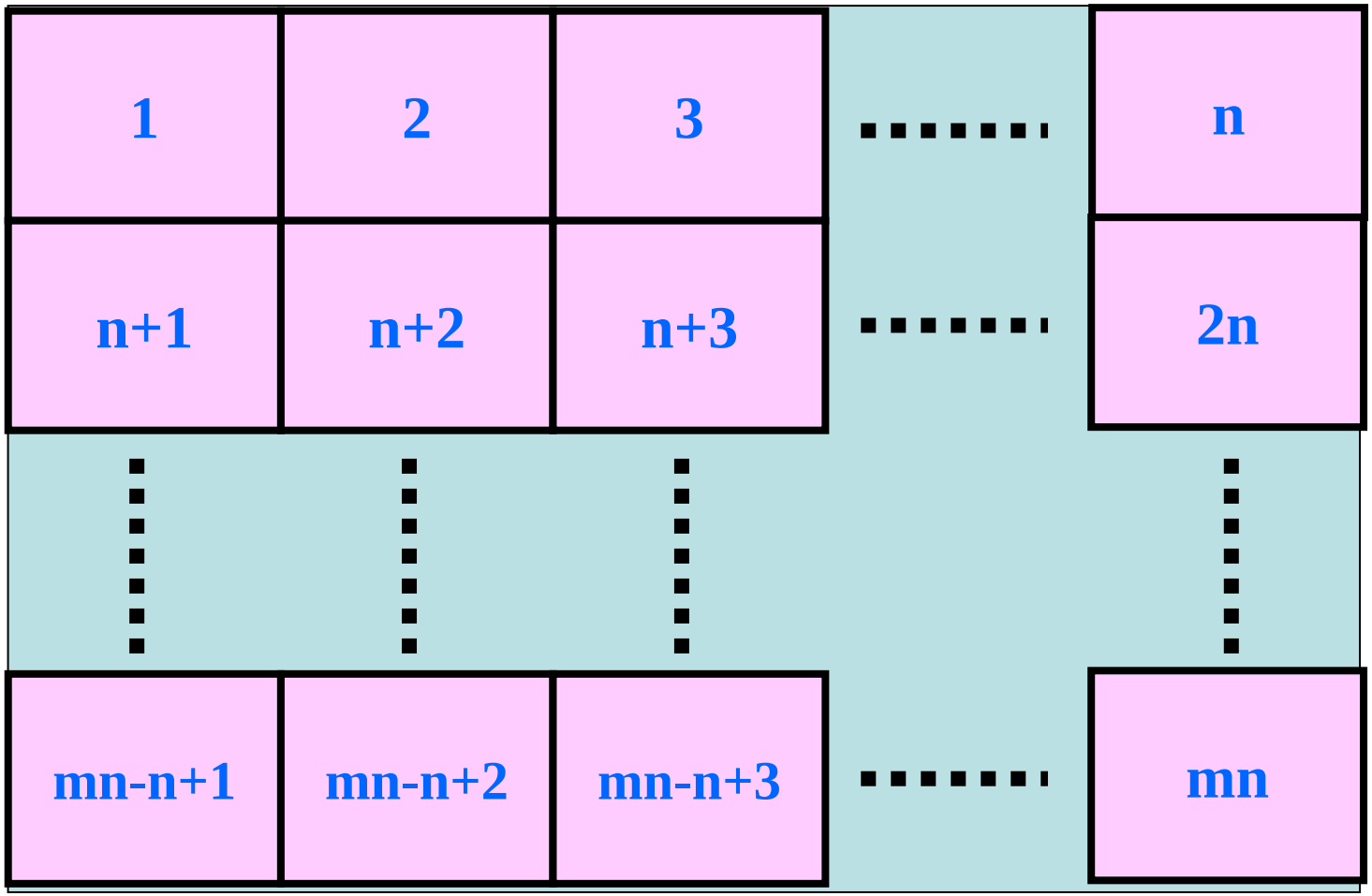
1枚のパネルに5つのラベルと3つのテキスト・フィールドと
2つのボタンを，5行2列に貼り付けたことになっている.

∴ この電卓を10個の部品を持ったパネルとして見てよい.

∴ javax.swing.JPanelの子クラスを作ればよい.

貯金電卓で使ったレイアウト

- GridLayoutは，下のようなレイアウト：



GridLayoutはコンテナの領域を mn等分して部品に与える．各領域に1つの部品が入る．

オブジェクト指向によるアプリケーション開発とは

変更されない箇所を軸に、頻繁に変更されるであろう箇所を
クラスやインターフェースに抽出するプログラミングスタイルである.

例: 簡単なGUI作成問題.

問: 頻繁に変更されるであろう箇所は何か？

答: ① 使われる部品 → JButton, JPanel, JFrame など

② 部品の色 → Color

③ 部品の配置方法 → GridLayout, FlowLayout, など

④ ボタンをクリックしたときの処理

⑤ フレームを閉じたときの処理

…などなど

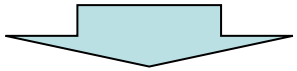
④, ⑤ → ActionListener, WindowListener, など

Javaには，これらの汎用的なクラスやインターフェースは用意されている.

我々も汎用的なクラスを書きたい!

ボタンがクリックされたときの処理

ボタンがクリックされたときの処理は**応用依存**



ライブラリ（抽象的なプログラム）を作成するときに，
その処理内容を定めることができない．

Point: その処理の内容が定まらなくても，
その処理を行うメソッドの呼び出し方がわかれば，
ライブラリを作成できる．

大抵，インターフェースにまとめる．

ActionListenerに含まれている唯一のメソッド：

Javaでは，そのインターフェースは
ActionListenerというもの．

```
void actionPerformed(ActionEvent e);
```

応用側でこのメソッドの中身（ボタンがクリックされたときの処理）を書く

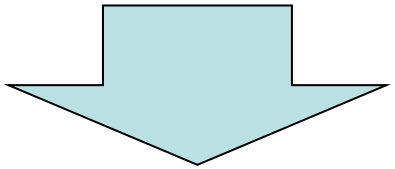
オブジェクト指向によるアプリケーション開発とは

変更されない箇所を軸に、頻繁に変更されるであろう箇所を
クラスやインターフェースに抽出するプログラミングスタイルである.

例：簡単なGUI作成問題.

問：変更されないであろう箇所は何か？
答：処理順序：フィールド（部品の名前など）の初期化，
部品の生成と配置，リスナーの生成と登録.

この処理順序はほとんどの簡単なGUIアプリケーションにおいて
共通と言えそうなので，固定しておきたい.



次頁のクラスを用意して，多くの簡単なGUI
アプリケーションで使えるようにする.

簡単なGUIアプリケーションの ひな型(template)

```
package myGUI ;

import  javax.swing.JPanel ;

public  abstract class AbstractGUI extends JPanel {
      //GUIを開始するメソッド．finalにしてあるので，子クラスでoverrideできない．
      public final void start( ) {
            initialize( ) ; //フィールド等を初期化
            arrangeComponents( ) ; //部品を生成してパネルに配置するメソッド
            addListeners( ) ; //部品に必要な自作リスナーを登録するメソッド
      } ;

      //フィールド等を初期化するメソッド．
      public void initialize( ) { }

      //部品を生成してパネルに配置メソッド
      public abstract void arrangeComponents( ) ;

      //部品に必要な自作リスナーを登録するメソッド
      public abstract void addListeners( )
}
```

必要に応じて子クラスで**override**.

この2つのメソッドはアプリケーションに依存するので，実際のアプリケーションを表す子クラスで実装することにする．

AbstractGUIを継承した電卓（全体の流れ）

```
package savingCal ;

import  myGUI.AbstractGUI ;
import  javax.swing.* ; //Swingライブラリの提供するJLabel等を使うので
import  java.awt.* ; // AWTライブラリの提供するレイアウト等を使うので
import  java.awt.event.* ;// ボタンが押されたときに発生するイベントを
                          // 処理する必要があるので
public  class  SavingCal  extends  AbstractGUI  {
    // 下記の4つからなる.
    ① 複数のメソッドで使う部品の宣言.
    ② initializeメソッドの中身（override）
    ③ arrangeComponentsメソッドの中身（implement）.
    ④ addListenersメソッドの中身（implement）.
}
```

①複数のメソッドで使う部品の宣言

```
private  JTextField[ ]  textFields
private  JLabel  totalLabel ;
private  Jbutton[ ]  buttons ;
private  String[ ]  tfNames ;
private  String[ ]  btnNames ;
```

getterメソッドも書いておいたほうがいいが,
ここで省略.

②initializeメソッドのoverride

```
// フィールドを初期化するメソッド.
@Override
public void initialize ( ) {
        String[ ] tempArray = {"年利率:", "年数:", "積立月額(円):"};
        tfNames = tempArray.clone( ) ;
        String[ ] tempArray2 = {"クリア", "計算"};
        btnNames = tempArray2.clone( ) ;
}
```

③ arrangeComponentsの実装

```
@Override
public void arrangeComponents ( ) {
    // パネルに部品を追加するために,
    // 部品の配置法（レイアウト）を設定する必要がある
    GridLayout  myLayout = new GridLayout(5, 2); // 部品の配置法
                                                 // として5行2列の格子を生成する
    setLayout(myLayout); // パネルへの部品配置をmyLayoutに設定


    //テキストフィールドの配列を生成
    textFields = new JTextField[tfNames.length] ;

    //ボタンの配列を生成
    buttons = new JButton[btnNames.length] ;
```

次頁に続く

③ arrangeComponentsの実装（続）

```
        // 最初の3行に部品を配置する
        for (int i = 0 ; i < tfNames.length ; i++) {
            add( new JLabel(tfNames[i], JLabel.RIGHT) ) ;
            textFields[i] = new JTextField(15) ;
            add(textFields[i]) ;
        }

        //4行目に部品を生成して配置する
        add( new JLabel("利子込み合計（円）: ", ], JLabel.RIGHT) ) ;
        totalLabel = new JLabel("") ;
        add(totalLabel) ;
        totalLabel.setOpaque(true) ;   // 背景色が見えるようにする
        totalLabel. setBackground(Color.YELLOW); //背景色を黄色にセット
        totalLabel.setForeground(Color.RED); //前景色を赤色にセット

        // 5行目の部品を生成して配置する
        for (int i = 0 ; i < btnNames.length ; i++) {
            buttons[i] = new JButton(btnNames[i]) ;
            add( buttons[i] ) ;
        }
} //arrangeComponentsメソッドの終わり
```

右寄せ

Javaですぐに使える色

Color.BLACK	（黒）
Color.BLUE	（青）
Color.CYAN	（シアン）
Color.DARK_GRAY	（暗い灰色）
Color.GRAY	（灰色）
Color.GREEN	（緑）
Color.LIGHT_GRAY	（明るい灰色）
Color.MAGENTA	（マゼンタ）
Color.ORANGE	（オレンジ）
Color.PINK	（ピンク）
Color.RED	（赤）
Color.WHITE	（白）
Color.YELLOW	（黄）

自作リスナークラス
を定義する前に，
Javaで（自分で作成し
なくても）使える色と
レイアウトを紹介する．

注：他の色を使いたい場合
自分でその色を作成する
ことができるが，今回は
その作成法を紹介しない．
次回に紹介する予定．

Javaで使えるレイアウト

GridLayout	部品をタイル状に配置する．すべての部品のサイズは同じになる．
BorderLayout	描画領域を上，下，左，右，中央の五つに分けてそれぞれに部品を配置する．それぞれはNORTH, SOUTH, WEST, EAST, CENTERで指定される．
FlowLayout	部品を横1行に並べる．ただし，横幅が足りない場合，改行して部品を配置し続く．
CardLayout	部品をカードをめくるように順番に配置する．
BoxLayout	部品を単一行，もしくは単一列に配置する．ただし横に配置するとき，横幅が足りなくても改行しない．
GridBagLayout	異なる大きさの部品をタイル状に配置する．

④ addListenersの実装

```
@Override
public void addListeners ( ) {
    addClearButtonListener( ) ; //「クリア」ボタンにリスナーを生成して付ける
    addCalcButtonListener( ) ; //「計算」ボタンにリスナーを生成して付ける
} //addListenersメソッドの終わり
```

```
private void addClearButtonListener ( ) {
    if ( buttons.length == 0 ) return ;
    // 無名リスナークラスのインスタンスを生成してクリア・ボタンに登録
    buttons[0].addActionListener(new ActionListener( ) {
        public void actionPerformed(ActionEvent e) {
            for (int i = 0 ; i < textFields.length ; i++)
                textFields[i].setText(“”) ; //テキストフィールドを
            totalLabel.setText("") ; //利子込み積立合計額をクリア
        }
    }) ;
} //addClearButtonListenerメソッドの終わり
```

setText: ラベルやテキスト・フィールドに指定の文字列をセットするメソッド

```
private void addCalcButtonListener ( ) {
      if ( buttons.length <= 1 ) return ;

      //無名リスナークラスのインスタンスを生成して計算ボタンに登録
      buttons[1].addActionListener(new ActionListener( ) {
          public void actionPerformed(ActionEvent e) {
              // テキスト・フィールドの値を取り込む
              double rate = Double.parseDouble(textFields[0].getText( )) ;
              int  years = Integer.parseInt(textFields[1].getText( )) ;
              int  monthsaving = Integer.parseInt(textFields[2].getText( )) ;

              // 次に，利子込み積立合計額を計算する
              int months = years * 12 ;                    //積立回数
              double  monthRate = 1.0 + rate / 1200.0; //月利率
              double  total = 0;                                  //合計金額
              for (int  i = 0 ; i < months ; i++) {
                  total += monthsaving ;
                  total *= monthRate ;
              }
              totalLabel.setText(String.valueOf(total)) ;// 合計をラベルとして表示
          }    }) ;
} //addCalcButtonListenerメソッドの終わり
```

getText: ラベルやテキスト・フィールドにある文字列をゲットするメソッド

Javaで用意された（一部の）
リスナー・インターフェース

インターフェース名	メソッド	部品の種類
ActionListener	actionPerformed	ボタン ラジオ・ボタン テキスト・フィールド
ItemListener	itemStateChanged	チョイス チェック・ボックス
AdjustmentListener	adjustmentValueChanged	スクロール・バー
MouseListener	mousePressed mouseReleased mouseClicked mouseEntered mouseExited	描画可能な GUIコンテナ （パネル等）
MouseMotionListener	mouseDragged mouseMoved	パネル等
ChangeListener	stateChanged	スライダー等

ドライバークラス

```
package testers ;

import savingCal.SavingCal ;
import  javax.swing.JFrame ;

public class FrameTester {
    //インスタンスの生成を禁止するためのコンストラクタ
     private FrameTester( ) { }
     public static void main(String arg[ ]) {
          EventQueue.invokeLater ( new Runnable( ) {
             public void run( ) {
                JFrame  frame = new JFrame("貯金電卓") ;// フレームを生成
                frame.setSize(300, 200) ;
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                SavingCal c = new SavingCal( ) ;  frame.add(c) ;  c.start( ) ;
                frame.setLocationByPlatform( true ) ;
                frame.setVisible( true ) ;
             }
          }) ;
    }
}
```

フレームをマウスで閉じれるように

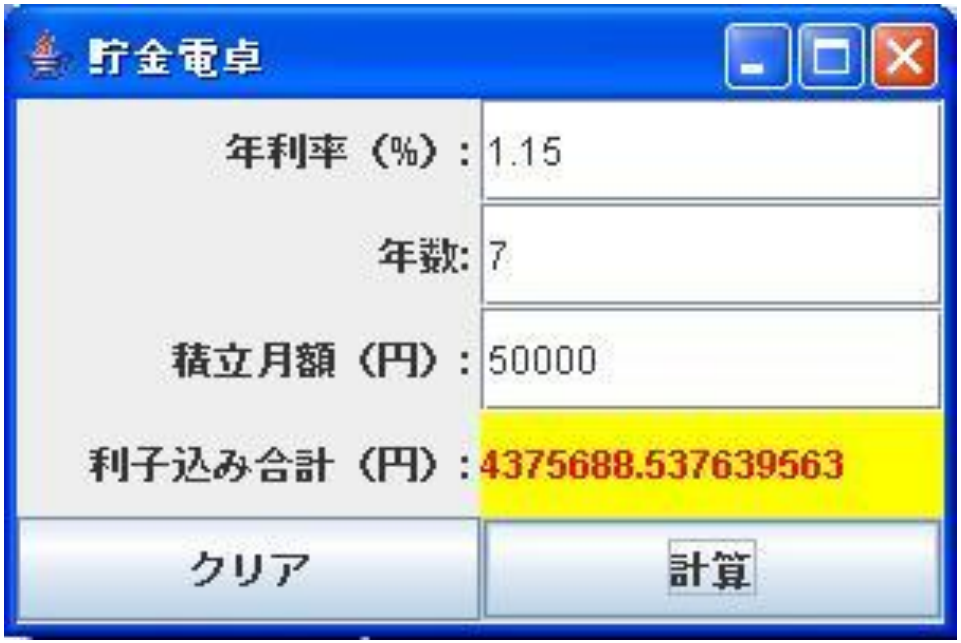
フレームを見えるようにする.

例外処理（try-catch-finally）

先のプログラムは

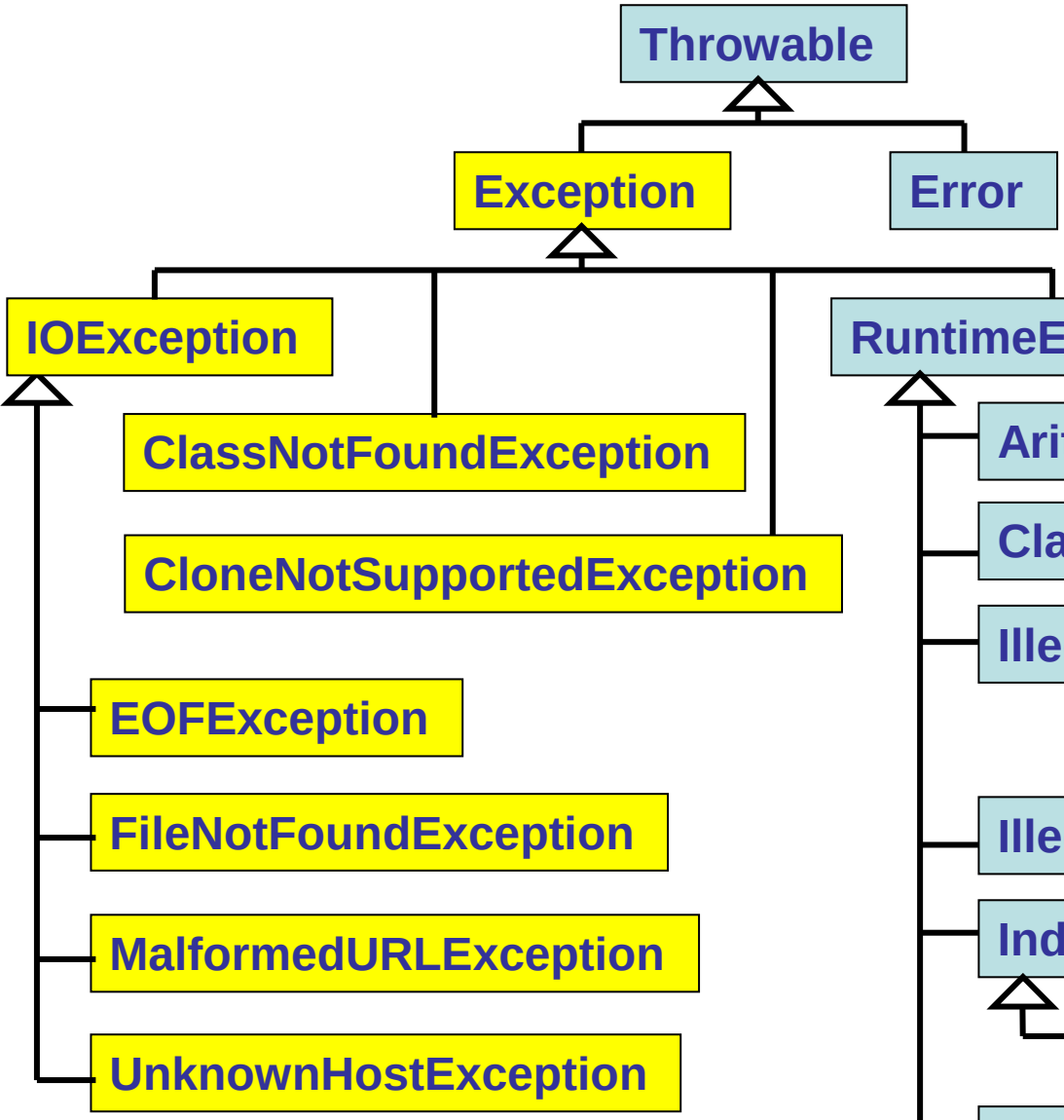
- 「年利率」に（範囲内の）実数にならないもの　または
- 「年数」に（範囲内の）整数にならないもの　または
- 「積立月額（円）」に（範囲内の）整数にならないもの

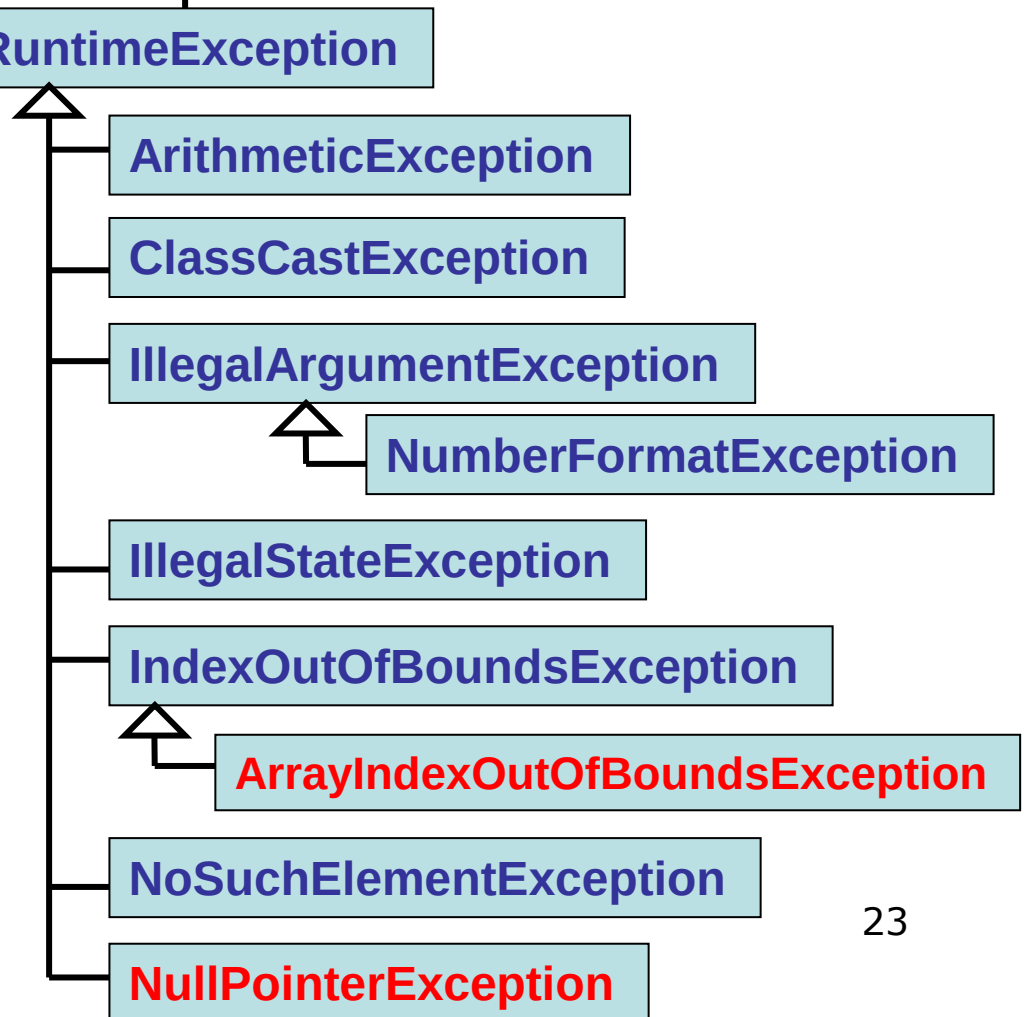
が入力されているとき，
「計算」ボタンをクリックすると，
プログラムはエラーメッセージを出す．



その理由：MyListenerクラスで
メソッドDouble.parseDouble と
Integer.parseInt を呼び出して
いる．その2つは入力が異常の
とき，NumberFormatException
という例外を出す．その例外をキャッチする必要がある．

Java言語の例外を表すクラス





Java言語の例外とは，
プログラムの誤りを通知し，
適切なエラー処理を行う
ための機構のことである．

黄色で塗った例外はコンパイラ
によってチェックされる例外．

例外のキャッチ：try-catch-finallyを使う

```
try {
    例外が発生しそうな処理
} catch (例外1クラス 変数) {
    例外処理1
} catch (例外2クラス 変数) {
    例外処理2
 …
} finally {
    共通処理
}
```

チェックされる例外が発生しそうな部分を上記のようにキャッチするか，または，その部分を含むメソッドでその例外を投げるように宣言しなければならない．

●例外が発生しなかった場合：
tryブロック内部が実行された後は，共通処理が実行される．

●例外1が発生した場合：
tryブロック内部で例外が発生した後は，対応する例外処理1が実行され，最後に共通処理が実行される．

●例外2が発生した場合：
tryブロック内部で例外が発生した後は，対応する例外処理2が実行され，最後に共通処理が実行される．

…

●例外1，例外2，…，以外の例外が発生した場合：
tryブロック内部で例外が発生した後は，共通処理が実行される．

try-catch-finallyのよい使い方

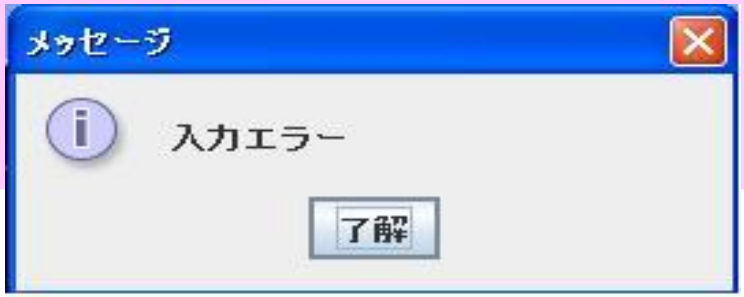
ここで発生した例外は

ここでキャッチされる.

```
try {
    try {
        例外が発生しそうな処理
    } finally {
        （例外が発生したか否か関係なく）必要な後片付け
    }
} catch (例外1クラス 変数) {
    例外処理1
} catch (例外2クラス 変数) {
    例外処理2
 ...
} catch (例外nクラス 変数) {
    例外処理n
}
```

addCalcButtonListenerメソッドの見直し

```
private void addCalcButtonListener ( ) {
    …
    public void actionPerformed(ActionEvent e) {
        try {
            … // actionPerformedメソッドの元々の中身をここに書く
        } catch ( NumberFormatException ex ) {
            // ここに例外exが発生したときの処理を書く
            JOptionPane.showMessageDialog(null , "入力エラー") ;
        }
    }
} //CalcButtonListenerクラスの終わり
```



結果のプログラムは SavingCal2.java

JOptionPaneはユーザに値の入力を求めたり情報を提示したりする標準のダイアログボックスを簡単に表示するために使われる.

Template Methodデザインパターン

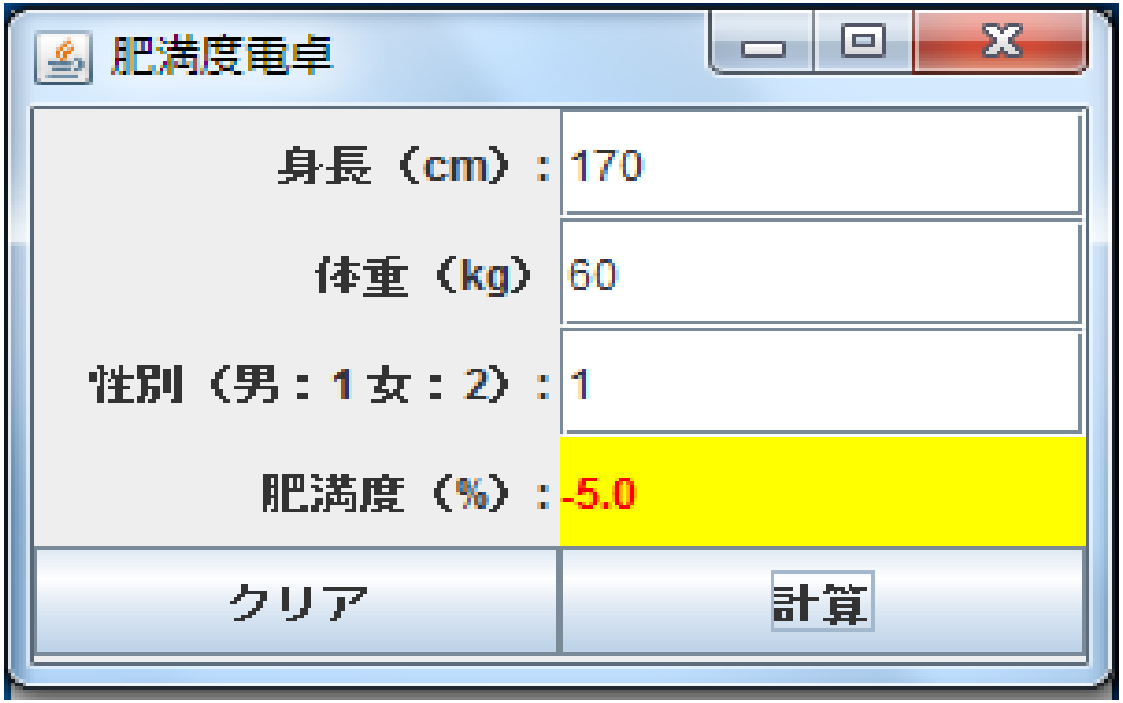
ポイント:
親クラスで処理の枠組みを定め，子クラスでその具体的内容を定める.

AbstractClass役（抽象クラス役）：テンプレートメソッド（電卓の例では，startメソッド）を実装する役．また，そのテンプレートメソッドで使っている抽象メソッドを宣言する．この抽象メソッドは，子クラスである**ConcreteClass役**によって実装される.
電卓の例では，**AbstractGUI**クラスがこの役をつとめている.

ConcreteClass役（具象クラス役）：**AbstractClass役**で定義された抽象メソッドを具体的に実装する役．ここで実装したメソッドは，**AbstractClass役**のテンプレートメソッドから呼び出される.
貯金電卓の例では，**SavingCal**クラスや**SavingCal2**クラスがこの役この役をつとめている.

演習課題

AbstractGUIクラスを拡張して，下記の肥満度電卓を作れ：



標準体重の計算法：
男の場合　（身長－80）＊0.7
女の場合　（身長－70）＊0.6

肥満度の計算法：
（体重－標準体重）／体重