第10章　補足

－Java プログラムを一から記述してみよう－

【学習内容とねらい】

これまで本テキストで扱ってきたのは主に Windows アプリケーション（Windows 上のボタンクリック等による動作するプログラム）でした（第 9 章ではアプレットの作成方法を学習しました)。ですから、皆は JBuilder を用いて Windows アプリケーションを作成する方法には習熟したはずです。

皆も経験した通り、JBuilder では、フレームの設計やイベント処理に関する多くの定型処理を自動的に記述してくれるので非常に便利です。しかしその反面、自動生成された Frame1.java のソースを眺めてみても、意味がよく分からない部分が多々あり不安あるいは不満を感じた人もいたことと思います。確かにそれら詳細が分からなくてもプログラムを作成できるところが JBuilder のメリットなのですが、より本格的に Java プログラミングを学習したい人にとっては、それら、言わばこれまで JBuilder に任せて来た部分を自分で理解しておく必要があります。そのためには、JBuilder のコード生成機能を用いずに一から自分（だけ）でプログラムを作成してみるのが一番です。

そこで、本章では、簡単な Windows アプリケーションを自分で最初から記述してみることにしましょう。本章の説明に沿ってプログラムを作成して行くと、これまで作成してきた Java アプリケーションプログラムの全体像が見えてくる筈です。そうすることで、JBuilder の便利さも理解できる様になり、したがってそれをより使いこなせるようになると思います。

＜第10章の構成＞

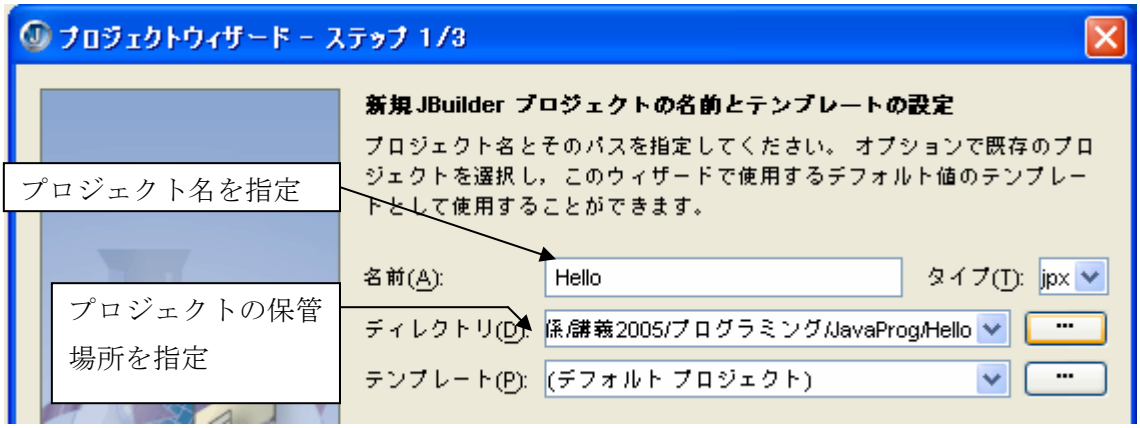
10-1 フレームのないプログラムの作成

まず、手始めにフレームのないプログラムを考えましょう。例えば 1-4 節で学習した様な、コンソール画面（コマンドプロンプト画面）に結果を表示するようなプログラムです。1-4 節では適当なエディタを用いてプログラムを記述し、その後でコンソール画面からコマンドを入力してプログラムを実行させました。これが JBuilder などの統合開発環境を用いない場合の、Java プログラムの作成・実行のやり方です。では、JBuilder を用いてもこのような（フレームのない）プログラムを作成・実行することはできるでしょうか？もちろん可能です。Java プログラムを一から記述する学習としてちょうど良い肩慣らしになるので、以下にその作成方法を学習しましょう。作成するのは画面に「Hello Java!」と表示するプログラムです。

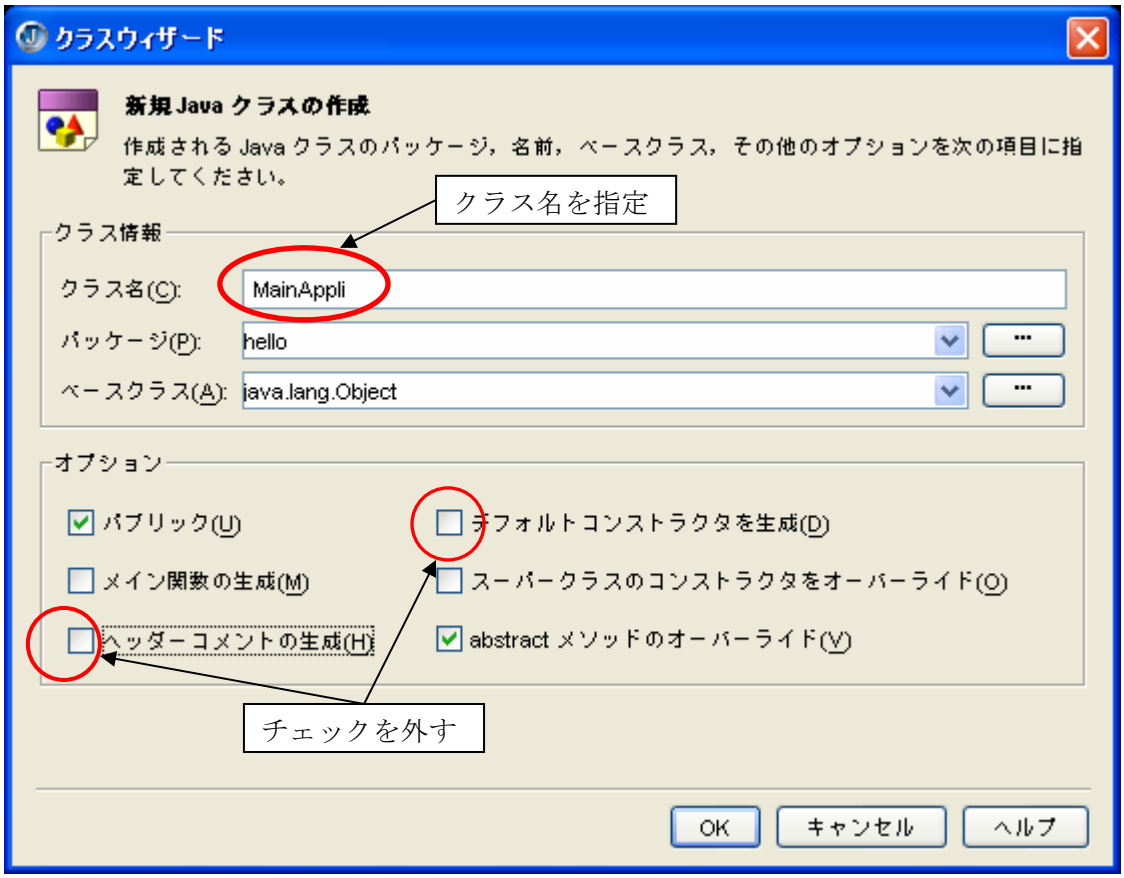
【例題 10-1-1 「Hello Java!」プログラム】

JBuilder を起動し、［ファイル］→［新規クラス］を選択して下さい。

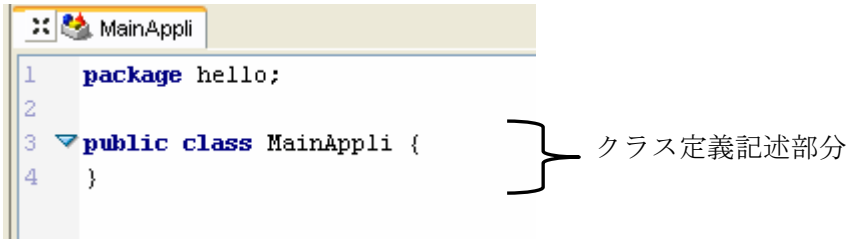
すると、アプリケーション作成時と同様に「プロジェクトウィザード」が現れます。ここでプロジェクト名を「Hello」と指定して下さい（名前は何でも良いのですが以下では「Hello」と指定したものとして説明します）。



［次へ（N)］ボタンをクリックして次へ進み、次のようにクラス名を「MainAppli」として下さい（やはり名前は何でも良いのですが以下では「MainAppli」と指定したものとして説明します)。また、「デフォルトコンストラクタを生成」欄と「ヘッダーコメントの生成」欄のチェックを外しておいて下さい。



［OK］ボタンをクリックすると次のエディタ画面に移ります。

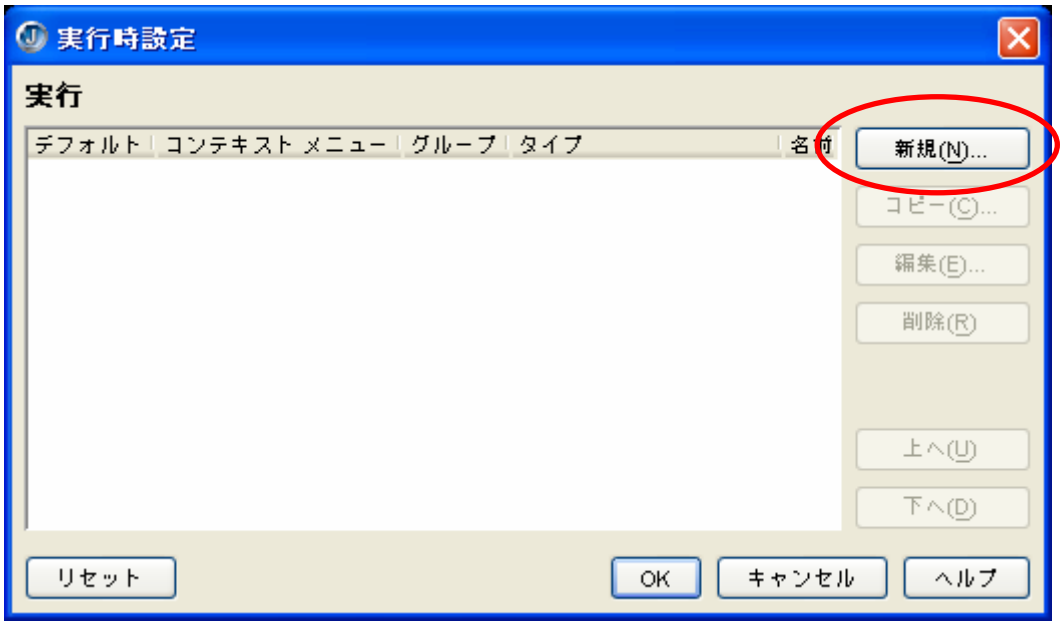


ここで、「クラス定義記述部分」を次のように記述して下さい。

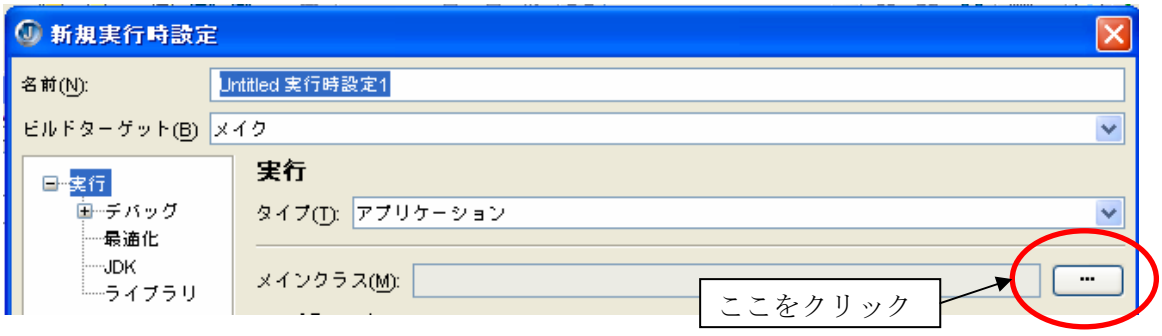
```
public class MainAppli {
  public static void main(String[] args) {
    System.out.println("Hello Java!");
  }
}
```

これは、1-4 節（p.27）で記述したものと（クラス名を除いて）同じです。

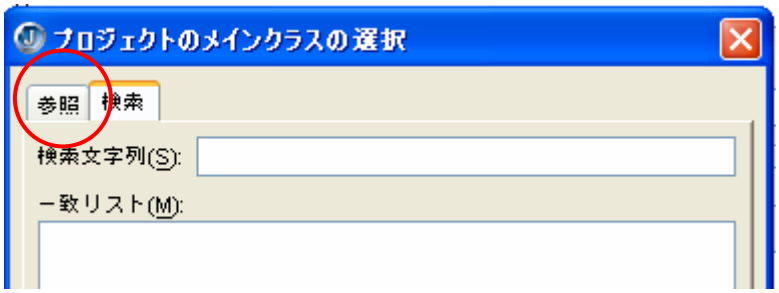
ここで通常通りプログラムを実行すると、次のような「実行時設定」画面が現れます。



これは、JBuilderが、メインクラス（mainメソッドを含むクラス）がどこにあるか分からないため、プログラムを実行できないことを意味します。これまでの様に「新規アプリケーション」を選択してプログラムを作成した場合は、メインクラスが『Application1.javaファイル内にある「Application1」というクラスである』ことが自動的に指定されていたので、プログラマが個別に指定する必要はなかったのです。ところが一般には、実行対象となるメインクラスがどこにあるかは、プログラマが指定しなければなりません。これを、**クラスパス**を指定すると言います。クラスパスを指定するためには、上の画面の［新規］ボタンをクリックします。そうすると、次の設定画面が現れます。

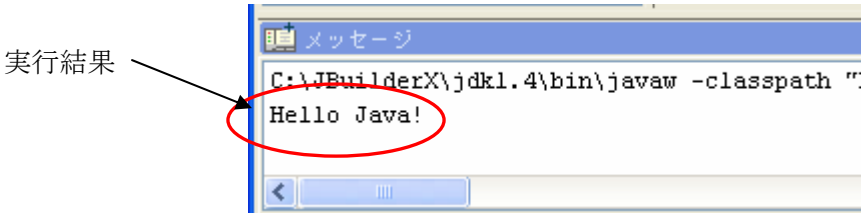


ここで、メインクラスのクラスパスを指定できます。メインクラス指定欄横の［･･･］ボタンをクリックして下さい。すると、次ページの画面が現れます。



ここで、参照タグをクリックして次のようなクラスリストを表示させて下さい。リストの中に今作成した「hello」パッケージ（プロジェクト）があるので、今対象となっているメインクラス「MainAppli」を選択して［OK］ボタンをクリックして下さい。

これでクラスパスを指定することができました。再び実行するとアプリケーションブラウザ下方のメッセージ欄に次のように結果が表示されるはずです。



　このように、メッセージ欄はコンソール画面（コマンドプロンプト画面）と同様の働きをします。

　以上の様に、クラスパスさえきちんと指定すれば、(Windows アプリケーションに限らず) どのような Java プログラムでも JBuilder を用いて作成・実行することができます。

【例題 10-1-2　データの入力】

さて、もう一度 main メソッドを眺めてみましょう。

```
public static void main(String[] args) {
  System.out.println("Hello Java!");
}
```

これを見ると、main メソッドは引数(String[] args)を持っています。これは何のために必要なのでしょうか？恐らく疑問に思った人も多いでしょう。実はこれはコンソール画面からデータを入力する際に必要となる変数です。

まず、引数「args」は文字列型の配列であることが分かるでしょう（配列は 4-12 節で学習しました)。ですからargsはargs[0]、args[1]、･･･の様な形で複数の値を持ち得ます。この引数の意味（役割）を理解するには、実際に入力データを使用した実例を確認するのが早道です。そこで、以下のその例を学習しましょう。

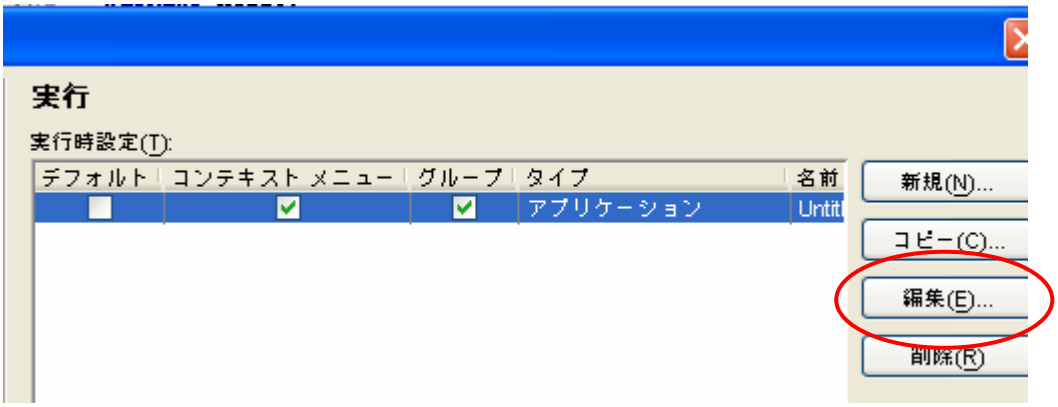
まず、main メソッドに以下の枠線部を追加して下さい。

```
public class MainAppli {
  public static void main(String[] args) {
    System.out.println("Hello Java!");
    int a=Integer.parseInt(args[0]);
    int b=Integer.parseInt(args[1]);
    System.out.println(a+b);
  }
}
```

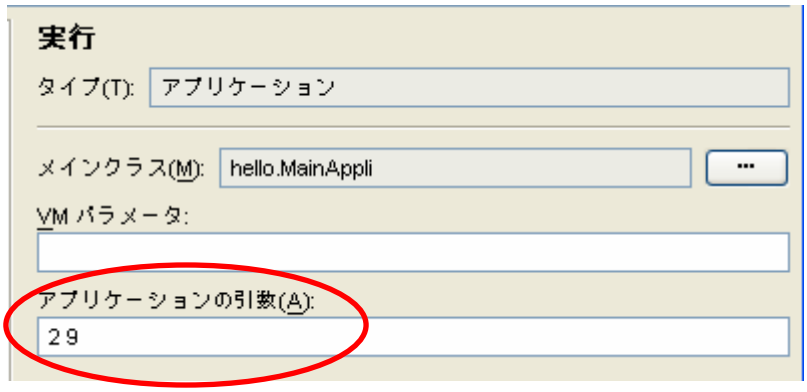
枠線部は、『0 番目と 1 番目の引数を整数 a,b として受け取りその合計を画面に表示する』という処理になっています。

さて、プログラム実行時に入力データを指定するためには、通常はコンソール画面から直接入力しますが、JBuilder では、先ほど開いた「実行時設定画面」で次のように指定します。

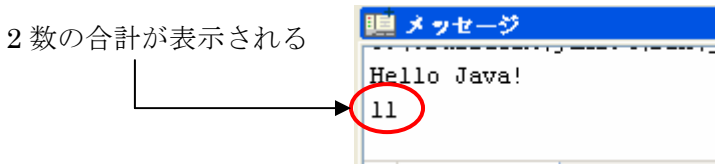
アプリケーションブラウザのメニューから［プロジェクト］→［プロジェクトプロパティ］を選択して下さい。すると、次のように、先ほどと同じ「実行時設定」画面が現れます。プログラムの実行に関わる設定はここで行うようになっています。ただし、すでに先ほどメインクラスのクラスパス指定という、設定を行ったので、今度はその情報に入力データの指定という情報を追加することになります。したがって、今度は［新規］ではなく、次のように［編集］ボタンをクリックして下さい。



すると、下の画面が現れるので、「アプリケーションの引数」欄に適当な２つの整数（下の例では２と９）を空白で区切って入力して下さい。なお、今の場合、整数入力なので入力は全て半角で行ってください。



これで、[OK] ボタンをクリックして設定を終了します。その後、プログラムを実行させると下のように、2 数の合計が結果として表示されます。

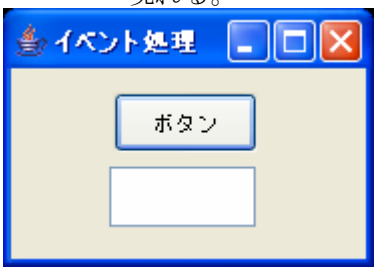


これで、main メソッドの引数 args の意味（役割）が分かったでしょう。

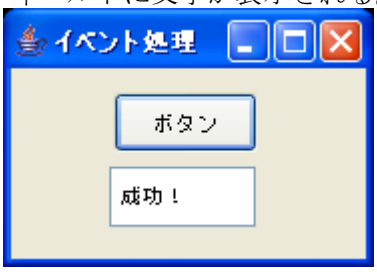
10-2 フレームの生成・表示

本節以降では、次のようなプログラムを作成する事にします。

プログラムを起動すると次の画面が現れる。



ボタンをクリックすると、テキストフィールドに文字が表示される。



本節ではまず、フレームを生成してそれを表示させる（だけの）プログラムを作成しましょう。もちろん、JBuilder のフレーム設計機能を用いずに自力で作成します。

【例題 10-2-1 フレームの生成・表示】

まず、前節で作成したプログラムは全て閉じてください。そして、前節同様、［ファイル］→［新規クラス］を選択して、新規クラスをプロジェクト名「WindowsAppli」、クラス名「FrameAppli」として作成して下さい。

すると、次のエディタ画面が現れるはずです。

```
package windowsappli;

public class FrameAppli {
}
```

ここで、以下のようにプログラムを記述して下さい。

```
package windowsappli;

import javax.swing.*; //フレームを用いるために必要      ①

public class FrameAppli {
  public static void main(String[] args) {
    JFrame frame=new JFrame(); //フレームの生成        ②
    frame.setTitle("イベント処理"); //フレームタイトルの設定      ③
    frame.setBounds(100,100,200,150); //フレームサイズの設定   ④
    frame.setVisible(true); //フレームの視覚化      ⑤
  }
}
```

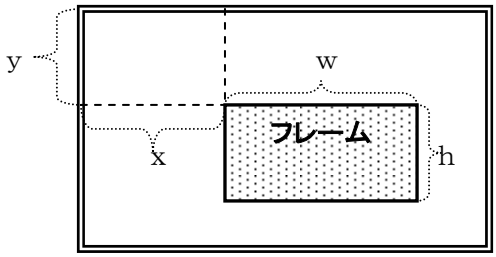
＜プログラムの解説＞

① フレームクラスは swing というパッケージに入っています。この中には、ボタンやテキストフィールドなどの各コンポーネントも含まれています。ですから、それらコンポーネントを用いるときには、冒頭でこのように swing パッケージを指定します。

② フレームは JFrame クラスというクラスで定義されています。ここでは、JFrame クラスのオブジェクトを frame という名前で生成しています。オブジェクトの生成について不明な点があれば第 7 章を読み返して下さい。

③ JFrame クラスには、`setTitle()`メソッドが定義されており、フレームのタイトルを引数として指定します。

④ `setBounds(x,y,w,h)`メソッドは、画面上の座標(x,y)の位置に、幅 w、高さ h の大きさでフレームを配置するメソッドです。

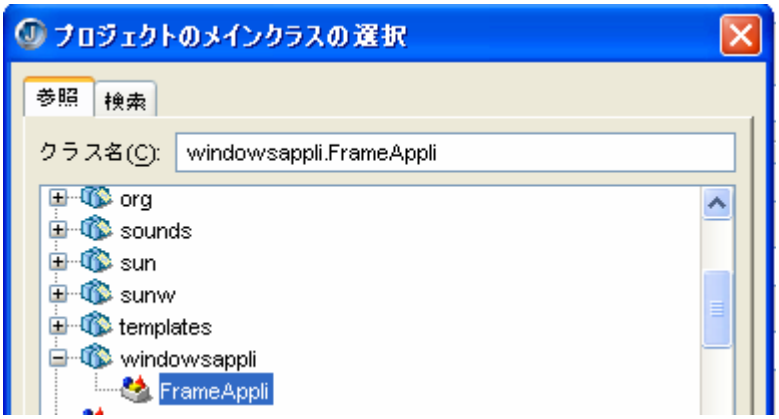


＜画面＞

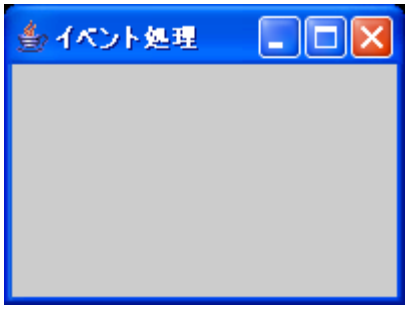
⑤ `setVisible(true)`メソッドにより、フレームを画面に表示できます。引数のデフォルト（既定値）は false に設定されているので、この文がなければプログラムを実行してもフレームは表示されません。少し不自然に思うかもしれませんが、複数のフレームを扱う場合などを想定するとフレームを表示させるタイミングはプログラマが決めた方が便利である、と Java 言語開発グループは考えた様です。

これらは JBuider が自動生成していた部分なので、初めて目にする部分もあると思いますが、いずれも意味は極めて単純なので理解に問題はないでしょう。

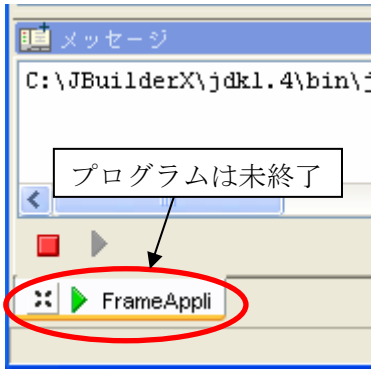
さて、作成したらプログラムを実行しましょう。そのためには、前節で学習した様にメインクラス（FrameAppli）のパスを指定しておかねばなりません。実行ボタンをクリックするか、あるいは［プロジェクト］→［プロジェクトプロパティ］を選択するかして、「実行時設定」画面を呼び出し、右のようにメインクラスのパスを指定して下さい。やり方が不明な場合は p.234 を参照して下さい。



クラスパスの指定後プログラムを実行すると、次のようなフレームが現れるはずです。



これで、実行時にウィンドウ（フレーム）を表示させるプログラムが出来ました。ひとまず、このフレームを閉じましょう。



ところが、JBuilder のアプリケーションブラウザの左下画面をみると、次のようにまだプログラムが終了していないことが分かります。実は、このままでは、フレームの右上隅の×をクリックしても、非表示になるだけでプログラムは終了しないのです。

×をクリックしたときにプログラムを終了させるためには、「×をクリックしてフレームを閉じる」というイベントが発生したときに、プログラムも終了する様にプログラムを記述する必要があります。次の例題プログラムに進んで下さい。

【例題 10-2-2　プログラムの終了処理（イベント処理）の追加】

上の例題プログラムに次の波線部と枠線部を追加して下さい。

```
package windowsappli;

import javax.swing.*; //フレームを用いるために必要
import java.awt.event.*; //イベント処理記述のために必要      ①

public class FrameAppli {
  public static void main(String[] args) {
    JFrame frame=new JFrame(); //フレームの生成
    frame.setTitle("イベント処理"); //フレームタイトルの設定
    frame.setBounds(100,100,200,150); //フレームサイズの設定
    frame.setVisible(true); //フレームの視覚化
    frame.addWindowListener(new Wclose()); //イベントリスナの登録
  }                                                    ②
}

class Wclose extends WindowAdapter{                    ③
  public void windowClosing(WindowEvent e) {           ④
    System.exit(0); //プログラムの終了
  }
}
```

＜プログラムの解説＞

① ③で必要になる WindowAdapter クラスは、java.awt.event というパッケージに含まれています。そのために、このパッケージを指定しておく必要があります。

② 今必要なのは、『Windows を閉じる』というイベントが発生した時にプログラムを終了するように記述する事です。addWindowListener("**イベントリスナ**")メソッドは、フレーム（ウィンドウ）に**イベントリスナ**を登録します。イベントリスナについては下のコラムを参照して下さい。今の場合、イベントリスナは WClose というクラスのオブジェクトです。

③ その WClose クラスは WindowAdapter クラスを継承して定義しています。**継承**については 7-4 節を参照して下さい。この WindowAdapter クラスには、フレーム（ウィンドウ）に関する様々なイベント（開く、閉じる、アクティブにする等々）発生時に呼び出されるメソッドが定義されています。

④ そのメソッドの一つが windowClosing というメソッドです。これは、フレーム（ウィンドウ）が閉じられるときに呼び出されるメソッドです。最初は何の処理も記述されていません。つまりメソッド名だけが指定された空っぽのメソッドです。ですからユーザが必要に応じて処理を書き込むようにします。ここでは、「System.exit(0)」という処理を記述しました。これは、プログラムを終了させるメソッドです。これで、フレーム（ウィンドウ）が閉じられた時、プログラムが終了する様になります。

コラム　イベント処理について

Java 言語開発グループは、イベント処理が、**イベントソース**、**イベント**そして**イベントリスナ**の 3 者から構成されるものと捉えました。上の例の場合、イベントソース、すなわちイベントの発生元はフレーム（ウィンドウ）です。そしてイベントは、「フレームを閉じる」という事象で、イベントリスナは WClose クラスのオブジェクトです。

Java言語では、イベント処理を記述する場合、イベントソースにイベントリスナを登録します。イベントリスナは“聞き役”という意味ですが、言わばイベントの監視役です。上の例で言うと、addWindowListener("**イベントリスナ**")という形で（フレームに）イベントリスナを登録しています。これにより、フレームにイベント発生を感知する機能が備わり、様々なイベントが発生した場合、イベントリスナは、当該イベントの種類に応じて対応するメソッドを起動します。例えば、フレームを閉じるというイベントが発生した場合には、リスナはwindowClosingというメソッドを起動し、フレームを開いたときにはwindowOpened()というメソッドを起動する、等々です。

プログラムを作成したら実行し、フレームを閉じてみて下さい。今度はプログラムも終了する筈です。これを確認して下さい。

10-3 コンポーネントの貼り付け

本節では、前節で作成したフレームにボタンとテキストフィールドを貼り付けましょう。【例題 10-2-2】で作成したプログラムを開いておいて下さい。

【例題 10-3-1 コンポーネントの貼り付け】

【例題 10-2-2】のプログラムに以下の波線部および枠線部を追加して下さい。

```
import javax.swing.*;  //フレームを用いるために必要
import java.awt.event.*; //イベント処理記述のために必要
import java.awt.*;  //Container クラスを用いるために必要     ①

public class FrameAppli {
  public static void main(String[] args) {
    JFrame frame=new JFrame(); //フレームの生成
    frame.setTitle("イベント処理"); //フレームタイトルの設定
    frame.setBounds(100,100,200,150); //フレームサイズの設定
    //コンポーネントの生成          ②
    JButton jBtn=new JButton(); //ボタンの生成
    jBtn.setBounds(50,20,100,30); //ボタンサイズの設定
    jBtn.setText("ボタン"); //ボタンの表示テキストの設定
    JTextField jTxt=new JTextField(); //テキストフィールドの生成
    jTxt.setBounds(50,70,100,30); //テキストサイズの設定
    //コンポーネントを貼り付ける
    Container ContPane=frame.getContentPane();    ③
    ContPane.setLayout(null); //null レイアウトの指定  ④
    ContPane.add(jBtn); //ボタンの貼り付け
    ContPane.add(jTxt); //テキストフィールドの貼り付け   ⑤
    frame.setVisible(true); //フレームの視覚化
    frame.addWindowListener(new Wclose());  //イベントリスナの登録
  }
}
```

＜プログラムの解説＞

① コンポーネントは直接フレームの上に貼り付けられません。コンポーネントを貼り付けるためには、貼り付けが可能な機能（**コンテナ**機能）を持ったクラスのオブジェクトである必要があります。代表的なものがそのものずばりの Container（コンテナ）クラスです。この Container クラスは java.awt パッケージに含まれているのでここでそれ

を指定しています。なお、JPanel クラスもコンテナ機能を持っています。

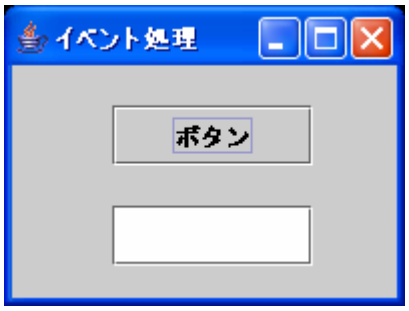
② ボタンコンポーネントおよびテキストフィールドコンポーネントのクラスはそれぞれ JButton および JTextField です。それが分かればこの「コンポーネントの生成」部分は理解できるでしょう。なお、`setBounds(x,y,w,h)`メソッドの意味は【例題 10-2-1】（p.239）で説明した通りですが、今の場合、これらコンポーネントはフレームの上に配置されることになるので、配置位置（`x,y`）は、フレームの左上隅を原点（`0,0`）としたときの座標となります。

③ 右辺の `frame.getContentPane()`により、フレームコンポーネント `frame` に付随した**コンテナ**を取得することができます。そしてそれを①で説明した Container クラスのオブジェクト `ContPane` に代入することで、以後、`ContPane` を `frame` のコンテナとして用いる事が出来ます。`ContPane` は（`Container` クラスのオブジェクトなので）その上にコンポーネントを貼り付ける事ができます（⑤参照）。そうして、`ContPane` にコンポーネントを貼り付ければ、結果的にフレームの上に貼り付けられることになります。この `ContPane` は、今まで目にしてきた（フレームの）`contentPane` に他なりません。2-1 節のコラム（p.36）を読み返してみて下さい。前よりは少し意味が分かると思います。

④ ここで、レイアウトを null に指定しています。これまでは、設計モードのプロパティで指定していましたが、その際、JBuilder がこのような文を生成してくれていたのです。

⑤ コンテナ ContPane にコンポーネントを貼り付けるためには、`add("コンポーネント")`メソッドを用います。

プログラムを作成して実行すると、次の画面が現れます。ただ、まだボタンクリック時のイベント処理は記述していません。それは次節で行います。



なお、フレームやボタンなどのコンポーネントの形状がこれまでと異なりますが、これは、コンポーネントの外観の指定が JBuilder による（デフォルトの）指定と異なるためです。JBuilder で新規アプリケーションとして作成すると Windows タイプの外観になるのですが、今はその指定をしていないので、Metal タイプという外観になっています。ただ、本質的な問題ではないので、以降はこのまま続けます。

10-4 イベント処理の追加

それでは、最後に、［ボタン］をクリックした時に、テキストフィールドに文字が表示されるようにイベント処理を追加しましょう。前節で作成したプログラムを開いておいて下さい。

【例題 10-4-1 ボタンイベントの追加】

【例題 10-3-1】のプログラムに以下の波線部と枠線部を追加して下さい。

```
public class FrameAppli {
  public static void main(String[] args) {
    JFrame frame=new JFrame(); //フレームの生成
            ・・・(変更なし)
    frame.setVisible(true); //フレームの視覚化
    //イベントハンドラの追加
    frame.addWindowListener(new Wclose()); //イベントリスナの登録
    jBtn.addActionListener(new bAction(jTxt));     ①
                          //ボタンクリックに関するイベントリスナの登録
 }
}

class Wclose extends WindowAdapter{
          ・・・(変更なし)
}
```

```
class  bAction implements ActionListener{    ②
  JTextField jTxtFld; //テキストフィールドをフィールド変数として宣言  ⑤
  public bAction(JTextField jTxt) {  //コンストラクタ     ④
    jTxtFld=jTxt; //引数の jTxt をフィールド変数 jTxtFld に代入
  }
  public void actionPerformed(ActionEvent e) {   ③
     jTxtFld.setText("成功！");
  }
}
```

＜プログラムの解説＞

① 【例題 10-2-2】でやった様に、**イベントリスナ**を登録しています。ただし、今度はボタンイベントに関するイベントリスナです。ボタンイベントはクリックしかないので、ボタンクリックに関するイベントリスナと言っても良いです。そのイベントリスナは `bAction` というクラスのオブジェクトで、コンストラクタの引数としてテキストフィールドを受け取ります。

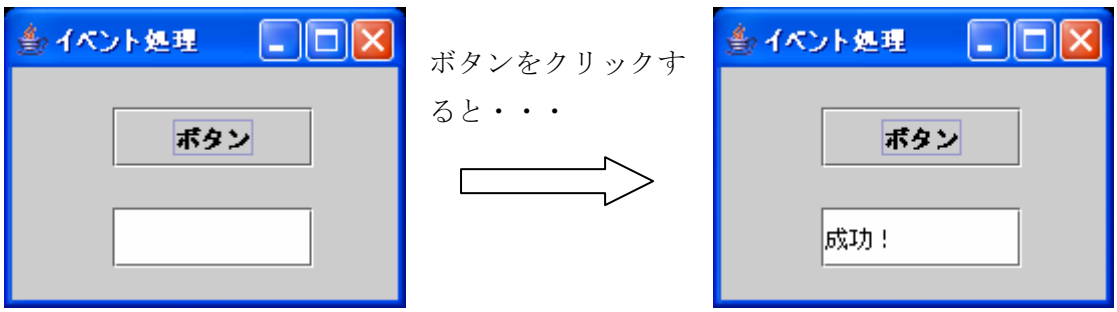
② クラス `bAction` は `ActionListener` という**インターフェース**を**実装(継承)**して定義されます。インターフェースについては次ページのコラムを参照して下さい。ここでは、【例題 10-2-2】で説明した WindowAdapter と同じように、『ボタンクリック・イベント発生時に呼び出されるメソッドがその名前だけ定義されているクラス』だと捉えて下さい。そして **`implements`** は**継承(`extends`)**と意味は同じです。

③ ボタンクリック時に呼び出されるメソッドは `actionPerformed ()`というメソッドです。そこで、ここに処理内容、つまり『(フレーム上の) テキストフィールドに「成功！」という文字列を表示させる』と言う処理を記述します。

④ ③の処理を実行するためには、メインクラスから当該テキストフィールド `jTxt` を受け取らなければなりません。そのために、コンストラクタの引数に（処理対象となる）テキストフィールドを受け取れるようにしているのです。そして引数として受け取ったテキストフィールド変数 `jTxt` を、③の処理で使用できるようにフィールド変数 `jTxtFld` に代入しています。引数は当該メソッド内でのみ有効な**ローカル変数**である点に注意して下さい。一方、フィールド変数はクラス内でアクセス可能なので③のメソッド `actionPerformed ()`内でも処理できるのです。

⑤ そのために、`JTextField` クラスの変数をフィールド変数として宣言しておく必要があります。

プログラムを作成したら実行して動作を確認して下さい。



コラム　インターフェースとは

インターフェースとは、名前のみが決まっていて処理内容を定義していないメソッド（群）からなるクラスの事です。処理内容は場合に応じて異なってもメソッドの名前は共有したい、つまりメソッド定義を標準化したい、という考えから導入されました。

ただし、インターフェースに含まれるメソッド群については、それを継承したクラスにおいて、（何もしないと言う処理を含めて）何らかの処理を定義する必要があります（定義しないメソッドがあるとエラーになります）。ActionListener インターフェースの場合は、actionPerformed ()メソッド一つのみなので、それを定義すれば済むのですが、10-2 節で学習した WindowEvent の場合は、開く、閉じる等、複数のイベントに対応したメソッドがあるので、例えば「閉じる」場合の処理だけを行いたい場合でも、残りのメソッドを全て記述しなければならず面倒です。

そこで、必要なメソッドのみを記述（定義）すれば済むリスナとして**アダプタ**というクラスが用意されました。それが、【例題 10-2-2】で出てきた WindowAdapter です。WindowEvent に対応するリスナ（インターフェース）としては WindowListener があります。

なお、**implements** は上の＜プログラムの解説＞でも述べた通り、継承（extends）と意味は同じです。継承元のクラス（スーパークラス）がインターフェースの場合のみ implements となります。

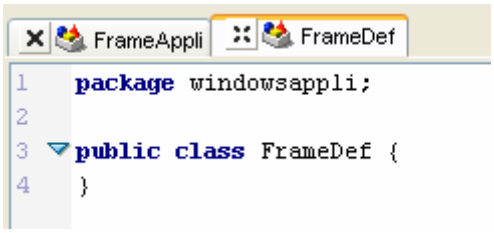
10-5 プログラムの整理

前節まででプログラムは完成しました。しかし、イベントリスナ・クラス以外は全て main メソッド内に記述したため、見通しが悪くなっています。また、フレームの設計を変更するたびに main メソッド全体をいじらなければならず、拡張する際にも不便です。そこで、このプログラムを見通しが良く拡張しやすい形に書き直しましょう。前節で作成したプログラムを開いておいて下さい。

【例題 10-5-1 フレームクラスの独立】

まず、フレームの設計やその上のコンポーネントに関するイベント処理については、一つのクラスとしてまとめた方がすっきりすると思われます。そこで、フレームに関する機能は全て一つのクラスにまとめてしまいましょう。

【例題 10-4-1】のプログラムを開いた状態で、「ファイル」メニューから「新規クラス」を選択して、新しいクラスを「FrameDef」という名前で作成します。Def は Definition（定義）の略の意味で用いました。すると、次のように FrameDef クラスの編集画面に移ります。



この、FrameDef クラスに前節で作成したプログラムのフレーム定義に関する部分を移設しましょう。どのように移動すれば良いかを、まず各自で考えてみてください。大まかな構造は次のようになります。

```
package windowsappli;
public class FrameDef extends JFrame{  //JFrame クラスを継承する
  //使用するコンポーネントを宣言・生成する部分
          ・・・   ・・・
  // フレームやコンポーネントの形状設定や配置およびイベントリスナの登録を行う
  public FrameDef() {   //上の処理をこのコンストラクタ内で行う
        ・・・   ・・・
  }
}

// アダプタ（WClose）とイベントリスナ（bAction）は以下にそのまま持ってくる。
      ・・・   ・・・
```

どうですか、自分で移設できたでしょうか？少し戸惑う部分もあったかもしれません。最終的には、FrameDef.java のプログラムは以下の様になります。

＜FrameDef.java ファイルの内容＞

```
package windowsappli;

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

public class FrameDef extends JFrame{  //JFrame クラスを継承する  ①
   //使用するコンポーネントを宣言・生成する部分
  JButton jBtn=new JButton(); //ボタンの生成
  JTextField jTxt=new JTextField(); //テキストフィールドの生成
  Container ContPane= this.getContentPane(); //コンテナの生成  ③
  // フレームやコンポーネントの形状設定や配置およびイベントリスナの登録を行う
  public FrameDef() {    //上の処理をこのコンストラクタ内で行う
    this.setTitle("イベント処理"); //フレームタイトルの設定
    this.setBounds(100,100,200,150); //フレームサイズの設定   ②
    jBtn.setBounds(50,20,100,30); //ボタンサイズの設定
    jBtn.setText("ボタン"); //ボタンの表示テキストの設定
    jTxt.setBounds(50,70,100,30); //テキストフィールドサイズの設定
    //コンポーネントを貼り付ける
    ContPane.setLayout(null); //null レイアウトの指定
    ContPane.add(jBtn); //ボタンの貼り付け
    ContPane.add(jTxt); //テキストフィールドの貼り付け
    //イベントハンドラの追加
    this.addWindowListener(new Wclose());  //イベントリスナの登録  ④
    jBtn.addActionListener(new bAction(jTxt));
                          //ボタンクリックに関するイベントリスナの登録
  }
}

class Wclose extends WindowAdapter{
  ・・・(前節のプログラムのまま)
}

class  bAction implements ActionListener{
  ・・・(前節のプログラムのまま)
}
```

＜プログラムの解説＞

① 今は（独自の）フレームクラスを定義しようとしているので、JFrame クラスを継承して作成します。

② この部分は前節では「`frame.setTitle("イベント処理")`」等のように記述していました。ところが今の場合、frame の様なフレームクラスのオブジェクトは生成できません。なぜなら、その元になるクラスをここで定義しようとしているからです。このような場合、**this** 変数を用います。9-3 節（p.229）でも説明した通り、this 変数は、今定義しているクラス（今の場合 FrameDef）から生成されるオブジェクトの総称です。③および④の this 変数も同様です。この部分が本プログラムのポイントです。

※ 上以外の部分は、単に前節のプログラムをコピーしただけなので問題ないでしょう。

さて、このように FrameDef クラスを定義すると、FrameAppli.java の方は次のように極めて簡単になります。

＜FrameAppli.java ファイルの内容＞

```
package windowsappli;

public class FrameAppli {
  public static void main(String[] args) {
    FrameDef frame=new FrameDef(); //FrameDef オブジェクトの生成
    frame.setVisible(true);  //生成したフレームの視覚化
  }
}
```

こうすることにより、フレームの設計あるいはイベント処理の内容を変更する場合でも、FrameAppli クラスは変更する必要はありません。ただ、FrameDef クラスを変更するだけで OK です。あるいは、別のフレームクラスに差し替えても結構です。このように、

◆ フレームの設計 → FrameDef クラス

◆ プログラムの実行（フレームの生成と表示） → FrameAppli クラス

というように、ひとまとまりの機能をクラスとしてまとめると、プログラムの拡張性あるいは再利用性が高まります。これがオブジェクト指向プログラミングの大きなメリットなのです。

プログラムの書き直しが終了したら実行して、きちんと動作することを確認して下さい。確認が済んだら、本章の仕上げとして次の例題に進んで下さい。

【例題 10-5-2　イベントリスナの改良】

上の例題で一通りの整理は出来たのですが、本章の最後に、ボタンをクリックした時のイベントリスナの記述をより拡張性のあるものに改良しましょう。

今の場合は、ボタンクリック時の処理は「テキストフィールドに文字を表示させる」という単純なものでした。しかし、ボタンクリック時に FrameDef クラスで定義した複数のコンポーネントを操作したり、あるいはより複雑な処理を必要として来たりすると、イベントリスナである bAction クラスの中に直接定義を記述すると見通しが悪くなって来ます。その事を考えると、やはりフレーム上の処理（の詳細）に関しては、FrameDef クラス内で定義した方が、まとまりが良いでしょう。つまり、上の例題で体験した『関連するひとまとまりの機能を当該クラスの中に集積させる』という方針を徹底させる訳です。そこで、FrameDef クラス内にボタンクリック時の処理内容を記述したメソッドを用意し、それを bAction クラスから呼び出すようにしましょう。

FrameDef クラスを次のように修正します。波線部を修正し、枠線部を追加して下さい。

＜FrameDef クラスの修正＞

```
public class FrameDef extends JFrame{  //JFrame クラスを継承する
  //使用するコンポーネントを宣言・生成する部分

      ・・・(そのまま)

  // フレームやコンポーネントの形状設定や配置およびイベントリスナの登録を行う
  public FrameDef() {  //上の処理をこのコンストラクタ内で行う

      ・・・(そのまま)

    //イベントハンドラの追加
    this.addWindowListener(new Wclose());//イベントリスナの登録
    jBtn.addActionListener(new bAction(this));   ①
                        //ボタンクリックに関するイベントリスナの登録
  }
  public void jBtn_Action(ActionEvent e) {
    jTxt.setText("成功！");
  }                                                          ②
}
```

＜プログラムの解説＞

① イベントリスナの引数を **this** 変数に変更します。なぜそうする必要があるかは、下のイベントリスナの修正を見れば分かります。

② ボタンクリック時の処理を「jBtn_Action」という名前のメソッドとして定義しています。メソッド名は何でも良いのですが、『どのコンポーネントの、どのようなイベントに対する処理か』が分かるような名前をつけることが推奨されています。

上に対応する様にイベントリスナを修正すると次のようになります。

＜イベントリスナ bAction の修正＞

```
class  bAction implements ActionListener{
  FrameDef frame;
  public bAction(FrameDef frm) {
    frame=frm;
  }
  public void actionPerformed(ActionEvent e) {
     frame.jBtn_Action(e); //メソッドを呼び出すだけ
  }
}
```

ポイントは枠線部です。ここで、FrameDef クラスで定義したメソッドを呼び出す様に修正しました。その際、イベントリスナは独立したクラスなので当該メソッドを呼び出すには、FrameDef クラスのオブジェクト名を先頭に付ける必要があります。そのためにイベントリスナ bAction は引数として this 変数、つまり FrameDef クラスのオブジェクトを受け取り、それをコンストラクタ内でフィールド変数 frame に代入しているのです。

さて、こうすることでイベントリスナはイベント発生時の処理内容が変わっても変更する必要がなくなりました。処理内容の変更はFrameDefクラス内のメソッドのみを変更すれば良いからです。これで、クラス間の独立性が高まり、あるいはクラス間の役割分担がより明確になり、したがってプログラムの拡張がより容易になった事が分かるでしょう。

さて、もう気付いたと思いますが、

- FrameDef.java → Frame1.java
- FrameAppli.java → Application1.java

と置き換えれば、これは、これまで JBuilder でアプリケーションを作成した場合のファイル構成に対応します。このように JBuilder は、Windows アプリケーションを効率的に作成できるようにこれら２つのファイルを用意し、プログラマが、フレームのデザインとイベント処理の記述に専念できるようにお膳立てをしてくれていたのです。JBuilder の便利さが理解できたでしょうか？