第6章　メソッドの定義

【学習内容とねらい】

これまで、コンポーネントの配置（デザイン）やイベントハンドラのプログラミングを行って来ました。そしてその過程で、**setText()**や**getText()**等、コンポーネントに定義されたメソッドを適宜利用してきました。本章では、そのメソッドを自分で定義してみましょう。と言っても、それ程大したことはありません。実は（これまで作成してきた）イベントハンドラもメソッドの一種なのです。

そこで本章では、そのなじみの深いイベントハンドラから出発して、それを、より一般的なメソッド、つまり特定のイベントに拘束されないメソッドに拡張する、という”流れ”で学習を進めています。ですから個々の内容は無理なく理解できるはずです。むしろ、題材を基本的なものに限定しているため退屈するかもしれません。しかし、本章の学習内容は、より本格的なプログラムを作成する際の必須事項です。今後、より実践的な Java 言語プログラミングへ進むためには、自在にメソッドを定義できる事が不可欠なのです。どうか（退屈でもバカにしないで）しっかりと学習して下さい。

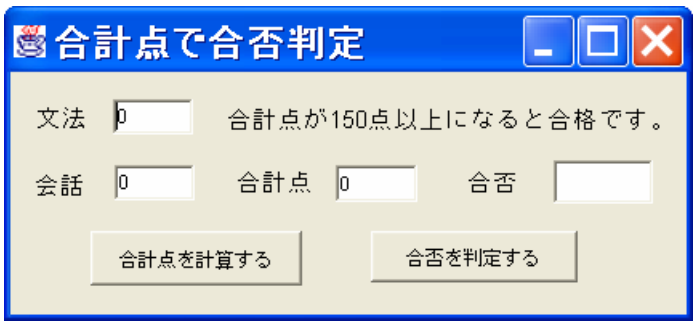
＜第6章の構成＞

6-1 メソッド（1）—2 つのボタンからなるプログラム—

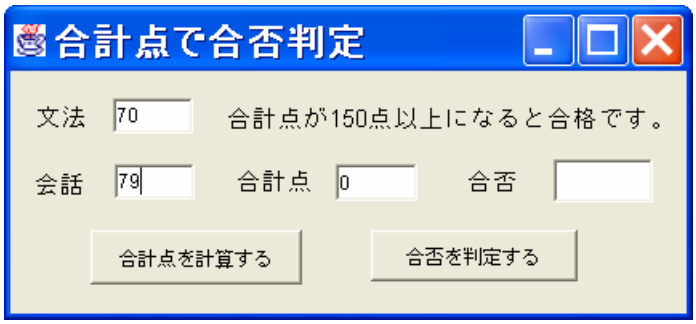
【基礎課題 6-1-1】

文法と会話の得点合計を求めてから合否を判定するプログラムを作りましょう。
ここに後の説明の都合上、二つの「ボタン」コンポーネントの name プロパティはそれぞれ「jButton1」および「jButton2」として下さい。

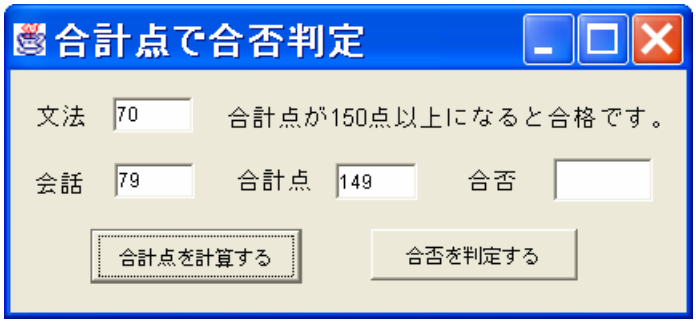
右のようなフレームを作って下さい。
動作内容は次の通りです。



文法と会話の得点を入力してから「合計点を計算する」ボタンを押すと

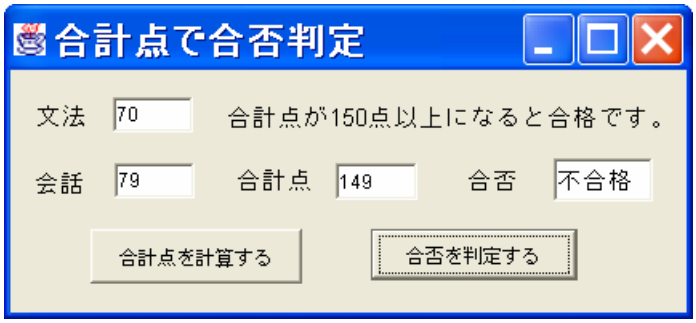


合計点が計算され、さらに「合否を判定する」ボタンを押すと



１５０点以上か否かに応じて、合否を判定する。

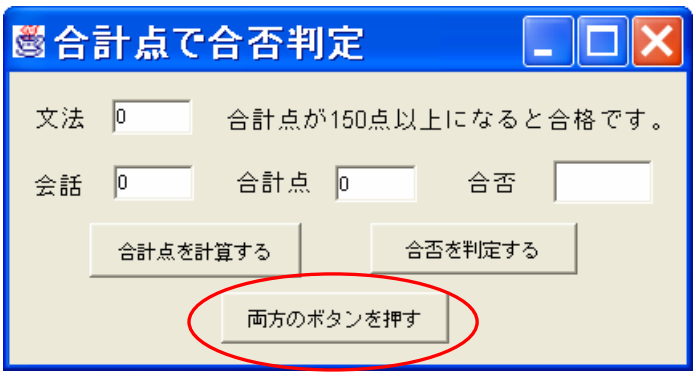
このプログラムは今までの学習範囲内で作成できるはずです。



6-2 メソッド（2）―1つのボタンですます―

【練習問題】

ボタンを 2 回押さずに、1 回ですます方法があります。フォームの下の方を少しだけ大きくして、第三のボタン「両方のボタンを押す」を付け加えて下さい（name プロパティは「jButton3」とします）。



第三のボタンを押したときの処理（イベントハンドラ）は、次のように書いて下さい。

```
void jButton3_actionPerformed(ActionEvent e) {
  jButton1_actionPerformed(e); //ボタン１を押せ
  jButton2_actionPerformed(e); //ボタン２を押せ
}
```

少し腑に落ちないかもしれませんが、とりあえず実行してみましょう。「文法」と「会話」の得点を入力してから、第三のボタンを押してみて、プログラムの動作を確かめましょう。

実行結果より、「合計点を計算する」ボタンも「合否を判定する」ボタンも使わなくてすむ事が分かったと思います。それなら、いっそのこと、これら 2 つのボタンの幅を 0 にして見えなくしたらどうなるでしょう（4-5 節の【練習問題】でも同様の事をやりましたね）。

予想　さて、ここで実行ボタンを押す前に、予想をして下さい。

① エラーが出て実行できない。
② エラーは出ないが、目的通りの動作はしない。
③ エラーは出ず、目的通りの動作をする。

あなたの予想は ＿＿＿＿＿＿

予想をしたら、実行ボタンを押してプログラムの動作を確かめましょう。

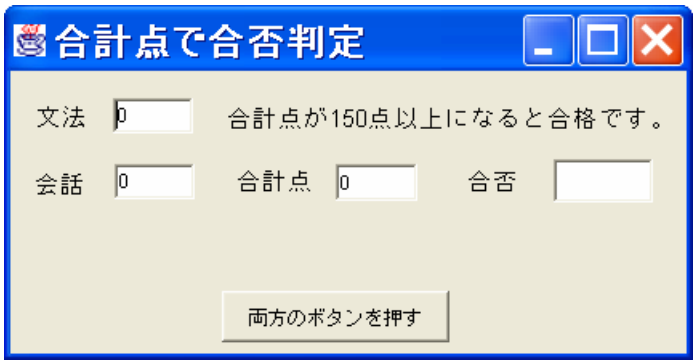
6-3 メソッド（3）―押せないボタンはいらないのか―

【練習問題】

「合計点を計算する」ボタンと「合否を判定する」ボタン を両方見えなくしても、プログラムは目的通りに動いと思います。

それならば、この２つのボタンをなくしてしまったらどうなるでしょう。

消したいボタンを選択して、DEL キーを押すとボタンが消えます。そこで、次のように２つのボタンを消しましょう。



予想　さて、ここで実行ボタンを押す前に、予想をして下さい。

① エラーが出て実行できない
② エラーは出ないが、目的通りの動作はしない
③ エラーは出ず、目的通りの動作をする

あなたの予想は＿＿＿＿＿＿＿＿

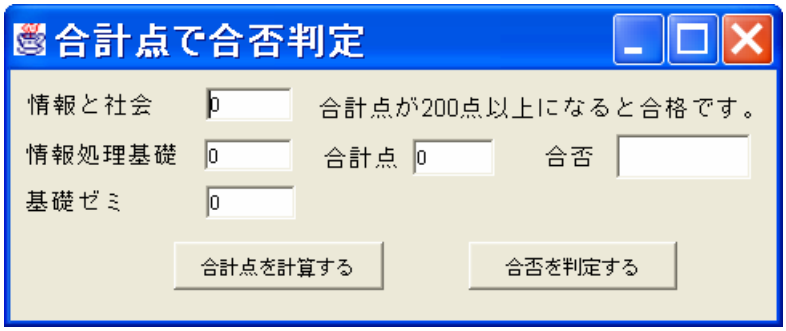
予想をしたら、実行ボタンを押してプログラムの動作を確かめましょう。

6-4　メソッド（4）―構造のはっきりしたプログラム―

今度は、３科目の合計点を計算してから、合否を判定するプログラムを作ります。

【基礎課題 6-4-1】

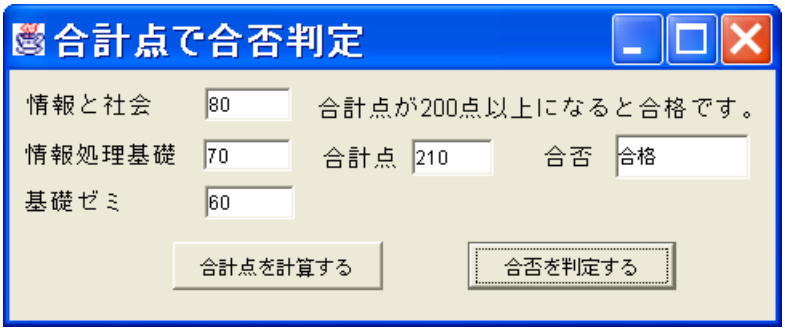
下のようなフレームを作って下さい。



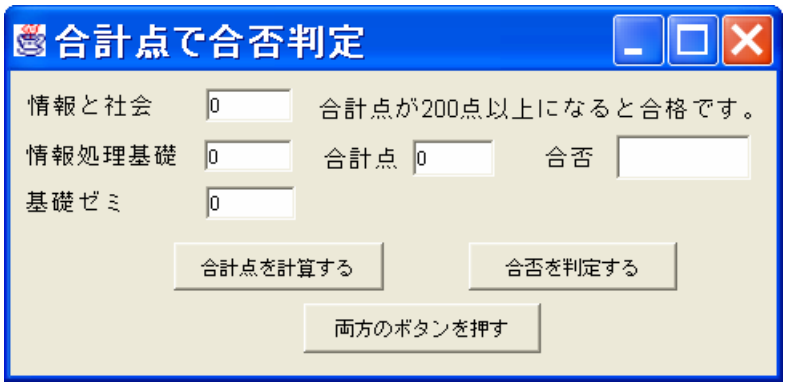
ボタンコンポーネントの name プロパティは、次の通りとします。

コンポーネント	name
ボタン「合計点を計算する」	jButton1
ボタン「合否を判定する」	jButton2

まず、それぞれのボタンに対して、それを押したときのイベントハンドラを記述して、プログラムが正しく動作するようにして下さい。

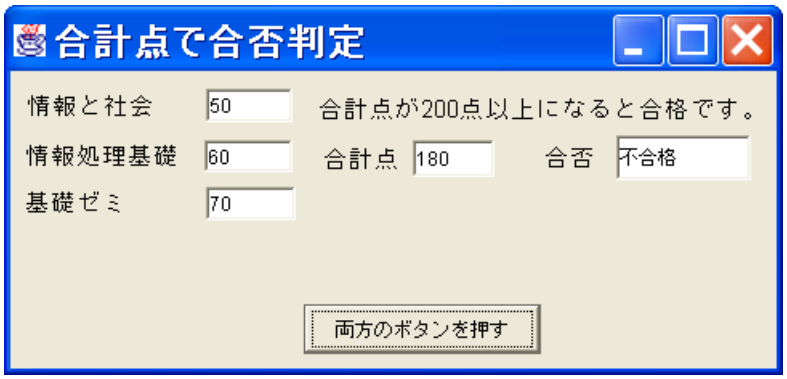


正しく動作することが確認できたら、第三のボタンを付け加えて、そのボタンを押したときのイベントハンドラを以下のように書いてください。




```
void jButton3_actionPerformed(ActionEvent e) {
  jButton1_actionPerformed(e); //ボタン１を押せ
  jButton2_actionPerformed(e); //ボタン２を押せ
}
```

最後に、フォームの上にある「合計点を計算する」ボタンと「合否を判定する」ボタンを削除してから、動作を確認してください。



前節同様、うまく動作するはずです。

6-5　メソッド(5) -jButton1 とjButton2 は本当にいらないのか-

【基礎課題 6-4-1】の改良を続けます。前節までの学習で分かった通り、最終的には「jButton1」と「jButton2」は必要ないようです。なぜなら削除してしまっても、きちんと動作するのですから・・・。ただ、両者のイベントハンドラ

`jButton1_actionPerformed`、 `jButton1_actionPerformed`

があれば、うまく行きそうです。

ところで、もはやボタンはなくなりしたがってクリックもできないのですから、「`jButton1_actionPerformed`」等という名称は実態に合いませんね。そこで、機能が分かるようにjButton1 については「Keisan」に、jButton2 のそれを「Hantei」に変えてみましょう。

上のように、イベントハンドラの名称を勝手に変更してもうまく動作するでしょうか？

予想　ここで実行ボタンを押す前に、予想をして下さい。

① エラーが出て実行できない。

② エラーは出ないが、目的通りの動作はしない。

③ エラーは出ず、目的通りの動作をする。

あなたの予想は＿＿＿＿＿＿＿＿

予想をしたら、実行ボタンを押してプログラムの動作を確かめましょう。

6-6 メソッド（6）—プログラムの構造—

もう一度コードエディタを見てみましょう。

Keisan

```
void Keisan(ActionEvent e) {
  int JohoS=Integer.parseInt(jTextFieldJohoS.getText());
  int JohoKiso=Integer.parseInt(jTextFieldJohoKiso.getText());
  int Zemi=Integer.parseInt(jTextFieldZemi.getText());
  int Total;
  Total=JohoS+JohoKiso+Zemi;
  jTextFieldTotal.setText(String.valueOf(Total));
}
```

Hantei

```
void Hantei(ActionEvent e) {
  int Total=Integer.parseInt(jTextFieldTotal.getText());
  if(Total>=200) {
    jTextFieldHantei.setText("合格");
  }
  else {
    jTextFieldHantei.setText("不合格");
  }
}
```

jButton3_actionPerformed

```
void jButton3_actionPerformed(ActionEvent e) {
  Keisan(e); //ボタン１を押せ
  Hantei(e); //ボタン２を押せ
}
```

このプログラムの（主な）処理は、

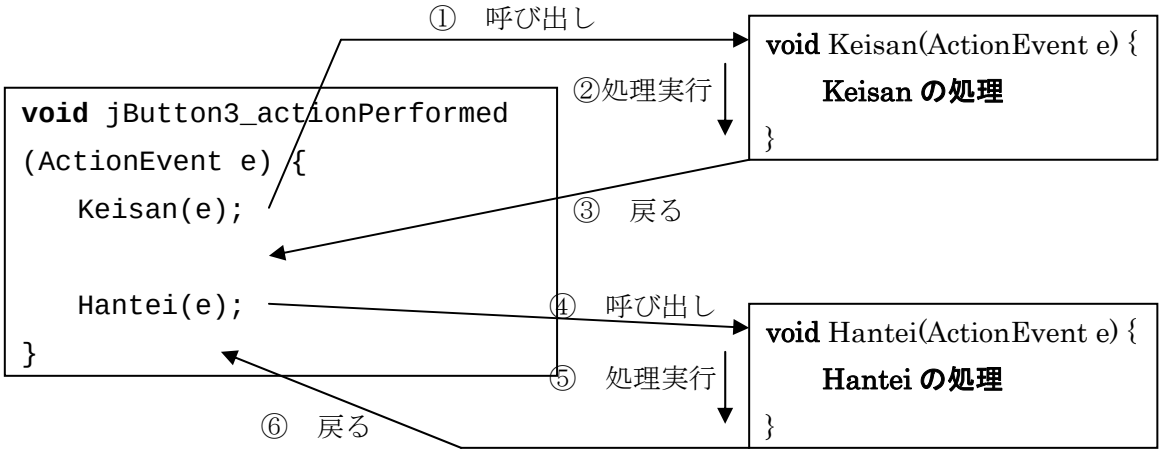
- Hantei
- Keisan
- jButton3_actionPerformed

という3つの部分からできています。

改めて説明すると、このような、プログラムの中の小さな独立部分が**メソッド**です。これまで用いてきたテキストフィールド・コンポーネントの **setText()**や **getText()**などの**メソッド**と全く同じモノです。

イベントハンドラもメソッドの一種です。しかし、メソッドはイベントハンドラに限らず、必要に応じて自在に定義することができます（そのやり方を学習するのが本章の主たる目的です）。実際、上の「**Hantei**」や「**Keisan**」は、（もはやボタンを削除してしまったので）イベントハンドラとしての意味はありません。ただ、jButton3 のイベントハンドラから呼び出されるだけの存在です。

さて、このプログラムでの、3 つのメソッドの関係は、次の図のようになっています。



今の場合、全体を統括するプログラムはjButton3_actionPerformedになっています。このように、全体統括のプログラムはシンプルに、細かな処理の部分はサブプログラムに任せるように書くのが、見やすいプログラムといえます。

コラム　ActionEvent とは？

jButton3_actionPerformed(**ActionEvent e**)などにある、（　）内のActionEventという**引数**がどういう意味を持っているのか、気になった人が多いと思います。これは、「ActionEvent型の変数e（正確にはActionEventクラスのオブジェクトe）をイベント発生時にイベントハンドラに渡す」という事を意味します。では、ActionEvent型変数eとはどのような意味を持つのでしょうか？実は、eには、

どのコンポーネントのどのようなイベントが発生したのか？

という情報を始め、イベントが発生したコンポーネントの様々な情報が記録されています。様々な情報が記録されているので変数ではなく、（複数の情報を記録できる）オブジェクトとなっている訳です。

その詳細は割愛しますが、では、なぜこのような引数を渡すようになっているのでしょう？それは例えば、イベントが発生したコンポーネントの種類や状態によって処理内容を変えたい（区分けしたい）場合に、その情報が必要になるからです。上の例で考えてみましょう。今jButton1～jButton3 のいずれをクリックしても**Keisan**メソッドに飛ぶとします。そして、どのボタンがクリックされたかによって、**Keisan**の処理内容を分岐させるとしましょう。このような場合、引数の（ActionEventクラスのオブジェクトである） e が役に立つのです。

6-7 戻り値のないメソッド

これまで何度も目にしてきた、イベントハンドラ（メソッド）を定義する

```
void jButton1_actionPerformed(ActionEvent e) {

}
```

という部分は、JBuilder が自動的に作ってくれました。少し、この書式をみてみましょう。Java 言語では、メソッドを定義する際の書き方は次のようになっています。

戻り値の型　　メソッド名（引数リスト）

これを上の例に当てはめると、対応は次のようになっています。

戻り値の型	メソッド名	引数
void	jButton1_actionPerformed	e

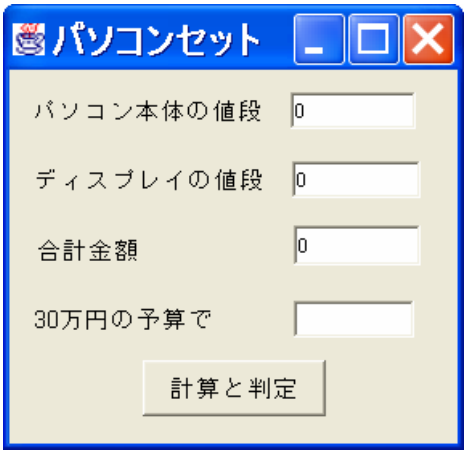
少し補足説明しておきましょう。

① 戻り値については次節【基礎課題 6-8-1】で説明します。
② 引数とは、この関数が呼びされた時、呼び出し元のプログラムから（呼び出された）メソッドの定義プログラムへ受け渡す変数のことです。これについても、次節【基礎課題 6-8-1】でもう少し説明します。

さて、上のメソッド定義部分は、自分で書くこともできます。その練習をしてみましょう。

【基礎課題 6-7-1】

下のようなフレームを作って下さい。



主なコンポーネントの name プロパティは、次のようにして下さい。

コンポーネント	name
ボタン「計算と判定」	jButtonMain

「パソコン本体の値段」と「ディスプレイの値段」を入力してから「計算と判定」ボタンを押すと、

① 合計金額を表示し、

② 合計金額が 300000 以下なら「買える。」そうでなければ「買えない。」と表示するプログラムを作ります。

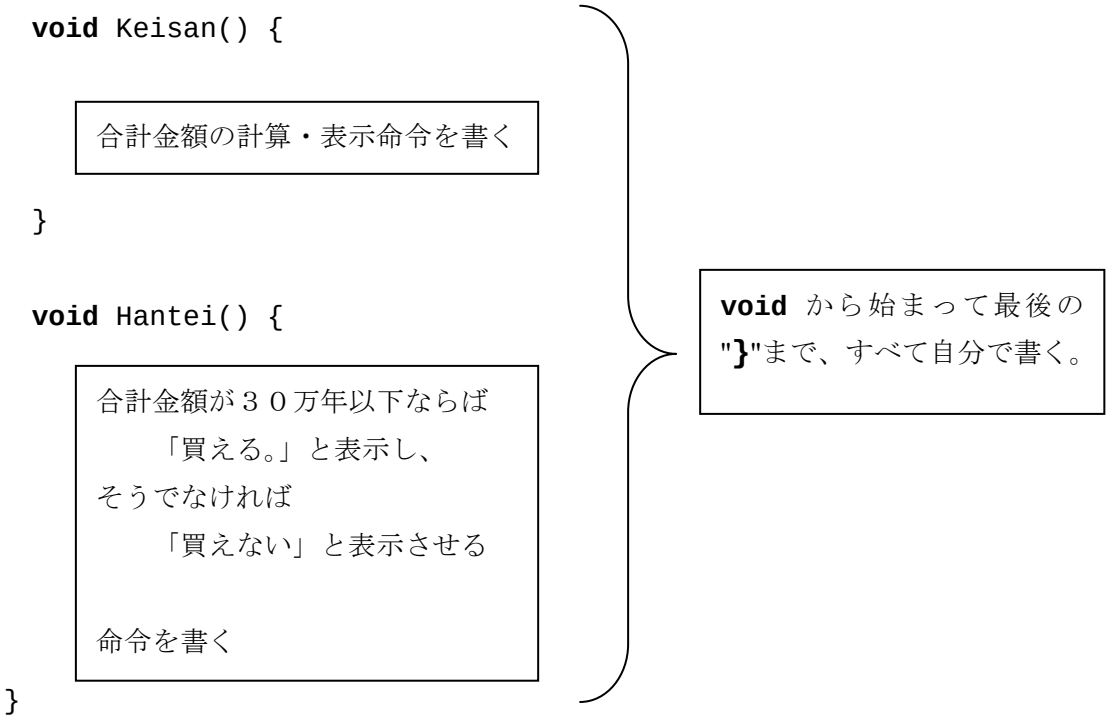
まず、jButtonMain を押したときのイベントハンドラを、以下のように書いて下さい。

```
void jButtonMain_actionPerformed(ActionEvent e) {
  Keisan();
  Hantei();
}
```

実際に入力するのはこの２行だけです

ここに、関数「Keisan」や「Hantei」にとって引数は必要ないので、( )内は空白にしています。しかし、( )自体は省略できないので注意して下さい。

次に、上に書いた「Keisan」と「Hantei」メソッドを、**自分で**次のように書きます。場所は、イベントハンドラ「jButtonMain_actionPerformed」の上でも下でも、どちらでも結構です。



作成したらプログラムを実行して動作を確認してください。

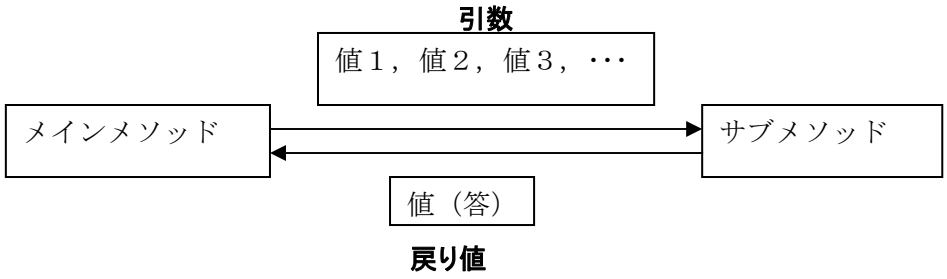
6-8 戻り値のあるメソッド(1) －平均値を求める－

これまでみてきた様に、一般に Java 言語のプログラム（より正確にはクラス）は複数のメソッドからなります。そして、全体を統括するメソッド（便宜的にメインメソッドと呼びましょう）は部分処理担当のメソッド（サブメソッドと呼びましょう）に実行命令を出し、指令を受けた（呼び出された）メソッドの方は決められた仕事をこなす、という処理の流れになっていました。→6-6 節の図参照

しかし、処理内容によっては、

1. メインメソッドがいくつかの値（引数）を渡し、
2. サブメソッドがそれをもとに値（答）を計算して、メインメソッドに（戻り値として）返す

という場面が必要になる場合があります。

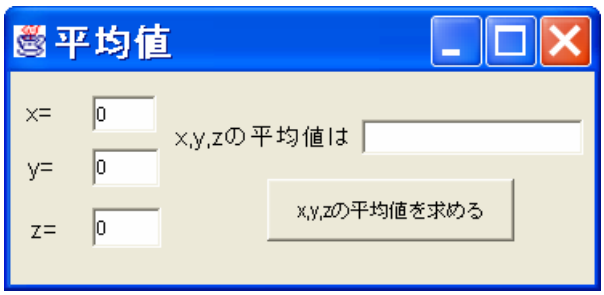


以下では、このような場合について考えて行きます。

【基礎課題 6-8-1】

3 つの数 x,y,z の値を渡すと平均値を計算して送り返すサブメソッドを作成・利用することで、以下のプログラムを完成させましょう。

まず、下のようなフレームを作って下さい。



主なコンポーネントの name プロパティは、次のようにして下さい。

コンポーネント	name
テキストフィールド「x」	jTextFieldX
テキストフィールド「y」	jTextFieldY
テキストフィールド「z」	jTextFieldZ
ボタン	jButtonMain
テキストフィールド「平均値」	jTextFieldHeikin

jButtonMain をクリックしたときのイベントハンドラは、次のように書いて下さい。

```
void jButtonMain_actionPerformed(ActionEvent e) {
  int x=Integer.parseInt(jTextFieldX.getText());
  int y=Integer.parseInt(jTextFieldY.getText());
  int z=Integer.parseInt(jTextFieldZ.getText());
  double a;
  a=Average(x,y,z); //平均値を求めるメソッドを呼び出す
  jTextFieldHeikin.setText(String.valueOf(a));
}
```

Average メソッドを作ります。下のように書いて下さい。

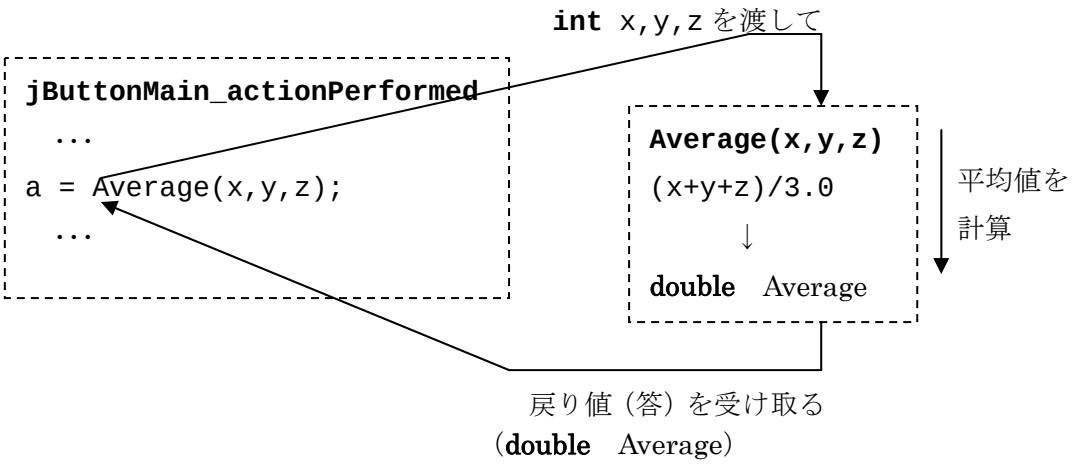
```
double Average(int x,int y,int z) {
  return (x+y+z)/3.0;
}
```

解説　Average(x,y,z)メソッドの定義について

- Average は、このメソッドの名前です。
- x、y、zは、メインメソッド（jButtonMain_actionPerformed）からAverageメソッドに渡される値、つまり**引数**です。
- Average(**int** x,･･･)の **int** は、当該引数が **int** 型（整数型）の変数であることを表しています。
- 引数がないメソッドの場合は、前節の keisan()などのように、（　）内を空白にします。
- **double** は、Average の「戻り値」、要するに Average の値が double 型（実数型）であることを表しています。
- 「**return** 式」の形で、「式」の計算結果（値）が Average の「戻り値」として与えられます。上の例では、（x,y,x）の平均値が戻り値として定義されます。
- 「(x+y+z)/3.0」と、分母を「3.0」としている理由については、【基礎課題 4-7-2】のコラムを参照してください。
- 前節までのように、戻り値のない関数の場合、戻り値の型としては「**void**」と書きます。void は「空っぽ」という意味です。

このプログラムの処理の流れは次にようになります。

＜ 処理の流れ（イメージ） ＞



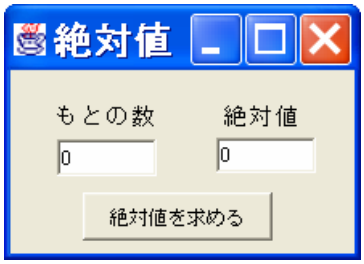
それでは、実行して動作を確認して下さい。

6-9　戻り値のあるメソッド(2)　－絶対値を求める－

今度は、数の絶対値を求めるメソッドを作ります。

【基礎課題 6-9-1】

下のようなフレームを持つプログラムを作って下さい。



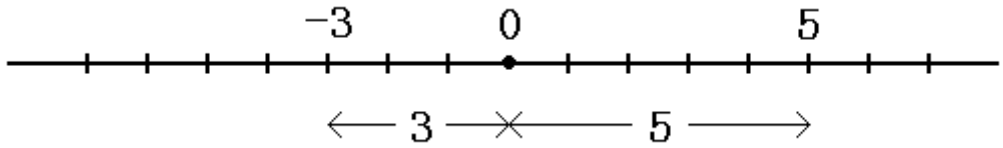
主なコンポーネントの name プロパティは、次のようにして下さい。

コンポーネント	name
左のテキストフィールド	jTextFieldFrom
右のテキストフィールド	jTextFieldTo
ボタン「絶対値を求める」	jButtonMain

まず、jButtonMain をクリックしたときの処理を、次のように書いてください。

```
void jButtonMain_actionPerformed(ActionEvent e) {
  int a,b;
  a=Integer.parseInt(jTextFieldFrom.getText());//「もとの数」をaに代入
  b=Zettaichi(a);  //aの絶対値をbに代入
  jTextFieldTo.setText(String.valueOf(b));  //bを「絶対値」欄に代入
}
```

次に、**Zettaichi** メソッドを作ります。
その前に、「絶対値」とはなんだったか、復習しておきましょう。ある数の絶対値とは、その数を数直線上にとった場合の、原点からの距離のことを指します。



－３の絶対値は、３　　Zettaichi(-3) ＝3
５の絶対値は、　５　　Zettaichi(5) ＝5

0 の絶対値は、＿＿＿＿＿＿
a の絶対値は、　a が正の数のとき、＿＿＿＿＿＿　　a が負の数のとき、＿＿＿＿＿＿

下の下線部を埋めて、絶対値メソッドを完成させてください。

```
int Zettaichi(int a) {
  if(a >= 0) {
    return ________;
  }
  else {
    return ________;
  }
}
```

完成したら実行し動作を確認してみましょう。

6-10　戻り値のあるメソッド(3)　ー数学関数 Math.abs( )ー

【基礎課題 6-10-1】

前節で作った絶対値を求めるプログラムを改変します。【基礎課題 6-9-1】を利用しましょう。以下の指示に従って下さい。

まず、プログラムから **Zettaichi** メソッドの定義部分を削除してください。

```
void jButtonMain_actionPerformed(ActionEvent e) {
  int a,b;
  a=Integer.parseInt(jTextFieldFrom.getText());//「もとの数」をaに代入
  b=Zettaichi(a);  //aの絶対値をbに代入
  jTextFieldTo.setText(String.valueOf(b));  //bを「絶対値」欄に代入
}
```

```
int Zettaichi(int a) {
  if(a>=0) {
    return a;
  }
  else {
    return -a;
  }
```

削除する

そして、ボタンのイベントハンドラ中の「Zettaichi」を「Math.abs」に変えてください。

Math.abs に変える

```
void jButtonMain_actionPerformed(ActionEvent e) {
  int a,b;
  a=Integer.parseInt(jTextFieldFrom.getText());//「もとの数」をaに代入
  b=Zettaichi(a);  //aの絶対値をbに代入
  jTextFieldTo.setText(String.valueOf(b));  //bを「絶対値」欄に代入
}
```

すべて保存してからプログラムを実行して、動作を確かめてください。

先ほどと同様うまく動作するはずです。でもなぜうまく行ったのでしょうか？・・・

一体、**Math.abs()**とは何なのでしょうか？　順を追って考えてみましょう。

① 「Math.abs()」の形を見ると、これまで色々なメソッドを用いてきた経験から

Math というクラスに定義された **abs()**というメソッドである

と考えられます。

② 上の動作確認から

Math.abs()はメソッド Zettaichi()と同じ働きをする

という事が分かります。

③ すると、

abs()は引数の値の絶対値を戻り値として返す（戻り値のある）メソッドらしい

という事が予想されます。

少し回りくどい説明をしましたが、上の点はこれまでの知識から推測できる事です。このように考える事ができたでしょうか？

さて、それでは **Math** クラスについて解説を行いましょう。Java 言語では、計算上よく用いられる数学上の関数を納めた **Math** クラスというクラス（通常「**算術クラス**」と呼ばれます）が用意されています。Math は Mathematics（数学）の略です。上の abs()メソッドは絶対値を求めるメソッドなのです。この他にも以下のようなメソッドが用意されています。Math クラスのメソッドについては、**関数**と呼ぶ方がしっくり来るでしょう。

Math クラスのメソッド（関数）の例

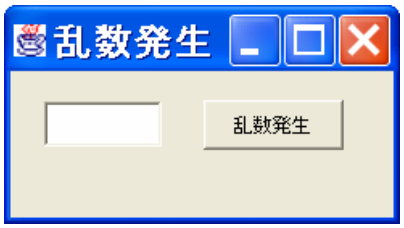
関数名	例
Math.abs()（絶対値関数）	Math.abs(-3)→3
Math.sqrt()	Math.sqrt(9)→3
Math.sin()（サイン関数）	Math.sin(0)　→　0
Math.cos()（コサイン関数）	Math.cos(0) → 1
Math.log()（対数関数）	Math.log(2.71828)→0.999999327347282
Math.exp()（指数関数）	Math.exp(1)→2.71828182845905
Math.max()（最大値関数）	Math.max(1,3)→3
Math.min()（最小値関数）	Math.min(1,3)→1

6-11　戻り値のあるメソッド(4)　ー数学関数 Math.random()ー

Java 言語が定義している数学関数でよく使うものに、**random()**があります。**random()**は、ランダムな数（**乱数**）を返すメソッド（関数）です。

【練習問題】

下のようなフレームを持つプログラムを作って下さい。



コンポーネント	name
テキストフィールド	jTextFieldNumber
ボタン「乱数発生」	jButton1

ボタンをクリックしたときのイベントハンドラを、次のように書いてください。

```
void jButton1_actionPerformed(ActionEvent e) {
  int Num; //乱数保管用の変数
  Num=(int) (10*Math.random()); //実数型→整数型への「型キャスト」
  jTextFieldNumber.setText(String.valueOf(Num));
}
```

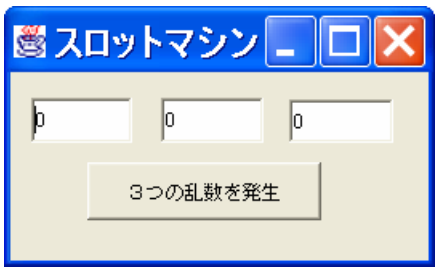
解説

- Math.random()は、０≦Math.random()＜1 の範囲の数のいずれかの値を、ランダムに（不規則に）返します。
- Math.random()は実数型（**double** 型）のメソッド（関数）です。
- (10*Math.random())と１０倍することによって、0≦Math.random()＜10 の範囲の乱数を発生させます。このようにして倍数を調節することで、任意の範囲の乱数を発生させる事ができます。
- 「Num=(**int**) (10*Math.random());」の部分で**型キャスト**を用いて、整数型に変換しています。その際、小数点以下は切り捨てられますから、Num は 0～9 までの範囲の（**整数の**）乱数となります。型キャストについては、4-7 節の解説を参照して下さい。

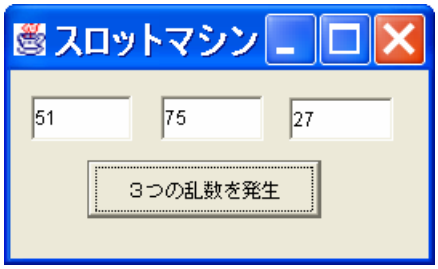
それでは、実行して動作を確かめてみてください。

【基礎課題 6-11-1】

右のようなフレームを作って下さい。



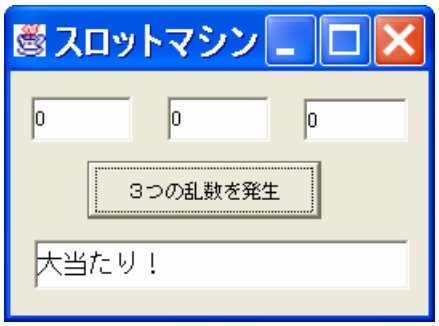
ボタンを押すと３つのテキストフィールドに０から９９までの（整数の）乱数を発生させるようにしてください。



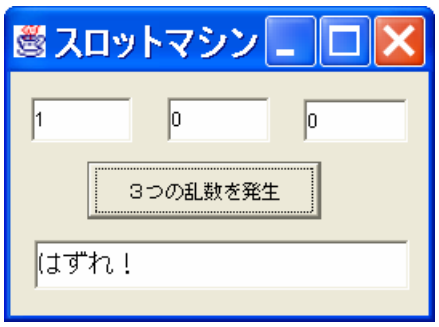
【応用課題 6-11-A】

次の様なプログラムを作って下さい。

ボタンを押すと３つのテキストフィールドに０から２までの乱数を発生させるようにしてください。そして、３つの数がそろったら「大当たり！」と表示して下さい。



３つの数がそろっていないときは「はずれ！」と表示して下さい。



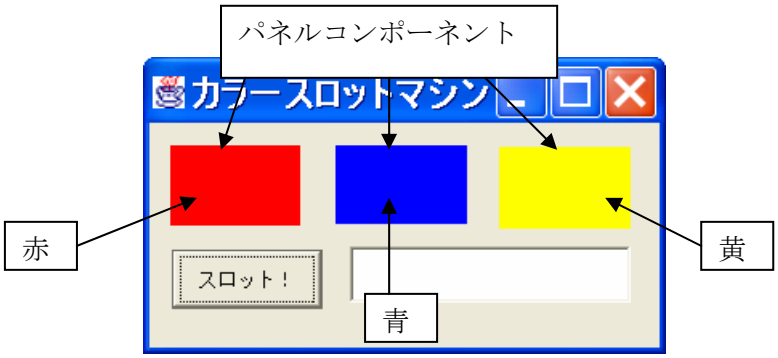
ヒント　３つの乱数を N1、N2、N3 とすると、３つとも揃う条件とは

（N1 と N2 が等しい）かつ（N2 と N3 が等しい）

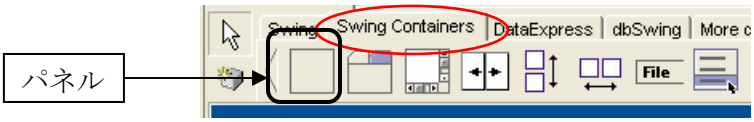
となります。

【応用課題 6-11-B】

下のようなフレームを持つ、パネルの色が赤／青／黄にランダムに変わるカラースロットマシンを作ります。



色の変わるパネルは、「パネルコンポーネント」を使います。パネルコンポーネントはコンポーネントパレットの「Swing Containers」タブにあります。これを、フレームに３つ貼ります。



パネルの色は上のように、最初は左から順に赤、青、黄にしておきます。コンポーネントの色の設定は **background** プロパティを変更すれば良かったですね（2-3 節参照）。

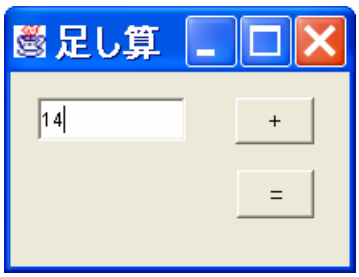
それでは、パネルの色をランダムに変えるプログラムを考えましょう。色々な方法がありますが、例えば、０、１、２の乱数を発生させて、０のとき赤、１のとき青、２のとき黄にするようにしましょう。コンポーネントの色をプログラム中で変更する方法は、【基礎課題 3-3-3】で学習しました。

最後に、色がそろったときには「大当たり！」、そろわなかったときには「はずれ！」のメッセージを出すのも忘れないでください。

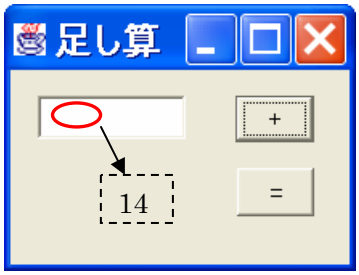
6-12　ローカル変数とインスタンス変数

【基礎課題 6-12-1】

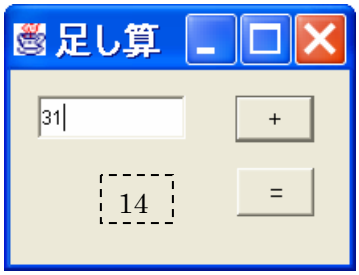
足し算だけができる簡単な電卓を作ってみましょう。



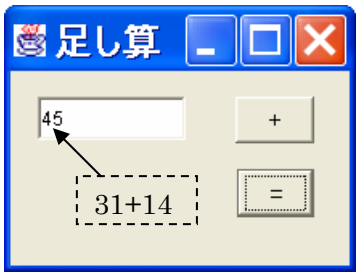
テキストフィールドに適当な数を入力して



「+」ボタンを押すと画面には現れない場所（変数）に値が保存され、テキストフィールドは空欄になり、



その後、テキストフィールドに別な数を入力して「=」ボタンを押すと



和が求まる、というプログラムです。

コンポーネントの name プロパティは次の通りとします。

コンポーネント	name
テキストフィールド	jTextFieldNumber
ボタン「＋」	jButtonPlus
ボタン「＝」	jButtonEqual

このプログラムでは、以下に示すように、整数型変数 **PrevNumber** を用意して、「+」ボタンが押されたらそこに値を格納するようにします。

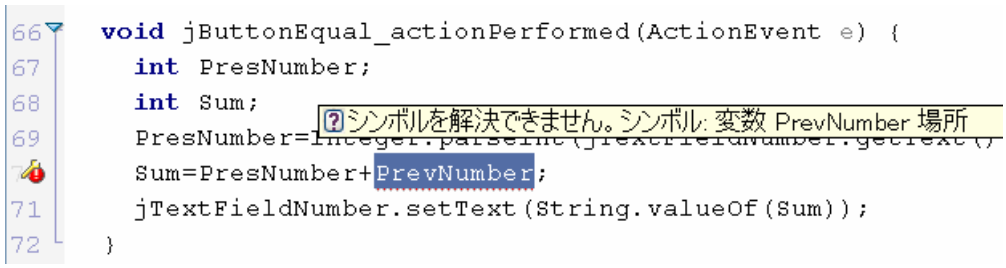
まず、「+」ボタンのイベントハンドラを書きましょう。

```
void jButtonPlus_actionPerformed(ActionEvent e) {
  int PrevNumber; //ここで変数 PrevNumber を用意
  PrevNumber=
        Integer.parseInt(jTextFieldNumber.getText());//値を格納する
  jTextFieldNumber.setText(""); //欄を空欄にする
}
```

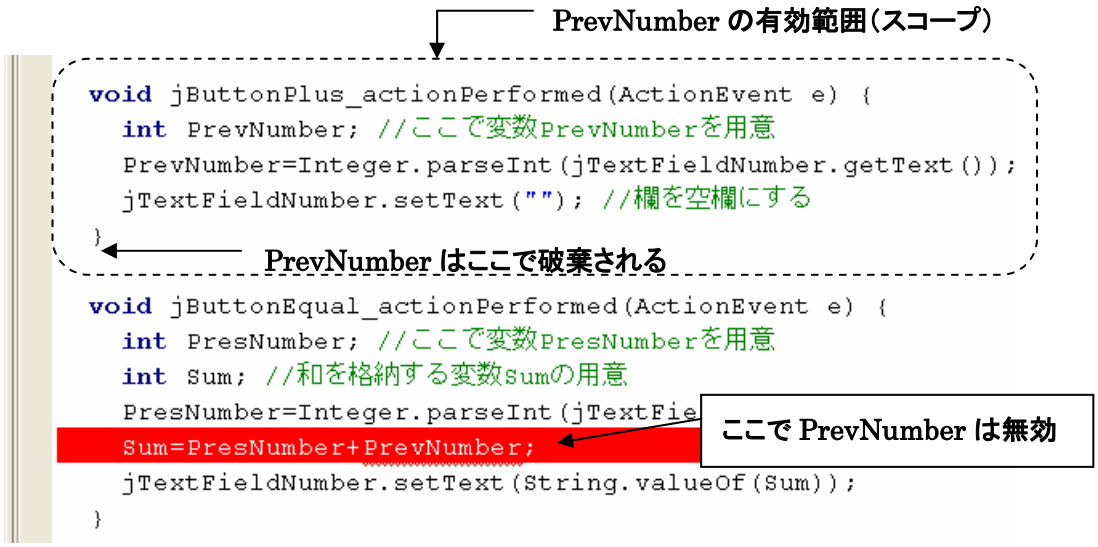
次に、「=」ボタンのイベントハンドラを書きましょう。

```
void jButtonEqual_actionPerformed(ActionEvent e) {
  int PresNumber; //（今現在の）欄内の値を格納する変数 PresNumber を用意
  int Sum; //和を格納する変数 Sum の用意
  PresNumber=
        Integer.parseInt(jTextFieldNumber.getText());//値を格納する
  Sum=PresNumber+PrevNumber; //和を計算する
  jTextFieldNumber.setText(String.valueOf(Sum));
}
```

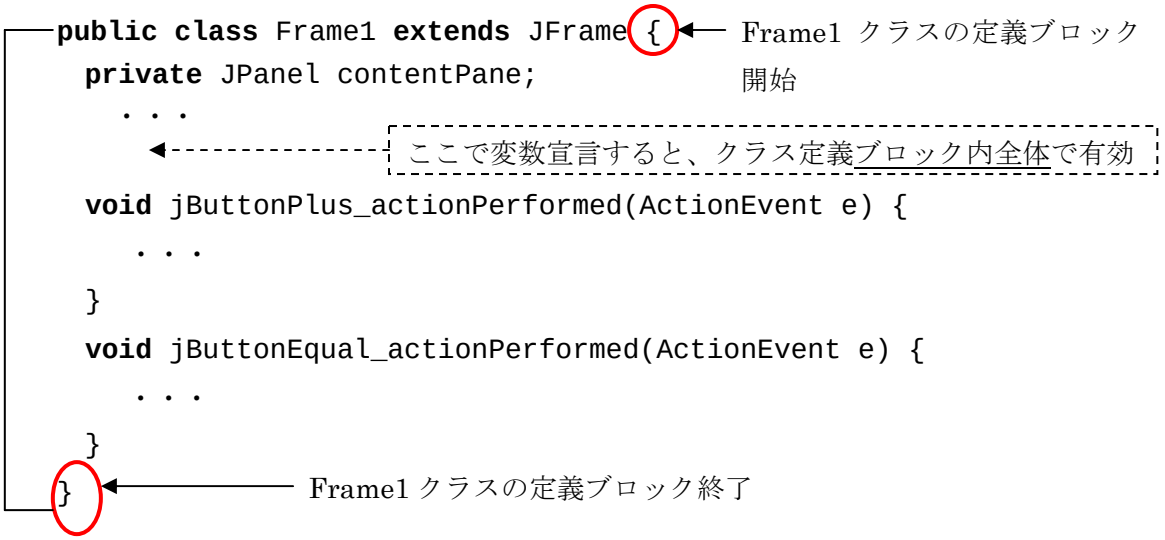
すでにこの時点で JBuilder からエラーの警告が出されているはずです。そこで、赤色の下線で示されたエラー箇所にカーソルを合わせると次のようにエラーメッセージが表示されます。



これは、4-6 節の「補足　エラーメッセージについて」で説明したように、「PrevNumber という変数が存在しない、つまり宣言されていない。」という意味のメッセージでしたね。でも、PrevNumber は確かに宣言したはずなのに、なぜ「見つけられない」のでしょうか？実は、Java言語には、「{　} で囲まれたブロック内で宣言された変数はそのブロック内でのみ有効である。」という決まりがあります。そして、メソッドの定義も一つのブロックです。ですから、メソッドでという一つのブロック内で定義された変数は、そのメソッド内でのみ有効なのです。具体的に示すと次のようになります。



では、どうしたらよいのでしょう? それは、上に述べた変数の有効範囲に関する原則をあてはめれば分かるように、メソッドの外側のブロック内で変数を定義すればよいのです。実は今の場合、各メソッドの定義は、以下に示すようにFrame1 というクラスの定義（ブロック）の中で行っています（改めてソースをスクロールして全体構造を確かめてみて下さい）。なお、クラスの定義の仕方については、第 7 章で学習します。



複数のメソッドで有効な変数を宣言するには、メソッド定義のブロックの外（そして Frame1 クラスの定義ブロックの中）であればどこでも良いのですが、分かりやすくするために、ここでは Frame1 クラス定義の先頭部分へ移動し、そこに次ページのように波線部を挿入して下さい。

```
public class Frame1 extends JFrame {
  private JPanel contentPane;
  private JTextField jTextFieldNumber = new JTextField();
  private JButton jButtonPlus = new JButton();
  private JButton jButtonEqual = new JButton();
  int PrevNumber; //一つ前の値を格納する変数
  //フレームのビルド
```

挿入

また、これで先ほど、メソッド定義のブロック内で行った変数宣言はいらなくなりました。波線部を削除してください。

```
  void jButtonPlus_actionPerformed(ActionEvent e) {
    int PrevNumber; //ここで変数 PrevNumber を用意
    PrevNumber=
          Integer.parseInt(jTextFieldNumber.getText());//値を格納する
    jTextFieldNumber.setText(""); //欄を空欄にする
  }
```

削除

これで実行してみましょう。　　今度はちゃんと動作するはずです。

このように、クラスの定義のブロック内のどのメソッドからも利用することができる変数を、Java 言語では**インスタンス変数**（下のコラム参照）と言います。一方、「メソッド」の中で宣言された変数は「ローカル変数（局所変数）」 と呼ばれます。また変数の有効範囲を**スコープ**と言います。

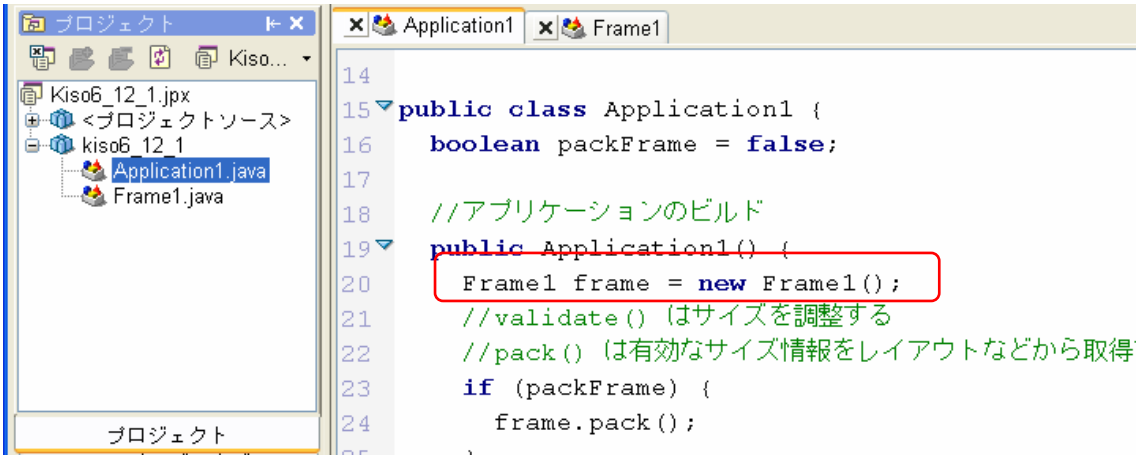
コラム　インスタンス変数について

またここで、**インスタンス**という聞き慣れない言葉が出てきました。インスタンス(instance)とは実例という意味で、あるクラスから生成されたオブジェクトのことを指します。3-1 節のコラムでも述べましたが、クラスはプロパティやメソッドという性質を定義した設計図のようなものです。そしてその設計図に従って作られた製品がインスタンスである、という訳です。このように、インスタンスはオブジェクトと同じ意味を持つのですが、Java 言語では、オブジェクトはインスタンスに加えて（4-12 節で学習した）配列も含むので、オブジェクトの方が少し広い概念と言うことになります。ともかく、実際に生成したインスタンスの中（全体）で有効な変数という意味で、**インスタンス変数**と名付けられている訳です。なお、同種のものに**クラス変数**がありますが、ここでは扱いません。

コラム　Frame1 クラスのインスタンスについて

さて、上のプログラムでは、Frame1 というクラスを定義しています。でも、「その Frame1 クラスのインスタンス（オブジェクト）はどこで生成されているの？」と疑問に思った人がいるかも知れません。実は、Frame1 クラスのインスタンスは Application1.java というプログラム内で生成されています。それを確認してみましょう。

プログジェクトペインの「Application1.java」をダブルクリックして下さい。すると、次のように内容ペインに Application.java のソースコードが表示されるはずです（下の画面は、若干下方にスクロールしています)。



ここに、

```
Frame1 frame = new Frame1();
```

という部分で、Frame1 クラスのインスタンスをframeという名前で生成しています（この記述方法やその意味については第７章で学習します)。つまり、Application1.javaというソースコードはFrame1 クラスのインスタンスを生成させる、という役割を持っているのです。以上より、一般にJBuilderで作成するアプリケーションでは、２つのソースコードに以下のような役割分担をさせていることが分かります。

ソースコード	役　　割
Frame1.java	ボタンやテキストフィールドなど、処理に応じたコンポーネントを配置し、さらに必要なメソッドを有する Frame1 クラスを定義する。
Application1.java	実際のプログラム実行に呼び出され、Frame1 クラスのインスタンスを生成させる。

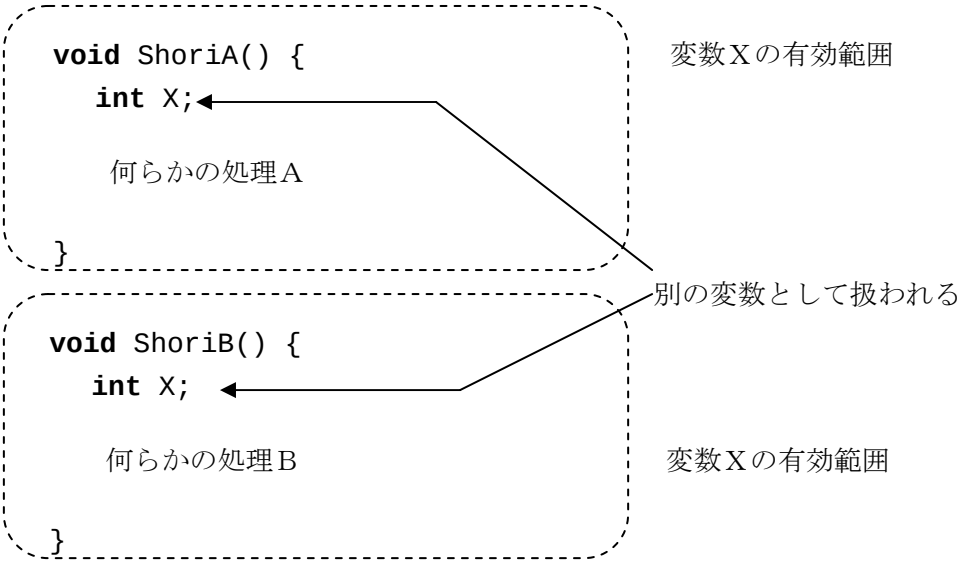
そして、これら２つのソースコードを一つのプロジェクトにまとめているのです。もちろん、このように２つのソースコードに分解せずに一つにまとめて書くこともできます。しかし、「クラスの定義」と「そのインスタンスを生成し利用する」部分に分けた方が、拡張が容易でまた管理もしやすいので、このようなスタイルをとっているのです。

コラム　全てインスタンス変数でも良いのでは？

「どんな変数もインスタンス変数にするほうが便利なのでは?」と考える人もいるかもしれません。すべての変数をインスタンス変数にするとどうなるでしょう?

1．インスタンス変数の「どのメソッドからも利用できる」という性質は、言い換えれば「どのメソッドからでも代入によりその値を変更することができる」ということです。これはつまり、変数の値がおかしくてプログラムが正しく動かなかったりエラーが発生したりした場合にはすべてのメソッドを疑わなければならないことを意味しています。逆にローカル変数であれば、他のメソッドからは手が出せないことが保証されているので、該当するメソッドだけを調べればよいのです。つまり、エラーの犯人捜しの”捜査範囲”が狭められるわけです。

2．当然の事ですが、変数のスコープ（有効範囲）内で同じ名前の変数を用いることはできません。したがって、全ての変数をインスタンス変数にすると、重複を避けるため、変数の名前をたくさん用意しなければなりません。しかし、ローカル変数はそのメソッド内だけで有効ですから、下の例のように、同じ名前の変数を複数のメソッドで使うことができます。



3．プログラミングの世界では、複数のメソッドで共有する必要のない限りは「ローカル変数」を使うのが鉄則とされています。