第４章　変数と型

【学習内容とねらい】

本章では、「変数」について学習します。変数といっても数学の方程式に出てくる変数とはちょっと意味が違います。プログラミングの世界では、変数とは、データの記憶（格納）場所のことを指します。　・・・　と言っても、まだピンと来ないかもしれませんね。しかし、実はすでに皆は「変数」に“遭遇”しているのです。

例えば、「テキストフィールド」コンポーネントの **text** プロパティにはテキストフィールド欄に入っている“文字列”が記憶されていますし、**enabled** プロパティには入力可能かどうか（True あるいは False）が記憶されていました。この意味でコンポーネントのプロパティは**変数**だったのです。ただし、この「text プロパティの変数」と「enabled プロパティの変数」には大きな違いがあります。というのは、**text** プロパティには任意の文字列が格納されますが、**enabled** プロパティには **True** か **False** のいずれかしか受け付けません。ですから一律に扱うことはできません。どうやら、変数はその性質に応じて“タイプ分け”しておくことが必要なようです。そこで、「変数の型」という概念が現れます。

以上を念頭において、本章の各節を一つ一つ確かめながら読み進んで行けば、「変数」および「変数の型」の概念を理解することができるはずです。変数はプログラミングの基礎中の基礎であり、また同時に初心者がつまずきやすいところでもあります。どうか、ていねいに流れを追いながら学習して行って下さい。

＜第4章の構成＞

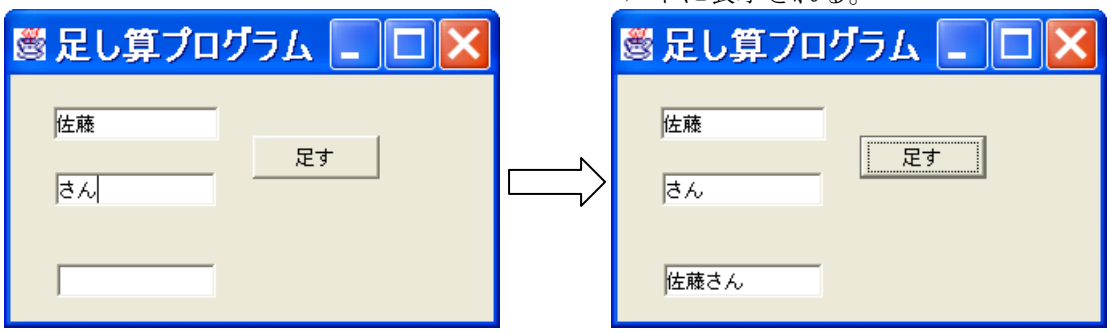
4-1　「+」演算子（1）―テキストフィールドによる足し算―

【基礎課題 4-1-1】

次のようなプログラムを作ってみましょう。

起動後、上の２つのテキストフィールドに適当な文字列を入力する。

入力後、[足す] ボタンをクリックすると、両者が連結されて３番目のテキストフィールドに表示される。



まずは上のようにコンポーネントを配置して下さい。各コンポーネントの name プロパティは次の通りとします。

コンポーネント	name
上のテキストフィールド	jTextField1
中のテキストフィールド	jTextField2
下のテキストフィールド	jTextFieldResult
ボタン	jButtonAdd

次に、ボタンを押したときの動作をプログラミングします。[足す] ボタンをクリックした時のイベントハンドラは次のようになります。

```
private void jButtonAddActionPerformed(ActionEvent evt) {
   jTextFieldResult.setText( jTextField1.getText()
                            +jTextField2.getText() );
}
```

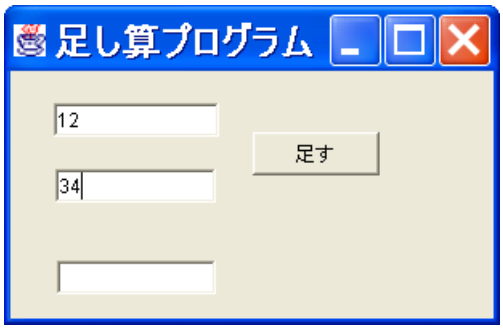
説明は後回しにして、とりあえず上のようにプログラミングして実行し、動作を確認してください。

さて、前章まで学習した知識で大体の点は理解できると思います。ただ１点新しい事項は、

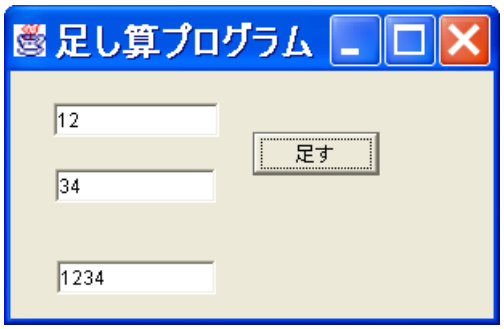
＊　文字列の足し算（連結）は「＋」演算子で行う。

という点だけです。「＋」演算子を用いればいくつでも文字列をつなげる事が可能です。

これは、あまり問題なく理解できるでしょう。では、今度は通常の数値としての足し算を試みてみましょう。



次のように、二つの整数を入力して［足す］ボタンをクリックしてみましょう。答えはどうなるでしょうか？



整数の足し算（のつもり）なので、両者の和（12+34=**46**）が表示されて欲しいところですが、実際は文字列の連結として、次の様な結果になってしまいます。

これは、文字列の連結としては当然の結果なのですが、このままでは、簡単な足し算もできません。数値の足し算を行うのは、一体どうすればよいのでしょうか？それを次節で学習しましょう。

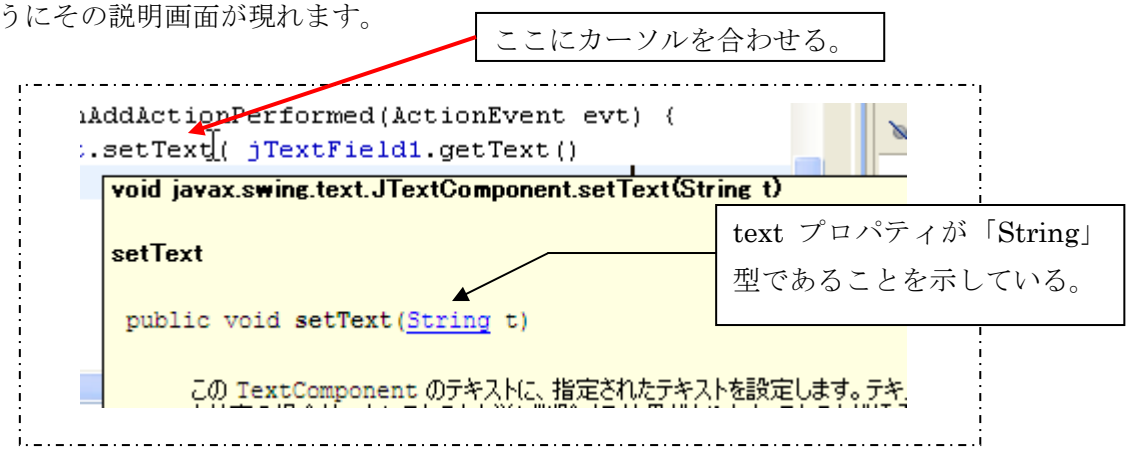
4-2 データの型

コンピュータで扱うデータには、文字や数値などさまざまな種類があります。それらをプログラミング言語では、データの**型**として区別しています。Java 言語では、文字列と整数に対して次のように型名が付けられています。

型の名前	日本語名	データ形式	足し算の結果
String 型	文字列型	文字	12 + 34 → 1234
int 型	整数型	整数	12 + 34 → 46

※　整数型の「int」は英語の「integer（整数）」の略から来ています。

実は、text プロパティは、**String** 型になっています。それは次のように確かめられます。プログラム中の `setText()`メソッドを記述している部分にカーソルを合わせると、下のようにその説明画面が現れます。



これにより、text プロパティに指定すべきデータの値（つまり text プロパティの値）が **String** 型であることが分かります。上と同様にして各プロパティの型を調べることができます。

ともかく、これで、text プロパティが String 型であることは分かりました。そしてそれは同時に、テキストフィールドに入力する限り（String 型なので）、整数として扱うことはできない事も意味します。何とか整数型（int 型）に変えることはできないのでしょうか？次節でその方法を学習しましょう。

4-3 型変換 ―整数型と文字列型の相互変換―

前節で述べたように、プログラミングの際には、「整数型←→文字列型」の変換手段が必要になります。そのために Java 言語では、次のようなメソッドを用意しています。

文字列型　→　整数型の変換	整数型　→　文字列型の変換
`Integer.parseInt(文字列)`	`String.valueOf(整数)`

＜ 解説 ＞

① 整数型（int 型）のデータに関する様々な操作を行うために、Java 言語では「Integer」クラスを用意しています。**`parseInt()`**はそこに定義されているメソッドです。

② 文字列型を表す String は String クラスというクラスでもあります。やはり、String クラスにも文字列に関する操作に必要なメソッドやプロパティが用意されており、**`valueOf()`**がその代表的なメソッドです。

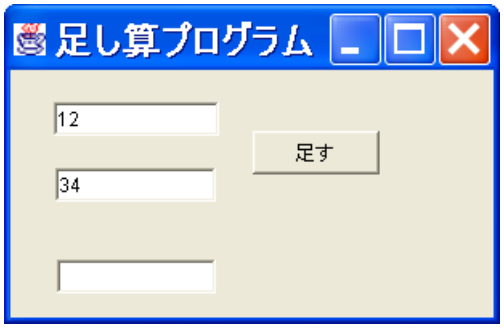
◆ 使用例

- 文字列"123"を整数 123 に変換する場合
  `Integer.parseInt("123")` → 123 という整数になる
- 整数 123 を文字列"123"に変換する場合
  `String.valueOf(123)` → "123"という文字列になる

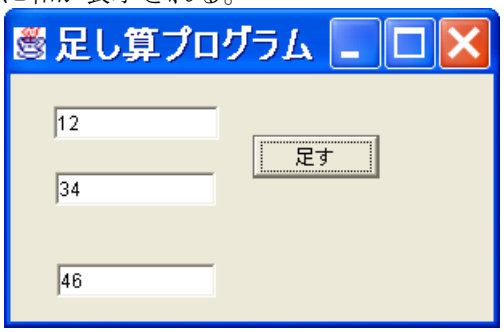
【基礎課題 4-3-1】

【基礎課題 4-1-1】を修正して、整数の和を求め表示するプログラムを作成しましょう。

上の２つの欄に整数を入力する。



［足す］ボタンをクリックすると、最下欄に和が表示される。



コンポーネントの name プロパティは【基礎課題 4-1-1】と同じとします。

例えば、「12+34」の演算結果（和）を一番下のテキストフィールドに表示させるためには、ボタンのイベントハンドラは次のようになります。

```
private void jButtonAddActionPerformed(ActionEvent evt) {
  jTextFieldResult.setText(String.valueOf(12+34));
}
```

すなわち、String.valueOf()メソッドで整数を文字列に変換してから、テキストフィールドの setText()メソッドに入れればよいのです。

これを、２つのテキストフィールドに入れた任意の整数の和を求めるようにするためには、上のプログラムで

```
12  →  Integer.parseInt(jTextField1.getText())
34  →  Integer.parseInt(jTextField2.getText())
```

と置き換えれば良いのです。そうすれば、もう求めるプログラムは分かるでしょう。

プログラムを作成したら、実行し色々な整数を入力してみて動作を確認してみてください。ともかくこれで、足し算を行うプログラムが完成しました。

4-4 改行・空白・コメント・大文字小文字

ここで、少し一息入れて、Java 言語のルール（約束事）を簡単に整理しておきましょう。

Java 言語のルール

(1) どこで改行しても、しなくてもよい。

(2) 空白がいくつ並んでいても１つの空白とみなす。

(3) 「/*」と「*/」で囲まれた部分、および「//」から行末までは無視される。

(4) 大文字と小文字を区別する。

（１） 次の２つのプログラムは同じものです。

```
private void jButtonAdd_actionPerformed(ActionEvent evt) {
  jTextFieldResult.setText(jTextField1.getText()+jText･･･);
}
```

```
private void jButtonAddActionPerformed(ActionEvent evt) {
  jTextFieldResult.setText(    改行
     jTextField1.getText()+jTextField2.getText() );
}
```

ただし、

```
private void jButtonAddActionPerformed(ActionEvent evt) {
  jTextFieldResult.set    改行
  Text(jTextField1.getText()+jTextField2.getText() );
}
```

というように字句の途中で改行してはいけません。

（２）次のプログラムは文法的には正しいプログラムです。

```
  private void jButtonAddActionPerformed(ActionEvent evt) {
jTextField1.setText("abc");
    jTextField2.setText("def");
  jTextFieldResult.setText(jTextField1.getText()
+       jTextField2.getText());
  }
```

しかし、このプログラムは、「**{　}** によって囲まれた部分がプログラムである」という構造が非常にわかりにくくなっています。また、余計な空白を入れると文の意味が分かりにくくなり、誤解が生じかねません。そこで、下のように記述したらどうでしょうか？

字下げ

```
private void jButtonAddActionPerformed(ActionEvent evt) {
    jTextField1.setText("abc");
    jTextField2.setText("def");
    jTextFieldResult.setText( jTextField1.getText()
                              +jTextField2.getText() );
}
```

かなりすっきりするでしょう？ここで特に注意して欲しいことは、**{　}** の間の行は先頭に空白を入れて先頭を揃えプログラムをみやすくする、という点です。これを「**字下げ**」といいます。字下げは、プログラムの体裁上のことなのでどうでもいいことの様に思うかもしれません。しかし、論理構造を視覚的に分かりやすく記述することは、ミスを減らす、あるいは作業効率を上げるという観点からも重要なことです。皆もきちんと字下げをするようにして下さい。Eclipseのエディタでは、Tabキーを押すことで適切な字下げがなされるようになっています。

（3）Java 言語はプログラムの中に半角の「 /*　」があると、そこから半角の「　*/　」までを無視します。このことを利用して、プログラムに自由に**コメント**をつけることができます。また、１行のみのコメントの場合は「//」を利用します。コメントには全角文字を使っても構いません。プログラムをわかりやすくするため、コメントを有効に使うようにしましょう。

```
private void jButtonAddActionPerformed(ActionEvent evt) {
  //２数の和を求めるプログラム
  jTextFieldResult.setText(
      jTextField1.getText()  //１番上のテキストフィールドの値
    + jTextField2.getText() /*２番目のテキストフィールドの値*/
  );
}
```

（4）Java 言語では大文字と小文字を区別します。ですから「jTextField」と「jTEXTField」や「jtextfield」などは、全て異なるものとして認識されます。このため、最初の内は、「jTextField」のつもりで、「jTextfield」などと記述してエラーになることがよくあります。注意してください。

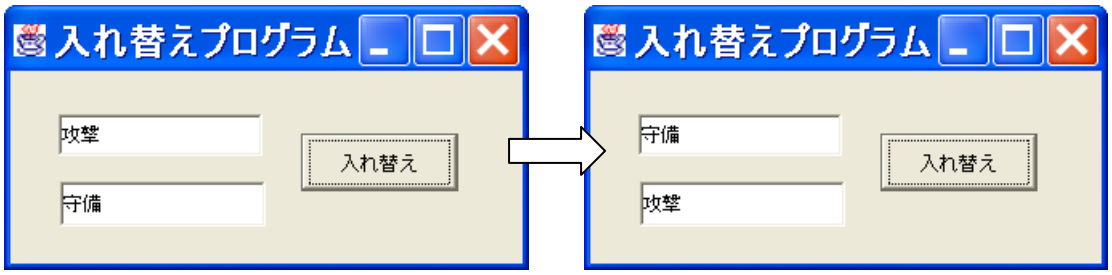
4-5 変数（1）―変数としての text プロパティ―

【基礎課題 4-5-1】

次のようなプログラムを作成しましょう。

プログラムを起動し、上下二つのテキストフィールドに適当な文字を入力する。

［入れ替え］ボタンをクリックすると、上下のテキストフィールドに入っていた文字が入れ替わる。



まず、上のようなフレームを作ってください。

貼り付けたコンポーネントの name プロパティは次の通りとします。

コンポーネント	name
上のテキストフィールド	jTextField1
下のテキストフィールド	jTextField2
ボタン	jButtonSwap

そして［入れ替え］ボタンをクリックした時のプログラムを下のように記述してください。

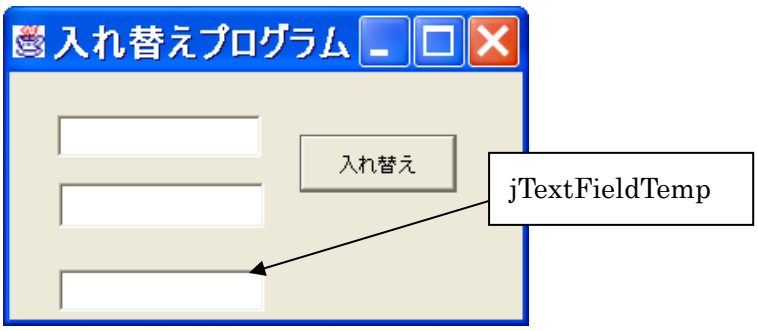
```
private void jButtonSwapActionPerformed(ActionEvent evt) {
  jTextField1.setText(jTextField2.getText()); //下欄→上欄への代入
  jTextField2.setText(jTextField1.getText()); //上欄→下欄への代入
}
```

上下のテキストフィールドに適当な言葉を入れて実行してみましょう。うまく動きましたか?

実はこのままではうまく入れ替えができませんね。それは何がまずいのでしょうか？また、どうしたらいいでしょうか。次ページに進む前に自分なりに考えてみて下さい。

さて、問題を解決しましょう。鋭い人は気づいたと思いますが、うまく入れ替えを行うためには、「上欄の値←下欄の値」と入力する前に、上欄の値（jTextField1.getText()）をどこかに保持しておく必要があります。

そのための”第三の”テキストフィールドを用意しましょう。次のように、データの中継用のテキストフィールドを加えてください。ここに、name プロパティの Temp とは「temporary（一時的）」から取っています。

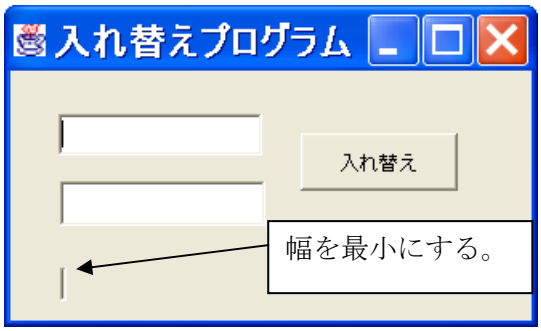


正しく入れ替えができるように、以下の空欄を埋めてプログラムを完成させ、動作を確かめてください。

```
private void jButtonSwapActionPerformed(ActionEvent evt) {
  jTextFieldTemp.setText(jTextField1.getText()); //上欄→Temp へ退避
  jTextField1.setText(jTextField2.getText());    //下欄→上欄への代入

                                                  ; //Temp→下欄への代入
}
```

【練習問題】



このままでもプログラムは正しく動きますが、中継用テキストフィールドが画面に表示されるのは格好悪いですね。そこで、次のように、この欄の幅を小さく、いっそのこと 0 にしてみましょう。これでも正しく動くでしょうか？予想を立ててから、実行して確かめてみましょう。

予想

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＿＿＿＿＿＿＿＿

4-6 変数（2）―宣言して使う変数―

前節で確かめた通り、中継欄jTextFieldTempは見えなくても問題なく動作します。入れ替えに必要なのは、text プロパティだけなのです。では、中継欄を使わない、もっとよい方法はないのでしょうか。

ここで、新しい方法を試してみます。まずは、jTextFieldTemp を削除してください。この時点で、プログラム中の jTextFieldTemp を参照している部分（２ヶ所）で「jTextFieldTemp を解決できません」というエラーが発生します。それは次ページの【基礎課題 4-6-1】で解消しますので、それまで気にしないで以下を読み進めて下さい。

ともかく、jTextFieldTemp を削除したので、それを中継欄に使うことはできなくなりました。そこで、今消した jTextFieldTemp のかわりにもっと単純な中継欄を作ります。次のように（前節のプログラムに）波線部を付け加えてください。

変数の宣言

```
private void jButtonSwapActionPerformed(ActionEvent evt) {
  String Temp;
  jTextFieldTemp.setText(jTextField1.getText()); //上欄→Temp へ退避
  jTextField1.setText(jTextField2.getText());    //下欄→上欄への代入
  jTextField2.setText(jTextFieldTemp.getText()); //Temp→下欄への代入
}
```

ここで Temp という入れ物ができました。そしてこの Temp に jTextFieldTemp.getText() のかわりをさせよう、というのがこれから行うことです。ところで、Temp のように値を保存しておく入れ物を「**変数**」といいます。（入れ物への）入力によって、その値が様々に変わるからです。今の場合 Temp は「テキストフィールドコンポーネントのtextプロパティ」と考えて結構です。

※　変数の名前は Temp でなくとも自由につけても構いませんが、なるべく一目見て内容がわかる名前をつけましょう。

jTextFieldTemp.getText()は String 型だったので、そのかわりとなる Temp も String 型でなくてはなりません。そのため「String Temp;」と書きました。（もし **int** 型が必要なら「**int** Temp;」と書きます。）このように、プログラム中で用いる変数の変数名およびその型を指定することを、「**変数の宣言**」と呼びます。

これで第一段階の準備はできましたが、まだ完成ではありません。次の基礎課題に進みましょう。

【基礎課題 4-6-1】

Temp を使って（ jTextFieldTemp.getText()を使わないで）プログラムを完成させると、次のようになります。

```
private void jButtonSwapActionPerformed(ActionEvent evt) {
  String Temp;
  Temp = jTextField1.getText();   //上欄→Temp へ退避（代入）
  jTextField1.setText(jTextField2.getText());  //下欄→上欄への代入
  jTextField2.setText(Temp);   //Temp→下欄への代入
}
```

このプログラムを理解するポイントは次の３点です。

① text プロパティは String 型の変数と同等である。

② 変数 b の値を変数 a に入力するには、次のように**代入(＝)**を用いる。

　　a ＝ b ；

一見すると、数学の等式のように見えるが、プログラミングでは、「右辺を左辺に**代入**する」という意味である点に注意。

③ テキストフィールドのようなオブジェクトの場合は、setText()のように所定のプロパティに値を代入するメソッドが備わっていたが、String 型変数である Temp にはそのメソッドがない。したがって、**代入(＝)**を用いる。

※　実は、setText()メソッドの中で実際に行われている処理は、「（　）内の文字列を text プロパティに**代入する**」という処理に他なりません。ですから、その中では代入文を用いているのです。

プログラムを作成したら実行し、動作を確認して下さい。

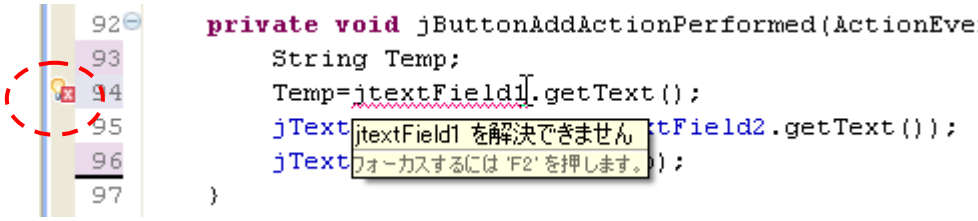
補足　エラーメッセージについて

プログラミング学習の初めの頃は、スペルミスなどのうっかりミスによるエラーが多発するものです。そのようなエラーを解決するためには、エラーメッセージを理解する必要があります。そこで、ここでは典型的なエラーを題材にして、Eclipse が示してくれるエラーメッセージに少し慣れておくことにしましょう。

① 変数名のスペルミスの場合

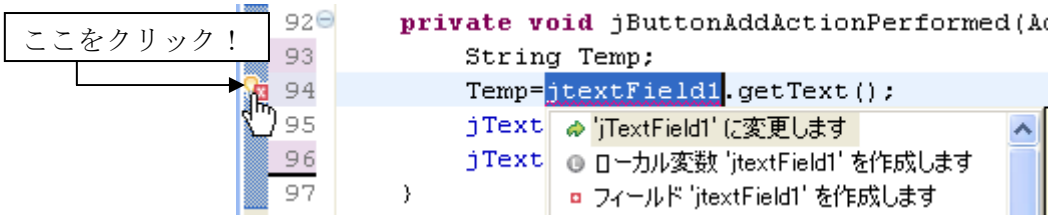
これは、最も多いミスです。例えば、上のプログラムで「jTextField1」とするべきとこ

ろを、誤って「jtextField1」とした、等の場合です。実際にそういう風に記述してみて下さい。すると、次のようにエラーがあると思われる行にエラーアイコンが表示されます。



さらにエラー候補となる部分が赤の下線で表示されます。今の場合、jtextField1 の部分ですね。そこで、ここにカーソルを合わせると、上の様にエラーメッセージが表れるようになっています。「jtextField1 を解決できません」が今の場合のメッセージ内容です。ここに“解決できない”という表現は文法用語で、要するに“そのようなものは見つからない”という意味です。実際に存在しているのは「jTextField1」であり、「jtextField1」という名前のコンポーネントは存在しないので、このようなエラーが出るのは当然ですね。これまでの経験から、演習中に学生が引き起こすエラーの大半は、このタイプのエラー（スペルミス）です。そして、何度このエラーが出ても自分で修正できない学生は結局プログラミングも上達しません。みなは、このエラーを見たらすぐに修正できるようになって欲しいと思います。

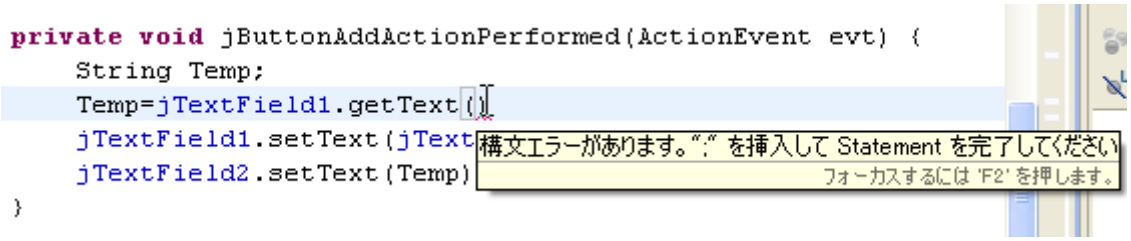
さて、“いいこと”を教えましょう。上のエラーアイコンの電球部分をクリックすると、エラーの修正候補を表示してくれる場合があります。実際、今の場合クリックしてみると下のように修正候補の先頭に「jTextField1 に変更します」という項目が現れます。



そこで、このメニューを選ぶと自動的に修正してくれます。どうです？Eclipse の支援機能は役に立つでしょう？

② ;のつけ忘れ

一つの文の区切りには「；」をつけなければならないのですが、最初はこのつけ忘れが多いようです。このときは、下のように、やはり、「；」が抜けている場所が赤の下線で表示され、上の①と同じくそこにカーソルを合わせると、「構文エラーがあります。”;”を挿入して Statement を完了してください」というエラーメッセージが表示されます。



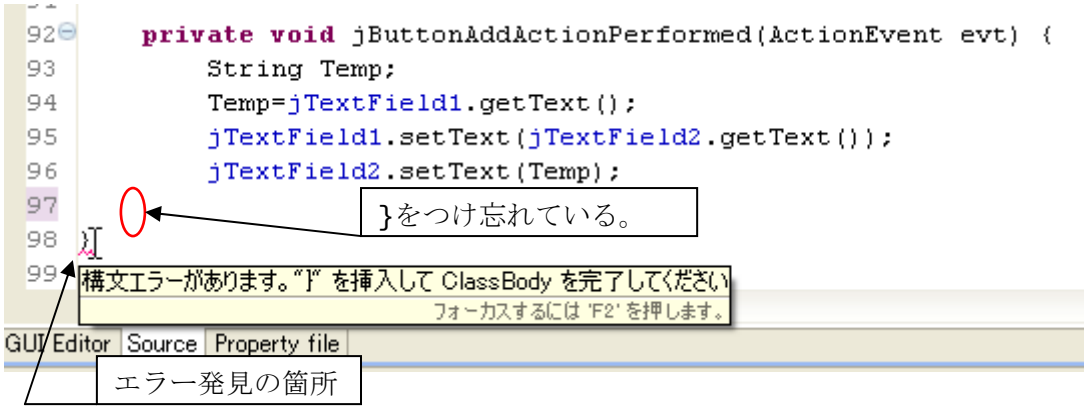
この場合は、エラーの修正の仕方を指示しているので、エラーの意味およびそれへの対応はすぐに分かると思います。

このように、Eclipse が示してくれるエラーメッセージを落ち着いて読めば、エラーの内容は大体分かります。エラーが出るとパニックに陥り、すぐに人に頼る人がいますが、それではいつまでたっても上達しません。プログラミングの上達は、自力でエラーを解決できるかどうかに強く関わっています。

③ { }が対応していない

Java 言語では、あるイベントハンドラなどのように、ひとまとまりの処理は { } でくくられます。{ } で囲まれた部分を**ブロック**と言います。当然のことですが、ブロック開始のかっこ { と終了のかっこ } の数は同じでなければなりません。そうでなければ、ブロックが閉じていない事になります。

やはり最初の内はこのかっこを書き忘れる、あるいは不用意に消してしまう、と言うミスが多いようです。この場合も②のケースと同様のエラーメッセージは出ますが、Eclipse が指摘する箇所は、必ずしも、エラーの箇所ではありません。例えば次の例をみて下さい。



この場合、エラー指摘の箇所は、}をつけ忘れた行ではなく、プログラムの最終行になっています。これは、{ の数と、 } の数が合致しない事が確定するのはプログラムの終端に到達してから、であるためです。ですから、この場合、エラー発見の箇所の上のどこかに } が抜けている箇所がある、という意味になります。

{ } については、自分が記述したブロックが閉じているかどうかを常に気にするようにすれば、ほとんどエラーは起きなくなります。4-4 節の(2)のところで字下げを励行したのは、実はこのブロックの閉じ忘れを防ぐことが目的だったのです。

コラム　変数名について

変数の名前は原則として自由につけて構わないのですが、いくつか例外があります。

- Java 言語では、漢字やひらがなも変数名として用いることができます。しかし、実際には全角文字と半角文字の切り替えや変換ミスによる余分なエラーを引き起こす可能性があるので、英数字（および「_」）だけで作られる名前にした方が良いでしょう。なお、途中に空白を含む名前はエラーになります。
- 数字で始まる名前はエラーになります。
- 「class」や「int」など、特別な意味を持つ名前（予約語と呼ばれます）を使ってはいけません。

コラム　変数の初期化

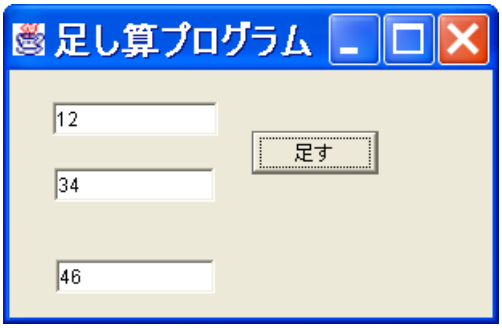
Java 言語では、次のように「変数の宣言」と、「変数の初期化（初期値の代入）」を同時に行うことができます。

```
private void jButtonSwapActionPerformed(ActionEvent evt) {
  String Temp=jTextField1.getText(); //変数 Temp の宣言と初期化
  jTextField1.setText(jTextField2.getText()); //下欄→上欄への代入
  jTextField2.setText(Temp); //Temp→下欄への代入
}
```

Java 言語のプログラムでは、上のように同時に初期化を行う場合が多いようです。

【基礎課題 4-6-2】

【基礎課題 4-3-1】で作った、整数の足し算プログラムを、変数を用いて書き直しましょう。



下線部を埋めてプログラムを完成させて下さい。

```
private void jButtonAddActionPerformed(ActionEvent evt) {
  int     Data1; //整数型変数（上欄用）の宣言
  ______  Data2; //整数型変数（下欄用）の宣言
  ______  Sum;   //整数型変数（和の保管用）の宣言
  Data1=Integer.parseInt(jTextField1.getText());
  Data2=Integer.parseInt(jTextField2.getText());
  Sum= Data1   +  Data2  ;
  jTextFieldResult.setText( ____________________________ );
}
```

今までどおり「足し算プログラム」として動くことを確認しましょう。

なお、変数の宣言では、

```
int  a;
int  b;          をまとめて      int a, b, c;
int  c;
```

```
String a;
String b;        をまとめて      String  a, b, c;
String c;
```

のように複数の変数をひとまとめにして宣言することができます。

コラム　変数を使った方がいいの？

「【基礎課題 4-3-1】で作ったプログラムとここで作ったプログラムと、どちらも同じく動作するけど、どちらがいいの?」という疑問を持った人がいるかもしれませんね。実は、それを実行するコンピュータから見れば、両者のプログラムは同等です。その意味では、どちらでも良いと言えます。

しかし、それを読む人間にとってはどうでしょうか？後から読み返してみると、一つの式に一度にまとめて記述するよりも、前ページの様に変数を用いた方が分かりやすいのではないでしょうか。一般に、入力データや、計算の答など、何らかの意味を持つ値については、いったん、変数を定義してそこに代入しておいた方が分かりやすく、また後に拡張等を行う際にもより容易になります。その意味では、なるべく変数を使った方が良いと言えます。

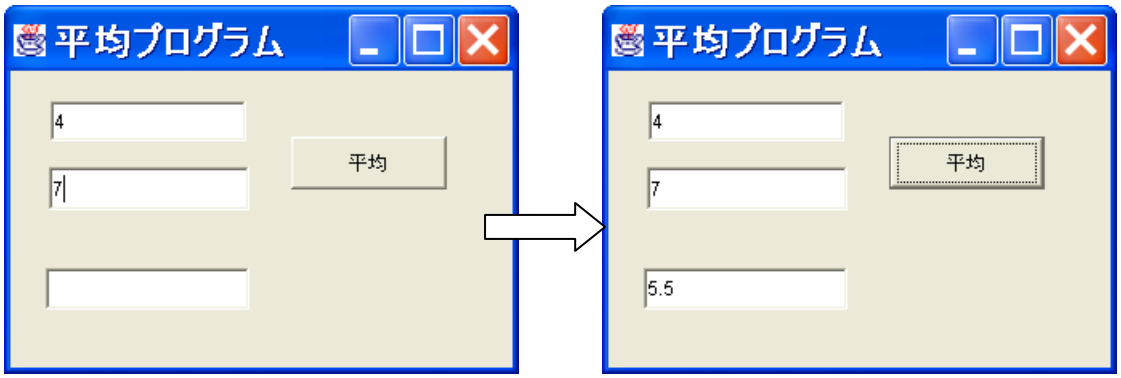
4-7 実数型変数

【基礎課題 4-7-1】

次のようなプログラムを作成しましょう。

上から２つのテキストフィールドに適当な整数を入力する。

［平均］ボタンをクリックすると、２数の平均値が最下欄に表示される。



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
上のテキストフィールド	jTextField1
中央のテキストフィールド	jTextField2
下のテキストフィールド	jTextFieldResult
ボタン	jButtonAvg

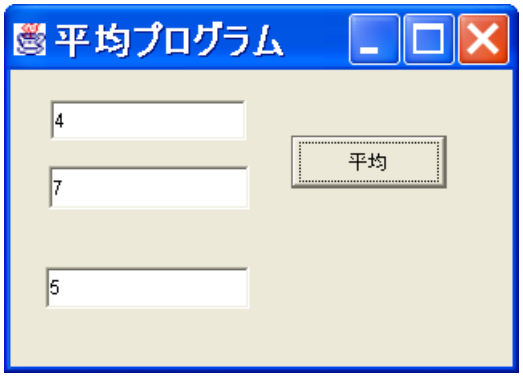
次に、イベントハンドラを作成しましょう。2 つの数の平均は、合計を 2 で割って求めるのですね。プログラムを次のように記述してください。

```
private void jButtonAvgActionPerformed(ActionEvent evt) {
  int Data1,Data2,Avg;  //変数の宣言   Avg は Average（平均）の略
  Data1=Integer.parseInt(jTextField1.getText());  //上の値の入力
  Data2=Integer.parseInt(jTextField2.getText());  //下の値の入力
  Avg=(Data1+Data2)/2;
  jTextFieldResult.setText(String.valueOf(Avg));
}
```

※　プログラミング言語では除算（割り算）は「／」で表します。

「ａ÷ｂ」　→　「ａ／ｂ」

実行してみましょう。結果は次のようになるはずです。



答えの小数点以下が切り捨てられていますね。これは、平均を代入する変数Avgがint型（整数型）であることが原因です。整数には、小数点以下が代入されず、切り捨てられてしまうのです。そこで、「**実数型**」の変数が必要になります。次の基礎課題に進みましょう。

【基礎課題 4-7-2】

Avg を整数型の変数ではなく小数点以下を受け付ける型の変数にしてしまいましょう。小数を受け付ける型は**実数型**と呼ばれ、「**double**」と表します。プログラムを以下のように修正してください（波線部が修正箇所です）。

```
private void jButtonAvgActionPerformed(ActionEvent evt) {
  int Data1,Data2;
  double Avg;  //実数型の宣言
  Data1=Integer.parseInt(jTextField1.getText());
  Data2=Integer.parseInt(jTextField2.getText());
  Avg=(Data1+Data2)/2.0;
  jTextFieldResult.setText(String.valueOf(Avg));
}
```

なお、String.valueOf()メソッドは、（ ）内が実数でも（整数の場合と同様に）文字列型に変換されます。
実行してみましょう。今度はきちんと小数点以下も表示されるはずです。

コラム　分母を 2.0 にしたのはなぜ？

平均 Avg を求めるところでなぜ分母を「2」ではなく、「2.0」に変更したのか、不思議に思うかもしれません。実は、「Avg=(Data1+Data2)/2」のままでは、たとえ、変数 Avg を実数型にしても小数点以下は切り捨てられます（各自試してみてください）。
というのは、Java言語では「整数／整数」の演算結果は整数にする、という規則があるからです。今の場合、分子のData1、Data2 および分母の 2 が全て整数であるため、このような問題が起こります。これを避けるためには、分母か分子の（少なくとも）どちらか一方を実数にする必要があります。そこで、分母の整数「2」を実数「2.0」に変更した訳です。

型変換の命令は、次のものがあります。必要になったらこのページを参照してください。

整数型から文字列型へ	`String.valueOf()`
文字列型から整数型へ	`Integer.parseInt()`
実数型から文字列型へ	`String.valueOf()`
文字列型から実数型へ	`Double.parseDouble()`
整数型から実数型へ	命令がなくても自動的に変換される
実数型から整数型へ	型キャストを用いる。以下の解説参照

＜実数型から整数型への変換＞

例えば、次のプログラムを実行すると、３行目のａの値は「１」となります。

```
1   int a;
2   double b=1.5;
3   a=(int) (b);
```

つまり、実数型変数ｂの値の小数点以下が切り捨てられてaに代入されている訳です。

Java 言語では、一般に「**(変数型名)（式)**」という形で、式の値を指定した型に変換できます。これを**型キャスト**と呼びます。Java 言語では、整数型から実数型への変換は自動的に行われますが、逆の「実数型→整数型」の場合は、小数点以下の切り捨てなど”情報の消失”を伴うので（うっかりミスを防ぐため）認めていません。そこで、そのような場合には上のように**型キャスト**を用いる必要があります。

＜様々な演算子＞

以下に Java 言語で通常よく用いられる演算子をまとめておきます。

Ⅰ. 算術演算子

演算	加算	減算	乗算	除算	剰余
表記	＋	－	＊	／	％

※　除算の場合、分母、分子が共に整数であれば答は整数（少数点以下切り捨て)、それ以外は実数。

※　剰余（余り）計算の演算結果（の例）は次の通り。

11 % 3 → 2　　11 % 4 → 3

Ⅱ. 代入演算子

プログラムを記述する場合は、「a = a + b」のようなタイプの式がよく出てきます。この入力の手間を省くため、以下の代入演算子が用意されています。

演算子	使用例	式の意味
+=	a += b	a = a + b
-=	a -= b	a = a – b
*=	a *= b	a = a * b
/=	a /= b	a = a / b
%=	a %= b	a = a % b

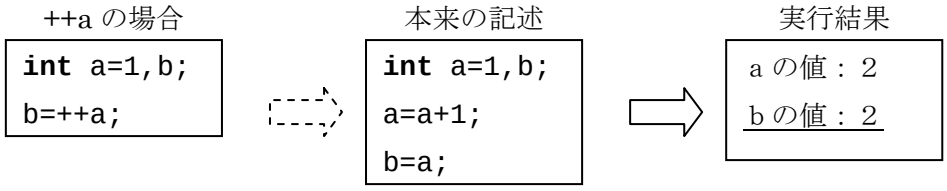
※ 「a += b」などの場合、「a + = b」のように、二つの演算子の間に空白を空けてはいけません。

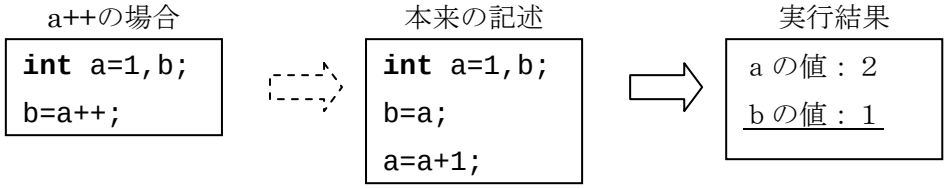
Ⅲ. 変数の値を一つ増やす、あるいは一つ減らす演算子

演算子	使用例	式の意味
++	a++ あるいは ++a	a = a + 1
--	a-- あるいは --a	a = a - 1

注意

「a++」と「++a」を式の中で用いた場合、演算子の優先順位の関係で式の値が異なる場合があります。次の例を見てください。省略しない**本来の記述**を見れば違いが分かると思います。

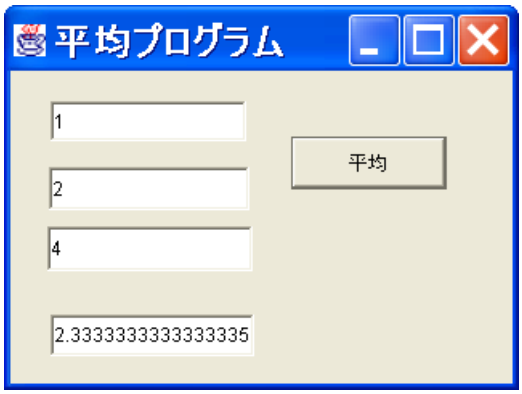




★ ⅡやⅢは入力の手間を省くために用意された演算子であり、これを知らなくてもプログラミングはできます。しかし、実際のプログラミングでは、本来の書式よりも、ⅡやⅢのような省略した演算子が多用される傾向があります。

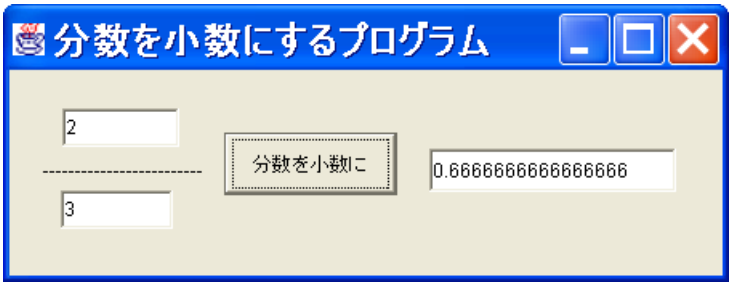
【基礎課題 4-7-3】

3 つの数の平均を求めるプログラムを作ってください。



【応用課題 4-7-A】

下のように、分数を小数にするプログラムを作ってください。下は、分子に「２」、分母に「３」を入力してボタンをクリックした例です。



ヒント

【基礎課題 4-7-2】で注意したように、「整数／整数」の答えは、小数点以下が切り捨てられてしまいます。→それを避けるにはどうしたらよいでしょうか？

⇨ 少なくとも分子か分母の値のいずれかを実数型の変数に代入すればよいですね。

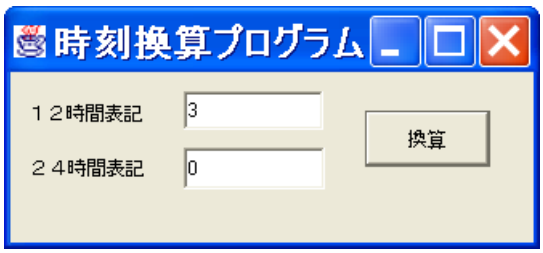
注意

実行後、分母の欄を空欄あるいは 0 にしたままの状態で、ボタンを押さないでください。分母が 0 の分数は計算できないので、エラーが発生します。

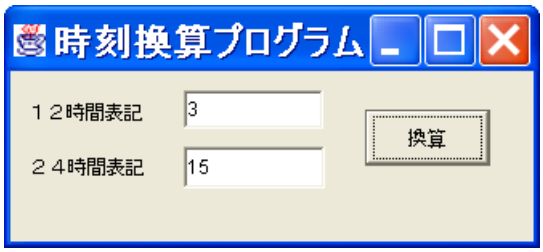
4-8 定数（1）―整数型定数―

【基礎課題 4-8-1】

今は午後 3 時だとします。



12 時間表記欄に時刻を入力し「換算」ボタンを押したら（最初は、両欄とも 0 が表示されているものとします）



午後の時刻の 12 時間表記から 24 時間表記に変換して表示する、というプログラムを作ってみましょう。

下の下線部には何と書いたらいいでしょうか。

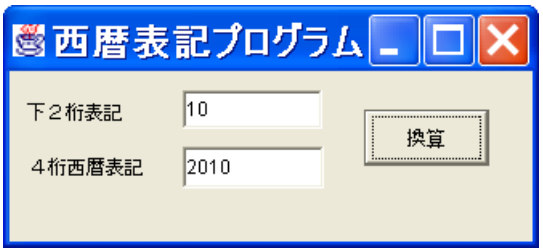
```
private void jButtonKansanActionPerformed(ActionEvent evt) {
  int Time12,Time24;
  Time12=Integer.parseInt(jTextField1.getText());
  Time24=Time12+_____;
  jTextFieldResult.setText(String.valueOf(Time24));
}
```

※　コンポーネントの name プロパティを次のようにしています。

上のテキストフィールド	jTextField1
下のテキストフィールド	jTextFieldResult
ボタン	jButtonKansan

【基礎課題 4-8-2】

今度は、下 2 桁の西暦略記から 4 桁の西暦表記を求めるプログラムを作ってください。ただし、西暦 2000 年以降とします。

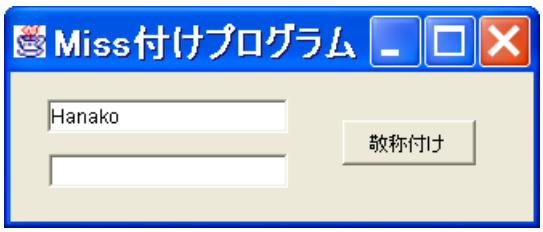


12 や 2000 のように、いつも決まった値であるものを「定数」といいます。この定数とは対照的に、変数は、代入によってその内容（値）が変わります。

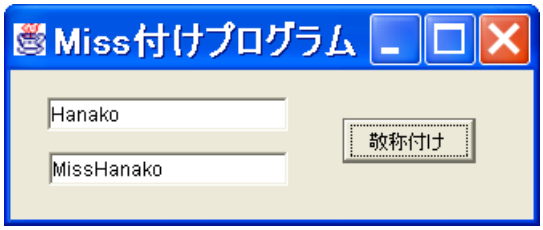
4-9 定数（2）―文字列型定数―

【基礎課題 4-9-1】

次のプログラムを作ってください。



左上の欄に女性の名前を入れてボタンを押すと



頭に「Miss」とつけるプログラムを作ります。

※　実は Miss や Mr.などの呼称は名字（姓）につけるもので、名前にこれをつけるのは英語としては不自然なのですが、ここでは気にしないでください。

各コンポーネントの name プロパティは次の通りとします。

コンポーネント	name
上のテキストフィールド	jTextField1
下のテキストフィールド	jTextField2
ボタン	jButtonMiss

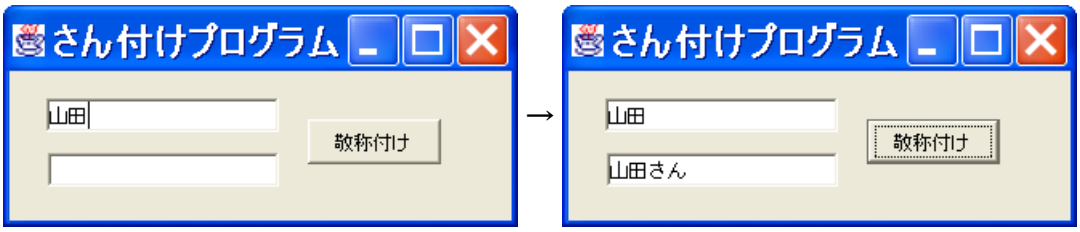
このプログラムのイベントハンドラは次のようになります。下線部を埋めてプログラムを完成させて下さい。

```
private void jButtonMissActionPerformed(ActionEvent evt) {
  String Name,MissName;
  Name=jTextField1.getText(); //名前の代入
  MissName= _____ + Name; //Miss 付け
  jTextField2.setText(MissName);
}
```

ヒント　3-1 節で、「" "で囲んだ部分は文字列とみなされる」と学習しました。ですから、表示したい文字を" "で囲めばよいのです。" "で囲まれた文字は、実は**文字列型定数**だったのです。

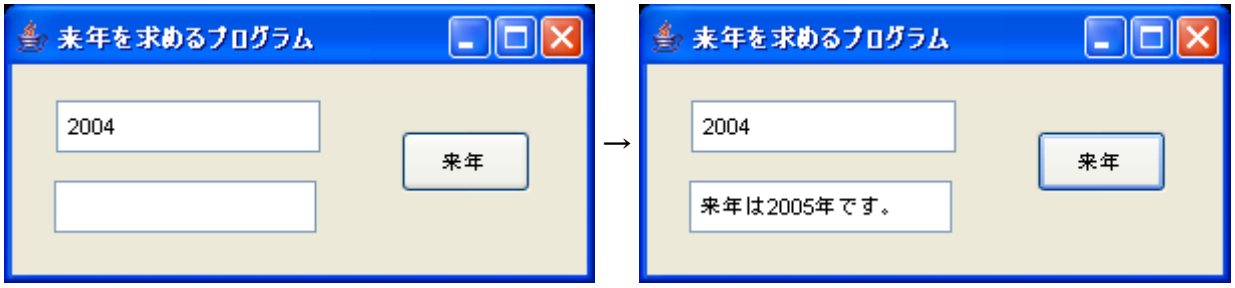
【基礎課題 4-9-2】

次のように「さん」づけするプログラムを作りましょう。



【応用課題 4-9-A】

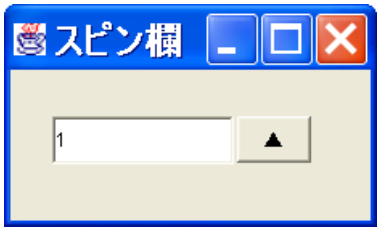
次のように来年が何年か教えてくれるプログラムを作りましょう。



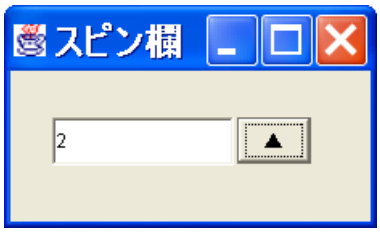
※　整数の足し算と文字列の連結の両方が含まれます。

4-10 変数と定数 ―整数加減欄を作ろう―

【基礎課題 4-10-1】



整数を入力して「▲」ボタンを押すと



整数の値が１つ増える。もう１回ボタンを押すと、さらに１増える・・、という具合にボタンを押す毎に値が１つずつ増えるプログラムを作ってみましょう。

※　▲は「さんかく」で変換して表示させて下さい。

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	jTextField1
ボタン	jButtonInc

jTextField1 のみでは数値の計算ができないので、**整数型変数** a を用意し、

(1) jTextField1 欄の値を a に代入
(2) a の値を 1 増やす
(3) a の値を jTextField1 に代入

の順に処理するようにしましょう。下線部を埋めてプログラムを完成させて下さい。

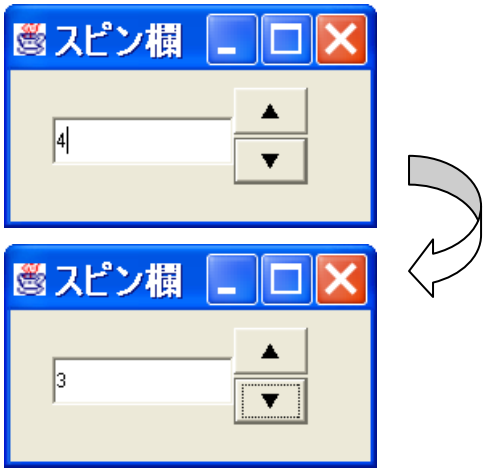
```
void jButtonIncActionPerformed(ActionEvent evt) {
  ______ a; //整数型変数 a の宣言
  a=Integer.parseInt(jTextField1.getText());
  a= ______ + 1;
  jTextField1.setText(String.valueOf(a));
}
```

実行してみましょう。

※　このプログラムがすんなりできたら大したもの！あなたは「代入」の意味をよく理解していることになります。

【基礎課題 4-10-2】

上のプログラムに、▼ボタンを加え、これをクリックすると、値が１つ減少するようにしてください。



▼ボタンをクリックすると､値が１つ減少する。

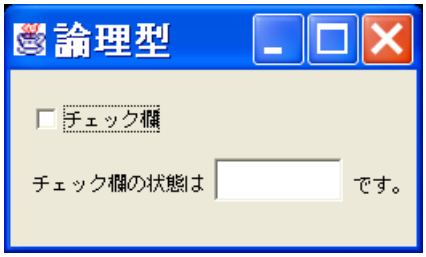
このような上下ボタンは、Windows アプリケーションを使っている時、使用したことがあると思います。

4-11　論理型変数

本節では論理型変数を学習します。これは、true（真）か false（偽）のいずれかの値のみを保管する変数です。各コンポーネントの **enabled** プロパティやチェックボックスコンポーネントの **selected** プロパティが、この論理型になります。次の課題で、チェックボックスコンポーネントの **selected** プロパティが論理型変数であることを確認しましょう。

【基礎課題 4-11-1】

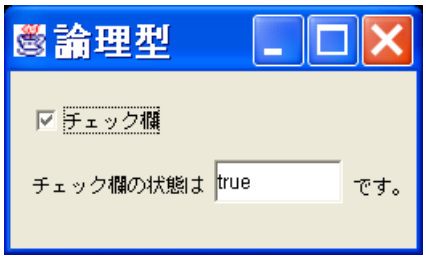
次のようなフレームを作って下さい。



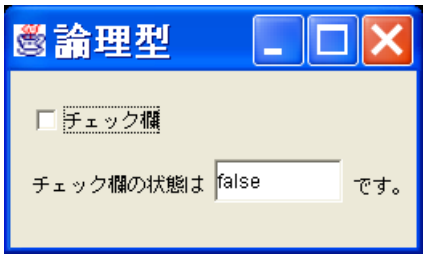
コンポーネントの name プロパティは下の通りとします。

コンポーネント	name
チェックボックス	jCheckBox1
テキストフィールド	jTextField1

作成するプログラムの動作内容は次の通りです。



チェック欄をチェックすると、テキストフィールドに「true」と表示される。



（もう一度チェックして）チェック欄のチェックを外すと、今度は「false」と表示される。

このプログラムはチェックボックスの **actionPerformed** イベントを用いています。チェックボックスのアクションは「欄をチェックすること」です。そこで、チェックボックスのイベントハンドラを次ページのように記述して下さい。

作成したら実行し、正しく動作することを確認して下さい。ここでは、**selected** プロパティが**論理型（boolean 型）変数**であることを理解できれば OK です。

```
1:    private void jCheckBox1ActionPerformed(ActionEvent evt) {
2:       boolean a; //論理型変数の宣言
3:       a=jCheckBox1.isSelected(); //チェック欄の状態の代入
4:       jTextField1.setText(String.valueOf(a));
5:    }
```

＜解説＞

① 論理型の変数は「**boolean** 変数名」の形で宣言します。

② 「チェックボックスがチェックされているかどうか」は **selected** プロパティに入っています。そこで、**text** プロパティの時のように、その値を確かめるには **getSelected()**というメソッドを用いれば良さそうな気がしますが、実はそうではありません。Java 言語では、プロパティの値が論理型（**boolean** 型）の場合は、その値を得るには **isSelected()**というメソッドを用います。少しややこしいですが、これはまぁそういう約束事だと思って下さい。

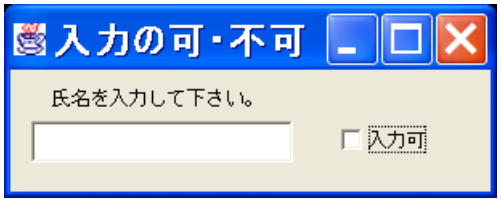
③ ３行目では、チェックボックスの **selected** プロパティの値を論理型変数 a に代入しています。変数 a にはチェック状態に応じて「**true**」あるいは「**false**」の値が代入されることになります。

④ ４行目で変数 a の値を（文字列に変換して）表示しています。なお、プログラムから分かるとおり、**String.valueOf()**メソッドは、引数が論理型変数でも OK です。このメソッドは引数がどのような型でも対応できるオールマイティのメソッドです。

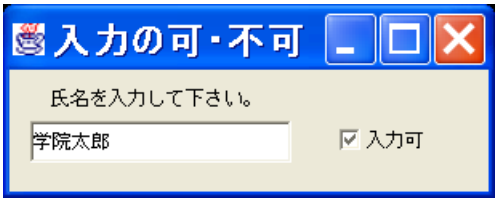
【応用課題 4-11-A】

次のようなプログラムを作って下さい。

起動すると下のような画面が現れる。
この状態では氏名を入力できない。



「入力可」欄をチェックすると、入力が可能になる。再びチェックを外すとまた入力不可の状態になる。



ヒント

① 入力の可・不可は、テキストフィールドの **enabled** プロパティで決まります。

② そのプロパティの値を変更するには **setEnabled()**メソッドを用います。

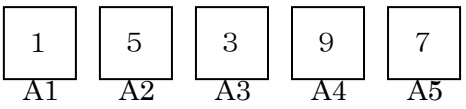
③ チェックボックスの **selected** プロパティとテキストフィールドの **enabled** プロパティの値を一致させれば良いのです。

4-12 配列

本章の最後に**配列**を学習します。配列とは、A[1]，A[2]，…のように、括弧内の数字（添え字）で変数の値を指定できる変数のことです。次の例を参照して下さい。

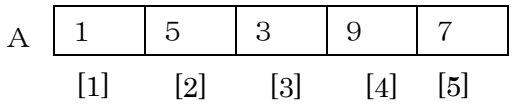
例　１，５，３，９，７という 5 個の整数を変数に保存する

★　一般の変数の場合：**5 個**の変数（A1，A2，…，A5）を用意する。



例えば、「9」は変数 **A4** に入っている。

★　配列の場合：大きさが「５」の配列変数Ａを**一個**用意する。



例えば、「9」は配列 **A[4]**に入っている。

コンピュータでは大量のデータを処理することが多いのですが、そのときに、個々のデータを（一般の）変数に保管するのは面倒です。そこで、添え字のみでデータを指定できる配列変数が必要になるのです。

Java 言語では、例えば 10 個の要素を持った整数型の配列 A を宣言する場合、次のように記述します。

```
int[] A;
A= new int[10];
```

あるいは上の２つをまとめて以下のように一度に書くこともできます。

```
int[] A = new int[10];
```

もし、実数型で要素数を 20 個持っている配列 B を宣言する場合は、同様に

```
double[] B = new double[20];
```

と宣言します。一般には

型名[] 変数名 = new 型名[要素数]；

の形になります。

何となく分かるとは思いますが、配列の宣言が単純に「int[10] A」等とはならずに、**new** 演算子を用いている点が少し気になりますね。その点については、第 7 章のクラスに

関する説明の部分で解説することにします。ここでは、上の記述方法を”約束事”だと了解しておいて下さい。ただ、「Java 言語では配列の扱いは通常の（整数型や実数型などの）変数と異なるらしい」という点は心に留めておきましょう。

補足

Java 言語の配列宣言については、次のような記述の仕方も認められています（[]が変数名の方に移動しています）。

```
int A[] = new int[10];
```

どちらを使っても良いですが、Java 言語を開発したサン・マイクロシステムズ社の開発チームは、前ページで説明した記述の方を推奨しているので、ここでは最初に説明したように、

```
int[] A = new int[10];
```

の記述を用いることにします。

配列の扱いについて、ごく基本的な事項のみを以下にまとめておきますので確認しておいて下さい。

＜配列要素の扱いに関する基本事項＞

① 配列要素は 0 番目から始まります。

```
int A[] = new int[10];
```

と宣言した場合、配列要素としてA[0]～A[9]（の 10 個の要素）が確保されます。最初の要素番号（**添え字**）が0 から始まることに注意して下さい。

② 各要素の参照や値の代入

配列の各要素、例えば A[1]などは、通常の変数と全く同様に扱えます。例えば、次の処理が実行されると・・・

```
int A[] = new int[3];
int B;
A[0]=10;  A[1]=3;  A[2]=8;   //各配列要素へ値を代入
B=A[2];    //３番目の配列要素の値を代入
```

A[0]～A[2]の値はそれぞれ 10,3,8 になり、B の値は 8 になります。

③ 配列要素の初期化

次のような記述により、配列の宣言と初期化を同時に行うことができます。

```
int c[]={1,2,3,4,5};
```

この例では、c[0]～c[4]の値がそれぞれ、1,2,…,5 になります。配列の大きさ（要素数）

は初期値の数で自動的に設定されます（上の場合は５）。

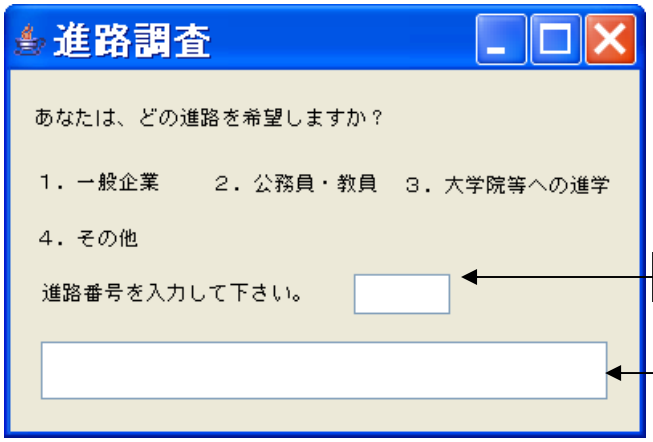
④　範囲外の要素へのアクセスの禁止

宣言された範囲外の添え字を指定した場合は、プログラム実行時にエラーとなります。例えば上の③の配列 c の場合、c[5]にアクセスしようとした場合、これは最大の添え字を越えていますからエラーになります。同様に、0 より小さい添え字を指定した場合もエラーとなります。

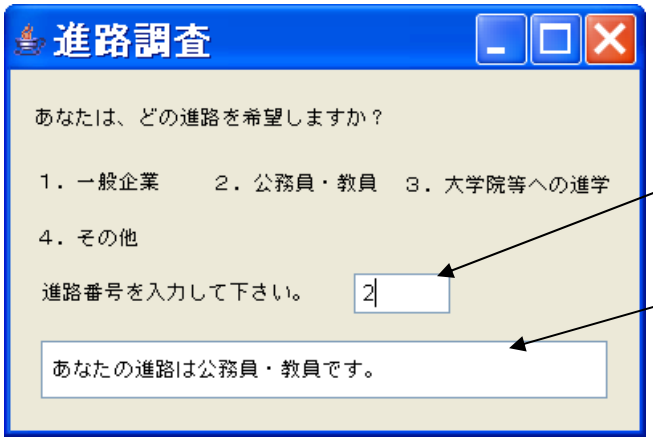
以上、配列に関する基礎事項を説明しましたが、恐らく説明だけではピンと来ないでしょう。またその必要性もよく分からないと思いますので、次の課題で実際に配列変数を応用してみましょう。なお、配列の本格的な応用については「データ構造とアルゴリズム論」で学習します。

【基礎課題 4-12-1】

次のような、進路希望調査を模したプログラムを作ってみます。



プログラムを起動すると、左の画面が現れます。



ここで、自分の希望進路に該当する番号を、入力欄に入力し［Enter］キーを押します。
すると、下のメッセージ欄に、選択した進路番号に応じて確認メッセージが表示されます。

このプログラムを自力で記述できたら、大したものです。まずは自分で考えてみて、その後以下の＜プログラム作成のポイント＞を参照して下さい。

＜プログラム作成のポイント＞

① この処理は、進路番号の入力欄である jTextFieldInput の **actionPerformed** イベントを用います。

② 記述するプログラム（処理）は、入力した進路番号に応じて確認メッセージ（の進路部分）が変わるというものです。

③ その、”変わる部分”（進路）を配列変数で表現すればよいのです。

どうですか、分かるでしょうか・・・？

答（プログラム）は次の通りになります。実際に作成して動作を確認してみて下さい。その後、下の問に答えて下さい。

＜jTextFieldInput のイベントハンドラ＞

```
private void jTextFieldInputActionPerformed(ActionEvent evt) {
  String[] Sinro=new String[4];  //文字列型配列の宣言・生成      ①
  // 配列 Sinro への値の入力
  Sinro[0]="一般企業";
  Sinro[1]="公務員・教員";                                        ②
  Sinro[2]="大学院等への進学";
  Sinro[3]="その他";
  // 進路番号を整数型変数へ入力
  int No=Integer.parseInt(jTextFieldInput.getText());      ③
  // メッセージの出力
  jTextFieldMessage.setText("あなたの進路は"+Sinro[No-1]+"です。");④
}
```

＜解説＞

① ４つの”進路”を保管するために、要素数４の文字列型配列変数 Sinro を用意します。

② ４つの進路をそれぞれ、配列 Sinro に代入します。

③ 希望進路の番号を、整数型変数 No に代入します。

④ 確認メッセージを表示します。”一般企業”等、各進路番号に対応した進路は Sinro[No-1]で表されます。

問

上の④において Sinro[No-1]のように要素番号が No ではなく、No-1 となっているのはなぜですか？担当の補助員に説明して下さい。