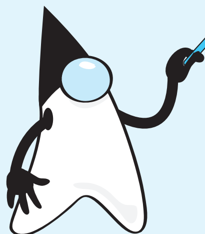
第 2 章

変数を使おう

本章では、数値や文字列を格納するための変数を学習します。変数に対して演算を施したり、キーボードから値を読み込んだりするプログラムを作成します。

- □型
- □変数と final 変数
- □整数と浮動小数点数
- □文字列と String 型
- □文字列と数値の連結
- □初期化と代入
- □演算子とオペランド
- □キーボードからの読込み
- □乱数の生成



2-1 変数

本章では、足し算や掛け算などの計算を行って、その結果を表示するプログラムを作っていきます。その手始めとなる本節では、計算結果の格納のために必要な《変数》について学習します。

演算結果の出力

ちょっとした計算を行って、その結果を表示するプログラムを作りましょう。**List 2-1** に示すのは、二つの整数値 57 と 32 の和を求めて表示するプログラムです。

List 2-1 Chap02/Sum1.java

```java
// 二つの整数値57と32の和を求めて表示

class Sum1 {

   public static void main(String[] args) {
      System.out.println(57 + 32);
   }
}
```

実行結果

```
89
```

数値の出力

System.out.println に続く ( ) 中の網かけ部に着目しましょう。前章のプログラムでは、この部分は文字列でした。本プログラムでは、数値を加算する式となっています。

いうまでもなく、57 **+** 32 の演算結果は 89 です。そのため、みなさんの期待どおり、整数値 89 が表示されます（その後に改行文字も出力されます）。

▶ **System**.out.print と **System**.out.println のメソッドが、文字列だけでなく整数値の表示を行えるのは、第 7 章で学習する《**多重定義**》が行われているからです。なお、この他にも、実数・論理型（第 5 章）・クラス型（第 8 章）などの表示が行えます。

整数リテラル

57 や 32 のような整数を表す定数は、**整数リテラル**（*integer literal*）と呼ばれます。

以下の二つはまったく異なりますので、混同しないようにしましょう。

- 57 … 整数リテラル（57という 1 個の整数値）。
- "57" … 文字列リテラル（2 個の文字 5 と 7 の並び）。

▶ 整数リテラルの詳細は、第 5 章で学習します。

*

さて、プログラムを実行して 89 とだけ表示されても、何のことだか分かりません。

▶ ただ 89 と表示するのであれば、プログラムは以下のようにも実現できます。

```java
System.out.println(89);
```

文字列と数値の連結

何の計算を行っているのかを、式として表示するように改良しましょう。**List 2-2** に示すのが、そのプログラムです。実行すると『57 + 32 = 89』と表示されます。

List 2-2 Chap02/Sum2.java

```
// 二つの整数値57と32の和を求めて表示

class Sum2 {

    public static void main(String[] args) {
        System.out.println("57 + 32 = " + (57 + 32));
    }
}
```

実行結果

```
57 + 32 = 89
```

出力にいたるまでに、複数の処理が行われます。その過程を示したのが **Fig.2-1** です。処理の流れを追いながら理解しましょう。

1 まず ( ) で囲まれた 57 + 32 の演算が行われます。( ) で囲まれた演算が**優先的に行われる**のは、私たちの日常生活での計算と同じです。

重要 優先的に行いたい演算は ( ) で囲もう。

2 89 が文字列 "89" に変換されます。というのも、以下の規則があるからです。

重要 『文字列 + 数値』あるいは『数値 + 文字列』の演算では、数値が文字列に変換された上で連結が行われる。

3 文字列 "57 + 32 = " と "89" とが連結されて "57 + 32 = 89" になります。この文字列が画面に表示されます。

▶ 『文字列 + 文字列』によって文字列が連結されることは前章で学習しました（p.16）。

Fig.2-1 文字列連結の過程 (List 2–2)

式 57 + 32 を囲む ( ) を取り除いたらどうなるかを実験してみましょう。そのためのプログラムが、**List 2-3** です。実行すると、おかしな結果が表示されます。57 と 32 の和が、なんと『5732』になります。

List 2-3 Chap02/Sum3.java

```
// 二つの整数値57と32の和を求めて表示（誤り）

class Sum3 {

   public static void main(String[] args) {
      System.out.println("57 + 32 = " + 57 + 32);
   }
}
```

実行結果

```
57 + 32 = 5732
```

文字列の連結や数値の加算を行う + の演算は、**左側から順に**行われます。これは、日常生活での足し算と同じです（一般に、$a + b + c$ は $(a + b) + c$ とみなされます）。

そのため、本プログラムでは **Fig.2-2** のように連結が行われます。『5732』と表示されるのは、『57』と『32』が連続して出力されるからです。

```
System.out.println("57 + 32 = " + 57 + 32 );
```
① 整数値57が文字列"57"に変換される。
```
System.out.println("57 + 32 = " + "57" + 32 );
```
② "57 + 32 = "と"57"が連結される。
```
System.out.println( "57 + 32 = 57" + 32 );
```
③ 整数値32が文字列"32"に変換される。
```
System.out.println( "57 + 32 = 57" + "32");
```
④ "57 + 32 = 57"と"32"が連結される。
```
System.out.println( "57 + 32 = 5732" );
```

Fig.2-2 文字列連結の過程（List 2–3）

加算の式を ( ) で囲まない別のプログラム例を **List 2-4** に示します。まずは、プログラムを実行してみましょう。

List 2-4 Chap02/Sum4.java

```
// 二つの整数値57と32の和を求めて表示

class Sum4 {

   public static void main(String[] args) {
      System.out.println(57 + 32 + "は57と32の和です。");
   }
}
```

実行結果

```
89は57と32の和です。
```

加算の式を囲む ( ) がないものの、**Fig.2-3** に示すように、うまくいきます。左側から順に演算が行われると、期待どおりの結果を生み出す構造だからです。

▶ すべての演算が左側から行われるわけではありません。右側から行われる演算もあります。詳しくは3–3 節で学習します。

```
System.out.println( 57 + 32 + "は57と32の和です。");
System.out.println(    89   + "は57と32の和です。");
System.out.println(  "89"   + "は57と32の和です。");
System.out.println(  "89は57と32の和です。"      );
```

① 57 + 32の演算が行われる。
② 整数値89が文字列"89"に変換される。
③ "89"と"は57と32の…"が連結される。

Fig.2-3 文字列連結の過程（List 2–4）

もっとも、必要がないからといって、完全に ( ) を省略してしまうと、プログラムが読みにくくなってしまいます。たとえ冗長(じょうちょう)となっても、以下のように ( ) で囲んだほうがプログラムの見通しがよくなります。

```
System.out.println((57 + 32) + "は57と32の和です。");
```

( ) は多すぎても、少なすぎても、読みにくくなります。臨機応変(りんきおうへん)に対応しましょう。

Column 2-1 文字列の連結と減算

本文では加算の演算結果を表示するプログラムを考えました。ここで減算を考えてみましょう。**List 2-2** の出力部を以下のように書きかえて実行すると、『57 - 32 = 25』と表示されます。

```
System.out.println("57 - 32 = " + (57 - 32));
```

それでは、**List 2-3** と同様に ( ) を省略してみましょう。

```
System.out.println("57 - 32 = " + 57 - 32);       // エラー
```

このプログラムは正しくありません。コンパイル時にエラーとなります。その理由は、以下のとおりです。

- まず最初に左側の演算 "57 - 32 = " + 57 が行われます。これは『文字列 + 数値』ですから、57 が文字列 "57" に変換された上で連結されます。演算結果は、文字列 "57 - 32 = 57" です。
- 続いて右側の演算 "57 - 32 = 57" - 32 が行われます。これは『文字列 - 数値』です。文字列から数値を引くことはできません。したがって、コンパイル時にエラーとなるのです。

変数

これまでのプログラムは、57 と 32 以外の数値の和を求めることができません。数値を変更する際は、プログラムに手を加える必要があります。もちろん、プログラムをコンパイルして、クラスファイルを作り直す作業も必要です。

値を自由に出し入れすることのできる**変数**（*variable*）を使うと、そのような煩（わずら）わしさから解放されます。

変数の宣言

変数とは、数値を格納するための《**箱**》のようなものです。いったん箱に値を入れておけば、その箱が存在する限り**値が保持されます**。また、値を書きかえるのも取り出すのも自由です。

プログラム中に複数の箱があると、どれが何のための箱なのかが分からなくなってしまいますから、箱には《**名前**》が必要です。

そのため、変数を使うときは、名前を与えた上で箱を作るための**宣言**（*declaration*）を行うことになっています。

以下に示すのが、*x* という名前の変数を宣言する**宣言文**（*declaration statement*）です。

```
int x;            // xという名前をもつint型変数の宣言
```

先頭の **int**（イント）は、『整数』という意味の語句 integer に由来します。この宣言によって、名前が *x* の変数（箱）が作られます（**Fig.2-4**）。

変数 *x* が扱えるのは“**整数**”だけです（たとえば3.5 といった“実数値”は扱えません）。これは、**int** という**型**（かた）（*type*）の性質です。

int は**型**であり、その型から作られた変数 *x* が **int** 型の**実体**というわけです。

重 要 **変数**を使うためには、まず宣言をして《**型**》と《**名前**》を与えよう。

▶ 本書では型名を含めたキーワード（p.84）を**太字**で表記し、変数名を*斜め文字*で表記します。

変数に値を入れ、その値を表示するプログラムを作りましょう。それが **List 2-5** です。

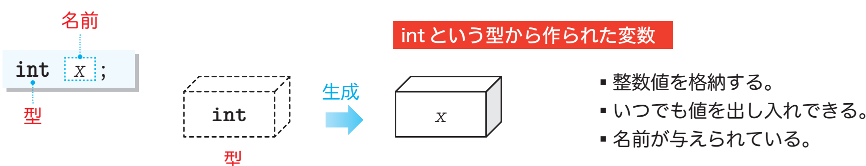


Fig.2-4 変数と宣言

List 2-5 Chap02/Variable.java

```
// 変数に値を代入して表示

class Variable {

    public static void main(String[] args) {
        int x;              // xはint型の変数            ← 宣言文

1 ─     x = 63;             // xに63を代入

2 ─     System.out.println(x);          // xの値を表示
    }
}
```

実行結果

```
63
```

代入演算子

変数に値を入れるのが 1 の箇所です。**Fig.2-5** に示すように、**=** は、右辺の値を左辺の変数に**代入する**ための記号であり、**代入演算子**（*assignment operator*）と呼ばれます。

数学のように『x と 63 が等しい。』といっているのではないことに注意しましょう。

▶ 演算子 **=** は、p.111 で学習する**複合代入演算子**と区別するために、**単純代入演算子**（*simple assignment operator*）とも呼ばれます。

なお、**int** 型で表現できる範囲は **-2,147,483,648** ～ **2,147,483,647** です（p.144）から、これ以外の値は代入できません。

Fig.2-5 代入演算子による変数への値の代入

変数の値の表示

変数に格納されている値は、いつでも取り出せます。2 では、**Fig.2-6** に示すように、変数 x の値を取り出して表示しています。

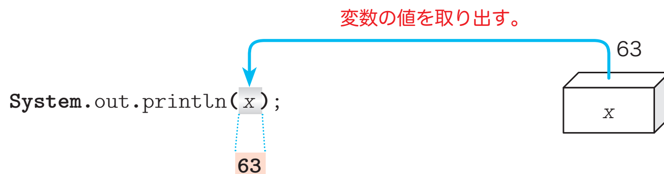


Fig.2-6 変数の値の取出しと表示

表示されるのは、x の《**値**》であって、《**変数名**》ではありません。また、以下の二つを混同しないようにしましょう。

```
System.out.println(x);          // 変数xの値を表示（整数値）
System.out.println("x");        // 『x』と表示（文字列）
```

次に、複数の変数を使ったプログラムに挑戦しましょう。List 2-6 に示すのは、int 型変数 x と y に値 63 と 18 を入れて、その合計と平均を表示するプログラムです。

List 2-6　Chap02/SumAve1.java

```
// 二つの変数xとyの合計と平均を表示

class SumAve1 {

   public static void main(String[] args) {
      int x;             // xはint型の変数          ❶
      int y;             // yはint型の変数

      x = 63;            // xに63を代入             ❷
      y = 18;            // yに18を代入

      System.out.println("xの値は" + x + "です。");          // xの値を表示   ❸
      System.out.println("yの値は" + y + "です。");          // yの値を表示
      System.out.println("合計は" + (x + y) + "です。");     // 合計を表示     ❹
      System.out.println("平均は" + (x + y) / 2 + "です。"); // 平均を表示
   }
}
```

実行結果

```
xの値は63です。
yの値は18です。
合計は81です。
平均は40です。
```

二つの変数 x と y を宣言しているのが、❶です。ここでは別々に宣言していますが、以下のようにコンマ文字 , で区切れば、二つ以上の変数を一度に宣言できます。

```
int x, y;          // int型の変数xとyを一度に宣言
```

もっとも、本プログラムのように 1 行ずつ別々に変数を宣言したほうが、個々の宣言に対するコメント（注釈）が記入しやすくなりますし、宣言の追加や削除も容易です。

▶ ただし、1 行ずつ宣言するとプログラムの行数が増えます。

変数 x と y に値を代入しているのが❷で、その値を表示しているのが❸です。

文字列と数値を + 演算子で結ぶと、数値が文字列に変換された上で連結されることを利用して表示を行っています（Fig.2-7）。

▶ まず最初に、文字列 "xの値は" と、変数 x の値 63 が文字列に変換された "63" とが連結されます。それから、文字列 "xの値は63" と、文字列 "です。" とが連結されます。最後に、連結の最終的な結果である文字列 "xの値は63です。" が表示されます。

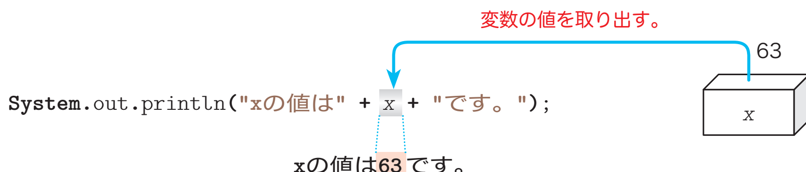


Fig.2-7　標準出力ストリームへの変数の値の出力

算術演算と演算のグループ化

4では、x と y の合計（x + y）と、平均（x + y） / 2 とを表示しています。スラッシュ記号 / は、**除算**を行うための記号です。

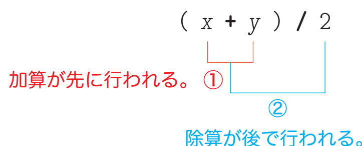
平均を求める式の構造を示したのが **Fig.2-8**aです。x + y が ( ) で囲まれているため、まず x + y による加算が行われ、それから 2 で割る除算が行われます。

*

もしも図bのように、( ) がなくて x + y / 2 となっていたら、x と y / 2 との和を求めることになります。私たちが日常行っている計算と同じで、**加減算よりも乗除算のほうが優先される**からです。

▶ すべての演算子と、その優先順位は p.88 にまとめています。

a xとyの平均を求める



b xに $\frac{y}{2}$ を加える

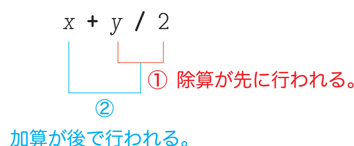


Fig.2-8 ( ) による演算順序の変更

なお、『整数 / 整数』の演算では、**小数部（小数点以下の部分）が切り捨てられます。**実行結果が示すように、63 と 18 の平均値が 40.5 ではなく 40 となるのは、そのためです。

重 要 『整数 / 整数』の演算結果は、小数部が切り捨てられた整数となる。

▶ 変数 x や y を宣言する宣言文には、『x は int 型の変数』や『y は int 型の変数』といったコメントが与えられています。これは、初学者である読者のみなさんに向けたコメントです。
　『x は int 型の変数』ということは、見た目で分かるため、実際のプログラムでは、このようなコメントを記述することはありません。本来は、“x が何のための変数なのか” といったことを簡潔に記述します。

演習 2–1

List 2-6 の2の箇所を、小数部をもつ実数値を x と y に代入するように変更して、その結果を考察せよ。

演習 2–2

三つの `int` 型変数に値を代入し、合計と平均を求めるプログラムを作成せよ。

変数と初期化

前のプログラムから、変数に値を代入する2の部分を削除するとどうなるかの検証を行います。**List 2-7** をコンパイルしてみましょう。

List 2-7 Chap02/SumAve2.java

```
// 二つの変数xとyの合計と平均を表示（誤り）

class SumAve2 {

    public static void main(String[] args) {
        int x;          // xはint型の変数
        int y;          // yはint型の変数

        System.out.println("xの値は" + x + "です。");          // xの値を表示
        System.out.println("yの値は" + y + "です。");          // yの値を表示
        System.out.println("合計は" + (x + y) + "です。");      // 合計を表示
        System.out.println("平均は" + (x + y) / 2 + "です。");  // 平均を表示
    }
}
```

値の入っていない変数の値を取り出そうとする。

実行結果
コンパイルエラーとなるため
実行できません。

コンパイル時にエラーとなるため、プログラムを実行することはできません。というのも、以下の規則があるからです。

重 要 値の入っていない変数からは、値は取り出せない。

初期化を伴う宣言

変数に入れるべき値が分かっているのであれば、その値を最初から変数に入れておいたほうがよいでしょう。

そのように修正したプログラムが **List 2-8** です。網かけ部の宣言によって、変数 x と変数 y は、その生成時に 63 と 18 という値で**初期化**（*initialize*）されます。= 記号の右側の部分は、変数に入れるべき値であって**初期化子**（*initializer*）と呼ばれます（**Fig.2-9**）。

重 要 変数の宣言時には、初期化子を与えて確実に初期化しよう。

変数が生成される際に入れるべき値を設定する。

```
int x = 63;
```

初期化子

Fig.2-9 初期化を伴う宣言

List 2-8 Chap02/SumAve3.java

```
// 二つの変数xとyの合計と平均を表示（変数を初期化）

class SumAve3 {

    public static void main(String[] args) {
        int x = 63;         // xはint型の変数
        int y = 18;         // yはint型の変数

        System.out.println("xの値は" + x + "です。");              // xの値を表示
        System.out.println("yの値は" + y + "です。");              // yの値を表示
        System.out.println("合計は" + (x + y) + "です。");         // 合計を表示
        System.out.println("平均は" + (x + y) / 2 + "です。");     // 平均を表示
    }
}
```

初期化

実行結果

```
xの値は63です。
yの値は18です。
合計は81です。
平均は40です。
```

初期化と代入

本プログラムで行っている《初期化》と **List 2-6**（p.32）で行った《代入》は、値を入れるという点では同じであるものの、そのタイミングが異なります。

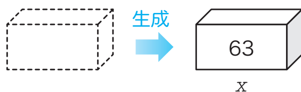
以下のように理解しましょう（**Fig.2-10**）。

- **初期化**：変数を生成するときに値を入れること。
- **代　入**：生成ずみの変数に値を入れること。

▶ 本書では、初期化を指定する記号=を細字で示し、代入演算子**=**を**太字**で示して区別しています。

ⓐ 初期化

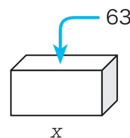
```
int x = 63;
```



変数の生成時に値を入れる。

ⓑ 代入

```
x = 63;
```



生成ずみの変数に値を入れる。

Fig.2-10　初期化と代入

複数の変数をまとめて宣言する際は**コンマで区切ります**（p.32）。そのため、本プログラムの網かけ部を 1 行にまとめた宣言は、以下のようになります。

```
int x = 63, y = 18;
```

2-2 キーボードからの入力

変数を使うことの最大のメリットは、自由に値を入れたり出したりできることです。本節では、キーボードから読み込んだ値を変数に入れる方法を学習します。

キーボードからの入力

キーボードから二つの整数値を読み込んで、それらに対して加算・減算・乗算・除算を行った結果を表示しましょう。そのプログラムを **List 2-9** に示します。

List 2-9 Chap02/ArithInt.java

```
// 二つの整数値を読み込んで加減乗除した値を表示

import java.util.Scanner;

class ArithInt {

    public static void main(String[] args) {
        Scanner stdIn = new Scanner(System.in);

        System.out.println("xとyを加減乗除します。");

        System.out.print("xの値：");    // xの値の入力を促す
        int x = stdIn.nextInt();        // xに整数値を読み込む

        System.out.print("yの値：");    // yの値の入力を促す
        int y = stdIn.nextInt();        // yに整数値を読み込む

        System.out.println("x + y = " + (x + y));   // x + yの値を表示
        System.out.println("x - y = " + (x - y));   // x - yの値を表示
        System.out.println("x * y = " + (x * y));   // x * yの値を表示
        System.out.println("x / y = " + (x / y));   // x / yの値を表示（商）
        System.out.println("x % y = " + (x % y));   // x % yの値を表示（剰余）
    }
}
```

注意!! 大文字の“アイ”です。（System.in）

1 int x = stdIn.nextInt();
2 int y = stdIn.nextInt();

実行例

```
xとyを加減乗除します。
xの値：7
yの値：5
x + y = 12
x - y = 2
x * y = 35
x / y = 1
x % y = 2
```

キーボードからの読込みには、いくつかの手続きが必要です。そのテクニックは高度ですから、現時点で理解する必要はありません。《決まり文句》として覚えましょう。

その要点を示したのが **Fig.2-11** です。

a プログラムの先頭に置きます。

b `main` メソッドの先頭に置きます。`System.in` は、キーボードと結び付くストリームである**標準入力ストリーム**（*standard input stream*）です。

▶ 画面に文字を表示する際に利用する `System.out` は、標準出力ストリームです（p.14）。

c キーボードから `int` 型整数値を読み込む部分です。プログラム中の `stdIn.nextInt()` が、キーボードから読み込んだ《値》となります。

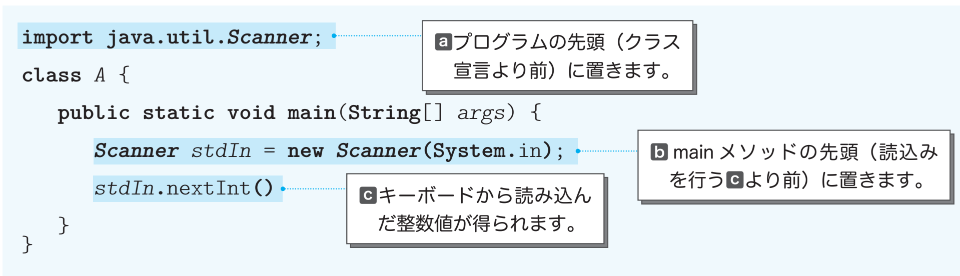


Fig.2-11　キーボードからの読込みを行うプログラム

キーボードから読み込んだ整数値を変数に格納する様子を **Fig.2-12** に示します。

▶　入力する値は、**int** 型で表現できる範囲 -2,147,483,648 ～ 2,147,483,647 に収まっていなければなりません。また、アルファベットや記号文字などを打ち込まないようにします（このあたりのことは、第 16 章で詳しく考察します）。

　ストリームとは、文字が流れる川のようなものです (p.14)。キーボードと結び付いた標準入力ストリーム **System**.in から文字や数値を取り出す《抽出装置》を表すための変数が *stdIn* です。*stdIn* という名前は、私が与えたものですから、他の名前に変更しても構いません（その場合は、プログラム中のすべての *stdIn* を変更します）。

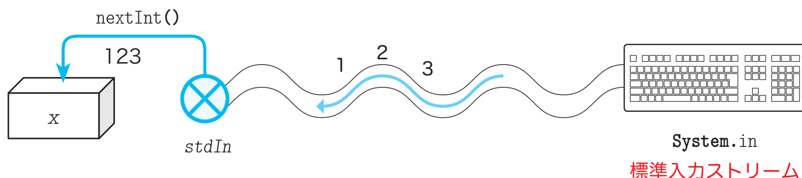


Fig.2-12　キーボードからの入力

本プログラムでは、宣言1と2の初期化子でcを利用しています。そのため、変数 *x* と変数 *y* は、キーボードから読み込まれた整数値で初期化されます。

さて、これらの宣言が、**main** メソッドの途中にあることに注意しましょう。メソッドの途中であっても、必要になった箇所で変数を宣言するのが原則です。

重 要　変数の宣言は、必要になった時点で行おう。

▶　本プログラムでは、読込みのための式 *stdIn*.nextInt() を初期化子として利用しています。したがって、変数 *x* と *y* は、キーボードから読み込んだ整数値で**初期化**されます。

　以下のように、いったん変数を宣言しておき、その後で変数に式 *stdIn*.nextInt() を**代入**することもできます（ただし、冗長になります）。

```
int x;                  // いったん宣言
x = stdIn.nextInt();    // それから代入
```

演算子とオペランド

本プログラムで初めて使っているのが、減算を行う -、乗算を行う *、除算の剰余（余り）を求める % です。

演算を行う + や - などの記号を**演算子**（*operator*）と呼び、演算の対象となる式のことを**オペランド**（*operand*）と呼びます。

たとえば、*x* と *y* の和を求める式 *x* + *y* において、演算子は + であって、オペランドは *x* と *y* の二つです（**Fig.2-13**）。

▶ 左側のオペランドを第１オペランドあるいは左オペランドと呼び、右側のオペランドを第２オペランドあるいは右オペランドと呼びます。

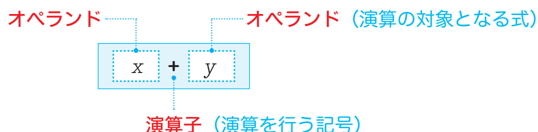


Fig.2-13　演算子とオペランド

本プログラムで利用している演算子 +, -, *, /, % の概略を **Table 2-1** と **Table 2-2** に示します。

いずれも２個のオペランドをもつ演算子です。このような演算子は、**２項演算子**（*binary operator*）と呼ばれます。その他に、オペランドが１個の**単項演算子**（*unary operator*）とオペランドが３個の**３項演算子**（*ternary operator*）があります。

▶ Java には４項以上の演算子はありません。

Table 2-1　加減演算子（additive operator）

x + *y*	*x* に *y* を加えた結果を生成。
x - *y*	*x* から *y* を減じた結果を生成。

Table 2-2　乗除演算子（multiplicative operator）

x * *y*	*x* に *y* を乗じた値を生成。
x / *y*	*x* を *y* で割った商を生成（*x*, *y* ともに整数であれば小数点以下は切捨て）。
x % *y*	*x* を *y* で割った剰余を生成。

Table 2-3　単項符号演算子（unary plus operator and unary minus operator）

+*x*	*x* そのものの値を生成。
-*x*	*x* の符号を反転した値を生成。

Table 2-3 に示すように、**+** 演算子と **-** 演算子には、２項演算子の他に、単項演算子版もあります。単項符号演算子を利用したプログラムを作りましょう。**List 2-10** は、整数値を読み込んで、その符号を反転した値を表示するプログラムです。

List 2-10 Chap02/Minus.java

```
// 整数値を読み込んで符号を反転した値を表示

import java.util.Scanner;

class Minus {

    public static void main(String[] args) {
        Scanner stdIn = new Scanner(System.in);

        System.out.print("整数値：");
        int a = stdIn.nextInt();    // aに整数値を読み込む

1       int b = -a;                 // aの符号を反転した値でbを初期化
2       System.out.println(a + "の符号を反転した値は" + b + "です。");
    }
}
```

実行例1

```
整数値：7⏎
7の符号を反転した値は-7です。
```

実行例2

```
整数値：-15⏎
-15の符号を反転した値は15です。
```

宣言1では、変数 *b* を -a で初期化しています。この単項 **-** 演算子は、**オペランドの符号を反転した値**を生成します。

*

もう一つの単項 **+** 演算子は、あまり使われません。というのも、+a は a の値そのものを表すからです。この演算子を利用すると、2の部分は以下のようにも実現できます。

```
System.out.println(+a + "の符号を反転した値は" + b + "です。");
```

もちろん、a の前に置かれた網かけ部の **+** は省略可能です。

Column 2-2 除算の演算結果

剰余を求める演算 a **%** b では、(a **/** b) ***** b **+** (a **%** b) が a と等しくなるような結果が生成されます。その際、演算結果の大きさと符号は、次のようになります。

- 大きさ … 割る数の大きさより小さくなる。
- 符　号 … 割られる数が負であれば負となり、割られる数が正であれば正となる。

/ 演算子と **%** 演算子の演算結果の具体例を以下に示します。

正÷正	5 / 3	→ 1	5 % 3	→ 2
正÷負	5 / (-3)	→ -1	5 % (-3)	→ 2
負÷正	(-5) / 3	→ -1	(-5) % 3	→ -2
負÷負	(-5) / (-3)	→ 1	(-5) % (-3)	→ -2

基本型

ここまでのプログラムで使った変数は、すべて **int** 型でした。

Java では多くの型が提供されるとともに、自分で型を作ることもできます。Java 言語が標準で提供している型を**基本型**（*primitive type*）と呼びます。基本型には、整数型や浮動小数点型などがあります。

整数型

整数を表す型です。代表的なのは、以下に示す四つの型です。

byte	1バイト整数	-128 ～ 127	
short	短い整数	-32,768 ～ 32,767	およそ±３万２千
int	整数	-2,147,483,648 ～ 2,147,483,647	およそ±21 億
long	長い整数	-9,223,372,036,854,775,808 ～ 9,223,372,036,854,775,807	±90京

型によって表現できる数値の範囲が異なります。自分が表したい数値の範囲によって、使い分けます。

浮動小数点型

実数を表す型です。以下に示す二つの型があります。

float	単精度浮動小数点数	±3.40282347E+38 ～ ±1.40239846E-45
double	倍精度浮動小数点数	±1.79769313486231507E+378 ～ ±4.94065645841246544E-324

実数の内部は**浮動小数点数**（*floating point number*）という形式で表現されています。とりあえずは、以下のように理解しておきましょう。

『実数を表す専門用語が浮動小数点数である。』

なお、3.14 とか 13.5 といった定数値は、**浮動小数点リテラル**（*floating point literal*）と呼ばれます。

*

この他にも、**文字型**（**char** 型）と**論理型**（**boolean** 型）があります。基本型の詳細は、第５章で学習します。

演習 2–3

右に示すように、キーボードから読み込んだ整数値をそのまま反復して表示するプログラムを作成せよ。

```
整数値：7
7と入力しましたね。
```

演習 2–4

右に示すように、キーボードから読み込んだ整数値に 10 を加えた値と 10 を減じた値を出力するプログラムを作成せよ。

```
整数値：7
10を加えた値は17です。
10を減じた値は-3です。
```

実数値の読込み

二つの**実数値**を加減乗除するプログラムを作りましょう。整数を表すための **int** 型は使えませんので、小数点以下の部分をもつ実数を扱える **double 型**を使います。

プログラムは **List 2-11** のようになります。

List 2-11 Chap02/ArithDouble.java

```
// 二つの実数値を読み込んで加減乗除した値を表示

import java.util.Scanner;

class ArithDouble {

    public static void main(String[] args) {
        Scanner stdIn = new Scanner(System.in);

        System.out.println("xとyを加減乗除します。");

        System.out.print("xの値：");      // xの値の入力を促す
        double x = stdIn.nextDouble();    // xに実数値を読み込む

        System.out.print("yの値：");      // yの値の入力を促す
        double y = stdIn.nextDouble();    // yに実数値を読み込む

        System.out.println("x + y = " + (x + y));  // x + yの値を表示
        System.out.println("x - y = " + (x - y));  // x - yの値を表示
        System.out.println("x * y = " + (x * y));  // x * yの値を表示
        System.out.println("x / y = " + (x / y));  // x / yの値を表示（商）
        System.out.println("x % y = " + (x % y));  // x % yの値を表示（剰余）
    }
}
```

実行例

```
xとyを加減乗除します。
xの値：9.75
yの値：2.5
x + y = 12.25
x - y = 7.25
x * y = 24.375
x / y = 3.9
x % y = 2.25
```

プログラムは **List 2-9** とほぼ同じです。変数 *x* と *y* の型が **double** 型になっている点が異なります。

もう一つ変更されているのが、網かけ部です。キーボードから **double** 型の実数値を読み込むときは、nextInt() ではなく nextDouble() を使います。

▶ 小数点以下の部分のない値をキーボードから打ち込む際は、小数点以下は省略可能です。たとえば 5.0 は、5.0 と入力しても、5 と入力しても、5. と入力してもよいことになっています。

演習 2-5

二つの実数値を読み込み、その和と平均を求めて表示するプログラムを作成せよ。

```
xの値：7.5
yの値：5.25
合計は12.75です。
平均は6.375です。
```

演習 2-6

三角形の底辺と高さを読み込んで、その面積を表示するプログラムを作成せよ。

```
三角形の面積を求めます。
底辺：7.5
高さ：2.5
面積は9.375です。
```

final 変数

円の半径をキーボードから読み込んで、その円の“**円周の長さ**”と“**面積**”を求めて表示するプログラムを作りましょう。それが **List 2-12** です。

List 2-12 Chap02/Circle1.java

```
// 円周の長さと円の面積を求める（その１：円周率を浮動小数点リテラルで表す）

import java.util.Scanner;

class Circle1 {

   public static void main(String[] args) {
      Scanner stdIn = new Scanner(System.in);

      System.out.print("半径：");
      double r = stdIn.nextDouble();    // 半径

      System.out.println("円周の長さは" + 2 * 3.14 * r + "です。");
      System.out.println("面積は" + 3.14 * r * r + "です。");
   }
}
```

実行例

```
半径：7.2⏎
円周の長さは45.216です。
面積は162.7776です。
```

円周と面積を求める公式を **Fig.2-14** に示します。式中の π（パイ）は、**円周率**です。

本プログラムでは、この公式どおりに、円周の長さと面積を求めています。

円周率 π を表すのが、網かけ部の浮動小数点リテラル 3.14 です。

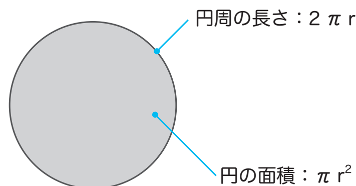


Fig.2-14　円の周囲の長さと面積

＊

さて、円周率は 3.14 ではなく、3.1415926535… と無限に続く値です。

円周の長さと面積をより正確に求めるために、円周率を 3.1416 に変えることを考えましょう。そのためには、網かけ部を変更することになります。

変更は２箇所だけですから、作業は容易です。しかし、もしプログラム中に 3.14 が数百箇所あったら、どうなるでしょうか。

エディタの《置換》機能を使えば、すべての 3.14 を 3.1416 に変更するのは容易です。とはいえ、円周率ではない値として 3.14 を使っている箇所がプログラム中にあるかもしれません。そのような箇所は、置換の対象から外す必要があります。すなわち、**選択的な置換が要求されるわけです**。

＊

このようなケースで効力を発揮するのが、**値を書きかえることのできない final（ファイナル）変数**です。`final` 変数を用いて書きかえたプログラムを **List 2-13** に示します。

宣言に `final` が付けられた *PI* は、3.1416 で初期化された `final` 変数となります。計算で円周率が必要な箇所では、その変数 *PI* の値を利用しています。

List 2-13　Chap02/Circle2.java

```
// 円周の長さと円の面積を求める（その２：円周率をfinal変数で表す）

import java.util.Scanner;

class Circle2 {

    public static void main(String[] args) {
        final double PI = 3.1416;          // 円周率
        Scanner stdIn = new Scanner(System.in);

        System.out.print("半径：");
        double r = stdIn.nextDouble();    // 半径

        System.out.println("円周の長さは" + 2 * PI * r + "です。");
        System.out.println("面積は" + PI * r * r + "です。");
    }
}
```

実行例

```
半径：7.2
円周の長さは45.23904です。
面積は162.860544です。
```

final 変数を利用するメリットは、以下のとおりです。

1 値の管理を一箇所に集約できる

円周率の値 3.1416 は、final 変数 *PI* の初期化子となっています。もし他の値（たとえば 3.14159）に変える場合、プログラムの変更は一箇所だけですみます。

また、タイプミスや置換の失敗などによって、たとえば 3.1416 と 3.14159 とを混在させてしまう、といったミスも防げます。

2 プログラムが読みやすくなる

プログラムの中では、数値ではなく変数名 *PI* で円周率を参照できますから、プログラムが読みやすくなります。

重 要　プログラム中に埋め込まれた数値は、何を表すためのものであるかが理解しにくい。final 変数として宣言して名前を与えるとよい。

なお、final 変数の名前は大文字とすることが推奨されています。final でない普通の変数と見分けやすくするためです。

▶　プログラム中に埋め込まれた、意図が分かりにくい数値は、**マジックナンバー**（*magic number*）と呼ばれます。final 変数を導入すると、マジックナンバーを除去できます。

final 変数は原則として初期化すべきです。なお、初期化されていない final 変数には、1回だけ値を代入できます。**すなわち、初期化と代入のいずれか一方によって、1回だけ値を入れられます（2回目はエラーとなります）。**

```
final int A = 1;

A = 2;            // エラー
```

```
final int B;
B = 1;            // ＯＫ
B = 2;            // エラー
```

▶　final には『最後の』という意味があります。クイズの“final アンサー”には、『最終決定版であって、もはや変更できない解答』というニュアンスがあります。それと同じです。

乱数の生成

キーボードから値を読み込むのではなく、**コンピュータに値を作ってもらうこともできます。** **List 2-14** に示すのが、そのプログラム例です。

このプログラムは、0 から 9 までの数値の一つを《ラッキーナンバー》として生成して表示します。

List 2-14 Chap02/LuckyNo.java

```
// 0〜9のラッキーナンバーを乱数で生成して表示

import java.util.Random;  ←[1]

class LuckyNo {

    public static void main(String[] args) {
[2]→    Random rand = new Random();
                                       [3]
        int lucky = rand.nextInt(10);          // 0〜9の乱数

        System.out.println("今日のラッキーナンバーは" + lucky + "です。");
    }
}
```

実行例

今日のラッキーナンバーは6です。

コンピュータが生成するランダムな値のことを**乱数**（らんすう）と呼びます。[1]・[2]・[3]は、乱数の生成に必要な《決まり文句》です（**Column 2-3**）。

▶ この《決まり文句》は、キーボードからの読込みを行うための《決まり文句》と似ています。注意すべき点も、ほぼ同じです。

- [1]は、クラス宣言より前に置かなければならない。
- [2]は、[3]より前に置かなければならない。

なお、[2]と[3]の変数名 *rand* は、他の名前に変更しても構いません。

肝心なのは[3]の箇所です。**Fig.2-15** に示すように、*rand*.nextInt(*n*) の部分は、**0 以上 n 未満のランダムな整数値となります。**

本プログラムでは *rand*.nextInt(10) となっていますので、その値は 0, 1, 2, …, 9 のいずれかとなります。

これで、変数 *lucky* は、0 以上 9 以下のどれか一つの値で初期化されます。

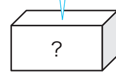


Fig.2-15 乱数の生成

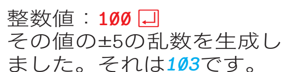
演習 2-7

以下に示すプログラムを作成せよ。

- 1 桁の正の整数値（すなわち 1 以上 9 以下の値）をランダムに生成して表示。
- 1 桁の負の整数値（すなわち -9 以上 -1 以下の値）をランダムに生成して表示。
- 2 桁の正の整数値（すなわち 10 以上 99 以下の値）をランダムに生成して表示。

演習 2–8

キーボードから読み込んだ整数値プラスマイナス 5 の範囲の整数値をランダムに生成して表示するプログラムを作成せよ。



演習 2–9

以下に示すプログラムを作成せよ（実数値の乱数の生成には nextDouble() を使うこと：**Column 2-3** 参照）。

- 0.0 以上 1.0 未満の実数値をランダムに生成して表示。
- 0.0 以上 10.0 未満の実数値をランダムに生成して表示。
- -1.0 以上 1.0 未満の実数値をランダムに生成して表示。

Column 2-3　乱数の生成

乱数の生成に必要な1・2・3については現時点で理解する必要はありません。第 7 章・第 10 章・第 11 章などの学習が終了した後に、この **Column** を読むとよいでしょう。

＊

Random は、Java が提供する莫大なクラスライブラリの中の一つです。*Random* クラスのインスタンスは、一連の擬似乱数（ぎじらんすう）を生成します。乱数は《無（む）》から生成されるのではなく、《種（たね）》と呼ばれる数値に対して種々の演算を行うことによって得られます（種とは、乱数を産み出すための卵（たまご）のようなものです。*Random* クラスでは 48 ビットの種が使われて、その種は線形合同法という計算法によって変更されます）。

Random クラスのインスタンスの生成は、以下のいずれかの形式で行えます。

```
a  Random rand = new Random();
b  Random rand = new Random(5);
```

List 2-14 で利用したのはaであり、乱数ジェネレータ（生成器）が新規に作られます。このとき、*Random* クラスの他のインスタンスと重複しないように《種》の値が自動的に決定されます。

プログラム側で明示的に《種》を与える方法がbです。与えられた種に基づいて乱数ジェネレータが生成されます。

＊

List 2-14 のプログラムでは、int 型整数の乱数を生成する nextInt メソッドを利用しました。この他にも **Table 2C-1** に示すメソッドがあります。用途や目的に応じて使い分けます。

Table 2C-1　Radom クラスのメソッド

メソッド	型	生成される値の範囲
nextBoolean()	boolean	true または false
nextInt()	int	-2147483648 ～ +2147483647
nextInt(*n*)	int	0 ～ *n* - 1
nextLong()	long	-9223372036854775808 ～ +9223372036854775807
nextDouble()	double	0.0 以上 1.0 未満
nextFloat()	float	0.0 以上 1.0 未満

なお、*Math* クラスでも乱数を生成するライブラリが提供されます（p.353）。

文字列の読込み

次は、数値ではなく**文字列**（文字の並び）を扱うプログラムを作りましょう。**List 2-15** は、名前を入力してもらって、挨拶を表示するプログラムです。

List 2-15 Chap02/HelloNext.java

```
// 名前を読み込んで挨拶する（その１：next()版）

import java.util.Scanner;

class HelloNext {

    public static void main(String[] args) {
        Scanner stdIn = new Scanner(System.in);

        System.out.print("お名前は：");
        String s = stdIn.next();    // 文字列を読み込む

        System.out.println("こんにちは" + s + "さん。");    // 表示
    }
}
```

実行例 1
```
お名前は：柴田望洋⏎
こんにちは柴田望洋さん。
```

実行例 2
```
お名前は：柴田 望洋⏎
こんにちは柴田さん。
```

読み込んだ文字列を格納する変数 s の型は、**String 型**です。これは、文字列を表すための型です（**Column 2-4**）。

＊

文字列の読込みに使うのが、網かけ部の next() です。

ただし、next() によるキーボードからの読込みでは、空白文字やタブ文字が文字列の区切りとみなされます。そのため、途中にスペース文字を入れて入力する実行例②では、"柴田" のみが s に読み込まれます。

＊

スペースも含めた１行分の入力を文字列として読み込むときに使うのが、nextLine() です。プログラムを **List 2-16** に示します。

List 2-16 Chap02/HelloNextLine.java

```
// 名前を読み込んで挨拶する（その２：nextLine()版）

import java.util.Scanner;

class HelloNextLine {

    public static void main(String[] args) {
        Scanner stdIn = new Scanner(System.in);

        System.out.print("お名前は：");
        String s = stdIn.nextLine();    // １行分の文字列を読み込む

        System.out.println("こんにちは" + s + "さん。");    // 表示
    }
}
```

実行例 1
```
お名前は：柴田望洋⏎
こんにちは柴田望洋さん。
```

実行例 2
```
お名前は：柴田 望洋⏎
こんにちは柴田 望洋さん。
```

String 型の変数に対しては、文字列による**初期化**や、文字列の**代入**も行えます。プログラム例を **List 2-17** に示します。

List 2-17 Chap02/StringInitAssign.java

```
// 文字列の初期化と代入

class StringInitAssign {

   public static void main(String[] args) {
      String s1 = "ABC";    // 初期化
      String s2 = "XYZ";    // 初期化

      s1 = "FBI";           // 代入（値を書きかえる）

      System.out.println("文字列s1は" + s1 + "です。");   // 表示
      System.out.println("文字列s2は" + s2 + "です。");   // 表示
   }
}
```

実行結果

```
文字列s1はFBIです。
文字列s2はXYZです。
```

文字列 *s1* は "ABC" で初期化されて、その後で "FBI" が代入されます。そのため、*s1* は "ABC" から "FBI" に変更されます。

重 要 文字列（文字の並び）は、String 型で表せる。

▶ String は、第 8 章以降で学習する《**クラス**》で作られた型です。現時点で詳しいことを理解する必要はありませんが、*s1* への "FBI" の代入は、“文字列の中身の書きかえ” ではなく、“参照先の書きかえ” です。詳細は第 15 章で学習します。

演習 2-10

右に示すように、名前の姓と名とを個別にキーボードから読み込んで、挨拶を行うプログラムを作成せよ。

```
姓：柴田 ⏎
名：望洋 ⏎
こんにちは柴田望洋さん。
```

Column 2-4 String 型は特殊な型

文字列を扱うための String 型は、基本型ではなくて、第 8 章以降で学習する《**クラス**》によって実現された型です（型名の先頭文字が大文字である点も int や double などと異なります）。

この型の変数は、単独の箱ではなく、文字列本体の箱と、それを参照する箱がセットとなっています。そのイメージを示したのが **Fig.2C-1** です（詳細は第 15 章で学習します）。

```
String s = "ABC";
```

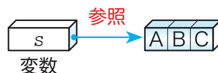


Fig.2C-1 String 型の変数と文字列

まとめ

- 数値などのデータを、自由に入れたり取り出したりできるのが、**変数**である。変数は必要になった時点で**型**と**名前**を与えて**宣言**する。

- 変数から値を取り出して利用する前に、**初期化**あるいは**代入**によって、その変数に値を入れておかなければならない。変数を生成する際に**初期化子**の値を入れるのが初期化であり、生成ずみの変数に値を入れるのが代入である。

- 変数は、必要になったときに宣言する。なお、宣言には初期化子を与え、変数を確実に初期化するとよい。

- `final` 変数には、初期化もしくは代入によって、値を1回だけ入れられる。定数に対して名前を与えるために利用するとよい。

- 数多くの型のうち、言語が提供する型が**基本型**である。

- 整数を表す**整数型**の一つが **`int` 型**である。

- 13 などの整数定数は、**整数リテラル**と呼ばれる。

- 実数（浮動小数点数）を表す**浮動小数点型**の一つが **`double` 型**である。

- 3.14 などの浮動小数点定数は、**浮動小数点リテラル**と呼ばれる。

- 文字列（文字の並び）を表すのは **`String` 型**である。この型は基本型ではない。

- 演算を行う記号が**演算子**、演算の対象となる式が**オペランド**である。演算子をオペランドの個数で分類すると、**単項演算子**・**２項演算子**・**３項演算子**の３種類がある。

- ( ) で囲まれた演算は優先的に行われる。

- 『文字列 + 数値』あるいは『数値 + 文字列』の演算では、数値が文字列に変換された上で連結が行われる。

- キーボードからの読込みを行う際は**標準入力ストリーム**を利用する。標準入力ストリームからの文字の読込みには、*Scanner* クラスの next... メソッドを利用する。

- **乱数**を生成すると、ランダムな値を作り出せる。乱数の生成には *Random* クラスの next... メソッドを利用する。

- 『整数 / 整数』の演算で得られる商は、小数部が切り捨てられた整数値である。

Chap02/Abc.java

```
import java.util.Random;

import java.util.Scanner;

class Abc {

    public static void main(String[] args) {

        Random rand = new Random();

        Scanner stdIn = new Scanner(System.in);

        int a;        // aはint型の変数

        a = 2;        // 代入（生成ずみの変数に値を入れる）

        int b = -1;   // 初期化（変数の生成時に値を入れる）

        double x = 1.5 * 2;

        // 値を書きかえられない変数（定数に名前を与える）
        final double PI = 3.14;

        x = rand.nextDouble();

        System.out.println(
            "半径" + x + "の円の面積は" +
            (PI * x * x) + "です。");

        System.out.print("整数aの値：");

        a = stdIn.nextInt();

        System.out.println("a / 2 = " + a / 2);

        System.out.println("a % 2 = " + a % 2);

        // 文字列型
        String s = "ABC";

        System.out.println("文字列sは" + s + "です。");

    }

}
```


乱数の生成

```
nextBoolean()
nextInt()
nextInt(n)
nextLong()
nextDouble()
nextFloat()
```

実行例

```
半径0.11992011858662233の
円の面積は
0.04515582140334483です。
整数aの値：7↵
a / 2 = 3
a % 2 = 1
文字列sはABCです。
```

キーボードからの読込み

```
nextBoolean()
nextByte()
nextShort()
nextInt()
nextLong()
nextDouble()
nextFloat()
next()
nextLine()
```

代入演算子	x = y		
加減演算子	x + y	x - y	
乗除演算子	x * y	x / y	x % y
単項符号演算子	+x	-x	

▶ 本文では、**int** 型整数を読み込む nextInt()、**double** 型実数を読み込む nextDouble()、文字列を読み込む next() と nextLine() を学習しました。読み込む型に応じてメソッドを使い分けます。