第1章

基本的なアルゴリズム

- アルゴリズムの定義
- 分岐
- 三値の最大値
- フローチャート（流れ図）
- 繰返し
- 1 から n までの和
- 正の値の読込み
- 九九の表示
- 三角形の表示

1-1 アルゴリズムとは

本節では、短く単純なプログラムを題材として、《アルゴリズム》とは何かを理解するとともに、その定義を学習します。

三値の最大値

まず最初に、**アルゴリズム**（*algorithm*）とは何かを、短く単純なプログラムを例にとって考えていくことにします。その題材として取り上げるのは、三つの値の《最大値》を求めるプログラムです。

プログラムを **List 1-1** に示します。変数 *a*，*b*，*c* に入れる値は、キーボードから読み込みます。そして、その最大値を変数 *max* に求めて表示します。

▶ キーボードからの数値の読込みには、***Scanner*** クラスを利用します（**Column 1-1**）。

List 1-1 Chap01/Max3.java

```
// 三つの整数値を読み込んで最大値を求めて表示

import java.util.Scanner;

class Max3 {

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

      System.out.println("三つの整数の最大値を求めます。");
      System.out.print("aの値：");  int a = stdIn.nextInt();
      System.out.print("bの値：");  int b = stdIn.nextInt();
      System.out.print("cの値：");  int c = stdIn.nextInt();

      int max = a;
      if (b > max) max = b;
      if (c > max) max = c;

      System.out.println("最大値は" + max + "です。");
   }
}
```

a, b, cの最大値を求めてmaxに代入

実行例

```
三つの整数の最大値を求めます。
aの値：1
bの値：3
cの値：2
最大値は3です。
```

まずはプログラムを実行して、動作を確認してみてください。

＊

さて、三つの変数 *a*，*b*，*c* の最大値を *max* として求めるのが、プログラムの網かけ部です。最大値を求める手順は、以下のようになっています。

1. *max* に *a* の値を入れる。
2. *b* の値が *max* よりも大きければ、*max* に *b* の値を入れる。
3. *c* の値が *max* よりも大きければ、*max* に *c* の値を入れる。

▶ 本書に示すプログラムは、ホームページからダウンロードできます（p.iv）。

Column 1-1　キーボードからの数値・文字列の読込み（その1）

キーボードからの数値・文字列の読込みには、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、《決まり文句》として覚えておくとよいでしょう。

Fig.1C-1 を見ながら、要点を理解していきましょう。

```
import java.util.Scanner;
class A {
    public static void main(String[] args) {
        Scanner stdIn = new Scanner(System.in);
        stdIn.nextInt()
    }
}
```

ⓐプログラムの先頭（クラス宣言より前）に置きます。

ⓑmainメソッドの先頭（読込みを行うⓒより前）に置きます。

ⓒキーボードから読み込んだ整数値が得られます。

Fig.1C-1　キーボードからの読込み

各部分の概要は、以下のとおりです。

ⓐ　`java.util` パッケージに所属する ***Scanner*** クラスを単純名で利用するための型インポート宣言です。プログラムの先頭（クラス宣言より前）に置きます。

※ 型インポート宣言・単純名については、**Column 2-4**（p.42）で解説しています。

ⓑ　`main` メソッドの先頭（読込みを行うⓒより前）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。

ⓒ　キーボードから `int` 型の整数値を読み込む部分です。メソッド呼出し式 *stdIn*`.nextInt()` を評価して得られるのが、キーボードから読み込んだ《値》です。

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示したのが、**Fig.1C-2** です。

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んではいけません。

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* という変数名は、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* を変更することになります）。

List 1-1 では、変数 *a*，*b*，*c* の宣言の初期化子でⓒを利用しています。そのため、三つの変数は、キーボードから読み込まれた整数値で初期化されることになります。

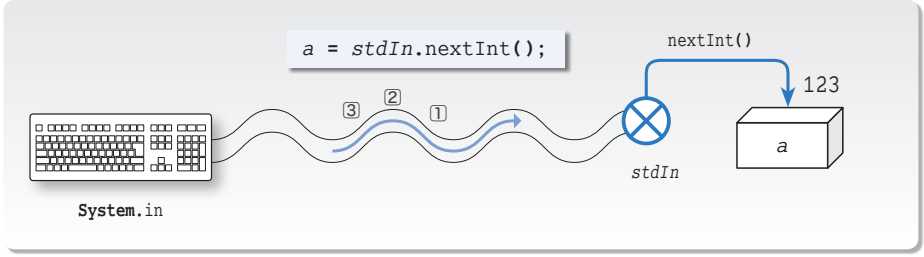

Fig.1C-2　キーボードからの読込み

三値の最大値を求めるための手続きを**流れ図＝フローチャート**（*flowchart*）として図式化すると、プログラムの流れが理解しやすくなります。**Fig.1-1** に示すのが、そのフローチャートです。

▶ フローチャートの記号は p.12 で解説します。

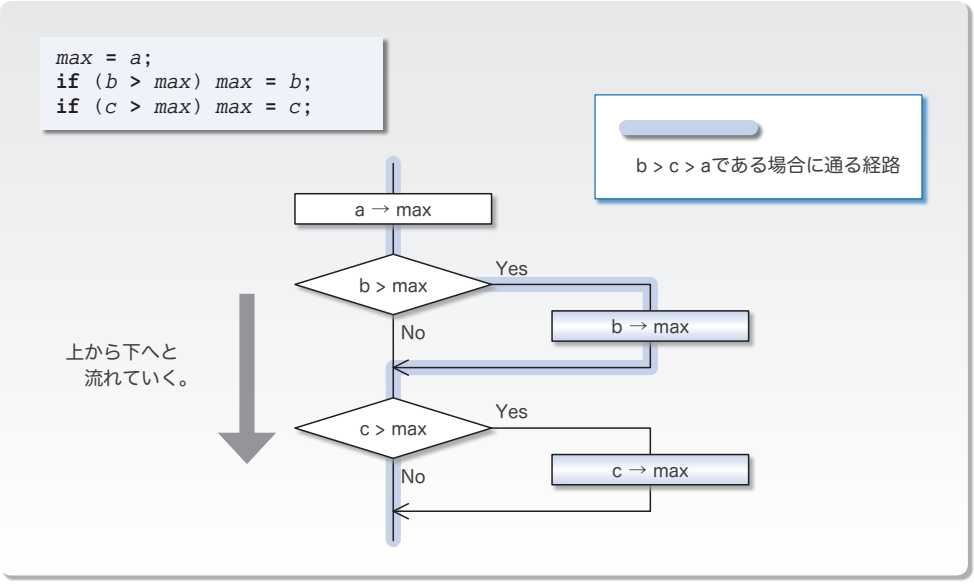

Fig.1-1 三値の最大値を求めるアルゴリズムの流れ図

プログラムの流れは、黒線 ── に沿って上から下へと流れていき、その過程で ▭ 内の処理が行われます。

ただし、◇を通過する際は、その中に記されている《条件》を評価した結果に応じて、Yes と No のいずれか一方をたどります。したがって、*b* > *max* や *c* > *max* が成立すれば（式 *b* > *max* や式 *c* > *max* を評価した値が **true** であれば）、Yes と書かれた**右側**に進み、そうでなければ No と書かれた**下側**に進みます。

▶ 本書では、**if** 文や **while** 文などの条件判定のための（ ）中の式のことを、**制御式**と呼びます。

二つに分岐するプログラムの流れの一方を通るわけですから、このようなプログラムの流れの分岐は、**双岐選択**と呼ばれます。

なお、▭ 内の矢印記号 → は、値の代入を表します。たとえば“*a* → *max*”は、

> 『変数 *a* の値を変数 *max* に代入せよ。』

という指示です。

▶ **List 1-1** の“**int** *max* = *a*;”の宣言で行われるのは、変数を作る際に値を入れる《**初期化**》で、**Fig.1-1** の“*max* = *a*;”で行われるのは、既に作られている変数に値を入れる《**代入**》です。

初期化と代入は異なるものですが、本書の解説では、厳密に区別する必要がない限り、両者をまとめて“代入”と呼ぶことにします。

p.2 に示した実行例のように、変数 a，b，c に対して 1，3，2 を入力すると、プログラムの流れはフローチャート上の**青い線**の経路をたどります。

それでは、他の値を想定して、フローチャートをなぞってみましょう。

変数 a，b，c の値が、1，2，3 や 3，2，1 であっても、最大値を求められます。また、三つの値が 5，5，5 とすべて等しかったり、1，3，1 と二つが等しくても、正しく最大値を求められます（**Fig.1-2**）。

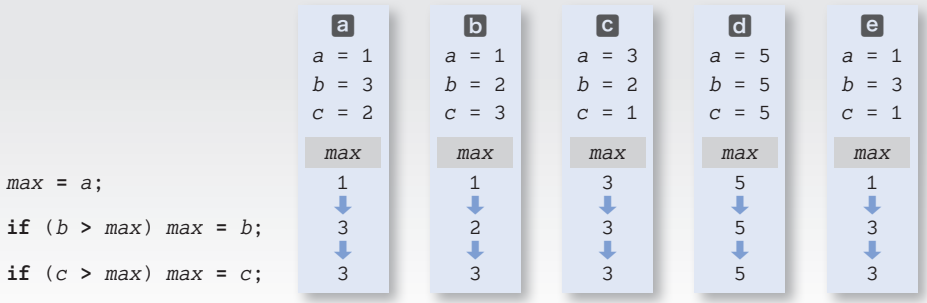

Fig.1-2　三値の最大値を求める過程における変数maxの値の変化

三つの変数 a，b，c の値が、6，10，7 や -10，100，10 であっても、フローチャート内の**青い線**をたどります。すなわち、b ＞ c ＞ a であれば、**同じ経路をたどります。**

Column 1-2　キーボードからの数値・文字列の読込み（その2）

Column 1-1 では、キーボードから **int** 型の整数値を読み込む方法を解説しました。読込み時に呼び出すメソッドは、型に応じて使い分ける必要があります。

読み込む型と、呼び出すメソッドの対応を **Table 1C-1** に示します。

Table 1C-1　Scannerクラスのnext...メソッド

メソッド	型	読み込む値
nextBoolean()	**boolean**	**true** または **false**
nextByte()	**byte**	-128 ～ +127
nextShort()	**short**	-32768 ～ +32767
nextInt()	**int**	-2147483648 ～ +2147483647
nextLong()	**long**	-9223372036854775808 ～ +9223372036854775807
nextFloat()	**float**	± 3.40282347E+38 ～ ± 1.40239846E-45
nextDouble()	**double**	± 1.79769313486231507E+378 ～ ±4.94065645841246544E-324
next()	**String**	文字列（スペース・改行などで区切られる）
nextLine()	**String**	1 行分の文字列

三値の**具体的な値**ではなく、すべての**大小関係**に対して、きちんと最大値を求められるかどうかを確認することにしましょう。確認を手作業で行うのは大変ですから、プログラムによって行うことにします。それが**List 1-2**に示すプログラムです。

List 1-2 Chap01/Max3m.java

```
// 三つの整数値の最大値を求めて表示（すべての大小関係に対して確認）

class Max3m {

    //--- a, b, cの最大値を求めて返却 ---//
    static int max3(int a, int b, int c) {
        int max = a;                        // 最大値
        if (b > max) max = b;
        if (c > max) max = c;

        return max;                         // 求めた最大値を呼出し元に返却
    }

    public static void main(String[] args) {
        System.out.println("max3(3,2,1) = " + max3(3, 2, 1));    // a>b>c
        System.out.println("max3(3,2,2) = " + max3(3, 2, 2));    // a>b=c
        System.out.println("max3(3,1,2) = " + max3(3, 1, 2));    // a>c>b
        System.out.println("max3(3,2,3) = " + max3(3, 2, 3));    // a=c>b
        System.out.println("max3(2,1,3) = " + max3(2, 1, 3));    // c>a>b
        System.out.println("max3(3,3,2) = " + max3(3, 3, 2));    // a=b>c
        System.out.println("max3(3,3,3) = " + max3(3, 3, 3));    // a=b=c
        System.out.println("max3(2,2,3) = " + max3(2, 2, 3));    // c>a=b
        System.out.println("max3(2,3,1) = " + max3(2, 3, 1));    // b>a>c
        System.out.println("max3(2,3,2) = " + max3(2, 3, 2));    // b>a=c
        System.out.println("max3(1,3,2) = " + max3(1, 3, 2));    // b>c>a
        System.out.println("max3(2,3,3) = " + max3(2, 3, 3));    // b=c>a
        System.out.println("max3(1,2,3) = " + max3(1, 2, 3));    // c>b>a
    }
}
```

実行結果

```
max3(3,2,1) = 3
max3(3,2,2) = 3
max3(3,1,2) = 3
max3(3,2,3) = 3
… 中略 …
max3(2,3,2) = 3
max3(1,3,2) = 3
max3(2,3,3) = 3
max3(1,2,3) = 3
```

最大値を求める部分は、何度も繰り返して利用されるため、本プログラムでは、独立した**メソッド**（*method*）として実現しています。網かけ部のメソッド *max3* は、受け取った三つの **int** 型仮引数 *a*，*b*，*c* の最大値を求めて、それを **int** 型の値として返します。

メソッド *max3* を呼び出すのは、**main** メソッドです。メソッド *max3* に三つの値を渡して呼び出して、メソッドから返却された値を表示します（**Column 1-3**）。

なお、結果が正しいかどうかを確認しやすくするために、すべての呼出しにおいて、最大値が 3 となるように組み合わせた値を渡しています。

*

プログラムを実行してみましょう。13 種類すべての組合せに対して 3 と表示され、最大値を正しく求めていることが確認できます。

▶ 大小関係が全部で 13 種類であることについては、**Column 1-4**（p.8）で解説しています。

JIS X0001 では《アルゴリズム》は次のように定義されています。

> **問題を解くためのものであって、明確に定義され、順序付けられた有限個の規則からなる集合。**

もちろん、いくら曖昧さのないように記述されていても、変数の値によって、解けたり解けなかったりするのでは、正しいアルゴリズムとはいえません。

ここでは、三値の最大値を求めるアルゴリズムが正しいことを、論理的に確認するとともに、プログラムの実行結果からも確認したわけです。

□ 演習 1-1

四値の最大値を求めるメソッドを作成せよ（もちろん、それをテストするプログラム＝クラスを作成しなければならない）。

```
static int max4(int a, int b, int c, int d)
```

□ 演習 1-2

三値の最小値を求めるメソッドを作成せよ。

```
static int min3(int a, int b, int c)
```

□ 演習 1-3

四値の最小値を求めるメソッドを作成せよ。

```
static int min4(int a, int b, int c, int d)
```

Column 1-3　メソッド呼出し式の評価

返却型が **void** でないメソッドは、**return** 文によって値を返却します。メソッド *max3* の場合、返却型は **int** であり、メソッドの末尾で変数 *max* の値を返却しています。

返却された値は、メソッド呼出し式の評価によって得られます。たとえば、*max*(3, 2, 1) と呼び出した場合、メソッド呼出し式 *max*(3, 2, 1) を評価した値が 3 となります。

Column 1-4　三値の大小関係と中央値

三値の大小関係の組合せ 13 種類は、**Fig.1C-3** によって列挙できます（このような木を**決定木**と呼びます）。

左端の枠から始めて、（　）内の条件が成立すれば上側の黒線を、成立しなければ下側の青線をたどっていきます。

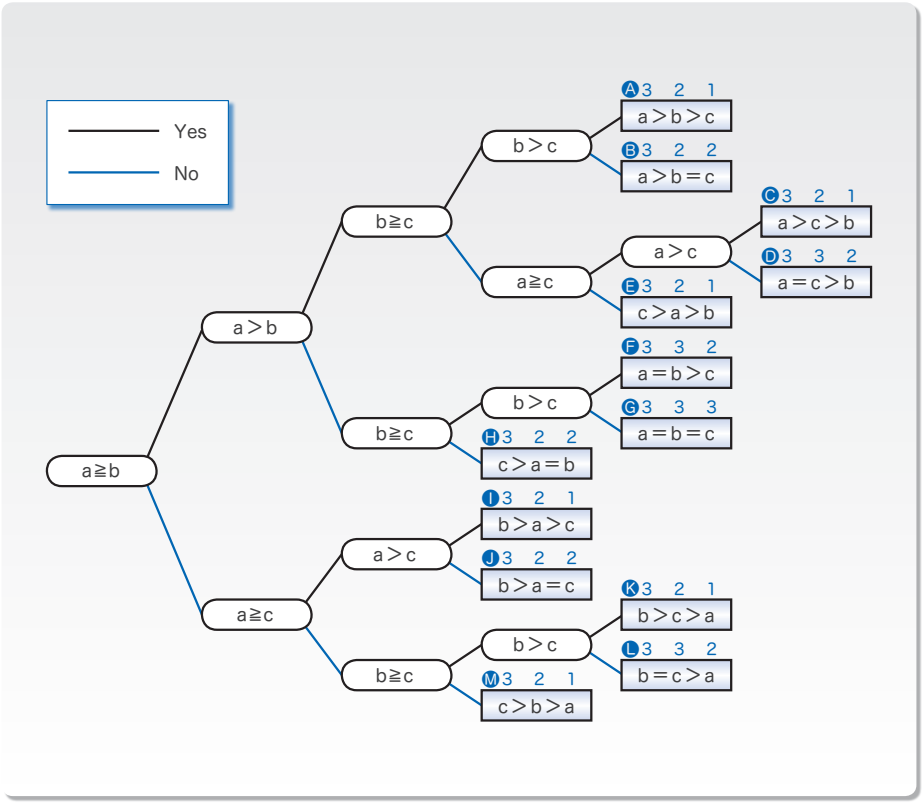

Fig.1C-3　三値の大小関係を列挙する決定木

右端の（　）内に示すのが、三つの変数 *a*，*b*，*c* の大小関係です。各枠の上に示す青い数値は、**List 1-2** のプログラムで利用した、三つの変数の値です（プログラムでは、Ⓐ，Ⓑ，…，Ⓜの 13 種類に対して、最大値を求めています）。

＊

なお、最大値・最小値とは異なり、中央値を求める手続きは、非常に複雑です（そのため、何通りものアルゴリズムが考えられます）。

List 1C-1 に示すのが、プログラムの一例です。各 `return` の横のⒶ，Ⓑ，…，Ⓜは、**Fig.1C-3** と対応しています。

＊

なお、三値の中央値を求める手続きは、『クイックソート』の改良アルゴリズム（第 6 章：p.218）などで応用されます。

List 1C-1 Chap01/Median.java

```
// 三つの整数値を読み込んで中央値を求めて表示

import java.util.Scanner;

class Median {

   static int med3(int a, int b, int c) {
      if (a >= b)
         if (b >= c)
            return b;  ←Ⓐ Ⓑ Ⓕ Ⓖ
         else if (a <= c)
            return a;  ←Ⓓ Ⓔ Ⓗ
         else
            return c;  ←Ⓒ
      else if (a > c)
         return a;     ←Ⓘ
      else if (b > c)
         return c;     ←Ⓙ Ⓚ
      else
         return b;     ←Ⓛ Ⓜ
   }

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

      System.out.println("三つの整数の中央値を求めます。");
      System.out.print("aの値：");  int a = stdIn.nextInt();
      System.out.print("bの値：");  int b = stdIn.nextInt();
      System.out.print("cの値：");  int c = stdIn.nextInt();

      System.out.println("中央値は" + med3(a, b, c) + "です。");
   }
}
```

実行例

```
三つの整数の中央値を求めます。
aの値：1↵
bの値：3↵
cの値：2↵
中央値は2です。
```

□ 演習 1-4

三値の大小関係 13 種類すべてに対して中央値を求めて表示するプログラムを作成せよ。

※ヒント：**List 1-2** と **List 1C-1** を参考にして作ること。

□ 演習 1-5

中央値を求める手続きは、以下のようにも実現できる。ただし、**List 1C-1** に示す *med3* と比較すると実行効率が悪い。その理由を考察せよ。

```
static int med3(int a, int b, int c) {
   if ((b >= a && c <= a) || (b <= a && c >= a))
      return a;
   else if ((a > b && c < b) || (a < b && c > b))
      return b;
   return c;
}
```

条件判定と分岐

読み込んだ整数値の符号（正／負／0）を判定する **List 1-3** で、プログラムの流れの分岐に対する理解を深めましょう。

List 1-3 Chap01/JudgeSign.java

```
// 読み込んだ整数値の正／負／０を判定

import java.util.Scanner;

class JudgeSign {

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

      System.out.print("整数を入力せよ：");
      int n = stdIn.nextInt();

      if (n > 0)
         System.out.println("それは正です。");  ←1
      else if (n < 0)
         System.out.println("それは負です。");  ←2
      else
         System.out.println("それは０です。");  ←3
   }
}
```

実行例1

```
整数を入力せよ：5
それは正です。
```

実行例2

```
整数を入力せよ：-5
それは負です。
```

実行例3

```
整数を入力せよ：0
それは０です。
```

Fig.1-3 に示すのは、網かけ部のフローチャートです。*n* の値が正であれば1が、負であれば2が、0 であれば3が実行されます。もちろん、実行されるのはいずれか**一つだけ**です。どれか二つが実行されたり、一つも実行されなかったり、ということはありません。

というのも、プログラムの流れは**三つに分岐している**からです。

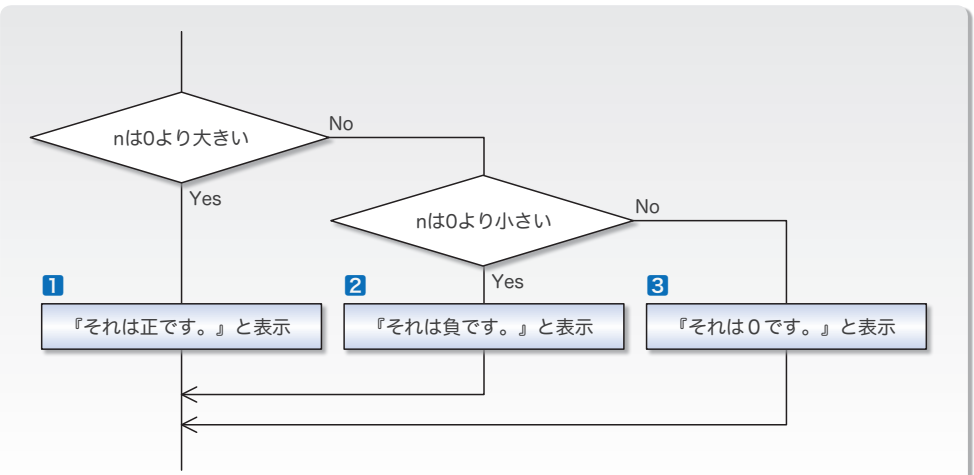

Fig.1-3 変数nの符号の判定

ここで、ちょっとした実験をします。プログラムの網かけ部を以下のように書きかえたプログラムを作ってみましょう。

リスト1

```
if (n == 1)
   System.out.println("それは１です。");  ←1
else if (n == 2)
   System.out.println("それは２です。");  ←2
else if (n == 3)
   System.out.println("それは３です。");  ←3
```

n の値が 1 であれば 1 が、2 であれば 2 が、3 であれば 3 が実行されます。

それでは、上記の **if** 文から網かけ部を削ったらどうなるでしょう。

構文は“**if**（式）文 **else if**（式）文 **else** 文”となります。これは、プログラムの流れを三つに分岐する **List 1-3** と同じ形式です。

ところが、実行の様子は異なります。n の値が 4 でも 5 でも -10 でも、とにかく 1 と 2 以外の値であれば 3 が実行されることになります。

というのも、網かけ部を削る前の リスト1 は、以下の **if** 文と同じ働きをしているからです。

```
if (n == 1)
   System.out.println("それは１です。");  ←1
else if (n == 2)
   System.out.println("それは２です。");  ←2
else if (n == 3)
   System.out.println("それは３です。");  ←3
else
   ;          // 空文（何もしない）
```

プログラムの流れは、実質的に**四つに分岐しています**。**List 1-3** の **if** 文とは構造が異なるのですから、網かけ部は削れないのです。

Column 1-5　演算子とオペランド

プログラミング言語の世界では、+ や - などの演算を行う記号を**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、大小関係の比較を行う式 a > b において、演算子は > であって、オペランドは a と b の二つです。

このように二つのオペランドをもつ演算子を**２項演算子**（*binary operator*）と呼びます。Java には、２項演算子のほかにも、オペランドが一つの**単項演算子**（*unary operator*）と、オペランドが三つの**３項演算子**（*ternary operator*）があります。

条件演算子 **? :** は、Java で唯一の３項演算子です。式 a ? b : c が評価されると、式 a を評価した値が **true** であれば b の値を生成し、**false** であれば c の値を生成します。

```
1 a = (x > y) ? x : y;
2 System.out.println((c == 0) ? "cはゼロ" : "cは非ゼロ");
```

1 では、x と y の大きいほうの値が a に代入され、2 では、変数 c の値が 0 であれば『c はゼロ』と表示され、そうでなければ『c は非ゼロ』と表示されます。

フローチャート（流れ図）の記号

問題の定義・分析・解法の図的表現である**流れ図＝フローチャート**（*flowchart*）と、その記号は、以下の規格で定義されています。

JIS X0121『**情報処理用流れ図・プログラム網図・システム資源図記号**』

ここでは、代表的な用語と記号を簡単に紹介します。

プログラム流れ図（program flowchart）

プログラム流れ図は、以下に示す記号から構成されます。

- 実際に行う演算を示す記号。
- 制御の流れを示す線記号。
- プログラム流れ図を理解し、かつ作成するのに便宜を与える特殊記号。

データ（data）

媒体を指定しないデータを表します（**Fig.1-4**）。

Fig.1-4 データ

処理（process）

任意の種類の処理機能を表します（**Fig.1-5**）。

たとえば、情報の値・形・位置を変えるように定義された演算もしくは演算群の実行、または、それに続くいくつかの流れの方向の一つを決定する演算もしくは演算群の実行を表します。

Fig.1-5 処理

定義済み処理（predefined process）

サブルーチンやモジュールなど、別の場所で定義された一つ以上の演算または命令群からなる処理を表します（**Fig.1-6**）。

Fig.1-6 処理

判断（decision）

一つの入り口といくつかの択一的な出口をもち、記号中に定義された条件の評価にしたがって、唯一の出口を選ぶ判断機能またはスイッチ形の機能を表します（**Fig.1-7**）。

想定される評価結果は、経路を表す線の近くに書きます。

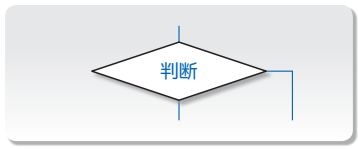

Fig.1-7 判断

ループ端（loop limit）

二つの部分から構成され、ループの始まりと終わりを表します（**Fig.1-8**）。記号の二つの部分には、**同じ名前**を与えます。

Fig.1-9 に示すように、ループの**始端記号**（前判定繰返しの場合）またはループの**終端記号**（後判定繰返しの場合）の中に、初期値（初期化）・増分・終了値（終了条件）を表記します。

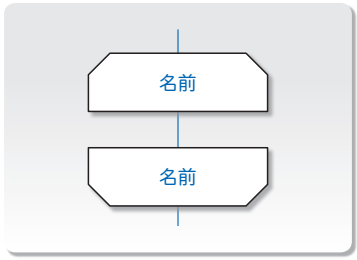

Fig.1-8 ループ端

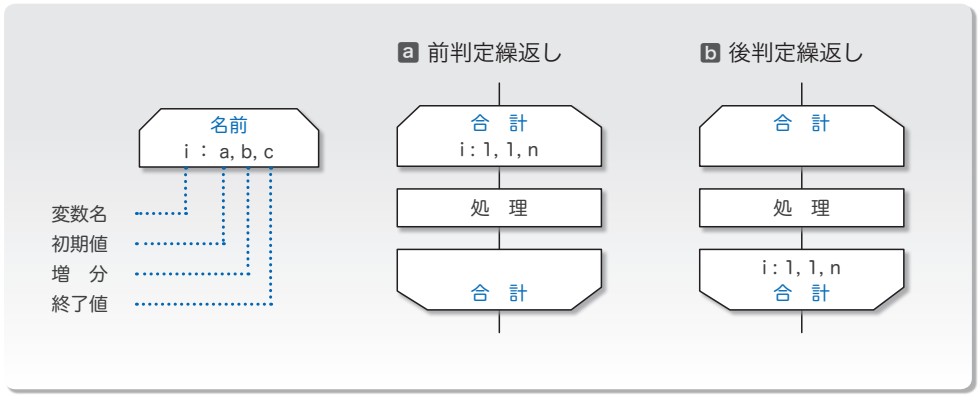

Fig.1-9 ループ端と初期化・増分・終了条件

▶ 図**a**と図**b**に示すのは、変数 i の値を 1 から n まで 1 ずつ増やしながら、『処理』を n 回繰り返すフローチャートです。なお、1，1，n の代わりに、1，2，…，n という表記を用いることもあります。

線（line）

制御の流れを表します（**Fig.1-10**）。

流れの向きを明示する必要があるときは、矢先を付けなければなりません。

なお、明示の必要がない場合も、見やすくするために矢先を付けても構いません。

Fig.1-10 線

端子（terminator）

外部環境への出口、または外部環境からの入り口を表します（**Fig.1-11**）。たとえば、プログラムの流れの**開始**もしくは**終了**を表します。

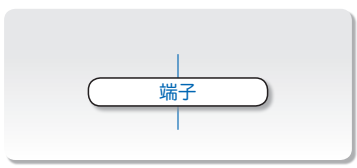

Fig.1-11 端子

この他に、並列処理、破線などの記号があります。

1-2 繰返し

本節では、プログラムの流れを繰り返すことによって実現される、単純なアルゴリズムを学習します。

1 から n までの整数の和を求める

1 から *n* までの整数の和を求めるアルゴリズムを考えましょう。

求めるのは、*n* が 2 であれば 1 + 2 で、*n* が 3 であれば 1 + 2 + 3 です。すなわち、一般的に表すと、以下の式の値を求めることになります。

```
1 + 2 + … + n
```

プログラムを**List 1-4**に、プログラム網かけ部のフローチャートを**Fig. 1-12**に示します。

List 1-4 Chap01/SumWhile.java

```
// 1, 2, …, nの和を求める（while文）

import java.util.Scanner;

class SumWhile {

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

      System.out.println("1からnまでの和を求めます。");
      System.out.print("nの値：");
      int n = stdIn.nextInt();

      int sum = 0;              // 和
      int i = 1;

      while (i <= n) {          // iがn以下であれば繰り返す
         sum += i;                 // sumにiを加える
         i++;                      // iの値をインクリメント
      }
      System.out.println("1から" + n + "までの和は" + sum + "です。");
   }
}
```

①

②

実行例

```
1からnまでの和を求めます。
nの値：5
1から5までの和は15です。
```

while 文による繰返し

while 文は、**前判定繰返し**（繰返しを続けるかどうかを処理実行の前に判定する繰返し）を行います。その形式は、以下のようになっており、制御式の評価によって得られる値が **true** である限り、文を繰り返し実行します。

```
while（制御式）文
```

なお、繰返しの対象となる文のことを、**ループ本体**と呼びます。

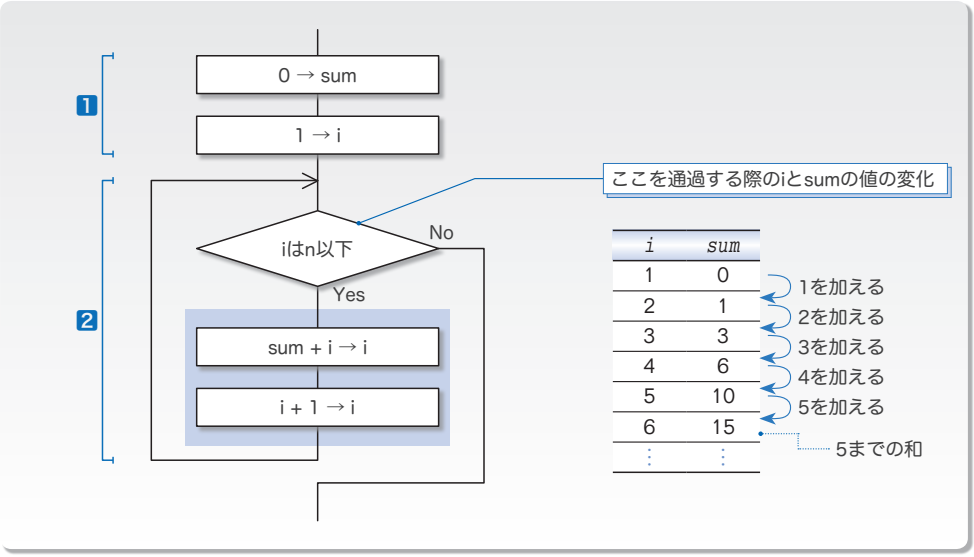

Fig.1-12 1からnまでの和を求めるフローチャートと変数の変化

プログラムとフローチャートの1と2は、以下のように働きます。

1 和を求めるための前準備です。和を格納するための変数 *sum* の値を 0 にして、繰返しを制御するための変数 *i* の値を 1 にします。

2 変数 *i* の値が *n* 以下であるあいだ、*i* の値を一つずつ増やしていきながら、ループ本体を繰り返し実行します。繰り返すのは *n* 回です。

▶ 複合代入演算子 += は右辺の値を左辺に加えます。単項演算子であるインクリメント演算子 ++ はオペランドをインクリメントします（値を一つ増やします）。

i が *n* 以下であるかどうかを判定する制御式 `i <= n`（フローチャートの◇）を通過する際の変数 *i* と *sum* の値は、図内に示した表のように変化します。プログラムと表を見比べながら理解しましょう。

制御式を初めて通過する際の変数 *i* と *sum* の値は1で設定した値です。その後、繰返しが行われるたびに変数 *i* の値はインクリメントされて一つずつ増えていきます。

変数 *sum* に入っている値は『それまでの和』であり、変数 *i* に入っている値は『次に加えることになる値』です。たとえば、*i* が 5 のときの変数 *sum* の値は『1 から 4 までの和』である 10 です（すなわち変数 *i* の値である 5 が加算される前の値です）。

なお、*i* の値が *n* を超えたときに **while** 文の繰返しが終了するため、最終的な *i* の値は、***n* ではなく *n* + 1 となる**ことに注意しましょう。

□ **演習 1-6**

List 1-4 の **while** 文終了時点における変数 *i* の値が *n* + 1 となることを確認せよ（変数 *i* の値を表示するように書きかえたプログラムを作成せよ）。

for 文による繰返し

特定の変数の値で制御する繰返しは、**while** 文ではなく **for** 文を用いたほうがスマートに実現できます。

1 から *n* までの整数の和を **for** 文で求めるように書きかえたプログラムが **List 1-5** です。

List 1-5 Chap01/SumFor.java

```
// 1, 2, …, nの和を求める（for文）

import java.util.Scanner;

class SumFor {

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

      System.out.println("1からnまでの和を求めます。");
      System.out.print("nの値：");
      int n = stdIn.nextInt();

      int sum = 0;       // 和

      for (int i = 1; i <= n; i++)
         sum += i;                // sumにiを加える

      System.out.println("1から" + n + "までの和は" + sum + "です。");
   }
}
```

実行例

```
1からnまでの和を求めます。
nの値：5
1から5までの和は15です。
```

和を求める網かけ部のフローチャートを **Fig.1-13** に示します。

六角形の**ループ端**（*loop limit*）は、繰返しの**開始点**と**終了点**を指示する記号です。同じ名前をもったループ始端とループ終端で囲まれた部分が繰り返されます。

したがって、変数 *i* の値を 1，2，3，… と、1 から *n* まで 1 ずつ増やしながら *sum* += *i*; を実行することになります。

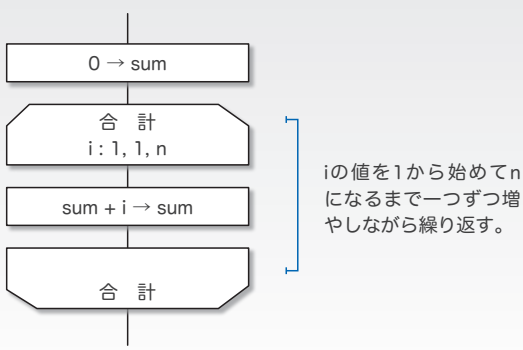

Fig.1-13　1からnまでの和を求めるフローチャート

for 文は、以下に示す形式です。

for（for 初期化部；制御式；for 更新部）文

for初期化部は、最初に一度だけ評価・実行されます。そして、制御式を評価した値が **true** である限り、文が繰り返し実行されます。その際、文を実行した直後に for 更新部が評価・実行されることになっています。

□ 演習 1-7

List 1-5 のプログラムをもとにして、たとえば *n* が 7 であれば、『1 から 7 までの和は 28 です。』と表示するのではなく、『1 + 2 + 3 + 4 + 5 + 6 + 7 = 28』と表示するプログラムを作成せよ。

□ 演習 1-8

たとえば、1 から 10 までの和は (1 + 10) * 5 によって求められる。ガウスの方法と呼ばれる、この方法を用いて和を求めるプログラムを作成せよ。

□ 演習 1-9

整数 *a*, *b* を含め、その間の全整数の和を求めて返す以下のメソッドを作成せよ。

```
static int sumof(int a, int b)
```

a と *b* の大小関係に関係なく和を求めること。たとえば *a* が 3 で *b* が 5 であれば 12 を、*a* が 6 で *b* が 4 であれば 15 を求めること。

Column 1-6　for 文について

for 文に関する文法規則は非常に複雑です。ここでは、いくつかの事項に絞って補足します。

■ for 初期化部・制御式・for 更新部のいずれも省略できる。

三つの部分は、いずれも省略できます（セミコロンは省略できません）。

■ for 初期化部で宣言された変数は、その for 文の中でのみ利用できる。

for 初期化部で宣言された変数は、**for** 文の終了とともに消えてしまいます。したがって、以下の 2 点に気をつけなければなりません。

- **for** 文の実行が終了した後にも値が必要であれば、以下のように、**for** 文に先立って変数を宣言しなければなりません。

```
int i;
for (i = 1; i <= n; i++)
   sum += i;
// for文終了後に変数iを利用する
```

- 同一メソッド内の複数の **for** 文で、同一名の変数を利用する場合は、各 **for** 文ごとに変数の宣言が必要です。

```
for (int i = 1; i <= 5; i++) sum += i;
for (int i = 1; i <= 7; i++) System.out.println(i);
```

正の値の読込み

List 1-5 のプログラム（p.16）を実行して、*n* に対して負の値である -5 を入力してみましょう。次のように表示されます。

```
1 から -5 までの和は 0 です。
```

これは、数学的に不正である以前に、感覚的にもおかしいですね。

そもそも、このプログラムでは、**正の値のみを** *n* に読み込むべきです。そのように改良したプログラムを **List 1-6** に示します。

List 1-6 Chap01/SumForPos.java

```
// 1, 2, …, nの和を求める（do文によって正の整数値のみをnに読み込む）

import java.util.Scanner;

class SumForPos {

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

      System.out.println("1からnまでの和を求めます。");

      do {
         System.out.print("nの値：");
         n = stdIn.nextInt();
      } while (n <= 0);

      int sum = 0;       // 和

      for (int i = 1; i <= n; i++)
         sum += i;                // sumにiを加える

      System.out.println("1から" + n + "までの和は" + sum + "です。");
   }
}
```

nが0より大きくなるまで繰り返す

実行例

```
1からnまでの和を求めます。
nの値：-6
nの値：0
nの値：10
1から10までの和は55です。
```

実行例に示すように、*n* の値として 0 以下の値が入力されると、再び『n の値：』と表示して再入力を促します。

その実現のために利用しているのが、以下の構文をもつ do 文です。

```
do 文 while（制御式）;
```

▶ while 文・for 文とは異なり、構文の末尾にセミコロン ; が付きます。

do 文は、処理を行った後に、繰返しを続けるかどうかの判断を行う**後判定繰返し**を実現する文です。

したがって、プログラム網かけ部のフローチャートは **Fig. 1-14** となります。

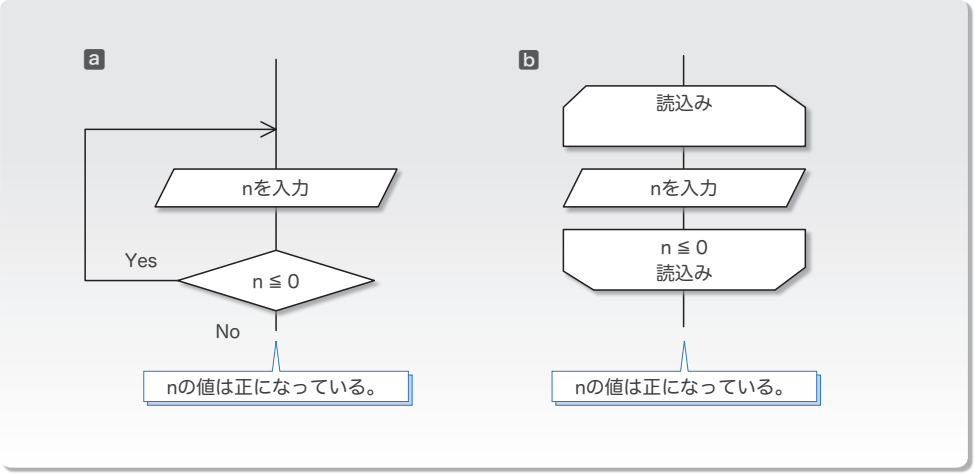

Fig.1-14 正の値の読込み

フローチャートの図aと図bは、本質的には同じです。もっとも、繰返しの条件を下側のループ端に書く図bは、**前判定繰返しとの見分けがつきにくい**ため、図aの書き方が好まれるようです。

*

変数 *n* に読み込まれた値が 0 以下である限り、ループ本体の実行が繰り返されます。そのため、**do** 文が終了したときは、*n* の値は必ず正となります。

*

do 文では、ループ本体が、必ず一度は実行されます。一方、**while** 文と **for** 文では、最初に制御式を評価した結果が **false** であれば、ループ本体は一度も実行されません。これが、前判定繰返しと後判定繰返しの大きな違いです。

□ 演習 1-10

右に示すように、二つの変数 *a*, *b* に整数値を読み込んで *b* - *a* の値を表示するプログラムを作成せよ。

なお、変数 *b* に読み込んだ値が *a* 以下であれば再入力させること。

```
aの値：6↵
bの値：6↵
aより大きな値を入力せよ！
bの値：8↵
b - aは2です。
```

□ 演習 1-11

正の整数値を読み込んで、その値の桁数を表示するプログラムを作成せよ。たとえば、135 を読み込んだら『その数は 3 桁です。』と表示し、1314 を読み込んだら『その数は 4 桁です。』と表示すること。

Column 1-7　論理演算とド・モルガンの法則

p.18 で学習した **List 1-6** は、キーボードから読み込む値を《正値》に制限するプログラムでした。**List 1C-2** に示すのは、読み込む値を《2 桁の正の整数値》に制限するプログラムです。

List 1C-2　Chap01/Digits.java

```
// ２桁の正の整数値（10〜99）を読み込む

import java.util.Scanner;

class Digits {

   public static void main(String[] args) {
      Scanner stdIn = new Scanner(System.in);
      int no;

      System.out.println("２桁の整数値を入力してください。");

      do {
         System.out.print("値は：");
         no = stdIn.nextInt();
      } while (no < 10 || no > 99);

      System.out.println("変数noの値は" + no + "になりました。");
   }
}
```

実行例

```
２桁の整数値を入力してください。
値は：5
値は：105
値は：57
変数noの値は57になりました。
```

読み込む値に制限を設けるために do 文を利用している点は、**List 1-6** と同じです。ただし、本プログラムでは、網かけ部の制御式によって、変数 *no* に読み込んだ値が 10 より小さいか、もしくは 99 より大きければ、ループ本体を繰り返すようになっています。

ここで利用している || は、論理和を求める論理和演算子です。そして、論理演算を行う、もう一つの演算子が、論理積を求める論理積演算子 && です。

これらの演算子の働きをまとめたのが、**Fig.1C-4** です。

a 論理積　両方とも真であれば真

x	*y*	*x* && *y*
true	true	true
true	false	false
false	true	false
false	false	false

b 論理和　一方でも真であれば真

x	*y*	*x* || *y*
true	true	true
true	false	true
false	true	true
false	false	false

Fig.1C-4　論理積演算子と論理和演算子

no に読み込んだ値が 5 であったとします。そうすると、式 *no* < 10 を評価した値は **true** となりますので、右オペランドの *no* > 99 を判定するまでもなく、制御式 *no* < 10 || *no* > 99 の全体も **true** となることが分かります（左オペランド *x* と右オペランド *y* の一方でも **true** であれば、論理式 *x* || *y* 全体が **true** となるからです）。

そのため、|| 演算子の左オペランドを評価した値が **true** であれば、**右オペランドの評価は行われない**ことになっています。

同様に、**&&** 演算子の場合は、左オペランドを評価した値が **false** であれば、**右オペランドの評価は行われない**ことになっています（もし一方でも **false** であれば、式全体が **false** となることが明確になるからです）。

このように、論理演算の式全体の評価結果が、左オペランドの評価の結果のみで明確になる場合に、右オペランドの評価が行われないことを**短絡評価**（*short circuit evaluation*）と呼びます。

＊

演算子 **&&** と **||** によく似た演算子として、演算子 **&** と **|** があります。演算子 **&** は論理積を求め、演算子 **|** は論理和を求めます。ただし、**&** と **|** による演算では、短絡評価が行われません。そのため、**&** と **|** は、論理演算のために使われることは少なく、整数型オペランドのビット単位の論理演算を行うために用いられるのが一般的です。

なお、排他的論理和（二つのオペランドの一方のみが真であれば **true**、そうでなければ **false**）を求める **^** 演算子による演算でも、短絡評価は行われません。そのため、この演算子も整数型オペランドのビット単位の論理演算を行うために用いられるのが一般的となっています。

＊

プログラムに戻りましょう。網かけ部の制御式を、論理補数演算子 **!** を用いて書きかえると、以下のようになります（論理補数演算子は、オペランドが **true** であれば **false** を生成し、オペランドが **false** であれば **true** を生成する、単項演算子です）。

```
!(no >= 10 && no <= 99)
```

『各条件の否定をとって、論理積・論理和を入れかえた式』の否定が、もとの条件と同じになることを、**ド・モルガンの法則**と呼びます。この法則を一般的に示すと、以下のようになります。

① *x* **&&** *y* と **!(!**x **||** **!**y**)** は等しい。
② *x* **||** *y* と **!(!**x **&&** **!**y**)** は等しい。

プログラムの制御式 *no* **<** 10 **||** *no* **>** 99 が、繰返しを続けるための**《継続条件》**であるのに対し、上記の式 **!(***no* **>=** 10 **&&** *no* **<=** 99**)** は、繰返しを終了するための**《終了条件》の否定**です。

すなわち、**Fig.1C-5** に示すイメージです。

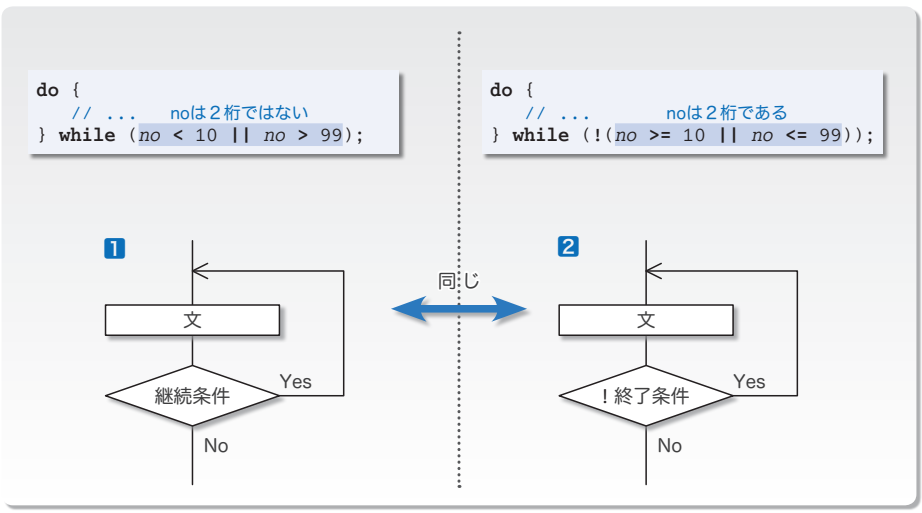

Fig.1C-5　繰返しの継続条件と終了条件

多重ループ

ここまでのプログラムは、単純な繰返しを行うものでした。繰返しの中で繰返しを行うこともできます。

そのような繰返しは、ループの入れ子の深さに応じて、**二重ループ**、**三重ループ**、…と呼ばれます。もちろん、その総称は『**多重ループ**』です。

九九の表

二重ループを用いたアルゴリズムの例として、《九九の表》を表示するプログラムを **List 1-7** に示します。

List 1-7 Chap01/Multi99Table.java

```
// 九九の表を表示

public class Multi99Table {

   public static void main(String[] args) {
      System.out.println("----- 九九の表 -----");

      for (int i = 1; i <= 9; i++) {
         for (int j = 1; j <= 9; j++)
            System.out.printf("%3d", i * j);
         System.out.println();
      }
   }
}
```

実行結果

```
----- 九九の表 -----
  1  2  3  4  5  6  7  8  9
  2  4  6  8 10 12 14 16 18
  3  6  9 12 15 18 21 24 27
  4  8 12 16 20 24 28 32 36
  5 10 15 20 25 30 35 40 45
  6 12 18 24 30 36 42 48 54
  7 14 21 28 35 42 49 56 63
  8 16 24 32 40 48 56 64 72
  9 18 27 36 45 54 63 72 81
```

表示を行う網かけ部のフローチャートを **Fig. 1-15** に示します。右側の図は、変数 *i* と *j* の値の変化を●と●で表したものです。

外側の **for** 文（行ループ）は、変数 *i* の値を 1 から 9 までインクリメントします。各繰返しは、表の 1 行目、2 行目、…、9 行目に対応します。すなわち、**縦方向の繰返し**です。

その各行で実行される内側の **for** 文（列ループ）は、変数 *j* の値を 1 から 9 までインクリメントします。これは、各行における**横方向の繰返し**です。

変数 *i* の値を 1 から 9 まで増やす《行ループ》は 9 回繰り返されます。その各繰返しで、変数 *j* の値を 1 から 9 まで増やす《列ループ》が 9 回繰り返されます。《列ループ》終了後の改行の出力は、次の行へと進むための準備です。

したがって、この二重ループでは、次のように処理が行われることになります。

- *i* が 1 のとき：*j* を 1 ⇨ 9 とインクリメントしながら 1 * *j* を表示。そして改行。
- *i* が 2 のとき：*j* を 1 ⇨ 9 とインクリメントしながら 2 * *j* を表示。そして改行。
- *i* が 3 のとき：*j* を 1 ⇨ 9 とインクリメントしながら 3 * *j* を表示。そして改行。

  … 中略 …

- *i* が 9 のとき：*j* を 1 ⇨ 9 とインクリメントしながら 9 * *j* を表示。そして改行。

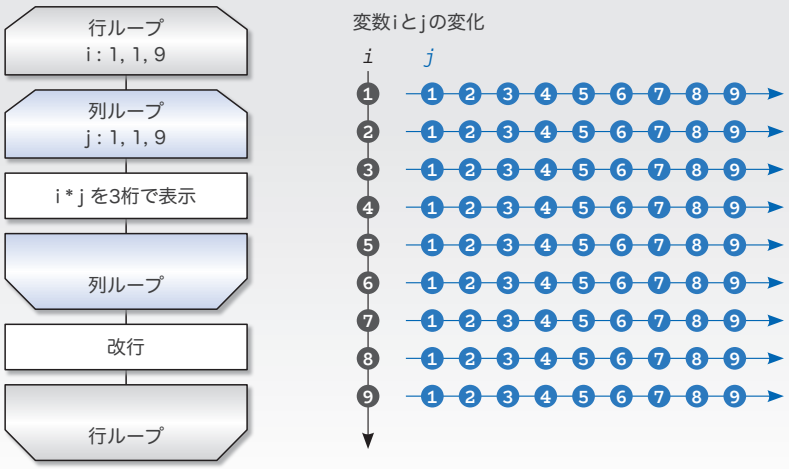

Fig.1-15 九九の表を表示するフローチャート

□ 演習 1-12

右のように、九九の表の上と左に、掛ける数を表示するプログラムを作成せよ。

表示には、縦線記号文字 '|'、マイナス記号文字 '-'、プラス記号文字 '+' を用いること。

```
   |  1  2  3  4  5  6  7  8  9
---+--------------------------
 1 |  1  2  3  4  5  6  7  8  9
 2 |  2  4  6  8 10 12 14 16 18
 3 |  3  6  9 12 15 18 21 24 27
 4 |  4  8 12 16 20 24 28 32 36
 5 |  5 10 15 20 25 30 35 40 45
 6 |  6 12 18 24 30 36 42 48 54
 7 |  7 14 21 28 35 42 49 56 63
 8 |  8 16 24 32 40 48 56 64 72
 9 |  9 18 27 36 45 54 63 72 81
```

□ 演習 1-13

九九の掛け算ではなく**足し算**を行う表を表示するプログラムを作成せよ。

□ 演習 1-14

右のように、読み込んだ段数を一辺としてもつ正方形を * 記号で表示するプログラムを作成せよ。

```
正方形を表示します。
段数は：5↵
*****
*****
*****
*****
*****
```

直角三角形の表示

二重ループを応用すると、記号文字を並べて三角形や四角形などの図形を表示することができます。**List 1-8** に示すのは、左下側が直角の三角形を表示するプログラムです。

▶ 段数 *n* には、正の値のみを読み込みます（黒網部）。

List 1-8 Chap01/TriangleLB.java

```
// 左下側が直角の三角形を表示

import java.util.Scanner;

public class TriangleLB {

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

      System.out.println("左下直角の三角形を表示します。");

      do {
         System.out.print("段数は：");
         n = stdIn.nextInt();
      } while (n <= 0);

      for (int i = 1; i <= n; i++) {
         for (int j = 1; j <= i; j++)
            System.out.print('*');
         System.out.println();
      }
   }
}
```

段数としては正値を読み込む

実行例

```
左下直角の三角形を表示します。
段数は：5
*
**
***
****
*****
```

直角三角形の表示を行う青網部のフローチャートが **Fig. 1-16** です。右側の図は、変数 *i* と *j* の変化を表したものです。

実行例のように、*n* の値が 5 である場合を例にとって、どのように処理が行われるかを考えましょう。

外側の **for** 文（行ループ）では、変数 *i* の値を 1 から *n* すなわち 5 までインクリメントします。これは、三角形の各行に対応する**縦方向の繰返し**です。

内側の **for** 文（列ループ）は、変数 *j* の値を 1 から *i* までインクリメントしながら表示を行います。これは、各行における**横方向の繰返し**です。

*

したがって、この 2 重ループは次のように動作することになります。

- *i* が 1 のとき：*j* を 1 ⇨ 1 とインクリメントしながら * を表示。そして改行。 *
- *i* が 2 のとき：*j* を 1 ⇨ 2 とインクリメントしながら * を表示。そして改行。 **
- *i* が 3 のとき：*j* を 1 ⇨ 3 とインクリメントしながら * を表示。そして改行。 ***
- *i* が 4 のとき：*j* を 1 ⇨ 4 とインクリメントしながら * を表示。そして改行。 ****
- *i* が 5 のとき：*j* を 1 ⇨ 5 とインクリメントしながら * を表示。そして改行。 *****

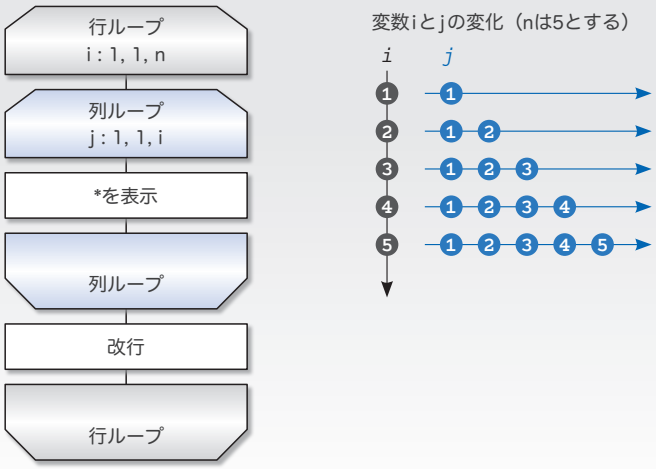

Fig.1-16 直角三角形を表示するフローチャート

すなわち、三角形を上から第1行～第n行と数えると、第i行目にi個の記号文字'*'を表示して、最終行である第n行目にはn個の記号文字'*'を表示するわけです。

□ 演習 1-15

直角三角形を表示する部分を独立させて、以下の形式のメソッドとして実現せよ。

```
static void triangleLB(int n)     // 左下側が直角の三角形を表示
```

さらに、直角が左上側、右上側、右下側の三角形を表示するメソッドを作成せよ。

```
static void triangleLU(int n)     // 左上側が直角の三角形を表示
static void triangleRU(int n)     // 右上側が直角の三角形を表示
static void triangleRB(int n)     // 右下側が直角の三角形を表示
```

□ 演習 1-16

n段のピラミッドを表示する関数を作成せよ（右は4段の例）。

```
   *
  ***
 *****
*******
```

```
static void spira(int n)
```

第i行目には`(i - 1) * 2 + 1`個の記号文字'*'を表示して、最終行である第n行目には`(n - 1) * 2 + 1`個の記号文字'*'を表示すること。

□ 演習 1-17

右のように、n段の数字ピラミッドを表示する関数を作成せよ。

```
   1
  222
 33333
4444444
```

```
static void npira(int n)
```

第i行目に表示する数字は`i % 10`によって得られる。