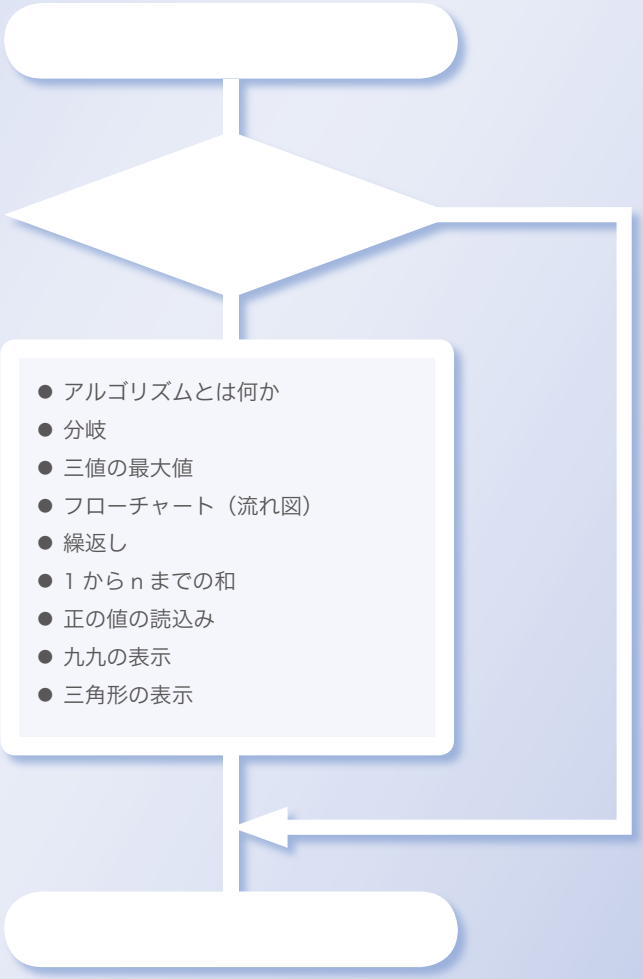
第1章

基本的なアルゴリズム



- アルゴリズムとは何か
- 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、単純なプログラムを通じて“アルゴリズム”とは何かを学習します。

■ 三値の最大値

まずは**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考えていきましょう。**List 1-1** は、変数 *a*, *b*, *c* に読み込まれた値の最大値を求めて、その値を表示するプログラムです。

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);
        int a, b, c;
        int max;                    // 最大値

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

        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。
```

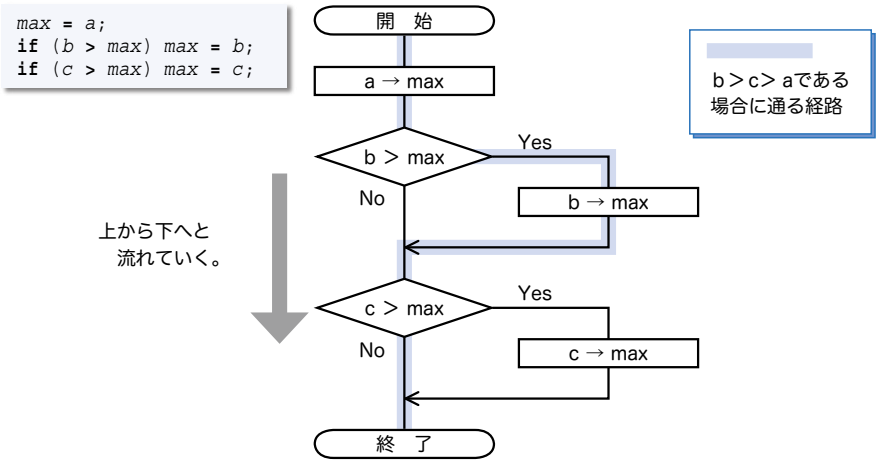
プログラムの網かけ部は、次の手順で最大値を求めます。

1. *max* に *a* の値を代入する。
2. *b* の値が *max* よりも大きければ、*max* に *b* の値を代入する。
3. *c* の値が *max* よりも大きければ、*max* に *c* の値を代入する。

この処理を表した**流れ図＝フローチャート**（*flowchart*）が **Fig. 1-1** です。黒線 ── に沿って上から下へと流れていき、□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。この分岐は**双岐選択**と呼ばれ、**if** 文の条件判定部に対応します。

Fig.1-1　三値の最大値を求めるアルゴリズムの流れ図



したがって、*b* > *max* や *c* > *max* が成立すれば 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**に進むことになります。

また、□内の矢印記号 → は代入を表します。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。

▶ フローチャートの記号は p.8 で解説します。

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、これ以外の値を想定して、フローチャートをなぞってみましょう。たとえば、変数 *a*, *b*, *c* の値が、1, 2, 3 や 3, 2, 1 であっても、正しく最大値を求められるでしょうか？　三つの値が 5, 5, 5 とすべて等しかったり、5, 3, 5 と二つが等しくても、正しく最大値を求められるでしょうか？

*

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**。

三値の**具体的な値**ではなく、すべての**大小関係**に対して、最大値を求められるかどうかを確認してみましょう。

そのプログラムが **List 1-2**（次ページ）です。*a*, *b*, *c* の具体的な値は任意ですから、どの組合せでも最大値が 3 となるようにしています。

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
```

プログラムを実行すると、13 種類すべての組合せ（**Column 1-1**）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。

なお、最大値を求める部分は何度も利用されるため、独立した**メソッド**（*method*）として実装しています（網かけ部）。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。

さて、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。

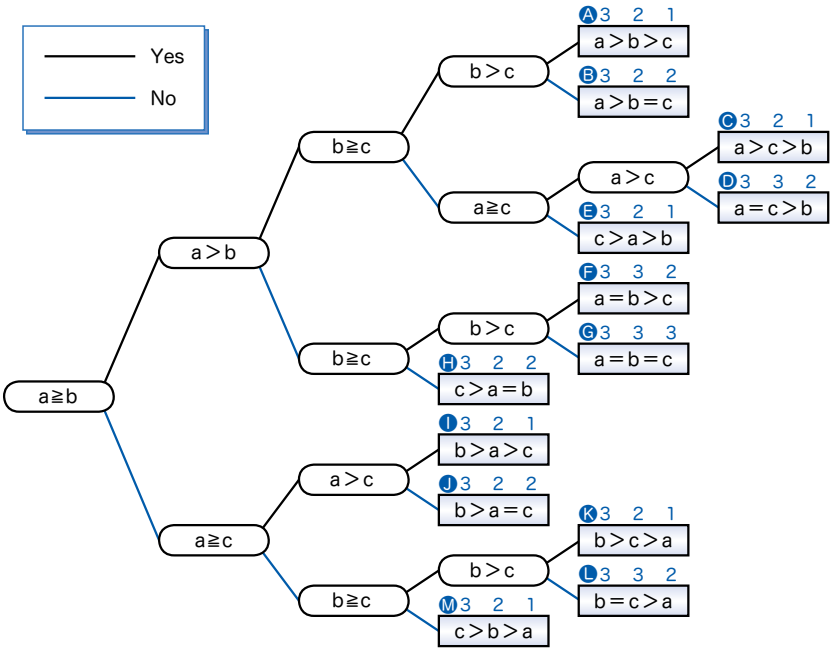
```
static int min4(int a, int b, int c, int d)
```

Column 1-1　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、下図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、枠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。右端の枠内が大小関係です。

なお、最大値・最小値とは異なり、中央値を求める手続きは複雑であり、右のようになります（各 `return` に与えられた注釈Ⓐ, Ⓑ, … は下の図と対応しています）。

```
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;       ⟵ ⓁⓂ
}
```



■ 条件判定と分岐

読み込んだ整数値の符号（正／負／0）を判定するプログラムを **List 1-3** に示します。網かけ部のフローチャートは **Fig.1-2** です。

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。");
        else if (n < 0)
            System.out.println("それは負です。");
        else
            System.out.println("それは０です。");
    }
}
```

実行例 1

```
整数を入力せよ：5
それは正です。
```

実行例 2

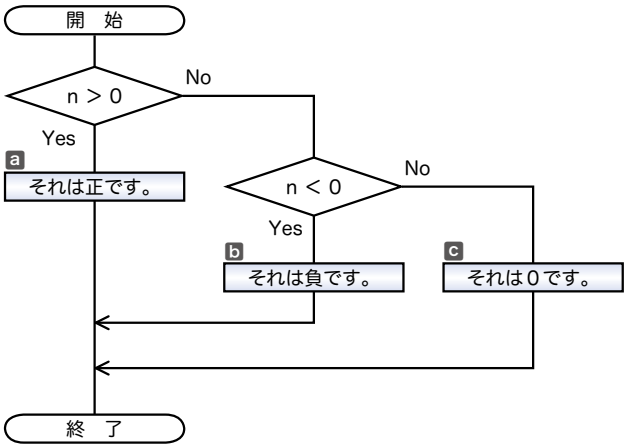
```
整数を入力せよ：-5
それは負です。
```

実行例 3

```
整数を入力せよ：0
それは０です。
```

n の値が正だと a、負だと b、0 だと c 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。a と b の両方が実行されたり、一つも実行されなかったり、ということはありません。プログラムの流れが**三つに分岐している**からです。

Fig.1-2 変数nの符号の判定



ここで、**if** 文の分岐について考えるために、ちょっと実験をします。**List 1-3** の網かけ部を以下のように書きかえてみましょう。

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
```

そうすると、*n* の値が 1 だと1、2 だと2、3 だと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、実行結果が変わってしまいます。*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が実行されてしまいます。

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
    ;              // 何もしない
```

プログラムの流れは実質的に**四つに分岐しているのです**。**List 1-3** の **if** 文とは構造が異なるのですから、網かけ部を削るようなことをしてはいけません。

■ Column 1-2　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* 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。

? : 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* を評価した値が真であれば *b* の値を生成し、偽であれば *c* の値を生成します。

■ 流れ図の記号

問題の定義、分析、解法の図的表現である**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、以下のものから構成されます。

- ■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。

■ 処理（process）

任意の種類の処理機能を表します。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。

■ 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。

■ 判断（decision）

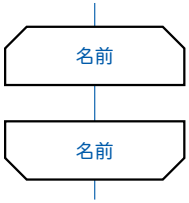
一つの入り口といくつかの択一的な出口をもち、記号中に定義された条件の評価にしたがって、唯一の出口を選ぶ判断機能またはスイッチ形の機能を表します。

想定される評価結果は、経路を表す線の近くに書きます。

■ 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。記号の二つの部分には、**同じ名前**を与えます。

初期化、増分、終了条件を、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）中に表記します。



■ 線（line）

制御の流れを表します。

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。

■ 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。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。

この他に、並列処理、破線などの記号があります。

演習 1-4

List 1-2 を参考にして、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。

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ものの、**Column 1-1** 中に示した *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;
}
```

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

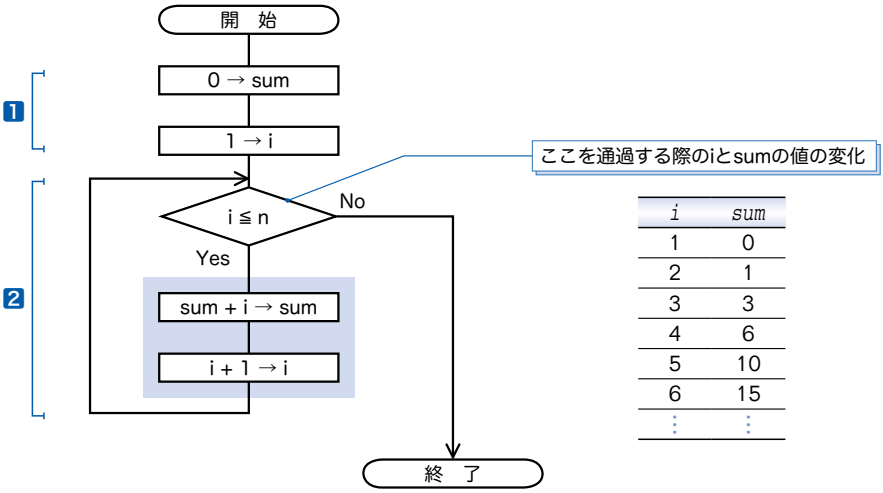
1 から *n* までの整数の和を求めるアルゴリズムを考えます。

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
```

フローチャートを **Fig. 1-3** に、プログラムを **List 1-4** に示します。

Fig. 1-3 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



i	*sum*
1	0
2	1
3	3
4	6
5	10
6	15
⋮	⋮

■ while 文による繰返し

繰返しを続けるかどうかを、処理の前に判定する**前判定繰返し**を行う **while** 文は、以下の形式です。

```
while（式）
    文
```

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();

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

        while (i <= n) {                // iがn以下であれば繰り返す
    2       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。");
    }
}
```

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。
```

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** 文では、最初に**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。この点が、**後判定繰返し**を実現する **do** 文（p.14）と大きく異なります。

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。和の格納先変数 *sum* の値を 0 にして、繰返しの制御用変数 *i* の値を 1 にします。

2 変数 *i* の値が *n* 以下である限り、*i* の値を一つずつ増やしていきながら網かけ部を繰り返し実行します。繰り返すのは *n* 回です。

i が *n* 以下であるかどうかを判定する◇を通過する際の変数 *i* と *sum* の値は、左ページの表に示すように変化します。

▶ 複合代入演算子 **+=** は**右辺の値を左辺に加えます。**単項演算子である増分演算子 **++** は**オペランドの値を一つ増やします。**

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* の値を表示するプログラムを作成せよ）。

■ for 文による繰返し

特定の変数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。

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。");
    }
}
```

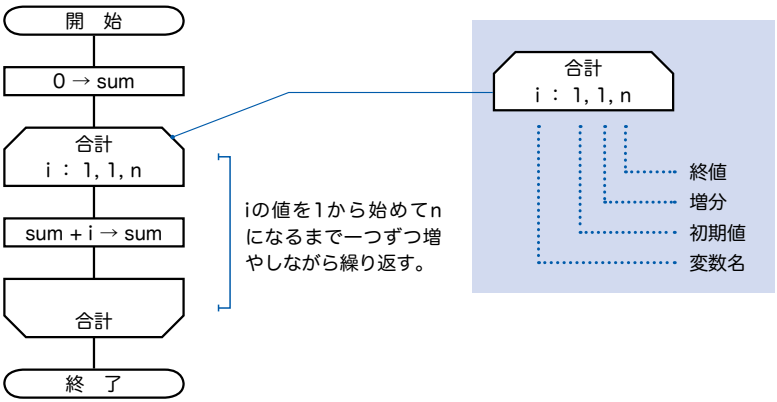
実行例

```
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.1-4** 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。

Fig.1-4 1からnまでの和を求めるフローチャート



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*; が実行されることになります。

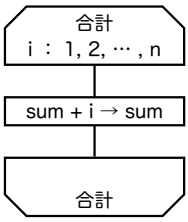
なお、繰返しを制御する式は、**Fig. 1-5** に示すように、変数の値をコンマで区切って並べ、省略部を … と記すこともできます。

for 文は、以下に示す形式です。

```
for (for初期化式 ; 式1; 式2)
    文
```

for初期化式は、最初に 1 度だけ実行されます。そして、**式**$_1$ が **true** である限り、**文**は何度も繰り返し実行されます。その際、**文**を実行した直後に**式**$_2$ が実行されることになっています。

Fig. 1-5　繰返し記号の別の表記法



▶ ***for初期化式***、**式**$_1$、**式**$_2$ のいずれも省略できます（セミコロンは省略できません）。

また、**for** 初期化式中で宣言された変数は、その **for** 文の中で**のみ**利用できるものであり、**for** 文の終了とともに消えてしまいます。

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;
```

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。』ではなくて『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

ガウスの方法（たとえば 1 から 10 までの和であれば (1 + 10) * 5 によって求める）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)
```

なお、*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 を返すこと。

■ 正の値の読込み

List 1-5 のプログラムを実行して、*n* に対して負の値である -5 を入力すると、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、数学的にも感覚的にもおかしな結果です。

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nに正の整数値のみを読み込む）

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。");
    }
}
```

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。
```

nが1以上になるまで繰り返す

実行例に示すように、*n* の値として 0 以下の値を入力すると、再び『n の値：』と表示して再入力を促します。

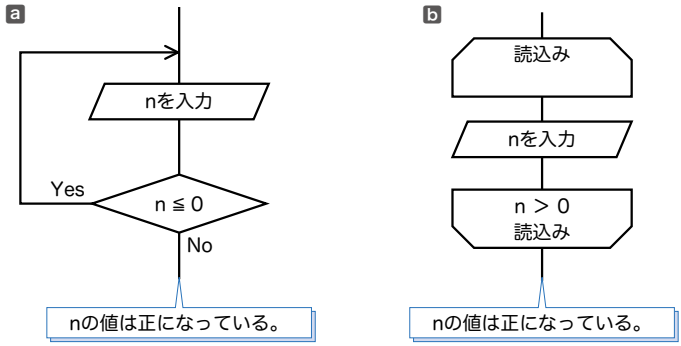
その実現のために利用しているのが、以下の構文をもつ **do** 文です。

```
do
    文
while (式);
```

▶ **while** 文や **for** 文などとは異なり、構文の末尾にセミコロン ; が付きます。

do 文は**後判定繰返し**を行いますので、プログラム網かけ部のフローチャートは**Fig.1-6** となります。

Fig.1-6 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。ただし、図bのように、繰返しの条件を下側のループ端に書く方法は、**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 以下である限り何度も繰返しが行われます。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。

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。

■ 多重ループ

ここまでは単純な繰返しでしたが、繰返しの中で繰返しを行うこともできます。

繰返し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。一般に、これらは、まとめて**多重ループ**と呼ばれます。

■ 九九の表

二重ループの例として、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. 1-7** 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、*i* の値を 1 から 9 まで一つずつ増やします。各繰返しにおいて、内側の **for** 文（列ループ）は、*j* の値を 1 から 9 まで 1 ずつ増やします。

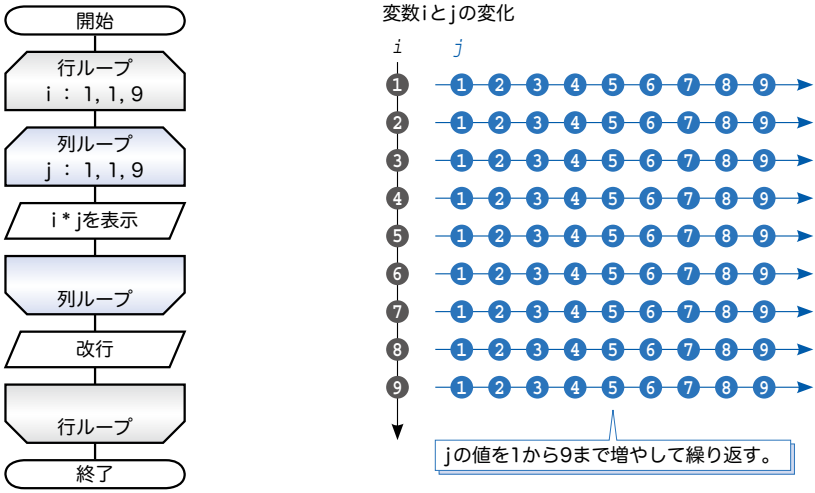
したがって、次のように処理が行われることになります。

- *i* が 1 のとき：
  j を 1 から 9 まで増やしながら 1 * *j* の値を 3 桁で表示して改行。
- *i* が 2 のとき：
  j を 1 から 9 まで増やしながら 2 * *j* の値を 3 桁で表示して改行。
- *i* が 3 のとき：
  j を 1 から 9 まで増やしながら 3 * *j* の値を 3 桁で表示して改行。

… 以下省略 …

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。

Fig. 1-7 九九の表を表示するフローチャート



その各繰返しで 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。なお、〔列ループ〕終了後の改行は、次の行へと進む準備です。

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、マイナス記号 -、プラス記号 +、縦線記号 | 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

右図のように、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

二重ループの別の例として、記号文字 * を利用して左下側が直角の三角形を表示するプログラムを **List 1-8** に示します。

直角三角形の表示を行う網掛け部のフローチャートが **Fig. 1-8** です。

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();
        }
    }
}
```

実行例

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

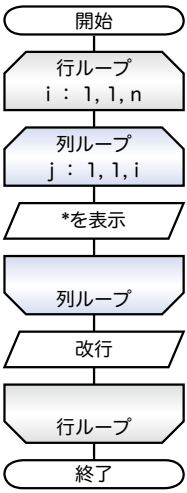
実行例のように、n の値が 5 である場合を例にとって、どのように処理が行われるかを考えましょう。

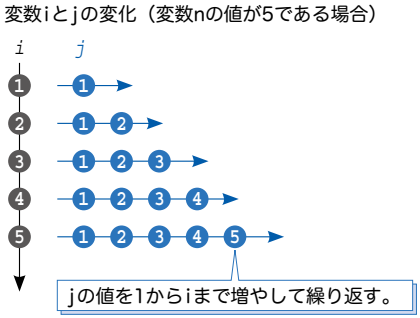
外側の **for** 文は、i の値を 1 から n まで一つずつ増やします。内側の **for** 文は、j の値を 1 から i まで一つずつ増やしながら表示を行います。したがって、次のように動作することになります。

- i が 1 のとき：j を 1 から 1 まで増やしながら * を表示して改行。 *
- i が 2 のとき：j を 1 から 2 まで増やしながら * を表示して改行。 **
- i が 3 のとき：j を 1 から 3 まで増やしながら * を表示して改行。 ***
- i が 4 のとき：j を 1 から 4 まで増やしながら * を表示して改行。 ****
- i が 5 のとき：j を 1 から 5 まで増やしながら * を表示して改行。 *****

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 * 記号を表示して、最終行である第 n 行目には n 個の * 記号を表示するわけです。

Fig. 1-8 直角三角形を表示するフローチャート





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（右図は 4 段の例）。

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

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

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

演習 1-17

右図のように、*n* 段の数字ピラミッドを表示する関数を作成せよ。

```
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。

```
   1
  222
 33333
4444444
```