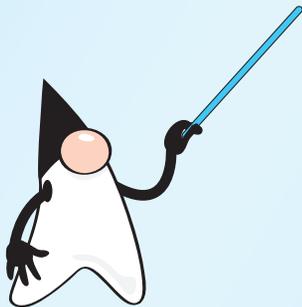
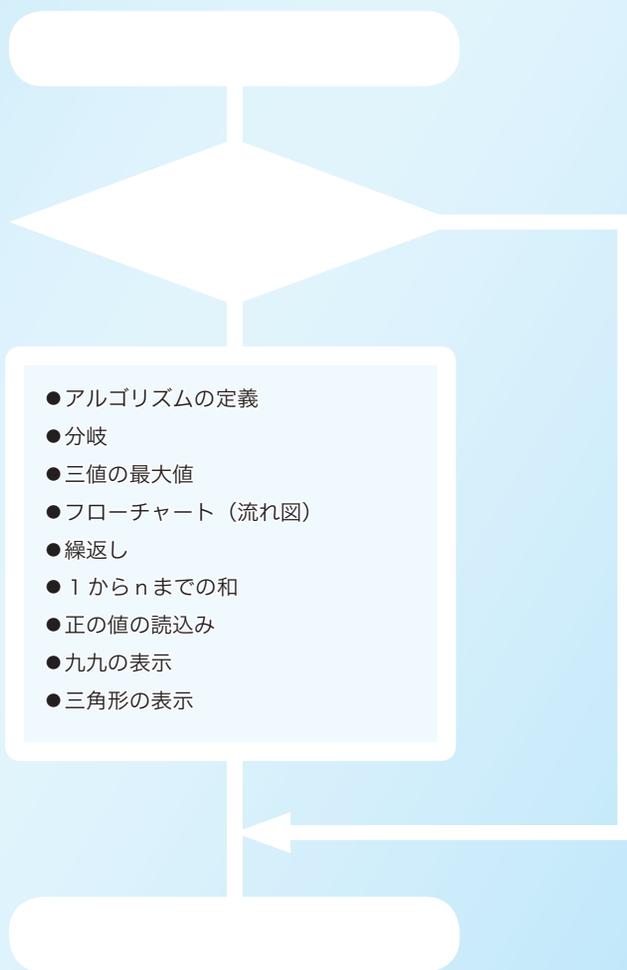


第1章

基本的なアルゴリズム



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、“そもそもアルゴリズム (algorithm) とは何か?”ということを、短くて単純なプログラムを例にとりて考えていきましょう。ここで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 に入れられるのは、キーボードから読み込んだ値です。それら三値の最大値を変数 max に求めて表示します。

まずは、プログラムを実行して、動作を確認しましょう。

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。");
```

```
    }
}
```

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。

← a, b, c の最大値を求めて max に代入

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。

三つの文が並んでいて、それらが順番に実行されます。複数の処理が順番に実行される構造は、^{じゅんじ}順次 (concatination) 構造と呼ばれますので、覚えておきましょう。

さて、①は単純な代入ですが、②と③は if 文です。() の中の式の評価結果に応じてプログラム実行の流れを変更する if 文は、^{せんたく}選択 (selection) 構造と呼ばれます。

Column 1-1

キーボードからの数値・文字列の読み込み (その1)

キーボードからの数値・文字列の読み込みには、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ういでしょう。

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();
    }
}
```

a プログラムの先頭 (クラス宣言より前) に置きます。

b main メソッドの先頭 (読み込みを行う **c** より前) に置きます。

c 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み込みを行うコード

各部分の概要は、以下のとおりです。

a `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』と『単純名』については、**Column 2-4** (p.42) で学習します。

b `main` メソッドの先頭 (読み込みを行う **c** 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。

c 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()` の評価 (**Column 1-3**: p.7) によって、キーボードから読み込んだ《値》が得られます。

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` に収まっている必要があります。また、アルファベットや記号文字などを打ち込んでではありません。

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` を変更します)。

List 1-1 では、変数 `a`, `b`, `c` の宣言の初期化子で **c** 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ます。

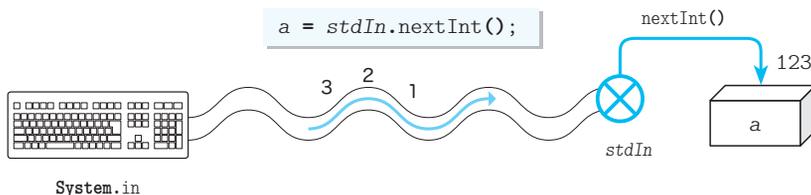


Fig.1C-2 キーボードからの読み込み

1-1

三値の最大値を求める手続きを理解しやすくするために、図で表しましょう。そのための図には、いろいろな種類があります。ここでは**流れ図**=**フローチャート** (flowchart) を使います。**Fig.1-1** に示すのが、そのフローチャートです。

- ▶ フローチャートの主要な記号は p.12 でまとめて学習します。

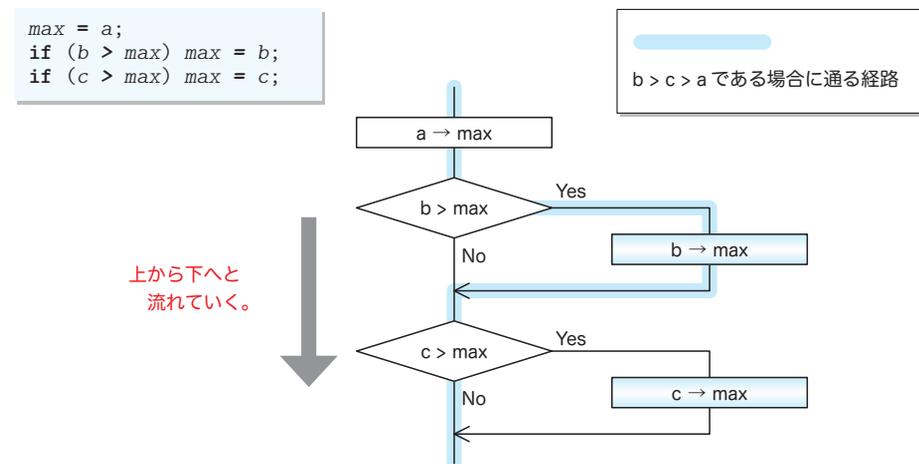


Fig.1-1 三値の最大値を求めるアルゴリズムの流れ図

プログラムの流れは、黒線 — に沿って上から下へと向かい、その過程で 内に記された処理が実行されます。

ただし、 を通過する際は、その中に記された《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。そのため、条件 $b > \max$ や条件 $c > \max$ 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 (**false** であれば) No と書かれた**下側**に進みます。

- ▶ if 文や while 文などの条件判定のために置かれる () 中の式は、**制御式**と呼ばれます。

プログラムの流れは、二つの分岐の**いずれか一方**を通るため、if 文による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 \rightarrow は、値の代入を表します。たとえば “ $a \rightarrow \max$ ” 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文脈に限り、両者をまとめて“代入”と呼んでいます。

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)。

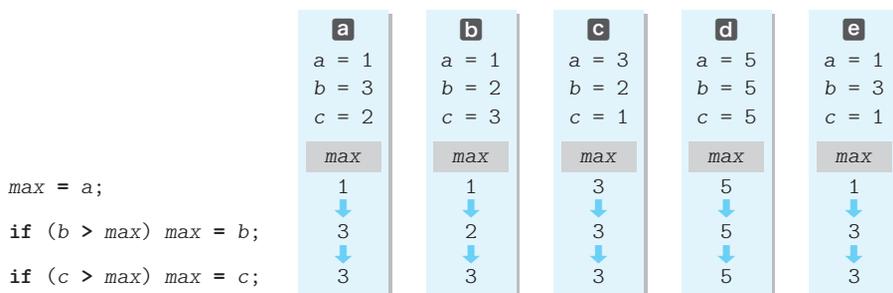


Fig.1-2 三値の最大値を求める過程における変数 max の値の変化

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、必ず同じ経路をたどります。

Column 1-2

キーボードからの数値・文字列の読み込み (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学習しました。読み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、最大値を正しく求められるかどうかを確認しましょう。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います。List 1-2 に示すのが、そのプログラムです。

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] a>b>c
        System.out.println("max3(3,2,2) = " + max3(3, 2, 2)); // [B] a>b=c
        System.out.println("max3(3,1,2) = " + max3(3, 1, 2)); // [C] a>c>b
        System.out.println("max3(3,2,3) = " + max3(3, 2, 3)); // [D] a=c>b
        System.out.println("max3(2,1,3) = " + max3(2, 1, 3)); // [E] c>a>b
        System.out.println("max3(3,3,2) = " + max3(3, 3, 2)); // [F] a=b>c
        System.out.println("max3(3,3,3) = " + max3(3, 3, 3)); // [G] a=b=c
        System.out.println("max3(2,2,3) = " + max3(2, 2, 3)); // [H] c>a=b
        System.out.println("max3(2,3,1) = " + max3(2, 3, 1)); // [I] b>a>c
        System.out.println("max3(2,3,2) = " + max3(2, 3, 2)); // [J] b>a=c
        System.out.println("max3(1,3,2) = " + max3(1, 3, 2)); // [K] b>c>a
        System.out.println("max3(2,3,3) = " + max3(2, 3, 3)); // [L] b=c>a
        System.out.println("max3(1,2,3) = " + max3(1, 2, 3)); // [M] 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

```

▶ コメント内の [A], [B], ..., [M] は、Fig.1C-4 (p.8) の A, B, ..., M に対応します。

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 max3 が、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 型の値として返却するメソッドです。

main メソッドでは、メソッド max3 に対して三つの値を実引数として与えて呼び出して、その返却値 (Column 1-3) を表示する処理を 13 回行っています。

*

計算結果が正しいかどうかを確認しやすくするために、本プログラムでは、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与えています。

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) で学習します。

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、いくら曖昧あいまい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ました。

- ▶ **JIS** (Japanese Industrial Standards) すなわち **日本工業規格**は、工業標準化法によって制定される鉱工業品に関する国の規格です。

■ 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)
```

※演習問題の解答は、ホームページ上で公開しています (p.iv)。

Column 1-3

メソッドの返却値とメソッド呼出し式の評価

メソッドは、処理を行った結果の値を、**return**文で呼出し元に返却します。メソッド `max3` の場合、返却値型は `int` 型であり、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、**Fig.1C-3** に示すように、メソッド呼出し式 `max(3, 2, 1)` を評価した値が、`int` 型の `3` となります。

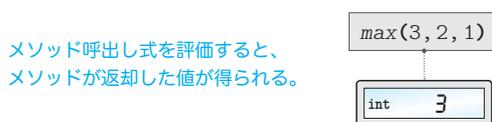


Fig.1C-3 メソッド呼出し式の評価

Column 1-4

三値の大小関係と中央値

■ 三値の大小関係の列挙

三値の大小関係の組合せが13種類であることを本文で学習しました。その組合せを列挙するのが、Fig.1C-4です。ちなみに、ここに示している図は、木の形をしていることから**決定木** (decision tree) と呼ばれます。

左端の枠 ($a \geq b$) からスタートして右側へと進みましょう。□内の条件が成立すれば**上側の線**をたどり、成立しなければ**下側の線**をたどっていきます。

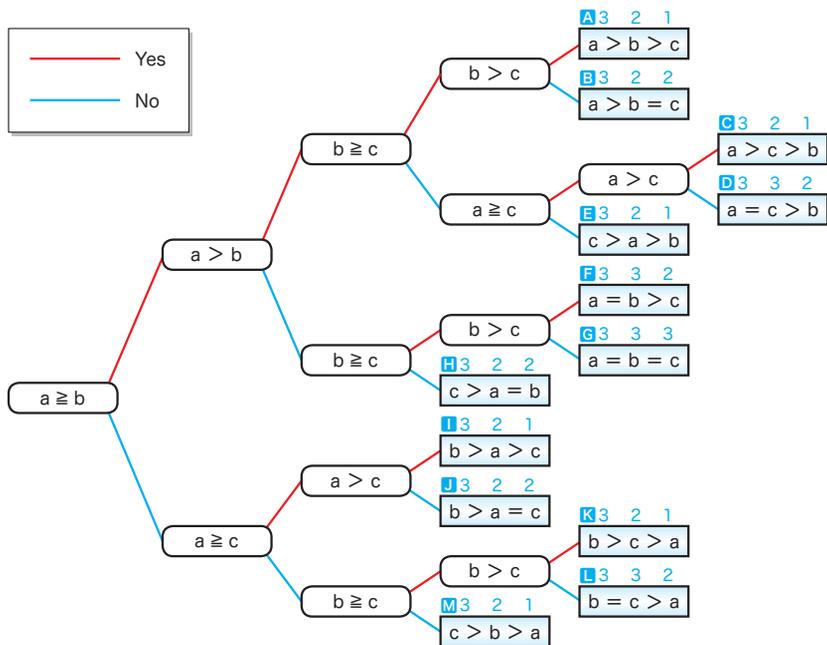


Fig.1C-4 三値の大小関係を列挙する決定木

右端の□内に示しているのが、三つの変数 a , b , c の大小関係です。その上に示している青い数値は、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、A, B, ..., M の13種類に対して、最大値を求めていました)。

■ 三値の中央値

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、数多く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 文の横の A, B, ..., M は、Fig.1C-4 と対応しています。

// 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。
```

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）などで応用されます。

演習 1-4

三値の大小関係 13 種類すべての組合せ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（うまく組み合わせて）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

List 1-3 は、読み込んだ整数値の符号（正／負／0）を判定・表示するプログラムです。本プログラムを通じて、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。変数 n の値が正であれば **1** が実行され、負であれば **2** が実行され、0 であれば **3** が実行されます。すなわち、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。プログラムの流れが三つに分岐しているからです。

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、右ページ **リスト1** のように書きかえたプログラムを作ってみましょう ("Chap01/Judge123A.java")。

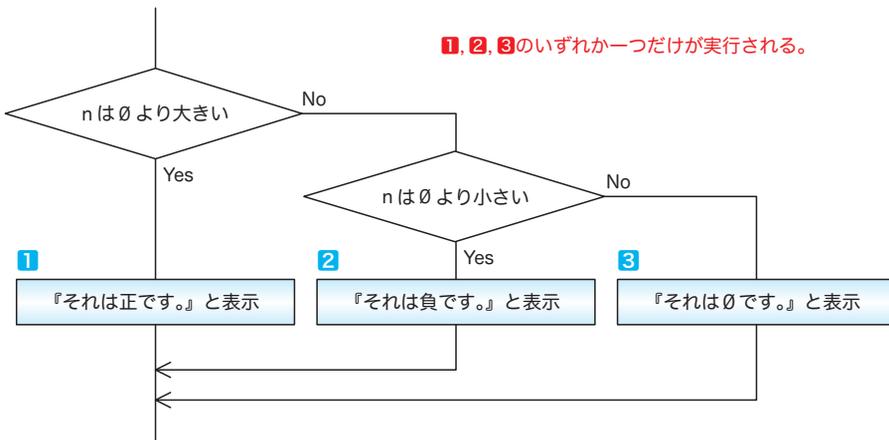


Fig.1-3 変数 n の符号の判定

```

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

```

リスト1

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。

このif文から黒網部を削った構文は“if (式) 文 else if (式) 文 else 文”となります。流れを三つに分岐するList 1-3と同じ形式です (“Chap01/Judge123B.java”)。

ところが、プログラムの流れの分岐の様子は異なります。右に示すように、 n の値が4でも-8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 (“Chap01/Judge123C.java”)。

整数を入力せよ：4
それは3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)

```

リスト2

同じ

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ため、黒網部は省略できません。

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は**演算子** (*operator*) と呼ばれ、演算の対象となる式のことは**オペランド** (*operand*) と呼ばれます。

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。

演算子は、オペランドの個数によって、以下の3種類に分類されます。

- **単項演算子** (*unary operator*) … オペランドが1個。例： $a++$
- **2項演算子** (*binary operator*) … オペランドが2個。例： $a < b$
- **3項演算子** (*ternary operator*) … オペランドが3個。例： $a ? b : c$

*

条件演算子 (*conditional operator*) $?$: 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 に代入されます。また、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の概要を学習します。

■ プログラム流れ図 (program flowchart)

プログラム流れ図には、以下に示す記号があります。

- 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 of 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 定義済み処理

■ 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) または終端記号 (後判定繰返しの場合) の中に、初期値 (初期化) と増分と終了値 (終了条件) とを表記します。

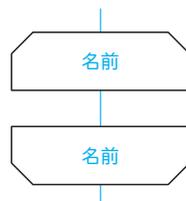


Fig.1-8 ループ端

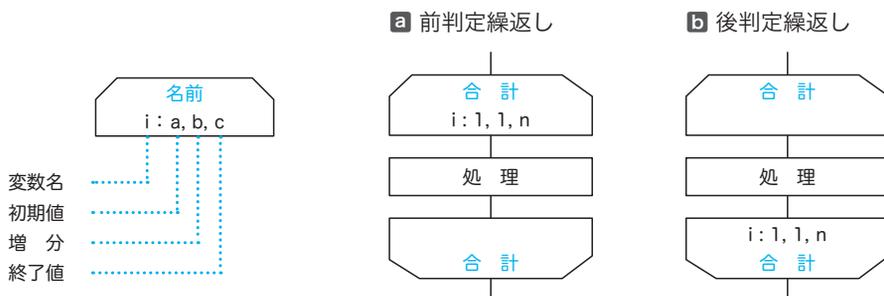


Fig.1-9 ループ端と初期値・増分・終了値

- ▶ 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

次に考えるのは、《1 から n までの整数の和を求めるアルゴリズム》です。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、右の式の値を求めます。

$$1 + 2 + \dots + n$$

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。");
    }
}
```

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

ある条件が成立しているあいだ、処理（文または命令の集まり）が繰り返し実行されるのは、**繰返し** (repetition) 構造であり、一般に**ループ** (loop) とも呼ばれます。

while 文は、繰返しを続けるかどうかを処理実行の**前**に判定するループです。このような構造は、**前判定繰返し**と呼ばれます。

以下に示すのが、**while** 文の形式です。制御式の評価によって得られる値が **true** である限り、文が繰り返し実行されます。

while (制御式) 文

なお、繰返しの対象となる文のことを、文法上、**ループ本体**と呼びます。

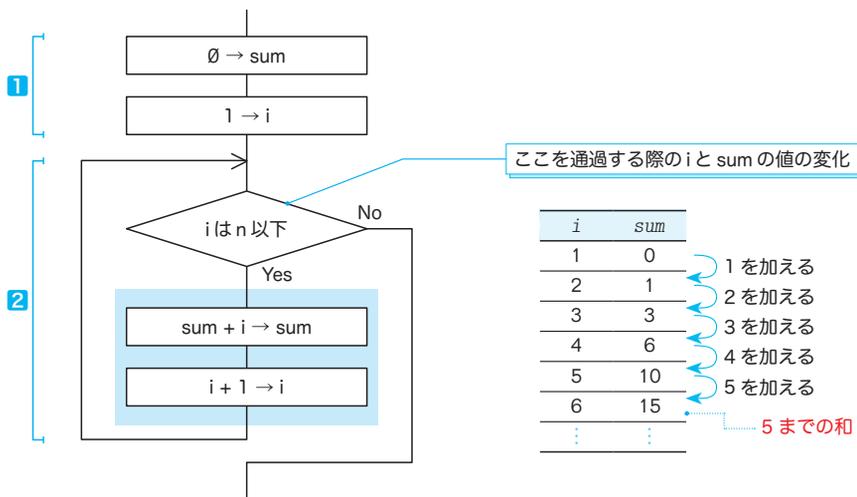


Fig.1-12 1から*n*までの和を求めるフローチャートと変数の変化

プログラムとフローチャートの**1**と**2**を理解しましょう。

- 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。
- 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て実行します。繰り返すのは *n* 回です。
 - 2項の複合代入演算子 `+=` は、右辺の値を左辺に加えます。また、単項の増分演算子 `++` 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* の値の変化をまとめたのが、図の右側の表です。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 1 と 0 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* の値は『それまでの和』であり、変数 *i* の値は『次に加える値』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 となります。

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すること）。

■ for文による繰返し

単一の変数の値でプログラムの流れを制御する繰返しは、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と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、ループ本体内の文 `sum += i;` を実行します。

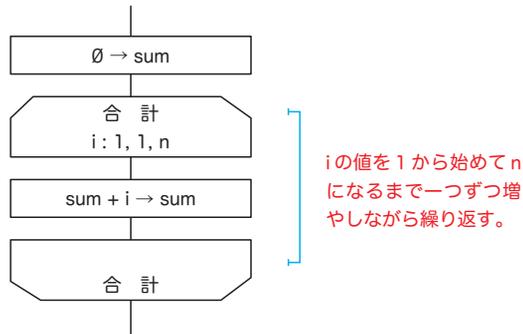


Fig.1-13 1からnまでの和を求めるフローチャート

以下に示すのが、for 文の形式です。

for (for 初期化部; 制御式; for 更新部) 文

for 初期化部は、最初に（繰返しが行われる前に）一度だけ評価・実行されます。その後、制御式を評価した値が `true` である限り、ループ本体である文が繰返し実行されます。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、そのあいだ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“無効”になります。そのため、以下の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ます。

```
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);
```

■ 正の値の読み込み

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ものです。

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。");
    }
}
```

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0より大きくなるまで繰り返す

プログラムを実行し、 n の値として0以下の値を入力すると、再び「 n の値:」と表示されて再入力促されます。

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文やfor文とは異なり、この構文の末尾にはセミコロン;が付きます。

do文は、処理を行った後、繰り返しを続けるかどうかの判断を行う後判定繰り返しを実現する文です。()の中の制御式を評価した値がtrueである限り、ループ本体の文が繰り返し実行されます。

Fig.1-14 に示すのが、プログラム網かけ部のフローチャートです。

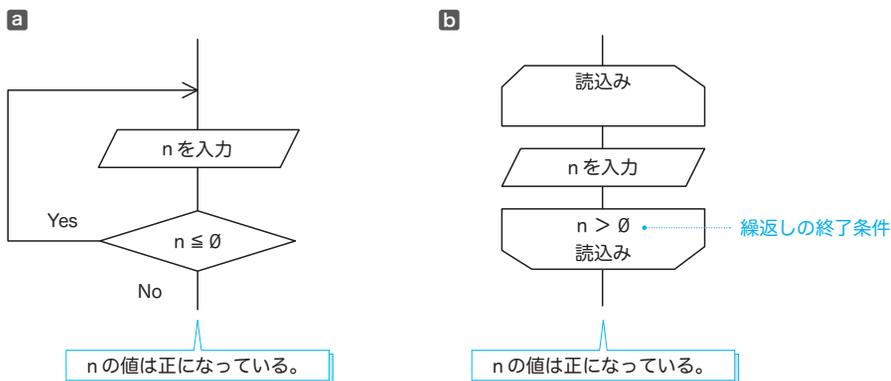


Fig.1-14 正の値の読み込み

図aと図bのフローチャートは、本質的には同じです。もっとも、繰返しの終了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付きにくいいため、図aの書き方が好まれるようです。

*

さて、本プログラムのdo文では、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文終了時のnの値は必ず正になります。

■ 前判定繰返しと後判定繰返しの相違点

前判定繰返しを行う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。一方、後判定繰返しを行うdo文では、ループ本体が必ず一度は実行されます。

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、変数bの値を再入力させること。

```
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。

■ 構造化プログラミング

単一の入り口点と単一の出口点をもつ構成要素だけを用いて、階層的に配置してプログラムを構成する手法を、**構造化プログラミング** (structured programming) といいます。構造化プログラミングでは、順次、選択、繰返しの3種類の制御の流れを利用します。

- ▶ 構造化プログラミングは、**せいこうそう 整構造プログラミング**とも呼ばれます。

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《正值》に限定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》に限定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。

これらの演算子の動きをまとめたのが、Fig.1C-5 です。

a 論理積

両方も真であれば真

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-5 論理積演算子と論理和演算子

■ 論理演算子の短絡評価

`no` に読み込んだ値が 5 であったとします。その場合、式 `no < 10` を評価した値は **true** ですから、右オペランドの `no > 99` を評価するまでもなく、制御式 `no < 10 || no > 99` の評価値が **true** になると判定できます（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 $x || y$ の評価値が **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ません。

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ません（もし一方でも **false** であれば、式全体が **false** になると判定できるからです）。

*

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。

■ ド・モルガンの法則

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（De Morgan's laws）といいます。この法則を一般的に示すと、以下ようになります。

- 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-6 に示すイメージです。

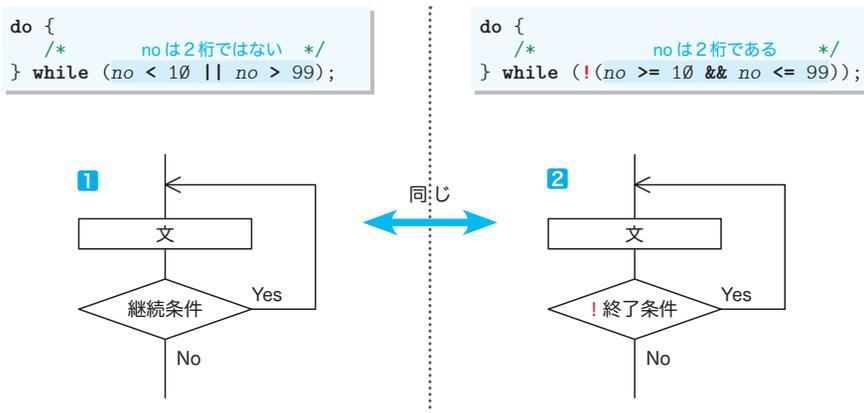


Fig.1C-6 繰返しの継続条件と終了条件

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学習しましょう。List 1-7 に示すのが、そのプログラムです。

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
```

九九の表の表示を行う網かけ部のフローチャートを Fig.1-15 に示しています。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。

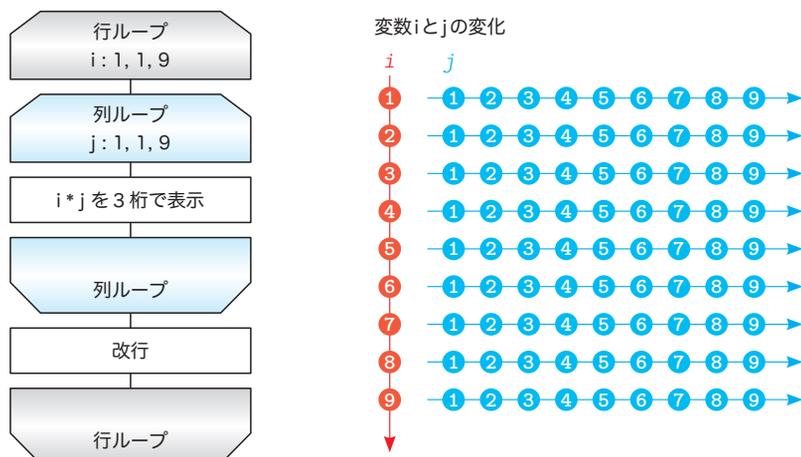


Fig.1-15 九九の表を表示するフローチャート

演習 1-12

右のように、上と左に掛ける数が付いた九九の表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。前問と同様に、表の上と左に足す数を表示すること。

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。

▶ **網かけ部**の **do** 文の働きで、変数 n に読み込む値を正の値に制限しています。

List 1-8

Chap01/TriangleLB.java

```
// 左下側が直角の二等辺三角形を表示
```

```
import java.util.Scanner;
```

```
public class TriangleLB {
```

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

```
        System.out.println("左下直角の二等辺三角形を表示します。");
```

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

段数として正値を読み込む

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

実行例

左下直角の二等辺三角形を表示します。
段数は : 5

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

直角三角形の表示を行う網かけ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、この二重ループは次のように処理が行われます。

- 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。 *****

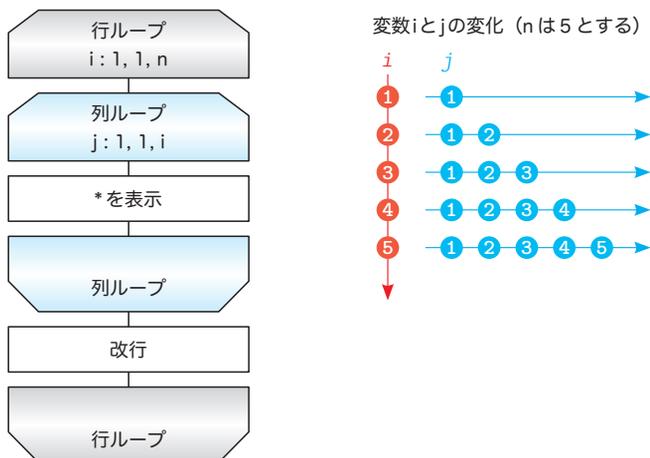


Fig.1-16 左下側が直角の二等辺三角形を表示するフローチャート

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 '*' を表示します。

演習 1-15

直角二等辺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) // 左下側が直角の二等辺三角形を表示
```

さらに、直角が左上側、右上側、右下側の二等辺三角形を表示するメソッドを作成せよ。

```
static void triangleLU(int n) // 左上側が直角の二等辺三角形を表示
```

```
static void triangleRU(int n) // 右上側が直角の二等辺三角形を表示
```

```
static void triangleRB(int n) // 右下側が直角の二等辺三角形を表示
```

演習 1-16

n 段のピラミッドを表示するメソッドを作成せよ (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メソッドを作成せよ。

```
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

```