第1章. ソースコードの見た目を整えよう

読みやすいプログラムを書くには、まず、ソースコードの見た目を整えて、きれいに書くことが大切です。
手書きで文章を書くときの動作に例えると、「丁寧な字で、位置をまっすぐ揃えて書く」という感覚に近いです。
この章では、ソースコードの見た目の整え方や、整える上で気をつけるポイントについて説明します。

1－1. インデント(字下げ)を行おう

ソースコードの見た目で最も重要なのがインデント(字下げ)です。 インデントとは、if や for など、波括弧 { } で作られるブロック内の各行を、1タブ分(半角スペース 4 個分等)下げることです。

インデントを行うことにより、プログラムの構造が把握しやすくなります。 逆に、インデントがきちんと行われていないソースは、どこからどこまでが分岐や繰り返しの範囲なのかが分かりづらく、 プログラムの内容を解析するのが難しくなります。

インデントが崩れているソースコードは、読みにくさのあまり、読む気を無くしてしまうことさえあります。 また、そのようなプログラムは、プログラムロジックの品質まであやしく思えてきてしまいます。 ブロックの構造もきちんと整えられないプログラマに、まともな論理構造が組み立てられるとは到底思えないからです。 プログラムを書く際は、必ずインデントを行うようにしましょう。

×：インデントが崩れているソースコード

```
/*
 * 指定された開始値から終了値まで、
 * 増分値にしたがって等差数列を表示する。
 */
void printSequenceOfNum(int startVal, int endVal, int zobun) {
int outputCnt

// 等差数列を表示
for (int i = startVal; i <= endVal; i += zobun) {
System.out.printf("%3d", i);

// 10個出力するごとに改行する
outputCnt++;
if ((outputCnt % 10) == 0) {
System.out.print("¥n");
}
}
}
```

どこからどこまでがforやifの範囲なのかが分かりづらい。

○：インデントがきちんとされているソースコード

```
/*
 * 指定された開始値から終了値まで、
 * 増分値にしたがって等差数列を表示する。
 */
void printSequenceOfNum(int startVal, int endVal, int zobun) {
    int outputCnt

    // 等差数列を表示
    for (int i = startVal; i <= endVal; i += zobun) {
        System.out.printf("%3d", i);

        // 10個出力するごとに改行する
        outputCnt++;
        if ((outputCnt % 10) == 0) {
            System.out.print("¥n");
        }
    }
}
```

タブ or スペース ？

インデントには、タブ文字を使用する方法と半角スペースを使用する方法があります。 インデントにタブ文字を使うか半角スペースを使うかは、各プロジェクトで決められていることもありますが、 プログラムを書く人の好みだったりすることもあります。

まともなエディタであれば、タブキーを押下したときに半角スペース 4 個分等に置き換えてインデントを行う機能を備えています。 また、タブ文字を半角スペース何個分の幅でエディタ画面上に表示するのかを指定することもできます。 よって、タブとスペースのどちらを使用してもコーディング作業に支障があるというようなことは特にありません。

ただし、半角スペースでインデントを行っていると、 たまにソースを修正する際に誤ってスペース1文字分だけずれてしまって微妙にインデントが崩れてしまうことがあります。 通常は、タブ文字を使用したほうが、インデントがきれいに整った状態を保ちやすいです。 一般的には、インデントには半角スペース 4 個分の幅のタブ文字を使用することが多いようです。

1－2. 空白(スペース)を入れよう

変数名と演算子の間や、括弧やカンマの前/後には、適宜半角スペースを挿入し、それぞれが識別しやすくなるようにしましょう。 全くスペースを入れずに記述してしまうと、変数と演算子等の区切りが分かりづらくなり、読みにくくなってしまいます。

以下に、スペースを挿入した記述例を示します。

・代入演算子、算術演算子

```
i = num + 10;
 ^ ^   ^ ^
```

・if 文、比較演算子

```
if (ret == 0) {
  ^    ^  ^  ^
```

※「(」とその直後の文字の間には空白を入れない。
「)」とその直前の文字の間には空白を入れない。

・for 文

```
for (expression1; expression2; expression3) {
   ^             ^            ^

for (int i = 0; i < arr.length; i++) {
   ^          ^               ^    ^
```

・3 項演算子

```
absNum = (num >= 0) ? num : -num;
                    ^ ^   ^ ^
```

・キャスト演算子

```
ave = (double) sum / count;
              ^
```

・配列、new 演算子

```
int[] arr = new int[10];
    ^     ^   ^
```

※ Java では、配列は型の後ろに[]が来る。
型と「[」の間には空白を入れない。

・アクセス修飾子、フィールド

```
private String message;
       ^      ^
```

・コンストラクタ

```
public HelloJava() {
                 ^

   message = "Hello, Java!";
}
```

※ コンストラクタ名と「(」の間には空白を入れない。

・メソッドの呼び出し

```
hj.print(name,  age);
                ^
```

※ ドット演算子の前後には空白を入れない。
　メソッド名の後ろに続く「(」の前には空白を入れない。

1－3. 空行を入れよう

プログラムは色々な処理が組み合わさって出来上がりますが、プログラムが長くなってくると処理の流れを 追うのが大変になってきます。

このとき、まとまった処理ごとにソースコードに空行が挿入されていると、 空行から空行までのまとまった処理をひとつの「かたまり」と認識して読むことができ、 プログラムの流れが追いやすくなります。 逆に、ソースコードに空行が全く挿入されていない場合、 どこからどこまでをひとつの処理のまとまりと見てよいのかが分かりづらくなり、読みにくくなります。

プログラムを記述する際は、関連した処理のまとまりを意識して、適宜空行を挿入しましょう。

×：空行が入っていないソースコード

```
public static void main(String[] args) {
    int startVal;
    int endVal;
    int zobun;
    // 開始値入力
    System.out.print("開始値：");
    startVal = Integer.parseInt(args[0]);
    // 終了値入力
    System.out.print("終了値：");
    endVal = Integer.parseInt(args[1]);
    // 増分値入力
    System.out.print("増分値：");
    zobun = Integer.parseInt(args[2]);
    // 数列表示
    System.out.println("----------------------------------------");
    printSequenceOfNum(startVal, endVal, zobun);
    System.out.println("----------------------------------------");
}
```

処理のまとまりを認識しづらい。

○：空行が入っているソースコード

```
    public static void main(String[] args) {
        int startVal;
        int endVal;
        int zobun;

        // 開始値入力
        System.out.print("開始値：");
        startVal = Integer.parseInt(args[0]);

        // 終了値入力
        System.out.print("終了値：");
        endVal = Integer.parseInt(args[1]);

        // 増分値入力
        System.out.print("増分値：");
        zobun = Integer.parseInt(args[2]);

        // 数列表示
        System.out.println("----------------------------------------");
        printSequenceOfNum(startVal, endVal, zobun);
        System.out.println("----------------------------------------");
    }
```

空行から次の空行までの間にある処理を一つのまとまりとして認識できる。

1－4. 波括弧｛｝の記述位置を統一しよう

Java では、波括弧 ｛｝ で関数/メソッドや if、for 文などのブロックを作りますが、波括弧を記述する位置は コーディングのスタイルによりさまざまな種類があります。

K&Rスタイル

```
if (data1 < 0) {
    System.out.println("不正");
    errorFlg = 1;
} else if (data2 < 0) {
    System.out.println("不正");
    errorFlg = 1;
} else {
    execute(data1, data2);
}
```

BSD/オールマンのスタイル

```
if (data1 < 0)
{
    System.out.println("不正");
    errorFlg = 1;
}
else if (data2 < 0)
{
    System.out.println("不正");
    errorFlg = 1;
}
else
{
    execute(data1, data2);
}
```

GUNスタイル

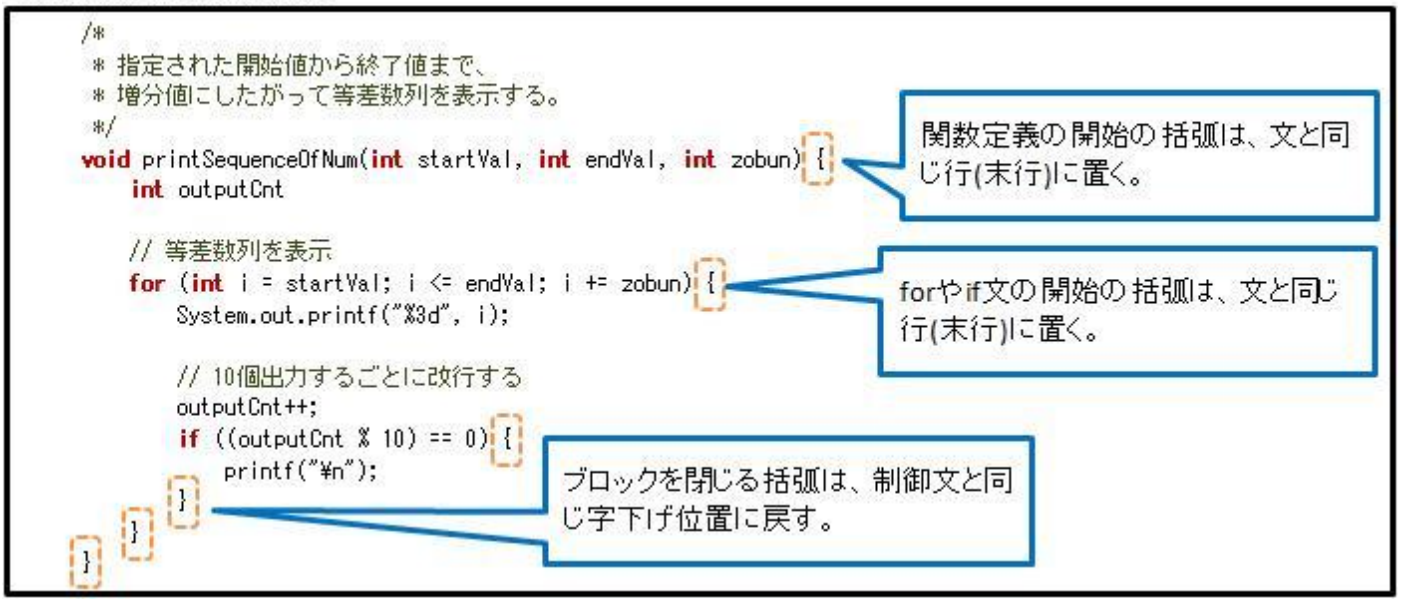
```
if (data1 < 0)
  {
    System.out.println("不正");
    errorFlg = 1;
  }
else if (data2 < 0)
  {
    System.out.println("不正");
    errorFlg = 1;
  }
else
  {
    execute(data1, data2);
  }
```

基本的に、どのスタイルを使用しても問題はありません。 しかし、ひとつのソースファイル内で複数のスタイルが混在していると、とても読みにくいソースになってしまいます。

波括弧の記述位置のスタイルは、一貫して同じものを使うようにしましょう。

Java で一般的な、K&R スタイルでのソースコードの記述例を以下に示します。

JavaのK&Rスタイルの記述例



1－5. 波括弧｛｝は省略せずに書こう

Java では、if や for 文などで処理が1文の場合は、波括弧を省略して記述することができます。 しかし、処理が1文の場合でも、波括弧は省略せずに必ず記述するようにしたほうがよいです。

波括弧を省略すると、以下のような問題があります。

- 文のインデントのみで処理部分を判別しなければならないため、やや明確さに欠ける。
- else-if 等で複数の判定が存在する場合に、波括弧のあるものとないものが混ざっていると読みにくくなる。
- ループや判定のネストが深くなると制御文の対応関係が把握しづらくなる。
- その周辺でバグがあった場合、「これは波括弧がなくて本当に大丈夫か？」「つけ忘れではないか？」等、括弧の有無による動作の違いやコードの妥当性をわざわざ検証しなければならなくなる。
- プログラムの修正の際に処理内容が複数文必要になったとき、処理を追加すると同時に記述されていなかった波括弧を付け加えないといけないため、少し面倒。
- if がネストしているときに else-if や else を記述する場合は、その else がどの if にかかっているかを把握するためにひとつひとつの if の制御の範囲を確認する必要があり、煩雑。うっかり読み誤ると難解なバグになる。

波括弧を必ず記述するようにした場合、若干行数が増えてしまいますが、上述のような問題は解消されます。 プログラムをより明確にし、余計な誤解を避け、少しでも安全にするため、波括弧は省略せずに書きましょう。

×：波括弧が省略されているソースコード

```
if (aaa > 0)
    bbbbbb = -1;

for (int i = 0; i < length; i++)
    zzzzzz(i)

-----------------------------------------------
if (aaaaa > 0)
    bbbbbb = -1;
else if (ccccc > 0) {
    ddddd = eeee;
    ffffff();
} else
    gggggg();

-----------------------------------------------
for (int i = 0; i < length; i++)
    if (aaaaa > 0)
        bbbbbb = -1;
    else if (ccccc > 0)
        if (ddddd > 0)
            while (xxxxx)
                ffffff();

        else
            ddddd = eeee;

    else
        gggggg();
```

文のインデントのみで処理部分を判別する必要がある。

・波括弧が省略されているものと記述されているものが混在していると統一性がなくなり、読みにくくなる。
・例えば「bbbbbb = -1;」の後ろに処理をもう一つ追加する場合、「if (aaaaa > 0)」に対して波括弧の記述も追加する必要があり、少し面倒。

・ループや分岐の制御文の対応が把握しづらい。
・例えば「else ddddd = eeee;」の処理が不要になったため削除する場合、そのまま削除だけすると「else gggggg();」が「if (ddddd > 0)」に対応するelseになってしまい、バグとなる。これを解消するために「else if (ccccc > 0)」に対して波括弧を追加しなければならない。インデントのみで制御範囲を判別しようとすると解決が難しいバグになる。

○：波括弧が省略せずに記述しているソースコード

```
if (aaa > 0) {
    bbbbbb = -1;
}

for (int i = 0; i < length; i++) {
    zzzzzz(i)
}
-----------------------------------------------
if (aaaaa > 0) {
    bbbbbb = -1;
} else if (ccccc > 0) {
    ddddd = eeee;
    ffffff();
} else
    gggggg();
}

-----------------------------------------------
for (int i = 0; i < length; i++) {
    if (aaaaa > 0) {
        bbbbbb = -1;
    } else if (ccccc > 0) {
        if (ddddd > 0) {
            while (xxxxx) {
                ffffff();
            }
        } else {
            ddddd = eeee;
        }
    } else {
        gggggg();
    }
}
```

波括弧により分岐/ループの制御の範囲がより明確になる。

・整然と並んでいて統一性があり、読みやすい。

・「bbbbbb = -1;」の後に処理をもう一つ追加する場合、その処理を１行追加するのみでよい。

・ループや分岐の制御の及ぶ範囲が明確。
・「else { ddddd = eeee; }」の処理を削除する場合、そのまま削除だけでよい。それ以外のif文の制御範囲などを気にする必要がない。

１－６. 一行が長くなりすぎないようにしよう

ソースコードの各行は、長くなりすぎないようにしましょう。

行が長すぎると、エディタのウィンドウ幅に収まらず、横スクロールバーを操作しなければならなくなります。コード内容を確認するためにいちいちスクロールバーを操作するのは面倒です。 一般的に、一行の長さは半角の 80～100 文字くらいまでに収めるようにしたほうがよいです。

しかし、条件式が複合条件になって if 文が長くなる場合など、行がどうしても長くなってしまうこともあります。 その場合は適宜改行を入れて読みやすくなるようにする必要があります。

改行する場合はインデントしたほうがよいです。 後続の行と明確に区別できるように2つ分のインデントを行うと読みやすくなります。

長い行に適宜改行を入れて読みやすくした例