CHAPTER

9

これまでのコードを改善する

この章では、これまでのコードをよりよくすることができるJavaプログラミングの技法と、それを実現するEclipseの機能を説明します。単によりよくするだけの機能というわけではなく、非常に重要な技法です。この技法をきちんと使いこなせることが、優秀なプログラマーになるための必須条件ともいえます。

9-1 定数とは

最初の改善対象のコードは、Sample8_5です。どこを改善するのかというと、最初に配列を定義している部分です。配列の各要素の0、2、5と記述していた箇所を改善します。

● **Sample8_5.java**

```
public class Sample8_5 {
    public static void main(String[] args) {
        int[] a = { 0, 5, 5, 0 };
        int[] b = { 2, 5, 2, 2, 5, 0 };
        int[] c = { 5, 0, 2, 0, 2, 5, 2 };

        int index = 0;
        boolean 同じ手を出していない = true;
        while (同じ手を出していない) {
            if (a[index % a.length] == b[index % b.length] &&
            b[index % b.length] == c[index % c.length]) {
                同じ手を出していない = false;
            } else {
                index++;
            }
        }
        System.out.println((index + 1) + "回目で全員が同じ手を出しました");
    }
}
```

改善後のコードのクラスをSample9_1としています。改善後のコードでは、0、2、5であった部分が、STONE、SCISSORS、PAPERに変わっています。0、2、5だけを見ても意味がわかりませんが、STONE、SCISSORS、PAPERであれば、石、はさみ、紙、つまり、グー、チョキ、パーということがわかりやすくなっています。

STONE、SCISSORS、PAPERというのは、定数として定義されています。定数は、変数と同じように参照可能です。大きな違いは、変数が変更可能な値であったのに対し、定数は変更不可能な定まった値という点です。定数はクラスのブロックに記述します。変数と区別が付きやすいように、定数は「大文字とアンダースコア（_）を組み合わせた名前」にします。また、変更できない最終的な値ということでfinal修飾子が型の前に付いています。変数にfinal修飾子が付いていると、その変数の値は変更不可能になります。privateとstatic修飾子については、のちの章で説明します。

●改善後のコード（Sample9_1.java）

```
public class Sample9_1 {

    private static final int STONE = 0;
    private static final int SCISSORS = 2;
    private static final int PAPER = 5;

    public static void main(String[] args) {
        int[] a = { STONE, PAPER, PAPER, STONE };
        int[] b = { SCISSORS, PAPER, SCISSORS, SCISSORS, PAPER,
        STONE };
        int[] c = { PAPER, STONE, SCISSORS, STONE, SCISSORS,
        PAPER, SCISSORS };

        int index = 0;
        boolean 同じ手を出していない = true;
        while (同じ手を出していない) {
            if (a[index % a.length] == b[index % b.length]
                    && b[index % b.length] == c[index % c.length]) {
                同じ手を出していない = false;
            } else {
                index++;
            }
        }
        System.out.println((index + 1) + "回目で全員が同じ手を出しました");
    }
}
```

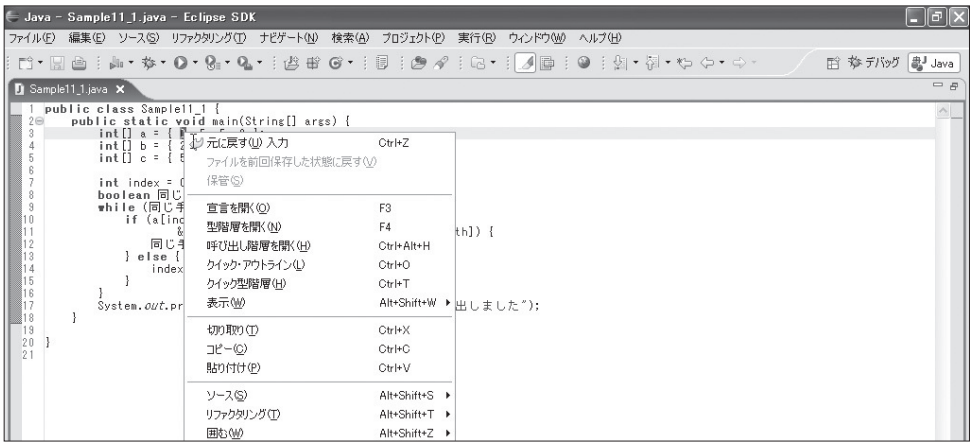
定数化を行うと、値を変更したくなった場合に定数の箇所のみを変更すればよいというメリットがあります。たとえば、商品の値段を計算するプログラムがあり、計算に消費税を使用していたとします。消費税の税率が変更になった場合に定数化を行っておけば1箇所の変更で済みますが、そうでない場合の変更は容易ではないでしょう。また、タイプミスした場合にコンパイルエラーとして検知することができるというメリットもあります。今回のサンプルの例であれば、最初の配列を{ 0, 6, 5, 0 };と、間違えて6を入力してもコンパイルが通ります。しかし定数化を行っておけば、{ATONE, PAPER, PAPER, STONE };と、最初のSをAとタイプミスした場合、ATONEはどこにも定義されていないのでコンパイルエラーとなり、間違いに気づくことができます。

今回の変更のように、プログラムの動いた結果は変更前と同じままで、内部のコードを改善することを「リファクタリング（refactoring）」と呼びます。本来は、リファクタリングの前後で動作結果が変わらないことを確認するためにテストを行います。テストについては、第19章「テスト」で説明します。

Eclipseを使用すると、修正箇所を一度の操作で定数にすることができます。

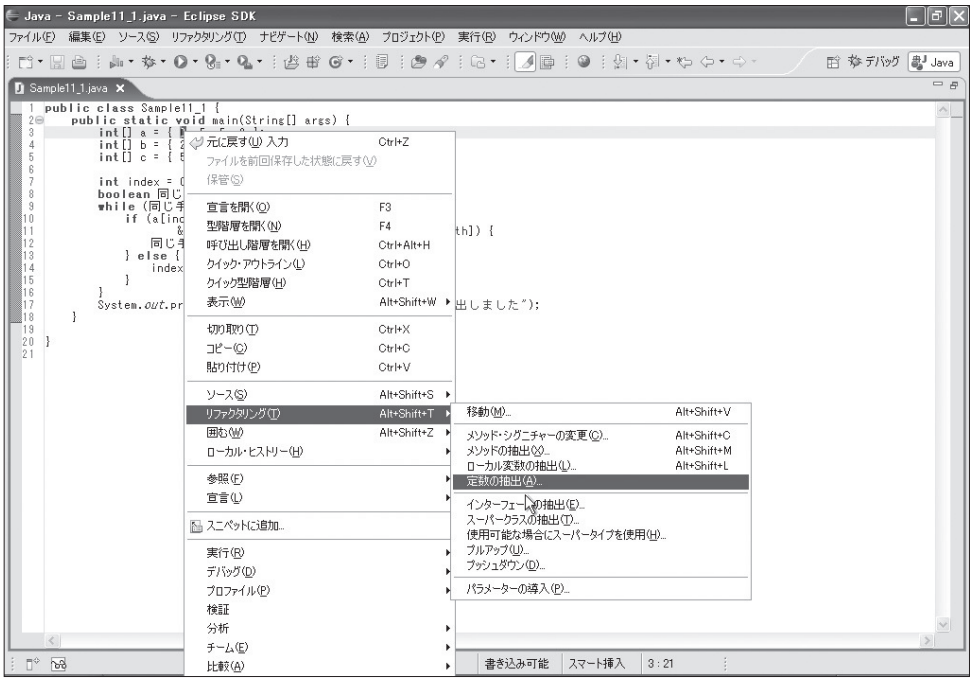
動画参照 ▶「リファクタリング（定数の抽出）」（CD-ROM/movie/const.html）

いちばん先頭の「0」を選択して、右クリックします。



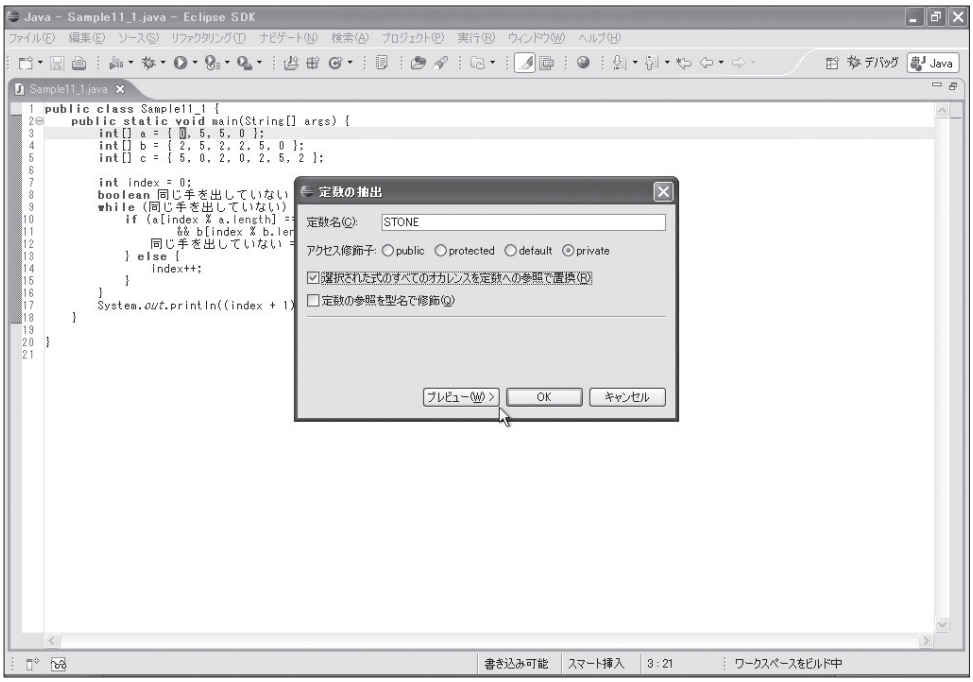
■図9-1-1　修正したい箇所を選択してからソースを右クリック

「リファクタリング＞定数の抽出」を選択します。



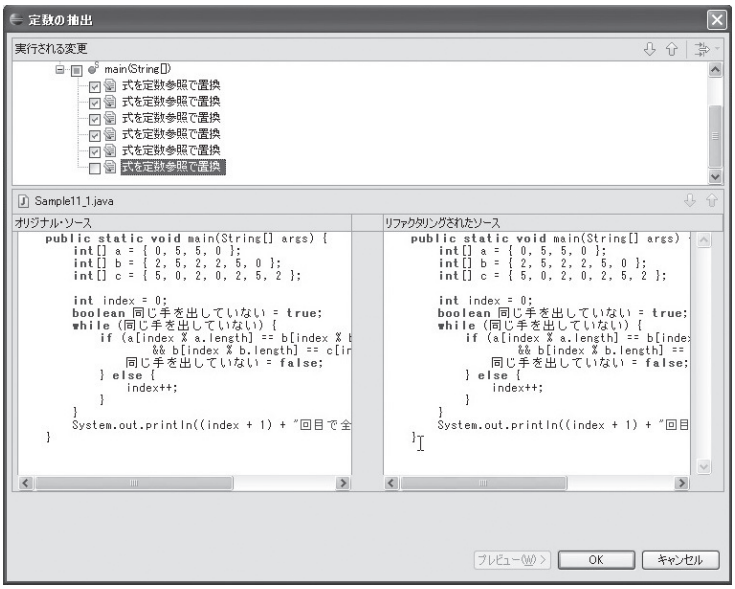
■図9-1-2 「リファクタリング＞定数の抽出」

定数名を入力します。ここでは、グーを意味する「STONE」と入力しました。そして、複数の箇所を同時に修正する場合には、「選択された式のすべてのオカレンスを定数への参照で置換」にチェックします。「プレビュー」ボタンをクリックし、どのようにソースが変更されるのか確認します。オカレンス（occurrence）とは、同じファイル内に存在する同じ記述のことです。occurrenceという単語を日本語に直訳すると「存在」になります。



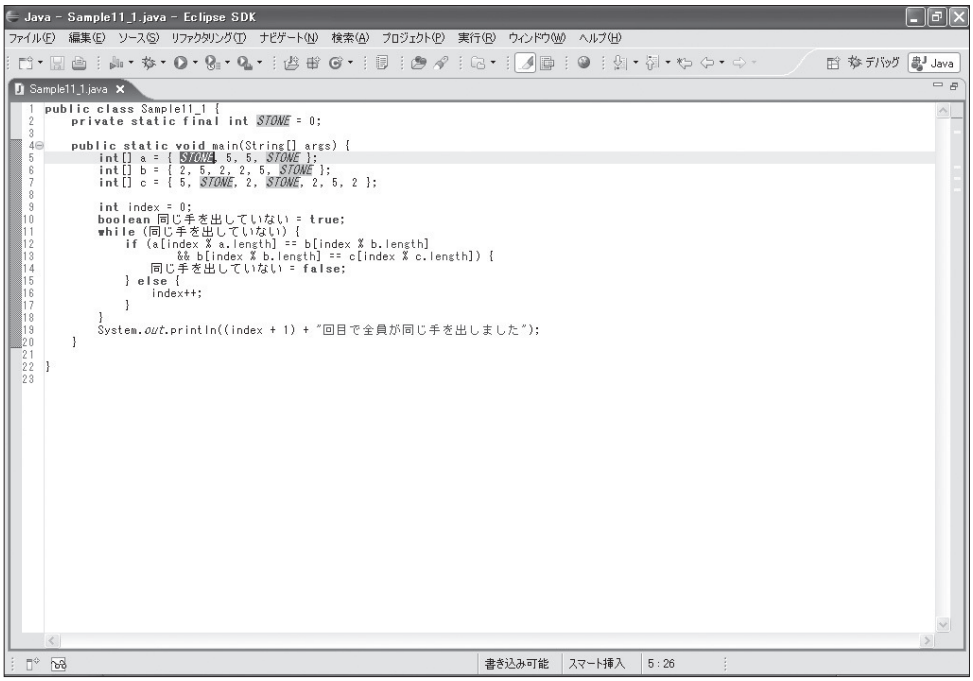
■図9-1-3　定数名の入力

プレビューのダイアログの「実行される変更」に変更がすべて表示されるので、一つ一つ選択し、変更が正しいかを確認します。正しくない変更の場合にはチェックを外します。ここでは、最後のint index = STONE;という箇所のチェックを外しました。



■図9-1-4 プレビューダイアログ

最後に「OK」ボタンをクリックすると、次の図のようにソースが変更されます。




```
public class Sample11_1 {
    private static final int STONE = 0;

    public static void main(String[] args) {
        int[] a = { STONE, 5, 5, STONE };
        int[] b = { 2, 5, 2, 2, 5, STONE };
        int[] c = { 5, STONE, 2, STONE, 2, 5, 2 };

        int index = 0;
        boolean 同じ手を出していない = true;
        while (同じ手を出していない) {
            if (a[index % a.length] == b[index % b.length]
                    && b[index % b.length] == c[index % c.length]) {
                同じ手を出していない = false;
            } else {
                index++;
            }
        }
        System.out.println((index + 1) + "回目で全員が同じ手を出しました");
    }

}
```

■図9-1-5 変更後のソース

配列内の2と5に対しても同様の操作を行えば完了です。

●定数の構文

```
static final 型 変数名 = 定数値;
```

定数はあとから値を代入することができないので、宣言と同時に値を代入する必要があります。

Column 定数の反対

定数の反対の用語は、変数ではなく「マジックナンバー」と呼ばれます。Sample8_5で言えば0、2、5のような数字です。このような数字は意味を持っていますが、プログラムを注意深く読まなければその意味がわかりません。ときには、プログラムを読んだだけでは理解できないこともあります。

次のような理由で、マジックナンバーはプログラム中に含まれないことが好ましいです。

・その数値の持つ意味がわかりづらい。
・数値を変更する場合に、複数の箇所を変更しなければならない可能性がある。

これらを避けるために、マジックナンバーが書いてある箇所を定数などに置き換えるといった処置がとられます。定数には意味のわかりやすい名前を付けるため、コードを見ただけで数値の意味を理解できるからです。定数の初期化の際に書かれる数値は2や5といった値ですが、この値はマジックナンバーとは呼びません。

また、for文を実行するときのfor(int i=0;i<length;i++)の0などは、iがループの回数を示している変数なので、ループ回数を初期化しているという意味になります。このような場合の数字はマジックナンバーとは呼びません。for(int i=3;i<length;i++)であれば、なぜ途中から実行しているのかがわからないので、これはマジックナンバーとなります。

Column 修飾子の記述順

定数の修飾子はprivate、static、finalの順で記述しましたが、実は、この修飾子の記述順を変更してもプログラムは同じ動作をします。ただ、修飾子の記述順は、Java言語仕様のセクション8.1.1、8.3.1、8.4.3で提案されています（http://java.sun.com/docs/books/jls/second_edition/html/classes.doc.html）。その順番は次の通りです。

◉表9-1-1 修飾子の記述順

記述順	修飾子
1	public
2	protected
3	private
4	abstract
5	static
6	final
7	transient
8	volatile
9	synchronized
10	native

```
private static final int STONE = 0;
static final private int SCISSORS = 2;
final private static int PAPER = 5;
```

このようなコードは、見た目が悪いだけではなく、同じ修飾子で修飾されているということが一見しただけではわからないですよね。同じ順で記述するように心がけておけば、一目でわかるコードになります。ですので、修飾子の順番を意識しながら記述するようにしましょう。そのための指標として、この表の順を参考にするのがよいでしょう。本書では、この表の順で記述します。なお、順番が異なるコードを書いた場合は、「ギブス」によって指摘されます。

9-2 メソッドとは

次の改善対象のコードはSample7_7です。

● **Sample7_7.java**

```
public class Sample7_7 {

    public static void main(String[] args) {
        int[] array1 = { 9, 8, 7, 6, 5 };
        int[] array2 = { 1, 2, 3 };

        System.out.println("array1の要素");
        for (int val : array1) {
            System.out.println(val);
        }
```

同じコード

```
        System.out.println("array2の要素");
        for (int val : array2) {
            System.out.println(val);
        }

        System.out.println("array2をarray1に代入すると");
        array1 = array2;

        System.out.println("array1の要素");
        for (int val : array1) {
            System.out.println(val);
        }
        System.out.println("array2の要素");
        for (int val : array2) {
            System.out.println(val);
        }

        System.out.println("array2の2番目の要素を5にすると");
        array2[1] = 5;

        System.out.println("array1の要素");
        for (int val : array1) {
            System.out.println(val);
        }
        System.out.println("array2の要素");
        for (int val : array2) {
            System.out.println(val);
        }
    }
}
```

同じコード

同じコード

コードを見ると、同じことを記述している箇所があります。このような重複している箇所を先ほどの定数のように1箇所にまとめることができます。1箇所にまとめたコードのクラスをSample9_2としています。

● Sample9_2.java

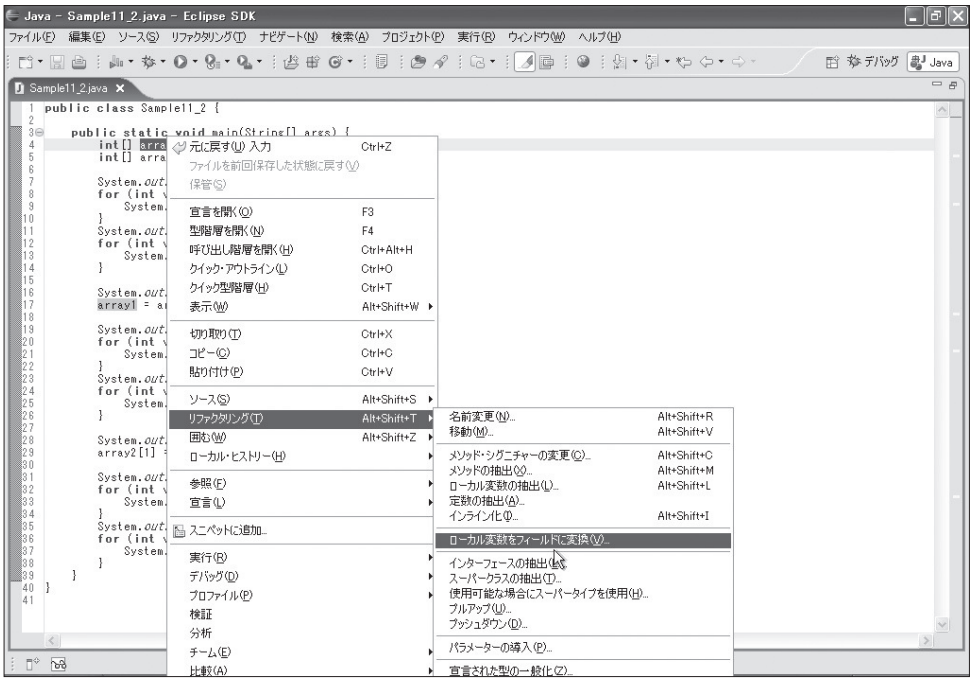
```
public class Sample9_2 {

    private static int[] array1s;
    private static int[] array2s;

    public static void main(String[] args) {
        array1s = new int[]{ 9, 8, 7, 6, 5 };
```

```
        array2s = new int[]{ 1, 2, 3 };

        printArrays();

        System.out.println("array2をarray1に代入すると");
        array1s = array2s;

        printArrays();

        System.out.println("array2の2番目の要素を5にすると");
        array2s[1] = 5;

        printArrays();
    }

    private static void printArrays() {
        System.out.println("array1の要素");
        for (int val : array1s) {
            System.out.println(val);
        }
        System.out.println("array2の要素");
        for (int val : array2s) {
            System.out.println(val);
        }
    }
}
```

mainメソッドを記述したときと同じようにコードの上から順番に記述することもできますが、Eclipseのリファクタリング機能を使うと簡単にメソッドを分割することができます。

まず、array1、array2をクラス変数にします。クラス変数については後述します。また、クラス変数はフィールドの一種です。フィールドについても後述します。Eclipse上では「フィールド」という表現がされています。

array1を選択したあと、ソースを右クリックして「リファクタリング＞ローカル変数をフィールドに変換」を選択します。

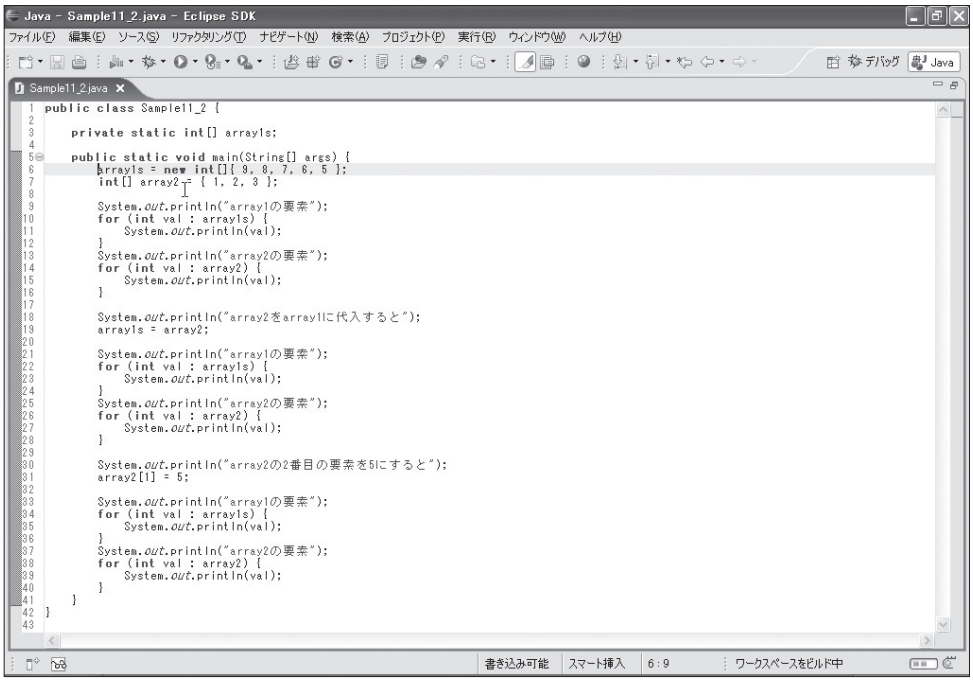


■図9-2-1 「リファクタリング>ローカル変数をフィールドに変換」を選択

「ローカル変数をフィールドに変換」ダイアログが表示されます。フィールド名の入力などができますが、今回はそのまま「OK」ボタンをクリックします。

■図9-2-2 「ローカル変数をフィールドに変換」ダイアログ

すると、変数がmainメソッドのブロックの上に移動します。

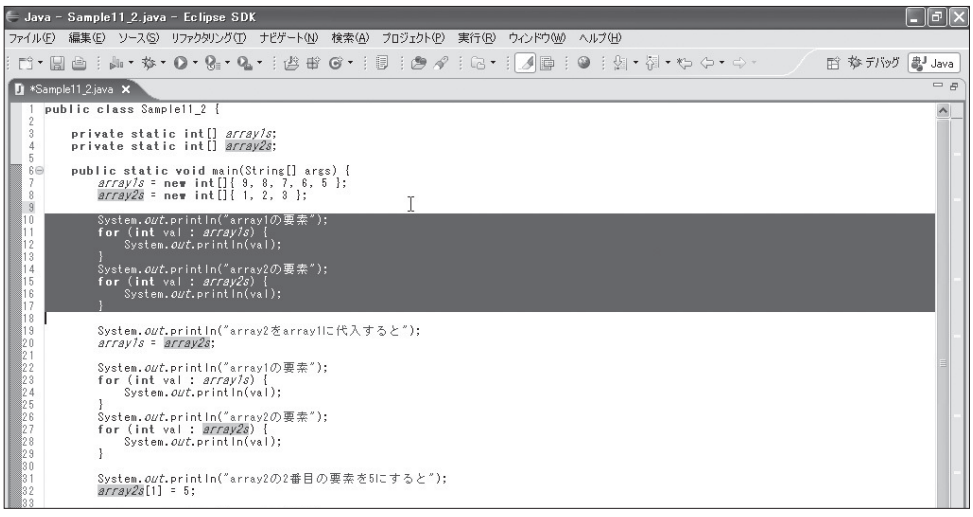



```
public class Sample11_2 {

    private static int[] array1s;

    public static void main(String[] args) {
        array1s = new int[]{ 9, 8, 7, 6, 5 };
        int[] array2 = { 1, 2, 3 };

        System.out.println("array1の要素");
        for (int val : array1s) {
            System.out.println(val);
        }
        System.out.println("array2の要素");
        for (int val : array2) {
            System.out.println(val);
        }

        System.out.println("array2をarray1に代入すると");
        array1s = array2;

        System.out.println("array1の要素");
        for (int val : array1s) {
            System.out.println(val);
        }
        System.out.println("array2の要素");
        for (int val : array2) {
            System.out.println(val);
        }

        System.out.println("array2の2番目の要素を5にすると");
        array2[1] = 5;

        System.out.println("array1の要素");
        for (int val : array1s) {
            System.out.println(val);
        }
        System.out.println("array2の要素");
        for (int val : array2) {
            System.out.println(val);
        }
    }
}
```

■図9-2-3　リファクタリング後

同様に、array2もクラス変数にします。
次は、同じコードの部分を共通化します。共通化したい部分のコードを選択します。

動画参照 ▶「リファクタリング（メソッドの抽出）」（CD-ROM/movie/method.html）




```
public class Sample11_2 {

    private static int[] array1s;
    private static int[] array2s;

    public static void main(String[] args) {
        array1s = new int[]{ 9, 8, 7, 6, 5 };
        array2s = new int[]{ 1, 2, 3 };

        System.out.println("array1の要素");
        for (int val : array1s) {
            System.out.println(val);
        }
        System.out.println("array2の要素");
        for (int val : array2s) {
            System.out.println(val);
        }

        System.out.println("array2をarray1に代入すると");
        array1s = array2s;

        System.out.println("array1の要素");
        for (int val : array1s) {
            System.out.println(val);
        }
        System.out.println("array2の要素");
        for (int val : array2s) {
            System.out.println(val);
        }

        System.out.println("array2の2番目の要素を5にすると");
        array2s[1] = 5;
```

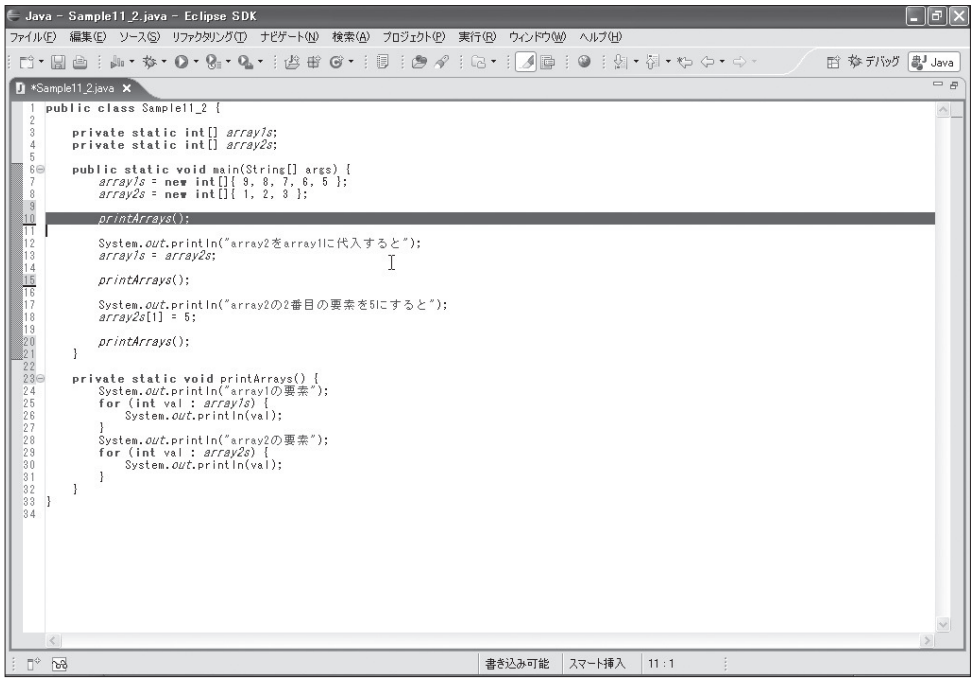
■図9-2-4　共通化したい箇所を選択

次に、「Alt」＋「Shift」＋「M」キーを押します。「M」は「Method」の頭文字です。ソースを右クリックして、「リファクタリング＞メソッドの抽出」を選択しても同じことができます。

「メソッドの抽出」ダイアログが表示されるので、抽出するメソッド名を入力し、「OK」ボタンをクリックします。先ほどと同じように、プレビューでリファクタリング後のコードを確認することもできます。

■図9-2-5 「メソッドの抽出」ダイアログ

すると、ソースを選択した部分と同じ箇所がprintArrays();に置き換わります。



■図9-2-6 リファクタリング後のコード

同じことを記述していた箇所が、printArrays();に置き換わりました。printArrays()がなにを意味しているのかというと、mainメソッドの下に定義されています。定義する方法はmainメソッドに似ています。

●メソッド定義の構文

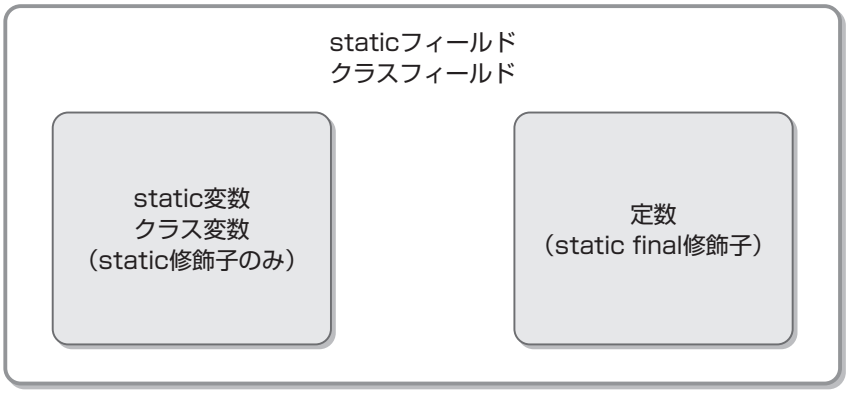
```
void メソッド名() {
    ステートメント;
}
```

メソッドのブロック内を除けば、mainメソッドとの違いは、メソッド名の違いと、メソッド名のうしろのカッコ内にString[] argsがないという点です。String[] argsについては第9章3節の「メソッドの引数」で説明します。publicという修飾子がprivateに変わっていますが、この違いについては、第10章「クラスを利用する」で説明します。

また、printArraysメソッドの中では、array1sとarray2sの変数を参照しています。別のメソッド、つまりmainメソッドの中に定義された変数を直接参照することはできないので、クラスブロックに変数を定義しています。第9章1節で説明した定数との違いは、final修飾子がないことです。final修飾子が付かず、static修飾子が付く変数を、「クラス変数」または「static変数」と呼びます。クラス変数はあとから値を代入することが可能なので、宣言と同時に値の初期化をしてもしなくても動作します。そして、「定数」と「クラス変数」を合わせて「クラスフィールド」、または「staticフィールド」と呼びます。

●クラス変数定義の構文

```
private static 型 クラス変数名;
```



■図9-2-7 「定数」「static変数」「クラスフィールド」の関係

9-3 メソッドの引数

もう少し、Sample9_2を見てみましょう。

● Sample9_2のprintArraysメソッド

```
private static void printArrays() {
    System.out.println("array1の要素");
    for (int val : array1) {
        System.out.println(val);
    }
    System.out.println("array2の要素");
    for (int val : array2) {
        System.out.println(val);
    }
}
```

printArraysメソッドを見ると、array1とarray2に対して同じような処理を行っています。これも1つのメソッドにまとめることができます。そのように修正したコードがSample9_3です。

● Sample9_3.java

```
public class Sample9_3 {
    private static int[] array1 = { 9, 8, 7, 6, 5 };

    private static int[] array2 = { 1, 2, 3 };

    public static void main(String[] args) {
        printArrays();

        System.out.println("array2をarray1に代入すると");
        array1 = array2;

        printArrays();

        System.out.println("array2の2番目の要素を5にすると");
        array2[1] = 5;

        printArrays();
    }

    private static void printArrays() {
        printArray("array1", array1);
```

```
        printArray("array2", array2);
    }

    private static void printArray(String arrayName, int[] array) {
        System.out.println(arrayName + "の要素");
        for (int val : array) {
            System.out.println(val);
        }
    }
}
```

①

printArraysメソッドからprintArrayメソッドを呼び出すようにしています。

まず、printArrayメソッドを見てみましょう。メソッドの中で、arrayNameとarrayという変数を参照しています。この変数はメソッドの中で定義されていませんし、クラス変数としても定義されていません。どこを参照しているのかというと、printArraysメソッドのメソッド名のうしろのカッコに定義している変数（①）を参照しています。この変数を「引数」と呼びます。この引数の値は、printArraysメソッドから渡されます。

printArraysメソッドの中身も見てみましょう。最初にprintArrayメソッドを呼び出し、引数に"array1"という文字列と、array1という配列を渡しています。そうすると、printArrayが実行されます。そのときの引数の値が文字列の "array1"と、クラス変数のarray1です。printArrayの実行が終了するとprintArraysに戻り、2つ目のprintArrayが同様に実行されます。

第9章2節では引数の説明を省略しましたが、引数を考慮すると、次のようにしてメソッドを定義することになります。

● メソッド定義の構文

```
private static void メソッド名(型 引数名1, 型 引数名2, ...) {
    ステートメント;
}
```

9-4 メソッドの戻り値

次は、Sample7_5を改善します。

● Sample7_5.java

```
public class Sample7_5 {
```

```
    public static void main(String[] args) {
        int[] points;
        points = new int[30];

        points[0] = 90;
        points[1] = 62;
        points[2] = 76;
        for (int i = 3; i < 15; i++) {
            points[i] = 75;
        }
        for (int i = 15; i < 30; i++) {
            points[i] = 70;
        }

        double sumPoint = 0;
        for (int point : points) {
            sumPoint = sumPoint + point;
        }
        double averagePoint = sumPoint / points.length;
        System.out.println("このクラスの平均点は" + averagePoint + "点です");

        for (int i = 0; i < points.length; i++) {
            System.out.println("出席番号" + (i + 1) + "番は、" +
            points[i] + "点です");
        }
    }
}
```

先ほどと同じように、配列を初期化している部分をメソッドとし、それを呼び出すようにします。コードの次の部分を選択し、先ほどと同じ手順でメソッドにします。

●メソッドにする箇所

```
points[0] = 90;
points[1] = 62;
points[2] = 76;
for (int i = 3; i < 15; i++) {
    points[i] = 75;
}
for (int i = 15; i < 30; i++) {
    points[i] = 70;
}
```

最後のfor文もprintPointsというメソッドにしましょう。手順は先ほどと同様です。

最後に平均値を求めている部分も別メソッドにします。コードの次の部分を選択し、平均値を計算するという意味のcalculateAverageというメソッド名にします。

●平均値を求めている部分

```
double sumPoint = 0;
for (int point : points) {
    sumPoint = sumPoint + point;
}
double averagePoint = sumPoint / points.length;
```

calculateAverageメソッドを見ると、いままでのメソッドとは少し異なる点があります。

● calculateAverageメソッド

```
private static double calculateAverage(int[] points) {
    double sumPoint = 0;
    for (int point : points) {
        sumPoint = sumPoint + point;
    }
    double averagePoint = sumPoint / points.length;
    return averagePoint;
}
```

staticのうしろがvoidではなく、doubleとなっている点がいままでと違うところです。もう1点は、メソッドの最後にreturnというステートメントがあることです。

まず、return文から説明します。return文は、メソッドの呼び出し元に、メソッド本体から特定の情報を返す場合に使用します。具体的には、calculateAverageメソッドから平均値という情報を、呼び出し元であるmainメソッドに返しています。この返す情報のことを「戻り値」と呼びます。戻り値は、double averagePoint = calculateAverage(points);とすることによって、mainメソッド内ではaveragePointとして参照することができるようになります。

次に、staticのうしろがvoidではなくdoubleとなっているという点ですが、これは、メソッドが情報を返すときに、どの型の情報を返しているのかということを表しています。今回は平均値をdouble型で返しているので、このようになっています。一方、voidは戻り値がないことを表しています。

return文は、メソッドの最後だけでなく、途中にも記述することができ、break文のような使い方もできます。void型の場合returnのうしろに値を書かず、セミコロン（;）で終わります。具体的にはreturn;とします。

戻り値も考慮すると、メソッド定義は次のようになります。

●メソッド定義の構文

```
private static 戻り値の型 メソッド名(型 引数名1, 型 引数名2, ...) {
    ステートメント;
    return ステートメント;
}
```

全体のコードは次のようになりました。

●改善後のコード（Sample9_4.java）

```
public class Sample9_4 {

    public static void main(String[] args) {
        int[] points = new int[30];

        initializeArray(points);

        double averagePoint = calculateAverage(points);
        System.out.println("このクラスの平均点は" + averagePoint + "点です");

        printPoints(points);
    }

    private static double calculateAverage(int[] points) {
        double sumPoint = 0;
        for (int point : points) {
            sumPoint = sumPoint + point;
        }
        double averagePoint = sumPoint / points.length;
        return averagePoint;
    }

    private static void printPoints(int[] points) {
        for (int i = 0; i < points.length; i++) {
            System.out.println("出席番号" + (i + 1) + "番は、" +
            points[i] + "点です");
        }
    }

    private static void initializeArray(int[] points) {
        points[0] = 90;
        points[1] = 62;
```

点数を格納する配列の定義
配列の初期化
平均値の計算
平均値の表示
全点数の表示

```
        points[2] = 76;
        for (int i = 3; i < 15; i++) {
            points[i] = 75;
        }
        for (int i = 15; i < 30; i++) {
            points[i] = 70;
        }
    }
}
```

mainメソッドの中を見ると、次のようになっていることがわかります。

・点数を格納する配列の定義
・配列の初期化
・平均値の計算
・平均値の表示
・全点数の表示

このプログラムが、なにを行っているのかという、プログラムの全体像がわかりますよね。メソッドを分割することによって、このようにコードを見ただけでプログラムがなにを行っているのかをわかるようにできます。これは強力な技法です。

メソッドへ分割するときの指針の一つには、「ある処理の塊に対して名前を付けることができるなら、別メソッドに切り出す」ということがあげられます。

9-5 メソッド名

命名規則

メソッド名には任意の識別子を利用することができますが、一般的な命名規則が存在します。まずは、そのメソッドの命名規則について説明します。

メソッド名は、英語名の場合、先頭を小文字にします（※注：Javaの場合です。言語によっては、メソッドの先頭を大文字にするものなどがあります)。複数の単語で構成される場合、単語の区切りを大文字にして定義します。「print_arrays」ではなく、「printArrays」というメソッド名になります。

メソッドの処理が、「～を行う」という場合、メソッド名は動詞ではじまります。たとえば「配列を表示する」であれば、「arrayPrint」ではなく、「printArray」となります。また、「現在の時間を取得する」であれば「getCurrentTime」となります。

booleanを返すようなメソッドであれば、第6章8節の「boolean型の変数名」と同じように、「isXxx」や「hasXxx」といったメソッド名を付けます。

きちんとした名前を付ける（名前重要）

メソッド名は、メソッドが行うことをすべて説明しているような名前にすることが重要です。これは、変数名の場合と同様です。名前に略語を用いるかという判断も、変数名の場合と同様に行ってください。

動詞と名詞の関係

「動詞＋名詞」のようなメソッド名になる場合があります。たとえば、printDocument();やcalculateAverage();などです。

オブジェクト指向言語であるJavaの場合、Document.print();やdocument.print();などと、「print()」というメソッドの呼び出しを行うことができます。Documentはクラスであり、documentはオブジェクトです。これについては、第12章「オブジェクト指向」以降の章で説明します。このように、メソッド名の前に記述されている単語から「名詞」を記述せずに、「動詞」のみで十分なメソッド名になることもあります。

ただし、print();のように、メソッドの前になにも記述せずにメソッドの呼び出しができる「staticインポート」という機能を使う場面が想定されるようであれば、名詞も含めた方がよいメソッド名になります。staticインポートについては、第10章「クラスを利用する」で説明します。

反意語の使用

プログラミングしていると、ある機能（メソッド）に対する反対の意味を持った機能（メソッド）を作る場合が多くあります。たとえば、「ファイルを開く／ファイルを閉じる」「最小値を求める／最大値を求める」「文字を表示する／文字を隠す」などが考えられます。そのような場合、メソッド名に反意語の組を用いると一貫性が保ちやすくなり、コードの読みやすさにもつながります。

次の表に一般的な反意語の組をあげました。

◉表9-5-1　反意語の組

add/remove	begin/end	create/destroy	get/set	get/release
first/last	increment/decrement	insert/delete	lock/unlock	min/max
next/previous	old/new	open/close	put/get	send/receive
show/hide	source/destination	source/target	start/stop	up/down

9-6 変数の寿命

変に聞こえるかもしれませんが、変数には「寿命」があります。プログラムを作るうえで変数は必要なものですが、変数の寿命が長いと変数に気を使う必要が増えてきます。変数を扱いやすくするため、変数の寿命は短くする必要があります。この節では、変数の寿命を短くする方法と、なぜ、変数の寿命は短い方がよいのかを具体的に説明します。

変数のスコープ

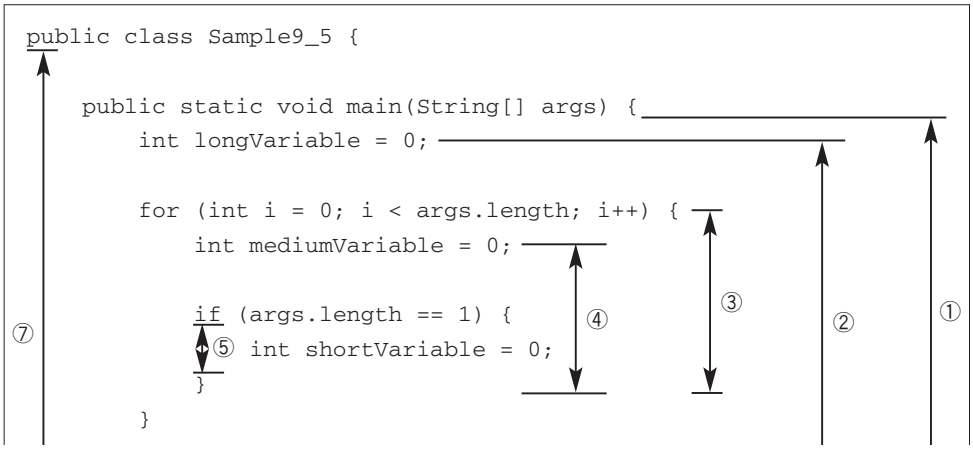
変数には寿命があり、寿命は長い場合も短い場合もあります。この寿命のことを「スコープ（Scope）」と呼びます。「可視性」と呼ぶ場合もあります。

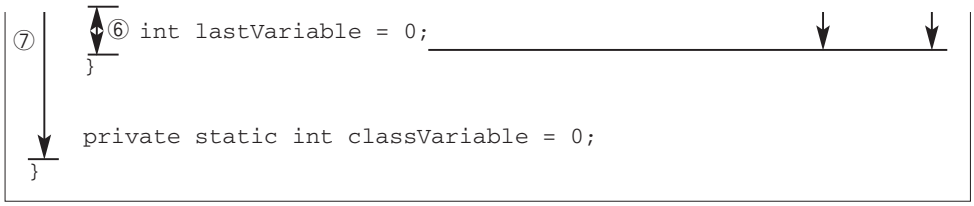
この、変数のスコープとは、コードのどの範囲まで知れ渡っているのかというもので、具体的には、どの範囲で参照可能なのかを意味します。

Sample9_5のコードを見てください。変数が7つ出ています。①引数のargs、②longVariable、③ループ変数のi、④mediumVariable、⑤shortVariable、⑥lastVariable、⑦クラス変数のclassVariableの7つです。これをスコープの広い順に並べ替えると、次のようになります。

⑦classVariable
①引数のargs
②longVariable
③ループ変数のi
④mediumVariable
⑤shortVariable
⑥lastVariable

● スコープ





変数のスコープは、基本的には変数が宣言されているブロックの中ということになります。クラス変数であれば、そのクラスに定義されたメソッドのどこからでも参照可能です。classVariableはmainメソッド内で参照できますし、mediumVariableは、for文の外では参照できません。また、メソッド内に定義されたローカル変数は、宣言された行からブロックが終了するまでの間では参照可能です。つまり、lastVariableはfor文の前では参照できません。

スコープを最小限に抑える

変数のスコープについて考えた場合、クラス変数のようにスコープが広い場合と、ループ変数のようにスコープが狭い場合では、次のようなメリット、デメリットがあります。

◉表9-6-1 スコープによるメリット／デメリット

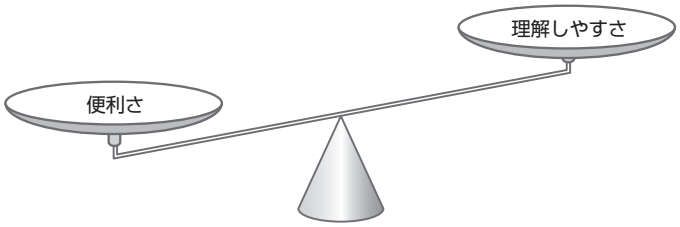
	メリット	デメリット
スコープが広い場合	参照が容易	使用されている場所を把握する必要がある
スコープが狭い場合	理解しやすい	つねに参照できない

これらのメリット、デメリットが、具体的にどのような場合なのかを説明します。

まず、スコープが広い場合のメリットの「参照が容易」とは、クラス変数などを参照する場合についてです。クラス変数への参照は、そのクラスに存在するどのメソッドからでも可能です。参照することに関しては、こちらの方が便利でしょう。しかし、その値はさまざまな箇所で使用されている可能性があるので、値を参照するだけならよいですが、値を代入した場合にその影響範囲を把握する必要が出てきます。あるメソッドがきちんと動作すればよいのではなく、すべてのメソッドがきちんと動作する必要があるからです。

次に、スコープが狭い場合のメリットの「理解しやすい」とは、ループ変数やループ内に定義された変数などは特定の箇所のみでしか参照できないので、変数がどのように使われているのかが理解しやすいということです。その代わり、ループ変数はループブロックの外では参照できないなど、つねに変数を参照することができるわけではありません。

変数を使用する場合、「理解しやすさ」と「便利さ」のトレードオフになります。



■図9-6-1 「理解しやすさ」と「便利さ」のトレードオフ

バグのないプログラムを作るには、コードの中身をきちんと把握していなければなりません。99%把握すればよいのではなく、100%把握しなければなりません。今日（こんにち）のプログラムの規模は非常に大きいものになっています。一度にそのすべてを把握するということは非常に困難です。一方、スコープの狭い変数を使用した場合は、そのメソッド内、あるいはそのループ内をバグのないコードにしておけば、他のメソッドのことまで気を使う必要がなくなります。

つまり、変数のスコープはできるだけ狭くした方がよいと言えます。

それでも、スコープの広い便利さを取る人がいるかもしれません。しかしそれは、コードを書く場合に便利であるだけで、コードを読む場合には困難になります。スコープが狭ければ、コードを読むことが容易になります。ではその場合、コードを書くことが困難かというと、ほとんど差は出ないでしょう。

■スコープを最小限に抑えるための方法

それでは、スコープを小さくするための方法をいくつか紹介しましょう。

❑ ループ変数など、ループ内で使用する変数はループ内で定義する

● よい例 ○

```
public static void main(String[] args) {
    for (int i=0; i<data.length; i++) {
        int temporary = data[i];
         (……中略……)
    }
```

● 悪い例 ×

```
public static void main(String[] args) {
    int i = 0;
    int temporary = 0;
    (……中略……)
    for (; i<data.length; i++) {
        temporary = data[i];
```

```
            (……中略……)
    }
```

これは、for文をif文に置き換えた場合でも同様です。

❑ 変数を宣言する場合は、使用する直前に宣言する

●よい例 ○

```
public static void main(String[] args) {
   (……中略……)
    int sum = getSum(data);
   (……中略……)
    int average = getAverage(data);
```

●悪い例 ×

```
public static void main(String[] args) {
    int sum;
    int average;
    (……中略……)
    sum = getSum(data);
    (……中略……)
    average = getAverage(data);
```

❑ 関連するステートメントはまとめて記述する

●よい例 ○

```
int[] oldData = getOldData();
int oldSum = getSum(oldData);
int oldAverage = oldSum/oldData.length;

int[] newData = getNewData();
int newSum = getSum(newData);
int newAverage = newSum/newData.length;
```

●悪い例 ×

```
int[] oldData = getOldData();
int oldSum = getSum(oldData);

int[] newData = getNewData();
int newSum = getSum(newData);
```

```
int oldAverage = oldSum/oldData.length;
int newAverage = newSum/newData.length;
```

❑ 関連するステートメントをまとめて別メソッドにする

● よい例 ○

```
int[] oldData = getOldData();
int oldAverage = getAverage(oldData);

int[] newData = getNewData();
int newAverage = getAverage(newData);
```

● 悪い例 ×

```
int[] oldData = getOldData();
int oldSum = getSum(oldData);
int oldAverage = oldSum/oldData.length;

int[] newData = getNewData();
int newSum = getSum(newData);
int newAverage = newSum/newData.length;
```

❑ 最も狭いスコープからはじめて、必要に応じてスコープを広くする

変数のスコープの広さを狭くするという作業は、狭いスコープを広くする作業よりもはるかに難しい作業です。ですので、適切なスコープが不明な場合は、最初はなるべく狭いスコープにしてコーディングするとよいでしょう。

9-7 コードの深さ

コードには深さというものがあります。深いコードは読み手に負担がかかります。どういったものが深いコードなのか、また深くしないためにはどうしたらよいのか。これらについて説明します。

ネストの深いコードと浅いコード

Sample9_6のコードを見てください。これは、変数charsの中身がJavaという文字の集合であった場合に、標準出力を行います。それ以外のときにはなにもしません。

● ネストが深いコード

```
public class Sample9_6 {
    public static void main(String[] args) {
```

```
        char[] chars = { 'J', 'a', 'v', 'a' };

        if (chars.length == 4) {
            if (chars[0] == 'J') {
                if (chars[1] == 'a') {
                    if (chars[2] == 'v') {
                        if (chars[3] == 'a') {
                            System.out.println("charsの中身はJavaで
                            した");
                        }
                    }
                }
            }
        }
    }
}
```

charsの配列の長さが4であるかを確認し、charsの先頭から文字が一致しているかを確認しています。しかし、このコードはifのネストが5つもあり、お世辞にもよいとは言えません。このコードは「ネストが深い」と言います。

ネストを浅くするテクニック

ネストが深いコードは、一般的にバグが発生しやすく、また、コードを理解することも困難になります。そこで、ネストを浅くするためのテクニックを紹介します。

❑ 条件を同じ階層に記述する

● Sample9_6のコードを改良した例（Sample9_7.java）

```
public class Sample9_7 {
    public static void main(String[] args) {
        char[] chars = { 'J', 'a', 'v', 'a' };

        if (chars.length != 4) {
            return;
        }
        if (chars[0] != 'J') {
            return;
        }
        if (chars[1] != 'a') {
            return;
        }
        if (chars[2] != 'v') {
```

```
            return;
        }
        if (chars[3] != 'a') {
            return;
        }
        System.out.println("charsの中身はJavaでした");
    }
}
```

条件に一致しないとわかった時点でreturnなどと記述し、その後の処理が行われないようにします。

❑ 再評価し、同じ階層に記述する

● ifが2重に記述されているコード

```
public static void main(String[] args) {
    if (isA()) {
        処理その1();
        if (isB()) {
            処理その2();
        }
    }
}
```

前述のようなコードがあった場合、次のようなコードに書き直すことができます。

● ifが2つ並列に記述されているコード

```
public static void main(String[] args) {
    boolean isA = isA();
    if (isA) {
        処理その1();
    }
    if (isA && isB()) {
        処理その2();
    }
}
```

2つ目の条件判定時に、再度isAという評価結果を参照することによって、ネストを浅くすることができます。

❑ if文やfor文などのブロックの内容をメソッドとして切り出す

先ほどのコードを例に取ると、次のようになります。修正したコードはネストの深さがどちらも同じになっています。

●処理の塊をメソッドとして切り出したコード

```
public static void main(String[] args) {
    if (isA()) {
        Aのときの処理();
    }
}

private static void Aのときの処理() {
    処理その1();
    if (isB()) {
        Bのときの処理();
    }
}
```


9-8 まとめ

・クラス内に処理の塊をメソッドとして定義することができます。
・メソッドに引数を渡して処理させることができます。
・メソッドの呼び出し元は、戻り値を受け取ることができます。
・クラス内に、定数を定義することができます。
・クラス内に、static変数を定義することができます。
・変数は属しているブロック内でしか参照できません。
・参照できる範囲をスコープと呼びます。
・スコープが狭い方が理解しやすいコードになります。
・コードにはネストが浅いコードとネストが深いコードがあります。
・条件を同じ階層に記述することによってネストを浅くすることができます。
・条件を再評価することによってネストを浅くすることができます。
・if文やfor文などのブロックの内容をメソッドとして切り出すことでネストを浅くすることができます。

メソッドをきちんと分割すると、自然にバグの少ないコードを記述できるようになります。処理の塊があれば、きちんとメソッドに分割するようにしましょう。

理解しやすいコードを記述するために、変数のスコープをできるだけ小さくします。また、ネストが深い場合、コードの可読性が悪くなり、さらに、処理の整合性が取りに

くくなるため、バグを誘発する危険性があります。ネストが深くならないように心がけましょう。

9-9 クイズ

Q9-1

次の［　　］を埋めてください。

外部から見たプログラムの動作を変えずにソースコードの内部構造を整理することを［　　］と呼びます。

Q9-2

次の［　　］を埋めてください。

定数名は、［　　］と_を組み合わせた名前にします。

Q9-3

次の［　　］を埋めてください。

Eclipseの機能では、［　　］キーを押すとメソッドの抽出を行うことができます。

Q9-4

Sample7_5を改善したコードがSample9_4ですが、さらにSample9_4を改善するとしたら、どのように改善しますか。また、Sample9_4を改善したコードをQuiz9_1というクラス名で作成してください。

Q9-5

本書で取り上げたスコープを最小限に抑えるための方法について、次の［　　］に当てはまるものを以下の選択肢から選んでください。

・［ A ］変数など、［ A ］内で使用する変数は［ A ］内で定義する
・変数を宣言する場合は、［ B ］で宣言する
・［ C ］はまとめて記述する
・［ D ］をまとめて別メソッドにする
・最も狭い［ E ］からはじめて、必要に応じて［ E ］を広くする

a）ループ
b）関連するステートメント
c）クラス変数
d）文字列
e）スコープ
f）定数
g）使用する直前
h）配列
i）クラス変数

Q9-6

次のプログラムには変数が7つ存在します。それぞれの変数のスコープを示してください。

● **Sample9_5.java**

```
public class Sample9_5 {

    public static void main(String[] args) {
        int longVariable = 0;

        for (int i = 0; i < args.length; i++) {
            int mediumVariable = 0;

            if (args.length == 1) {
                int shortVariable = 0;
            }
        }

        int lastVariable = 0;
    }

    private static int classVariable = 0;
}
```

Q9-7

次のコードのネストを浅くしてください。

● **Quiz9_2.java**

```
public class Quiz9_2 {
    public static void method(int value) {
```

```
        if (value != 0) {
            if (value % 2 == 0) {
                System.out.println("valueは偶数です");
            } else {
                System.out.println("valueは奇数です");
            }
        } else {
            System.out.println("valueはゼロです");
        }
    }
}
```

Q9-8

次のコードは、ある映画のチケットの料金計算をするコードです。料金体系は次の表の通りで、いちばん安くなる価格が適用されます。このコードのネストを浅くしてください。

◉表9-9-1 料金体系

一般	1800円
3歳以下	無料
小人（15歳以下）	1000円
レディースデー（水曜日）	1000円
シニア（60歳以上）	1200円

● Quiz9_3.java

```
public class Quiz9_3 {
    public static int 料金を計算する(boolean isMale, int age, boolean
    isWednesday) {
        if (age <= 3) {
            return 0;
        } else {
            if (age <= 15) {
                return 1000;
            }
            if (isMale == false) {
                if (isWednesday == true) {
                    return 1000;
                }
            }
            if (60 <= age) {
                return 1200;
```

```
            }
        }
        return 1800;
    }
}
```