CHAPTER

# 11

# コメントとコーディング規約

これまではコードの書き方を学んできました。ソースコードの中には、コードの内容を説明する「コメント」を記述することができます。本章では、コメントの記述方法について説明します。

また、よりよいコードにするためのコーディング規約についても説明します。

# 11-1 コメントとは

まずは、次のコードを見てください。次のコードは、前の章で使用したSample9_4にコメントを追加したものです。

**●コメントを追加したSample9_4のコード（Sample9_4.java）**

```
package sample.sample11;

public class Sample9_4 {

    /**
     * メインメソッドです。
     */
    public static void main(String[] args) {
        int[] points = new int[30]; //長さが30の配列を宣言します

        //配列を初期化します
        initializeArray(points);

        /* 平均点を求めます */
        double averagePoint = calculateAverage(points);
        //平均点を表示します
        System.out.println("このクラスの平均点は" + averagePoint + "点です");

        printPoints(points);
    }

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

コードを記述するには大きく2通りの方法があります。

**●行コメントの記述方法**

```
//コメント
```

**●ブロックコメントの記述方法**

```
/*
コメント
*/
```

行コメントの場合は、"//"を記述した位置から行の終わりまでがコメントになります。

ブロックコメントは、"/*"と"*/"の間がコメントとなります。この間にコメントを記述することができます。ブロックコメントは複数行にわたって記述することができます。

コメントには制約がありませんので、自由に記述することができます。たとえば、プログラムの説明文などプログラムの処理と関係のあることや、プログラムの処理と関係のないことであってもメモとして記述することができます。このメモをコメントと呼びます。コンパイラはコメントを無視して処理するため、コメントにはなにを書いてもプログラムの動作に影響を与えません。

コメントは、コードを読みやすくするために、コードに対して補助的な役割を果たします。

## 11-2 適切なコメント

前節の最後で、「コメントは、コードを読みやすくするために、コードに対して補助的な役割を果たします。」と書きました。しかしコメントを追加したSample9_4のコードを

読むと、冗長なもののように見えます。

**●コメントを追加したSample9_4のコード（その1）**

```
public class Sample9_4 {
  /**
   * メインメソッドです。
   */
public static void main(String[] args) {
```

public static void main(String[] args)と記述してあれば、そのメソッドは誰がどう見てもメインメソッドでしょう。

**●コメントを追加したSample9_4のコード（その2）**

```
int[] points = new int[30]; //長さが30の配列を宣言します
```

これもコメントがなくても、長さが30の配列を宣言していることは一目瞭然です。

**●コメントを追加したSample9_4のコード（その3）**

```
//配列を初期化します
initializeArray(points);
```

メソッド名を見れば配列を初期化していることがわかります。英語を理解できないという人がいるかもしれませんが、この程度の英単語であれば一般教養のレベルと言ってよいでしょう。

このように、読めばわかるコードに対して冗長なコメントを記述することは、「不適切なコメント」と言えます。では、どのようなコメントが適切と言えるのでしょうか？

コードを読んだだけではわからないようなことを記述すると、そのコメントの価値は高いものと言えます。たとえば次のようなものがあげられます。

**・コードの意図を説明したもの。**
**・書籍の見出しや目次のように、コードの要約になっているもの。**
**・著作権の告知やバージョン情報といった、コードでは表していない情報。**

また、コメントの記述には以下のことに注意する必要があります。

**・コメントを記述したあとにコードを修正すると、コードとコメントに食い違いが生じる場合があります。**

**・冗長なコメントが大量に記述されている場合、コードを読みづらくする可能性があります。**

コメントにはコードほどの厳格さはありません。コンパイラによるチェックがなにも行われないからです。あなたがコメントを読んだときに、間違った内容が記述されている場合があるかもしれません。しかし、他人のコードを読むとき、あるいは、自分が書いたコードであっても1年後や2年後に読んだ場合には、コメントはコードを読む助けになるでしょう。また、コメントを記述する際に、コメントが書きにくいと感じることがあるかもしれません。それは、コードが複雑でよくない場合や、自分自身がコードを理解していない場合が考えられます。コメントを書くことで、よりよいコードに修正するきっかけが生まれることもあります。

最適なコメントを考えるのは難しい作業ですが、コメントの記述内容を見直した次のコードを見ると、最初のコードと比べ、コードが理解しやすくなっていると思います。

**●コメントの記述内容を見直したコード（Sample9_4.java）**

```
package sample.sample11.better;

public class Sample9_4 {

    /**
     * 長さ30のint型の配列を用意し、その平均値の表示と、各値を表示します。
     */
    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);
    }

    public 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");
        }
    }

    /**
     * 配列の値を初期化します。
     * サンプルコード用の値を代入しているため、値の意味、規則性はありません。
     */
    private static void initializeArray(int[] points) {
        //最初の3件は任意の値を代入する
        points[0] = 90;
        points[1] = 62;
        points[2] = 76;

        //面倒なので、ループで同じ値を代入する
        for (int i = 3; i < 15; i++) {
            points[i] = 75;
        }
        for (int i = 15; i < 30; i++) {
            points[i] = 70;
        }
    }
}
```

## 11-3 Javadoc

ブロックコメントを拡張したコメントに「ドキュメンテーションコメント」というものがあります。"/**"ではじまり、"*/"で終わります。

**● ドキュメンテーションコメントの記述方法**

```
/**
ドキュメンテーションコメント
*/
```

ドキュメンテーションコメントを記述すると、ツールを使うことでAPIドキュメントを機械的に生成することができます。APIとはApplication Program Interfaceの略で、クラスの集合のようなものです。各ソフトウェアの開発者がすべての機能を一からプログ

ラミングするのは困難で無駄が多いため、多くのソフトウェアが共通して利用する機能は、APIというまとまった形で提供されています。JavaのAPIドキュメントは、

**http://java.sun.com/javase/ja/6/docs/ja/api/index.html**

から参照することができます。

ドキュメンテーションコメントから、JDKの「Javadoc」というツールを使って自動的にマニュアルを生成することができます。そのため、「Javadocコメント」とも呼びます。略してこのコメントを「Javadoc」と呼ぶことが多いです。また、APIドキュメントを「Javadoc」と呼ぶこともあります。

**● Javadocコメントを記述したコード（Sample9_4.java）**

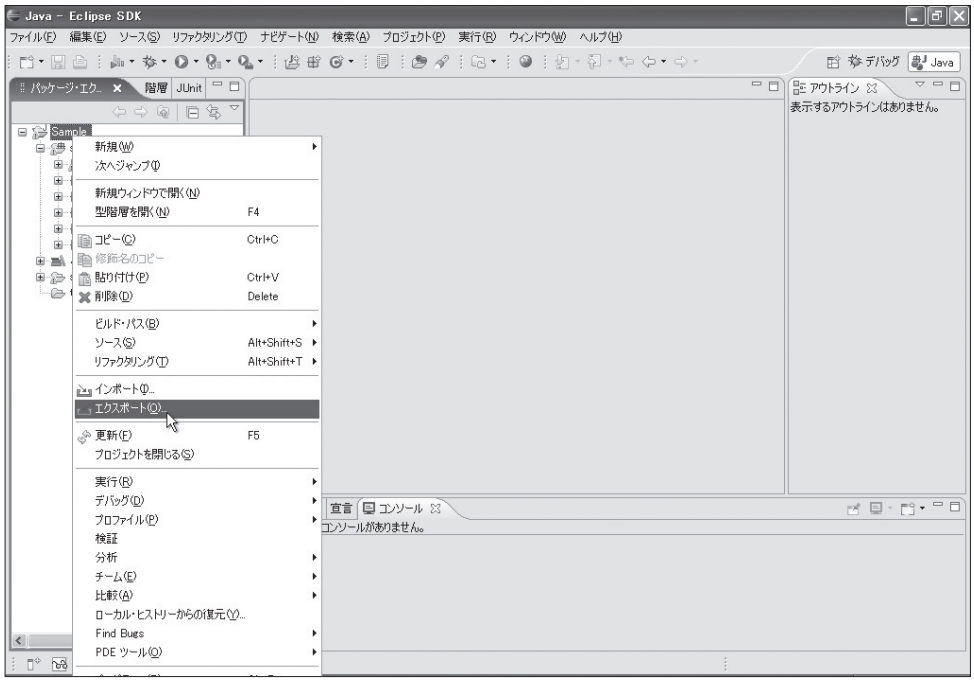
```
package sample.sample11.javadoc;

/**
 * Sample9_4にJavadocコメントを記述した例です。
 *
 * @version 1.0
 * @author Satsohi Kimura
 */
public class Sample9_4 {

    /**
     * 長さ30のint型の配列を用意し、その平均値の表示と、各値を表示します。
     *
     * @param args この引数は使用していません
     */
    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);
    }

    public 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");
        }
    }

    /**
     * 配列の値を初期化します。
     * サンプルコード用の値を代入しているため、値の意味、規則性はありません。
     */
    private static void initializeArray(int[] points) {
        //最初の3件は任意の値を代入する
        points[0] = 90;
        points[1] = 62;
        points[2] = 76;

        //面倒なので、ループで同じ値を代入する
        for (int i = 3; i < 15; i++) {
            points[i] = 75;
        }
        for (int i = 15; i < 30; i++) {
            points[i] = 70;
        }
    }
}
```

Javadocには、単純な文章の他に、@version、@author、@paramといった特殊な記述をすることで、バージョンや作成者情報であることを示すことができます。このアットマーク（@）ではじまる記述を「タグ」と呼びます。

EclipseからAPIドキュメントを生成する方法は次の図を参考にしてください。

プロジェクトを右クリックし、「エクスポート」を選択します。




**■図11-3-1 「エクスポート」を選択**

「エクスポート」ダイアログが表示されるので、「Java＞Javadoc」を選択し、「次へ」ボタンをクリックします。


**■図11-3-2 「エクスポート」ダイアログ**

「Javadocが生成される型の選択」で、どのクラスのJavadocを生成するのかを選択し

ます。今回はすべてのクラスを選択しました。

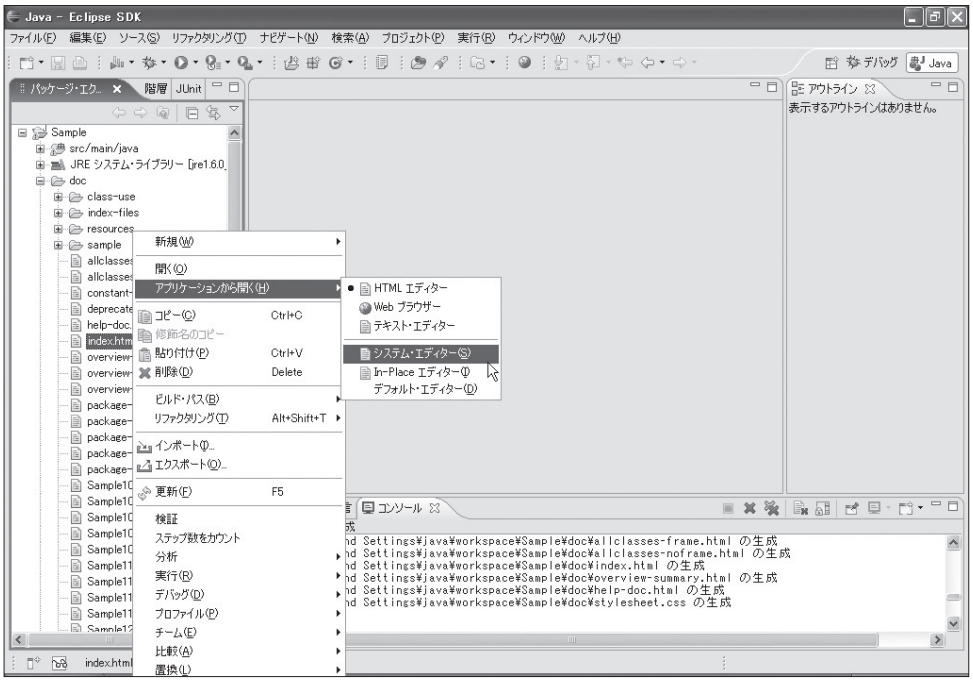
「次の可視性を持つメンバーにJavadocを作成」で「protected」を選択します。一般的に、Javadocを生成するときの設定は「protected」を選択し、「終了」ボタンをクリックします。protectedの意味については、第13章「クラスを拡張する」で説明します。


**■図11-3-3 「protected」を選択**

「Javadocロケーションの更新」ダイアログが表示されたら、「すべてはい」をクリックします。すると、Javadocの生成が開始します。

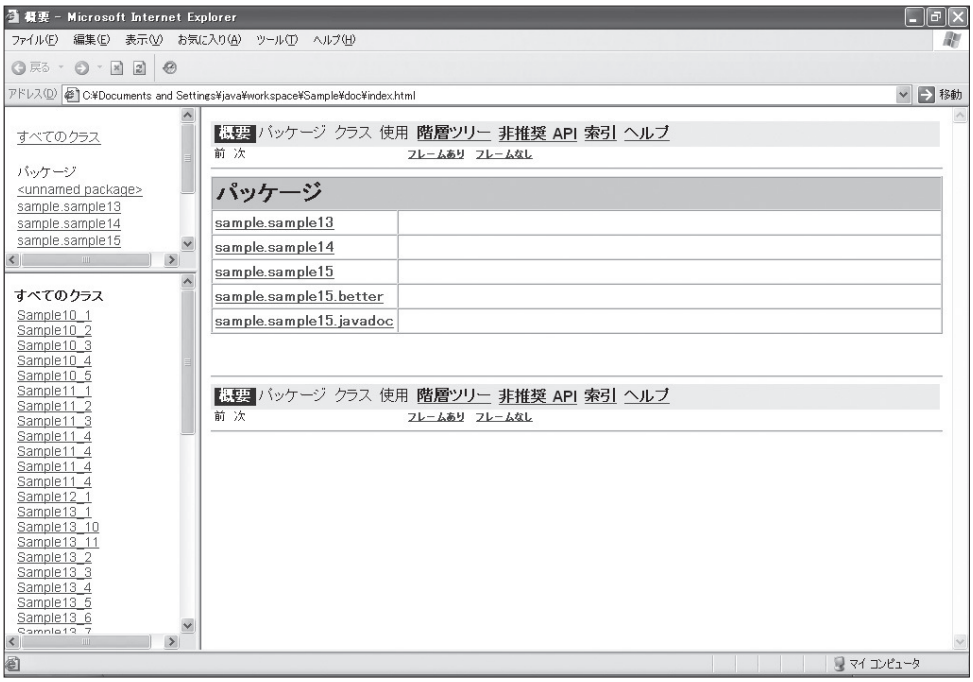

**■図11-3-4 「Javadocロケーションの更新」ダイアログ**

出力先をデフォルトのまま指定しなかったので、Javadocの生成が終了すると、パッケージ・エクスプローラに「doc」というフォルダが作成されます。この中にあるindex.htmlを右クリックし、「システム・エディター」を選択します。



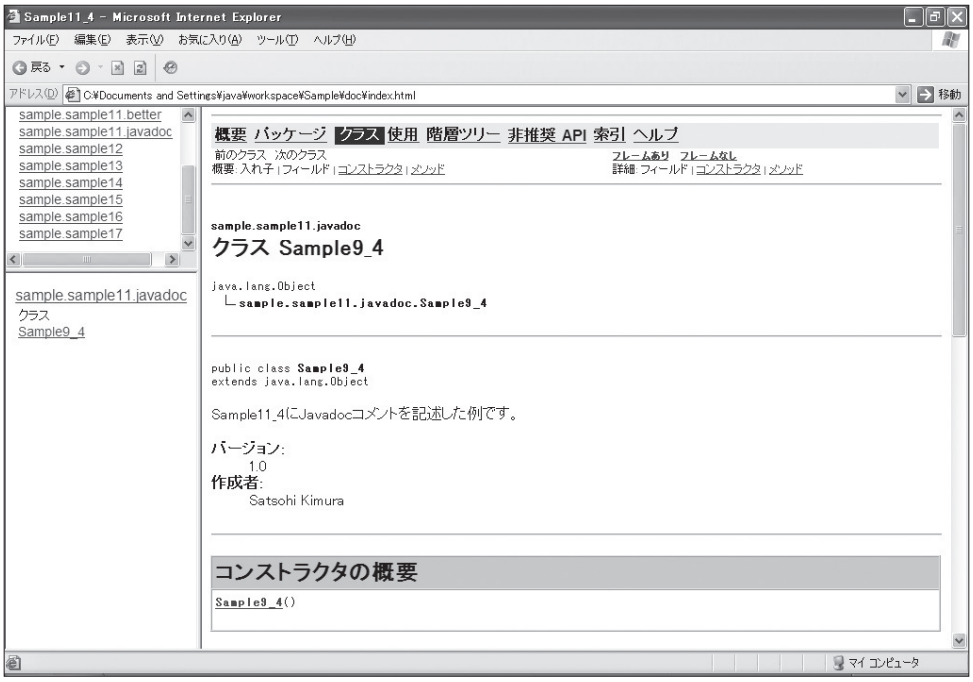

**■図11-3-5 「システム・エディター」を選択**

するとブラウザが起動し、生成されたJavadocを見ることができます。




**■図11-3-6 生成されたJavadocの例**

タグを使用して情報記述した場合は、次の図のように出力されます。




**■図11-3-7　タグを付けたクラスのJavadocの例**

ここで紹介したタグ以外にも、Javadocで使用できるタグがあります。それらについては、付録C-1を参照してください。

## 11-4 TODOコメント

コメントの中には特別な意味を持ったものがあります。

たとえば、「TODO（スペース）」ではじまるTODOコメントがあります。TODOとは「To do」のことで、「やらなければならないこと」という意味です。このTODOコメントを記述すると、Eclipseの「タスク」ビューに該当行が表示されます。タスクビューに一覧で表示されるので、忘れてはならないことなどもTODOコメントで記述するとよいでしょう。

TODOコメント以外の特殊なコメントには、FIXMEコメントとXXXコメントがあります。「コードが正しくない、または正しく動いていない、修正が必要」という場合に、FIXMEコメントを記述します。XXXコメントは、「正しくないがとりあえず動いてはいる」という場合に記述します。

◉表11-4-1 特殊なコメント

| | |
|---|---|
| TODO | やらなければならないこと |
| FIXME | 正しくない。正しく動いていない。修正が必要 |
| XXX | 正しくないがとりあえず動いてはいる |

これらのコメントはいずれも大文字で記述し、あとにスペースを含め、そのうしろにコメントを記述します。また、この3つのコメントはタスクビューに表示されます。

筆者の経験上、XXXコメントはあまり使いません。正しくないということは、将来修正するということです。ですので、XXXコメントではなく、TODOコメントやFIXMEコメントを使用しています。

◉表11-4-2 特殊なコメントの例

| | |
|---|---|
| TODOコメント | `//TODO 変数名をわかりやすく変更する` |
| TODOコメント | `//TODO このメソッドはあとで作成` |
| FIXMEコメント | `//FIXME 第二引数がfalseの場合の動作が正しくないので要修正` |
| FIXMEコメント | `//FIXME テストが失敗するのであとで修正` |

タスクビューに表示されるコメントは、TODOなどが記述された「行のみ」です。次のようなコメントは、タスクビューには「TODO」としか表示されないので注意してください。

●複数行に分かれたTODOコメント

```
//TODO
//あとで実装する（とりあえず、trueを返すようにしておく）。
```

## 11-5 コーディング規約

世の中にはさまざまなコーディング規約が存在します。一般的なコーディング規約が述べている項目にはおもに次のようなものがあります。

**・ネーミング規約に関する項目**
**・コードのフォーマットに関する項目**
**・多くの人が間違う問題に関する項目**

「ネーミング規約に関する項目」については、本書でも取り上げました。きちんと表現している名前を付けるというものでした。本書で述べた内容よりも、さらに細かい規約

を設けている場合もあります。

「コードのフォーマットに関する項目」については、1メソッドの行数は20行以下といったコードサイズに関するものから、カッコの前後にスペースを設けるか設けないかといった、細かい内容まで存在します。これは、Eclipseのフォーマット機能を使うことで容易に実現可能です。コードサイズについては後述します。

「多くの人が間違う問題に関する項目」については、「このようなコーディングをするとこういった不具合を招く可能性があるので、次のようにコーディングしてください。」といった内容が具体的に書かれている場合があります。このような項目については、バグを発生させないための参考になるテクニックが多くあります。

これらの規約は、「コードの可読性を上げる」「バグを抑制する」など、よりよいコードにするためのものです。フリーで利用できるJavaのコーディング規約には次のものがあります。

一般的な規約についてはこれらを参考にしてください。

**Javaコーディング規約2004**
**http://www.objectclub.jp/community/codingstandard/JavaCodingStandard2004.pdf**

**Java コーディング標準（オブジェクト倶楽部バージョン）**
**http://www.objectclub.jp/community/codingstandard/CodingStd.doc**
**http://www.objectclub.jp/community/codingstandard/CodingStd.pdf**

**Java 言語 コーディング規約**
**http://www.tcct.zaq.ne.jp/ayato/programming/java/codeconv_jp/**

## プログラムサイズ

さて、プログラムサイズに関してですが、プログラムにはさまざまなサイズがあります。たとえば、次のような項目があげられます。

- **パッケージの数**
- **クラスの数**
- **1クラスの行数**
- **1クラスのメソッドの数**
- **1クラスの変数の数**
- **1メソッドの行数**
- **1行の文字数**

「1クラスのコードが10万行を超えている」、また、「1万行近いメソッドがある」とい

ったことを、筆者は聞いたことがあります。このような巨大なコードをどう思いますか？決してよいコードとは言えません。

一般的に巨大なコードは短いコードよりもバグの発生頻度が増えると言われています。たとえば、同じ1000行のコードでも、1メソッドで1000行の場合と、1000行の処理が20個のメソッドに分かれている場合とでは、1メソッドで1000行の場合の方がバグが多いということになります。また、巨大なコードは、可読性も悪くなります。

このような観点から筆者がプログラミングするときに意識している規約があります。それは、「15ポイントルール」というものです。

## 15ポイントルール

15ポイントルールとは、メソッドのサイズに関するルールです。それは、コードにポイントを付け、そのポイントの合計が15ポイント以内になるようにメソッドを作成するというものです。

ポイントの換算ルールについて説明します。

コードを記述しなければバグは発生しません。コードを書くたびにバグが含まれる確率は増えていきます。単純なステートメントよりも、条件分岐やループ処理を含んだ方がバグを含む確率は大きいです。このような考え方がポイントの換算ルールのベースです。また、本書で説明しているように、変数名の長さが適切でない場合や深くネストされたコードも、バグを含む確率は大きくなります。

### ❑ 1ステートメント

1ポイントとします。行数ではなくステートメント数をカウントします。ステートメントがない空行やカッコだけの行は、ポイントに含めません。ただし、初期設定（引数などから値の取得）を行うステートメントは除きます。また、catchも1ポイントとして換算します（catchについては、第17章「例外」で説明します）。

**●2ポイントのコード**

```
logger.debug("Hello, world!");
logger.debug(1 + 1);
```

**●0ポイントのコード**

```
private static Person person = new Person();
public static void main(String[] args) {
    String name = args[0];  //引数から値を取得
    String personName = person.getName();  //オブジェクトから値を取得
```

### ❑ 条件文（if、switch、三項演算子）

2ポイントとします。ただし条件式が複数存在する場合には、条件式1つにつき1ポイントを加算します。

**● 2ポイントのコード**

```
if(20 < age) {
```

**● 3ポイントのコード**

```
if(20 < age && isMale == false) {
```

**● 4ポイントのコード**

```
if((20 < age && isMale == false) || 60 < age) {
```

### ❑ ループ（for、while）

3ポイントとします。ただし条件式が複数存在する場合には、条件式1つにつき1ポイントを加算します。

**● 3ポイントのコード**

```
for(String val : array) {
```

**● 3ポイントのコード**

```
while(isStop == false) {
```

**● 4ポイントのコード**

```
while(isStop == false || isPause == false) {
```

### ❑ 変数名

変数名が20文字を超える変数を定義した場合は、1ポイントとします。

**● 0ポイントのコード**

```
int velocity;
```

**● 1ポイントのコード**

```
int maximumVelocityOfTrain;
```

### ❑ 階層

多くのコーディング規約では、「1行が半角80文字を超えないようにする」という項目があります。理由をきちんと説明している規約はあまり目にしたことがありませんが、インデントが深いコードを書かないための項目です。

これを定量的にポイントに換算します。

ブロックの階層が3を超える場合、ポイントを加算します。4階層目のブロックを1ポイント、5階層目のブロックを2ポイントとします。5階層のブロックの場合、4、5階層目のポイントで計3ポイント加算されます。

**●9ポイントのコード**

```
public static void main(String[] args) {
    if (isA()) { //1階層目（ifで2ポイント加算）
        if (isB()) { //2階層目（ifで2ポイント加算）
            if (isC()) { //3階層目（ifで2ポイント加算）
                if (isD()) { //4階層目（ifで2ポイント加算、4階層目なのでさらに1ポイント加算）

                }

            }
        }
    }
}
```

これ以上メソッドを分割できない、最適な細分化がされているなど、処理の最小の塊と考えてよい場合には、15ポイントを超えてもかまいません。

また、15ポイントを超えるようなコードを書く場合には、そのコードをよく理解していないということも考えられます。たとえば料理を作るときに、「下ごしらえ」や「隠し味」にあたる工程があります。しかし、その料理についての知識がまったくない場合、すべての作業が単なる手順としてのみ認識されてしまうということに似ています。

**●細分化されている**

```
料理する() {
    下ごしらえする();
    メインの調理をする();
    隠し味を加える();

}
```

●細分化されていない

```
料理する() {
    XXXを切る();
    XXXをoooに漬ける();
    YYYを切る();
    XXXをYYYと一緒に炒める();
    pppを加える();
    qqqを加える();
    rrrを加える();
}
```

また、メソッドに分割していなくても、空行を挿入することで細分化することはできます。次のようなコードであれば、空行の前後で処理がまとまっているということがわかります。「料理する()」メソッドが15ポイントを超えていなければ、別のメソッドに分割しなくてもよいかもしれません。

●空行による細分化

```
料理する() {
    XXXを切る();
    XXXをoooに漬ける();

    YYYを切る();
    XXXをYYYと一緒に炒める();
    pppを加える();
    qqqを加える();

    rrrを加える();
}
```

## 「正しいソースの書き方養成ギブス」

コードが15ポイントルールを守っているかどうかをチェックするツールを本書の読者のために作成しました。このツールはEclipseプラグインから利用できます。

これは、「正しいソースの書き方養成ギブス」という意味から略して「ギブス」と名付けました。つねにギブスを使用していると、正しいソースの書き方が身に付くように、という思いがあります。

ギブスでは、15ポイントルールのチェック以外の項目についてもチェックを行っています。チェックしている項目は次の通りです。なぜそのようなチェックを行うのかという理由の大半は、本書で説明済みです。カッコ内の章または節を参照してください。

### ❑ 15ポイントルールを守っているかをチェック

メソッドのサイズが15ポイントを超えている場合に警告を表示します（この節「コーディング規約」参照）。

### ❑ switch文にdefault句を記述しているかをチェック

defaultブロックは、if-else文のelseブロックのような使い方ではなく、通常おこりえない場合を記述しておくとよいです。つまり例外をスローするコードである場合がほとんどを占めると言ってもよいでしょう。例外をスローするようにしておくと、switchの入力値として想定外の値が使用された場合にすぐに気づくことができます。そのため、default句を記述していない場合、ギブスは、記述するように促します（第6章6節「switch文」参照）。

**●正しいdefault句の使い方**

```
switch (month) {
case JANUARY:
(……中略……)
case DECEMBER:
(……中略……)
default:
    throw new IllegalArgumentException("不正な月 : " + month);
}
```

### ❑ マジックナンバーを使用していないかチェック

定数ではなく、マジックナンバーを使用していないかチェックします（第9章1節「定数とは」参照）。

### ❑ きちんとカッコを使用しているかチェック

たとえば、ifブロックの中にステートメントが1しかない場合でもカッコを使用するように促します（第6章1節「if文」参照）。

### ❑ 不要な修飾子を使用していないかチェック

不要な修飾子は、不要な情報をコードに記述しているということです。不要な修飾子を記述している場合は警告を表示します。また、不要な修飾子は無用な横スクロールを発生させます（第15章1節「インターフェースとは」参照）。

### ❑ 不正な例外処理をしていないかをチェック

例外をキャッチし、なにも処理しない。または、デバッグログのみなど、不正な例外処理を行っている可能性がある場合には、警告を表示します（第17章「例外」参照）。

### ❑ 標準出力を使用していないかチェック

ログ出力ではなく、標準出力を行っている場合、警告を表示します（第16章1節「入出力」参照）。

### ❑ mainメソッドを使用していないかチェック

mainメソッドは、アプリケーションを起動したときに呼び出されます。実際の仕事の現場でmainメソッドを記述することはほとんどありません。動作確認を行うためにmainメソッドを記述しているならば、JUnitでテストを行うようにしてください。そのような警告を表示します（第19章「テスト」参照）。

また、mainメソッドを記述するクラスは、Mainという名前にすることが慣習となっています。Mainクラスにmainメソッドを記述している場合には、警告を表示しません。

### ❑ 例外のオブジェクトを生成後きちんとthrowしているかチェック

本書の執筆中に、あるベテランプログラマーが行っていた間違いです。このときは運よくコードレビュー時に発見できましたが、そうでない場合の影響は致命的なものです。例外のオブジェクトの生成だけしか行っていない場合には、警告を表示します（例外については、第17章「例外」で説明します）。

**●間違ったコード**

```
} catch(Exception e) {
    new RuntimeException(e);
}
```

**●本来のコード**

```
} catch(Exception e) {
    throw new RuntimeException(e);
}
```

### ❑ スペルチェック

コードに記述している単語が、本当に存在する英単語かをチェックします。これは、単純なスペルミスの発生に気づくだけではなく、間違えた英単語を使用している場合にも有効です。日本人プログラマーがよく使用する動詞の「regist」という単語は存在せず、「register」が正しいというのは有名な話です。このような単語を使用しないためにもスペルチェックは有効です。辞書に存在しない英単語を使用した場合、警告が表示されます。そのときは、辞書で確認をするようにしてください。

### Column ギプス／キブス

骨折などの怪我をしたとき、患部を固定・保護するために包帯を石膏で固めて使用するのがギプスです。ギプスと聞いて、漫画「巨人の星」の「大リーグボール養成ギプス」を思い出す人もいることでしょう。

このギプスは、「石膏」を意味するドイツ語の「Gips」に由来する言葉です。綴りを見ればわかるように、その読みは「ギプス」です。「ギブス」ではありません。

さて、2007/11/25現在のGoogleでの検索件数を比べてみましょう。

◉**表11-5-1　ギプスvsギブスの検索結果**

| | |
|---|---|
| **ギプス** | 295,000件 |
| **ギブス** | 492,000件 |
| **大リーグボール養成ギプス** | 550件 |
| **大リーグボール養成ギブス** | 10,400件 |

どちらかと言えば、「ギブス」の方が主流のように思えます。

ギブスの方がピンとくる人の方が多いようなので、ツールの名前はギブスにしました。ただし、英語名はgipsとしています。

新しいEclipseにギブスをインストールする方法などは、下記のURLでサポートしていますので、こちらをごらんください。

**http://dodododo.jp/java/gips/index.html**

## 11-6 まとめ

- **"//"を記述した位置から行の終わりまでがコメントになります。**
- **ブロックコメントは、"/*"と"*/"の間がコメントになります。**
- **コメントには、任意のメモを記述することができます。**
- **コードの意図を説明したもの、コードの要約になっているものをコメントとして記述するとよいです。**
- **Javadocを利用するとAPIドキュメントを出力することができます。**
- **TODOコメント、FIXMEコメント、XXXコメントという特殊なコメントがあります。**

コードを補助するようなコメントは記述するように心がけましょう。またJavadocコメントも利用者に有効な情報を与えることができます。

## 11-7 クイズ

### Q11-1

次の［　　］を埋めてください。

行コメントを記述するには、［　　］に続けてコメントを記述します。

### Q11-2

次の［　　］を埋めてください。

ブロックコメントを記述するには、［ A ］から［ B ］までの間にコメントを記述します。

### Q11-3

次の［　　］を埋めてください。

ドキュメントコメントを記述するには、［ A ］から［ B ］までの間にコメントを記述します。

### Q11-4

次の［　　］を埋めてください。

［ A ］を説明したもの、［ B ］になっているものをコメントとして記述するとよいです。

**Column スクリーンセイバーの解除方法**

パソコンの電源を入れたまま使用せずに放置していると、設定にもよりますが、多くのパソコンはスクリーンセイバーに切り替わります。スクリーンセイバーを解除するには、キーボードのキーを押すなど、パソコンに対してなにかしら入力をする必要があります。

コンピュータに詳しくない人の多くは、「スペース」キーや「Enter」キーを押したり、マウスをクリックします。しかしそのようなことをすると、画面上の意図しないボタンをクリックしてしまうなど、なにが起こるのかわかりません。このようなスクリーンセイバーの解除方法はよくありません。

「Shift」キーを押すのが、よいスクリーンセイバーの解除方法といえます。