コーディング規約

（Java）

第1.1版

平成20年10月24日

NTTアドバンステクノロジ株式会社

**変更履歴と版情報・問い合わせ先**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 文書名 | | コーディング規約（Java）　第1.0版 | | | |
| 作成者 | |  | | | |
| 承認者 | | 印　<承認日>　　　　年　　月　　日 | | | |
| 問合せ先 | | 開発推進部 | | | |
| ファイル名 | |  | | | |
| 改訂履歴 | | | | | |
| 版数 | 変更箇所 | | 変更内容・理由 | 担当者 | 作成日 |
| 0.5 | 全体 | | 新規作成 |  | 2008年4月1日 |
| 0.9 | ２～４章 | | SA本のコーディング規約に差し替え |  | 2008年9月29日 |
| 1.0 | 全体 | | コメント反映 |  | 2008年10月24日 |
| 1.1 | 全体 | | 新人研修用に変更 | 受講者 | 2009年6月24日 |

**目次**

[**まえがき 3**](#_Toc233706680)

[**1．コーディングの心得　五ヶ条 4**](#_Toc233706681)

[**２．規約 6**](#_Toc233706682)

[**２．１　ファイル 6**](#_Toc233706683)

[**２．１．１　ファイル名 6**](#_Toc233706684)

[**２．１．２　ソースファイル内の記述順 6**](#_Toc233706685)

[**２．２　宣言 7**](#_Toc233706686)

[**２．２．１　変数宣言 7**](#_Toc233706687)

[**２．２．２　クラス／インタフェースの宣言 8**](#_Toc233706688)

[**２．３　文（ステートメント） 8**](#_Toc233706689)

[**２．４　インデント 11**](#_Toc233706690)

[**２．５　スペース 11**](#_Toc233706691)

[**２．６　命名方法 12**](#_Toc233706692)

[**２．６．１　一般的な名前の付け方 12**](#_Toc233706693)

[**２．６．２　パッケージ 12**](#_Toc233706694)

[**２．６．３　クラス／インタフェース 12**](#_Toc233706695)

[**２．６．４　メソッド 13**](#_Toc233706696)

[**２．６．５　変数 13**](#_Toc233706697)

[**２．６．６　定数（static final） 13**](#_Toc233706698)

[**２．７　コメント 13**](#_Toc233706699)

[**２．７．１　実装コメント 13**](#_Toc233706700)

[**２．７．２　ドキュメント用コメント 15**](#_Toc233706701)

[**２．８　その他 16**](#_Toc233706702)

[**２．８．１　定数リテラル 16**](#_Toc233706703)

[**２．８．２　代入 16**](#_Toc233706704)

[**２．８．３　カッコの使用 16**](#_Toc233706705)

[**２．８．４　３項演算子？： 16**](#_Toc233706706)

[**２．８．５　数値0との比較 17**](#_Toc233706707)

[**２．８．６　ヌル文字との比較 17**](#_Toc233706708)

[**２．８．７　フィールド 17**](#_Toc233706709)

[**３．セキュアプログラミング 17**](#_Toc233706710)

[**３．１　クラスの使用制限 17**](#_Toc233706711)

[**３．２　パーミッション 17**](#_Toc233706712)

[**３．３　カプセル化 18**](#_Toc233706713)

[**３．４　シリアル化 18**](#_Toc233706714)

[**３．５　不正な継承 18**](#_Toc233706715)

[**３．６　レースコンディション 19**](#_Toc233706716)

[**４．コーディング例 20**](#_Toc233706717)

[**付録１　Javadocについて 23**](#_Toc233706718)

[**付録１．１　Javadocを記述すべきクラス、メソッド、フィールド 23**](#_Toc233706719)

[**付録１．２　Javadocの記述レベル 24**](#_Toc233706720)

[**付録１．３　Javadoc生成の際の注意 25**](#_Toc233706721)

[**付録２．１　文章規定 26**](#_Toc233706722)

[**２．１．１ 通常コメント記入例 26**](#_Toc233706723)

[**２．１．１ Javadoc記入例 26**](#_Toc233706724)

[**付録３．１　用語集表記規則 28**](#_Toc233706725)

[**付録３．２　用語集 28**](#_Toc233706726)

まえがき

**■　適用目的**

本規約は、Javaを使ったソフトウェア開発において、コーディング作業における標準的なコーディングルールを規定することを目的とする。

**■　適用工程**

本規約は、NTT-ATが管理する開発プロジェクトの、コーディング/単体試験工程に適用する。

**■　適用対象者**

本規約は、Javaプログラム開発者に対し適用する。

**■　適用条件**

本規約は、Javaを使用してコーディングを実施することを適用の条件とする。

本規約をカストマイズして、各プロジェクト固有の規約を作成し、適用することができる。

**■　参考資料**

本規約のほかに、一般的に流通するコーディング規約として、（株）電通国際情報サービスが作成したＪａｖａコーディング規約2004がある。　参考のため、参照先を以下に記す。

<http://www.objectclub.jp/community/codingstandard/JavaCodingStandard2004.pdf>

Ｊａｖａコーディング規約2004は、（株）電通国際情報サービスが公開し、自由に利用・改変が許されたものである。　保守性を重視したコーディング作法となっていることが特徴で、本書の第１章も、このコーディング規約を参考としている。

# **1．コーディングの心得　五ヶ条**

コーディングは、本来、非常に知的な作業です。

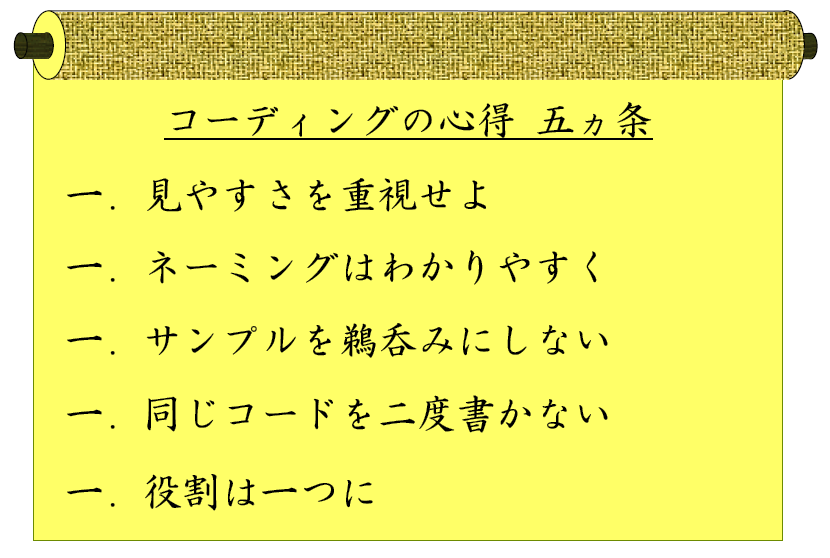
長いプログラムを記述すること(ステップ数)によって生産性が評価されたのは、過去の時代の出来事です。現在は、クラスやメソッドの役割が明確で、ロジックが読みやすく、保守性に優れたプログラムを記述することが評価されます。

コーディング規約は、コードの書き方に関する一種のパターンと言うこともでき、コードの保守性を向上させる具体的な方法を示しています。したがって、規約の一つ一つの意図を理解し、守ることが重要です。しかし、保守性に優れたコードを作成するためには、コーディング規約を守ることに加えて、良いコードを記述するための基本的な心構えをしっかり心に留めておく必要があります。

ここでは、この心構えを「コーディングの心得 五ヶ条」として紹介します。

どの心得もごく当たり前に思えるかもしれません。しかし、これらの心得と照らして既存のコードを参照すると、実践されていないコードが意外と多いことに気づかれる方も多いのではないでしょうか。

特に、コーディング経験の浅いプログラマの方は、この「コーディングの心得 五ヶ条」を理解し、コーディング時に自分で考えて実践することをお勧めします。



◇見やすさを重視せよ

「良いコード」の基本は、「他の人が読んでもわかりやすいと感じられるコード」と言えます。コードの見やすさは、フォーマットはもちろん、ロジックの簡潔さやAPIの常識的な使い方などから生まれます。コーディングにあたっては、常に他の人の視点を意識しながら、見やすさに気を配って記述しましょう。

また、自分で記述したコードであっても、しばらくたってから読み返してみると理解に時間がかかった経験はないでしょうか。「3日前に書いたコードは他人のコードと同じ」ということも言われます。見やすさを重視することは、他の人のためだけでなく、自分のためにもなります。

◇ネーミングはわかりやすく

コーディングでは、様々な変数やメソッドなどにネーミング(名前付け)しなければなりません。ネーミングとは、本来、その対象の本質を表すような名前を考える作業です。大変難易度の高い作業ですが、一方で適当に行ってもコードの動作は変わらないため、人によっては手を抜きがちです。しかし、ネーミングの良し悪しは、コードの可読性に非常に大きな影響を及ぼします。

例えば、「C0001」というクラス名があるとします。これでは、何を表すクラスなのかすぐにわかりません。また、「int p = 5000;」という記述があるとします。プログラマに聞くと、変数名pは価格(Price)の略だと言ったとします。であれば略さずに、「int price = 5000;」としたほうがわかりやすいと思いませんか？

「ネーミングはわかりやすく」の背景には、読んで内容が理解できるという意味で、文章のようなプログラミングを行うという考え方があります。

◇サンプルを鵜呑みにしない

サンプルコードを活用すること自体は、著作権等を侵害しなければ問題ないでしょう。問題なのは、その内容や背景を理解しないまま、サンプルコードだけを鵜呑みにして、「おまじない」として表面的に適用してしまうことです。

コードを「おまじない」ととらえていては、サンプルコードの間違いを気づかないまま適用してしまうこともあります。

例えば、ストリームのクローズ処理を行っていないサンプルコードであっても、それに気づかずに自分のコードに適用してしまい、後で思わぬ障害を引き起こすということもありえます。サンプルコードは、そこで説明する内容に絞ったコードが多いため、このような例はよく見られます。

また、サンプルコードをそのまま適用した結果、自分が記述すべきコードには必要のないコードが含まれてしまうこともあります。その場合、コードの可読性を下げる原因になります。

自分のコードは、自分で深く理解して記述しましょう。

◇同じコードは二度書かない

コードをコピー・ペーストしていませんか？コピー・ペーストしてしまうと、何らかの修正をする際に、全ての個所に同じ修正をする羽目になります。同じコードが現れるようならまとめて一つにし、外に出してコールするような書き方をすべきです。

同じコードをまとめる作業は、どちらかといえば、コーディング時よりリファクタリング(ソフトウェアの外部的振る舞いを変更せずに内部構造を改善する作業)で行われることが多いでしょう。しかし、コーディング時からできるだけ気をつけておきたいことです。

◇役割は一つに

メソッドの役割が明確で、かつ1 つであれば単体テストが行いやすくなります。つまり、コードの「試験性」が高まります。また、役割が一つであれば、後でコードを変更する際に修正箇所がわかりやすいため、障害修正に要する時間が短くなります。つまり、コードの「保守性」があがります。

例えば、「チェックをして実行する」機能を実現するために、checkAndDo()メソッドが存在したとします。このメソッドはcheck()メソッドとdo()メソッドに分割すべきです。なぜなら、checkAndDo()メソッドのチェックロジックに誤りがあった場合、本来はdo()メソッドに書かれる内容まで把握する必要があるからです。分割してあれば、check()メソッドだけの変更で済みます。

このことはクラスの設計にも言えることです。

備考）本章は、（株）電通国際情報サービスが作成したＪａｖａコーディング規約2004を参考としました。

# **２．規約**

　Ｊａｖａ言語のコーディング規約を以下のように定める。

## **２．１　ファイル**

## **２．１．１　ファイル名**

ソースファイルとクラスファイルの拡張子は、以下のものを使う。

・ソースファイル：　java

・クラスファイル：　class

ディレクトリの内容を説明するテキストファイルを作成する場合、READMEというファイル名を使用する。

## **２．１．２　ソースファイル内の記述順**

１つのソースファイルには１つのクラスまたはインタフェースを記述する。ただし、privateクラスやインタフェースを、関連するpublicクラスと同じソースファイル内に記述しても良い。その場合、publicクラスを最初に記述すること。

１つのソースファイル内は、以下の順で記述する。

①　ソースファイルを説明するコメント

②　package宣言文

③　import宣言文

④　クラス／インタフェースのコード

（１）ソースファイルを説明するコメント

/\*

\* クラス名

\*

\* バージョン情報

\*

\* 著作権表示

\*/

（２）package宣言文

package宣言文のスタイルは、「２．６．２　パッケージ」を参照。

（３）import宣言文

import宣言文は、以下の順に記述する。

①　java.\*

②　javax.\*

③　その他

なお、import文においてはクラス名まで記述し、”\*”は使用しない。

また、java.langパッケージについては記述しなくて良い。

（４）クラス／インタフェースのコード

クラスおよびインタフェースのコードは、以下の順に記述する。

①　ドキュメント用コメント

「２．７．２　ドキュメント用コメント」を参照。

②　classまたはinterfaceの宣言

③　実装コメント

「２．７．１　実装コメント」を参照。

④　クラス変数

public、protected、privateの順に宣言する。

⑤　フィールド

public、protected、privateの順に宣言する。

⑥　コンストラクタ

⑦　メソッド

コードを読みやすくするために、機能ごとにまとめて記述する。

public、protected、privateの順よりも、機能ごとのまとまりを重視する。

## **２．２　宣言**

## **２．２．１　変数宣言**

１行にひとつの変数を宣言する。

（悪い例）

int level, size;

（良い例）

int level; // インデント・レベル

int size; // テーブルの大きさ

宣言する場所は、ブロック"{ ... }"の先頭とする。ただし、forループのインデックスは例外で、カッコの中に書いて良い。

変数の初期化が単純な場合は、変数の宣言時に初期化する。

（例）

public void myMethod() {

int length = 10; // ブロックの先頭、初期化

for (int i = 0; i < length; i++) { // iはforループのインデックス

・・・

}

}

外側のブロックにある変数の宣言を隠してしまうローカル変数の宣言はしてはならない。たとえば、外側のブロックにある変数と同じ名前の変数を内側のブロックで宣言してはならない。

（悪い例）

func(){

int count;

if (*condition*){

int count; // ×

}

}

## **２．２．２　クラス／インタフェースの宣言**

クラス／インタフェースの宣言の記述方法を下記に示す。

class MyClass {

int var1; // フィールド

MyClass(int i) { // コンストラクタ

this.var1 = i;

}

someMethod() { // インスタンスメソッド

・・・

}

・・・

}

メソッド名とカッコ"("の間にスペースを入れない。

"{"は、宣言文の行の終わりに置く。

"}"は、行の先頭の位置に置く。ただし、"{"と"}"の間に文がない場合、"{}"と記述する。

メソッド間には空行を挿入する。

## **２．３　文（ステートメント）**

（１）単一ステートメント

１行にひとつのステートメントのみを書く。明らかな理由がなければ、カンマ演算子を使用して複数のステートメントをまとめてはならない。

（悪い例）

if (err) {

i++; j--; // ×

System.out.println(”error”), return -1; // ×

}

（２）複合ステートメント

・ブレース”{}”で囲まれるステートメントはインデントする。

・”{”は、複合ステートメントの始まる行の終わりに置く。

・”}”は、複合ステートメントの始まりと同じインデントの位置に置く。

・if-elseステートメントやforステートメントのような制御文の中に、ステートメントがひとつしかなくてもブレースで囲む。

（３）returnステートメント

return文にカッコを使わない。（条件式を書く場合は例外。）

（例）

return;

return success;

return (size ? size : defaultSize);

（４）ifステートメント

if (*condition*) {

*statements*;

}

if (*condition*) {

*statements*;

} else {

*statements*;

}

if (*condition*) {

*statements*;

} else if (*condition*) {

*statements*;

} else if (*condition*) {

*statements*;

}

（注）常にブレース”{}”を使用する。

（５）forステートメント

for (*initialization; condition; update*) {

*statements*;

}

空のステートメントは次のように書く。

for (*initialization; condition; update*) {

;

}

（注）常にブレース”{}”を使用する。

（６）whileステートメント

while (*condition*) {

*statements*;

}

空のステートメントは次のように書く。

while (*condition*) {

;

}

（注）常にブレース”{}”を使用する。

（７）do-whileステートメント

do {

*statements*;

} while (*condition*);

（８）switchステートメント

switch (*condition*) {

case ABC:

*statements*;

// 下へ続く

case DEF:

*statements*;

break;

case XYZ:

*statements*;

break;

default:

*statements*;

break;

}

default節は必ず付ける。それぞれのcase節、default節はbreak文で終わらせる。

意図的にcase節の終わりを通り越していくコードを書く場合には、それが誤りでないことをはっきりと示すため、break文を書く位置に”// 下へ続く”のようなコメントを書く。

（９）try-catchステートメント

try {

*statements*;

} catch (Exception e) {

*statements*;

} finally {

*statements*;

}

## **２．４　インデント**

インデントはスペース４文字単位で行う。（TABを使う場合は４文字を指定する。）

１行は８０文字以内とする。１行が８０文字を超える場合、下記の原則に従って改行する。

①　カンマの後で改行する。

②　演算子の前で改行する。

③　上位レベルの改行を優先する。

④　同じレベルにある前の行にそろえる。

⑤　以上の原則ではコードが読みにくくなる場合、単純にスペース４個のインデントを付ける。

（例１）

firstMethod(aaaaaaaaaa,

secondMethod(bbbbbbbbbb, // ４文字下げる

cccccccccc)); // さらに４文字下げる

（例２）

private static synchronized getTicketOfWorldCupByCupNoodle(int anArg,

Object anotherArg, String yetAnotherArg,

Object andStillAnother) { // ⑤のケース

...

}

## **２．５　スペース**

（１）空行

空行を使用して、関連したコードをまとめてプログラムを読みやすくする。たとえば次のような位置に使用する。

・クラスの宣言と宣言の間（ただし、基本的には１ファイルに１クラス）

・メソッドの定義と定義の間

・メソッドの中でローカル変数宣言と最初のステートメントの間

・メソッドの中で論理的なまとまりのあるコードとコードの間

（２）空白文字

下記のケースでは、空白文字を使用する。

・キーワードとカッコ”(”の間

・引数リストの中のカンマ”,”の後

・”.”と”->”演算子を除く２項演算子とオペランドの間

単項マイナス、インクリメント、デクリメント演算子のような単項演算子とオペランドの間はスペースを入れない。

・forステートメントの中の式と式の間

・キャストの後

　なお、メソッド名の後には空白を入れない

（例）

while (true) {

．．．

}

foo(i, j);

a += c + d;

a = (a + b) / (c \* d);

n++;

b = (int) a;

for (*expr1*; *expr2*; *expr3*)

## **２．****６　命名方法**

## **２．６．１　一般的な名前の付け方**

・読みやすく、覚えやすい名前を付ける。

・略語を使用しない。ただしURLやHTMLのように広く使われているものはよい。

・名前に自分の意味や処理内容を語らせること。

（悪い例）

ret = openFile(x);

if (ret) ．．．

（良い例）

success = openFile(FileName);

if (success) ．．．

## ２．６．２　パッケージ

・パッケージ名はすべて小文字とする。パッケージ名には「jp.co.ntt\_at」を付与する。

（本研修でのパッケージ名）

jp.co.ntt\_at.saa.trng2009.pims

jp.co.ntt\_at：AT社共通

saa：SAア

trng2009：新人研修

pims：プロジェクト名

## **２．６．３　クラス／インタフェース**

・クラス／インタフェース名は名詞を使用する。

・単語の先頭は大文字。単語の区切りは大文字とする。

・省略表記しない。

・インタフェースは"I"で始める。

・基底クラスは"B"で始める（本研修においてのみ適用）。

（悪い例）

public class makeCOFFEE

（良い例）

public class CoffeeMaker

public interface IInterface

## **２．６．４　メソッド**

・メソッド名は動詞を使用する。

・大文字と小文字を使用し、大文字を単語の区切りとする。小文字で始める。

・動作を表現する単語であること。

・そのメソッドが何をするかはっきりするような名前をつける。

・特に、フィールドの設定および取得に関しては、setXX、getXXという名称を使用する。

（悪い例）

Method1()

（良い例）

grindCoffeeBeans()

## **２．６．５　変数**

・大文字と小文字を使用し、大文字を単語の区切りとする。小文字で始める。

・簡潔でかつ意味の分かる名称をつけること。１文字変数は、原則使用禁止。

・ただし、カウンタなどの一時変数として、i、j、k、m、n、c、d、e のみを認める。

## **２．６．６　定数（static final）**

すべて大文字で記述する。単語の間はアンダースコア”\_”で区切る。

（例）

static final int MY\_CONSTANT\_VALUE = 0;

## **２．７　コメント**

## **２．７．１　実装コメント**

実装コメントは、コードの内容を説明するために記述する。

実装コメントには、ブロックコメント、１行コメント、後置コメント、コメントアウトの４種類がある。

（１）ブロックコメント

ファイル、メソッド、データ構造、アルゴリズムに関するコメントを記述する。

メソッド内に記述しても良い。

対象コードと同じインデントにする。

ブロックコメントの直前には、空行を入れる。

ファイルに関するコメントについては、「２．１．２　（1）ソースファイルを説明するコメント」を参照。

ソースコードのコメントアウトには使わない。

（例）

/\*

\* ブロックコメント

\* ブロックコメント

\*/

someMethod(firstParameter, secondParameter, thirdParameter);

（２）１行コメント

対象コードの１行前に記述する、１行で終わるコメント。

対象コードと同じインデントにする。

１行コメントの直前には、空行を入れる。

コメントが２行以上になる場合、ブロックコメントを使用する。

「ｆor文インデックスiを初期化（0をセット）」の様に「[target]の[what]を[do]（説明）」の形で日本語を用いて表記する事により、可読性を高める。

（例）

//変数iが100より大きい場合に処理（100は含まない）

if (i > 100) {

//someMethodの引数に変数iをセット

someMethod(i);

}

（３）後置コメント

対象コードの行末に記述するコメント。

後置コメントが連続する場合、後置コメントのインデントをそろえる。

後置コメントは変数以外では使用しない。

（例）

if (length > 0) {

return true; //文字入力あり

} else {

return false; //文字入力なし

}

（４）コメントアウト

ソースコードをコメントアウトするときに使う。IDEのコメント化機能がある場合はそちらを用いる（本研修ではeclipseのコメント化機能を用いる）。

（例）

if (length > 0) {

return length;

//} else if (length < 0) {

// return -1;

} else {

return 0;

}

## **２．７．２　ドキュメント用コメント**

ドキュメント用コメントは、javadocコマンドでドキュメントを自動生成するために記述する。

主要なタグについてのみ、記述方法を下記に示す。詳細についてはjavadocコマンドの仕様を参照。

（１）クラス仕様

１行目は、"/\*\*"とする。

２行目以降は、インデントされた"\*"で開始する。

最後の行は、インデントされた”\*/” とする。

javadocはHTML形式で出力されるため、明示的な改行には<BR>、パラグラフ分けには<P></P>を使用する。

@author、@versionを必須とする。

/\*\*

\* クラスの仕様

\*

\* @author 作者

\* @version バージョン

\* @see 参照するクラス名

\* @since バージョン

\*/

（２）メソッド仕様

様式は、クラス仕様と同じ。

引数が存在する場合は、全ての引数について、必ず@paramを書く。

型がvoid以外の場合は、必ず@returnを書く。

例外をスローする場合は、全ての例外について、必ず@exceptionを書く。

/\*\*

\* メソッドの仕様

\*

\* @param 引数名　説明

\* @return 説明

\* @exception クラス名　説明

\* @see 参照するクラス名

\*/

（３）フィールド

　フィールドの内容を記述する。

/\*\*

\* フィールドの説明

\*/

（４）例

　「３．コーディング例」を参照。

## **２．８　その他**

## **２．８．１　定数リテラル**

プログラムの中で定数リテラルを直接使用しない。ただし、forループのカウンタの初期値に現れる、-1、0、1等は良い。

（悪い例）

for (i = 0; i < 100; i++)

（良い例）

static final int MAX\_ENTRIES = 100;

．．．

for (i = 0; i < MAX\_ENTRIES; i++)

## **２．８．２　代入**

１行で複数の変数に値を代入しない。

代入演算子”=”を比較演算子”==”と間違いやすいところで使用しない。

変数への代入を別の文の一部で行わない。

（悪い例）

a = b = 1;

while (c++ = d++) ．．．

y = (a = b + c) + x;

（良い例）

while ((c++ = d++) != 0) ．．．

## **２．８．３　カッコの使用**

演算子の優先順位に依存した書き方をしないで、カッコを使用してその意味を明瞭にする。

（悪い例）

if (a == b && c == d)

（良い例）

if ((a == b) && (c == d))

## **２．８．４　３項演算子？：**

３項演算子”?:”の”?”の前にある２項演算子を含む式は、カッコで括る。

（例）

(x >= 0) ? x : -x

## **２．８．５　数値0との比較**

　数値0との比較は明示的に行う。

（悪い例）

while (balance) ．．．

（良い例）

while (balance != 0)

## **２．８．６　ヌル文字との比較**

　ヌル文字との比較は明示的に行う。

（悪い例）

while (c)

（良い例）

while (c != ’\u0000’)

## **２．８．７　フィールド**

フィールドは public にしない。

# **３．セキュアプログラミング**

開発するソフトウェアがセキュリティ脆弱性を持たないように、注意すべき点を記述する。本章の内容は、ＩＰＡセキュリティセンターのＷｅｂページ上のセキュア・プログラミング講座から抜粋した。詳細については、下記のＵＲＬを参照すること。

http://www.ipa.go.jp/security/awareness/vendor/programming/index.html

## **３．１　クラスの使用制限**

一般のプログラマが担当する部分について、使用を許可するＳＤＫのクラスを下記の安全なクラスに限定する。

・java.langパッケージのうち、Runtime、System、Thread、ThreadGroupの４つを除いたクラス

　ただし、java.lang.reflectとjava.lang.refパッケージは含まない。

・java.utilパッケージのクラス

　ただし、java.util.jarとjava.util.zipパッケージは含まない。

・java.mathパッケージのクラス

上記以外のクラスは熟練したプログラマだけが使用する。

## **３．２　パーミッション**

下記のパーミッションは危険なため、原則として与えない。

・java.security.AllPermission

・java.io.FilePermission("<<ALL FILES>>")

・java.lang.RuntimePermission

・java.security.SecurityPermission("setPermission")

・java.net.SocketPermission("\*","accept,connect")

## **３．３　カプセル化**

扱いに注意を要するクラスに対しては、リードオンリーのインタフェースを用意して、不用意なデータ更新を防ぐ。

（例）

public interface ISecretMoney\_ReadOnly {

long getIncome(); // Incomeフィールドを参照

long getExpence(); // Expenceフィールドを参照

}

public class SecretMoney\_Body {

private long Income;

private long Expence;

public long getIncome() .. // Incomeフィールドを参照

public long getExpence() .. // Expenceフィールドを参照

public void setIncome(long yen) .. // Incomeフィールドを設定

public void setExpence(long yen) .. // Expenceフィールドを設定

..

}

上記の例では、ISecretMoney\_ReadOnlyオブジェクトは、データを更新するsetIncomeやsetExpenceメソッドを使用できない。

## **３．４　シリアル化**

シリアル化のフォーマットは公開されているため、解読される恐れがある。

機密情報については、transientフィールドの指定を行い、シリアライズの対象からはずすか、暗号化により情報の漏洩を防ぐ。

ＲＭＩやＥＪＢは内部的にシリアライズの機能を使用しているため、機密情報については暗号化の対策が必要になる。

## **３．５　不正な継承**

クラスを継承して、そのクラスのオブジェクトであるふりをして、不正な処理を実行される恐れがある。

パスワードクラスのように重要なクラスは、finalクラスの宣言をすることにより、不正な継承を防ぐ。

（例）

public final class PassWord {

（省略）

}

## **３．６　レースコンディション**

マルチスレッドアプリケーションでは、レースコンディション問題が発生する恐れがある。レースコンディション問題とは、複数のスレッドの同期が図られていないときに、１つのスレッドが他のスレッドの動作に依存する時に発生するバグのこと。

複数のスレッドで共有する資源へのアクセスを同期させることにより、レースコンディション問題を防ぐ。下記の例では、Add100()メソッドをsynchronizedメソッドにすることにより、Add100()メソッドが並列に実行されるのを防いでいる。表示される「値:」は、常に300になる。

一方、synchronizedメソッドにしなかった場合、表示される「値:」は、実行のたびに異なる結果（200か300）になる。

（例）

public class ThreadTest implement Runnable {

int i = 100;

public static void main(String args[]) {

ThreadTest mt = new ThreadTest();

Thread th1 = new Thread(mt);

Thread th2 = new Thread(mt);

th1.start();

th2.start();

Thread.sleep(10 \* 1000); // 10秒待つ

System.out.println("値:" + i);

}

public synchronized void Add100() {

i += 100;

SomeHeavyJob(); // 何かの重い処理

}

public void run() {

Add100();

}

}

# **４．コーディング例**

/\*

\* Base64EncodeInputStream

\*

\* version 1.0 (2006/3/29)

\*

\* Copyright © NTT-AT 2006

\*/

import java.io.InputStream;

import java.io.IOException;

import java.util.StringTokenizer;

import javax.commerce.util.BASE64Encoder;

/\*\*

\* InputStreamの内容をBase64にエンコードする.

\* <pre>

\* Base64EncodeInputStream in =

\* new Base64EncodeInputStream(new InputStream());

\* int i = 0;

\* while ((i = in.read()) != -1) {

\* ...

\* }

\* </pre>

\* とすることでBase64にエンコードされたバイト列を抽出できる。

\*

\* @author Yasuda, Tetsu

\* @version 0.1 (2005/7/24)

\* @version 0.2 (2005/7/27)

\*/

public class Base64EncodeInputStream extends ExchangedInputStream {

/\*\*

\* 内部でコントロールするためのInputStream

\*/

private InputStream inputStream;

/\*\*

\* 内部で使用するBase64処理用クラス

\*/

private BASE64Encoder base64encoder;

/\*\*

\* 内部バッファの大きさを規定した整数

\*/

static final int BUFFER\_SIZE = 1023;

/\*\*

\* InputStreamを指定するコンストラクタ。

\*

\* @param inputstream 入力もとのストリーム

\*/

public Base64EncodeInputStream(InputStream input) {

// 入力用バッファの生成

this.inputBuffer = new byte[BUFFER\_SIZE];

this.inputStream = input;

this.base64encoder = new BASE64Encoder();

}

/\*\*

\* ストリーム接続を終了する。

\*/

public void close() throws IOException {

inputStream.close();

base64encoder = null;

outputBuffer = null;

inputBuffer = null;

}

/\*\*

\* InputStreamから入力されたバイト列にBase64エンコード処理を

\* 施すメソッド。

\* バッファの大きさはBUFFER\_SIZEで規定される。

\* outputBufferにエンコード処理されたバイト列が入る。

\* 有効なバイト数はreadableに格納される。

\*/

protected void getOutputBuffer() throws IOException {

/\*

\* inputStreamからinputBufferへバイト列を読みこむ。

\* readableには読みこんだバイト数が入る。

\*/

readable = inputStream.read(inputBuffer, 0, BUFFER\_SIZE);

// ストリームの最後まで来てしまった場合

if (readable == -1) {

return;

}

/\*

\* inputBufferの内容をbufferへコピーする。

\* readable < BUFFER\_SIZEの場合を考慮した。

\*/

byte buffer[] = new byte[readable];

for (int i = 0; i < readable; i++) {

buffer[i] = inputBuffer[i];

}

/\*

\* Base64エンコード処理を行う。

\* BASE64Encoderでは76文字毎になぜか改行コード(0x0d0a)を

\* 挿入されてしまうのでそれを除去する。

\*/

String str = base64encoder.encodeBuffer(buffer);

String nocrlf = "";

// 改行コードで分割

byte crlf[] = {13, 10};

StringTokenizer token = new StringTokenizer(str, new String(crlf));

while (token.hasMoreTokens() == true) {

nocrlf = nocrlf + token.nextToken();

}

// 改行コードを除去した文字列でバイト列を組みなおす

outputBuffer = nocrlf.getBytes();

// 読みこめるのはoutputBufferの長さだけ

readable = outputBuffer.length;

// outputBufferが更新されたのでreadingも初期化する。

reading = 0;

}

}

# **付録１　Javadocについて**

ここでは、Javadocを記述すべきクラス、メソッド、フィールドおよび記述内容について、まとめている。

# **付録１．１　Javadocを記述すべきクラス、メソッド、フィールド**

Javadocを記述すべきクラス、メソッドおよびフィールドの条件を説明する。

**<クラス>**

* 新規作成、改造したすべてのクラス（以下のクラスは記述しないこと）

**<記述しないクラス>**

* + 完全に流用、つまり、一箇所も手を入れる必要がないクラス
  + まったく使用しないクラス

**<メソッド>**

* 新規作成、改造および流用したすべてのメソッド。  
  ただし、以下のメソッドは記述しない。

**<記述しないメソッド>**

* + 完全に流用するクラス内のメソッド
  + まったく使用しないメソッド

**<フィールド>**

* 新規作成、流用したすべてのフィールド。  
  ただし、以下のフィールドは記述しない。

**<記述しないフィールド>**

* + 完全に流用するクラス内のフィールド
  + まったく使用しないフィールド

# **付録１．２　Javadocの記述レベル**

Javadocの記述レベルおよび記述するパラメータについて、説明する。

**<記述レベル>**

Javadocは、設計書の位置付けであると共に、技術移管時の説明書として使用することを前提としている。ソースコードを初めて読む人が、理解の補助として活用できることが前提となっている。そのため、Javadocの記述例を作成し、その記述を基に、記述レベルの意識合わせを行うこととする。

**<記述パラメータ>※複数の項目に関しては、内容ではなくタグを複数許可するかどうか。**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| パラメータ | 記述個所 | 記述箇所に対する記述内容 | 必須 | 複数 |
| パラメータ名 | クラス | クラスの説明（流用クラスについては、Javadocを記述しない） | - | - |
| メソッド | メソッドの説明 | - | - |
| フィールド | フィールドの説明 | - | - |
| @see | クラス | 関連項目（流用クラスについては、Javadocを記述しない） | - | ○ |
| メソッド | 関連項目 | - | ○ |
| フィールド | - | ○ |
| @author | クラス | ・ クラス内メソッドの作成者、改造者の名前をすべて記述すること。なお、改造クラスの場合、作成者の先頭に「NTT-AT」を記述すること  例: @author: Author1, Author2, Author3  \*　流用クラスについては、Javadocを記述しない | ○ | ○ |
| メソッド | ・ それぞれのメソッドの作成者、改造者の名前をすべて記述すること  ・ 改造メソッドの場合、作成者として「NTT-AT」を先頭に記述すること  ・ 流用メソッドの作成者は「NTT-AT」とすること  \*1 Javadocとしては表示されないが、ソースコードを見て認識できることが意味を持つ  \*2 メソッドの作成者は、原則としてクラスの@authorと同一の文字列を使用すること。但し、クラスの@authorが「Takashi Ono(t-ono)」のように、()付けで記述されている場合、「t-ono」としてもよい | - | ○ |
| @version | クラス | ・案件ごとに決めること  ・バージョンを更新した場合は、編集ではなく追記すること  \* 流用クラスについては、Javadocを記述しない | ○ | ○ |
| メソッド | 案件ごとに決めること  \* Javadocとしては表示されないが、ソースコードを見て認識できることが意味を持つ | - | ○ |
| @since | クラス | 案件ごとに決めること  \* 流用クラスについては、Javadocを記述しない | ○ | ○ |
| メソッド | 案件ごとに決めること | - | ○ |
| @param | メソッド | メソッドの引数（引数の扱いとして、下限、上限を設ける場合は明記すること） | ○※ | ○ |
| @return | メソッド | メソッドの返り値（正常時及び異常時の返り値について記述する） | ○※ | - |
| @throws | メソッド | スローする例外（Exception発生状況も記述すること） | ○※ | - |

※必要がない場合は省略可

**<記述例>**

|  |  |  |
| --- | --- | --- |
| パラメータ | 記述個所 | 記述例 |
| @see | クラス | @see 関連項目  \*　任意でIDE（Eclipse）による自動生成も可能 |
| メソッド | @see 関連項目  \*　任意でIDE（Eclipse）による自動生成も可能 |
| フィールド |
| @author | クラス | @author 著者  @author: Author1, Author2, Author3 |
| メソッド | 改造メソッドの場合  @author: NTT-AT,Author1,Author2,Author3  流用メソッドの作成者は「NTT-AT」とすること  　　@author: NTT-AT |
| @version | クラス | @version バージョン  @version 1.6 |
| @since | クラス | @since バージョン  @since 1.0 |
| @param | メソッド | @param 引数名 引数の説明  @param staffData 従業員情報 |
| @return | メソッド | @return 戻り値名 戻り値の説明  @return new ActionForward("/WEB-INF/pages/menu.jsp");  ActionForword型のオブジェクト（次の表示先をセット） |
| @throws | メソッド | @throws Exception名 例外の説明  @throws SQLException データベースからの例外をスローします |

# **付録１．３　Javadoc生成の際の注意**

Javadoc生成の際には、メソッドの中身はコメントのみでも構わないが、クラス宣言、メソッド宣言（継承、実装関係、アクセス指定子、戻り値、引数、例外等）、フィールドは、実装と相違なく記述してから生成すること。

getter、setterの記述については、下記記述例を参考にして下さい。また@throwsパラメータについては適宜追記して下さい。

・getter記述例

　　getterは説明と@returnパラメータを記述します。

/\*

\* フィールドを取得します。

\* @return フィールド

\*/

public int getField() {

return field;

}

・setter記述例

　　setterは説明と@paramパラメータを記述します。

/\*\*

\* フィールドを設定します。

\* @param field フィールド

\*/

public void setField(int field) {

this.field = field;

}

**付録２　文章規定**

# **付録２．１　文章規定**

## **２．１．１ 通常コメント記入例**

　文章全体の規定としては、日本語の場合は「、」や「。」を使い、アルファベット表記の場合は「,」や「.」を使う事とする。

　作成者等の名前は、文法に則りFirstnameFamilyname

　通常コメントにおける表記では、原則、体言止めを用いる。1行コメントでは「、」や「。」は原則使わない。コメントの表記方法については2.７.1 (2) １行コメントの例を参考にする。

（例）

　public void myMethod() {

int length = 10; //ｆor文継続条件lengthを初期化（10をセット）

for (int i = 0; i < length; i++) { //ｆor文インデックスiを初期化（0をセット）

・・・

}

}

/\*

\* ブロックコメント

\* ××するメソッド

\*/

someMethod(firstParameter, secondParameter, thirdParameter);

## **２．１．２ Javadoc記入例**

　Javadocにおける説明部では、原則「です・ます調」を用いる。

　Javadocタグの後は半角スペースを入れ、体言止めを用いる。

（例：必須条件のみ）

/\*\*

\* HelloJavadocという文字を表示します。

\* クラスの詳細説明です。このクラスでは他のクラスを呼び出しません。<BR>

\* 特記事項はありません。

\* @author FirstnameFamilyName

\* @version 0.1 (2009/6/25)

\* @version 0.2 (2009/6/30)

\* @since 0.1(2009/6/25)

\*/

public class HelloJavadoc{

private String comment;

/\*\*

\* コンストラクタです。引数commentで受け取った値をフィールドcommentにセットします。

\*/

public HelloJavadoc(String comment) throws Exception{

this.comment = comment;

}

/\*\*

\* 値を表示するメソッドです。

\*/

public void purintWord() throws Exception{

System.out.println(this.comment);

}

}

（例：全条件）

/\*\*

\* HelloJavadocという文字を表示します。

\* クラスの詳細説明です。このクラスでは他のクラスを呼び出しません。<BR>

\* 特記事項はありません。

\* @see

\* @author FirstnameFamilyName,HanakoSato

\* @author TaroYamada

\* @version 0.1 (2009/6/25)

\* @version 0.2 (2009/6/30)

\* @since 0.1(2009/6/25)

\*/

class HelloJavadoc{

private String comment;

/\*\*

\* コンストラクタです。

\* 引数commentで受け取った値をフィールドcommentにセットします。

\* author TaroYamada

\* @param comment 引数です

\* @throws Exception 予期せぬ例外

\*/

public HelloJavadoc(String comment) throws Exception{

this.comment = comment;

}

/\*\*

\* 値を表示するメソッドです。

\* author TaroYamada

\* @return comment　String型のフィールドcommentを返します;

\* @throws Exception 予期せぬ例外

\*/

public void purintWord() throws Exception{

System.out.println(this.comment);

return this.comment;

}

}

**付録３　用語集**

# **付録３．１　用語集表記規則**

　用語集は体言止めを用いて、アルファベットは半角を用いる。参考URLがある場合は文末に記載する。必要な場合は適宜改行する。文頭は１文字全角スペースを入れる。

# **付録３．２　用語集**

|  |  |  |
| --- | --- | --- |
| 語句 | 区分 | 意味 |
| シリアル化 | セキュア | シリアル化とは、オブジェクトの状態を永続化または転送できる形式に変換するプロセスのこと。  　シリアル化を補完するプロセスとして逆シリアル化があり、これは、ストリームをオブジェクトに変換するプロセス。  　これらのプロセスを組み合わせて使用することで、データを簡単に格納したり、転送したりできる。 |
| Javadoc | ソフトウェア | Javadocとは、サン・マイクロシステムズが開発したコンピュータソフトで、JavaのソースコードからHTML形式のAPI仕様書を生成する。URL（<http://ja.wikipedia.org/wiki/Javadoc>） |
| **パーミッション** |  | パーミッションとは、コンピュータのハードディスクなどに保存されているファイルやディレクトリに対するユーザのアクセス権のこと。URL（<http://e-words.jp/w/E38391E383BCE3839FE38383E382B7E383A7E383B3.html>） |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |