

# ドキュメントサンプル集(標準化編)

## コーディング規約(Java)

---

# 目次

1. Java Coding Rule.....	1
1.1. ファイル名規約.....	1
1.1.1. 拡張子規約.....	1
1.2. プログラム全体構成基準.....	1
1.2.1. ソースプログラム全般に関する標準.....	1
1.2.2. 記述順序に関する基準.....	2
1.2.3. 先頭コメント.....	3
1.2.4. package 文.....	3
1.2.5. import 文.....	4
1.2.6. クラス／インタフェース定義.....	4
1.2.7. 改版履歴.....	7
1.3. インデント.....	8
1.4. コメント.....	9
1.4.1. インプリメンテーションコメントの書式.....	9
1.4.2. ドキュメンテーションコメントの書式.....	12
1.5. 宣言部.....	15
1.5.1. 1行に記述する宣言.....	15
1.5.2. 修飾子.....	15
1.5.3. 配列の宣言.....	16
1.5.4. 初期化.....	16
1.5.5. 記述場所.....	16
1.5.6. クラス／インタフェースの宣言.....	16
1.6. ステートメント.....	17
1.6.1. 1ステートメント.....	17
1.6.2. ブロックステートメント.....	17
1.6.3. return 文.....	17
1.6.4. if, if-else, if else-if else 文.....	18
1.6.5. for 文.....	19
1.6.6. while 文.....	19
1.6.7. do-while 文.....	19
1.6.8. switch-case 文.....	20
1.6.9. try-catch 文.....	21
1.7. 空白.....	22
1.7.1. 空行.....	22
1.7.2. 空白.....	23

# 1.Java Coding Rule

## 1.1.ファイル名規約

### 1.1.1.拡張子規約

Java 関連のファイルに対し、拡張子は以下の標準に従うものとします。

表 1-1

ファイル種別	拡張子
Java ソースファイル	.java
Java バイナリファイル	.class

## 1.2.プログラム全体構成基準

### 1.2.1.ソースプログラム全般に関する標準

Java ソースプログラムは以下の標準に基本的に従うものとします。

- ソースファイルは ShiftJIS にて記述するものとします。
- 但し、ShiftJIS にて記述した際に問題が発生する文字(“~” など)を指定する際には Unicode-Escape の利用を可とします。この際、どの文字を表しているのかをコメントとして記述するものとします。
- ソースファイル内の 1行は 120byte 以内で記述するものとします。この上限を超える場合には適宜改行することとします。
- 1 ファイルに記述されるソースプログラムのライン数は、2000 行以内とし、これ以上のライン数にならないようプログラムを設計することとします。  
200行以上のメソッドは可能な限り分割するものとします。但し、ロジック上不可能な場合、および分割することにより可読性が失われる場合を除くものとします。
- 1ファイルに記述されるソースの全行数に対するコメント行数の割合は 30% 程度とします。
- 1メソッド内の全行数に対するコメント行数の割合は 30% 程度とします。
- 1つの Javaソースファイルの中には1つの publicクラスまたは publicインタフェースのみを含むものとします。
- 1つの Javaソースファイルの中には関連する複数の privateクラス、または privateインタフェースを記述することが出来るものとします。
- Java ソースのファイル名は、そのファイルに含まれる public クラスまたは public インタフェースの名前を使用するものとします。

## 1.2.2.記述順序に関する基準

Java ソースは以下の記述順序に従って記述します。

- (1) 先頭コメント
- (2) package名
- (3) import 文
- (4) クラス定義
- (5) (改版履歴)

また、セクション(section)が容易に判別できるように、クラスの先頭で下記のようにブロック間に1行分のスラッシュ(slash 斜線)でコメント行を入れるものとします。

### イメージ

```
// 先頭コメント
```

```
package パッケージ名;
```

```
import 宣言
```

```
////////////////////////////////////
```

```
Class 又は Interface 定義
```

```
改版履歴
```

### 1.2.3.先頭コメント

Java ソースの先頭を含む「先頭コメント」には以下の形式で記述します。

先頭コメントには、以下の項目を含む必要があります。

- コピーライト (copyright)
- システム名・サブシステム名
- クラス名( このソースファイルを代表する public のクラスまたはインタフェース名 )
- 機能概要

#### 例

```
////////////////////////////////////  
//  
// Copyright 2005 DHEE Corporation. All Rights Reserved  
//  
////////////////////////////////////  
  
/* =====  
 * システム名      : ProjectZERO  
 * サブシステム名 :  
 * Class 名       :  
 * ファイル名     : $RCSfile$  
 * 概要           :  
 * ===== */
```

### 1.2.4.package 文

先頭コメントに続いてpackage名を記述するものとします。

実際のパッケージ名は命名規約を参照してください。

#### 例

```
package com.nec.jp.projectzero.xxxxx;
```

### 1.2.5.import 文

import 文は package 文に続いて記述するものとします。

#### 例

```
import com.nec.jp.orteus.struts.foundation.bean.LogicData;  
import com.nec.jp.orteus.xaf.foundation.*;
```

“\*”<sup>りよう</sup> を利用したパッケージの一括<sup>ばっけーじ</sup> import<sup>いつかつ</sup> を行<sup>おこなわず</sup>わずに、利用<sup>りよう</sup>するクラス<sup>くらす</sup>を全<sup>すべて</sup>て記述<sup>きじゆつ</sup>するものとします。

### 1.2.6.クラス／インタフェース定義

クラスおよびインタフェース<sup>いんたふえーす</sup>の定義<sup>ていぎ</sup>は以下<sup>いか</sup>の順序<sup>じゆんじょ</sup>に従<sup>したがって</sup>って記述<sup>きじゆつ</sup>しなければなりません。

- (1) クラス／インタフェースドキュメンテーションコメント<sup>くらす いんたふえーすどきゅめんてーしょんこめんと</sup>
- (2) class/interface 宣言文
- (3) クラス／インタフェースインプリメンテーションコメント(class/interface implementation comment)
- (4) クラス変数
- (5) メンバ変数(member)
- (6) コンストラクタ (constructor)
- (7) メソッド(method)

#### クラス／インタフェース ドキュメンテーションコメント

- コメントの内容については「1.4 コメント」を参照してください。

#### class/interface 宣言文

- ドキュメンテーションコメントに続いて class または interface の宣言文を記述するものとします。

#### 例

```
public class Foo extends Bar implements Baz, Woo;
```

#### 例

```
public interface Foo extends Woo;
```

1行<sup>ぎょう</sup>の文字数制限<sup>もじすうせいげん</sup>を超過<sup>ちやうか</sup>しない限り<sup>かぎり</sup>改行<sup>かいぎょう</sup>は行<sup>おこなわず</sup>わず、改行<sup>かいぎょう</sup>が必要な場合<sup>ひつよう</sup>には “extends” または “implements” から改行<sup>かいぎょう</sup>するものとします。

## クラス／インタフェース インプリメンテーションコメント

- クラスまたはインタフェース全体にかかわるコメントで、ドキュメンテーションコメントとしては適当でないようなコメントがあれば必要に応じて記述します。

## 定数

- 最初に public 定数、次に protected 定数、次に修飾子なし(同一パッケージ内で参照可能) 定数、最後に private 定数を記述します。

## クラス変数

- 最初に public 変数、次に protected 変数、次に修飾子なし(同一パッケージ内で参照可能) 変数、最後に private 変数を記述します。

## メンバ変数

- 最初に public 変数、次に protected 変数、次に修飾子なし(同一パッケージ内で参照可能) 変数、最後に private 変数を記述します。

## コンストラクタ

- クラスをインスタンス化(instance)する際に必要な初期化を行うコンストラクタの実体を記述します。但し、初期化が必要のないクラスの場合にはコンストラクタ自体の記述を省略するものとします。
- 一方、static メソッドしか持たないユーティリティクラス (utility class) の場合には、初期化が必要ない場合でも利用クラスからインスタンス化されないよう private のコンストラクタ(constructor)を記述するものとします。

## メソッド

- メソッドの記述順は、ソースコードの読みやすさを考慮して、アクセス(access)制御御飾りよりも「メソッドが実現する機能」によってグループ化して記述します。
- 上記の各セクションは容易に判別できるようにするため、下記のように各セクション間に1行分の「/」でコメントを入れることとします。また、このコメント行は各宣言部と同じ幅のインデント(indent)を行うこととします。

## 例

```
public class AsXxxxYyyy{
    定数宣言
    //////////////////////////////////////
    クラス変数宣言
    //////////////////////////////////////
    インスタンス変数宣言
    //////////////////////////////////////
    コンストラクタ
    //////////////////////////////////////
    一般メソッド
    //////////////////////////////////////
    setter/getter
    //////////////////////////////////////
    inner クラス定義
}
```

- inner クラス自体の宣言順序は、上記に倣うものとします。
  - inner クラス内の宣言順序も、上記に倣うものとします。
  - inner クラスが複数あるときは、各 inner クラスもスラッシュコメント行で区切ることとします。
- メソッド宣言部で、メソッドがさらに意味のあるブロックに分けられる場合は、それもスラッシュコメント行で区切ります。

### 例

```
// ユーザに関する処理メソッド群
```

```
.....
```

```
// グループに関する処理メソッド群
```



1.2.7.改版履歴

Java <sup>そーすまっぴ</sup>ソース末尾には、<sup>かいはんりれき</sup>改版履歴<sup>いれる</sup>を入れることとします。

### 例

[illegible]

## 1.3. インデント

ソースコードのインデントは以下の基準に従うものとします。

- インデントは半角スペース(space)4文字とします。TABは環境により幅が異なる可能性があるため原則利用しないものとします。
- 1文を複数行にまたがって記述する必要がある場合には以下の原則に従うものとします。
  - 継続行は適当な数(最低 2スペース)のインデントを行うものとします。
  - 但し、本来のインデントと同じ半角スペース 4文字分のインデントは可読性が著しく低下するため利用してはなりません。
  - 「,」の後で改行します。
  - 演算子の前で改行します。
  - 低位の構造で改行するよりは、高位の構造で改行します。
  - 前行の開始構造と同じ構造で新しい行を開始します。
  - 上記原則によってコードが読みにくいものになる場合は、または行長に対して適当な空きがない場合は、インデントするものとします。

以上により、以下のコーディングサンプルでは、上のサンプルコードの方がよりよい改行となります。

### 良い例

```
longName1 = longName2 * (longName3 + longName4 - longName5)
    + 4 * longName6;
```

### 悪い例

```
longName1 = longName2 * (longName3 + longName4
    - longName5) + 4 * longName6;
```

ブロックを構成する中括弧('{', '}')に関して、中括弧を開く際は前の行にぶら下げることとします。

中括弧を閉じる際は、中括弧を開いた行のインデントと、インデントが同じ深さになるようにすることとします。

また、中括弧の中に書かれたコードは必ず、字下げされることとします。

### 例

```
if (boolean-expression) {
    // some logic
}
```

## 1.4.コメント

Javaプログラムには、インプリメンテーションコメントとドキュメンテーションコメントの2種類のコメントがあります。

インプリメンテーションコメントは、コードそのもの、または特別なコードの記述についての説明を記述します。

ドキュメンテーションコメントは、コードの記述そのものという観点ではなく、コードが実現する機能の仕様について記述するもので、ソースコードをもたない開発者に読まれることを前提に記述します。

コメントはソースコードそのものからは読み取ることができない追加情報を記述するものとします。

### 1.4.1.インプリメンテーションコメントの書式

インプリメンテーションコメントには、block、single-line、trailing、end-of-line の4つのスタイルがあります。

#### (1)Block コメント

Block コメントによって、ファイル、メソッド、データ構造、アルゴリズム(algorithm)および仕様についての説明を記述します。

ファイルの先頭、メソッドの先頭、あるいはメソッド中でも記述します。

メソッド内に記述される Block コメントは、ソースコードと同じレベルのインデントを行うものとします。

Block コメントの前には空行を1行はさみ、ソースコードと分けて記述するものとします。

#### 例

```
/*  
 * この HashMap は Foo -> Bar の Map を保持し、Buz メソッドにおいて  
 * 内容が走査される。  
 */
```

#### Single-Line コメント

短いコメントは、Single-Line コメントによって記述します。

同レベルのインデントの途中に Single-Line コメントを入れる場合には、直前に空行を1行はさみ、ソースコードと分けて記述するものとします。

#### 例

```
if (boolean-expression) {  
    /* if-clause の処理内容説明を記述します */  
}  
  
/* このコメント以降の処理ブロックに対する内容説明を記述します */
```

## Trailing コメント

非常に短いコメントは、それ自身が説明するソースコードと同じ行に記述することができます。

ただし、Trailing コメントより End-of-line コメントを利用して記述することを推奨します。

### 例

```
if (a == 2) {  
    return true;                /* 短いコメント */  
} else {  
    return isPrime(a-1);        /* 同一インデントで短いコメント */  
}
```

## End-Of-Line コメント

非常に短いコメントはそれ自身が説明するソースコード同じ行に記述するものとします。ただし、連続した複数行にわたるコメントを記述する目的で使用してはなりません。

### 例

```
if (foo > 1) {  
    .....  
    // この行はコメント  
} else {  
    return false;                // return 動作を説明するコメント  
}  
  
//if (bar > 1) {  
//    .....  
//    // この行はコメント  
//} else {  
//    return false;                // return 動作を説明するコメント  
//}
```

ただし、複数の行の記述を無効にするために利用する際には利用可能とします。

また、諸般の事情で実装が不可能な部分を明記するために TODO: コメントを記述するための利用も可とします。TODO コメントはコーディング完了のタイミングでソースコード上に残ってはいけません。

### 例)

```
try {  
    // some-logic  
} catch (FooException e) {  
    // TODO: エラー処理方式未決のため未実装  
} catch (BarException e) {  
    // TODO: エラー処理方式未決のため未実装  
} finally {  
    try {  
        if (stmt != null) {  
            stmt.close();  
        }  
    } catch (SQLException e) {  
    }  
    stmt = null;  
}
```

- ドキュメンテーションコメントはメソッド、または、コンストラクタを記述するブロックの中には置かないこととします。(Java ソースに関するドキュメント作成ツールである javadoc は、ドキュメンテーションコメントをコメント後の最初の宣言文と関連づけて解釈する為です。)

## クラス

クラスの javadoc は以下の内容を記述します。

表 1-1

記述内容	記述の必要性	
	処理クラス	データ保持クラス
そのクラスが何のためにあるのか	有	有
クラスが行う処理の詳細説明	有 自明の場合は無	無
クラスの利用例	有 自明の場合は無	無
クラスの推奨されない利用例	無	無
前提条件 (利用するにあたって先に生成しておかなければいけないクラスなど)	無	無

クラスの処理説明、利用例、前提条件に関しては、メソッドの記述よりも粒度が粗くても構いませんが、クラスの主要な機能を利用する為に必要な情報を記述します。

関連して参照したほうがよいクラスがあれば、必要に応じて @see タグを利用します。

## メソッド

メソッドの javadoc は以下の形式で記述し、最低限以下に示した javadoc タグを利用します。

### 例

```
/**
 * メソッドの役割を表す簡潔な一文(‘。’で終わらせる)
 * 詳細説明(複数行記述可能)
 * .....
 * .....
 * @param 引数名1 役割1
 * @param 引数名2 役割2
 * .....
 * @return 返却値の意味
 * @exception 例外1 throwされる場合の条件／説明1
 * @exception 例外2 throwされる場合の条件／説明2
 * .....
 */
```

詳細説明には必要に応じて以下の内容を含めます。

- メソッドの事前条件、事後条件
- もしあれば、override に関する制約条件。すなわち、override 必須か、override 不可か、など。
- override によるカスタマイズ (customize) を前提とする場合は、動作がどのようにカスタマイズされるかの情報も記述します。
- 

例外には以下の内容を記述します。

- throw する可能性のある全ての例外(throws 句に現れないものも含む)
- 各例外の発生条件
- 例外が例外コードを持ち、コード体系で定められた意味に対し、そのメソッド特有の用法が付加される場合には、そのコード説明

例) コード体系で定められたコード ERROR\_ACCESS\_DENIED の意味が「アクセス権エラー」であるが、メソッドAでは同コードを「ユーザ作成に関するアクセス権がない為に発生」の意味で使用しているような場合

その他、関連して参照したほうが良いクラスまたはメソッドがあれば、必要に応じて @see タグを記述します。

## フィールド

フィールドの javadoc は、原則として1行で記述します。

### 例

```
/** ユーザ名を表す */
```

```
String name;
```

---



## 1.5.宣言部

### 1.5.1.1 行に記述する宣言

1行に1つの宣言を行うこととします。

#### 良い例

```
int width;  
int height;
```

#### 悪い例

```
int width, height;
```

かた ことなる せんげん 1ぎょう きじゅつ  
型の異なる宣言を1行に記述してはなりません。

#### 悪い例

```
int width, height[];
```

### 1.5.2.修飾子

かくせんげん たいするしゅうしよくこ い か じゅん きじゅつ  
各宣言に対する修飾子は以下の順に記述します。

1. public, protected, private または無し
2. abstract
3. static
4. final
5. synchronized
6. native
7. strictfp
8. transient
9. volatile

### 1.5.3. 配列の宣言

配列はメンバ名ではなく型名に “[ ] ” を付加して宣言するものとします。

#### 良い例

```
int[] plotPointsX;  
int[] plotPointxY;
```

#### 悪い例

```
int plotPointsX[];  
int plotPointsY[];
```

### 1.5.4. 初期化

ローカル変数の初期値は宣言文で与えることとします。

### 1.5.5. 記述場所

各宣言は該当のメンバを使用する処理ブロックの直前以前に記述されていれば良いものとします。

また、for ループのインデックス変数は for 構文中に宣言も可能とします。

上位レベルブロックで使用されている変数名を、下位レベルブロックで再宣言し利用することは可読性を損なうため原則として禁止します。

### 1.5.6. クラス/インタフェースの宣言

メソッド名とメソッドの引数記述を開始する “ ( ” の間にはスペースを置かないこととします。

ブロックを開始する “ { ” は宣言文と同一行上にあることとします。

ブロックを終了する “ } ” は対応するブロック開始宣言文と同一のインデントであることとします。

各メソッド宣言の間は空行によって分離されていることとします。

## 1.6.ステートメント

### 1.6.1.1ステートメント

1行には1ステートメント（statement）を記述するものとします。

#### 良い例

```
count--;  
args++;
```

以下のように複数ステートメントを1行に記述してはなりません。

#### 悪い例

```
count--; args++;
```

### 1.6.2.ブロックステートメント

ブロックステートメントとは、‘{’ および ‘}’ に囲われたステートメントのリストのことであり、以下の記述規則に従うものとします。

- “{” に囲まれたステートメントは、“}” そのものよりも1レベルインデントして記述します。
- “{” は構成ステートメントの開始行の最後に記述します。
- “}” は構成ステートメントの開始行と同じインデントで記述します。

プログラムの制御構造を記述する if, while, for, do-while の記述において、対象ステートメントが 1文の場合においてもブロックを利用するものとします。

### 1.6.3.return 文

return文では、原則として括弧は使用しないこととします。

但し、返却値をより明確にするための記述として使用する場合はこの限りではありません。

#### 例

```
return;
```

#### 例

```
return mydisk.size()  
;
```

#### 例

```
return (size > 0 ? size : DEFAULT_SIZE);
```

#### 1.6.4.if, if-else, if else-if else 文

if 文は以下の形式で記述します。

##### 例

```
if (boolean-expression) {  
    statement;  
    .....  
}
```

##### 例

```
if (boolean-expression1) {  
    statement;  
    .....  
} else if (boolean-expression2) {  
    statement;  
    .....  
} else {  
    statement;  
    .....  
}
```

##### 例

```
if (boolean-expression1) {  
    statement;  
    .....  
} else if (boolean-expression2) {  
    statement;  
    .....  
} else {  
    statement;  
    .....  
}
```

### 1.6.5.for 文

for 文は以下の形式で記述するものとします。

#### 例

```
for (initialization; boolean-expression; updates) {  
    statement;  
    .....  
}
```

くりかえしじこう すてーとめんと そんざい すべて しより 繰り返し実行されるステートメントが存在しない( 全ての処理が、initialization, condition, および update において完結する ばあい 場合には、ブロックを使用せず以下の形式で記述するものとします。

#### 例

```
for (initialization; boolean-expression; updates)  
    ;
```

### 1.6.6.while 文

while 文は以下の形式で記述するものとします。

#### 例

```
while (boolean-expression) {  
    statement;  
    .....  
}
```

### 1.6.7.do-while 文

do-while 文は以下の形式で記述するものとします。

#### 例

```
do {  
    statement;  
    .....  
} while (boolean-expression);
```

## 1.6.8.switch-case 文

switch 文は以下の形式で記述されるものとします。

### 例

```
switch (expression) {  
  case constant1 :  
    statement;  
    .....  
    // fall-through  
  case constant2 :  
    statement;  
    .....  
    break;  
  case constant3 :  
    .....  
    break;  
  default :  
    statement;  
    .....  
    break;  
}
```

1 つの case<sup>くうち</sup>句内で処理が完結せず次の case<sup>しより</sup>に処理が流れる場合 (case<sup>かんけつ</sup>ブロックに break<sup>つぎ</sup>がない場合)、通常<sup>ふつじょう</sup>break が記述される場所にコメント ( // fall-through ) を記述することとします。

switch 文においては必ず default を記述するものとし、また default ケースにおいても break を記述することとします。

### 1.6.9.try-catch 文

try-catch 文は以下の形式で記述されるものとします。

#### 例

```
try {  
    statement;  
    .....  
} catch (ThrowableClass e) {  
    statement;  
    .....  
}
```

finally 節を使用する場合には、以下の形式で記述されるものとします。

#### 例

```
try {  
    statement;  
    .....  
} catch (ThrowableClass e) {  
    statement;  
    .....  
} finally {  
    statement;  
    .....  
}
```

## 1.7.空白

### 1.7.1.空行

論理的につながりのあるコードの組を空行によって区切ることで結合度の強い行をまとめることになり、ソースコードの可読性が向上します。

2行の連続した空行は以下の状況において使用します。

- ソースコードの2つのセクションを区切る場合
- 2つのクラスまたはインタフェース定義の間を区切る場合

1行の空行は以下の状況において使用します。

- 2つのメソッド定義の間を区切る場合
- メソッド内のローカル変数定義と処理を行うコードとの間を区切る場合
- ブロックコメントまたはシングルラインコメントの前
- メソッド内の論理的な組を区切って、読みやすさ向上がさせる場合



## 1.7.2.空白

空白は以下の状況において使用します。

- 引き数リストのカンマの後ろ
- ‘(‘ の前、’)’ の後
- 2項以上の演算子の前後

### 例

```
a = (a + b) / (c * d);  
while (d++ == s++) {  
    n++;  
}
```

for 文の各条件子

### 例

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

一方、以下の状況においては、空白を入れてはなりません。

- キャスト演算子とキャスト対象の間

### 例

```
myMethod( (int)a, (String)s );
```