社内研修Java実習におけるコーディングルール

2023年3月16日更新

目次

[1 導入 4](#_Toc129856504)

[1.1 用語についての注記 4](#_Toc129856505)

[1.2 ガイドについての注記 4](#_Toc129856506)

[2 ソースファイルの基本事項 5](#_Toc129856507)

[2.1 ファイル名 5](#_Toc129856508)

[2.2 ファイルエンコーディング：UTF-8 5](#_Toc129856509)

[2.3 特殊文字 5](#_Toc129856510)

[2.3.1 空白 5](#_Toc129856511)

[2.3.2 特別なエスケープシーケンス 5](#_Toc129856512)

[2.3.3 非ASCII文字 5](#_Toc129856513)

[3 ソースファイル構造 7](#_Toc129856514)

[3.1 ライセンスあるいはコピーライトの情報（もしあるならば） 7](#_Toc129856515)

[3.2 パッケージ文 7](#_Toc129856516)

[3.3 インポート文 7](#_Toc129856517)

[3.3.1 ワイルドカードインポートは禁止 7](#_Toc129856518)

[3.3.2 改行禁止 7](#_Toc129856519)

[3.3.3 順序と空白 7](#_Toc129856520)

[3.3.4 クラスをstatic importしない 8](#_Toc129856521)

[3.4 クラス宣言 8](#_Toc129856522)

[3.4.1 1個だけのトップレベルクラスの宣言 8](#_Toc129856523)

[3.4.2 クラスメンバーの順序 8](#_Toc129856524)

[4 フォーマット 9](#_Toc129856525)

[4.1 中括弧 9](#_Toc129856526)

[4.1.1 使えるところでは中括弧は使う 9](#_Toc129856527)

[4.1.2 空でないブロック：K&Rスタイル 9](#_Toc129856528)

[4.1.3 空ブロックは簡潔に 10](#_Toc129856529)

[4.2 ブロックのインデントは空白2個である 10](#_Toc129856530)

[4.3 1行毎に1個の文 11](#_Toc129856531)

[4.4 1行の文字数制限 100文字 11](#_Toc129856532)

[4.5 行の折り返し 11](#_Toc129856533)

[4.5.1 どこで改行するか 12](#_Toc129856534)

[4.5.2 連続する行は少なくとも４文字インデントする 12](#_Toc129856535)

[4.6 空白 13](#_Toc129856536)

[4.6.1 垂直の空白 13](#_Toc129856537)

[4.6.2 水平の空白 13](#_Toc129856538)

[4.6.3 水平位置揃え：全く不要 14](#_Toc129856539)

[4.7 グループ化の括弧 推奨 14](#_Toc129856540)

[4.8 各構造物 15](#_Toc129856541)

[4.8.1 列挙型 15](#_Toc129856542)

[4.8.2 変数宣言 15](#_Toc129856543)

[4.8.3 配列 16](#_Toc129856544)

[4.8.4 switch文 16](#_Toc129856545)

[4.8.5 アノテーション 17](#_Toc129856546)

[4.8.6 コメント 18](#_Toc129856547)

[4.8.7 修飾子 18](#_Toc129856548)

[4.8.8 数値リテラル 19](#_Toc129856549)

[5 命名 20](#_Toc129856550)

[5.1 すべての識別子への共通ルール 20](#_Toc129856551)

[5.2 識別子の種類ごとのルール 20](#_Toc129856552)

[5.2.1 パッケージ名 20](#_Toc129856553)

[5.2.2 クラス名 20](#_Toc129856554)

[5.2.3 メソッド名 20](#_Toc129856555)

[5.2.4 定数名 21](#_Toc129856556)

[5.2.5 定数でないフィールド名 22](#_Toc129856557)

[5.2.6 パラメータ名 22](#_Toc129856558)

[5.2.7 ローカル変数名 22](#_Toc129856559)

[5.2.8 型変数名 22](#_Toc129856560)

[5.3 キャメルケースの定義 22](#_Toc129856561)

[6 プログラミングの実践 24](#_Toc129856562)

[6.1 @Override：常に使う 24](#_Toc129856563)

[6.2 キャッチされた例外：無視しない 24](#_Toc129856564)

[6.3 staticなメンバー：クラスを使って修飾する 25](#_Toc129856565)

[6.4 ファイナライザ：使わない 25](#_Toc129856566)

[7 Javadoc 26](#_Toc129856567)

[7.1 フォーマット 26](#_Toc129856568)

[7.1.1 一般的なフォーマット 26](#_Toc129856569)

[7.1.2 段落 26](#_Toc129856570)

[7.1.3 ブロックタグ 26](#_Toc129856571)

[7.2 要約の記述 26](#_Toc129856572)

[7.3 Javadocが使われる場所 27](#_Toc129856573)

[7.3.1 例外：自己叙述的なメソッド 27](#_Toc129856574)

[7.3.2 例外：オーバーライドするメソッド 27](#_Toc129856575)

[7.3.3 必須ではないJavadoc 27](#_Toc129856576)

# 初めに

このコーディングルールは、Google社が公開している**Google Java Style Guideの非公式和訳版**を、システム技研の実習に合わせて改変したものです。

原本を確認したい場合は以下のURLを参照してください。

* Google「Google Java Style Guide」, CC-By 3.0  
  https://google.github.io/styleguide/javaguide.html   
  (2023/03/16確認済み)
* 「Google Java Style Guide(非公式和訳)」, CC-By 3.0  
  https://kazurof.github.io/GoogleJavaStyle-ja/  
  (2023/03/16確認済み)

# 導入

この文書はJava プログラミング言語のソースコードのコーディング標準の 完全 な定義を提供する。

他のプログラミングスタイルガイドのように、問題の対象範囲は審美的なフォーマットの問題だけでなく他の種類の規約やコーディング標準も含まれる。しかしながらこの文書は私達が全世界的に従う 当然の規則 に優先的に注力しており（人間やマシンのいずれでも）明確に実施できない助言をすることを避けている。

## 用語についての注記

本文書において、特別に断りのない限り、

1. クラスという用語は、「通常の」クラス、列挙型 、インターフェース、アノテーション型(@interface)を包括的に意味する。
2. （クラスの）メンバという用語は、ネストしたクラス、フィールド、メソッド、コンストラクタ、つまり初期化子とコメントを除いた全てのトップレベルの内容を包括的に意味する。
3. コメントという用語は、常に実装のコメントを意味する。「ドキュメンテーションコメント」という言い方は使わない。代わりに共通の言葉である「Javadoc」を使う。

本文書では他の用語の注記は適宜行われる。

## ガイドについての注記

この文書内のサンプルコードは**参考**である。 つまり、例がこの文書に書かれていても、それはコードを記述するための 唯一の 方法ではない場合がある。例に出される補足的なフォーマットの仕方はルールとして強調されるべきではない。

# ソースファイルの基本事項

## ファイル名

ソースファイル名はそのファイルに(正確に1つ)入っているトップレベルクラスの大文字小文字を区別した名前に加えて .java という拡張子が付いていること。

## ファイルエンコーディング：UTF-8

ソースファイルは UTF-8 でエンコードされていること。

## 特殊文字

### 空白

改行以外では、 ASCII 水平スペース文字 (0x20) はソース内でどこに現れても良い唯一の空白文字である。以下のことを意味する。

1. Stringと文字リテラルでのこれ以外の空白文字はエスケープされること。
2. タブ文字をインデントの目的で **使ってはならない。**

### 特別なエスケープシーケンス

特別なエスケープシーケンスを持つ全ての文字(\b, \t, \n, \f, \r, \", \' と \\)については8進数表記(\012)やUnicodeエスケープ(\u000a)でなく、通常のエスケープシーケンスで表記する。

### 非ASCII文字

残りの非ASCII文字については実際のUnicode文字（例：∞）あるいは同等のUnicodeエスケープ(例： \u221e)が使われる。

この選択は ソースコードを読むことや理解することが簡単になるかどうかのみに依存するが、文字列リテラルやコメント以外でのUnicodeエスケープは強く推奨されない。

Tip: Unicodeエスケープしている場合や、Unicode文字が使われている時でも、説明のコメントがあるとわかりやすい。

例 説明

String unitAbbrev = "μs"; 最良。コメントなしでも完全で明確。

String unitAbbrev = "\u03bcs"; // "μs" 許容される。しかしこう書く理由はない。

String unitAbbrev = "\u03bcs"; // ギリシャ文字ミューと "s" 許容される。しかし奇妙で間違えやすい。

String unitAbbrev = "\u03bcs"; だめ。読者はこれが何なのか分からない。

return '\ufeff' + content; // バイトオーダーマーク 良い。表示されない文字にはエスケープを使い必要ならコメントする。

Tip: ただ何かのプログラムが非ASCII文字を正しく処理しないという危惧だけでコードを読みにくくしてはならない。もしそのような事が起こる場合は、そのプログラムが **壊れている** のであってそちらが **修正** されるべきである。

# ソースファイル構造

ソースファイルの内容は **以下の順序** であること。

1. ライセンスあるいはコピーライトの情報（もしあるならば）
2. package文
3. import文
4. ただ1個のトップレベルクラス。

ソースに書かれている内容それぞれの分離には ただ1行の空行 を使うこと。

## ライセンスあるいはコピーライトの情報（もしあるならば）

もしファイルにライセンスあるいはコピーライトの情報があるならばここに入る。

## パッケージ文

パッケージ文は **改行してはならない。** 文字数制限（4.4節 文字数制限は100文字 ）はパッケージ文には適用されない。

## インポート文

### ワイルドカードインポートは禁止

ワイルドカードインポート はstaticであってもなくても **使ってはならない。**

### 改行禁止

import文は **改行してはならない。** 文字数制限（4.4節 文字数制限は100文字 ）はimport文には適用されない。

### 順序と空白

インポート文は以下の順序で並べられる。

1. 全てのstaticインポートを単一のブロックでまとめる。
2. 全ての非staticインポートを単一のブロックでまとめる。

もしもstaticインポートと非staticインポートの両方があるなら、一つの空行が両者を分離する。import節間に、空行は存在しない。

各ブロック内で、importされた名前はASCII順序で並べられる。（ 注釈： このことはインポート文全体がASCII順になっていることと同じではない。なぜなら ‘.’ は’;’ より先にソートされるからである。）

### クラスをstatic importしない

staticなネストしたクラスをimportするのにstatic importを使わない。それらは通常のインポートで使われる。

## クラス宣言

### 1個だけのトップレベルクラスの宣言

各トップレベルクラスは1個のファイルに保存される。

### クラスメンバーの順序

クラスメンバーや初期化子の順序の選択はわかりやすさに多大な影響を与える。しかしながら唯一の解法は無い。クラスが異なれば内容は異なった順序で並べられる。

重要な事はそれぞれのクラスはそのメンバを 何らかの合理的な順序 で並べ、クラスのメンテナンスをする人が尋ねられた時に答えられるようなることである。例えば新しいメソッドはクラスの最後に入れることが慣例であるかのように追加されてはならない。それは「追加された日の順」という結果になるだけであって、合理的な順序ではない。

#### オーバーロードしているメソッド群を分離してはならない

クラスに複数のコンストラクタや同じ名前を持つメソッドがある場合は、別のコードを（privateなメンバであっても）入れることなく連続して並べる。

# フォーマット

用語についての注釈： 「ブロック様の構造物(block-like construct)」とは、クラス、メソッド、コンストラクタの本体を指す。すべての配列初期化子は任意に「ブロック様の構造物」とみなされて良い。4.8.3.1節 配列初期化子 を参照。

## 中括弧

### 使えるところでは中括弧は使う

中括弧は if else for do while 文において本体が空でも1行しかなくても使われる。

### 空でないブロック：K&Rスタイル

中括弧は空でないブロックや、ブロック様の構造物ではカーニハン・リッチースタイル(Egyptian Brackets )に従う。

* 開始中括弧の前に改行を入れない。
* 開始中括弧の後に改行を入れる。
* 終了中括弧の前に改行を入れる。
* 終了中括弧の後に改行を入れる。但し、終了中括弧が文やメソッドの本体を終える時のみである。例えば終了中括弧の後に else や、カンマが続く場合は改行をしない。

例：

return () -> {

while (condition()) {

method();

}

};

return new MyClass() {

@Override public void method() {

if (condition()) {

try {

something();

} catch (ProblemException e) {

recover();

}

} else if (otherCondition()) {

somethingElse();

} else {

lastThing();

}

}

};

列挙型でのいくつかの例外は、4.8.1節 列挙型 にて示される。

### 空ブロックは簡潔に

空ブロックやブロック様の構造物は(Section 4.1.2で示したように) K & R スタイルでもよい。

また、開始括弧直後で( {} ) の間に文字や改行無しで閉じてよい。 但し、 if/else あるいは try/catch/finally のような複数ブロックの文の場合を除く。

例：

// これは問題ない

void doNothing() {}

// これも同様に問題ない

void doNothing() {

}

// これは問題がある。複数ブロックの文では簡単な空ブロックをつかってはならない。

try {

doSomething();

} catch (Exception e) {}

## ブロックのインデントは空白2個である

新しいブロックあるいはブロック様の構造物が開始した時インデントは空白2個づつ増える。ブロックが終了したら、インデントは1個前のレベルに戻る。インデントレベルはブロックを通じてコードとコメントに適用される。4.1.2節の例を参照のこと。（ 空でないブロック：K&Rスタイル ）

## 1行毎に1個の文

各文の末尾は改行でなくてはならない。

## 1行の文字数制限 100文字

Java コードの1行の文字数制限は100文字である。文字とは任意のUnicodeコードポイントを意味する。 以下の例外を除き、この制限を超えた行は4.5節 行の折り返し で述べるように改行されなくてはならない。

Tip: 画面表示上の大小が違っていても、それぞれのUnicodeコードポイントを１個と数える。例えば全角文字を使う場合、この規約を厳密に守る箇所では制限に達する前に改行して良い。

例外：

1. 文字数制限に従うのが不可能の場合。（例えば、Javadoc内の長いURL、長いJSNIメソッド1参照）
2. パッケージ文とインポート文　（3.2 パッケージ文 と 3.3 インポート文 を参照のこと）
3. コメント内の、コンソールにコピー＆ペーストされるようなコマンド。

訳注：JavaScript Native Interface 。Google Web Tookit においてアプリからJavaScript コードを呼ぶことができる。

## 行の折り返し

用語の注記： ある意味では規約に合致している1行のコードを複数行に分けた場合、この行為を行の折り返しと呼ぶ。

どんな状況にも合う改行方法を正確に示すような統一的で決定的なやり方は存在しない。同じコードを改行する正しい方法は複数存在する。

注記： 改行の典型的な理由は1行の文字数制限を超えることを避けることであるが、文字数制限を実際に満たしているコードであっても作者の裁量で改行される場合がある。

Tip: メソッドやローカル変数の抽出をすれば、改行をせずに問題を解決できる場合がある。

### どこで改行するか

改行の第一原則は、 高い文法のレベル で改行することである。また、

1. 代入でない演算子で改行するときは、シンボルの前で改行する。（これはJavaScriptやC++のような他の言語のGoogleスタイルの慣習とは異なることに注意すること。）
   * このことは、以下の「演算子ライクな」シンボルにも適用される。
     1. ドット演算子( . )
     2. メソッド参照でのコロン２個( :: )
     3. 型演算子の & 記号( <T extends Foo & Bar> )
     4. catch節でのパイプ記号 ( catch (FooException | BarException e) )
2. 行が代入演算子で改行されるときは、通常シンボルの後ろで改行される。しかしどちらの方法でも問題はない。
3. このことは拡張 for (“foreach”) 文の「代入演算子のような」コロンにも適用される。
4. メソッドやコンストラクタ名に続く開始括弧（ ( ）は直後に続いて書かれる。
5. カンマ( , )はその前のトークンの直後に続いて書かれる。

行はラムダの矢印の隣で改行されることはない。但しラムダの本体が単一のカッコなしの式である場合は矢印の直後で改行して良い。

例：

MyLambda<String, Long, Object> lambda =

(String label, Long value, Object obj) -> {

...

};

Predicate<String> predicate = str ->

longExpressionInvolving(str);

注記： 改行の第一目的はコードを明確にすることである。行数が小さくなるようにコードを書く必要はない。

### 連続する行は少なくとも４文字インデントする

改行の際、連続する先頭行に続く各行は少なくとも空白4個分、元からインデントする。

複数の連続した行がある場合、インデントは4以上ならいくつでも良い。一般的に、2個の連続した行が同じインデントレベルであることは、文法的に相似の要素であることと同じである。

4.6.3節の 水平位置揃え：全く不要 は、あるトークンを前の行に揃えるためいくつかの空白を入れるという非推奨のやり方を防ぐものである。

## 空白

### 垂直の空白

単一の空行は、以下の場合で使われる。

1. クラスの連続するメンバか初期化子の間。フィールド、コンストラクタ、メソッド、ネストしたクラス、static初期化子、インスタンス初期化子。

* 例外 ：2個の連続するフィールド（その間にコードがないもの）間での空行は任意である。そのような空行はフィールドの 論理的なグループ分け をするのに必要である。
* 例外 ：enum 定数間の改行は 4.8.1. 節で議論される

1. ステートメントの間で、コードを論理的にグループ分けしたいため。
2. 必要な場合、クラスの最初のメンバあるいは初期化子の前と最終メンバあるいは初期化子の後。（推奨も禁止もしない）
3. 本文書の別の節で入れるよう求められた場所（3節の ソースファイル構造 や、3.3節の インポート文 など）

複数の連続した空行を入れて良いが、必須でも推奨でもない。

### 水平の空白

言語かあるいは他のスタイルルールの求めを超え、リテラル、コメント、Javadoc以外で単一のASCII空白は以下の場所 のみ において使って良い。

1. 予約語（ if, for , catch ）とその行での開始小括弧 ( ( )の間。
2. 予約語（ else, catch ）とその行での前に来る終了中括弧( } )との間。
3. 開始中括弧( { )の前すべて。ただし以下の2個の例外を除く

* @SomeAnnotation({a, b}) (空白は使わない。)
* String[][] x = {{"foo"}}; ( {{ の中に空白は不要。項目8を参照)

1. すべての二項あるいは三項演算子の両側。また、以下の様な演算子ライクなシンボルにも適用する。
   * 連続する型パラメータ間のアンパサンド。 <T extends Foo & Bar>
   * 複数の例外を処理するcatchブロックでのパイプ。 catch (FooException | BarException e)
   * 拡張for文(“foreach”) でのコロン。 ( : )
   * ラムダ式での矢印。@(String str) →; str.length()@
2. 以下は除外する。
   * Object::toString と書かれるような、メソッド参照でのコロン２個
   * object.toString と書かれるような、ドット演算子。
3. , : ; あるいはキャストの閉じ括弧 （ ) ） の後ろ。
4. 行末コメントを開始するスラッシュ２個（ // ）の両側。ここでは複数の空白が許されるが必須ではない。
5. 型と変数の宣言の間。 List<String> list
6. 任意で、配列初期化子の両括弧の中。
   * new int[] {5, 6} と new int[] { 5, 6 } は両方有効。

注意：このルールは行頭行末の空白について要求や禁止をするものと解釈してはならない。内側の空白のみに適用される。

### 水平位置揃え：全く不要

用語の注釈： 水平位置揃えとは前行のトークン(変数名、型名)の真下に次行のトークンが来るように、入れるスペース数を調整するやり方のことである。

このやり方は許されるが、Googleスタイルでは 決して要求されない。 すでにこうなっている箇所をそのまま 維持 することすら必要ない。

これは水平位置揃えをやっている例とやっていない例である。

private int x; // これは良い

private Color color; // これも良い

private int x; // 許容される。しかし今後の編集で

private Color color; // 揃えられなくなるかもしれない。

Tip: カラムの調整は可読性を上げるが将来のメンテナンスで問題になる。一行だけ直したいときを考えてほしい。この変更は以前のきれいな並びをおかしくするだろう。このようなことが 発生しうる。 このことは開発者(多分君)に近くの行を同様になおせと求める。そして修正範囲の拡大を引き起こす。一行の変更が長大な変更となる。最悪意味のない作業になり、良くても変更履歴を汚染し、レビューが遅くなり、マージの衝突がおこるようになる。

## グループ化の括弧 推奨

追加のグループ化の括弧は作者とレビュアーが括弧なしでもコードは誤解される余地がないと認めるか、コードが読みやすくなる時のみ省くことが出来る。すべての読者がJava演算子の優先度表を覚えていると仮定するのは合理的では ない。

## 各構造物

### 列挙型

列挙定数値後のカンマの後ろの改行は任意である。追加の空行（大抵はたった一つ）も許可される。

ありうる可能性は、以下のとおり。

private enum Answer {

YES {

@Override public String toString() {

return "yes";

}

},

NO,

MAYBE

}

定数値にメソッドもドキュメンテーションもない列挙型は必要に応じて配列の初期化と同様に整形してよい。(4.8.3.1 節 配列の初期化 を参照)

private enum Suit { CLUBS, HEARTS, SPADES, DIAMONDS }

列挙型は クラスである のでクラスに対する他のルールが全て適用される。

### 変数宣言

#### 宣言ごとに一個の変数

フィールドでもローカル変数でも変数宣言は一個だけの変数を宣言する。 int a, b; のような宣言は使わない。

例外： 複数の変数宣言をすることは、forループのヘッダーであれば実施して良い。

#### 必要なときに宣言する

ローカル変数はそれを含むブロックやブロック様の構造物の先頭で慣習的に宣言されては ならない。 代わりに、ローカル変数はそのスコープを最小化するために(合理的な範囲で)最初に使う場所の近くで宣言される。ローカル変数宣言は通常は初期化がされるか、あるいは宣言されてすぐに初期化される。

### 配列

#### 配列の初期化：ブロック様で良い

配列の初期化はあたかも「ブロック様構造物」のようにやって良い。例えば以下の例はすべて有効である。網羅的なリストでは無い。

new int[] { new int[] {

0, 1, 2, 3 0,

} 1,

2,

new int[] { 3,

0, 1, }

2, 3

} new int[]

{0, 1, 2, 3}

#### Cのような配列宣言は禁止

角括弧は 型 の一部を構成するが、変数はそうではない。 String[] args は良い。 String args[] はダメ。

### switch文

用語についての注釈 スイッチブロックの括弧の内側は一個以上の文グループである。それぞれの文グループは一個以上のスイッチラベル（ case FOO: でも default: であっても）とそれに続く一個以上の文であるか、あるいは最後の文グループの場合は0個以上の文である。

#### インデンテーション

他のブロックと同様に、スイッチブロックは空白2個でインデントされる。

スイッチラベルの後に改行が入り、まさにブロックが開始したかのようにインデントレベルは2上がる。次のスイッチラベルではブロックが終わったかのように以前のインデンテーションレベルに戻る。

#### フォールスルーにはコメントを入れる

スイッチブロック内では、各ステートメントグループは突然処理が終了する（ break か continue か return か例外のスロー）か、実行が次のステートメントグループに進むようなコメントが付けられるかのどちらかだけである。フォールスルーということを示すコメントも効果的である。 （典型的には // fall through ）この特別なコメントは、最後のステートメントグループには必要ない。

例えば：

switch (input) {

case 1:

case 2:

prepareOneOrTwo();

// fall through

case 3:

handleOneTwoOrThree();

break;

default:

handleLargeNumber(input);

}

注意： case 1 の後にコメントは必要ない。ステートメントグループの終わりのみに必要である。

#### default 節は必要

各スイッチ文はたとえコードがない場合でも default ステートメントグループが必須である。

例外：enum型でのswitch文は、その型の全てのあり得る値を明示的に含めているならば、@default@ ステートメントグループを省略して良い。これによりIDEや静的コード分析ツールが何らかの値が漏れていることについて警告を発することが出来るようになる。

### アノテーション

クラス、メソッド、コンストラクタに付けられるアノテーションは、ドキュメンテーションブロックの直後に配置される。そして、各アノテーションは１行に１個設定される。

これらの改行は行折り返し(4.5 節 行折り返し )に従わない。それ故、インデンテーションレベルも上がらない。例えば：

@Override

@Nullable

public String getNameIfPresent() { ... }

例外： 単一のパラメータ無しのアノテーションの場合はシグネチャー行の先頭に現れて良い。例えば：

@Override public int hashCode() { ... }

フィールドへのアノテーションもドキュメンテーションブロックの直後に現れる。しかしこの場合、複数のアノテーション(パラメータが付きの可能性もある)が同じ行に現れても良い。例えば：

@Partial @Mock DataLoader loader;

パラメータやローカル変数や型についてのアノテーションには特にルールはありません。

### コメント

この節では実装コメントについて扱う。Javadocについては7節 Javadoc にて個別に扱われる。

すべての改行の前には任意で空白が入り実装コメントが続く。そのようなコメントは行を非空白にします。

#### ブロックコメントスタイル

ブロックコメントは周りのコードと同じレベルにインデントされる。 /\* ... \*/ でも //... でも同じである。複数行 /\* ... \*/ コメントについては \* の位置を前の行の \* と同じに揃えなくてはならない。

/\*

\* これは // これも /\* こんなかたち

\* 良い // 良い \* であっても良い。 \*/

\*/

コメントはアスタリスクや他の文字で描かれた箱で囲わない。

Tip: 複数行コメントを書く際必要に応じ自動フォーマット機能で行折り返ししたい場合は /\* ... \*/ スタイルを使うと良い。多くのフォーマッタは // ... スタイルのコメントの改行を直さない。

### 修飾子

クラスやメンバの修飾子が出現する場合、Java言語仕様が推奨する順序で出現しなくてはならない。

public protected private abstract default static final transient volatile synchronized native strictfp

### 数値リテラル

long 型の数値リテラルは大文字の L を末尾に使う。小文字は使わない。数値 1 との混乱を避ける。例えば 3000000000l ではなく 3000000000L を使う。

# 命名

## すべての識別子への共通ルール

識別子はASCII文字、数字のみを使う。また後述するいくつかの場合ではアンダースコアも使う。それゆえ、有効な識別子名は正規表現 \w+ にマッチする。

(訳注:5.2.4の 定数名 の場合と、5.2.3の メソッド名 でのUnitのテストメソッド名の場合)

本スタイルでは、 name\_ 、 mName 、 s\_name や kName といったような特別な接尾辞・接頭辞は 使わない。

## 識別子の種類ごとのルール

### パッケージ名

パッケージ名はすべて小文字で連続する単語をそのまま繋げる。アンダースコアは使わない。例えば、 com.example.deepspace であって、 com.example.deepSpace や com.example.deep\_space は使わない。

### クラス名

クラス名は大文字キャメルケース(UpperCamelCase)で命名する。

クラス名は大抵名詞か名詞句である。例えば、 Character や、 ImmutableList である。インターフェース名も名詞か名詞句である。（例えば List である。）しかし、場合によっては形容詞や形容詞句になる。（例えば、 Readable である。）

アノテーション型に対する特定のルールや確立した規約はない。

テストクラスはテスト対象クラス名で始まり、 Test で終わるよう命名される。例えば HashTest や、 HashIntegrationTest である。

### メソッド名

メソッド名は、小文字キャメルケース(lowerCamelCase)で命名される。

メソッド名は大抵動詞か動詞句である。例えば、 sendMessage や stop である。

アンダースコアはJUnitのメソッド名において、名前の論理的コンポーネント名（lowerCamelCaseで書かれる）を分離するために使ってよい。典型的なパターンは test<MethodUnderTest>\_<state> で、例えば testPop\_emptyStack である。テストメソッドを命名する正しい唯一の方法はない。

### 定数名

定数は コンスタントケース(CONSTANT\_CASE)で命名する。すべて大文字で、各単語を１個のアンダースコアで区切る。しかし定数とは正確には何だろうか。

定数とは、その内容が不変であってそのメソッドは検知可能な副作用を持たないような static final なフィールドである。

例えば、プリミティブ型、String、不変な型、不変な型の不変なコレクションである。

もしインスタンスの観測可能な状態が変化できるなら、それは定数ではない。

単に絶対にオブジェクトを変更しないことを意図するだけでは大抵は不十分である。

例：

// 定数である

static final int NUMBER = 5;

static final ImmutableList<String> NAMES = ImmutableList.of("Ed", "Ann");

static final ImmutableMap<String, Integer> AGES = ImmutableMap.of("Ed", 35, "Ann", 32);

static final Joiner COMMA\_JOINER = Joiner.on(','); // Joiner は不変であるので。

static final SomeMutableType[] EMPTY\_ARRAY = {};

enum SomeEnum { ENUM\_CONSTANT }

// 定数でない

static String nonFinal = "non-final";

final String nonStatic = "non-static";

static final Set<String> mutableCollection = new HashSet<String>();

static final ImmutableSet<SomeMutableType> mutableElements = ImmutableSet.of(mutable);

static final ImmutableMap<String, SomeMutableType> mutableValues =

ImmutableMap.of("Ed", mutableInstance, "Ann", mutableInstance2);

static final Logger logger = Logger.getLogger(MyClass.getName());

static final String[] nonEmptyArray = {"these", "can", "change"};

これらの名前は大抵名詞か名詞句である。

### 定数でないフィールド名

定数でないフィールド名（staticであってもなくても）は小文字キャメルケース(lowerCamelCase)で命名される。

これらの名前は大抵名詞か名詞句である。例えば computedValues や index である。

### パラメータ名

パラメータ名は小文字キャメルケース(lowerCamelCase)で命名される。

一文字のパラメータ名はpublic なメソッドでは避けるべきである。

### ローカル変数名

ローカル変数名は小文字キャメルケース(lowerCamelCase)で命名される。

final で不変であってもローカル変数は定数とは見なされないので定数として命名されるのは避けるべきである。

### 型変数名

型変数名は以下の２つのやり方のうちいずれかで命名される。

* 一つの大文字アルファベット。それに１個の数字が続いて良い。例： E, T, X, T2
* クラスの命名（5.2.2節 クラス名 ）の後ろに、大文字Tを付加する。例： RequestT, FooBarT

## キャメルケースの定義

時として、"IPv6"や、"iOS"のような頭字語や見慣れない形があるように、英語のフレーズをキャメルケースに変換する合理的な方法は複数ある。正確さを維持するため、Google Styleでは以下のような（ほぼ）決定的な手順を定義する。

名前の通常の形から始めて、

1. 言葉を素のASCIIに変換し、アポストロフィを除去する。例えば、"Müller’s algorithm" は “Muellers algorithm”に変換される。
2. これをスペースや残っている句読点（大抵はハイフン）で分離し、単語に分割する。
   * 推奨：もしもある単語が通常の利用において既に慣習的にキャメルケースの形をとっているならばそれも分解する。（例：「AdWords」 を 「ad words」にする。）「iOS」のような単語そのものは真にキャメルケースになっておらず、どんな規約にも当てはまらないのでこの推奨も適用されないないことに注意すること。
3. （頭字語を含めて）すべてを小文字にする。そして以下の語の最初の文字を大文字にする。
   * それぞれの単語。この場合大文字キャメルケースとなる。
   * 最初の単語を除いたそれぞれの単語。この場合小文字キャメルケースになる。
4. 最後に、すべての単語を１個の識別子として連結する。

元々の単語の大文字小文字はほとんど全て無視される。例えば、

表 5-1

|  |  |  |
| --- | --- | --- |
| もともとの形 | 正しい変換例 | 誤った変換例 |
| “XML HTTP request” | XmlHttpRequest | XMLHTTPRequest |
| “new customer ID” | newCustomerId | newCustomerID |
| “inner stopwatch” | innerStopwatch | innerStopWatch |
| “supports IPv6 on iOS?” | supportsIpv6OnIos | supportsIPv6OnIOS |
| “YouTube importer” | YouTubeImporter  YoutubeImporter (非推奨) |  |

注釈： いくつかの単語は英語として曖昧にハイフン付けされている。例えば、"nonempty" と “non-empty” はどちらも正しい。それゆえ同様に checkNonempty と checkNonEmpty というメソッド名はどちらも正しい。

# プログラミングの実践

## @Override：常に使う

適用可能ならば、メソッドには @Overrideアノテーションを付与する。これは親クラスのメソッドをオーバーライドするクラスのメソッドや、インターフェースのメソッドを実装するクラスのメソッドや、親インターフェースのメソッドを再定義する子インターフェースのメソッドにもあてはまる。

例外： @Override は親メソッドが@Deprecated の場合書かなくて良い。

## キャッチされた例外：無視しない

以下の例を除き、キャッチした例外についてなにも対応しないことが正しいことはめったにない。（典型的な対応はログを取るか、ありえない場合ならば AssertionError として再スローすることである。）

キャッチ節でなにもしないことが本当に適切であるならば、それを正当化する理由をコメントで説明する。

try {

int i = Integer.parseInt(response);

return handleNumericResponse(i);

} catch (NumberFormatException ok) {

// 数字ではない。正常であるので続行する。

}

return handleTextResponse(response);

例外： テストにおいて、キャッチされた例外が、 expected と命名されているあるいは始まる名前である場合は、コメントなしで無視してよい。以下の例はテスト対象のコードが期待した型の例外をスローすることを確認するためのよくあるイディオムであり、コメントは不要である。

try {

emptyStack.pop();

fail();

} catch (NoSuchElementException expected) {

}

## staticなメンバ：クラスを使って修飾する

staticなメンバーを修飾する必要がある場合はクラス名を使う。そのクラスの変数や式経由で使ってはならない。

Foo aFoo = ...;

Foo.aStaticMethod(); // 良い

aFoo.aStaticMethod(); // 悪い

somethingThatYieldsAFoo().aStaticMethod(); // とても悪い

## ファイナライザ：使わない

Object クラスの finalize() メソッドを使うケースは非常に稀である。

Tip: これをやってはならない。絶対的に必要ならば、まず Effective JavaのItem7 「ファイナライザを避ける」を必ず熟読しなくてはならない。そしてこれをやってはならない。

# Javadoc

## フォーマット

### 一般的なフォーマット

Javadocブロックの基本的なフォーマットはこの例で表される。

/\*\*

\* 複数行のJavadocテキストはここに書かれる。

\* 普通に改行される。

\*/

public int method(String p1) { ... }

一行の例はこうである。

/\*\* 特に短いJavadoc \*/

基本的な形は常に適用される。 一行の形はJavadocブロック全体（コメントマーカを含む）が１行で書ける場合のみ置き換えることができる。これは@returnタグのようなブロックタグが存在しない場合のみ適用可能であることに注意すること。

### 段落

一つの空行つまり先頭のアスタリスク(\*)のみの行が段落と段落の間に挿入される。もしブロックタググループがあるならばその前にも挿入される。最初以外のすべての段落には、最初の単語の前に空白無しで <p> が入れられる。

### ブロックタグ

全ての標準の&ブロックタグで& で、実際に利用されるものは @param, @return, @throws, @deprecatedの順で現れる。これらの４つには記述が必ず存在しなくてはならない。ブロックタグが１行コメントに収まらない場合、２行目以降は @ の位置からスペース４個以上インデントされる。

訳注：本家のJavadocの文書 でいうところの 「Javadocタグ」のこと。参照 Javadoc タグ

## 要約の記述

全てのJavadocブロックは簡単な 要約の記述 から始まる。この記述はとても重要である。なぜならクラスやメソッドの索引のような特別な場所に現れる唯一のテキストだからである。

この記述は小さな断片の形である。つまり名詞句か動詞句であって、完全な文であってはならない。「このクラス {@code Foo} は…」とか「このメソッドは…を返す。」で始まってはならないし、「結果を保存しなさい。」という命令形であってもならない。他方においてこの断片はあたかも文であるかのように大文字に変えられたり、句読点が付けられる。

Tip: /\*\* @return 顧客ID \*/ といった簡単なJavadocを書くことはよくある間違いである。これは間違いで正しくは以下のように直されるべきである。 /\*\* 顧客IDを返す。 \*/

## Javadocが使われる場所

少なくとも、Javadocは public なクラスとそのクラスの public 、 protected なメンバに書かれる。但し以下の例外がある。

追加のJavadocコメントがある場合がある。7.3.4節 不要なJavadoc にて解説される。

### 例外：自己叙述的なメソッド

Javadocは getFoo のような簡単で明確なメソッドの場合は必須ではない。つまり、「fooを返す」以外の意味ある情報が本当に無い場合である。

Tip: 重要： 典型的な読者が知りたがるような関連情報を省略することを正当化するためにこの例外を引用するのは適切ではない。例えば、「getCanonicalName」というメソッドにおいて典型的な読者が「canonical name」という語の意味を知らないかもしれない場合、（単に /\*\* canonical name を返す。 \*/ と書くだけであるという理由で）省略してはならない。

### 例外：オーバーライドするメソッド

親クラスのメソッドをオーバーライドするメソッドについてJavadocは必須ではない。

### 必須ではないJavadoc

訳注：節番号が飛んでいるが、原文に従う。

他のクラスとメンバにもJavadocは必要に応じて書かれる。

クラスやメンバの全体の目的を記述するのに実装コメントが使われているならば、そのコメントは代わりにJavadocで書かれる。（ /\*\* ）

必須ではないJavadocは7.1.2, 7.1.3, 7.2 節のフォーマット規則に従うことがもちろん推奨であるが、必ずしも必要ではない。