Google Java Style (非公式和訳)

この文書について

本文書はGoogleのJavaコーディング規約である Google Java Style の非公式和訳です。2014/03/21 Revision: r130 の版を使っています。技術的に正確である事を意図して訳してありますが、どこかで間違えているかもしれません。本家は随時更新される様子ですが、こちらが追いつかなくなるかもしれません。生暖かく見守ってやってください。誤訳、誤植の指摘を歓迎いたします。

1 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Googleのコーディング標準の 完全 な定義を提供する。下記のルールに従うJavaソースファイルのみが、Googleスタイルであるとみなされる。

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（人間でもマシンでも）明確に実施できない助言をすることを避けている。

1.1 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. 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。代わりに共通的に使われている、「Javadoc」という言葉を使う。

本文書では他の用語の注記は必要に応じて現れる。

1.2 ガイドについての注記

この文書内のサンプルコードは規約に 従っていない。つまり、サンプルコードはGoogleスタイル文書に書かれているが、これが絶対に正しいやり方である保証はない。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。

2 ソースファイルの基本事項

2.1 ファイル名

ソースファイル名はそれが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と加えて .java という拡張子が付いていること。

2.2 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。

2.3 特殊文字

2.3.1 空白

行区切り文字以外では、ASCII 水平スペース文字 (0×20) はソース内でどこに現れても良い唯一の空白文字です。つまり、

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 使ってはいけない。

ことを意味します。

2.3.2 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。

2.3.3 非ASCII文字

残りの非ASCII文字については ソースコードを読むことや理解することが簡単になる かどうかのみを基準にして実際のUnicode文字（例：∞）あるいは同等のUnicodeエスケープ(例： ¥u221e)を使うかの判断を行う。

TIP: Unicodeエスケープの場合時たま実際に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です。

例	説明
`String unitAbbrev = "μs";`	最高。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。読者や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: 単に何かのプログラムが非ASCII文字を正しく処理しないからといってコードを読みにくくしないで下さい。もしそのような事が起こる場合はそのプログラムがおかしいのであってそちらが修正されるべきです。

3 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。

それぞれの分離には ただ1個の空行 を使うこと。

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）

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。

3.2 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80あるいは100文字）はパッケージ文には適用されない。

3.3 インポート文

3.3.1 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。

3.3.2 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80あるいは100文字）はimport文には適用されない。

3.3.3 順序と空白

インポート文は以下のグループに以下の順序で分けられる。グループは1行の空白で分離される。

1. すべてのstatic importを単一のグループにまとめる。
2. com.google のインポート。（ソースファイルが com.google パッケージに属する場合のみ）
3. サードパーティのインポート。トップレベルパッケージ毎に1個のグループとしASCII文字の順序で

    - 例: android, com, junit, org, sun
4. java パッケージのインポート
5. javax パッケージのインポート

空行が入らないグループ毎にインポートした名前はASCII順に並べなければならない。(注釈: セミコロンがあるとソート結果は変わってしまう。この意味で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。)

3.4 クラス宣言

3.4.1 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。

3.4.2 クラスメンバーの順序

クラスメンバーの順序はわかりやすさに多大な影響を与えるが唯一の解法は無い。クラス毎にメンバーの並びが異なっていて良い。

重要な事はそれぞれのクラスのメンバーは誰かに訊かれた時に答えられるような 何らかの合理的な順序 で並べることである。例えば新しいメソッドはクラスの最後になんとなくで追加されてはならない。それは追加された日の順になっているだけであって、論理的ではない。

3.4.2.1 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連続して並べる。間に別のメンバーを入れてはならない。

4 フォーマット

用語についての注釈: “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。すべての配列は必要に応じて"block-like construct"とみなされて良い。4.8.3.1節 配列初期化子を参照。

4.1 中括弧

4.1.1 使えるところでは中括弧は使う

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

4.1.2 空でないブロックではK&Rスタイルに従う

中括弧は空でないブロックや、"block-like construct"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。
- もし終了中括弧が文やメソッドの本体を終えるならばその中括弧の後に改行を入れる。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 new MyClass() {
  @Override public void method() {
    if (condition()) {
      try {
```

```
        something();
      } catch (ProblemException e) {
        recover();
      }
    }
  }
};
```

列挙型でのいくつかの例外はセクション4.8.1 列挙型にて示される。

4.1.3 空ブロックは簡潔に

空ブロックや空のブロックのような構造物は開始括弧直後に文字や改行無しで閉じてよい。({}) 但し、 if/else-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:

```
void doNothing() {}
```

4.2 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個まえのレベルに戻る。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。(4.1.2 空でないブロックではK&Rスタイルに従う。)

4.3 1行毎に1個の文

各文は、末尾に改行が来なくてはならない。

4.4 1行の文字数制限 80文字か100文字

プロジェクトごとに1行の文字数制限を80文字か100文字いずれかで自由に決定して良い。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。

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。

4.5 行の折り返し

用語の注記: 別の意味で正当に単一行を占めているコードを複数行に分けるとき、通常は文字数制限を超えないように分ける。この活動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ありません。同じコード片を改行する正しい方法は複数あるものです。

> TIP: メソッドやローカル変数の抽出は改行をせずに問題を解決する場合があります。

4.5.1 どこで改行するか

改行の第一原則は、 高い文法のレベル で改行することです。つまり、

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します。(これはJavaScriptやC++のような他の言語でのGoogleスタイルで使われている週間と同じであることに注意して下さい)
    - このことはドット演算子(.)や、型演算子の＆記号(<T extends Foo & Bar>)や、catch節でのパイプ記号 (catch (FooException | BarException e))といった演算子のようなシンボ

ルにも適用されます。
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します。しかしどちらでも受け入れられます。
    - 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ます。
3. メソッドやコンストラクタ名に続く開始括弧(()はそれに続いて書かれます。改行されません。
4. カンマ(,)はその前のトークンに続いて書かれます。カンマの直前で改行されません。

4.5.2 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、連続する先頭行のに続く各行は少なくとも空白4個分元からインデントされます。

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です。一般的に、2個の連続した行がもし文法的に並行した要素で始まるならばそのときのみ同じインデントレベルであるべきです。

“4.6.3節の水平位置揃えは全く不要”は、あるトークンを前の行に揃えるため必要な分の空白を入れるというまずいやり方を防止します。

4.6 空白

4.6.1 垂直の空白

単一の空行が生成される状況は以下のとおり。

1. クラスの連続するメンバ(あるいは初期化子)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。
2. メソッド本体内で、文を論理的にグループ分けしたい場合。
3. 必要な場合、クラスの最初のメンバーの前と最終メンバーの後。(推奨も拒否もしない)
4. 本文書の別の節で入れるよう求められた場所(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。

4.6.2 水平の空白

言語かあるいは他のスタイルルールの要求であるかによらず、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)
4. すべてのバイナリ、tenary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。(:)
5. , : ; あるいはキャストの閉じ括弧 ( )) の後ろ。
6. 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。
7. 型と変数の宣言の間。`List<String> list`
8. 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も禁止もしない。内側の空白のみについて当てはまる。

4.6.3 水平位置揃えは全く不要

用語の注釈: 水平位置揃えは前行のトークン(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す。

やって良いですがGoogleスタイルでは 決して要求されません。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求めれられない

これはやっている例とやっていない例です。

```
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る。良くても変更履歴を汚くする。レビューが遅くなり、マージの衝突がおこるようになる。。

4.7 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、コードを読みやすく書いた時のみなくすことが出来ます。すべての読者がJava演算子の優先度表を記憶していると仮定するのは合理的でありません。

4.8 各構造物

4.8.1 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。

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任意に配列の初期化と同じやり方で整形してよい。(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適用される他のルールが適用される。

4.8.2 変数宣言

4.8.2.1 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れない。

4.8.2.2 必要なときに宣言して速やかに初期化する

ローカル変数はそれを含むブロックやブロック構造物の先頭でなんとなく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最初に使う場所(理由がある)の近くで宣言される。ローカル変数宣言はたいていは初期化子がある。あるいは宣言直後に初期化される。

4.8.3 配列

4.8.3.1 配列の初期化はブロックのようにやって良い。

配列の初期化はあたかも「block-like construct」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}
```

4.8.3.2 Cのような宣言は禁止

型名と角括弧で型の表現に使えるが、変数で使ってはならない。`String[] args` は良い。`String args[]` はダメ。

4.8.4 スイッチ文

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す。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)とそれに続く続く一個以上の文です。

4.8.4.1 インデンテーション

他のブロックがそうであるように、スイッチブロックのインデントは2です

スイッチラベルの後改行が入り、あたかもブロックが開始したかのようにインデントレベルは2上がります。次のスイッチラベルはあたかもブロックの終わったように前のインデンテーションレベルに戻ります。

4.8.4.2 フォールスルー 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止まる(`break` か `continue` か `return` か例外スローか)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みです。フォールスルーということを示すコメントも効果的です。`// fall through` など。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ありません。

例えば:

```
switch (input) {
 case 1:
 case 2:
   prepareOneOrTwo();
   // fall through
 case 3:
   handleOneTwoOrThree();
   break;
 default:
   handleLargeNumber(input);
}
```

4.8.4.3 default 節は必要

各スイッチ文はたとえコードがなかったとしても `default` ステートメントグループが必要です。

4.8.5 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ます。そして、各アノテーションは1行に1個設定されます。これらの改行は行折り返し(4.5

節　行折り返し) に従いません。それ故、インデンテーションレベルも上がりません。例えば:

```
@Override
@Nullable
public String getNameIfPresent() { ... }
```

例外: パラメータ無しのアノテーションが1個だけの場合はシグネチャー行の先頭に来ても良いです。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です。しかしこの場合、複数のアノテーション(@parameterized など)が同じ行に現れても良いです。例えば:

```
@Partial @Mock DataLoader loader;
```

パラメータや、ローカル変数へのアノテーションについては特にルールはありません。

4.8.6 コメント

4.8.6.1 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 * の位置を先頭行の * 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れることは無い。

> 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。

4.8.7 修飾子

クラスやメンバの修飾子は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 static final transient volatile synchronized native strictfp
```

4.8.8 数値リテラル

長い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 3000000l ではなく 300000L を使う。

5 命名

5.1 すべての識別子への共通ルール

識別子はASCII文字のみを使い、数字と大文字小文字とアンダースコアである。それゆえ、有効な識別子名は正規表現 ¥w+ にマッチする。

Google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・接頭辞は使われない。

5.2 識別子の種類ごとのルール

5.2.1 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。

5.2.2 クラス名

クラス名は大文字キャメルケースで命名する。

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

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ありません。

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ます。例えば HashTest や、HashIntegrationTest です。

5.2.3 メソッド名

メソッド名は、小文字キャメルケースで命名されます。

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

アンダースコアはJUnitのメソッド名で、論理的コンポーネント名を分離する場合で使ってよいです。典型的なパターンは test<MethodUnderTest>_<state> で、例えば testPop_emptyStack です。テストメソッドを命名する正しい唯一の方法はありません。

5.2.4 定数名

定数は コンスタントケースで命名します。すべて大文字で、各単語をアンダースコアで区切ります。しかし定数とは一体何でしょう？

すべての定数は static final なフィールドです。しかし、すべての static final なフィールドが定数であるとは限りません。
コンスタントケースを選ぶ前に、そのフィールドが定数と感じるか考えてみましょう。例えば、そのインスタンスの可視な状態が変更できるならば、殆どの場合定数ではありません。決して変更されないオブジェクトの振りをするだけでは大抵不十分です。例えば、

```
// 定数である
static final int NUMBER = 5;
static final ImmutableList<String> NAMES = ImmutableList.of("Ed", "Ann");
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 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す。

5.2.5 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で命名されます。

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

5.2.6 パラメータ名

パラメータ名は小文字のキャメルケースです。

一文字のパラメータ名は避けるべきです。

5.2.7 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で命名されます。他の種類の命名よりも自由に短縮されます。

しかしながら一文字の名前は一時的なループ変数を除いて避けるべきです。

`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そのようにスタイルされるのは避けるべきです。

5.2.8 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ます。

- 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 参照)の後ろに、大文字Tを付加する。例: `RequestT`, `FooBarT`

5.3 キャメルケースの定義

"IPv6"や、"iOS"のような頭字語や見慣れない単語があるように、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いくつかあります。正確さを維持するため、Google Styleでは以下のように(ほぼ)決定的な方法を定義します。

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. これを単語に分割する。つまり、スペースや残っている句読点、ハイフンで分離する。
    - 推奨:もしもある単語が一般的にキャメルケースの形になっていたら、これを分解する。(例:「AdWords」を「ad words」にする。)「iOS」のような単語は本当はキャメルケースになっていない
      規約に当てはまらないのでこの推奨は適用しないことに注意する。
3. (頭字語を含めて)すべてを小文字にする。そして最初の文字を大文字にする。
    - 各単語とは大文字キャメルケースとなるか、最初の単語を除いて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。

元々の大文字小文字はほぼ無視される。例えば、

元々の形	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 [1]	

[1] やってよいが推奨されない。

> 注釈: いくつかの単語は英語では曖昧にハイフン付けされています。例えば、"nonempty" と "non-empty" はどちらも正しいです。ですので、checkNonempty や checkNonEmpty というメソッド名はどちらも正しいです。

6 プログラミングの実践

6.1 @Override を常に使う。

許される場所ならばメソッドは @Overrideアノテーションをつけられます。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当てはまります。

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

6.2 キャッチした例外を無視しない

以下の例外を除き、キャッチした例外に対してなにも対応しないのが正しいことはめったにありません。大抵の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す。

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します。

```
try {
  int i = Integer.parseInt(response);
  return handleNumericResponse(i);
} catch (NumberFormatException ok) {
  // 数字ではない。このような場合もあるので別の処理を続ける。
}
return handleTextResponse(response);
```

例外: テストにおいて例外がスローされることを期待する場合はキャッチした例外はコメントなしで無視されます。以下の例はテスト対象のメソッ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、コメントは不要です。

```
try {
  emptyStack.pop();
  fail();
} catch (NoSuchElementException expected) {
}
```

6.3 staticなメンバーはクラスを使って修飾する。

staticなメンバーへを使う場合はクラス名経由で使います。そのクラスの変数や式経由で使ってはいけません。

```
Foo aFoo = ...;
Foo.aStaticMethod(); // 良い
aFoo.aStaticMethod(); // 悪い
somethingThatYieldsAFoo().aStaticMethod(); // とても悪い
```

6.4 ファイナライザは使わない。

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

> TIP: これをやってはいけません。貴方は最初に 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りません。そしてこれをやってはいけません。

7 Javadoc

7.1 フォーマット

7.1.1 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ます。

```
/**
 * 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) { ... }
```

一行の例はこれです。

```
/** 特に短いJavadoc */
```

基本的な形は常に適用されます。Javadocタグがない場合やコメントマーカを含めたJavadocブロック全体において1行で書くほうが体裁が良い場合は一行の形で代用できます。

7.1.2 段落

一つの空行つまり行頭のアスタリスク(*)のみの行が段落と段落の間に挿入される。もしjavadoc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。

7.1.3 javadoc タグ [2]

標準のjavadoc タグで使われるものは @param, @return, @throws, @deprecatedの順で現れる。これらの4つには記述が必ずつかなくてはならない。javadoc 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。

[2] 訳注:javadoc での @param といった節のこと。原文では"at-clauses"であるが、GoogleJavaStyleでしか使っていない用語なのでjavadocの元文書に基づきjavadoc タグと翻訳した。参照 javadoc タグ

7.2 要約の記述

クラスとメンバーのJavadocは簡単な 要約の記述 から始まります。この記述はとても重要です。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す。

この記述は小さな断片の形です。つまり名詞句か動詞句であって文であってはなりません。「このクラス {@code Foo} は、、、」とか「このメソッドはナニナニを返す。」で始まってはいけませんし、「結果を保存しなさい。」という命令形でもいけません。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、句読点が付けられます。

> 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す。
> これは間違いで正しくは以下のように直されるべきです。 /** 顧客IDを返す。 */

7.3 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` なメンバーに書かれます。但し以下の例外が有ります。

クラスとメンバー以外にもJavadocは必要に応じて書かれます。クラス、メソッド、フィールドの全体の振る舞いや目的を記述するのに実装コメントが使われている場合は代わりにJavadocで書かれます。(より統一的で、ツールとの親和性も高いです。)

7.3.1 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ありません。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す。

> 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ありません。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 `/** Returns the canonical name. */` と書くだけであるという理由で)省略してはいけません。

7.3.2 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はJavadocは必須ではありません。