Javaコーディング規約

Future Enterprise Coding Standards

Table of Contents

本コーディング規約は、世の中のシステム開発プロジェクトのために無償で提供致します。  
ただし、掲載内容および利用に際して発生した問題、それに伴う損害については、フューチャーアーキテクト株式会社は一切の責務を負わないものとします。  
また、掲載している情報は予告なく変更することがございますので、あらかじめご了承下さい。

# はじめに

一般に利用・参照されているJavaコーディング規約やガイドラインを以下に示す。本規約の作成においても、下記規約類を参照・抜粋している。

|  |  |  |
| --- | --- | --- |
| 規約 | 著作者 | URL |
| Code Conventions for the Java Programming Language | Sun Microsystems | <http://www.oracle.com/technetwork/java/codeconvtoc-136057.html> |
| Writing Robust Java Code | Scott W. Ambler | <http://www.ambysoft.com/downloads/javaCodingStandards.pdf> |
| オブジェクト倶楽部版 Javaコーディング標準 | オブジェクト倶楽部 | <http://objectclub.jp/community/codingstandard/CodingStd.pdf> |
| 電通国際情報際サービス版 Javaコーディング規約2004 | 電通国際情報サービス | <http://objectclub.jp/community/codingstandard/JavaCodingStandard2004.pdf> |
| JJGuideline （Java - J2EE Conventions and Guidelines） | Stephan.J & JCS Team | <http://www.fedict.belgium.be/sites/default/files/downloads/Java_J2EE_conventions_and_guidelines_EN.pdf> |
| Google Java Style (非公式和訳) | Google | <https://kazurof.github.io/GoogleJavaStyle-ja/> |
| Acroquest Technology Javaコーディング規約 | Acroquest Technology | <https://www.acroquest.co.jp/webworkshop/javacordingrule/Acroquest_JavaCodingStandard_6_7.pdf> |

※ Sun Microsystemsの規約はJava草創期から一応の標準という位置づけだったが、オブジェクト指向、及び、その開発環境の普及・発展によって、設計やコーディングにおいて、直接的に有用な知識や豊富な指針を含むような優れた規約や、ツールなどによる機械的な準拠チェックと連携する規約が普及してきている。

# 規約の重要性

標準としての規約を定義し、遵守することの重要性を以下に示す。

* ソフトウェアメンテナンスにおける、可読性・保守性・拡張性の向上
* 問題を起こしやすい実装を未然に回避することによる、品質・生産性の向上
* 標準規約を通して得られる一般的な実装知識やノウハウ（＝学習効果）

## コーディングの心得

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ある。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る。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考えることもでき、コードの保守性を向上させる具体的な方法を示している。したがって、規約の一つ一つの意図を理解し、守ることが重要になる。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る。以下では、その心得について述べる。

【コーディングの心得5か条】  
1. 見やすさを重視せよ  
2. ネーミングはわかりやすく  
3. サンプルを鵜呑みにしない  
4. 同じコードを二度書かない  
5. 役割は一つに

### 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」。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る。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する必要がある。例えば、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誰にもあるはず。「3日前に書いたコードは他人のコードと同じ」ということもよく言われる。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る。コードを読んでもすぐに理解できないような実装は、再考（リファクタリング）の必要がある。

### 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）する必要がある。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ある。大変難易度の高い作業だが、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となる。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す。例えば、「C0001」というクラス名があるとする。これでは、何を表すクラスなのかすぐにはわからないだろう。また、「int p = 5000;」という記述があるとする。プログラマに聞くと、変数名pは価格(Price)の略だと言うのだが、それならば略さずに、「int price = 5000;」としたほうが分かりやすいはずである。「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、という考え方に基づく。

### 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。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ある。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る。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可能性がある。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る。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場合もある。その場合、コードの可読性を下げる原因となる。自分のコードは、自分で深く理解して記述すべきである。

### 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る。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にすべきである。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。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もある。

### 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る。つまり、コードの「試験性」が高まる。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る。つまり、コードの「保守性」があがることになる。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する。この場合、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ある。なぜなら、checkAndDo()メソッドのcheck()ロジックに誤りがあった場合、do()メソッドに書かれる内容まで把握する必要が生じるためである。分割してあれば、check()メソッドだけの変更で済む。このことはクラスの設計にもあてはまる。

# ネーミング規約

## 全般

* 大文字・小文字の違いで名前を区別しない。
* 良い例：
* private int carNumber;  
  private int trainNumber;
* 悪い例：
* private int num;  
  private int Num;

## パッケージ

* パッケージ名はすべて小文字にする
* パッケージ名は意味のある名前にする
* サブパッケージ名の重複は可能

## クラス

* クラス名は単語の先頭を大文字にする  
  良い例：
* public class Entry {
* 悪い例：
* public class entry {
* インターフェース名はクラス名に準ずる

## メソッド

* コンストラクタと同じ名前のメソッドはつくらない
* メソッド名は区切りのみ大文字にする  
  良い例：
* public String getName() {  
   //・・・  
  }
* 悪い例：
* public String getname() {  
   //・・・  
  }  
  public String GETNAME() {  
   //・・・  
  }
* 変換メソッド名は「"to"+オブジェクト名」にする  
  良い例：
* public String toString() {
* 悪い例：
* public String string() {
* ゲッターメソッド名は「"get"+属性名」にする  
  型がbooleanの場合は「"is"+属性名」にする
* セッターメソッド名は「"set"+属性名」にする
* boolean変数を返すメソッド名はtrue/falseの状態がわかるようにする
* 良い例：
* public boolean isAsleep(){  
  }  
  public boolean exists(){  
  }  
  public boolean hasExpired(){  
  }

## 引数

* メソッドのパラメータ名とインスタンス変数名を一緒にしない  
  ただし、アクセサメソッドやコンストラクタなど、統合開発環境の機能により自動生成するものに関しては可とする。  
  アンダースコア　\_　をつけての区別は原則禁止とする。
* 良い例：
* public double calc(double rate) {  
   return this.value \* rate;  
  }
* 悪い例：
* public double calc(double value) {  
   return this.value \* value;  
  }  
    
  public double calc(double \_value) {  
   return this.value \* \_value;  
  }

## 変数全般

* boolean変数はtrue/false の状態がわかるようにする
* 良い例：
* private boolean isOpen;
* 悪い例：
* private boolean flag;
* 定数は全てstatic finalとし、すべて大文字、区切りは"\_"
* 良い例：
* private static final String SYSTEM\_NAME = "販売管理システム";
* 変数名は小文字とし、単語の区切りのみ大文字にする
* 良い例：
* private String thisIsString;
* 変数名に固有名詞が含まれる場合、先頭をのぞき、単語の区切り以外に大文字を使用してもよい
* 良い例：
* private String thisIsIPAddress;

## ローカル変数

* スコープが狭い変数名は省略した名前でもよい  
  良い例：
* if (・・・){  
   String s = "・・・・";  
   //変数sを利用した処理 数行  
  }
* 悪い例：
* String s = "・・・・";  
  if (・・・){  
   //変数sを利用した処理  
  }  
  ・・・  
  if (・・・){  
   //変数sを利用した処理  
  }
* 変数sの利用範囲が広いので役割が明確になる編素名に変更する。
* for 文のループカウンタは、ネストごとに"i","j","k"・・・を使う

# コーディング規約

## 全般

* 原則としてオブジェクトの参照にはインターフェースを利用する  
  オブジェクトを参照する際は、そのオブジェクトの実装クラスを用いて宣言できるが、実装クラスに適切なインターフェースが存在している場合は、必ずインターフェースを用いて宣言すること。
* 良い例：
* List<Entry> list = new ArrayList<>();  
  Map<String, String> map = new HashMap<>();
* 悪い例：
* ArrayList<Entry> list = new ArrayList<>();  
  HashMap<String, String> map = new HashMap<>();
* 推奨されないAPI を使用しない  
  アノテーション@Deprecatedで指定されたメソッドは利用しないこと。
* 使われないコードは書かない
* 宣言は適切な権限で行うこと（public, protected, private）
* final を適切に利用する  
  継承されないクラス、オーバーライドされないメソッド、値の変わらない変数（つまり定数）等、変化のないもの/変化させたくないものについてはfinal で宣言する。
* 良い例：
* //継承されないクラス  
  public final class CalculateUtils {  
   //・・・  
  }  
    
  //値の変わらない変数（定数）  
  private static final String MESSAGE = "・・・";  
    
  //オーバーライドされないメソッド  
  public final int sum(/\*変化させたくない値\*/final int... values) {  
   int sumValue = 0;  
   for (/\*変化させたくない値\*/final int value : values) {  
   sumValue += value;  
   }  
   return sumValue;  
  }

## フォーマット

* インデントは空白文字4文字分のTabを使用する
* 長すぎる行は避ける
* { の後にステートメントを記述しない  
  良い例：
* if (s == null) {  
   return 0;  
  }
* 悪い例：
* if (s == null) {return 0;}
* 1行に2つ以上のステートメントを記述しない  
  悪い例：
* } catch (Exception e) {  
   log.error("Error", e);return null;  
  }
* カンマの後には空白文字を  
  良い例：
* process(x, y, z);
* 悪い例：
* process(x,y,z);
* 代入演算子（ = , += , -= , …）の前後には空白文字を挿入する  
  良い例：
* int a = x;  
  a += 10;
* 悪い例：
* int a=x;  
  a+= 10;
* for 文内のセミコロンの後には空白文字を挿入する  
  良い例：
* for (int i = 0; i < array.length; i++) {  
   //・・・  
  }
* 悪い例：
* for (int i = 0;i < array.length ;i++) {  
   //・・・  
  }
* ++ や -- とオペランドの間には空白文字を入れない  
  良い例：
* i++;
* 悪い例：
* i ++;
* ビット演算子（ | 、 & 、 ^ 、 << 、 >> ）の前後には空白文字を挿入する
* 論理演算子（ || 、&&）の前後には空白文字を挿入する
* 関係演算子（ < 、 > 、 >= 、 <=、==、 != ）の前後には空白文字を挿入する
* 算術演算子（ ＋ 、 － 、 ＊ 、 / 、 % ）の前後には空白文字を挿入する
* return文ではカッコを使わない  
  良い例：
* int answer = (a + b + c) \* d;  
  return answer;
* 悪い例：
* return ((a + b + c) \* d);
* ifなどの条件式でbooleanの変数を比較しない
* 良い例：
* if (hasStock)
* 悪い例：
* if (hasStock == true)
* 不等号の向きは左向き（ < 、 <= ）にする  
  良い例：
* if (from <= x && x <= to) {
* 悪い例：
* if (x >= from && x <= to) {

## コメント

* ファイルの先頭へのCopyrightの表記について  
  ソースのファイルヘッダにコピーライト標記は法的拘束力がないため、不要とする。  
  ただし、顧客からの要求があった場合を除く。
* Javadoc コメントには、少なくともauthorとversion(クラス)、paramとreturn とexception(メソッド)を記述する
  + 今後もバージョンアップのリリースが予定されているソースでは、上記に加えてsince（バージョン）を記述する
  + @Overrideのあるメソッドでは、上記に加えて{@Inherit}を記述する
* Javadoc　クラスヘッダコメントのフォーマットは以下の通り
* 良い例：
* /\*\*  
   \* Action（or Bean）クラス　メニュー名称  
   \*  
   \* @author 姓 名  
   \* @version バージョン YYYY/MM/DD 説明  
   \*/
* コメントは必要なものだけを簡潔に  
  悪い例：
* /\*\*  
   \* 文字列に変換  
   \*/  
  @Override  
  public String toString() {  
    
  /\*\*  
   \* コピー  
   \*  
   \* @return コピーしたインスタンス  
   \*/  
  public Entry copy() {
* 不要なコメントは記載しない
  + コードからすぐわかること・冗長なコメント
  + 名前の説明  
    コメントではなくわかりやすい名前を付ける。
  + 別システムで管理している内容  
    ソースコード管理システム、バグトラッキングシステムで管理している内容はソースコードにコメントで記載する必要はない。
    - コメントアウトされたコード  
      ソースコード管理システムで管理されている

## インポート

* java.langパッケージはインポートしない
* 悪い例：
* import java.lang.String;//必要のない記述
* 原則としてstaticインポートしない  
  JUnitの作成やフレームワークとしてstaticインポートが推奨されるような場合は利用してもよい
* 原則としてオンデマンドのインポート宣言(type-import-on-demand declaration)（アスタリスク\*によるインポート） は行わない
* 悪い例：
* import java.util.\*;

## コンストラクタ

* public 宣言していないクラスにはpublic権限のコンストラクタを作らない  
  良い例：
* class Entry {  
   //・・・  
   Entry(int id) {  
   //・・・  
   }
* 悪い例：
* class Entry {  
   //・・・  
   public Entry(int id) {  
   //・・・  
   }
* インスタンスメンバを持たない（staticメンバのみの）クラスは、private権限のコンストラクタを作成する

## メソッド

* オーバーライドさせたくないメソッドはfinalを利用する
* 戻り値が配列のメソッドで、戻る配列のサイズが0の場合、メソッドを使用するクライアントの余計なnullチェックのロジックを回避するため、nullではなく長さゼロの配列を戻すようにする。 良い例：
* public String[] toArray(String s) {  
   if (s == null || s.isEmpty()) {  
   return ArrayUtils.EMPTY\_STRING\_ARRAY;  
   }  
   return new String[] { s };  
  }  
    
  public List<String> toList(String s) {  
   if (s == null || s.isEmpty()) {  
   return Collections.emptyList();  
   }  
   return List.of(s);  
  }
* 悪い例：
* public String[] toArray(String s) {  
   if (s == null || s.isEmpty()) {  
   return null;  
   }  
   return new String[] { s };  
  }  
    
  public List<String> toList(String s) {  
   if (s == null || s.isEmpty()) {  
   return null;  
   }  
   return List.of(s);  
  }
* メソッドは1つの役割にする

## クラスメソッド

* クラスメソッドを利用するときは、クラス名を使って呼び出す  
  良い例：
* int comp = Integer.compare(x, y);
* 悪い例：
* Integer a = //  
  int comp = a.compare(x, y);

## 変数全般

* 1つのステートメントには1つの変数宣言  
  良い例：
* /\*\* 科目コード \*/  
  private String code;  
  /\*\* 科目名 \*/  
  private String name;  
  /\*\* 科目略名 \*/  
  private String shortName;
* 悪い例：
* private String code, name, shortName;
* リテラルは使用しない  
  リテラルとは、コード中に、表現が定数として直接現れており、記号やリストで表現することができないものを指す（数値、文字列両方含む　通称マジックナンバー）。コードの可読性・保守性の低下を防ぐために、リテラル定数（static final フィールド）を使用すること。  
  例外：-1,0,1 等をカウント値としてループ処理等で使用するような場合
* 良い例：
* private static final double ONE\_MILE\_METRE = 1609.344;  
    
  public double mileToMetre(double mi) {  
   return mi \* ONE\_MILE\_METRE;  
  }
* 悪い例：
* public double mileToMetre(double mi) {  
   return mi \* 1609.344;  
  }
  + リテラル定数の名前はその値の意味を正しく表現したものにする
  + 悪い例：
  + private static final int ZERO = 0;
* 配列宣言は「型名[]」にする
* 良い例：
* private int[] sampleArray = new int[10];
* 悪い例：
* private int sampleArray[] = new int[10];
* できるだけローカル変数を利用する  
  ローカル変数で事足りるものをインスタンス変数として利用するなど、必要のないインスタンス変数を定義すると、パフォーマンスや可読性の低下やの大きな要因となる上、マルチスレッドを意識した際に不整合がおきる可能性があるので、インスタンス変数は必要性を充分に考慮してから使用すること。
* 定数はfinalで宣言する
* ローカル変数とインスタンス変数を使いわける

## 定数

* public で宣言するクラス変数とインスタンス変数は、定数のみとし、 static final で定義する  
  final ではない static な定数は作成しない。
* 良い例：
* public static final String PROTOCOL\_HTTP = "http";
* 定数（ static フィールド）に、 static ではないメソッドから書き込まない
* 定数は、プリミティブ型もしくは、不変（Immutable）オブジェクトで参照する
  + 不変Listの生成にはList.of()を利用する
  + 良い例：
  + public static final List<Integer> VALUES = List.of(1, 2, 3, 4, 5);
  + 悪い例：
  + public static final List<Integer> VALUES = Arrays.asList(1, 2, 3, 4, 5);
  + 不変Setの生成にはSet.of()を利用する
  + 不変Mapの生成にはMap.of()を利用する
  + 良い例：
  + public static final Map<Integer, String> VALUES\_MAP = Map.of(1, "A", 2, "B", 3, "C");
  + 悪い例：
  + public static final Map<Integer, String> VALUES\_MAP = new HashMap<>() {  
     {  
     put(1, "A");  
     put(2, "B");  
     put(3, "C");  
     }  
    };
  + 不変な配列インスタンスは長さ0の配列以外は生成不可能なため、外部から参照される（public）定数では利用せず、List等への置き換えをすること
  + 良い例：
  + public static final List<Integer> VALUES = List.of(1, 2, 3, 4, 5);
  + 悪い例：
  + public static final int[] VALUES = { 1, 2, 3, 4, 5 };

## インスタンス変数

* インスタンス変数はprivateにする
* 良い例：
* public class Employee {  
   private long id;  
    
   //・・・  
   //getter/setter  
  }
* 悪い例：
* public class Employee {  
   public long id;  
    
   //・・・  
   //getter/setter  
  }

## クラス変数

* public static final 宣言した配列を利用しない  
  ※「定数」を参照
* クラス変数にはクラス名を使用してアクセスすること
* 良い例：
* BigDecimal b = BigDecimal.ZERO;
* 悪い例：
* BigDecimal a = //  
  BigDecimal b = a.ZERO;

## ローカル変数

* ローカル変数は利用する直前で宣言する  
  行間の程度にもよるが、ある程度まとめて宣言するのはOKとする。
* 良い例：
* for (int i = 0; i < lines.length; i++) {  
   String line = lines[i];  
   //lineの処理  
  }
* 悪い例：
* String line;  
  for (int i = 0; i < lines.length; i++) {  
   line = lines[i];  
   //lineの処理  
  }
* ローカル変数は安易に再利用しない  
  一度宣言したローカル変数を、複数の目的で安易に使いまわさないこと。ローカル変数は、役割ごとに新しいものを宣言して初期化することにより、コードの可読性・保守性の向上、及びコンパイラの最適化の促進をはかる。

## 引数

* メソッド引数への代入は行わない  
  原則としてfinalで宣言する。
* 良い例：
* public void add(final int value) {  
   //・・・  
  }

## 継承

* スーパークラスのインスタンス変数をサブクラスでオーバーライドしない  
  スーパークラスと同じ名前のフィールドをサブクラスで宣言しないこと。 同じ名前のフィールドを宣言すると、スーパークラスのフィールドはサブクラスで宣言されたフィールドによって隠ぺいされてしまうので、他の人の混乱を招くことを防ぐため重複する名前は付けないこと。
* 悪い例：
* public class Abs {  
   protected String name;  
  }  
    
  public class Sub extends Abs {  
   protected String name;//Abs#nameは隠ぺいされる  
  }
* スーパークラスのメソッドをオーバーライドするときは@Overrideアノテーションを指定する。
* 良い例：
* public class Abs {  
   protected void process() {  
    
   }  
  }  
    
  public class Sub extends Abs {  
   @Override  
   protected void process() {  
    
   }  
  }
* 悪い例：
* public class Abs {  
   protected void process() {  
    
   }  
  }  
    
  public class Sub extends Abs {  
   //@Overrideアノテーションの指定がない  
   protected void process() {  
    
   }  
  }
* スーパークラスでprivate 宣言されているメソッドと同じ名前のメソッドをサブクラスで定義しない  
  スーパークラスにあるprivate メソッドと同じ名前のメソッドをサブクラスで定義しないこと。private メソッドはオーバーライドされず全く別のメソッドとして扱われ、他の人の混乱を招き、バグにつながる恐れがある。

## インナークラス

* 原則としてインナークラスは利用しない  
  一つのjavaファイルに複数のクラスを記載するのはNGとする。また無名クラスを利用するのも原則としてNGとする。  
  Enumの定数固有メソッド実装(constant-specific method implementation)、Java8のラムダ式は内部的にインナークラスとされるがこれらは許可する。

## メンバー順序

* 以下の順で記述する
  1. staticフィールド
  2. staticイニシャライザー
  3. staticメソッド
  4. フィールド
  5. イニシャライザー
  6. コンストラクター
  7. メソッド
* 同一カテゴリー内では以下の可視性の順で記述する
  1. public
  2. protected
  3. パッケージprivate
  4. private

## インスタンス

* オブジェクト同士はequals()メソッドで比較する
* 良い例：
* String s1 = "text";  
  String s2 = "text";  
  if (s1.equals(s2)) {  
   //・・・  
  }
* 悪い例：
* String s1 = "text";  
  String s2 = "text";  
  if (s1 == s2) {  
   //・・・  
  }
* ただしEnumの場合は==演算子を利用して比較する
* equals()メソッドで比較する際、左辺のオブジェクトがnullにならないように制御すること。
* Class 名を利用した比較をおこなわない
* 良い例：
* (o instanceof Foo)
* 悪い例：
* ("hoge.Foo".equals(o.getClass().getName()))

## 制御構造

* 制御文（ if , else , while , for , do while ）の { } は省略しない
* 良い例：
* if (s == null) {  
   return;  
  }
* 悪い例：
* if (s == null)  
   return;
* ステートメントが無い {} ブロックを利用しない  
  悪い例：
* //{}内の記述が無い  
  if (s == null) {  
  }
* if / while の条件式で = は利用しない  
  良い例：
* boolean a =//  
  if (!a) {  
   //・・・  
  }
* 悪い例：
* boolean a =//  
  if (a = false) {//コーディングミス  
   //・・・  
  }  
    
    
  boolean a =//  
  boolean b =//  
  if (a = b) {//おそらくコーディングミス  
   //・・・  
  }
* for と while の使い分けを意識する
* for 文を利用した繰り返し処理中でループ変数の値を変更しない  
  悪い例：
* String[] array = { /\*・・・\*/ };  
  for (int i = 0; i < array.length; i++) {  
   //・・・  
   i += 2;//NG  
  }  
    
  for (String s : array) {  
   //・・・  
   s = "string";//NG  
  }
* for 文のカウンタは特別な事情がない限り、0から始める
* 配列やリストなどの全要素に対するループ処理は拡張for文を使用する。  
  良い例：
* for (int value : array) {  
   //・・・  
  }  
    
  for (String value : list) {  
   //・・・  
  }
* 配列をコピーするときはArrays.copyOf()メソッドを利用する
* 良い例：
* int[] newArray = Arrays.copyOf(array, array.length);
* 悪い例：
* int[] newArray = new int[array.length];  
  System.arraycopy(array, 0, newArray, 0, array.length);
* 繰り返し処理中のオブジェクトの生成は最小限にする
* if 文とelse 文の繰り返しやswitch 文の利用はなるべく避け、オブジェクト指向の手法を利用する  
  良い例：
* CodingKind codingKind = toCodingKind(kind);  
  d = codingKind.encode(s);  
    
  //---  
    
  CodingKind codingKind = toCodingKind(kind);  
  s = codingKind.decode(d);
* 悪い例：
* switch (kind) {  
  case 1:  
   d = encode1(s);  
   break;  
  case 2:  
   d = encode2(s);  
   break;  
  default:  
   break;  
  }  
    
  //---  
    
  switch (kind) {  
  case 1:  
   s = decode1(d);  
   break;  
  case 2:  
   s = decode2(d);  
   break;  
  default:  
   break;  
  }
* 繰り返し処理の内部で try ブロックを利用しない  
  特に理由がない場合は繰り返し処理の外にtryブロックを記載する。  
  ただし、繰り返し処理内部で例外をキャッチし処理を行いたい場合は繰り返し処理の内部でtryブロックを利用してもよい。
* 良い例：
* for (String s : array) {  
   BigDecimal num;  
   try {  
   num = new BigDecimal(s);  
   } catch (NumberFormatException e) {  
   num = BigDecimal.ZERO;  
   }  
   //・・・  
  }

## 文字列操作

* 文字列同士が同じ値かを比較するときは、equals()メソッドを利用する  
  良い例：
* String s1 = "text";  
  String s2 = "text";  
  if (s1.equals(s2)) {  
   //・・・  
  }
* 悪い例：
* String s1 = "text";  
  String s2 = "text";  
  if (s1 == s2) {  
   //・・・  
  }
* 文字列リテラルはnew しない  
  良い例：
* String s = "";
* 悪い例：
* String s = new String();
* 更新される文字列にはStringBuilder クラスを利用する  
  良い例：
* StringBuilder builder = new StringBuilder();  
  for (String s : array) {  
   builder.append(s);  
  }  
  System.out.println(builder.toString());
* 悪い例：
* String string = "";  
  for (String s : array) {  
   string += s;  
  }  
  System.out.println(string);
* スレッドセーフ性が保証されていない箇所ではStringBufferクラスを利用する
* [※パフォーマンスについても記載しているので参考にしてください](#文字列連結)
* １ステートメントのみで行われる文字列の連結には+演算子を利用する
* 良い例：
* String s = s1 + s2;  
    
  return s1 + s2 + s3 + s4 + s5;
* 悪い例：
* String s = new StringBuilder(s1).append(s2).toString();  
    
  return new StringBuilder(s1).append(s2).append(s3).append(s4).append(s5).toString();
* 更新されない文字列にはString クラスを利用する
* 文字列リテラルと定数を比較するときは、文字列リテラルのequals()メソッドを利用する  
  良い例：
* private static final String PROTOCOL\_HTTP = "http";  
    
  if (PROTOCOL\_HTTP.equals(url.getProtocol())) {  
    
  }
* 悪い例：
* private static final String PROTOCOL\_HTTP = "http";  
    
  if (url.getProtocol().equals(PROTOCOL\_HTTP)) {  
    
  }
* プリミティブ型とString オブジェクトの変換には、変換用のメソッドを利用する  
  良い例：
* int i = 1000;  
  String s = String.valueOf(i);// "1000"  
  s = NumberFormat.getNumberInstance().format(i);// 3桁区切り "1,000"  
    
  boolean b = true;  
  s = String.valueOf(b);// true/false  
  s = BooleanUtils.toStringOnOff(b);// on/off
* 文字列の中に、ある文字が含まれているか調べるには、contains()メソッドを利用する
* システム依存記号（ ¥n 、 ¥r など）は使用しない。  
  悪い例：
* String text = Arrays.stream(array)  
   .collect(Collectors.joining("\n"));

## 数値

* 誤差の無い計算をするときは、BigDecimal クラスを使う  
  浮動小数点演算は科学技術計算に利用するもので、誤差が発生する。これに対して、クラス「BigDecimal」は、文字列で数値の計算を行うので、金額などの正確な計算に適している。BigDecimal ではインスタンス生成時に指定された桁数での精度が保証される。
* 数値の比較は精度に気をつける  
  良い例：
* BigDecimal a = new BigDecimal("1");  
  BigDecimal b = new BigDecimal("1.0");  
  if (a.compareTo(b) == 0) {  
   System.out.println("一致");  
  }
* 悪い例：
* BigDecimal a = new BigDecimal("1");  
  BigDecimal b = new BigDecimal("1.0");  
    
  if (a.equals(b)) {  
   System.out.println("精度が違うためこの分岐には入らない");  
  }
* 低精度なプリミティブ型にキャストしない
* BigDecimalをString変換する際はtoString()ではなくtoPlainString()を利用すること  
  toString()を利用した場合、指数表記になることがあります。

## 日付

* 日付の文字列のフォーマットには、SimpleDateFormatまたはDateTimeFormatterを使う  
  良い例：
* Date date = new Date();  
  SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy/MM/dd");  
  String s = dateFormat.format(date);

## 三項演算子

* 入れ子の三項演算子の利用は禁止  
  可読性が悪くなるので三項演算子を入れ子で行うのは禁止。

## コレクション

* Java2以降のコレクションクラスを利用する  
  Vector クラス、Hashtable クラス、Enumeration 等は、特にこれらを利用する理由がなければ、インターフェースを統一する目的で、これらの代わりにList(ArrayList クラス)、Map(HashMap クラス)、Iterator を使用すること。List などのインターフェースを利用することでJDK1.2 で整理されたわかりやすいメソッドを利用でき、また、インターフェースの特性から呼び出し元を変更せずに実装クラスを変更することができる。
* 特定の型のオブジェクトだけを受け入れるコレクションクラスを利用する
* 良い例：
* List<Map<String, String>> list = new ArrayList<>();
* ListのソートはList.sort()を利用する  
  Listクラスの要素をソートする際はJava8で追加されたList.sort()を利用すること。  
  Java 7以前で利用されていたCollections.sort()は利用しないこと。
* Collection.forEach()は利用しない。拡張for文の利用を検討する  
  Java8で追加されたメソッド。  
  拡張for文を利用したほうが多くの場合でデバッグに有利であり、可読性においてもforEachの優位性は少ないため、forEachは原則利用しない。拡張for文を利用する。  
  具体的には下記のメソッドを利用しないこと。
  + Collection#forEach
  + Set#forEach
  + List#forEach
* ※ Map#forEachはこの限りではない
* 良い例：
* for(String s : List.of("A", "B")) {  
   //処理  
  }
* 悪い例：
* List.of("A", "B").forEach(s -> {  
   //処理  
  });
* ただし、メソッド参照で処理できる場合はforEachを利用する。  
  （デバッグのデメリットがほとんどなく、他と比較して処理効率が良いため）
* 良い例：
* List.of("A", "B").forEach(this::process);
* 悪い例：
* for(String s : List.of("A", "B")) {  
   this.process(s);  
  }
* Arrays.asList()は利用せず、List.of()を利用する  
  Java9で追加されたメソッド。  
  配列をListに置き換える場合や、単純な固定のListを生成する際にはList.of()を利用する。
  + Arrays.asList()とList.of()の違い  
    List.of()で生成したListは、完全に不変（Immutable）なListで、  
    Arrays.asList()で生成したListは、サイズのみ不変で、set等による値の操作が可能なListです。  
    また、setを行った場合、Arrays.asList()に与えられた配列インスタンスにも影響します。

## ラムダ式・メソッド参照・コンストラクタ参照

* ラムダ式が利用できる箇所はラムダ式を利用してよい  
  [※パフォーマンスについても記載しているので参考にしてください](#ラムダ式メソッド参照コンストラクタ参照-1)
* ただし、メソッド参照・コンストラクタ参照が利用できる場合はメソッド参照・コンストラクタ参照を利用する
* 良い例：
* String::compareToIgnoreCase
* 悪い例：
* (s1, s2) -> s1.compareToIgnoreCase(s2)
* 良い例：
* BigDecimal::add
* 悪い例：
* (b1, b2) -> b1.add(b2)
* ラムダ式記述の際、型宣言は省略記法で記述する
* 良い例：
* (s1, s2) -> s1 + "=" + s2
* 悪い例：
* (String s1, String s2) -> s1 + "=" + s2
* 原則ラムダ式内の行数は1行とする  
  複数行で利用したい場合は、privateメソッドを作成しメソッド参照を利用する
* 良い例：
* this::getMax  
    
  private int getMax(int i1, int i2) {  
   if (i1 > i2) {  
   return i1;  
   } else {  
   return i2;  
   }  
  }
* 悪い例：
* (i1, i2) -> {  
   if (i1 > i2) {  
   return i1;  
   } else {  
   return i2;  
   }  
  }
* 原則ラムダ式は1行記述に限定するので、中カッコ、returnは必ず省略する
* 良い例：
* (s1, s2) -> s1 + "=" + s2
* 悪い例：
* (s1, s2) -> {  
   return s1 + "=" + s2;  
  }

## 実質的final（effectively final）

* 実質的finalを利用する  
  変数宣言にfinalを記載しなくてよい。

## Stream API

* 利用してよい  
  [※パフォーマンスについても記載しているので参考にしてください](#stream-api-1)
* 並列ストリームは利用しないこと  
  悪い例：
* Stream<?> s = list.parallelStream();  
  Stream<?> s = list.stream().parallel();
* StreamAPI記述の際の改行位置は、各中間処理・末端処理前のピリオドの前で改行する
* 良い例：
* List<Character> alphabetLower = list.stream()  
   .filter(Character::isAlphabetic)  
   .map(Character::toLowerCase)  
   .collect(Collectors.toList());
* 悪い例：
* List<Character> alphabetLower = list.stream().filter(Character::isAlphabetic)  
   .map(Character::toLowerCase).collect(Collectors.toList());  
    
  List<Character> alphabetLower = list  
   .stream()  
   .filter(Character::isAlphabetic)  
   .map(Character::toLowerCase)  
   .collect(Collectors.toList());
* Indentは統合開発環境の提供するフォーマッタに合わせる
* 中間処理の数は3つ（3行）程度までを推奨する  
  中間処理の記述が多くなると可読性も悪くなり、デバッグも難しくなるため、3行程度を目安にロジックを検討すること。
* コメントは、原則として処理中には記載しない  
  難解になってしまった場合のみ処理中の記載を認める
* 良い例：
* // クラスFooのフィールドStrの値で昇順にソートし、フィールドStrの要素を取得して処理する。  
  hogeList.stream()  
   .sorted(Comparator.comparing(Foo::getStr))  
   .map(Foo::getStr)  
   .forEach(this::proc);
* 悪い例：
* hogeList.stream()  
   .sorted(Comparator.comparing(Foo::getStr)) //クラスFooのフィールドStrの値で昇順にソート  
   .map (Foo::getStr) //フィールドStrの要素を取得  
   .forEach(this::proc); //処理  
    
    
  hogeList.stream()  
   //クラスFooのフィールドStrの値で昇順にソート  
   .sorted(Comparator.comparing(Foo::getStr))  
   //フィールドStrの要素を取得  
   .map (Foo::getStr)  
   //処理  
   .forEach(this::proc);
* Streamは極力変数代入しないこと  
  Streamは中間処理、末端処理を行うと使用済みとなり、以降同じインスタンスは利用できない。  
  変数代入はほとんどの場合意味をなさず、むしろミスの元となるため極力変数代入はしないこと。
* 良い例：
* List<String> list1 = Stream.of("A", "B", "C")  
   .map(String::toLowerCase)  
   .collect(Collectors.toList());  
    
  List<String> list2 = Stream.of("A", "B", "C")  
   .map(s -> s + s)  
   .collect(Collectors.toList());
* 悪い例：
* Stream<String> stream = Stream.of("A", "B", "C");  
  Stream<String> stream1 = stream.map(String::toLowerCase);  
  List<String> list1 = stream1.collect(Collectors.toList());  
    
  Stream<String> stream2 = stream.map(s -> s + s);//コーディングミス streamは使用済のためエラーになる  
  List<String> list2 = stream2.collect(Collectors.toList());

## Optional

* Optionalは同メソッド内で値を取り出す場合は極力変数代入しないこと  
  Optionalとその値の変数は同じものを示す名前となり、同じ意味の変数名が複数現れることで可読性が下がるため、Optionalの変数代入は行わないこととする。
* 良い例：
* Employee employee = findEmployee(employeeId)  
   .orElseThrow(IllegalArgumentException::new);
* 悪い例：
* Optional<Employee> employeeOpt = findEmployee(employeeId);  
  Employee employee = employeeOpt.orElseThrow(IllegalArgumentException::new);
* 直接、値を取り出すことなくOptionalでのみ扱う場合はOptionalを変数代入してもよい。
* 良い例：
* Optional<Employee> employee = findEmployee(employeeId);  
    
  Dept dept = employee.map(Employee::getDivision)  
   .map(Division::getDept)  
   .orElse(null);  
    
  Role role = employee.map(Employee::getRole)  
   .orElse(null);  
    
  //-----  
    
  Optional<Employee> employee = findEmployee(employeeId);  
  //・・・処理  
  return employee;

## ストリーム（InputStream OutputStream）

* ストリームを扱うAPI を利用するときは、try-with-resources文で後処理をする
* 良い例：
* try(InputStream inputStream = Files.newInputStream(Paths.get("HOGE.txt")) {  
   //inputStreamに対する処理を記載  
  }
* ObjectOutputStream ではreset()を利用する

## リソースの解放

* リソース解放を必要とするクラスを利用するときは、try-with-resources文で後処理をする
* 良い例：
* try(InputStream inputStream = Files.newInputStream(Paths.get("HOGE.txt")){  
   //inputStreamに対する処理を記載  
  }
* リソース解放を必要とするクラスを作成する場合はAutoCloseableをimplementsする  
  AutoCloseableをimplementsすることでtry-with-resources文が利用できるようになります。

## 例外

* catch 文で受け取る例外は、詳細な例外クラスで受け取る
* 良い例：
* try (InputStream inputStream = Files.newInputStream(Paths.get("HOGE.txt")) {  
   //・・・  
  } catch (IOException e) {  
   log.error("Error", e);  
   throw e;  
  }
* 悪い例：
* try (InputStream inputStream = Files.newInputStream(Paths.get("HOGE.txt")) {  
   //・・・  
  } catch (Exception e) {//範囲が広すぎる例外クラスの利用はNG  
   log.error("Error", e);  
   throw e;  
  }
* Exception クラスのオブジェクトを生成してスローしない
* catch ブロックでは基本、例外処理をする。ただし処理を書いてはいけない部分もあるので、その部分については、"// ignore" というコメントを記述すること。
* 例外クラスは無駄に定義しない

## ガベージコレクション

* 原則finalize()のオーバーライド実装は禁止
* もしfinalize()をオーバーライドした場合はsuper.finalize() を呼び出す
* アプリケーションからfinalize()を呼び出さない

# コメント規約

## よいコメントの鉄則

* コードを明確化するコメントを書く  
  コードにコメントを書く理由は、自分自身、一緒に仕事をしている人、後に関わる開発者にとってコードをより理解しやすいものにするためである。
* コメント化する価値がないプログラムならば、実行するに値しない  
  有用な格言。コメントは必須。
* 過剰な装飾は使わない （例：見出し状のコメント）  
  1960年代から1970年代の典型的なCOBOLプログラマにはアスタリスク( \* )でコメントを囲った箱を書く習慣があった。彼らの芸術的な主張を表わしているのかもしれないが、率直に言えばそれは製品に加わるちょっとした価値に比べれば大きな時間の無駄である。かわいいコードではなくきれいなコードを書くはずである。さらに、コードを表示するディスプレイや印刷するプリントに使われるフォントはプロポーショナルだったりそうでなかったりして、箱をきれいに整列させることは難しい。
* コメントはシンプルに  
  かつて見たもっとも最良のコメントは、シンプルな要点をまとめた注釈であった。なにも本を書く必要はなく、他の人がコードを理解するに十分な情報を提供するだけでよいのである。
* コードを書く前に先にコメントを記述する  
  コードをコメント化する最良の方法は、コードを書く前にコメントを書くことである。それが、コードを書く前にコードがどのように動作するかについて考えるよい機会となり、コメントの存在を保障することにもつながる。少なくともコードを書いた時にコメントすべきである。コメントによってコードが理解しやすくなることで、コードの開発中にアドバンテージを得ることができる。コードにコメントを書く時間を費やせば、それによって得られるものがある。
* コメントには、なぜそうなのかを書く。コードを読めば分かることを書かない  
  基本的に、コードの一部分を見ればそれが何かを理解することはできる。例えば、以下のコードを見て、$1000以上の注文については5%ディスカウントされることは理解できる。なぜそうなのか？大きな注文ではディスカウントがつきものだというビジネスルールがあるのだろうか？大きな注文に時間限定サービスがあるのか、それともずっとサービスがあるのか？これを書いたプログラマの気前がよかったのか？  
  どこかソースコード中か別な文書にコメントされていない限り、それがなぜなのかを知ることはできない。
* if (grandTotal >= 1000.00) {  
   grandTotal = grandTotal \* 0.95;  
  }
* なお、メソッドコメントには、適切なjavadocコメント（タグ）のほかに、以下の内容も可能な限り明記すること。
  + 副作用のある処理の場合は、その内容　（→ メソッドの引数オブジェクトがメソッド内で変更されるケースなど）
  + 既知のバグ　（→ 判明しているが修正しないことにした場合など）
  + 影響のある事前条件、事後条件　（→ メソッドが正しく動作するための前提について）
  + 並行性　（→　マルチスレッドでアクセスされた場合の動作について）
  + 該当メソッドの使用例やサンプルコード
* TODOコメント  
  設計者確認待ち、共通処理の作成待ちなどの理由により、実装時にTODOがある場合、下記のようにコメントを記述する。  
  （EclipseのTODOコメント形式を採用）  
  例）
* //TODO：ワークフローの仕様決定待ち　関連チケット#12345

## Javaコメント（3種類）の使い分け

Javaでは3種類のコメントが使える。javadocコメントは/\*\*で開始され、\*/で終わる。C風コメントは/\*で開始され\*/で終わる。単一行コメントは//で開始され、そのソースコード行が終わるまで続く。以下の表ではコメントの使い方とその例を示す。（コメントのスタイルに関しては、前述の「標準規約に準拠したコーディング例」を参照）

|  |  |  |
| --- | --- | --- |
| コメント種類 | 使用方法 | 例 |
| javadocコメント/\*\* comment \*/ | interface、class、メソッド、フィールドの直前に書く。コメントはjavadocによって処理され、外部ドキュメント(HTML)として生成される。（この形式以外のコメントはドキュメントとして出力されないことに注意） | /\*\* \* 顧客（Customer）- \* 顧客はわれわれがサービスまたは製品を売った人物 \* もしくは組織のいずれかである。 \* @author 開発太郎 \*/ |
| C 風コメント/\* comment \*/ | 特定のコードを無効化したいが、後で使用するかもしれないので残しておくためにコメント化する時や、デバッグ時に一時的に無効化するときに使用する。 | /\* このコードはJ.T.Kirkによって1997.12.9に前述のコードと置き換えたためコメント化した。2年間不要であるならば削除せよ。 ... (ソースコード) \*/ |
| 単一行コメント// comment | メソッド内にて、ビジネスロジック、コードの概要、一時変数の定義内容などを記述する。 | // 1995年2月に開始されたX氏の寛大なキャンペーンで// 定められた通り1000$を超える請求には、全て5%割引を// 適用する。 |

※ ロジック中に、頻繁にC風コメントでコメントを書くとまとめてコメントアウトする場合に不便なため、基本的にロジック中では単一行コメントを利用すること。

# パフォーマンス

パフォーマンスを考慮したJavaのコーディングについて以下に示す。

※ パフォーマンスはjreのバージョンやスペックによって変化します。本内容はjre1.8.0\_74での検証結果を元にした内容です。

※ 性能計測結果についての記載がありますが、あくまでも参考値です。性能を保証するものではありません。

## Stream API

Java8で追加されたStream APIでの記述は、可読性も高く、簡潔に書けますが、パフォーマンス・性能面で注意が必要な場合があります。

Listの処理を行う際、拡張for文で処理する場合はIteratorインスタンスが1つだけ生成されますが、Stream APIで処理する場合、最初のStreamインスタンスに加え、各中間処理ごとにもStreamインスタンスが生成され、その分の性能劣化が懸念されます。  
以下に処理例と計測結果を記載します。

* 拡張for文
* List<String> list = //数値文字列のList  
  List<String> resultList = new ArrayList<>();  
  for (String string : list) {  
   if (string.endsWith("0")) {  
   resultList.add(string);  
   }  
  }  
  return resultList;
* Stream API
* List<String> list = //数値文字列のList  
  List<String> resultList = list.stream()  
   .filter(s -> s.endsWith("0"))  
   .collect(Collectors.toList());  
  return resultList;
* 計測結果

|  |  |  |
| --- | --- | --- |
| * 処理するListの件数 | * 拡張for文 (ms) | * StreamAPI (ms) |
| * 100万件 | * 7 | * 9 |
| * 1,000万件 | * 88 | * 114 |
| * 1億件 | * 949 | * 1,026 |
| * 2億件 | * 1,822 | * 2,081 |

小中規模の処理量であれば考慮するほどの性能差はありませんが、大量の処理が見込まれる場合は考慮が必要です。  
また、Stream APIは並列処理（スレッド処理）の機能をサポートしていますので、利用できる場合は並列処理も含めての検証が必要です。

## ラムダ式・メソッド参照・コンストラクタ参照

Java8で追加されたラムダ式・メソッド参照・コンストラクタ参照は、匿名クラスを利用するよりも効率的です。  
積極的な利用を推奨します。

以下にComparatorを生成した場合の計測結果を記載します。

* 匿名クラス
* Comparator<String> c = new Comparator<String>() {  
   @Override  
   public int compare(String o1, String o2) {  
   return o1.compareToIgnoreCase(o2);  
   }  
  };
* ラムダ式
* Comparator<String> c = (o1, o2) -> o1.compareToIgnoreCase(o2);
* メソッド参照
* Comparator<String> c = String::compareToIgnoreCase;
* 計測結果

|  |  |  |  |
| --- | --- | --- | --- |
| * 処理件数 | * 匿名クラス (ms) | * ラムダ式 (ms) | * メソッド参照 (ms) |
| * 10億回 | * 380 | * 0(計測不能) | * 0(計測不能) |
| * 100億回 | * 6,374 | * 0(計測不能) | * 0(計測不能) |
| * 1京回 | * (30秒以上) | * 14 | * 10 |

ラムダ式は外部の変数を利用する場合、匿名クラスとほぼ同じ動作をします。

* 匿名クラス
* new Comparator<String>() {  
   @Override  
   public int compare(String o1, String o2) {  
   return arg.equals("DESC") ? o2.compareToIgnoreCase(o1)  
   : o1.compareToIgnoreCase(o2);  
   }  
   }
* ラムダ式
* Comparator<String> c = (o1, o2) -> arg.equals("DESC") ? o2.compareToIgnoreCase(o1)  
   : o1.compareToIgnoreCase(o2);
* 計測結果

|  |  |  |
| --- | --- | --- |
| * 処理件数 | * 匿名クラス (ms) | * ラムダ式 (ms) |
| * 10億回（パラメータあり） | * 571 | * 572 |
| * 100億回（パラメータあり） | * 9,900 | * 9,864 |

## 文字列連結

### 文字列連結（繰り返し）

文字列連結を繰り返し処理中で行う際、+演算子で処理することはアンチパターンとして知られています。  
繰り返し処理中の文字列連結は、 StringBuilder、 StringJoiner、 StringBuffer を利用します。  
また、コレクション要素の結合であればString#joinが利用できます。

以下に処理例と計測結果を記載します。

* +演算子
* String s = "";  
  for (int i = 0; i < list.size(); i++) {  
   String string = list.get(i);  
   if (i > 0) {  
   s += ",";  
   }  
   s += string;  
  }  
  return s;
* StringBuilder
* StringBuilder sb = new StringBuilder();  
  for (int i = 0; i < list.size(); i++) {  
   String string = list.get(i);  
   if (i > 0) {  
   sb.append(",");  
   }  
   sb.append(string);  
  }  
  return sb.toString();
* StringBuffer
* StringBuffer sb = new StringBuffer();  
  for (int i = 0; i < list.size(); i++) {  
   String string = list.get(i);  
   if (i > 0) {  
   sb.append(",");  
   }  
   sb.append(string);  
  }  
  return sb.toString();
* String#join
* return String.join(",", list);
* 計測結果

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| * 処理するListの件数 | * +演算子 (ms) | * StringBuilder (ms) | * StringBuffer (ms) | * String#join (ms) |
| * 1,000件 | * 5 | * 0(計測不能) | * 0(計測不能) | * 0(計測不能) |
| * 1万件 | * 1,016 | * 1 | * 1 | * 1 |
| * 10万件 | * (30秒以上) | * 2 | * 5 | * 5 |
| * 100万件 | * (30秒以上) | * 29 | * 42 | * 51 |

### 文字列連結（定数）

基本的に処理中の文字列連結では+演算子は使わないで処理するほうがパフォーマンスが高くなりますが、定数の場合は+演算子で定義するほうがパフォーマンスが高いです。

たとえば以下のように、処理したい場合、

private static final String CONST\_A = "A";  
private static final String CONST\_B = "B";  
private static final String CONST\_AB = CONST\_A + CONST\_B;

StringBuilderで処理しようとすると以下のようになります。

private static final String CONST\_AB = new StringBuilder(CONST\_A).append(CONST\_B).toString();

しかし、これらをバイトコード上で確認するとそれぞれ以下のようになります。

* +演算子
* private static final java.lang.String CONST\_AB = "AB";
* StringBuilder
* private static final java.lang.String CONST\_AB;  
    
  static {};  
   0 new java.lang.StringBuilder [20]  
   3 dup  
   4 ldc <String "A"> [8]  
   6 invokespecial java.lang.StringBuilder(java.lang.String) [22]  
   9 ldc <String "B"> [11]  
   11 invokevirtual java.lang.StringBuilder.append(java.lang.String) : java.lang.StringBuilder [26]  
   14 invokevirtual java.lang.StringBuilder.toString() : java.lang.String [30]  
   17 putstatic jp.co.packagename.ClassName.CONST\_AB : java.lang.String [34]  
   20 return

+演算子を利用した場合コンパイル時に最適化され、文字列"A"と"B"をあらかじめ結合してclassが作成されます。  
StringBuilderを利用した場合は最適化はされず、記述した通りの処理が行われます。

計測した場合、下記のようになります。

* 計測結果

|  |  |  |
| --- | --- | --- |
| * 処理回数 | * StringBuilder (ms) | * +演算子 (ms) |
| * 5,000万回 | * 559 | * 0(計測不能) |
| * 1億回 | * 1,059 | * 0(計測不能) |

通常、定数処理を大量に処理することは考えられないので性能問題になることはありませんが、+演算子を利用したほうがパフォーマンスが高いこともあるということを理解してください。

## Listの種類

ListにはArrayListのようなRandomAccessをimplementsした、ランダムアクセスをサポートしているクラスと、  
LinkedListのようなランダムアクセスをサポートしていない（シーケンシャルアクセス）クラスが存在します。  
RandomAccessではないListは、List#getなどインデックスを利用するような操作のパフォーマンスが低いので注意してください。

以下に処理例と計測結果を記載します。

* for文(List#get(int)によるループ)
* int size = list.size();  
  for (int i = 0; i < size; i++) {  
   String s = list.get(i);  
   //処理  
  }
* 拡張for文
* for (String s : list) {  
   //処理  
  }
* forEach
* list.forEach(this::処理);
* 計測結果

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| * 処理するListの件数 | * ArrayListfor文(List#get(int)によるループ) (ms) | * LinkedListfor文(List#get(int)によるループ) (ms) | * ArrayList拡張for文 (ms) | * LinkedList拡張for文 (ms) | * ArrayListforEach (ms) | * LinkedListforEach (ms) |
| * 1万件 | * 0(計測不能) | * 73 | * 0(計測不能) | * 0(計測不能) | * 0(計測不能) | * 0(計測不能) |
| * 10万件 | * 0(計測不能) | * 7,576 | * 0(計測不能) | * 0(計測不能) | * 1 | * 2 |
| * 20万件 | * 0(計測不能) | * 17,740 | * 0(計測不能) | * 0(計測不能) | * 0(計測不能) | * 0(計測不能) |
| * 50万件 | * 0(計測不能) | * (30秒以上) | * 0(計測不能) | * 2 | * 0(計測不能) | * 2 |
| * 100万件 | * 1 | * (30秒以上) | * 0(計測不能) | * 4 | * 0(計測不能) | * 4 |
| * 1,000万件 | * 16 | * (30秒以上) | * 8 | * 45 | * 6 | * 44 |

ランダムアクセスをサポートしているListがシーケンシャルアクセス（iteratorを利用した処理など）で遅いということはないので、  
ループの処理は拡張for文等、Iteratorによるループで記述するのが無難です。  
List#getでの処理をすべて禁止することはできませんが、高いパフォーマンスが求められる場合はListの種類にも注目してみてください。

## オートボクシング

Javaにはintのようなプリミティブ型と、Integerのようなプリミティブ型の値をクラスとして扱うためのラッパークラスが存在し、  
これらを意識せずコードを記述できるオートボクシングという機能があります。

しかし、オートボクシングを利用しているためにコーディング時に気づかない不要な処理を行っている可能性があるため、性能問題を避けるためには理解が必要です。

### オートボクシング例

下記の2つの処理は等価ではありません。

* オートボクシングあり
* int value1 = 1;  
  Integer value2 = value1;  
  int value3 = value2;
* キャストなし
* int value1 = 1;  
  int value2 = value1;  
  int value3 = value2;

「オートボクシングあり」の処理を等価なオートボクシングの無い処理に置き換えると以下の処理になります。

* オートボクシングなし
* int value1 = 1;  
  Integer value2 = Integer.valueOf(value1);  
  int value3 = value2.intValue();

「オートボクシングあり」のコードをコンパイルするとバイトコード上は「オートボクシングなし」の処理を行うこととなり、  
もし「キャストなし」の処理を意図していたなら、Integer#valueOfとInteger#intValueの処理が不要な処理です。

### オートボクシング性能

性能差が少ないため、ほとんど問題にはなりませんが、FindBugs等、静的解析で検出される問題のため、理解が必要です。

以下に計測結果を記載します。

* オートボクシング
* int value1 = 1;  
  Integer value2 = value1;  
  int value3 = value2;
* オートボクシングなし
* int value1 = 1;  
  int value2 = value1;  
  int value3 = value2;
* 計測結果
* （単位：マイクロ秒）

|  |  |  |
| --- | --- | --- |
| * 処理回数 | * オートボクシング (マイクロ秒) | * オートボクシングなし (マイクロ秒) |
| * 2,000兆回 | * 10max:297min:0 | * 4max:154min:0 |

## StringからInteger・Longへの変換

数値文字列のStringをIntegerに変換するには、Integer#valueOf(String)を利用して下記のように記述します。

String s = "1";  
Integer value = Integer.valueOf(s);

しかし、下記のようにも記述できます。

String s = "1";  
Integer value = new Integer(s);

これらの違いは、  
new Integer(s)とした場合、必ずIntegerインスタンスが生成されますが、  
Integer.valueOf(s)とした場合は -128から127の間の数値であればキャッシュから取り出すためインスタンスを生成しません。

このため、前者のInteger#valueOf(String)を利用した記述のほうが効率的です。  
Long#valueOf(String)も同様です。

性能差が少ないため、ほとんど問題にはなりませんが、FindBugs等、静的解析で検出される問題のため、理解が必要です。

また、Stringからの変換だけでなく、intやlongからの変換も#valueOfが効率的ですが、オートボクシングを利用した場合、コンパイルで自動的にこれらの処理に変換されるため、記述することはありません。

## Stringからint・longへの変換

数値文字列のStringをintに変換するには、Integer#parseInt(String)を利用して下記のように記述します。

String s = "1";  
int value = Integer.parseInt(s);

しかし、オートボクシングが利用できるため、意図せず下記のように記述ミスをする場合があります。

String s = "1";  
int value = Integer.valueOf(s);//取得したIntegerインスタンスをオートボクシングでintにcastしている

String s = "1";  
int value = new Integer(s);//生成したIntegerインスタンスをオートボクシングでintにcastしている

「オートボクシング」の説明に記載した通り、性能に差が出るだけでなく、  
記述から明らかにミスであることが解るため、FindBugs等、静的解析で検出されるコードです。

longへの変換の場合はLong#parseLong(String)を利用します

以下に計測結果を記載します。

* 計測結果

|  |  |  |
| --- | --- | --- |
| * 処理回数 | * Integer.valueOf(String) (ms) | * Integer#parseInt(String) (ms) |
| * 1,000万回 | * 396 | * 318 |
| * 1億回 | * 4,060 | * 3,077 |

## BigDecimalのZEROとの比較

BigDecimalの正・負・ZEROの判定はBigDecimal#signumを利用します。  
compareToを利用してBigDecimal.ZEROと比較しても同じことができますが、signumを利用したほうが効率的です。

以下に処理例と計測結果を記載します。

* compareTo利用
* BigDecimal value = new BigDecimal("0.0");  
  if (value.compareTo(BigDecimal.ZERO) == 0) {
* signum利用
* BigDecimal value = new BigDecimal("0.0");  
  if (value.signum() == 0) {
* 計測結果
* （単位：マイクロ秒）

|  |  |  |
| --- | --- | --- |
| * 処理回数 | * compareTo利用 (マイクロ秒) | * signum利用 (マイクロ秒) |
| * 1京回 | * 527max:26,367min:0 | * 424max:21,213min:0 |

性能差が少ないので、必ずしもsignumを利用する必要はありませんが、大量に処理する場合など、高いパフォーマンスが求められる場合は意識してください。