Java コーディング標準

オブジェクト倶楽部 平鍋健児

hiranabe@esm.co.jp

初版 2000 年 11 月 22 日
改訂 1 版 2001 年 2 月 9 日
改訂 2 版 2001 年 2 月 10 日
改訂 3 版 2001 年 2 月 12 日
改訂 4 版 2001 年 2 月 14 日
改訂 5 版 2001 年 3 月 12 日
改訂 6 版 2001 年 3 月 24 日
改訂 7 版 2001 年 8 月 31 日
改訂 7 版 2001 年 10 月 31 日
改訂 8 版 2001 年 11 月 9 日
改訂 9 版 2002 年 11 月 5 日
改訂 10 版 2006 年 4 月 26 日

オリジナル：http://www.objectclub.jp/community/codingstandard/CodingStd.doc

目次

2. 方針

このコーディング標準は，ソフトウェア開発プロジェクト(※)において java でコーディングする際のルール，推奨，および迷った時の指針を提供するものである．

標準策定の方針は，読みやすくメンテナンスしやすいコードを書くことである．実際のコーディングにあたっては，プロジェクトメンバー全員がこのルールに合意していることが必要である．

実プロジェクトにおいては，このコーディング標準をカスタマイズして用いることを推奨する．

※　特に Extreme Programming プロジェクトでの利用を意識したものではなく, あらゆる Java を使ったプロジェクトでの利用を想定している．

3. ファイル構成

(1)　ファイル名

public クラスはそのクラス名の 1 ファイルにする．

例: public class Point は，Point.java に入れる．

パッケージ内の非パブリッククラスは，そのクラスが主に使われるパブリッククラスのファイルに含めて良い(この場合，*.java **と** *.class **が対応しないことに注意)．**

(2)　ファイルの位置

プロジェクトのルートディレクトリを決め，パッケージ名の “.” をディレクトリ階層に置

き換えた位置に入れる.

例: myProject.framework パッケージは, <ProjectRoot>/myProject/framework ディレクトリに配置.

例: JP.co.esm.wiki.extremedomo パッケージは,
<ProjectRoot>/JP/co/esm/wiki/extremedomo ディレクトリに配置

パッケージ名をドメイン名から作成するか, 任意に作成するかはプロジェクトの方針とする. 公開, 出荷予定であれば基本的にはドメイン名から作成することを推奨する.

(3) テストクラス名

クラス `ClassName` のユニットテストクラス名は, `ClassNameTest` とする. パッケージ毎テストは, `LastPackageNameTest` とする.

例: `Point2D` クラスなら, Point2DTest.java を作成.
例: `JP.co.esm.wiki.extremedomo`パッケージなら, ExtremeDomoTest.java を作成.

理由: 一貫性のある名前づけ. テストコードは使い方のサンプル, デモともなる.
別法: `ClassName` のユニットテストクラス名を `ClassNameUt` とする(`Ut` は UnitTest の略).

(4) テストクラスの位置

テストクラスは, 被テストクラスと同じディレクトリ, もしくは, そのサブディレクトリ "test" に配置する.

理由: 物理的に近い位置でないとメンテが忘れ去られる. 製品コードとの分離については, 別のツール(makefile, Ant の build.xml など)で調整可能.

別法: テストサブディレクトリ名を "ut" とする.

別法: テストクラスのディレクトリを分離する．JP.co.esm.wiki.extremedomo パッケージのテストを，test.JP.co.esm.wiki.extremedomo とする．

4. 命名規則

(5) パッケージ名

“.” で区切られた文字.

```
JP.co.your.domainname.projectname
junit.framework
```

(6) ファイル名

パブリックなクラス名は，コンパイラの規則により，ファイル名と同じでなければならない(大文字小文字の区別を含めて).

(7) クラス名

先頭大文字．あとは区切りを大文字.

```
CapitalizedWithInternalWordsAlsoCapitalized
```

(8) 例外クラス名

最後を Exception としたクラス名.

```
ClassNameEndsWithException
```

(9) インターフェイス名

クラス名に同じ．ただし，class と区別の必要があれば，最初に I を付ける．

```
INameOfInterface
```

また，クラスにある能力を加えるミキシン(mix-in)的な利用の場合，その能力を示す形容詞とし，-able を接尾にする．

例: `Runnable`, `Clonabe`, `Serializable`, …

(10) 実装クラス名

特に interface と区別の必要があれば，最後に Impl を付ける．

```
ClassNameEndsWithImpl
```

(11) 抽象クラス名

抽象クラス名に適当な名前が無いとき, Abstract から始まりサブクラス名を連想させる名前を付ける．

```
AbstractBeforeSubClassName
```

(12) 定数(static final)

大文字を “_” でつないだもの．

```
UPPER_CASE_WITH_UNDERSCORES
```

(13) メソッド名

最初小文字で，あとは区切りを大文字．

```
firstWordLowerCaseButInternalWordsCapitalized()
```

(14) ファクトリメソッド(オブジェクトを new するもの)

```
X newX()
X createX()
```

(15) コンバータメソッド(オブジェクトを別のオブジェクトに変換するもの)

```
X toX()
```

(16) 属性の取得メソッド

```
X x()
X getX()    // JavaBeans でプロパティとして扱える(推奨)
boolean isEnabled()  // JavaBeans でプロパティとして扱える(推奨)
```

(17) 属性の設定メソッド

```
void setX(X value) // JavaBeans でプロパティとして扱える(推奨)
```

(18) boolean 変数を返すメソッド

is + 形容詞，can + 動詞，has + 過去分詞，三単元動詞，三単元動詞 + 名詞.

```
boolean isEmpty() // JavaBeans でプロパティとして扱える(推奨)
boolean  empty() // だめ！'空にする'という動詞的な意味に取れるため良くない.
boolean canGet()
boolean hasChanged()
boolean contains(Object)
```

```
boolean containsKey(Key)
```

理由: if, while 文等の条件が読みやすくなる．また true がどちらの意味か分かりやすい．

(19) boolean 変数

形容詞，is + 形容詞，can + 動詞，has + 過去分詞，三単元動詞，三単元動詞 + 名詞．

```
boolean isEmpty
boolean dirty
boolean containsMoreElements
```

(20) 英語と日本語

すべての識別子の名前は英語を基本とし，別に，日英対応用語辞書を作成してプロジェクトの全ライフサイクルでメンテナンスすること．

(21) 名前の対称性

クラス名，メソッド名を付ける際は，以下の英語の対称性に気を付ける．

```
add/remove
insert/delete
get/set
start/stop
begin/end
send/receive
first/last
get/release
put/get
up/down
show/hide
source/target
open/close
```

```
source/destination
increment/decrement
lock/unlock
old/new
next/previous
```

(22) ループカウンタ

スコープ（通用範囲）が狭いループカウンタ，イテレータに `i`, `j`, `k` という名前をこの順に使う．

(23) スコープが狭い名前

スコープが狭い変数名は，型名を略したものを使って良い．

例： `ServeletContext sc = getServletContext();`

(24) 意味がとれる名前

変数名から役割が読み取れる名前を好め．

悪い例: `copy(s1, s2)`
良い例: `copy(from, to)` あるいは `copy(source, destination)`

(25) 無意味な名前

`Info`, `Data`, `Temp`, `Str`, `Buf` という名前は再考を要する．

悪い例: `double temp = Math.sqrt(b*b - 4*a*c);`
良い例: `double determinant = Math.sqrt(b*b - 4*a*c);`

(26) 大文字小文字

大文字と小文字は別な文字として扱われるが，それのみで区別される名前を付けてはならない.

(27) その他

その他，プロジェクトによっては以下の命名則を用いる場合がある.

ローカル変数:

```
lower_case_with_underscore
```

`private/protected` 変数:

`_prefixUnderscore` もしくは `suffixUnderscore_`

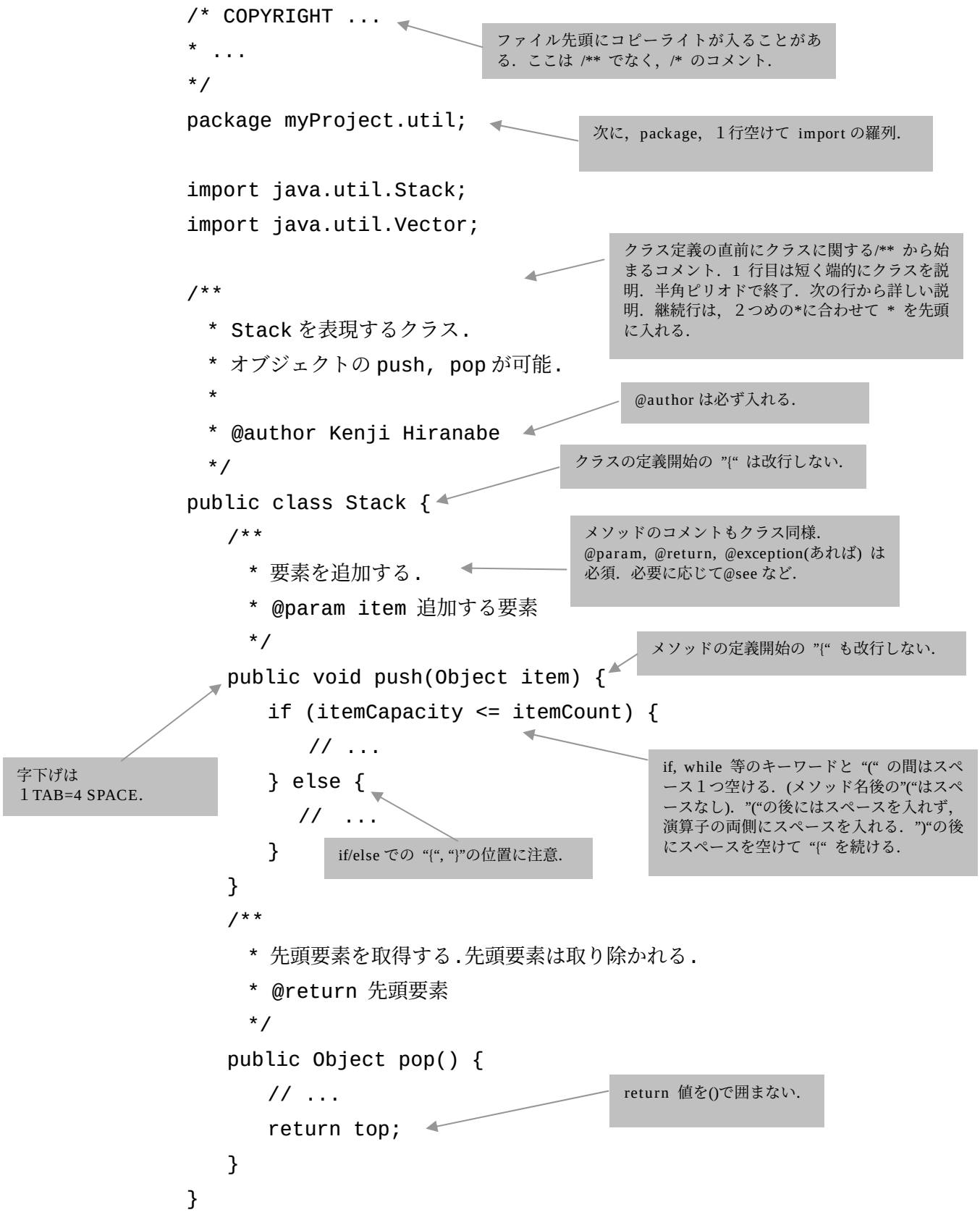
`static private/protected` 変数:

`__twoPrefixUnderscores` もしくは `twoSuffixUnderscores__`

5. ガイドライン

(28) コーディングスタイル

コーディングスタイルは，Sun Microsystems, Inc の JDK ソースに準じる．インデンテーションは基本的に K&R の C 言語スタイルと同じだが，クラスおよびメソッドの定義開始の”{“を改行せずに書く.



別法: インデント，{} の位置については，各自の創造性を阻害することのないよう，敢えて規定しない.

(29) 長い行

一行は最大 80 桁とし，それを超える場合は行を分割する．分割の指針は，(1) ローカル変数を利用，(2)カンマで改行，(3)優先度の低い演算子の前で改行，とする.

例:

```
double  length  =  Math.sqrt(Math.pow(Math.random(),  2.0)  +
Math.pow(Math.random(), 2.0));

// 方針(1)
double xSquared = Math.pow(Math.random(), 2.0);
double ySquared = Math.pow(Math.random(), 2.0);
double length = Math.sqrt(xSquared + ySquared);

// 方針(2)
double length = Math.sqrt(Math.pow(Math.random(), 2.0,
                          Math.pow(Math.random(), 2.0);

// 方針(3)
return this == obj
               || (this.obj instanceof MyClass
                   && this.field == obj.field);
```

(30) 長い宣言行

クラス，メソッドの宣言が長い場合，(1)extends/implements/throws 節で改行，(2) カンマで改行とする.

例:

```
public class LongNameClassImplemenation
             extends AbstractImplementation,
             implements Serializable, Cloneable {
   private void longNameInternalIOMethod(int a, int b)
                 throws IOException {
         // …
   }
   public void longMethodSignature(int a, int b, int c,
                                   int d, int e, int f) {
         // …
   }
   // …
}
```

(31) import

`import` では，`*` をなるべく使わない．同じパッケージから３つ以上のクラスを import する場合は，`*` を使う．

理由: 依存性を明確化．多くの*を使った import があると，読み手が苦労する．ただし，あるパッケージをヘビーに使用する場合などは．バランスから*を使った方が見やすい．

(32) abstract class vs. interface

抽象クラス(abstract class)はなるべく使わず，interface を多用せよ．abstract class は，一部実装があり，一部抽象メソッドであるような場合にのみ使うこと．

理由: interface は幾つでも継承できるが，class は 1 つだけ．1 つから継承してしまうと，もう継承できずもったいない．

(33) public variable

インスタンス変数は，極力 public にせず，妥当なアクセスメソッドを設けること.

理由: オブジェクト指向の標準. クラスの内部状態に勝手にアクセスさせるのはよくない. ただし，以下の条件をすべて満たす場合，インスタンス変数を public にし，直接アクセスさせてもよい.

- そのインスタンス変数が他のインスタンス変数と独立であり，単独で変更されても内部の整合性をくずさない.
- どちらにしても，getX()/setX() メソッドを書く.
- インスタンス変数の実装が将来に渡って変更されないことが根拠付けられる.

また, 上記に当てはまらない場合でも, 極度に速度を気にする場合は, この限りではない. (ただし，慎重にコメントすること)

例: Stack クラスにおいて，itemCount 属性を public にしてはならないが，Point クラスにおいて x, y を public にしてもいいかもしれない(極端に速度を気にする場合，ex. Java3D の Vector/Point クラス).

```
Stack s = new Stack();
s.itemCount = 79;  //  ここで内部状態が崩れてしまう.


Point p = new Point();
p.x = 30;          //  内部状態の整合性は崩れない.
```

(34) 初期化

初期化をあてにしない(参照が null に初期化されているとか). また，２度初期化しない.

悪い例：

```
class PoorInitialization {
   private name = “initial_name”;
   public Sample() {
```

```
        name = “initial_name”;
    }
```

理由: 初期化に関するバグを最小化する.

(35) static 変数を避ける

`static` 変数(クラス変数)は極力避ける．(`static final` 定数は除く)

理由: `static` 変数は，セミグローバルと言って良い．より文脈依存なコードを招き，副作用を覆いかくしてしまう.

(36) final を好め

もしインスタンス変数が，作成されたあとに決して変化しないなら，`final` を積極的に使用する．また，メソッドの引数の参照先を変更しないなら，`final` とせよ.

理由: `final` は synchronization や コンパイルの効率化などが適用されやすい．内部クラスから引数を参照する場合は，`final` である必要がある.

(37) private vs. protected

`private` よりは，`protected` を使用すること.

理由: `private` は確実にそのクラス外からの使用をシャットアウトできるが，クライアントが，より細かいチューニングを subclass 化によって行うことを出来なくしてしまう.

別法: `private` をより好んで使え．`protected` にしてしまうと以降，変更が起ったときにそれを継承している全クラスに影響が出てしまう.

(38) get/set メソッド

無闇にインスタンス変数へのアクセスメソッド `getX()/setX()` を作成して `public` にすることは避ける．その必要性を検討し，もっと意味のあるメソッドにする．

理由: インスタンス変数は，他のインスタンス変数に依存していることが多い．クラス内部の整合性を崩してはならない．

(39) 変数隠し

スーパークラスの変数名と，同じ変数名を使う事は避けよ．

理由: 一般的にはこれはバグである．もし意図があるならコメントせよ．

(40) 配列宣言

配列の宣言は，`Type[] arrayName` とせよ．

理由: `Type arrayName[]` は C からの名残として残っているに過ぎない．

例:

```
static void main(String[] args);  --- ○
static void main(String args[]);  --- ×
```

(41) public メソッド

クラスの `public` メソッドは,「自動販売機のインターフェイス」を目標に．分かりやすく，使いかたを間違っても内部の整合性はこわれないように設計する．また，可能ならば契約による設計(Design by Contract)を行い，クラスの不変条件と共にメソッドの事前･事後条件をコードで表現せよ．

(42) 状態取得と状態変更の分離

メソッドは,「1 つの事」を行うように設計せよ．特に，状態変更と状態取得の２つのサービスを１つのメソッドで行わない. 状態を変更するメソッドの `return` 値は `void` にせよ．`Stack` の例では， `top()` と `removeTop()` の 2 つの方が，`pop()` より良い.

理由 1: １つの事を行うメソッドの方が分かりやすい(`Stack` の例は，慣用の方が強い為, `pop()` が好まれるだけ).

理由 2: 並行性の制御，例外の安全保証がしやすい(参考: C++では，pop()メソッドが例外安全に出来ない理由で，標準ライブラリでは pop()は値を返さない仕様となっている).

理由 3: サブクラス化による拡張がしやすい.

(43) this の return

クライアントの便宜を考えたつもりでも，`this` を `return` することはなるべく避ける.

理由: `a.meth1().meth2().meth3()` というような連鎖は，一般的に synchronization 上の問題の元になる.

(44) メソッドの多重定義

引数のタイプによるメソッドのオーバーロードはなるべく避ける(引数の数が違うものは OK である). 特に，継承と絡むと厄介である.

例:

× : `draw(Line), draw(Rectangle)`
○ : `drawLine(Line), drawRectangle(Rectangle)`
○ : `draw(Shape)`

(45) equals()と hashCode()

Object.equals()メソッドをオーバーライドするなら, 同時に hashCode()メソッドもオーバーライドせよ．逆も同じ．

理由: コンテナクラス(Hashtable)などに対応するため．

(46) clone()

もし，clone() メソッドが使われるようなら，Cloneable を実装し明示的にそれを書くこと．

例:

```
class Foo implements Cloneable {
	// ...
	public Object clone() {
		try {
			Foo foo = (Foo)super.clone();
			// Foo クラスの属性のクローン
			// ...
		} catch (CloneNotSupportedException e) {
			// Cloneable を implements しているのだから起り得ない
			throw new InternalError();
		}
	}
}
```

理由: shallow copy ではよくないケースがほとんどである．

(47) デフォルトコンストラクタ

可能ならいつでもデフォルトのコンストラクタ(引数がないもの)を用意せよ．

理由: Class.newInstance() でクラス名文字列からダイナミックにそのクラスを作成可能.

(48) abstract method in abstract classes

abstract クラスでは，no-op のメソッドを書くより，明示的に abstact メソッドと宣言せよ．また，共有可能なデフォルトの実装を用意できるなら，それを protected とし，サブクラスが 1 行で処理を書けるようにせよ.

理由: java コンパイラは，コンパイル時に実装されていない abstract メソッドを検出できるため，単に実装を忘れていただけ，というバグを回避できる.

(49) オブジェクトの同値比較

オブジェクトの比較では equals()メソッドを使い，== を使うな．特に， String の比較では == を使用してはならない.

理由: もし実装者が equals()を用意しているなら, それを使ってほしくて実装したはず. equals()のデフォルトの実装は，単なる == である.

理由: ユニットテストでは，assertEquals が equals() を利用しているため，簡単に同値テストが書ける.

(50) 宣言と初期化

ローカル変数は，初期値と共に宣言せよ.

理由: 変数の値に関する仮定を最小化する.

悪い例:

```
void f(int start) {
        int i, j;  // 初期値なしの宣言
```

```
        // 多くのコード
        // ...
        i = start + 1;
        j = i + 1;
        // i, j を使う
}
```

良い例:

```
void f(int start) {
        // 多くのコード
        // ...
        // 使う前，はじめて宣言と初期化
        int i = start + 1;
        int j = i + 1;
        // i, j を使う

}
```

(51) ローカル変数の再利用は悪

ローカル変数を使い回しするより，新しいものを宣言して初期化せよ.

理由: 変数の値に関する仮定を最小化する.
理由: コンパイラの最適化を助ける.

悪い例:

```
void f(int N, int delta) {
        int i;  // 初期値なしの宣言
        for (i = 0; i < N; i++) {
                // i を使う
        }
        for (i = 0; i < N; i++) {// また i を使う
                if (...) {
                        break;
```

```
                }
        }
        if (i != N) { // 最後まで回ったかの判定に i を使っている
                // ...
        }
        i = N – delta*2; // またまた再利用
        // ...
}
```

良い例:

```
void f(int N, int delta) {
        for (int i = 0; i < N; i++) {
                // i を使う
        }
        for (int i = 0; i < N; i++) {
                // 別の i を使う
                if (...) {
                        found = true;
                        break;
                }
        }
        if (found) {
                // ...
        }
        int total = N – delta*2; // 別の意味ある変数
        // ...
}
```

(52) if/while 条件中の “=”

`if`, `while` の条件には，代入 “=” を使ってはならない.

理由: ほとんどの場合，バグである．java コンパイラは，`boolean` 型でないかぎり，このバグをキャッチできるが.

(53) 大小比較演算子

“<”, “<=”を好んで使い，”>”, “>=”はなるべく避けよ.

理由: 大小の方向を統一し，右側を大きい方にすることで，混乱を避ける.

(54) キャスト

キャストは，できる限り `instanceof` の条件文で囲め.

```
C cx = null;
if (x instanceof C)
       cx = (C)x;
else
       evasiveAction();
```

理由: これで,「オブジェクトがそのインスタンスじゃなかったら?」とういことを常に考える癖がつく．ただし，キャスト出来ない場合がバグである，と判断できる場合は，この限りではない.

(55) 例外クラス

例外クラスは大域的な性格をもち，多用するとプログラムの流れを読みにくくしてしまうことを認識する.

例外クラスは，新たに作成するよりも，JDK 標準パッケージに含まれているものを利用できれば利用する.

例: `IOException`, `NoSuchFileException`, `IllegalArgumentException`, などは利用しやすい標準例外.

新たな例外の作成は，そのパッケージ全体のインターフェイスとして検討すること.

(56) メソッド引数の変更は悪

原則としてメソッドの引数は入力であり，出力としては使わないこと．すなわちメソッド内部で引数の状態を変更するメソッドを呼ばないこと．出力引数に新たなオブジェクトを代入しないこと(可能なら `final` とせよ).

悪い例:

```
void moveX(Point p, int dx) {
	p.setX(p.getX()+dx); // 引数を変更している(なるべく避ける)
}

void moveX(Point p, int dx) {
	p = new Point(p.getX()+dx, p.getY());
	// これは呼び出し側に伝わらない
}
```

例外: パフォーマンスを気にする場合

(57) メソッド引数の名前

メソッドの引数は, 読みやすいものにすること. 特に, インスタンス変数と重なった場合, this を活用し，引数の読みやすさを犠牲にしないこと.

悪い例:

```
void reset(int x_, int y_)  {
	x = x_;
	y = y_;
}
```

良い例:

```
void reset(int x, int y)  { // 引数名を x_, y_ などとしない
	this.x = x;
	this.y = y;
```

```
}
```

(58) toString()

`toString()` メソッドは可能ならいつでも実装すること.

理由 1: `System.out.println(object)` でいつでもプリントできる.
理由 2: ユニットテスト等で失敗した場合の表示が分かりやすくなる.

(59) switch,if/else の繰り返しは悪

`switch` 文で分岐する処理が現れた時には，よくない設計の兆候だと考え，ポリモーフィズムで実現できないか再考する．特に同じような `switch` が 2 箇所以上現れたら，必ずポリモーフィズム，FactoryMethod，Prototype パターン等でリファクタリングすること．`if/else` の連続も同様．さらに，`null` チェックを行う同様の `if` が多くの場所に現れたら，NullObject パターンの利用を検討せよ.

(60) String と基本型との変換

`int` から `String` またその逆変換は，以下のようにする(他の基本型も同様).

```
String s = String.valueOf(i);
int i = Integer.parseInt(s);
```

理由: 他の書き方もあるが，上記が最も分かりやすく効率的.
別法: (推奨しない)

```
String s = “” + i;
String s = new Integer(i).toString();
String s = Integer.toString(i);  // これは悪くない
int i = new Integer(s).intValue();
int i = Integer.valueOf(s).intValue();
```

(61) コレクション

環境が許せば，JDK1.2 以降のコレクションクラスを用いよ．すなわち，`Vector, Hashtable, Enumeration` ではなく，`List (ArrayList), Map(HashMap), Iterator` を好め.

理由 1: より簡潔で論理的，一貫性のあるメソッド名が使える.
理由 2: `List, Set, Map` インターフェイスにより，インターフェイスを変更せずに実装を取り替えることが可能.
理由 3: 同期化がオプションであるため，より高速なコードが書ける(可能性がある).

参考: JDK1.2 コレクションについての利用ガイド
http://www.objectclub.jp/technicaldoc/java/jdk

6. コメント

(62) javadoc の活用

/** コメント */ を多いに活用すること．このコメントは，javadoc や同様のツールによって HTML 形式でのドキュメントに変換することができる.

java のコメントには，3 種類ある.

/** ... */　javadoc コメント．html 形式でドキュメント出力される.
/* */　通常のコメント,内部的
//　通常のコメント,内部的

public クラス，メソッド，フィールドには必ず /** */ コメントを付ける.

(63) 長いコメント

コメントが複数行に渡る場合は，最初の短い一文で何が言いたいかを書き，その後に長い

コメントを付けること.

(64) javadoc タグ

/** */コメント中，@ から始まるキーワード(javadoc タグ)を挙げる.

```
@author author-name
@param paramName description
@return description of return value
@exception exceptionName description
@see string
@see URL
@see classname#methodname
```

param, return に特別の注意が必要な場合，その旨コメントする．例えば，引数が出力用であり，変更される場合.

例 1:

```
/**
  * 境界ボックスを得る.
  *
  * @param b 境界ボックス(メモリと速度効率のため，出力引数)
  */
  void getBBox(BBox b) { b.min = this.min; b.max = this.max; }
```

(65) クラスコメント

/** */ コメントは，機能概要，すなわち外部的な仕様の記述に用い，クラスおよびメソッドの定義が始まる直前に書く．このコメントの第 1 行は，特別な扱いを持つ．すなわち，html の Method Index に使用される．よって，最初の 1 行は，コメント対象の外部的な機能の短い説明となる．その行は，半角ピリオド(.)，もしくは
 HTML タグで終る．この 1 行目に続けて，機能を説明する.

例:

```
/**
 * スタックを表現するクラス.
 * スタックは先入れ後だしのデータ構造.
 * <p>
 * 要素数は，count に保存し，要素は，Vector に入れる.
 *
 * @see      util.Vector
 * @author   yourNameHere
 */
public class Stack {
/**
* 現在の要素数.
* 非負の数であり，capacity 以下である.
*/
protected int count;


/**
 * 一番上の要素を取り出す.
 * 要素数は 1 つ減る.
 * <pre>
 * 使用例:
 * Stack s = new Stack(10);
 * s.push(99);
 * int i = pop(); // i は 99 のはず
 * </pre>
 * @return  上の要素
 */
 public int pop() { ... }
}
```

コメント中に，使用例など書く場合，<pre> </pre> で囲って自動つめ，改行を抑制するとよい.

(66) // vs. /* */

メソッドやクラスの内部的なコメントは，/* */ か，// を使用する．長さで判断してよい．1 行コメントは，// を使った方がよい．

例 1:

```
/*
* 戦略:
* 1. まず node を探す
* 2. clone する
* 3. inserter に clone を追加要請
* 4. 成功したら，node を削除
*/
```

例 2:

```
int index = -1; // -1 は invalid な値を意味する
```

参考: この方がさらによい．

```
static final int INVALID= -1;
int index = INVALID;
```

(67) Design by Contract(契約による設計)

契約による設計を行うため，プロジェクトに Assert クラスを用意せよ．Assert クラスを使って，契約を表現せよ．

例:

```
class Stack {
        private int capacity;
        private int itemCount;
        public void push(Object o) {
                Assert.require(o != null); // 事前条件
```

```
                // ...
                // ...
                Assert.ensure(this.contains(o)); // 事後条件
        }
        public boolean invariant() { // 不変条件
                Assert.invariant(0 <= capacity);
                Assert.invariant(0 <= itemCount);
                Assert.invariant(itemCount <= capacity);
                return true;
        }
}
```

注意: ユーザ入力チェックなどを assert してはいけない．バグを捕まえるために assert せよ．

補足: J2SE1.4 以降では，assert キーワードを利用せよ．

7. パフォーマンス

(68) まず計測

パフォーマンス改善はまず計測から始めよ．当てずっぽうではだめ．

(69) new

java では，new は時間が掛かる．ヘビーなループの中で new が呼ばれる場合，必要ならば出力引数を用いる．

```
X getX() {
        return new X(this.value);
}
```

が遅い場合，呼び出し側に new を任せ，

```
void getX(X x) {
       x.setValue(this.value);
}
```

とせよ．

(70) synchronized

synchronized は時間が掛かる．全クラスを同期化せず，必要な部分のみ synchronized せよ．また，Vector, Hashtable はデフォルトで同期化のオーバーヘッドがある．ArrayList, HashMap を好んで用い，必要な部分のみ同期化せよ (Collections.synchronizedCollection による外部同期).

(71) 変数への null の代入

使われない変数が大量に発生する場合，積極的に null を代入せよ．特に，配列の要素(パフォーマンス要求が厳しい場合).

理由: ガベージコレクションを助ける．

8. その他

(72) 自分で新しく作る前に相談

他人が作成したクラスに対するある操作が新たに必要となるとき，自分でそのクラスを extends して新たなクラスを作成したり，そのクラスをインスタンス変数として持つクラスを作成するより，まずそのクラスの作成者に相談すること．汎用的な形でその要望を満たしてくれれば，全体をコンパクトにできる．

(73) 複雑な設計は悪

設計で迷った場合，多くのケースは ‘Simplicity’ を重視した方が，java 言語の特性と良く合う．java 言語の設計原理は KISS(Keep It Small and Simple)である．また，後のメンテナビリティにも ‘Simplicity’ は重要である．

(74) パフォーマンス調整は測定後

最初からパフォーマンスを気にしたコーディングをするべきではない．読みやすさ，保守のしやすさを優先する．パフォーマンスは測定してから改善する．

(75) トリッキーなコードは悪

Java の平均プログラマに分かるようなコードを書く．演算子の順序，初期化に関する規則など，誰もが必ずしも自信をもって答えられないような仮定を持ち込まず，() を使って演算順序を明確にしたり，明示的な初期化を行った方が読みやすい．

```
悪い例: return cond == 0 ? a < b && b < c : d == 1;
良い例: return (cond == 0) ? ((a < b) && (b < c)) : (d == 1);
```

悪い例:

```
// 単位行列を作るが，時間もかかるし誰も読めない．
for (int i = 1; i <= N; i++)
       for (int j = 1; j <= N; j++)
              M[i-1][j-1] = (i/j)* (j/i);
```

(76) 100%正しいことはない

ここに書かれていることに，100% 準拠する必要はない．迷ったら考えを整理し，相談すること．十分な理由があってルールから外れることはよくある．コミュニケーションができるチームの助けとなることが，このコーディング標準の目的である．

9. 謝辞

このコーディング標準をまとめるにあたって，太田健一郎さん，栗原哲也さん，高橋徹さん，小藪隆史さん，牛尾剛さん，井芹義博，山崎貴弘さんから有用なコメントを頂きました．ありがとうございました.

10.参考資料

Kenji Hiranabe, Java コーディング標準(オリジナル)
http://www.objectclub.jp/community/codingstandard/CodingStd.doc

Jeff Langr, Essential JAVA STYLE – Patterns for Implementation
1999, Prentice Hall

Doug Lea's Draft Java Coding Standard
http://gee.cs.oswego.edu/dl/html/javaCodingStd.html

Mark Fussell's Java Development Standards
http://www.chimu.com/publications/javaStandards/index.html

Macadamian Technology coding conventions　for C++ and Java
http://www.macadamian.com/codingconventions.htm

AmbySoft Inc. Java Coding Standards, Elements of Java Style
http://www.ambysoft.com/essays/javaCodingStandards.html
http://www.ambysoft.com/books/elementsJavaStyle.html

Javasoft coding standards
http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html

以上