Javaコーディング標準（Java 5.0 対応版）

(C) Copyright https://www.code4u.jp All Rights Reserved.

初版 2008 年 1 月 29 日

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

1. 目次

Java コーディング標準 1
1. 目次 2
2. 方針 7
3. ファイル構成 7
4. 命名規則（全般） 8
(1) メソッド名、変数名は可能な限りデータベースの列名で使う。データベースの列名ではない場合は、ローマ字ではなく、英語名を使う。（★★★） 8
(2) 大文字・小文字の違いで名前を区別しない。（★★★） 8
5. 命名規則（パッケージ、クラス） 9
(3) パッケージ名はすべて小文字（☆☆☆） 9
(4) クラス名は先頭を大文字、各単語の区切りを大文字（★★★） 9
(5) Web 層クラスは継承元クラスによって決める（★★★） 9
(6) アクセッサ層クラスは、一対で作成する。（★★★） 10
(7) サブシステム共通クラス名はサブシステム名を最初につける。（★★★） 10
(8) 例外クラス名には名前の最後に Exception をつける（★★★） 10
(9) インターフェース名はクラス名に準ずる。（★★★） 10
(10) 能力付加型インターフェースは名前の最後に able をつける（★★★） 10
(11) 実装クラス名はクラス名に準ずる（★★★） 11
(12) 抽象クラス名は名前の最初に Abstract をつける。(☆☆☆） 11
(13) テストクラス名は「テスト対象クラス名＋Test」とする。（★★★） 11
6. 命名規則（メソッド） 12
(14) メソッド名は最初は英小文字、単語の区切りのみ大文字にする。（☆☆☆） 12
(15) ファクトリメソッドは、create で始める。（★★） 12
(16) 変換メソッドは、to で始める。（★★） 12
(17) 属性の取得メソッドは get で始める。（★★★） 12
(18) 属性の設定メソッドは set で始める。（★★★） 13
(19) boolean 変数を返すメソッドは、状態のわかる名前を指定する。（★★★） 13
(20) 名前の対称性を意識する（★★） 13
7. 命名規則（変数全般） 14
(21) （定数を除く）変数名は、最初は英小文字、単語の区切りのみ大文字にする。（★★★） 14
(22) データベースの列名にあてはまる変数は、列名をすべて小文字。（★★★） 14
(23) ハンガリアン記法を使用しない。（★★★） 14
(24) 定数は全て static final とし、すべて大文字、区切りは"_"とする。（☆☆☆） 14
(25) 変数隠し（その１）。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。（☆☆☆） 15
(26) 変数隠し（その２）。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。（★★） 15
(27) boolean 変数は true/false の状態がわかるようにする。（★★） 15

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

(28) 変数名には役割が反映された名前をつける。（★★） 15
(29) スコープが狭い変数名は省略した名前でもよい。（★★） 15
(30) for 文のループカウンタは、ネストごとに"i","j","k"・・・を使う。（★★） 16
8. コーディング標準（全般） 17
(31) 使われないコードは書かない。（☆☆☆） 17
(32) System.out.println() でデバッグログを出力しない。（★★★） 17
(33) オブジェクトの参照にはインターフェースを利用する。（★★） 18
(34) 推奨されない API を使用しない。（★★） 18
(35) 宣言は適切な権限で（★） 19
(36) プリミティブ型と参照型の違いを意識する（定義編）（★） 19
(37) プリミティブ型と参照型の違いを意識する（引数編）（★） 20
9. コーディング標準（メトリクス） 22
(38) 1 行のコードは、原則 80 文字まで。（☆☆☆） 22
(39) 1 メソッドの行数は約 20 行以下で記述する。（☆☆☆） 22
(40) 1 クラスの行数は約 600 行以下で記述する。（☆☆☆） 23
(41) 1 クラス内の public メソッド数は 30 個以下で定義する（☆☆） 23
(42) 循環的複雑さを大きくしない（★） 23
10. コーディング標準（フォーマット） 24
(43) タブを利用せず、空白文字を利用する。（☆☆☆） 24
(44) インデントは空白文字 4 文字分。（★★★） 24
(45) "{"の後にステートメントを記述しない（★★★） 24
(46) ブロックを開始する左中括弧({)の前に改行を挿入しない。（☆☆☆） 24
(47) カンマの後には空白文字を（☆☆☆） 25
(48) 代入演算子（=, +=, -=, …）の前後には空白文字を（☆☆☆） 26
(49) for 文内のセミコロンの後には空白文字を（☆☆☆） 26
(50) "++"や"--"とオペランドの間には空白文字を入れない（☆☆☆） 26
(51) ビット演算子（"|"、"&"、"^"、"<<"、">>"）の前後には空白文字を（☆☆☆） 26
(52) 論理演算子（"||"、"&&"）の前後には空白文字を（☆☆☆） 27
(53) 関係演算子（"<"、">"、 ">="、 "<="、"=="、"!="）の前後には空白文字を（☆☆☆） 27
(54) 算術演算子（"＋"、"－"、 "＊"、 "/"、"%"）の前後には空白文字を（☆☆☆） 27
(55) boolean 変数は既に比較済み（☆☆☆） 28
(56) return 文ではカッコを使わない（★★） 28
(57) 不等号の向きは左向き（"<"、"<="）にする（★★） 28
11. コーディング標準（コメント） 29
(58) /* ～ */コメントは使用しない。（★★★） 29
(59) 行末コメントはできるだけ使用しない。（☆☆） 29
(60) フィールドコメントは必ず記述する。（☆☆☆） 30
(61) クラスコメントは必ず記述する。（☆☆☆） 31
(62) メソッドコメントは必ず記述する。（☆☆☆） 32
(63) コメントは必要なものだけを簡潔に（★） 32
12. コーディング標準（クラス） 33

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

(64) java.lang パッケージはインポートしない(☆☆☆) 33
(65) import では, * を使用しない。(☆☆☆) 33
(66) 使用しない import は記述しない。(☆☆☆) 33
(67) フィールドを宣言する順序は public(+)、protected(#)、デフォルト(~)、private(-)の順(☆☆☆) 33
13. コーディング標準(メソッド) 34
(68) デフォルトコンストラクタは、原則作成する(☆☆) 34
(69) static メソッドのみのユーティリティクラスは、コンストラクタを private にする。(☆☆☆) 34
(70) static メソッドのみのユーティリティクラスは、final でクラスを宣言する。(☆☆☆) 34
(71) メソッドの引数は、最大 10 個とする。(☆☆☆) 34
(72) 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(☆☆☆) 34
(73) 1 行に 2 つ以上のステートメントを記述しない(★★★) 35
(74) サイズが 0 の配列を利用する(★) 35
(75) メソッドは 1 つの役割にする(★) 36
(76) 引数の数が同じメソッドのオーバーロードは利用しない(★) 37
(77) equals() メソッドを実装した場合は、hashCode()メソッドも実装する(★★) 38
14. コーディング標準(変数全般) 40
(78) 1 つのステートメントには 1 つの変数宣言(☆☆☆) 40
(79) リテラル(マジックナンバー)を使用しない。(☆☆☆) 40
(80) 配列宣言は「型名[]」にする(☆☆☆) 41
(81) 式内部での代入をしない。(☆☆☆) 41
(82) 定数は static final で宣言する(★★★) 41
(83) できるだけローカル変数を利用する(★) 42
(84) ローカル変数とインスタンス変数を使いわける(★) 42
15. コーディング標準(インスタンス変数) 43
(85) インスタンス変数は private にする(★★★) 43
(86) Action の関連するクラスでは、インスタンス変数を使用しない。(★★★) 43
(87) オブジェクトどうしは equals()メソッドで比較する。(★★★) 43
(88) static 変数を避ける(★★) 44
(89) インスタンス変数初期化のタイミングは「コンストラクタ」または、「インスタンス変数宣言時」または「初めて値が get されるタイミング(Lazy Initialization)」のいずれかにせよ(★) 44
16. コーディング標準(クラス変数) 46
(90) public static final 宣言した配列を利用しない。(★★★) 46
(91) クラス変数にはクラス名を使用してアクセス(☆☆☆) 46
17. コーディング標準(ローカル変数) 47
(92) ローカル変数は安易に再利用しない(★★) 47
(93) ローカル変数は利用する直前で宣言する(変数のスコープを意識する)(★★) 47
(94) ローカル変数は, 初期値と共に宣言せよ。(★★) 48
18. コーディング標準(制御構造) 49
(95) 三項演算子を利用しない。(☆☆☆) 49
(96) 制御文(if, else, while, for, do while)の“{}”は省略しない。(☆☆☆) 49
(97) ステートメントが無い{}ブロックを利用しない(☆☆) 50

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

(98) if/while の条件式で"="は利用しない(☆☆☆) 50
(99) switch 文では default を記述する(☆☆☆) 50
(100) 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(☆☆☆) 50
(101) 配列をコピーするときは System.arraycopy ()メソッドを利用する(★★) 51
(102) for 文のカウンタは 0 から始める(★★) 51
(103) ループ条件部でメソッドコールしない(★★) 51
(104) 論理演算子は、&& もしくは || を使用する。(★★) 52
(105) while ループより for ループを使用する(★) 53
(106) break や continue は使わないほうがわかりやすい(★) 54
(107) 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(★) 55
(108) 繰り返し処理中のオブジェクトの生成は最小限にする(★) 56
(109) 繰り返し処理の内部でtryブロックを利用しない(例外あり) (★) 57
19. コーディング標準(文字列操作) 58
(110) 文字列どうしが同じ値かを比較するときは、equals()メソッドを利用する。(☆☆☆) 58
(111) 文字列リテラルは new しない(★★★) 58
(112) 更新される文字列には StringBuilder(StringBuffer) クラスを利用する(★★★) 59
(113) StringBuilder(StringBuffer)クラスは初期容量を設定する。(★★) 59
(114) 更新されない文字列には String クラスを利用する(★★★) 60
(115) 文字列リテラルと変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(★★★) 60
(116) 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(★★★) 61
(117) 文字列の中に、ある文字が含まれているか調べるには、charAt()メソッドを利用する(★★) 61
(118) システム依存記号(\n、\r など)は使用しない(★★) 62
20. コーディング標準(数値) 63
(119) 低精度なプリミティブ型にキャストしない(★★★) 63
(120) 誤差の無い計算をするときは、BigDecimal クラスを使う(★) 63
(121) 数値の比較は精度に気をつける(★) 64
21. コーディング標準(日付) 65
(122) 日付を表す配列には、long の配列を利用する(★) 65
22. コーディング標準(コレクション) 66
(123) Java2 以降のコレクションクラスを好め(★★★) 66
(124) List クラスは初期容量を設定する。(★★) 66
(125) 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(★) 66
23. コーディング標準(ストリーム) 68
(126) ストリームを扱う API を利用するときは、finally ブロックで後処理をする(★★★) 68
(127) ObjectOutputStream では reset()を利用する(★★) 68
24. コーディング標準(例外) 70
(128) catch 文で受け取る例外は、詳細な例外クラスで受け取る(★★★) 70
(129) Exception クラスのオブジェクトを生成してスローしない(★★★) 71
(130) catch ブロックでは必ず処理をする(☆☆☆) 71
(131) Exception クラスの printStackTrace()を使用しない。(★★★) 72
(132) finally ブロックの中で return を記述しない(★★) 72

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

(133) Error、Throwable クラスを継承しない(★★) 72
(134) 例外クラスは無駄に定義しない(★★) 72
25. コーディング標準(スレッド) 73
(135) スレッドは原則 Runnable を実装 (★★) 73
(136) ウェイト中のスレッドを再開するときは notifyAll()メソッドを利用する(★★) 73
(137) Thread クラスの yield()メソッドは利用しない(★★) 73
(138) synchronized ブロックのあるメソッドを呼び出さない(★★) 74
(139) wait()、notify()、notifyAll()メソッドは、synchronized ブロック内から利用する(★★) 74
(140) wait()メソッドは while ブロック内から利用する(★★) 74
(141) ポーリングを利用せずに wait()、notifyAll()メソッドによる待ち合わせを利用する(★★) 76
(142) 同期化(synchronized)の適用は必要な部分だけにする(★★) 78
26. コーディング標準(ガーベッジコレクション) 80
(143) アプリケーションから finalize()を呼び出さない(★★★) 80
(144) System.gc()を実行しない(★★★) 80
(145) 明示的 null 代入(★★) 80
(146) finalize() をオーバーライドした場合は super.finalize() を呼び出す(★★) 80
27. コーディング標準(画面:JSP) 82
(147) JSP では Java コードではなくカスタムタグを使用する(★★) 82
(148) 文字の出力は、カスタムタグの write タグを使用する(★★★) 82
(149) html タグ、カスタムタグはすべて小文字で記述する。(★★★) 82
(150) JSP の変数は、page スコープを利用する。(★★) 83
(151) JSP のインクルードは、<jsp:include>を使用する。(★★) 83
(152) 半角スペースの表示は、 を使用する。(★★) 84
(153) <tabel>タグの見出しは、<th>で定義する。(★★) 84
(154) <th><td>タグでは、width 属性を指定せずに nowrap 属性を指定する。(★★) 84
(155) JSP で使用する文字コード指定は、Windows-31J とする(★★) 84
(156) 画面個別でスタイルの変更(文字列の右寄せ、文字色の変更など)を変更するときは、カスタムタグの style 属性を使用する。(★) 84
28. その他 85
(157) 自分で新しく作る前に相談(★) 85
(158) 複雑な設計は悪(★) 85
(159) トリッキーなコードは悪(★) 85
29. 参考資料 86

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

2. 方針

このコーディング標準は，〇〇〇ソフトウェア開発プロジェクト（以下、本プロジェクトと記述する）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す。

このコーディング標準は、本プロジェクト固有のコーディング標準で、その他は特に言及しないかぎり Oracle Java 5.0 のコーディング規約に従います。

Eclipse の CheckStyle でこのコーディング標準に記述されている内容をチェックするようにしていますので、開発時は本プロジェクトで準備された CheckStyle フォーマットを利用するようにしてください。
ただし、このコーディング標準の内容をすべて CheckStyle でチェックできるわけでありませんので、各自、このコーディング標準を意識して開発にあたってください。

このコーディング標準は、プロジェクトでの開発の基準となりますが、場合によっては、このコーディング標準に書かれている内容を無理に守ることで、ソースコードの可読性を下げることも考えられます。
その場合、フレームワーク担当者、上位者に相談、もしくはこのコーディング標準の修正を提案するなどし、有効に活用するようにしてください。

※ このコーディング標準内のサンプルの中には、わかりやすさを重視したため、コーディング標準を意図的に守っていないものも存在します。

タイトル行に重要度を示します。

（☆☆☆）	必ず守るべき項目です。 CheckStyle で警告メッセージを出力します。 もしくは、Eclipse のエラー/警告の設定で警告を出力します。
（☆☆）	CheckStyle で情報メッセージを出力します。 特別な理由がない限り、守るべき項目です
（★★★）	CheckStyle でチェックできませんが、必ず守るべき項目です。
（★★）	CheckStyle でチェックできませんが、特別な理由がない限り、守るべき項目です
（★）	コーディングの指針などです。 開発の参考としてください

3. ファイル構成

別途、フレームワーク資料を参照してください。

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

4. 命名規則（全般）

(1) メソッド名、変数名は可能な限りデータベースの列名で使う。データベースの列名ではない場合は、ローマ字ではなく、英語名を使う。（★★★）

名前をつける時はすべて英語を基本としてください。
ただし、データベースの列名と一致する場合は、その名称を優先して使用します。

＜違反サンプル＞
```
private boolean ZAIKO = false; // 違反
public boolean hasZaiko(){   // 違反
```

＜修正サンプル＞
```
private boolean stock = false; //修正済み
public boolean hasStock(){   // 修正済み
```

(2) 大文字・小文字の違いで名前を区別しない。（★★★）

Java の仕様で、大文字と小文字は別の文字として扱われますが、その違いだけで区別される名前を付けないでください。

＜違反サンプル＞
```
private int number;
private int Number;   // 違反
```

＜修正サンプル＞
```
private int carNumber;
private int trainNumber;    // 修正済み
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

5. 命名規則（パッケージ、クラス）

(3)　パッケージ名はすべて小文字（☆☆☆）

パッケージ名はすべて小文字で統一してください。これは、Java の一般的なルールです。

(4)　クラス名は先頭を大文字、各単語の区切りを大文字（★★★）

クラス名は先頭を大文字にしてください。クラス名が複数の単語で構成されている場合は、各単語の先頭（区切り）を大文字にしてください。これは、Java の一般的なルールです。

＜違反サンプル＞

```
public class sampleclass {    // 違反
}
```

＜修正サンプル＞

```
public class SampleClass {    // 修正済み
}
```

(5)　Web 層クラスは継承元クラスによって決める（★★★）

Web 層で使用するクラス名は、継承するクラスによって以下のように命名してください。

＜画面＞

JSP： 画面ID ＋ (*連番)　※ 複数必要な場合のみ

＜アクション＞

Action 継承クラス： 画面ID ＋ “Action”

PopulateHandler 継承クラス： 画面ID ＋ “PH”

CheckHandler 継承クラス： 画面ID ＋ “CH”

StateHandler 継承クラス： 画面ID ＋ “SH”

＜データモデル＞

データモデルクラス： 画面ID ＋*連番 ＋ “DTO”

*連番：0 から順に使用します。

＜サンプル＞

画面IDが X0001 の場合

- 画面： X0001.jsp （複数の場合は、X00010.jsp X00011.jsp）
- アクション： X0001Action.java、X0001PH.java、X0001CH.java、X0001SH.java
- データモデル： X00010DTO.java、X00011DTO.java

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

(6)　アクセッサ層クラスは、一対で作成する。（★★★）

アクセッサとデータクラスは一対で作成してください。

アクセッサ：　画面ID　＋　*連番　＋　“DA”

データクラス：　画面ID　＋　*連番　＋　“DO”

*連番：0 から順に使用します。

> *<サンプル>*
>
> 画面IDが X0001　の場合
>
> ・ アクセッサ：　X00010DA.java
>
> ・ データクラス；　X00010DO.java

(7)　サブシステム共通クラス名はサブシステム名を最初につける。（★★★）

サブシステム共通クラスは、サブシステム名を最初につけた名前にしてください。

共通クラスが1つのみのときは、「サブシステム名　＋ Common」としてください。

(8)　例外クラス名には名前の最後に Exception をつける（★★★）

例外クラスは、クラス名の最後を Exception とします。

<サンプル>

```
public class SampleException extends Exception{
}
```

(9)　インターフェース名はクラス名に準ずる。（★★★）

インターフェースの命名規則は基本的にクラス名に準じます。

例外として、クラス名との区別が必要であれば、先頭に”I”をつけます。

<サンプル>

```
public class Sample extends ISample{
}
```

(10)　能力付加型インターフェースは名前の最後に able をつける（★★★）

クラスに対して、ある能力を付加するようなインターフェース（例えば Runnable, Cloneable 等があり
ます）を定義する場合は、これにならって、その能力を示す形容詞(～able)を名前としてください。

<サンプル>

```
public class Sample extends Pluggable {
}
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

(11)　実装クラス名はクラス名に準ずる（★★★）

実装クラス名の命名規則は基本的にクラス名に準じます。
例外として、インターフェースとの区別が必要であれば、最後に”Impl”をつけてください。

＜サンプル＞

```
public class SampleImpl extends Sample {
}
```

(12)　抽象クラス名は名前の最初に Abstract をつける。(☆☆☆）

抽象クラスは、Abstract　で始まるクラス名とします。

＜サンプル＞

```
public class AbstractSample {
}
```

(13)　テストクラス名は「テスト対象クラス名＋Test」とする。（★★★）

テストクラス名は、「テスト対象クラス名　＋　Test」とします。

＜サンプル＞

```
public class SampleTest extends TestCase {
}
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

6. 命名規則（メソッド）

(14) メソッド名の最初は英小文字、単語の区切りのみ大文字にする。（☆☆☆）

メソッド名は、最初の単語の場合はすべて小文字で記述してください。
メソッド名が複数の単語で構成されている場合は、二語目以降の単語の先頭を大文字にしてください。
これは、Java の一般的なルールです。

＜違反サンプル＞
```
public void Samplemethod() {    // 違反
}
```

＜修正サンプル＞
```
public void sampleMethod() {   // 修正済み
}
```

(15) ファクトリメソッドは、create で始める。（★★）

オブジェクトを生成するメソッド（ファクトリメソッド）の名前は、”create”で始め、その後にこのメソッドで生成されるオブジェクト名を続けてください。

＜サンプル＞
```
public Sample createSample(){
}
```

(16) 変換メソッドは、to で始める。（★★）

オブジェクトを別のオブジェクトに変換するメソッド（コンバータメソッド）は、”to”で始め、その後に変換後のオブジェクトの名前を続けてください。

＜サンプル＞
```
public Sample toSample() {
}
```

(17) 属性の取得メソッドは get で始める。（★★★）

属性を取得するメソッド（ゲッターメソッド）の名前は、”get”で始め、取得する属性名を続けてください。
これは、JavaBeans の規約です。

＜サンプル＞
```
private String sample;
public String getSample(){
    return sample;
}
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

(18) 属性の設定メソッドは set で始める。(★★★)

属性を設定するメソッド(セッターメソッド)の名前は、"set"で始め、設定する属性名を続けてください。
これは、JavaBeans の規約です。

<サンプル>

```
private String sample;
public void setSample(String sample){
	this.sample = sample;
}
```

(19) boolean 変数を返すメソッドは、状態のわかる名前を指定する。(★★★)

boolean 変数を返すメソッド名は、その返り値 true/false がどのような状態を指しているのかわかる名前にしてください。
記述形式は、Yes または No を表す疑問文の形式(例:is+名刺)にすると良いでしょう。
一般的には、is、can、has などで始まる名前を使用します。

<サンプル>

```
public boolean isAsleep(){
}
public boolean canSpeak(){
}
public boolean hasExpired(){
}
public boolean exists(){
}
public boolean containsKey(){
}
```

(20) 名前の対称性を意識する(★★)

役割、機能等が対になっているメソッドの名前は英単語の対称性を意識したものにしてください。

<サンプル>

add/remove	first/last	open/close
insert/delete	get/release	source/destination
get/set	put/get	increment/decrement
start/stop	up/down	lock/unlock
begin/end	show/hide	old/new
send/receive	source/target	next/previous

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

7. 命名規則（変数全般）

(21) （定数を除く）変数名は、最初は英小文字、単語の区切りのみ大文字にする。（★★★）

変数名は、最初の単語の場合はすべて小文字で記述してください。
変数名が複数の単語で構成されている場合は、二語目以降の単語の先頭を大文字にしてください。
これは、Java の一般的なルールです。

(22) データベースの列名にあてはまる変数は、列名をすべて小文字。（★★★）

データベースの列名には、大文字小文字の区別がありません。（定義は大文字ですることが多いです。）
Java では変数名を小文字で扱うことが基本なので、列名をすべて小文字で扱います。
開発者によって単語の区切りの判断が異なる恐れがあるため、区切り文字を大文字にしません。

(23) ハンガリアン記法を使用しない。（★★★）

ハンガリアン記法は変数名やクラス名などの識別子に特別な接頭文字、または接尾文字をつけることで、他の人がその識別子を見たときに、識別子の使用方法、データ型情報、スコープ範囲などを分かるようにするための命名法です。
しかし、以下のような理由により本プロジェクトではハンガリアン記法を採用しません。

- 以前と比べ、ツール（本プロジェクトでは Eclipse）が進化しており、型の判別が容易にできます。
- Java ではプリミティブ型だけでなくオブジェクトを多く扱うため、そのたびに接頭語を決めると可読性にかけます。
- ローカル変数は、変数のスコープを意識して再利用をしないことを前提としますので、使用する直前で宣言を行うことで接頭語を使用しなくても型の判別ができます。
- 仕様変更でオブジェクト型の変更が必要な場合、識別子も変更しないと、名前と実態が一致しなくなるため、柔軟性に欠けます。
- 識別子の文字数が、接頭辞の分だけ長くなり、冗長になります。
- 変数名は、データベースの列名と一致させることを原則とするため接頭語を付加した場合に列名と変数名が一致しなくなります。

(24) 定数は全て static final とし、すべて大文字、区切りは”_”とする。（☆☆☆）

定数は全て static final 宣言し、変数名はすべて大文字で記述してください。
定数名が複数の単語で構成されている場合は、各単語の間は”_”で区切ってください。

＜違反サンプル＞

```
public static final int SampleValue = 10;   // 違反
```

＜修正サンプル＞

```
public static final int SAMPLE_VALUE = 10; //修正済み
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

(25)　変数隠し(その1)。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。(☆☆☆)

メソッドのパラメータには、クラスのメンバ名と競合する名前を付けないでください。
ただし、コンストラクタと setter メソッドは例外とします。

＜違反サンプル＞
```
List list;
public void sampleMethod(List list) {   //違反
}
```

＜修正サンプル＞
```
List list;
public void sampleMethod(List dataList) {  //修正済み
}
```

(26)　変数隠し(その2)。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。(★★)

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れてしまいます。
他の人の混乱を招きますので重複する名前は付けないようにしてください。

(27)　boolean 変数は true/false の状態がわかるようにする。(★★)

boolean 変数はその変数の true/false がどのような状態を指しているのかわかる名前にしてください。
記述形式は、Yes または No を表す疑問文の形式(例:is+名刺)にすると良いでしょう。
注意:上記規約に沿っていても、適切に状況を表現していなくては意味がありません。

＜サンプル＞
```
private boolean isAsleep;
private boolean canSpeak;
private boolean hasExpired;
private boolean exists;
private boolean hasValue;
```

(28)　変数名には役割が反映された名前をつける。(★★)

製造が終わった後に読み直したときに、何のために宣言した変数なのかわからなくなることがあります。
変数にはその役割が反映された名前を付けてください。

＜違反サンプル＞
```
private String str1;
private String str2;    // 違反
```

＜修正サンプル＞
```
private String serverName;
private String clientName;      //修正済み
```

(29)　スコープが狭い変数名は省略した名前でもよい。(★★)

本来、変数名には役割を反映した名前を用いるべきですが、スコープが狭い変数は識別が比較的容易な
なため、型名の略など、省略した名前に用いてもかまいません。

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

スコープが狭いとは、行数が少なく、ネストされたブロックが含まれていないことをさします。

<違反サンプル>

```
import java.io.*;
public class Sample {
    public static void main(String[] args){
        String str1 = "一行:";  //違反
        try{
            String str2 = br.readLine();   //違反ではない
            System.out.print(str1);
            System.out.println(str2);
            br.close();
        }catch(IOException e){
            e.printStackTrace();
        }
    }
}
```

<修正サンプル>

```
import java.io.*;
public class Sample {
    public static void main(String[] args){
        String index = "一行:";  //修正済み
        try{
            String str2 = br.readLine();   //違反ではない
            System.out.print(index);
            System.out.println(str2);
            br.close();
        }catch(IOException e){
            e.printStackTrace();
        }
    }
}
```

(30) for 文のループカウンタは、ネストごとに"i","j","k"・・・を使う。(★★)

for 文のループカウンタには、その階層ごとに"i"、"j"、"k"という文字をこの順序(アルファベット順)で使用してください。

<違反サンプル>

```
for(int j = 0; j < 10; j++){
    for(int n = 0; n < 10; n++){
        for(int t = 0; t < 10; t++){
        }
    }
}
```

<修正サンプル>

```
for(int i = 0; i < 10; i++){
    for(int j = 0; j < 10; j++){
        for(int k = 0; k < 10; k++){
        }
    }
}
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

8. コーディング標準（全般）

(31) 使われないコードは書かない。（☆☆☆）

使われていない private メソッドや変数、あるいはローカル変数を記述してはいけません。
これらが存在すると、コードの可読性を低下させ、無駄なリソースを消費することになります。
必要ないものの場合は削除してください。

<違反サンプル>

```
public class BadSample{
    public static void main(String[] args){
        usedMethod();
    }
    private void usedMethod(){
    }
    //違反 使われていないメソッド
    private void unusedMethod(){
    }
}
```

<修正サンプル>

```
public class BadSample{
    public static void main(String[] args){
        usedMethod();
    }
    private void usedMethod(){
    }
    // 修正済み
}
```

(32) System.out.println() でデバッグログを出力しない。（★★★）

デバッグログの出力には、専用のクラスが準備してあります。
System.out.println()でログを出力すると、どこでログが出力されてなんのためのログなのか判断できないことがあります。
開発時の一時的なログ出力をする場合であっても回避すべきです。

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

(33) 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ます。
しかし、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してください。
インターフェースを利用することのメリットは、コードの柔軟性が上がることにあります。
実装クラスを用いて宣言してしまった場合、後になって、実装クラスを変更するとなると、全ての参照個所を変更しなければなりません。
一方、インターフェースを利用して宣言していれば、インスタンスを生成する個所を変更するだけで、実装クラスを変更することが出来ます。

<違反サンプル>

```
import java.util.*;
public class BadSample {
    public static void main(String[] args){
        ArrayList sampleList = new ArrayList();   // 違反
        BadSample badSample = new BadSample();
        badSample.badMethod(sampleList);
    }
    private void badMethod(ArrayList input){
        // 違反:呼びだし元の実装クラスが
        // LinkedList クラスに変更されると、
        // このメソッドも変更しなければならない。
    }
}
```

<修正サンプル>

```
import java.util.*;
public class FixedSample {
    public static void main(String[] args){
        List sampleList = new ArrayList ();   //修正済み
        FixedSample fixedSample = new FixedSample();
        fixedSample.fixedMethod(sampleList);
    }
    private void fixedMethod(List input){
        // 修正済み:呼び出し元の実装クラスが
        // LinkedList クラスに変更されても、
        // このメソッドは変更しなくて良い。
    }
}
```

(34) 推奨されない API を使用しない。(★★)

“推奨されない”と指定されたクラス、メソッド等は使用しないでください。これらの機能が必要な際は、JavaDoc 内に示されている代替案等を参照してください。

<違反サンプル>

```
import java.util.Date;
public class BadSample {
    public static void main(String[] args){
        Date sampleDate = new Date();   //違反
        System.out.println(sampleDate.getYear());
    }
}
```

<修正サンプル>

```
import java.util.Calendar;
public class FixedSample {
    public static void main(String[] args){
        Calendar sampleCalendar =
                Calendar.getInstance();   //修正済み
        System.out.println(
                sampleCalendar.get(Calendar.YEAR));
    }
}
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

(35) 宣言は適切な権限で（★）

private、public 等、アクセス修飾子の意味を十分理解し、クラス、メソッド、変数、定数等は適切な権限で宣言するようにしてください。

＜違反サンプル＞	＜修正サンプル＞

//違反 クラス内部用のメソッドが public 宣言されている
public void internalMethod{
}
	// 修正済み
	private void internalMethod{
	}

(36) プリミティブ型と参照型の違いを意識する（定義編）（★）

プリミティブ型および参照型は、主に下記 3 点について特性が異なります。

◇実行速度とリソース消費

◇デフォルト値（インスタンス変数宣言時）

◇データ構造

以下にそれぞれの相違点について説明します。

◇実行速度とリソース消費

プリミティブ型は参照型と異なり、使用する際にオブジェクトを新たに生成する必要がありません。これにより、時間とリソースを節約することができます。

◇デフォルト値（インスタンス変数宣言時）

・プリミティブ型：型によって異なる

・参照型：null

（参考）プリミティブ型のデフォルト値

byte ： (byte) 0

short ： (short) 0

int ： 0

long ： 0L

float ： 0.0f

double ： 0.0d

char ： '\u0000'

boolean ： false

◇データ構造

プリミティブ型と参照型のデータ構造はそれぞれ下記のとおりです。

・プリミティブ型：値そのものを持っている

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

・参照型：値への参照（値がある場所へのポインタ）を持っている

したがって、代入操作の際には、それぞれの型において下記のような違いがあることに注意してください。
・プリミティブ型：値そのものが代入される

（例）

```
int i = 5;
int j = i;
```

上記の場合、i と j はそれぞれ'5'という値を持った別の変数となる。

i に対する変更が j に影響を及ぼすことはない（逆も同様）。

・参照型：値への参照が代入される

（例）

```
Integer k = new Integer(5);
Integet l = k;
```

上記の場合、i と j は同一の値を指すことになる。

i に対して変更を行うと、j も変更されることになる（逆も同様）。

(37) プリミティブ型と参照型の違いを意識する（引数編）（★）

引数がプリミティブ型である場合には値そのものが渡されますが、参照型の場合にはインスタンスへの参照が値として渡されます。したがって、参照型の引数の状態をメソッド内でむやみに変更すると、複数箇所からそのインスタンスが参照されているような場合、不整合を引き起こすことがあります。
参照型の引数を持つメソッドの場合は、渡された引数を直接操作しないようにしてください。

＜サンプル＞

```
import java.awt.Point;
public class Sample {
    public static void main(String[] args){
        int intValue = 0;
        Point point = new Point(0, 0);
        Sample sample = new Sample ();
        sample.printValue(intValue, point);
        sample.modifyValue(intValue, point);
        sample.printValue(intValue, point);
    }
    private void modifyValue(int value, Point point){
        value = 10;
        point.setLocation(55, 77);
    }
    private void printValue(int value, Point point){
        StringBuffer buffer = new StringBuffer();
        buffer.append(value);
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

```
            buffer.append(" : ");
            buffer.append(point);
            System.out.println(buffer);
        }
}
```

【実行結果】

```
0 : java.awt.Point[x=0,y=0]
0 : java.awt.Point[x=55,y=77]
```

上記の例では、プリミティブ型の変数 **intValue** と、参照型の変数 **point** を扱っています。

実行結果を見るとわかるように、**modifyValue()**メソッドを実行する前と後では、プリミティブ型の **intValue** は変化していないのに対し、参照型の **point** は変更されています。

プリミティブ型（**intValue**）は引数として値のコピーが渡され、そのコピー値に対して操作を行うため、**intValue** 自体には影響を及ぼしません。

一方、参照型（**point**）は引数として値の参照が渡されるので、メソッド内での操作が **point** の参照している値に影響を及ぼすことになります。

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

9. コーディング標準（メトリクス）

(38) 1 行のコードは、原則 80 文字まで。（☆☆☆）

1コード行は原則として 80 文字程度までにしてください。
モニタ上で見づらいだけでなく、メール送信や印刷の際に見にくくなったり、フォーマットがくずれてしまったりします。
改行場所の目安を以下に示します。

① extends/implements/throws 節で改行する
② カンマのあとで改行する
③ 優先度の低い演算子の前で改行する
④ 変数に代入するなどして、複数行に処理を分割する

また、改行した行の最初は修正サンプルの③の様に見やすい位置であわせるようにしてください。

＜違反サンプル＞

```
// 違反 長すぎる！！
double length = Math.sqrt(Math.pow(Math.random(), 2.0) + Math.pow(Math.random(), 2.0));
```

＜修正サンプル＞

① extends/implements/throws 節で改行

```
public class LongNameClassImplemenation
                    extends AbstractImplementation,
                    implements Serializable, Cloneable {
    private void longNameInternalIOMethod(int a, int b)
                        throws IOException {
    }
}
```

② カンマのあとで改行

```
public void fixedMethod(
                boolean booleanValue,
                String stringValue,
                int intValue){
}
```

③ 演算子の前で改行

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

④ 変数に代入

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

(39) 1 メソッドの行数は約 20 行以下で記述する。（☆☆☆）

1 メソッドの行数はコメントも含めて 20 行程度までが理想です。多くても 150 行程度にとどめてください。
これ以上行数が増えてしまう場合は、複数のメソッドに分割する等、設計の見直しを行ってください。

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

(40)　1 クラスの行数は約 600 行以下で記述する。（☆☆☆）

1 クラスの行数はコメントも含めて 600 行程度までが理想です。多くても 1000 行程度にとどめてください。
これ以上行数が増えてしまう場合は、複数のクラスに分割する等、設計の見直しを行ってください。

(41)　1 クラス内の public メソッド数は 30 個以下で定義する（☆☆）

1 クラス内の public メソッドは 30 個以下にとどめてください。
特に、共通ユーティリティクラスなどを複数の人で作業するクラスは、ソースが肥大化する傾向があります。
そのときは、クラスを分割する等、設計の見直しを行ってください。

(42)　循環的複雑さを大きくしない（★）

ブロックのネストの数、クラス/メソッド呼び出し、複合した条件部といった数が多くなれば多くなるほど、コードは複雑になります。
コードが複雑になるにつれて可読性/保守性は低下し、バグが発生する可能性も高くなってしまいます。

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

10. コーディング標準（フォーマット）

(43) タブを利用せず、空白文字を利用する。（☆☆☆）

インデントのために空白文字とタブをあわせて使っていると、エディタの設定によってはインデントがずれてしまい、コードの可読性が落ちる場合があります。
タブではなく、すべて空白文字を使うようにしてください。
開発前に Eclipse の設定をしてください。

(44) インデントは空白文字 4 文字分。（★★★）

インデントの空白は 4 文字分にしてください。統一することによって、他者にとっても読みやすくなります。

(45) "{"の後にステートメントを記述しない（★★★）

"{"の後にステートメントを記述してはいけません。どんなに短いステートメントでも、きちんと改行するようにしましょう。

＜違反サンプル＞

```
private void badSampleMethod() { int i = 0;
    // 違反
}
```

＜修正サンプル＞

```
private void fixedSampleMethod(){
    int i = 0;    // 修正済み
}
```

(46) ブロックを開始する左中括弧（{）の前に改行を挿入しない。（☆☆☆）

クラス、メソッド、制御文(if、switch、for、while、try、catch、)などの開始を意味する左中括弧の記述については、以下の2種類の記述が可能です。
開発者によって、記述にばらつきがあるとソースの可読性が下がります。
そのため、本プロジェクトでは左中括弧の後ろでの改行を標準とします。

① 左中括弧の後ろで改行（**本プロジェクトでの標準**）

```
public class Sample {
    public void method(String value) {
        try {
            if (･･･) {
            } else if (･･･) {
            } else {
            }
            for (･･･) {
            }
        } catch (Exception e) {
        }
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

```
        }
}
```

② 左中括弧の前で改行(使用禁止)

```
public class Sample
{
        public void method(String value)
        {
                try
                {
                        if (・・・)
                        {
                        }
                        else if (・・・)
                        {
                        }
                        else
                        {
                        }
                        for (・・・)
                        {
                        }
                }
                catch (Exception e)
                {
                }
        }
}
```

(47) カンマの後には空白文字を(☆☆☆)

カンマの後には空白文字を入れてください。

コードの可読性が向上します。

<違反サンプル>

```
// 違反
public void badSample (int score,int number){
}
```

<修正サンプル>

```
// 修正済み
public void fixedSample (int score, int number){
}
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

(48)　代入演算子（=, +=, -=, …）の前後には空白文字を（☆☆☆）

代入演算子の前後には空白文字を入れてください。コードの可読性が向上します。

＜違反サンプル＞

```
①a=1; // 違反
②b-=a; // 違反
```

＜修正サンプル＞

```
①a = 1; // 修正済み
②b -= a; // 修正済み
```

(49)　for 文内のセミコロンの後には空白文字を（☆☆☆）

for 文内のセミコロンの後には空白文字を入れてください。コードの可読性が向上します。

＜違反サンプル＞

```
for (int i = 0;i < 1000;i++){     //違反
}
```

＜修正サンプル＞

```
for (int i = 0; i < 1000; i++){     //修正済み
}
```

(50)　“++”や”--“とオペランドの間には空白文字を入れない（☆☆☆）

前置単項演算子または後置演算子の”++”と”--“と、オペランド（演算の対象となる値や変数）の間には空白文字を入れてはいけません。他の演算と間違えたりするなど、紛らわしくなってしまいます。

＜違反サンプル＞

```
System.out.println(“value :” + ++ a); // 違反
System.out.println(“value :” + a ++); // 違反
```

＜修正サンプル＞

```
System.out.println(“value :” + ++a); // 修正済み
System.out.println(“value :” + a++); // 修正済み
```

(51)　ビット演算子（”|”、”&”、”^”、”<<”、”>>”）の前後には空白文字を（☆☆☆）

ビット演算子の前後には空白文字を入れてください。コードの可読性が向上します。

＜違反サンプル＞

```
①int a = x >>y; // 違反
②int b = a& x; // 違反
③int c = x^y; // 違反
```

＜修正サンプル＞

```
①int a = x >> y; // 修正済み
②int b = a & x; // 修正済み
③int c = x ^ y; // 修正済み
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

(52) 論理演算子(”||”、”&&”)の前後には空白文字を(☆☆☆)

論理演算子の前後には空白文字を入れてください。コードの可読性が向上します。

＜違反サンプル＞

```
①if (a|| b){ // 違反
}
②if (a &&b){ // 違反
}
③if (a&&b){ // 違反
}
```

＜修正サンプル＞

```
①if (a || b){ // 修正済み
}
②if (a && b){ // 修正済み
}
③if (a && b){ // 修正済み
}
```

(53) 関係演算子(”<”、”>”、 ”>=”、 ”<=”、”==”、”!=”)の前後には空白文字を(☆☆☆)

関係演算子の前後には空白文字を入れてください。コードの可読性が向上します。

＜違反サンプル＞

```
①if (a<= b){ // 違反
}
②if (a ==b){ // 違反
}
③if (a!=b){ // 違反
}
```

＜修正サンプル＞

```
①if (a <= b){ // 修正済み
}
②if (a == b){ // 修正済み
}
③if (a != b){ // 修正済み
}
```

(54) 算術演算子(”+”、”-”、 ”*”、 ”/”、”%”)の前後には空白文字を(☆☆☆)

算術演算子の前後には空白文字を入れてください。コードの可読性が向上します。

＜違反サンプル＞

```
①if (a+ b){ // 違反
}
②if (a %b){ // 違反
}
③if (a/b){ // 違反
}
```

＜修正サンプル＞

```
①if (a + b){ // 修正済み
}
②if (a % b){ // 修正済み
}
③if (a / b){ // 修正済み
}
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

(55) boolean 変数は既に比較済み（☆☆☆）

boolean 変数は比較するまでもなく、それ自体が条件の結果を表していますので、これを true と比較する記述は冗長となり可読性が低下します。

＜違反サンプル＞	＜修正サンプル＞

①while(hasStock == true) { // 違反
}
②if(hasStock == true){ // 違反
}
	①while(hasStock) { // 修正済み
	}
	②if(hasStock){ // 修正済み
	}

(56) return 文ではカッコを使わない（★★）

retrun 文では、不要な括弧は使用しないようにしてください。
Java の仕様上、return できるオブジェクトはひとつだけです。
括弧を使用すると return 文を何らかのメソッドと見間違えてしまうなど、可読性の低下につながる可能性があります。
また、カッコの必要な演算がある場合は事前に演算を済ませましょう。

＜違反サンプル＞	＜修正サンプル＞

①return (a + b); // 違反
②return (int)(a); // 違反
	①return a + b; // 修正済み
	②return (int) a; // 修正済み

(57) 不等号の向きは左向き("<"、"<=")にする（★★）

不等号の向きを統一することで、コードが読みやすくなります。
特に意図することがある場合を除いて、向きは左向きに統一してください。

例外：
- 定数との比較時には、定数を常に右側におくことで可読性が良くなることがあります。
- 演算の中心となってコードに何度も現われる変数は、変数を常に左側に置くことで可読性が良くなることがあります。

＜違反サンプル＞	＜修正サンプル＞

if(a < i){
}else if(a > i){ // 違反
}else{
}
	if(a < i){
	}else if(i < a){ // 修正済み
	}else{
	}

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

11. コーディング標準（コメント）

(58)　/＊　～　＊/コメントは使用しない。（★★★）

Java で使用できるコメントは3種類あります。

この中で、/＊　～　＊/は、コメントの場所が複数行にまたがるため見失う恐れがあります。

（特にソースを一括で Grep したときなど、実装行とコメント行の判別ができません。）

Eclipse では、「Ctr + /」で一括して単一行(//)コメントに変更できますのでこれを使用するようにしてください。

① javadoc コメント

```
/**
 ＊ これは、javadoc コメントです。
 ＊ クラス、フィールド、メソッドに必ず付加します。
 ＊/
```

(2) 通常コメント（単一行）

```
// これは、単一行のコメントです。
```

(3) 通常コメント（複数行）

```
/*
 このコメントは、複数行のコメントです。　//　違反
 このコメントは、使用禁止です。
 ＊/
```

(59)　行末コメントはできるだけ使用しない。（☆☆）

行末のコメントは、ソースの可読性を下げ、後のメンテナンス性も損ねる恐れがあります。

コメントは、できるだけ該当ソースの前後の行に記述します。

＜違反サンプル＞

```
int code = 0;   // コードを定義します。
Int message = 0; // メッセージを定義します。

if (code = 1) { // コードが 1 ならば
}
```

＜修正サンプル＞

```
// コードを定義します。
int code = 0;
// メッセージを定義します。
Int message = 0;

// if の前の行に記述する
// コードが 1 ならば
if (code = 1) {
      // もしくは if の中に記述する
      // コードが 1 ならば
}
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

ただし、例外として、文字列結合などで同じメソッドが複数回連続する場合など、行末コメントを使用しないと可読性を損ねると判断できるときは行末コメントを利用します。

```
// ここは、行末コメントを利用することでソースの可読性に配慮します。
StringBuilder builder = new StringBuilder();
builder.append("SELECT AAA");           // 取得項目です。
builder.append("FROM TABLE");           // テーブル名です。
builder.append("WHERE AAA = '1'"); // 条件文です。
builder.append("AND BBB = '2'");     // 条件文です。
```

(60) フィールドコメントは必ず記述する。（☆☆☆）

フィールドのコメントは、必ず記述してください。
ただし、テーブルの列名ではない変数を定義する場合は、変数名から用途が判断できるように配慮してください。
フィールドコメントがなくても判断できる名前が最適です。

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

(61) クラスコメントは必ず記述する。(☆☆☆)

/** */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ます。
コメントは、html で処理されますので、改行
が必要ですが、@***などのタグは自動で改行されます。

このコメントの最初の 1 行は、クラスやメソッドの概要説明に使用されるため特別な意味合いがあります。
そのため、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します。
また、2 行目以降、説明が長くなる場合は、<pre>タグを使用します。
<pre>タグのコメントは、そのままの状態で表示されるので、インデントや改行コードは必要ありません。

サンプルにある、作成、改定履歴は本プロジェクトでは必須です。

(javadoc タグ: 抜粋)

@author [作成者]

クラスの作成者及び更新者を記述する。
複数記述する場合は 1 人ずつタグを記述する

@version [バージョン]

クラスのバージョンを記述する。

(javadoc xdoclet タグ) ※別途資料を参照

@struts.action

アクションに紐づくタグ情報を記述します。

@struts.action-forward

ActionForwad に紐づくタグを記述します。

<u>＜サンプル＞</u>

```
/**
 * ここに、機能の短い説明を記述します。
 * <pre>
 * ここに、機能の詳しい説明を記述します。
 * pre タグ内では、改行タグは必要ありません。
 * </pre>
 * <pre>
 * <b>作成:</b>
 * 2007/12/06   ○○○システム株式会社 △△△
 * Copyright (C) 2007 ○○○ SYSTEM Co.,LTD. All Rights Reserved.
 *
 * <b>改定履歴:</b>
 * <%-- 改定履歴 [START] ------------------------- --%>
 * 1.00   2007/12/06   新規作成
 * <%-- 改定履歴 [END]   ------------------------- --%>
 * </pre>
 * @author ○○○システム株式会社 △△△
 */
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

(62) メソッドコメントは必ず記述する。(☆☆☆)

メソッドコメントは通常 public のみ必須とされますが、メソッドの可視性(public、protected、デフォルト、private)にかかわらず、必ず記述してください。
param、return、exception タグは必須です。ただし、戻り値が void のときは return を記述しないなど、使用しないタグは、記述しないでください。

(javadoc タグ: 抜粋)

@param [名前] [説明]
　パラメータの名前とその説明を宣言されている順に記述する。
@return [説明]
　返り値があれば、返り値の説明を記述する。
@exception [名前] [説明]
　メソッドが呼ばれた際に throw される可能性のある例外名とその説明を記述する。
@see [説明]
@see [クラス、メソッド、変数]
　関連項目を記述します。

```
/*
 * ここに、機能の短い説明を記述します。
 * <pre>
 * ここに、機能の詳しい説明を記述します。
 * </pre>
 * @param args1   実行パラメータ
 * @return        戻り値
 * @throws Exception 例外
 * @see     jp.code4u.Sample#method
 */
```

(63) コメントは必要なものだけを簡潔に(★)

必要以上にコメントを書くと修正しにくくなります。
そのために「コメントは必要なものだけを簡潔に」記述します。
コメントを読みやすくするコツとして、下記のようなものが挙げられます。
(1) フィールドの宣言時、わかりやすく命名することによってコメントで説明する手間を省く。
(2) 修正者名、修正日等は書かない。
(3) (コメントがやむを得ず長くなる場合)最初の一文に要旨をまとめる。
ロジックを都度説明する必要はありません。
複雑な場合はコメントが必要ですが、複雑なロジックは意味を複数持つので複数のメソッドにわけ、メソッドの JavaDoc として説明する方がよいコードとなります。

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

12. コーディング標準（クラス）

(64) java.lang パッケージはインポートしない（☆☆☆）

java.lang パッケージは非明示的にインポートされます。
開発者はこのパッケージを明示的にインポートする必要はありません。

(65) import では，* を使用しない。（☆☆☆）

依存性を明確化、多くの*を使った import があると，読み手が苦労します。
Eclipse では、「Ctrl + Shift + O（インポートの編成）」を活用してください。

(66) 使用しない import は記述しない。（☆☆☆）

使用しない import を記述するとソースの可読性を下げることになります。
Eclipse では、「Ctrl + Shift + O（インポートの編成）」を活用してください。

(67) フィールドを宣言する順序は public（+）、protected（#）、デフォルト（~）、private（-）の順（☆☆☆）

コードの可読性を保つため、フィールドの宣言はアクセス修飾子によって順序だてるようにしてください。

＜違反サンプル＞

```
public class BadSample {
    int rank;
    private int age;
    public int number;    //違反
}
```

＜修正サンプル＞

```
public class BadSample {
    public int number;    // 修正済み
    int rank;
    private int age;
}
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

13. コーディング標準（メソッド）

(68) デフォルトコンストラクタは、原則作成する（☆☆）

デフォルトコンストラクタを明示的に用意することで、下記のことが可能になります。

- Class.newInstance()を使用して動的にインスタンスの生成を行う
- インスタンス変数のデフォルト値を決め、インスタンス生成時にその値で初期化を行う

例外として、引数ありのコンストラクタでインスタンスを生成することを前提としているクラスの場合、デフォルトコンストラクタを作成する必要はありません。

(69) static メソッドのみのユーティリティクラスは、コンストラクタを private にする。（☆☆☆）

static メソッドのみで構成されるクラスは、インスタンスを生成しません。
そのため、意図しない動作を抑制するためコンストラクタは private で宣言します。

(70) static メソッドのみのユーティリティクラスは、final でクラスを宣言する。（☆☆☆）

static メソッドのみで構成されるクラスは、インスタンスを生成しないため継承の必要がありません。
そのため、クラスを final で定義することでクラスの継承を抑制します。

(71) メソッドの引数は、最大 10 個とする。（☆☆☆）

１メソッドまたはコンストラクタのパラメータの数は最大 10 個までとしてください。
冗長な引数は、利用性、可読性を下げます。
可能であれば、引数をデータクラスにするなど設計の見直しをしてください。

(72) 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（☆☆☆）

クラスメソッド使用の際にはクラス名を用いるようにしてください。
これによって、コードがより明瞭になり、可読性が向上します。

＜違反サンプル＞

```
public class BadSample {
    public static void sampleClassMethod(){}
    public void sampleMethod(){
        BadSample object = new BadSample();
        object.sampleClassMethod();   // 違反
    }
}
```

＜修正サンプル＞

```
public class FixedSample {
    public static void sampleClassMethod(){}
    public void sampleMethod(){
        FixedSample object = new FixedSample();
        FixedSample.sampleClassMethod();   //修正済み
    }
}
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

(73) 1 行に 2 つ以上のステートメントを記述しない(★★★)

1 行には 2 つ以上のステートメントを記述してはいけません。可読性が低下してしまいます。
どんなに短いステートメントでも改行するようにしましょう。

<u>*＜違反サンプル＞*</u>

```
public int badSampleMethod(int top, int bottom) {
    int result = top - bottom; return result;   //違反
}
```

<u>*＜修正サンプル＞*</u>

```
public int badSampleMethod(int top, int bottom) {
    int result = top - bottom;
    return result;   // 修正済み
}
```

(74) サイズが 0 の配列を利用する(★)

戻り値が配列のメソッドで、null を返却するのは避けるべきです。
null が戻される場合、メソッドを使用するクライアントは余計な null チェックのロジックを書かなければなりません。
null ではなく長さゼロの配列を戻すようにしてください。

<u>*＜違反サンプル＞*</u>

```
import java.util.List;
public class BadSample{
    public String[] getStringList(List slist) {
        if (slist.size() == 0){
            return null; //  違反
        }else{
            String[] result =
                (String[])slist.toArray (new String[slist.size()];
            return result;
        }
    }
}
```

<u>*＜修正サンプル＞*</u>

```
import java.util.List;
public class FixedSample{
    public String[] getStringList(List slist){
        //修正済み
        String[] result =   (String[])slist.toArray (
                new String[slist.size()]);
        return result;
    }
}
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

(75) メソッドは 1 つの役割にする(★)

1つのメソッド内で複数の処理を行うことは推奨しません。
1つのメソッドで質の異なる複数の処理を行うと、可読性・保守性・拡張性・再利用性のすべてにおいて悪影響をもたらすことになります。
メソッドは機能ごとに分割してください。

<違反サンプル>

```
import java.awt.Point;
public class BadSample {
    public static void main(String[] args){
        Point point = new Point(55, 77);
        BadSample smpl = new BadSample();
        smpl.switchXandY(point);
    }
    // 違反 値の変更だけでなく表示も行っている
    private void switchXandY(Point point){
        StringBuffer sb = new StringBuffer();
        sb.append("X is ");
        sb.append(point.getX());
        sb.append(" and Y is ");
        sb.append(point.getY());
        System.out.println(sb);
        double x = point.getX();
        double y = point.getY();
        point.setLocation(y, x);
        sb.setLength(0);
        sb.append("X is ");
        sb.append(point.getX());
        sb.append(" and Y is ");
        sb.append(point.getY());
        System.out.println(sb);
    }
}
```

<修正サンプル>

```
import java.awt.Point;
public class FixedSample {
    public static void main(String[] args){
        Point point = new Point(55, 77);
        FixedSample smpl = new FixedSample();
        smpl.printValue(point);
        smpl.switchXandY(point);
        smpl.printValue(point);
    }
    // 修正済み X と Y を入れ替えるメソッド
    private void switchXandY(Point point){
        double x = point.getX();
        double y = point.getY();
        point.setLocation(y, x);
    }
    // 修正済み 値を表示するメソッド
    private void printValue(Point point){
        StringBuffer sb = new StringBuffer();
        sb.append("X is ");
        sb.append(point.getX());
        sb.append(" and Y is ");
        sb.append(point.getY());
        System.out.println(sb);
    }
}
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

(76) 引数の数が同じメソッドのオーバーロードは利用しない(★)

メソッドのオーバーロードにおいて、引数の数が同じになっていると、ソースコードを追いかけて引数の型を確かめなければ、どのメソッドが実行されているのかわかりにくいことがあります。
そのため、下の解説に記述したように、ソースコードの可読性が落ちてしまいます。
＜参考＞
オーバーロードされたメソッドの引数が継承関係にある場合、さらにユーザに思わぬ混乱を招くことがあります。
なるべく避けるようにしてください。また、そのようなメソッドを使う場合も細心の注意が必要です。

＜サンプル＞

下記のサンプルでは、オブジェクトの区別をする classify メソッドが 3 つ作成されています。
main メソッドで、インスタンスを生成し、classify メソッドを呼び出しています。

```
public class BadSample{
    public String classify(Object sampleObject){
        return "Unknown Object";
    }
    public String classify(String sampleString){
        return "String";
    }
    public String classify(Exception sampleException){
        return "Exception";
    }
    public static void main(String args[]){
        BadSample badSample = new BadSample();
        Object sample1 = new Object(); // ①
        String sample2 = new String(); // ②
        Exception sample3 = new Exception(); // ③
        // ～～ 様々な処理の実行 ～～
        System.out.println(badSample.classify(sample1)); // ⑪
        System.out.println(badSample.classify(sample2)); // ⑫
        System.out.println(badSample.classify(sample3)); // ⑬
    }
}
```

この結果、①、②、③に応じて

```
Unknown Object
String
Exception
```

と表示されます。
しかし、⑪、⑫、⑬の行だけを見ると、どのメソッドが実行されているかわかりません。
そのため、①、②、③の行に戻って、インスタンスの型を確認する必要があります。

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

(77) equals() メソッドを実装した場合は、hashCode()メソッドも実装する（★★）

Object.equals()をオーバーライドしたとき、Object.hashCode()も一緒にオーバーライドしてください。

java.lang.Object の仕様により、equals()メソッドにより等しいと判断されるオブジェクトは hashCode()によって返される値も等しくなければなりません。逆の場合も同じことが言えます。

この契約に違反した場合、HashMap、HashSet、Hashtable を含むすべてのハッシュに基づくコレクションにオブジェクトが使用された場合正しく機能しません。

equals()と hashCode()のどちらかをオーバーライドした場合は、必ずもう一方もオーバーライドしてください。

＜サンプル＞

以下の IDNumber クラスを使って equals()メソッドと hashCode()メソッドの実装例を見てみます。

以下の equals()メソッドは、id 属性の値が等しければ true を返すことになっています：

```
public final class IDNumber {
    private final int id;
    public IDNumber(int id){
        this.id = id;
    }
    public boolean equals(Object object){
        boolean isEqual = false;
        if (object == this) {
            isEqual = true;
        } else if (object instanceof IDNumber){
            IDNumber idNum = (IDNumber)object;
            if (idNum.id == this.id) { // id の値が等しければ true を返す
                isEqual = true;
            }
        }
        return isEqual;
    }
}
```

上記 IDNumber クラスはまだ equals()メソッドしかオーバーライドされていません。

この状態で、IDNumber クラスを HashMap で使用してみます。

```
Map map = new HashMap();
map.put( new IDNumber(123), "Hanako" );
```

一見、この HashMap に対して map.get(new IDNumber(123))を実行すれば、"Hanako"が取得できるように見えます。しかし、返ってくる値は null です。

デフォルトの hashCode()メソッドのままでは id の数値が同値でも、新たにインスタンスが作られればそのインスタンスに対して新たなハッシュコードを与えます。上記サンプルでも、HashMap の中身を get()する際に新しいインスタンスを生成していますので、id の値は同じでも別のハッシュコードが与えられ、異なる IDNumber インスタンスとして認識されたのです。

これを回避するためには hashCode()メソッドが思い通りの振る舞いをするようにオーバーライドします。

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

なお、hashCode()メソッドのオーバーライドは慎重に行ってください。同じインスタンスが同じ hashCode を返さなくてはいけません。下記は、今回のサンプルにおける hashCode()メソッドの例です。

```
public int hashCode() {
    int result = 13; //適当な素数
    result = 171 * result + id; // 基数として 171 を選んでいる
    return result;
}
```

ここで hashCode()メソッドの実装方法を紹介しましたが、他にも方法はたくさんあります。

このように、hashCode()をオーバーライドして正しい結果が返るように実装をすれば、ハッシュに基づくコレクションクラスを使用した際でも正しい結果を得ることができます。

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

14. コーディング標準（変数全般）

(78)　1 つのステートメントには 1 つの変数宣言（☆☆☆）

1 つのステートメントでは 1 つの変数のみ宣言するようにしてください。
これにより、コードの可読性を保つことが出来ます。
「ループ条件部でメソッドコールしない」に記述してありますが、for の宣言部では変数のスコープ、実行速度を優先するため、例外とします。

<u>＜違反サンプル＞</u>

```
public class BadSample {
    private String firstName, lastName;   //違反
}
```

<u>＜修正サンプル＞</u>

```
public class FixedSample {
    private String firstName;    //修正済み
    private String lastName;
}
```

for 文は変数のスコープ、および実行速度優先のため以下のようなサイズを取得する宣言を可とします。

```
List list = new ArrayList();
for (int i = 0, n = list.size(); i < n; i++) { //違反ではない
}
```

(79)　リテラル（マジックナンバー）を使用しない。（☆☆☆）

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します（数値、文字列両方含む）。
リテラルの使用は、コードの可読性・保守性を下げてしまいます。
これを防ぐために、リテラル定数（final static フィールド）を使用するようにしてください。
ただし、定数化することのみを目的とした意味のない定数の宣言は避けてください。

例外として、-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は使用可能です。

<u>＜違反サンプル＞</u>

```
public class BadSample {
    //違反
    private int[] sampleArray = new int[10];
    // 意味のない宣言は、可読性を下げます。
    private static final int TEN = 10
}
```

<u>＜修正サンプル＞</u>

```
public class FixedSample {
    //修正済み
    private int[] sampleArray = new int[ARRAY_SIZE];
    // 定数は用途がわかるように宣言します。
    private static final int ARRAY_SIZE = 10;
}
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

(80) 配列宣言は「型名[]」にする(☆☆☆)

下記の違反サンプルのような宣言形式は、C 言語の名残として残っているようですが、コードの一貫性を保つために、配列の宣言形式はコード間で統一してください。

<違反サンプル>

```
public class BadSample {
    private int sampleArray[] = new int[10];   //違反
}
```

<修正サンプル>

```
public class FixedSample {
    private int[] sampleArray = new int[10];   //修正済み
}
```

(81) 式内部での代入をしない。(☆☆☆)

式内部での代入とは、例えば String s = Integer.toString(i = 2); のようなことをさします。
このような記述は、変数が設定されているすべての箇所を判別し難くなってしまいます。
すべての代入はそのためだけのトップレベルのステートメントで行い、可読性を向上すべきです。

<違反サンプル>

```
public void sample(String str) {
    int i = 0;
    String s = String.valueOf(i = 2);     // 違反
}
```

<修正サンプル>

```
public void sample(String str) {
    int i = 0;
    i = 2;   // 修正済み
    String s = String.valueOf(i);   // 修正済み
}
```

(82) 定数は static final で宣言する(★★★)

クラス全体を通して変化しない値は定数です。
このような定数に対し static final を宣言することによって、この値が変わらないことを明示でき、コードの可読性が向上します。

<違反サンプル>

```
public class BadSample {
    private int constant = 5;     //違反
    private int getSize(int number) {
        return number * constant;
    }
}
```

<修正サンプル>

```
public class FixedSample {
    private static final int CONSTANT = 5;   //修正済み
    private int getSize(int number) {
        return number * CONSTANT;
    }
}
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

(83) できるだけローカル変数を利用する(★)

クラス変数やインスタンス変数へのアクセスは、ローカル変数へのアクセスに比べて時間がかかってしまいます。変数に頻繁にアクセスする時は、可能であればローカル変数を用いてください。

<違反サンプル>

```
public class BadSample {
    private int result;
    public void addNumber(int[] numbers) {
        for (int i = 0; i < numbers.length; i++){
            result += numbers[i]; //違反
        }
    }
}
```

<修正サンプル>

```
public class FixedSample {
    private int result;
    public void addNumber(int[] numbers) {
        //修正済み一時的なローカル変数
        int tempSum = result;
        for (int i = 0; i < numbers.length; i++){
        tempSum += numbers[i];
        }
        result = tempSum;
    }
}
```

(84) ローカル変数とインスタンス変数を使いわける(★)

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してはいけません。
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もあります。
インスタンス変数は必要性を充分に考慮してから使ってください。

<違反サンプル>

```
public class BadSample {
    // 違反 ひとつのメソッド内でしか使われない変数
    private int value;
    // value はこのメソッドでしか使われない
    public void calcValue(SomeObj inValue){
        // インスタンス変数である必要がない
        value = inValue.getData();
        for(int i = 0; i < value; i++){
            // 処理
        }
    }
}
```

<修正サンプル>

```
public class FixedSample {
    public void calcValue(SomeObj inValue){
        // 修正済みメソッド内で宣言
        int value = inValue.getData();
        for(int i = 0; i < value; i++){
            // 処理
        }
    }
}
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

15. コーディング標準（インスタンス変数）

(85) インスタンス変数は private にする（★★★）

インスタンス変数を public またはデフォルト（package-private）にし、直接操作するようなコードを書いてはいけません。
オブジェクト指向のカプセル化の考え方として、クラスの内部状態に誰でもアクセスできるということは好ましくありません。
適切な get/set メソッドを定義し、そのメソッドを介してのみインスタンス変数にアクセスできるようにしてください。

＜違反サンプル＞

```
public class BadSample {
   public int value = 10; //違反
}
public class BadSampleMain {
   public static void main(String[] args) {
      BadSample sample = new BadSample();
      sample.value += 10;   //直接アクセスしている
      System.out.println(sample.value);
   }
}
```

＜修正サンプル＞

```
public class FixedSample {
   private int value = 10; //修正済み
   public int getValue() {
      return value;
   }
}
public class FixedSampleMain {
   public static void main(String[] args) {
      FixedSample sample = new FixedSample();
      int result = sample.getValue();   //修正済み
      result += 10;
      System.out.println(result);
   }
}
```

(86) Action の関連するクラスでは、インスタンス変数を使用しない。（★★★）

Action、PopulateHandler、CheckHandler、StateHandler は、再利用を前提としたクラスです。
そのため、システム内で生成されるインスタンスは 1 つです。
これらのクラス内でインスタンス変数を使用すると複数のユーザがこのクラスを利用する場合、変数が共有されるため意図しない動作をする恐れがあります。
必要な変数は、メソッドの引数で渡すようにして下さい。

(87) オブジェクトどうしは equals()メソッドで比較する。（★★★）

オブジェクトの値の等値比較を”==”してはいけません。
”==”を使った場合、オブジェクトの値が同じかどうかを比較しているのではなく、インスタンスが同じかどうかを比較していることになります。
オブジェクトの値の等値比較は equals()メソッドで行ってください。
equals()メソッドのデフォルト実装はただの”==”比較ですが、大抵はそれぞれのオブジェクトの等値比較にふさわしい実装がオーバーライドされています。

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

例えば、String クラスの equals()メソッドは String に格納されている文字列値が等しいかどうかを比較するように実装されています。

＜違反サンプル＞

```
public class BadSample {
   public static void compare(String left, String right){
      if( left == right ){ // 違反
         System.out.println("They are equal.");
      } else {
         System.out.println("They are NOT equal.");
      }
   }
}
```

＜修正サンプル＞

```
public class FixedSample {
   public static void compare(String left, String right) {
      if( left.equals(right) ){   //修正済み
         System.out.println("They are equal.");
      } else {
         System.out.println("They are NOT equal.");
      }
   }
}
```

(88) static 変数を避ける(★★)

static 変数(クラス変数)は極力避けましょう。(static final 定数は除く)
static 変数は、セミグローバルと言って良い扱いになりますので文脈依存なコードを招き、副作用を覆いかくしてしまう恐れがあります。

(89) インスタンス変数初期化のタイミングは「コンストラクタ」または、「インスタンス変数宣言時」または「初めて値が get されるタイミング(Lazy Initialization)」のいずれかにせよ(★)

インスタンス変数の初期化のタイミングは、「コンストラクタ」「インスタンス変数宣言時」「初めて値が get されるタイミング」という 3 通りが考えられます。それぞれのメリット/デメリットは下記のとおりです。

【コンストラクタ】
◇メリット
・スレッドセーフなコードとなる
・インスタンスごとに異なる値での初期化が可能である
◇デメリット
・変数の宣言と初期化が離れたところで行われるので、初期化忘れの起こる可能性がある

【インスタンス変数宣言時】
◇メリット
・スレッドセーフなコードとなる
・インスタンス生成が早い
◇デメリット
・インスタンスごとに初期化できない

【初めて値が get されるタイミング】
◇メリット

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

・サブクラスを定義したとき、get メソッドのオーバーライドにより初期化操作もオーバーライドできる。
・値を get する際、get メソッドを呼び出す側での null チェックが必要ない（通常 get メソッド内でチェックするため）。
・初期化のタイミングをインスタンス生成時とずらした実装が実現できる。

◇デメリット

・複数のスレッドから同時に get メソッドが呼ばれる場合、不整合が起こりうる。
・get メソッドを呼ぶたびに初期化されているかどうかのチェックが行われ、負荷がかかる。

＜サンプル＞

【コンストラクタで初期化】

```
public class Sample1{
	private String name;
	public Sample1(){
		name = "Suzuki";
	}
	public String getName(){
		return name;
	}
}
```

【インスタンス変数宣言時に初期化】

```
public class Sample2{
	private String name = "Suzuki";
	public Sample2(){
	}
	public String getName(){
		return name;
	}
}
```

【初めて値が get されるタイミングで初期化】

```
public class Sample3{
	private String name;
	public Sample3(){
	}
	public String getName(){
		if(name == null){ //初期化されているかどうかチェック
		name = "Suzuki";
		}
		return name;
	}
}
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

16. コーディング標準（クラス変数）

(90) public static final 宣言した配列を利用しない。（★★★）

違反サンプルにあるとおり、final で配列を宣言していても、不変なのは配列のサイズのみであり、配列の要素は変更可能です。
保持している要素を変えられたくない場合は、Collections クラスの unmodifiableList()メソッド等を使用し、読み取り専用のコレクションを生成してください。

＜違反サンプル＞

```
public class BadSample {
    //違反
    public static final int[] SAMPLE_ARRAY = {0, 1, 2, 3};
    public static void main(String[] args){
        SAMPLE_ARRAY[0] = 1; //値を変えられる
    }
}
```

＜修正サンプル＞

```
import java.util.*;
public class FixedSample {
    //修正済み
    private static final int[] SAMPLE_ARRAY = {0, 1, 2, 3};
    public static final List IMMUTABLE_ARRAY =
        Collections.unmodifiableList(
            Arrays.asList(sampleArray));
}
```

(91) クラス変数にはクラス名を使用してアクセス（☆☆☆）

クラス変数使用の際にはクラス名を用いるようにしてください。
これにより、コードが意味的にもより明瞭になり、可読性が向上します。

＜違反サンプル＞

```
public class BadSample {
    public static final int STATIC_VALUE = 10;
    public void sampleMethod(){
        BadSample object = new BadSample();
        //違反
        int localValue = object.STATIC_VALUE;
    }
}
```

＜修正サンプル＞

```
public class FixedSample {
    public static final int STATIC_VALUE = 10;
    public void sampleMethod(){
        FixedSample object = new FixedSample();
        //修正済み
        int localValue = FixedSample.STATIC_VALUE;
    }
}
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

17. コーディング標準(ローカル変数)

(92) 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してはいけません。
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してください。
これにより、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ことができます。

<違反サンプル>

```
public class BadSample {
   public void method(int a) {
      int i;   //初期値なしの宣言
      for (i = 0; i < a; i++) {
         //i を使う
      }
      for (i = 0; i < a; i++) {
         //また i を使う
      }
      i = a * 2;
      //またまた i を使う
   }
}
```

<修正サンプル>

```
public class FixedSample {
   public void method(int a) {
      for (int i = 0; i < a; i++) {
         //i を使う
      }
      for (int i = 0; i < a; i++) {
         //別の i を使う
      }
      //別の意味ある変数
      int total = a * 2;
   }
}
```

(93) ローカル変数は利用する直前で宣言する(変数のスコープを意識する)(★★)

変数をソースの上のほうの広いスコープな場所に、そろえてまとめて、定義するやり方を良しとしてする考え方もありますが、ローカル変数を宣言する場所と使用する場所が離れている場合、コードの可読性・保守性を低下させる要因となってしまいます。
このようなことを防ぐため、ローカル変数は利用する直前で宣言するようにしてください。
変数のスコープは必要最小限に留めるように意識してください。

ただし、例外として、finally 句内で明示的に null を代入する場合、try の前にまとめて宣言をすることがあります。

<サンプル>

```
public void method() {
      Object obj1 = null;   // finally 句で null を代入する
      try {
            int sum = 0;   // この変数は、try の内で有効です。
            if (sum > 0) {
                  int index = 0;   // この変数は、この if 文の中でのみ有効です。
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

```
            }
            if (obj1 == null) {
                int index = 0;   // この変数は、この if 文の中でのみ有効です。
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            obj1 = null;   // 明示的な null の代入。
        }
    }
```

(94) ローカル変数は，初期値と共に宣言せよ。(★★)

インスタンス変数とは異なり、ローカル変数は初期値が自動的に設定されないため、明示的に初期化をする必要があります。

<違反サンプル>

```
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
}
```

<修正サンプル>

```
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
}
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

18. コーディング標準（制御構造）

(95) 三項演算子を利用しない。（☆☆☆）

三項演算子を利用すると、可読性が下がる恐れがあります。明示的に、if を使用しましょう。
例外として、JSP では if の記述が冗長となり可読性を下げる場合があります。その場合も、できるだけ <logic:equals>などのカスタムタグを利用することで回避してください。

＜違反サンプル＞

```
public class BadSample {
    public static void main(String[] args) {
        int x = 10;
        int y = 20;
        String s = x == y ? "同じ" : "違う";   // 違反
        System.out.println("s の値は " + s + " です。");
    }
}
```

＜修正サンプル＞

```
public class FixedSample {
    public static void main(String[] args) {
        int x = 10;
        int y = 20;
        String s = null;
        if (x == y) {
            s = "同じ";
        } else {
            s = "違う";
        }   // 修正済み
        System.out.println("s の値は " + s + " です。");
    }
}
```

(96) 制御文（if, else, while, for, do while）の“{ }”は省略しない。（☆☆☆）

たとえ処理が一行で完結していても”{}”は省略しないでください。
文内の処理がどこで完結しているかがわかりにくくなります。
新たに処理を加えた際の”{}”の付け忘れによるバグを防ぐためにも必ずつけるようにしましょう。

＜違反サンプル＞

```
public class BadSample{
    public void badSampleMethod(boolean flag){
        if (flag) // 違反
            System.out.println("if の中");
            System.out.println("if の外");
    }
}
```

＜修正サンプル＞

```
public class FixedSample{
    public void fixedSampleMethod(boolean flag){
        if (flag) { // 修正済み
            System.out.println("if の中");
            System.out.println("if の中");
        }
    }
}
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

(97) ステートメントが無い{}ブロックを利用しない（☆☆）

if 文や for 文にはステートメントを記述してください。開発途中でコードが書きかけになる場合などには、下記のような点を意識してください。

- 適切な処理を記述しておく
- まとめてコメントアウトしておく

(98) if/while の条件式で"="は利用しない（☆☆☆）

if あるいは while の条件内に代入演算子（"="）を使用するケースは、メソッドの返り値を代入するケースを除いて、ほとんど考えられません。
多くの場合、"=="と間違えているか、あるいは、条件文の外で行うべき代入を条件文内で行っています。
なお、"="で代入する変数が boolean 型であればコンパイルは通りますが、それ以外の型ではコンパイル時にエラーとして検出できます。

(99) switch 文では default を記述する（☆☆☆）

たとえ全てのありうるケースをカバーしていると開発者が確信していたとしても、 それは default の分岐で、例えばアサーションを使用して表現すべきです。
このようにして、コードを後の変更、例えば列挙型での新しい型の導入などから守ることができます。
また、default は他のすべての case の後に記述します。
Java では default が switch 文中のどこに現れても構いません。
しかし最後の case の次にあったほうが、可読性は高くなります。

(100) 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（☆☆☆）

for 文のカウンタは条件式以外で操作されるべきではありません。
制御構造がわかりにくいため間違いやすくなり、また、間違った場合に誤りを見つけ出すことが難しくなります。

＜違反サンプル＞

```
public class BadSample{
  int badSampleMethod(){
    int result = 0;
    for (int i = 0; i < 100; i++){
      i += 1; // 違反
      result += i;
    }
    return result;
  }
}
```

＜修正サンプル＞

```
public class FixedSample{
  int fixedSampleMethod(){
    int result = 0;
    // 修正済み
    for (int i = 0; i < 50; i++){
      result += (2 * i + 1);
    }
    return result;
  }
}
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

(101) 配列をコピーするときは System.arraycopy ()メソッドを利用する(★★)

System.arraycopy()は配列をコピーするためのメソッドです。
自分でコピー処理を書くのは無駄な上に、間違いを含んだり実行速度が劣ることがあるので、System.arraycopy()を使うようにしてください。※System.arraycopy の使用方法については Javadoc を参照してください

<違反サンプル>

```java
public class BadSample{
    String[] copy(String[] orgArray){
        if(orgArray == null){
            return null;
        }
        int length = orgArray.length;
        String[] copyArray = new String[length];
        for (int i = 0; i < length; i++){
            copyArray[i] = orgArray[i]; // 違反
        }
        return copyArray;
    }
}
```

<修正サンプル>

```java
public class FixedSample{
    String[] copy(String[] orgArray){
        if(orgArray == null){
            return null;
        }
        int length = orgArray.length;
        String[] copyArray = new String[length];
        // 修正済み
        System.arraycopy(
            orgArray, 0, copyArray, 0, length);
        return copyArray;
    }
}
```

(102) for 文のカウンタは 0 から始める(★★)

for 文内で使用するカウンタの初期値は、特に理由がなければ 0 としてください。
例えば、for 文内で配列の要素にカウンタを用いてアクセスする場合などに、i を 0 以外の値から始めてしまった場合、可読性が落ちてしまうことがあります。

<違反サンプル>

```java
public class BadSample{
    public static void main(String[] args){
        int[] testArray = new int[10];
        for (int i = 1; i <= 10; i++){     //違反
            testArray[i-1] = i;
        }
    }
}
```

<修正サンプル>

```java
public class FixedSample {
    public static void main(String[] args){
        int[] testArray = new int[10];
        for (int i = 0; i < 10; i++){     //修正済み
            testArray[i] = i + 1;
        }
    }
}
```

(103) ループ条件部でメソッドコールしない(★★)

ループ条件でのメソッド呼び出しを避けて実行速度を上げましょう。
java.util.List、java.util.Map の実装クラスのループをするときは、インデックスが必要な場合を除き、java.util.Iterator を使用することを推奨します。

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

サンプルでは for 宣言部で複数の宣言をどうじに行っていますが、for ループ内のみこれを可とします。
（リストのサイズを取得するメソッドを for ループ内に記述しているのは、変数のスコープが for ループ外に広がることになるのを避けるためです。）

＜違反サンプル＞

1 回のループにつき必ず strings.size() が呼ばれます。

```
List list = new ArrayList();
for (int i = 0; i < list.size(); i++) {
}
```

＜修正サンプル＞

値をキャッシュすることで不要な処理を省きましょう。

```
List list = new ArrayList();
for (int i = 0, n = list.size(); i < n; i++) {
}
```

なお、配列の長さを取得する際は算出処理がないので効果はありません。

```
String[] strings = new String[10];
for (int i = 0; i < strings.length; i++) {
}
```

(104) 論理演算子は、&& もしくは || を使用する。（★★）

論理積と論理和には以下に示すように 2 種類あります。

論理積（&、もしくは&&）：2 つの項がともに true のときに結果が true となります。

論理和（|、もしくは||）：2 つの項のどちらか一方が true のときに true となります。

&と&&（|と||）のちがいは、式の評価方法で、&を使うと、式の両辺が必ず評価されます。
&&の方を使うと、式の左辺が false と分かった時点で、false を返し、右辺は評価されません。
特別な意味がない限り、&&もしくは||を使用してください。

＜サンプル＞

```
public class Sample {
    public static void main(String[] args) {
        // ① || を使用した論理演算
        if (printA() || printB()) {
            printC();
        }
        // ② | を使用した論理演算
        if (printA() || printB()) {
            printC();
        }
    }
    public static boolean printA() {
        System.out.println("Aを実行しました。");
        return true;
    }
    public static boolean printB() {
        System.out.println("Bを実行しました。");
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

```
            return true;
        }
        public static boolean printC() {
            System.out.println("Cを実行しました。");
            return true;
        }
    }
```

----

①の実行結果

Aを実行しました。

Cを実行しました。

②の実行結果

Aを実行しました。

Bを実行しました。

Cを実行しました。

----

①と②の実行結果を比較してわかるとおり、|| の右側の式は評価されません。

(105) while ループより for ループを使用する(★)

開発者の好みにもよりますが、繰り返し回数が決まっているものは「for ループ」、条件を満たしている間繰り返すのは「while ループ」、と使い分けていることが多いと思います。

ここでは、変数のスコープを意識して for ループを活用することで不具合を減らすようにします。

<違反サンプル>

これは『while ループ』による繰り返しの例です。

イテレータのループ変数「i」の要素が存在している間、処理を繰り返します。

```
Iterator i = c.iterator;
while (i.hasNext()) {
    doHoge(i.next());
}
/* 以後、変数 i は使わない。 */
//: 別の処理
```

通常この場合、ループから抜けた後は変数「i」を使うことはありません。

(同じ変数を別のループで使用することは、ローカル変数の再利用禁止の規約違反となります。)

ところで、似たようなコードを書く時にコピー&ペーストをすることはよくあると思います。

下の例をご覧ください。

1 つ目のループのコードをコピー&ペーストして 2 つ目のループを書いています。

```
Iterator i = c.iterator;
while (i.hasNext()) {
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

```
        doHoge(i.next());
    }
    //: 別の処理
    Iterator i2 = c2.iterator;
    while (i.hasNext()) {   // i.hasNext() は誤り！
        doHoge(i2.next());
    }
```

コピー＆ペーストした後、2 つ目のループの条件文で正しい変数名に直すことを忘れたため間違いが起きています。
変数「i」を使っていることがミスであり実装として不具合なのですが、コンパイラレベルでは正常なので誤りに気づきません。

<u>＜修正サンプル＞</u>

ループ変数の内容がループ終了後に不要なのであれば、『while ループ』ではなく『for ループ』を使っても同じ処理が行えます。

```
    for (Iterator i = c.iterator; i.hasNext(); ) {
        doHoge(i.next());
    }
```

先ほどと同様の間違いがあった場合どうでしょう。

```
    for (Iterator i = c.iterator; i.hasNext(); ) {
        doHoge(i.next());
    }
    //: 別の処理
    for (Iterator i2 = c2.iterator; i.hasNext(); ) {   // ここでコンパイルエラー！
        doHoge(i2.next());
    }
```

コンパイルしようとすると、変数「i」が宣言されていないという理由でエラーが検知されます。
また、ループ変数「i」と「i2」は各々の『for ループ』の中だけでしか利用されていませんので、
「i」と「i2」は同じ名前で定義することができます。

```
    for (Iterator i = c.iterator; i.hasNext(); ) {
        doHoge(i.next());
    }
    //: 別の処理
    for (Iterator i = c2.iterator; i.hasNext(); ) {   //i2 を i に書き換える。
        doHoge(i.next());
    }
```

(106) break や continue は使わないほうがわかりやすい（★）

ループ内での処理の制御に break や continue を使用すると、制御構造が複雑化し可読性が低下してしまいます。
このようなロジックの大部分は、break や continue を使用せずとも記述できる場合がほとんどです。
制御構造を単純化し可読性を向上させるために、break や continue はできるだけ使用しないでください。

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

(107) 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(★)

switch 文や多くの if/else 文が繰り返されているコードは可読性が落ち、メンテナンスも大変です。
これらの問題は設計の問題でもありますが、ポリモーフィズム、FactoryMethod、Prototype、State パターンなどの利用を検討してください。
以下に一例としてポリモーフィズムでの変更例を示します:

<違反サンプル>

下記はポリモーフィズムを使わないサンプルです:

```
public class BadSample{
   public static void main(String[] args) {
      Object[]animals = new Object[2];
      animals [0] = new Dog();
      animals [1] = new Cat();
      int size = animals.length;
      for (int i = 0; i < size; i++){
         // if 文で分岐をしなくてはならない
         // チェックする型増えると if 文も増加
         if(animals[i] instanceof Dog){
            ((Dog)animals[i]).bark();
         }else if(animals[i] instanceof Cat){
            ((Cat)animals[i]).meow();
         }
      }
   }
}
public class Dog {
   // Dog クラス独自の鳴くメソッド
   public void bark(){
      System.out.println("わんわん");
   }
}
public class Cat extends Animal {
   // Cat クラス独自の鳴くメソッド
   public void meow(){
      System.out.println("にゃー");
   }
}
```

<修正サンプル>

左記のサンプルをポリモーフィズムを使って修正しました。コードがシンプルになったことがわかると思います:

```
public class FixedSample{
   public static void main(String[] args) {
      Animal[]animals = new Animal[2];
      animals [0] = new Dog();
      animals [1] = new Cat();
      int size = animals.length;
      for (int i = 0; i < size; i++){
         // インスタンスを判別する分岐が必要ない
         animals [i].cry();
      }
   }
}
public abstract class Animal{
   // 共通の鳴くメソッド
   abstract void cry();
}
public class Dog extends Animal{
   // Animal の抽象メソッドを実装
   public void cry(){
      System.out.println("わんわん");
   }
}
public class Cat extends Animal{
   // Animal の抽象メソッドを実装
   public void cry(){
      System.out.println("にゃー");
   }
}
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

(108) 繰り返し処理中のオブジェクトの生成は最小限にする(★)

繰り返し回数の多いループの内部でオブジェクトを生成することは、生成に時間がかかるため回数によっては、大きなパフォーマンス低下につながることがあります。
これを回避するには、ループ外でインスタンス作成しそれを再利用します。

<u><違反サンプル></u>

```
public class BadSample {
    public void badSampleMethod(String[] array){
        for (int i = 0; i < array.length; i++) {
            // 違反
            StringBuffer sampleBuffer = new StringBuffer();
            sampleBuffer.append("log: ");
            sampleBuffer.append(array[i]);
            System.out.println(sampleBuffer.toString());
        }
    }
}
```

以下の例は、再利用したつもりのプログラムバグです。
1 つのインスタンスを

```
    public static void main(String[] args) {
        List list = new ArrayList();
        Map map = new HashMap();
        for (int i = 0; i < 3; i++) {
            map.clear();
            map.put("index", String.valueOf(i));
            list.add(map);   // 参照を add している
        }
        for (int i = 0; i < 3; i++) {
            map = (Map) list.get(i);
            System.out.println(map.get("index"));
        }
    }
```

【実行結果】

```
2
2
2
```

<u><修正サンプル></u>

```
public class FixedSample {
    public void fixedSampleMethod(String[] array){
        // 修正済み
        StringBuffer sampleBuffer = new StringBuffer();
        for (int i = 0; i < array.length; i++) {
            sampleBuffer.append("log: ");
            sampleBuffer.append(array[i]);
            System.out.println(sampleBuffer.toString());
            // 初期状態に戻す
            sampleBuffer.setLength(0);
        }
    }
}
```

List へ add する必要があるときは、変数のスコープも考慮して(変数を再利用しない)、for ループ内でインスタンスを生成するのがよいでしょう。

```
    public static void main(String[] args) {
        List list = new ArrayList();
        for (int i = 0; i < 3; i++) {
            Map map = new HashMap();   // 修正済み
            map.put("index", String.valueOf(i));
            list.add(map);
        }
        for (int i = 0; i < 3; i++) {
            Map map = (Map) list.get(i);   // ちがう map
            System.out.println(map.get("index"));
        }
    }
```

【実行結果】

```
0
1
2
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

(109) 繰り返し処理の内部でtryブロックを利用しない(例外あり)（★）

ループの中に try/catch ブロックはできるだけ置かないでください。

パフォーマンスの低下につながり、ループ内の処理も大変見にくくなります。

特に理由がない場合はループの外で try/catch を行ってください。

【例外】

例外が発生しても、適宜例外処理をしてループ処理を最後まで実行したい場合。

<u>＜違反サンプル＞</u>

```
public class BadSample{
  public void method(String[] str){
    int size = str.length;
    // 違反ループの中に try/catch ブロック
    for(int i = 0; i < size; i++){
      try {
        int num =
        Integer.parseInt(str[i]);
        someOtherMethod(num);
      }catch(NumberFormatException e) {
        // 数値でない文字列の場合発生
        e.printStackTrace();
      }
    }
  }
  private void someOtherMethod(int i){
  // 処理
  }
}
```

<u>＜修正サンプル＞</u>

```
public class FixedSample{
  public void method(String[] str){
    int size = str.length;
    // 修正済みループの外に try/catch ブロック
    try {
      for(int i = 0; i < size; i++){
        int num =
        Integer.parseInt(str[i]);
        someOtherMethod(num);
      }
    } catch(NumberFormatException e) {
    // 数値でない文字列の場合発生
      e.printStackTrace();
    }
  }
  private void someOtherMethod(int i){
  // 処理
  }
}
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

19. コーディング標準（文字列操作）

(110) 文字列どうしが同じ値かを比較するときは、equals()メソッドを利用する。（☆☆☆）

文字列どうしの比較に"=="や"!="演算子を使用した場合、String が同じ文字列かどうかを比較するのではなく、同じインスタンスかどうかをチェックすることになり、バグの原因ともなり得ます。
したがって、同じ文字列かどうかを比較するには String クラスの equals()メソッドを使用してください。
リテラルの場合は、内容が同じであれば、コンパイラが同じインスタンスに最適化するため、"=="を用いても String クラスの equals()メソッドを結果は同じですが、コード上、この規約は守ってください。

＜違反サンプル＞

```
public class BadSample {
    public boolean compare(
            String name, String anotherName){
        return name == anotherName;     //違反
    }
}
```

＜修正サンプル＞

```
public class FixedSample {
    public boolean compare(
            String name, String anotherName){
        return name.equals(anotherName);   //修正済み
    }
}
```

(111) 文字列リテラルは new しない（★★★）

文字列定数を保持するための String オブジェクトを生成する際、new を使用すると、JavaVM によってはその都度インスタンスが生成されることになるので、余分なメモリ領域を消費してしまうことになります。
文字列定数には new を使用する必要はありません。

＜違反サンプル＞

```
public class BadSample {
    //違反
    private String sampleString =
                  new String (“Literal”);
        public void print() {
        System.out.println(sampleString);
    }
}
```

＜修正サンプル＞

```
public class FixedSample {
    //修正済み
    private String sampleString = “Literal”;
    public void print() {
        System.out.println(sampleString);
    }
}
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

(112) 更新される文字列には StringBuilder(StringBuffer) クラスを利用する(★★★)

String で定義した変更できない文字列(固定文字列)を”+=”演算子を用いて連結は極力避けてください。
String のオブジェクトは固定文字列であり、連結するとその都度オブジェクトを作り直すことになるため、パフォーマンスが低下してしまいます。
変更される文字列は StringBuilder(StringBuffer) で定義し、文字列連結の際には StringBuilder(StringBuffer)クラスの append()メソッドを用いてください。

＜違反サンプル＞

```
public class BadSample {
    public String getFruitName() {
        String fruit = “apples”;
        fruit += “, bananas”; //違反
        return fruit;
    }
}
```

＜修正サンプル＞

```
public class FixedSample {
    public String getFruitName() {
        StringBuffer fruit = new StringBuffer(15);
        fruit.append(“apples”); //修正済み
        fruit.append(“, bananas”); //修正済み
        return fruit.toString
    }
}
```

(113) StringBuilder(StringBuffer)クラスは初期容量を設定する。(★★)

StringBuilder(StringBuffer)や List クラスは初期容量を設定することでパフォーマンスの向上につながります。
初期容量が予め推測できるときは必ず設定しましょう。

＜違反サンプル＞

```
public class FixedSample {
    public String getFruitName() {
        //違反　初期値が設定されていない
        StringBuffer fruit = new StringBuffer();
        fruit.append(“apples”);
        fruit.append(“, bananas”);
        return fruit.toString
    }
}
```

＜修正サンプル＞

```
public class FixedSample {
    public String getFruitName() {
        //修正済み
        StringBuffer fruit = new StringBuffer(15);
        fruit.append(“apples”); //修正済み
        fruit.append(“, bananas”); //修正済み
        return fruit.toString
    }
}
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

(114) 更新されない文字列には String クラスを利用する(★★★)

変更されない文字列にもかかわらず、StringBuilder(StringBuffer)オブジェクトとして定義しないでください。
文字列定数に対しては String オブジェクトを使用し、メモリ操作がおこなわないようにするとパフォーマンスが向上します。

<違反サンプル>

```
public class BadSample {
    public void displayMessage() {
        //違反
        StringBuilder build = new StringBuilder("Message");
        String output = build.toString();
        System.out.println(output);
    }
}
```

<修正サンプル>

```
public class FixedSample {
    public void displayMessage () {
        //修正済み
        String output = "Message";
        System.out.println(output);
    }
}
```

(115) 文字列リテラルと変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(★★★)

文字列比較のために equals()メソッドを使用する場合、String オブジェクト.equals([文字列リテラル])と[文字列リテラル].equals(String オブジェクト)という 2 通りの記述が可能です。
ここで、前者では equals()メソッドを呼ぶ前に、文字列が null でないことを確認する必要がありますが、後者では null の確認は必要ありません。
なぜなら、[文字列リテラル]は null でないことがわかっているからです。
したがって、文字列比較には[文字列リテラル].equals(string)を用いるようにしてください。
これにより null チェックのコード漏れを防ぎ、コードの可読性・保守性を向上させることが出来ます。

<違反サンプル>

```
public class BadSample {
    public boolean validate(String userInput) {
        //違反
        return (userInput != null && userInput.equals("y");
    }
}
```

<修正サンプル>

```
public class FixedSample {
    public boolean validate(String userInput) {
    //修正済み
    return "y".equals(userInput);
    }
}
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

(116) 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（★★★）

String 型とプリミティブ型との変換には方法は種々ありますが、用意されている変換用のメソッドを使用するのが最もわかりやすく、処理も効率的です。

＜違反サンプル＞

```
// int -> String への変換
String sample = “” + integer;
String sample = (new Integer(integer)).toString();
// String -> int への変換
int integer = (new Integer(sample)).intValue();
int integer = Integer.valueOf(sample).intValue();
```

＜修正サンプル＞

```
// int -> String への変換
String sample = String.valueOf(integer);
// String -> int への変換
int integer = Integer.parseInt(sample);
```

(117) 文字列の中に、ある文字が含まれているか調べるには、charAt()メソッドを利用する（★★）

文字列の中に、ある文字が含まれているかを調べるとき、引数が 1 文字の場合は、String クラスの charAt()メソッドを用いてください。startsWith()は、複数文字列を引数の対象とするメソッドです。

引数が 1 文字の startsWith()メソッドを使っても動作しますが、これは String API の間違った使用法です。

注意: startsWith() を chartAt(0) に置き換える際は、まず文字列の長さが少なくとも 1 文字分はあることを確認してください。

＜違反サンプル＞

```
public class BadSample {
    public boolean checkHead(
    String checkString) {
        return (checkString.startsWith("E")   //違反
    }
}
```

＜修正サンプル＞

```
public class FixedSample {
    public boolean checkHead(
    String checkString) {
        return (checkString.length() > 0
            && checkString.charAt(0) == 'E')   //修正済み
    }
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

(118) システム依存記号（\n、\r など）は使用しない（★★）

OS が異なると、改行コードとして利用される文字や文字列も異なります。
そのため、「\n」や「\r」等を改行コードとしてコードにリテラルで利用してしまうと、可搬性が悪くなります（Java の特徴である『Write Once,Run Anywhere』という可搬性が失われます）。
改行コードをコードで利用したい時は、System.getProperty()を使用し、システムに応じた改行コードを取得し、利用するようにしてください。

<u>＜違反サンプル＞</u>

```
public class BadSample {
    public void displayMessage(String message) {
        //違反
        StringBuffer buf = new StringBuffer("Message:\n");
        buf.append(message);
        System.out.println(buf);
    }
}
```

<u>＜修正サンプル＞</u>

```
public class FixedSample {
    public void displayMessage(String message) {
        StringBuffer buf = new StringBuffer("Message:");
        //修正済み
        buf.append(System.getProperty("line.separator"));
        buf.append(message);
        System.out.println(buf);
    }
}
```

その他として、①println と print を使い分ける、

```
    StringWriter buffer = new StringWriter();
    PrintWriter pw = new PrintWriter(buffer);
    pw.println("１行目");   // 改行の挿入
    pw.println("２行目");   // 改行の挿入
    pw.print("３行目");
    String s = buffer.toString();
```

②クラスに準備されているメソッドを利用する、

```
    BufferedWriter bw
        = new BufferedWriter(new FileWriter(file));
    bw.write("こんにちは");
    bw.newLine();     // 改行の挿入
    bw.write("お元気ですか？");
    bw.newLine();     // 改行の挿入
    bw.close();
```

などの方法があります。

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

20. コーディング標準（数値）

(119) 低精度なプリミティブ型にキャストしない（★★★）

数値を表すプリミティブ型の変数には数値の精度が定義されています。
高い精度の変数を低い精度の型でキャストをおこなうとキャストした型の精度に値が変更されてしまいます。
そのため、計算によっては誤差が生じてしまう可能性があります。

＜サンプル＞

以下のプログラムでは、double 型の数値を int 型にキャストします：

```
public class CastTest{
    public static void main(String[] args){
        double doubleType = 2.75;
        int intType = (int)doubleType; // double を int にキャスト
        System.out.println("double : " + doubleType);
        System.out.println("int : " + intType);
    }
}
```

実行結果は以下のとおりです：

```
double : 2.75
int : 2
```

このことを充分に理解しないでキャストを行った場合、予想外の結果を生むことになります。

(120) 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、誤差が発生します。
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ます。
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ます。

＜サンプル＞

以下は double の演算誤差を示すプログラムです：

```
import java.math.BigDecimal;
public class BigDecimalTest {
    public static void main(String[] args) {
        double d = 0.0;
        BigDecimal b = new BigDecimal("0.0");
        for (int i = 0; i < 100; i++) { // それぞれに 0.1 を 100 回足す
            d += 0.1;
            b = b.add(new BigDecimal("0.1"));
        }
        System.out.println("double:" + d);
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

```
            System.out.println("BigDecimal:" + b.toString());
        }
}
```

このプログラムの実行結果は以下のとおりです：

double:9.999999999999998

BigDecimal:10.0

このように、場合によっては充分な配慮が必要です。

(121) 数値の比較は精度に気をつける(★)

前規約(『誤差の無い計算を必要とするときは BigDecimal クラスを使う』)において説明したように、浮動小数点には丸め誤差が生じます。

これを他の数値と比較する際、想定している結果が得られるかどうかの保証はありません。

＜違反サンプル＞

```
public class BadSample {
    void method(double value) {
        // 注意！
        // 事前の計算で value に丸め誤差があるかも…
        if (value == 100){
            System.out.println("value=100");
        }else{
            System.out.println("value!=100");
        }
    }
}
```

＜修正サンプル＞

丸め誤差が生じることによって問題が起こるようなアプリケーションの場合は double や float のかわりに BigDecimal を使うか、Double や Float などの浮動小数点のラッパークラスにある equals()メソッドを使っての比較などを考慮してみてください。

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

21. コーディング標準（日付）

(122) 日付を表す配列には、long の配列を利用する（★）

Date 型のインスタンスは他のインスタンスと比較して、リソースを多く消費するため、単純に日付だけを扱いたい場合はこれらが無駄にリソースを消費することとなります。
パフォーマンスを重視する場合は Date 型の配列の変わりに long 型の配列を使用するようにしてください。

＜違反サンプル＞

```
import java.util.*;
public class BadSample {
    public static void main(String[] args){
        int length = 3;
        Date[] dates = new Date[length];
        for(int i = 0; i < length; i++){
            Calendar now =
            Calendar.getInstance();
            dates[i] = now.getTime();
            long time = dates[i].getTime();
            System.out.println(time);
        }
    }
}
```

【実行結果】
1078115074691
1078115074701
1078115074701

【解説】
現在の時刻は 1970 年 1 月 1 日 00:00:00 GMT からのミリ秒数で表すことができます。Date クラス getTime()メソッドを使えばその数値を取得できます。

＜修正サンプル＞

```
public class FixedSample {
    public static void main(String[] args){
        int length = 3;
        long[] times = new long[length];
        for(int i = 0; i < length; i ++){
            times[i] =
            System.currentTimeMillis();
            System.out.println(times[i]);
        }
    }
}
```

【実行結果】（左記サンプルと同時刻に実行された場合）
1078115074691
1078115074701
1078115074701

【解説】
System.getCurrentTimeMillis()によって左記と同じ値を得ることができます。
long の値「time」を Date オブジェクトの値にしたいときは、下記のようにします：

```
Calendar cal = Calendar.getInstance();
cal.setTimeInMillis(time);
Date date = cal.getTime();
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

22. コーディング標準（コレクション）

(123) Java2 以降のコレクションクラスを好め（★★★）

Java2 では Vector クラス、Hashtable クラス、Enumeration クラスに変わるクラスとして、List(ArrayList クラス)、Map(HashMap クラス)、Iterator が準備されています。
List などのインタ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す。
また、インタ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ます。

(124) List クラスは初期容量を設定する。（★★）

StringBuilder（StringBuffer）や List クラスは初期容量を設定することでパフォーマンスの向上につながります。
初期容量が予め推測できるときは必ず設定しましょう。

＜違反サンプル＞

```
public class BadSample {
    private List = new ArrayList();   //違反
}
```

＜修正サンプル＞

```
public class BadSample {
    // 修正済み
    private static final int RECORD_SIZE = 10;
    private List = new ArrayList(RECORD_SIZE);
}
```

(125) 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（★）

コレクションにある特定の型のオブジェクトだけを格納する場合、特定の型のオブジェクトのみを扱うコレクションクラスを新しく定義する方法があります。
例えば、String しか格納できないようなコレクションを自分で定義するのです。
このようなコレクションクラスを定義する利点は、

- あやまって異なる型のオブジェクトを格納してしまうことがないので、ClassCastException が発生しない
- コレクションからオブジェクトを取り出す際、いちいちキャストを行わなくてもよい（instanceof のチェックロジックも必要ない）

上記のことから、バグの可能性を減らすことができ、コードもシンプルになり可読性を向上させることができます。

＜サンプル＞

以下のクラスは String オブジェクトのみを扱う HashMap のサンプルです:

```
public class StringHashMap {
    private HashMap map = new HashMap();
    public void put( String key, String value ){
        map.put( key, value );
    }
    public String get( String key ){
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

```
        return (String)map.get(key);
    }
    …
}
```

これはごく単純なサンプルですが、スーパークラスとして汎用的なメソッドを持つクラスを宣言しそれを拡張するなど、この他にも色々な方法が考えられます。個々の用途に合う、拡張の容易な設計を考えてみてください。
（JDK 標準パッケージの Collection インターフェースに対する実装は、引数や戻り値が結局 Object 型になってしまうので、このように新しく定義する必要があります。）
#JDK1.5.0 より Generic Types が利用できますので、活用してください。

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

23. コーディング標準（ストリーム）

(126) ストリームを扱う API を利用するときは、finally ブロックで後処理をする（★★★）

メモリリークを回避するためのストリームのクローズは必須です。
途中で例外が発生した場合でも必ずクローズされるように、close()メソッドは finally ブロックに記述してください。

＜違反サンプル＞

下記のサンプルコードでは例外が発生した場合にストリームがクローズされません。

```
public class BadSample {
    public void badSampleMethod(File file) {
        try {
            BufferedReader reader =
                new BufferedReader(new FileReader(file));
            reader.read();
            reader.close();   //違反
        } catch (FileNotFoundException fnfe) {
            fnfe.printStackTrace();
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
    }
}
```

＜修正サンプル＞

finally ブロックに close()メソッドを記述すれば、例外が発生した場合にも確実にクローズされます。

```
public class FixedSample {
    public void fixedSampleMethod(File file) {
        BufferedReader reader = null;
        try {
            reader =
                new BufferedReader(new FileReader(file));
            reader.read();
        } catch (FileNotFoundException fnfe) {
            fnfe.printStackTrace();
        } catch (IOException ioe1) {
            ioe1.printStackTrace();
        } finally {
            try{
                reader.close(); // 修正済み
            }catch(IOException ioe2){
                ioe2.printStackTrace();
            }
        }
    }
}
```

(127) ObjectOutputStream では reset()を利用する（★★）

ObjectOutputStream オブジェクトを使うメソッドでは適宜 reset()を呼び出してください。
ObjectOutputStream クラスはその機能上、reset()が呼び出されるまで、書き込まれるすべてのオブジェクトの参照を持ちつづけます。
reset()メソッドが呼び出されるまで、これらのオブジェクトは GC によって回収されることはありません。
reset()メソッドを呼ばずに ObjectOutputStream オブジェクトによって多くのオブジェクトを書き出した場合、OutOfMemoryError が発生する可能性があります。

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

例外：同じオブジェクトを繰り返しストリームに書き込む場合は、アクセス速度を高速にするために、送信されるオブジェクトがストリームによってキャッシュされる機能を利用するため、reset()メソッドを使わないでください。

<u>＜違反サンプル＞</u>

```
public class BadSample {
  public void writeToStream(Object input)
      throws IOException {
    ObjectOutputStream stream =
      new ObjectOutputStream(
          new FileOutputStream("output"));
    // 違反 ストリームを reset()していない
    stream.writeObject(input);
  }
}
```

<u>＜修正サンプル＞</u>

```
public class FixedSample {
  public void writeToStream(Object input)
      throws IOException {
    ObjectOutputStream stream =
      new ObjectOutputStream(
          new FileOutputStream("output"));
    stream.writeObject(input);
    stream.reset(); // 修正済み
  }
}
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

24. コーディング標準（例外）

(128) catch 文で受け取る例外は、詳細な例外クラスで受け取る（★★★）

面倒だからといって catch ステートメントで Exception、RuntimeException、Throwable などをキャッチしてはいけません。これらのスーパークラスで例外を受け取り、例外処理をおこなう場合、catch ブロック内で例外の種類に応じた適切な処理ができません。

また、本来呼び出し元にスローされるべき例外が誤って受け取られる可能性もあります。

例外を処理するときには常に Exception、RuntimeException、Throwable のサブクラスをキャッチしてください。

例外：フレームワーク等、アーキテクチャ上の例外の最終処理においてこれらを使用する場合は問題ありません。

＜違反サンプル＞

```
import java.io.*;
public class BadSample{
  public void badSampleMethod(File file) {
    BufferedReader reader = null;
    try {
      reader =
        new BufferedReader(new FileReader(file));
      reader.read();
    } catch (Exception e) {
      // 違反
      e.printStackTrace();
    } finally {
      try{
        reader.close();
      }catch(Exception e2){
        // 違反
        e2.printStackTrace();
      }
    }
  }
}
```

＜修正サンプル＞

```
import java.io.*;
public class FixedSample{
  public void fixedSampleMethod(File file) {
    BufferedReader reader = null;
    try {
      reader =
        new BufferedReader(new FileReader(file));
      reader.read();
    } catch (FileNotFoundException fnfe) {
      //修正済み
      fnfe.printStackTrace();
    } catch (IOException ioe1) {
      //修正済み
      ioe1.printStackTrace();
    } finally {
      try{
        reader.close();
      }catch(IOException ioe2){
        //修正済み
        ioe2.printStackTrace();
      }
    }
  }
}
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

(129) Exception クラスのオブジェクトを生成してスローしない（★★★）

Exception クラスのオブジェクトを生成して、例外をスローしてはいけません。

Exception クラスはすべての例外のスーパークラスです。Exception クラスのインスタンスを例外として受け取ったコードでは、その例外の種類や処理に対してポリモーフィズムが利用できなくなります。

必ず適切なサブクラスのオブジェクトをスローしてください。

＜違反サンプル＞

```
public class BadSample {
   public void badSampleMethod()
         throws Exception { // 違反
      throw new Exception(); // 違反
   }
}
```

＜修正サンプル＞

```
public class FixedSample {
   public void fixedSampleMethod ()
         throws NoSuchMethodException {  //修正済み
      throw new NoSuchMethodException (); //修正済み
   }
}
```

(130) catch ブロックでは必ず処理をする（☆☆☆）

catch ブロックを空にしないで必ず例外に対する処理を行ってください。

例外は処理されるためにキャッチするものです。何もしないと新たなバグにもつながります。

処理ができない場合でもログへ出力するなど、例外が発生したことを確認できるようにしてください。

例外：意図的に処理を記述しない catch ブロックを使用したい場合はこの限りではありません。

＜違反サンプル＞

```
import java.io.*;
public class BadSample {
   public void badSampleMethod(File file) {
      BufferedReader reader = null;
      try {
         reader =
            new BufferedReader(new FileReader(file));
         reader.read();
      } catch (FileNotFoundException fnfe) {
         //違反
      } catch (IOException ioe1) {
         //違反
      } finally {
         try{
            reader.close();
         }catch(IOException ioe2){
            //違反
         }
      }
```

＜修正サンプル＞

```
import java.io.*;
public class FixedSample {
   public void fixedSampleMethod(File file) {
      BufferedReader reader = null;
      try {
         reader =
            new BufferedReader(new FileReader(file));
         reader.read();
      } catch (FileNotFoundException fnfe) {
         fnfe.printStackTrace(); //修正済み
      } catch (IOException ioe1) {
         ioe1.printStackTrace(); //修正済み
      } finally {
         try{
            reader.close();
         }catch(IOException ioe2){
            ioe2.printStackTrace(); //修正済み
         }
      }
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

(131) Exception クラスの printStackTrace()を使用しない。(★★★)

Exception クラスの printStackTrace メソッドをログ出力クラスで管理できないため、エラーメールが送信される(もしくは、されない)などフレームワークとして予期しない動作をする恐れがあります。
必ず、上位クラスに throw をするか、ログ出力クラスを使用してください。。

(132) finally ブロックの中で return を記述しない(★★)

finally ブロックに記述された内容は、その前で例外がキャッチされていても、またはキャッチされていなくても必ず実行されます。finally ブロックには、ストリームのクローズ処理など、必ず行うべき処理を記述してください。
finally ブロックの中で return 文を記述したり、メソッドの返り値に影響がある記述をしてはいけません。

(133) Error、Throwable クラスを継承しない(★★)

Error クラスのサブクラスを定義しないでください。
Error クラスは、アプリケーションではキャッチされるべきではない重大な問題を表しています。
そのため、アプリケーションで Error クラスのサブクラスを定義してはいけません。
アプリケーションで定義する例外は Exception クラスを継承します。
また、Throwable のサブクラスを定義してはいけません。
Throwable クラスは Exception と Error のスーパークラスです。
このクラスからアプリケーションの例外を継承すると、エラーと例外の意味があいまいになります。
Error クラスは上で書いたようにアプリケーションで継承すべきではありません。
同じように、アプリケーションは Thorwable クラスを継承すべきではありません。
アプリケーションで定義する例外は Exception クラスを継承します。

(134) 例外クラスは無駄に定義しない(★★)

できる限り JDK 標準パッケージに含まれる例外を利用してください。
JDK 標準パッケージに含まれているものでは表現できないようなアプリケーション独自の例外が考えられる場合のみ新たに例外を定義してください。
また、新たに例外を定義する場合は特定のメソッドだけで利用される例外を定義するのではなく、例外の必要性を検討した上で、アプリケーション全体で利用できる形で定義します。

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

25. コーディング標準（スレッド）

(135) スレッドは原則 Runnable を実装（★★）

スレッドの実現方法は以下の二つがあります：

① Thread のサブクラスをつくる

② Runnable インターフェースを実装する

スレッドを実現する際は、原則として②の Runnable インターフェースを実装する方法をとってください。

例えば、ラジコンカーを動かすプログラムがあるとします。

コントローラオブジェクトと車オブジェクトは別々のスレッド上で動作しますが、これらのオブジェクトはあくまでもコントローラとしての機能を持つ Controller クラスと、ラジコンカーとしての車の機能を持つ Car クラスとして宣言されるべきです。

スレッドとしての機能を持つかもしれませんが、Thread クラスのサブクラスとして宣言されるべきではありません。

Java は多重継承をサポートしていないため、Thread クラスを継承すると他のクラスは継承できません。

特に run()メソッド以外の Thread クラスの基本的な機能を拡張したい、というとき以外は Thread クラスをサブクラス化しない方がいいでしょう。

(136) ウェイト中のスレッドを再開するときは notifyAll()メソッドを利用する（★★）

ウェイト中のスレッドを活性化するときは、notify()メソッドより notifyAll()メソッドを使うようにしてください。

あるオブジェクトに対して wait()メソッドが呼ばれると、現在のスレッドは一度処理を停止してウェイトセットに入ります。

ウェイトセットからスレッドを再開させるには notify()メソッドまたは notifyAll()メソッドを呼ぶ方法があります。

notify()メソッドは同一のオブジェクト上で複数のスレッドがウェイト中の場合、複数あるスレッドのうちのひとつだけを再開させますが、どのスレッドが再開するのか予測できません。

従って再開されるべきスレッドがその先もずっと再開されない可能性があります。

これを避けるため、下記の前提がない場合は、待ち状態のすべてのスレッドを再開させる notifyAll()メソッドを使ってください。

【notify()メソッドでもよい場合】

① 待ち状態のスレッドがひとつだけの場合

② 待機中のスレッドがすべて同じ条件を持っている場合（どのスレッドが呼ばれてもかまわない場合）

(137) Thread クラスの yield()メソッドは利用しない（★★）

Thread.yield()メソッドは JavaVM によって制御方法が異なるため、アプリケーションの動作は保証されません。

Thread.yield()メソッドはあくまでもスレッドの切り替えを促すだけです。

Thread.yield()メソッドを呼んだ場合、スレッド A が制御を放棄した後、どのスレッドに制御が移るか保証されません。

また他のスレッドに制御が移るかどうかも保証の限りではありません。

その後の動作が保証されるべき処理には Thread.yield()を使わないような設計を行ってください。

yield()メソッドではスレッドが持っているロックを解放しません。

ロックをもった状態で yield()メソッドを呼び出すと予期せぬ動作をすることがあります。

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

ロックをもった状態でスレッドの切り替えを意図する場合は、ロックしているオブジェクトの wait()メソッドを利用してください。

(138) synchronized ブロックのあるメソッドを呼び出さない（★★）

synchronized ブロックの中から synchronized ブロックのあるメソッド(synchronized メソッドを含む)を呼び出していませんか？それらのデッドロック発生の可能性を考慮しましたか？
デッドロックは以下の条件が満たされたときに発生する可能性があります：
① 複数のロック対象がある
② ある対象のロックをとったまま他の対象のロックをとりにいく
③ それらの対象へのロック順序が決まっていない
synchronized ブロックから synchronized ブロックのあるメソッドを呼び出すという処理は、あるオブジェクトのロックをとった上で更に別のオブジェクトのロックをとりにいくため、①と②の条件を既に満たしています。
もし、これに加えて対象へのロックの順序が決まっていなければ、容易にデッドロックが発生し得ることになります。
ロック対象のリソースのロック順序が厳密に定められていない限り synchronized ブロックから synchronized ブロックのあるメソッドは呼び出さないでください。

（解説）
デッドロックとは、ふたつのスレッド同士が、相手方がロックをとっているオブジェクトの開放を待ち合ってしまう状態を言います。例えば、ふたりの板前がいたとします。それぞれ仕事をこなすには包丁とまな板の両方が必要ですが、包丁とまな板は1つずつしかありません。板前 A はまず包丁を取り、板前 B はまな板を取りましたがお互い包丁とまな板の両方がなければ仕事ができません。
従って、A は B がまな板を手放すのを、B は A が包丁を手放すのをお互い頑固に待ち、いつまでたっても仕事ができなくなってしまう…というような状況です。
以下では、スレッド A（板前 A）とスレッド B（板前 B）が X（包丁）と Y（まな板）というリソースの開放を互いに待ち合っているイメージを図式化しました：

この問題は、ロック対象へのアクセス順序を定めておけば解決できます。例えば、包丁とまな板のうち「必ず包丁を先にとらなければならない」というルールを定めれば、板前 A が先に包丁を取った場合、板前 B は板前 A が仕事を終えて包丁を手放すまで待たなければなりませんが、板前Aが仕事を終えた後に板前Bは包丁をとることができますので、デッドロックは発生しません。

(139) wait()、notify()、notifyAll()メソッドは、synchronized ブロック内から利用する（★★）

wait()メソッド、notify()メソッド、notifyAll()メソッドはオブジェクトのロックを取った状態でないと呼び出すことができません。
対象オブジェクトのロックを獲得せずに呼び出すと、java.lang.IllegalMonitorStateException が発生します。

(140) wait()メソッドは while ブロック内から利用する（★★）

wait()メソッドは必ず while ループ内部で利用してください。
wait()が呼ばれるのは、多くの場合なんらかの条件をスレッドに持たせる場合です。
while ブロック以外で wait()メソッドを呼び出すと、条件の前提が壊れてしまう可能性があります。

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

<違反サンプル>

```
synchronized(car){
    if (fuel == null){ // 違反
        try{
            car.wait();
        }catch(InterruptedException e){
            e.printStackTrace();
        }
    }
    car.drive(); // 処理（走る）
}
```

car オブジェクト(車)は fuel(燃料)がなければ動くことができません。

従って、fuel フィールドが null の場合スレッドは wait()するようになっています。

この処理は一見正常に動くように思われます。

しかし、以下のようなことが起こった場合はどうでしょうか:

① 故意に、あるいは誤って notifyAll()が呼び出され、fuel がまだ null なのにも関わらず、スレッドが再開してしまった

② この処理とは関係のない場所で notifyAll()が呼ばれ、スレッドが再開してしまった

③ fuel に燃料が補充され、notifyAll()が呼ばれたが、他のスレッドが先に car のロックを獲得して fuel の状態を変えてしまった

④ なんらかの VM の異常により、スレッドが再開してしまった

このようなことが起こった場合、fuel が null でないときに car.drive()が正常に呼ばれる(車が走る)、という保証はありません。

<修正サンプル>

```
synchronized(car){
    while (fuel == null){ // 修正済み
        try{
            car.wait();
        }catch(InterruptedException e){
            e.printStackTrace();
        }
    }
    car.drive(); // 処理（走る）
}
```

左記違反サンプルを修正するには、if を while に変更するだけです。

if 文では一度文内の処理が実行されたあとは if 文から抜け次の操作に移ってしまいますが、while 文は、文内の処理が実行されたあと再度条件を評価します。

再度評価した結果が true であれば、再び文内の処理を行いますし、false で
あれば while 文を抜けて次の操作へと進みます:

従って、fuel がもし null のままスレッドが再開してしまっても、再度条件の評価が行われ fuel が null ならば再びスレ

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

ッドは wait()するので、car.drive()が呼ばれるときには fuel は null でないことが保証されます。

(141) ポーリングを利用せずに wait()、notifyAll()メソッドによる待ち合わせを利用する（★★）

ポーリングループはパフォーマンスを低下される原因ともなるので wait()メソッドと notifyAll()メソッドを利用した Thread の同期機能を利用して処理をしてください。
ポーリングループとは、ある条件が満たされたときにだけ処理を行うという場合に、条件が満たされたかどうかを自身で問い合わせにいくループのことを言います（逆の場合は、条件が満たされたことを相手側から通知されることによって処理に移ります）。
ポーリングループにより頻繁に問い合わせした場合はそれだけプロセッササイクルを多く消費します。
また、一定時間置いて随時問い合わせした場合でも、一度目の問い合わせの直後に条件が満たされた場合でも次の問い合わせが行われるまでは処理に移れませんので、プロセスの占有時間が長くなってしまいます。
一方、条件が満たされなかった場合に wait()メソッドが呼ばれ、条件が整ったら notifyAll()メソッドが呼ばれてスレッドが再開されるようにしておけば、無駄な問い合わせをする必要がなく大変スマートになります。

＜違反サンプル＞

```
public class Robot implements Runnable{
    private static byte[] commands;
    private RobotController controller;
    // …
    public void run(){
        synchronized(controller){
            while (commands == null){
                try{
                    Thread.sleep(10000); // 違反
                }catch(InterruptedException e){
                    e.printStackTrace();
                }
            }
            int size = commands.length;
            for(int i = 0; i < size; i++){
                this.processCommand(commands[i]);
            }
            commands = null;
        }
    }
    public void storeCommands(byte[]commands){
        this.commands = commands;
    }
    private void processCommand(byte command){
        // …
    }
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

```
}
public class RobotController{
}
```

ここでの while 文は、以下の処理が行われると次の処理へ移る条件が満たされます:

```
Robot.storeCommands(commands);
```

しかし、この処理が行われるまでは 10000 ミリ秒(10 秒)毎に while の条件の判定が行われ、その都度 sleep()をすることになります。また、条件が満たされても、sleep()が終わるまで待たなければ次の処理へ進めません。

<修正サンプル>

```
public class Robot implements Runnable{
    private static byte[] commands;
    private RobotController controller;
    // …
    public void run(){
        synchronized(controller){
            while (commands == null){
                try{
                    // 修正済み
                    controller.wait();
                }catch(InterruptedException e){
                    e.printStackTrace();
                }
            }
            int size = commands.length;
            for(int i = 0; i < size; i++){
            processCommand(commands[i]);
        }
        commands = null;
        }
    }
    public void storeCommands(byte[] commands){
        this.commands = commands;
    }
    private void processCommand(byte command){
    // …
    }
}
public class RobotController{
}
```

こちらのサンプルは、sleep()を行っていた部分を wait()に修正しただけです。

ここでは while 文の条件が満たされていなかった場合 wait()します。

そして、下記のように条件を満たしたあとに notifyAll()を呼んでやれば、wait()していた controller が再開されます:

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

```
Robot.storeCommands(commands);
controller.notifyAll();
```

(142) 同期化(synchronized)の適用は必要な部分だけにする(★★)

同期化を行うには二通りの方法があります。
メソッドを synchronized 宣言する方法と、synchronized ブロックを利用する方法です。

【①synchronized メソッド】

```
synchronized void method() {
}
```

【②synchronized ブロック】

```
void method() {
    synchronized(<インスタンス>) {
}
```

②の synchronized ブロックの<インスタンス>部分にはロックをとるインスタンスを記述します。
では、①の synchronized メソッドは何に対してロックをとっているのでしょうか。
①のメソッドは以下の記述と同等の意味になります:

```
void method() {
    synchronized (this) {
        …
    }
}
```

ご覧のように、synchronized メソッドは this に対してロックをとっています(static synchronized メソッドの場合はそのクラスの java.lang.Class インスタンスがロックされます)。
ロックされているオブジェクトは他から synchronized アクセスすることはできませんので、this に対してロックをした場合、ロックがとられている間(メソッドが実行されている間)は同じオブジェクトの synchronized メソッドを呼び出すことができません。
これは、スレッドセーフではあるかもしれませんが、そのメソッドとは同期をとらなくてもよいメソッドまでもがロックが解除されるまで待たされてしまうかもしれません。
this に対するロックは広範囲に渡って影響を及ぼすことがあることを念頭に置いてください。
従って、まずは synchronized ブロックで部分的なロックが実現できるかどうかを考慮し、メソッド全体に同期が必要と判断された場合のみメソッドを synchronized 宣言するようにしてください。
synchronized ブロックを利用する場合も、同期化の適用範囲は必要最小限に「同期化しなければならない部分だけ同期化する」ことが原則です。
synchronized の適用範囲を最小限にとどめることによって、以下のリスクを軽減することができます:

- デッドロック発生の可能性
- パフォーマンスの低下
- デッドロック発生時のデバッグのコスト(可読性の低下)

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

＜違反サンプル＞

```
public class BadSample {
  private String name;
  private long birthday;
 public synchronized void setName(String name){//違反
    this.name = name;
  }
  public synchronized void setBirthDay(long day){//違反
    birthday = day;
  }
}
```

※必要以上に同期をとっています。

setName()メソッド実行中には関係のない setBirthDay()メソッドも呼べません。

＜修正サンプル＞

```
public class FixedSample {
  private String name;
  private long birthday;
  public void setName(String name){  // 修正済み
    synchronized(this.name){       // 修正済み
      this.name = name;
    }
  }
  public void setBirthDay(long day){ // 修正済み
    synchronized(birthday){      // 修正済み
      birthday = day;
    }
  }
}
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

26. コーディング標準（ガーベッジコレクション）

(143) アプリケーションから finalize()を呼び出さない（★★★）

finalize()はオブジェクトが収集されるときにガーベッジコレクタによって自動的に呼び出されます。
明示的に finalize()を呼び出した場合でもガーベッジコレクタによって再度 finalize()が呼ばれます。
その際、finalize()メソッド内の処理によっては finalize()が二度呼ばれてしまうことによって不整合が生じ、バグのもととなる可能性もありますので、作成したアプリケーションからは呼ばないでください。

(144) System.gc()を実行しない（★★★）

アプリケーション中では、特別な理由がない限り、決して System.gc()を実行しないでください。
通常の状況では、System.gc()を使用した明示的な GC を実行しなくても、自動的に行われる GC だけでも全く問題はありません。
また、System.gc()を追加しても、パフォーマンスが向上することはほとんどありません。
Java のヒープメモリ中に作成された一時オブジェクトは、GC（ガーベイジコレクション）によってメモリから開放されます。
この GC が起こるのは以下の三つの場合です。

1. ヒープメモリ中に新規オブジェクトを作成するために必要な空き領域が足りなくなったとき
   （AF：Allocation Failure による GC）
2. プログラム中で System.gc()が実行されたとき
3. JVM で実行する処理がなくなってアイドル状態になったとき（Async GC）

このうち、3.の GC は WAS では起こりません。
WAS 内部では、サーブレットや EJB を実行していないときでもバックグランドで多くの処理が動いているためです。
2.と 1.の GC において、開放されるメモリの種類に違いはありません。
つまり、System.gc()を繰り返したからといって、別段効率的にメモリが開放されるわけではなく、使用可能なメモリが増えるわけではありません。
それどころか、System.gc()を繰り返すことにより、パフォーマンスに深刻なトラブルが発生することになります。

(145) 明示的 null 代入（★★）

明示的 null 代入は、単に使い終わった参照オブジェクトに null を代入するというものです。
この手法はオブジェクトをよく早くガベージコレクションの対象にするという考え方から生まれました。少なくとも理論上はそれが正しいとされています。
特にオブジェクトの参照を配列に保持しておく場合などは積極的に活用してください。

(146) finalize() をオーバーライドした場合は super.finalize() を呼び出す（★★）

finalize()メソッドはオブジェクトが消滅する際にあらゆるオブジェクトで必要な処理を JavaVM が行うためのメソッドです。

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

このメソッドをオーバーライドし **super.finalize()**を明示的に呼ばなかった場合、スーパークラスの **finalize()**メソッドは実行されなくなってしまい、必要な処理が実行されない可能性があります。

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

27. コーディング標準（画面：JSP）

(147) JSP では Java コードではなくカスタムタグを使用する（★★）

JSP では、<% ～ %>タグを使用することで Java コードを記述できますが、できるだけフレームワーク標準で準備されているカスタムタグを使用するようにしてください。
これは、カスタムタグでフレームワーク標準の動作を指定していたり、ソースの可読性をあげることになります。
本プロジェクトのフレームワークで準備しているカスタムタグについては、別途資料を参照してください。
ここでは、例として struts 標準で準備しているカスタムタグを説明します。

＜サンプル＞

① 変数の扱いは、struts-bean で準備されているカスタムタグを使用します。

<% String sample = result.getSample(); %>

⇒ <bean:define name="result" property="sample" type="java.lang.String" />

② html タグの出力は、struts-html で準備されているカスタムタグを使用します。

<input:text name="sample" value="<%= result.getSample() %>"

⇒ <html:text name="result" property="sample" />

③ ロジックは、struts-logic で準備されているカスタムタグを使用します。

<% if (sample) { %>

⇒ <logic:empty name="sample"></logic:empty>

<% for (int i = 0; i < sampleArr.lenght; i++) { %>

⇒ <logic:iterate id="sample" name="sampleArr"></logic:iterate>

(148) 文字の出力は、カスタムタグの write タグを使用する（★★★）

文字列の出力は、カスタムタグの write タグを必ず使用してください。
<%= %>で行うこともできますが、write タグではクロスサイトスプリクティング対応など、画面表示に特化した文字列の変換をしています。

例外として、html の input タグをなどの value 属性に直接値を指定する場合は、<%= ～ %>を使用します。
これは、表示用に文字列の変換をすることで文字コードが変化して、予期しない動作をする恐れがあるためです。
（例として、半角スペースは は代用できますが、半角スペースと では同じスペースとして表示されていても、実際の文字コードが異なります。）
基本的には、html タグに応じてカスタムタグが準備されていますので、これらのカスタムタグを使用してください。

(149) html タグ、カスタムタグはすべて小文字で記述する。（★★★）

Html の仕様上は、大文字小文字の制限はありません。
ただし、

① XHTML では、小文字で書かなければならない。

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

② カスタムタグは一般的に小文字を使用する。
③ 全体で大文字小文字の統一をした方がソースの可読性が高い。

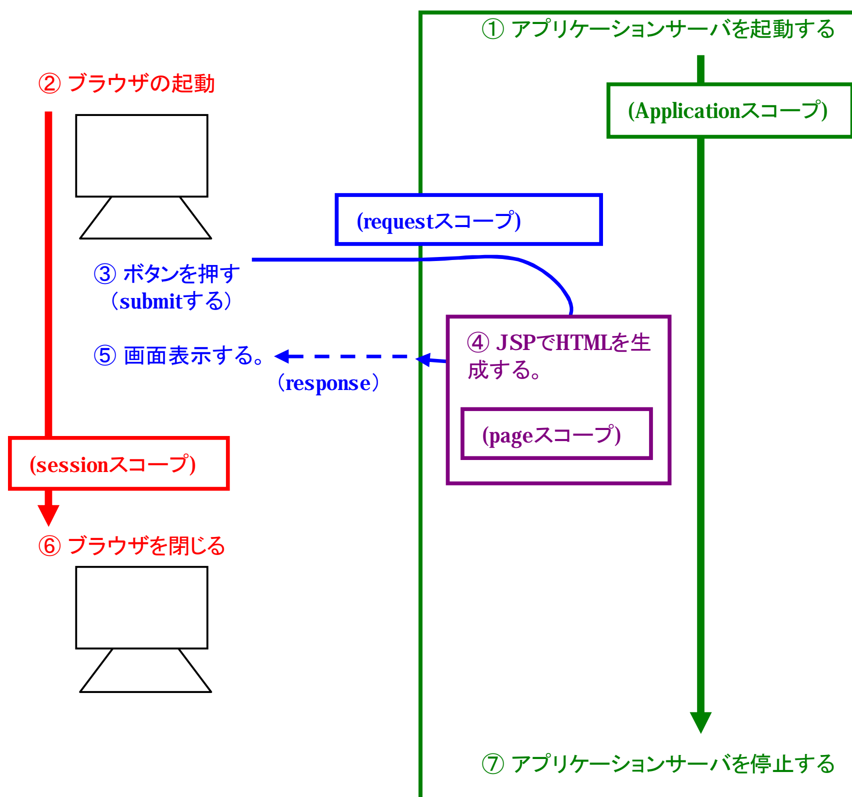
上記を理由として、タグはすべて小文字で記述してください。

(150) JSP の変数は、page スコープを利用する。(★★)

JSP の変数スコープを意識しましょう。通常の開発では、JSP の変数は page スコープを使用します。
また、同時に Servlet の変数スコープについても理解をしておきましょう。

page：　JSP ページ
request：　HTTP リクエスト
session：　HTTP セッション
application：　Web アプリケーション

＜JSP、およびServletでの変数のスコープを意識する＞



(151) JSP のインクルードは、<jsp:include>を使用する。(★★)

JSP のインクルードは、<jsp:include>と<%@ include %>の 2 つがあります。
本プロジェクトでは、以下の理由により<jsp:include>を使用します。

① <%@ include %>タグはインクルードされたファイルが更新されても、JSP 自体が更新されなければ JSP に更新は反映されない。(ただし、実行速度は、<%@ include %>が若干よいとされます。)
② <%@ include %>タグの場合、変数のスコープが複数の JSP にまたがるためソースの可読性に問題があり、またカプセル化の概念に反する。

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

(152) 半角スペースの表示は、 を使用する。（★★）

JSP（Html）では、半角スペースは文字列として認識しません。
半角スペースは、 で表示します。

△（半角スペース）　:
< : <
> : >
& : &
" : "
' : '

また、ソース上の文字列と文字列の間の改行コードは半角スペースに変換されて画面表示されます。

(153) <tabel>タグの見出しは、<th>で定義する。（★★）

表の行をセル単位に区切るには、<td>タグを用いますが、見出しとして使用するセルは<th>タグで定義します。

(154) <th><td>タグでは、width 属性を指定せずに nowrap 属性を指定する。（★★）

nowrap 属性を指定することで、セルで文字が折り返されることを回避します。
ただし、nowrap 属性が指定されていたとしても width 属性が指定された場合はこちらが優先されるため桁数がオーバーした場合はセル内で折り返して表示されます。

(155) JSP で使用する文字コード指定は、Windows-31J とする（★★）

Shift_JIS は機種依存文字（㈱、①など）に対応していないため、JSP の文字コード指定は Windows-31J を使用します。

(156) 画面個別でスタイルの変更（文字列の右寄せ、文字色の変更など）を変更するときは、カスタムタグの style 属性を使用する。（★）

標準的な画面表示は、フレームワーク共通のスタイルシートで定義しています。
これに外れる場合は、カスタムタグの style 属性を使用してください。
ただし、カスタムタグの style 属性を使用することでフレームワーク動作が無効になりますので、注意が必要です。
複数画面でスタイルの追加があるときは、フレームワーク共通のスタイルシートを追加することを推奨します。

＜参考：スタイルの優先順位＞

カスタムタグの style 属性
↑
フレームワーク共通のスタイルシート
↑
html タグ

style 属性は、フレームワーク共通のスタイルシートよりも優先順位が高くなります。
したがって、style 属性を使用したスタイルについてはフレームワークでの変更が反映されません。
使用には注意が必要です。

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

28. その他

(157) 自分で新しく作る前に相談（★）

他人が作成したクラス（フレームワークを含みます）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してください。
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ます。

(158) 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 言語の特性と良く会います。
java 言語の設計原理は KISS(Keep It Small and Simple)です。
また，後のメンテナビリティにも‘Simplicity’は重要です。

(159) トリッキーなコードは悪（★）

誰が見てもわかるようなソースコードを書きましょう。
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ソースコードが読みやすいです。
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);
```

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved

29. 参考資料

オブジェクト倶楽部 コーディング規約の会

URL: http://www.objectclub.jp/community/codingstandard/

・オブジェクト倶楽部版 Java コーディング規約

・株式会社電通国際情報サービス版 Java コーディング規約 2004

Checkstyle 日本語訳

以上

Copyright (c) 2008 https://www.code4u.jp All Rights Reserved