Java コーディング規約

本書は、Acroquest Technology 株式会社が作成した Java 言語に関するコーディング規約です。

本書は、世の中の Java 言語を利用するシステム開発プロジェクトの成功のため、無償で提供致します。

ただし、掲載内容および利用に際して発生した問題、それに伴う損害については、作成者である Acroquest　Technology 株式会社は一切の責務を負わないものとします。

また、掲載している情報は予告なく変更することがございますので、あらかじめご了承下さい。

ACR-0501-0204

—目次—

1 概要

1.1 名称

本書は、「Java コーディング規約」とする。

1.2 目的

本書は、コーディングにて Java を用いた開発を行う際に参照する規約や推奨事項を記述するものである。本書を参照することで、Java を用いたプログラムコードの品質水準を一定以上に維持することを目的とする。

2 はじめに

2.1 意義

ソフトウェアは、作成されるだけでなくそのライフサイクルを通して継続的にメンテナンスされるものである。このため、ソフトウェアの重要な一部分であるプログラムコードは保守性に優れ読みやすく記述されている必要がある。

表記法がバラバラなコード群は、保守を行う側にとって可読性を著しく低下させる。その結果、修正の度に元のソースコードを「読み解く」という不要なコストを発生させ、効率が悪い。

保守を行うのは、決して他のメンバばかりでない。それは最初に開発したプログラマ自身かもしれない。初期開発時から標準を守り、保守性に優れたコードを書くことは、自分自身の開発効率を向上させることにも繋がる。

組織で作成されるプログラムは、コーディング表現方法に規約を定め、プログラム開発に携わるメンバ間で周知・徹底する。これにより、可読性に優れ、保守性の高いソフトウェアを作成する。

以上の目的を達成するため、本書で Java コーディング規約を定める。

2.2 基本方針

(1) 分かりやすさ/見やすさの重視

他の人が読んでも分かりやすいコードであることは、「良いコード」の基本である。ここで言う「他の人」は第三者かもしれないし、近未来の実装者自身かもしれない。実装時の考えはコードとしてしか残っておらず、実装者自身も、数日も経てばそのときの考えは忘れてしまっているため、どちらにとっても「他人のコード」である。

そのため、コーディングを行う際には、常に分かりやすさ/見やすさに気を配ることが必要である。

(2) コードの重複の禁止（DRY（Don't repeat yourself）の原則）

Copy&Paste によるコードの重複は、絶対に行ってはならない。なぜならば、何らかの修正をする際に、コピーをした全ての箇所に修正をしなければならないからであり、これは、保守性、バグ修正の妨げとなる。

同じようなコードが現れるようならば、別メソッドとして記述し、共通化を図ることが必要である。

(3) ネーミングの分かりやすさへの配慮

コーディングでは、様々な変数やメソッドなどの名前を決める必要がある（ネーミング）。ネーミングは、その対象の本質を表すような名前を考える作業である。

名前の良し悪しは、コードの可読性に非常に大きな影響を与えるため、名前を見て、その変数やメソッドなどの役割が分かるようにする必要がある。

(4) 役割の単一化

クラスやメソッドの役割が明確であり、かつ、その役割が一つであることが望ましい。なぜならば、役割が一つであることは、必要以上にコードが複雑になることを避け、保守性の高いコードを作成することに繋がるためである。

(5) 保守性の優先

テクニックに凝ったコードよりも、可読性の高いコードを作成することの方が、プログラムの品質を向上させる上では重要である。

このため、原則、保守性に優れていると判断される記述を優先する。

(6) 規約チェックの自動化

コードスタイルおよび標準的な規約のチェックについては、フォーマッタや規約チェックツールを利用して、フォーマットや違反コードの検出を自動化する。

本章は、このコードフォーマッタの設定に対する設計を与える役割も果たす。

なお、空白や改行などの軽微なスタイルにはついては、本書では割愛している。これらは、ツールのルールを定義することで、自動でフォーマットできるためである。そのため、上記については、別途、各ツールのルールファイルを作成することとする。

ただし、開発者は、フォーマッタやチェックツールによる自動化に依存せず、本書の内容を理解して、普段から読みやすいコードを記述することを実践するのが望ましい。

以下に標準として利用する規約チェックの自動化ツールについて説明する。

① Eclipse標準機能

Eclipse の標準機能により、ソースコードのフォーマットが可能である。ルールを標準化することで、単純なスタイルのフォーマットを自動的に行うことができる。

No.	チェックツール	機能
1	Formatter	両方とも、ソースコードの整形を行う機能であるが、主に以下のような整形を行う。 【Formatter】 改行、スペースといったスタイルの整形 【Clean Up】 条件文を括弧で囲むかどうか、フィールド・アクセスやメソッド・アクセスに'this'修飾子を使うかどうかといった一貫性のあるコードへの修正
2	Clean Up	

これらのルールは、定義ファイルでカスタマイズが可能である。

プロジェクト毎にルールファイルを作成し、全プロジェクトメンバで共有することが重要である。

② 静的チェックツール

以下の静的チェックツールを利用することで、命名規則やプログラムの問題を自動的にチェックすることができる。

Eclipse プラグインが存在するものもあるため、それらを積極的に活用し、実装時から品質の高いコードを作成することが重要である。

No.	チェックツール	チェック内容
1	CheckStyle	コーディング規約をチェックする。
2	FindBugs	静的解析を行い、バグの可能性があるコードを指摘する。
3	JavaNCSS	循環的複雑度（CCN：Cyclomatic Complexity Number）を計測し、ソースコードの複雑度をチェックする。
4	CPD	重複コードの有無をチェックする。

開発者は、プラグインがエラーを検出した場合には直ぐに修正すること。

エラーは、原則 0 件に保つことが重要である。

(7) 実コードとテストコード

作成したプログラムを JUnit などの自動テストツールを利用してテストを行う際には、開発者がプログラム本体のコード（実コード）とは別にテストコードを作成する必要がある。

これらのテストコードは、実コードとは別のディレクトリツリーで管理する。こうすることで、納品物件としてのコードと試験のための内部コードを明確に切り離し、管理を単純化する。

(8) 本書で想定するJDKのversion

本書では JavaSE 7.0 以降の JDK ランタイムを使用することを前提とする。

2.3 メンテナンス方針

2.3.1 規約の変更

本章は、組織で統一されたコーディングの標準を定義し、SEPG（Software Engineering Process Group）がその管理を行う。

開発者は、特に理由のない限りこの規約に従い、遵守すること（ただし、本書を基に、プロジェクトでカスタマイズしたコーディング規約を定めることは可能）。

本書に対する改善を要望する場合は、SEPG に対しリクエストを提出し、メンテナンスを依頼する。

2.3.2 フィードバック

プロジェクトで固有の規約を使用する際は、本書の管理を行っているSEPG にその理由と共にフィードバックを行うこと。これは本来、コーディング規約が現場の開発者の経験を汎用化したものであり、その経験に基づいて成長させていくべきものだからである。

3 プロジェクト構成

Java 言語の場合、原則、Eclipse を利用して開発を行う。

Eclipse のプロジェクトは、以下のディレクトリ構成を標準とする（Maven プロジェクトのディレクトリ構成を基準としている）。ソースコードの位置やコンパイルで生成されるクラスファイルの出力場所などは、Eclipse で適宜設定する。

```
Project
 |  build.xml                  …Ant 用ビルドスクリプト（ビルド成果物作成用）
 |  pom.xml                    …Maven 用ビルドスクリプト（レポート出力用）
 |  eclipse-cleanup.xml        …Eclipse の Cleanup ルール
 |  eclipse-codeformat.xml     …Eclipse のコードフォーマットルール
 |  checkstyle.xml             …CheckStyle ルール定義ファイル
 |  findbugs-exclude.xml       …FindBugs の除外ルール定義ファイル
 |
 ├──bin                        …起動スクリプトやデータ投入スクリプト
 ├──config                     …設定ファイル
 ├──data                       …マスタデータやテスト用のデータファイル
 ├──lib                        …リリースに関係するライブラリ
 ├──libext                     …リリースに関係しないライブラリ
 ├──src
 |   ├──main
 |   |   ├──java               …ソースコード
 |   |   └──resources          …リソースファイル（java ソース以外を配置）
 |   └──test
 |       ├──java               …テスト用ソースコード
 |       └──resources          …テスト用リソースファイル
 └──target
     ├──classes                …ソースコードをコンパイルしたクラスファイル
     └──test-classes           …テスト用ソースコードをコンパイルしたクラスファイル
```

4 パッケージ構成

4.1 プロジェクト

プロジェクトで作成されるソースコードは、プロジェクトで定めたパッケージ構成に従ってクラス/インタフェースを設計し、配置する必要がある。

4.2 基本的なパッケージ構成

4.2.1 階層の俯瞰

パッケージ名は全世界で一意であることを保証するため、ドメイン名を逆順にしたプレフィックスをつける。ただし、納品条件がある場合は、納品条件を優先する。

本書では、アクロクエストテクノロジー株式会社（以後、当社と記す）を主体とした開発プロジェクトでのパッケージ構成を規定する。

4.2.2 各階層の意味

- 第 1 階層～第 3 階層（jp.co.acroquest）
  第 1 階層～第 3 階層までは、開発プロジェクトの主体となる組織のドメイン名を逆順にしたプレフィックスを用いる。すなわち、当社を主体とした開発プロジェクトで作成する成果物のパッケージ名は、jp.co.acroquest で開始する。
- 第 4 階層以降
  第 4 階層以降は、そのクラス/インタフェース群の分類を段階的に詳細化して設定する。

  第 4 階層以降のパッケージ構成に関する一般的な指針を以下に示す。

表 4-1 第 4 階層以降のパッケージ構成の指針

階層	分類内容	例
第 1～3 階層	ドメイン名	`jp.co.acroquest`
第 4 階層	プロジェクト名	`gnms`
第 5 階層	サブシステム名 または コンポーネント名	`disp` `appmgr`

第 6 階層	コンポーネント名
	(詳細化した分類)
	※必要に応じて設定する
		`info`
		`processor`
		`impl`

- パッケージ階層は、個々のプロジェクト開始時に設計・規定し、メンバ間で共有する。また、管理者を定めて継続的にメンテナンスすることが望ましい。

5 ソースコード構成

5.1 全体構成

ソースコードの構成は、以下のようになる。

ファイルヘッダ部	
パッケージ記述部	
インポート宣言部	
クラスヘッダ部	
クラス定義部	定数
	フィールド
	コンストラクタ
	メソッド
	内部クラス

5.2 各構成要素

(1) ファイルヘッダ部

各ソースファイルは、その先頭に版権を表すコメントを記述する必要がある。
これにより、ソースコードの著作権、責任範囲を明確にし、組織の資産・権利を保護する。
版権は 1 ソースファイルにつきただ 1 つのみ記述し、複数記述してはならない。
版権は、以下の通り記述する。

```
/**
 * Copyright (c) Acroquest Technology Co., Ltd. All Rights Reserved.
 * Please read the associated COPYRIGHTS file for more details.
 *
 * THE SOFTWARE IS PROVIDED BY Acroquest Technology Co., Ltd.,
 * WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING
 * BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
 * IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDER BE LIABLE FOR ANY
 * CLAIM, DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING
 * OR DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.
 */
```

(2) パッケージ記述部

ソースファイルに記述するクラス、インタフェースが属するパッケージ名を記述する。

(3) インポート宣言部

ソースファイルに記述するクラス、インタフェースが利用する外部パッケージのクラスまたはインタフェースのインポートを宣言する。
アスタリスク(”*”)は使用せず、各クラス/インタフェース名まで記述すること。

(4) クラスヘッダ部

クラス定義に先立ち、宣言部の直前にクラスの説明を javadoc コメントとして記述する。
javadocコメントの記述方法については、「8.1 Javadoc」を参照すること。

(5) クラス定義部

クラス定義は、以下の順序で各要素を構成する。

- 定数
- フィールド
- コンストラクタ
- メソッド
- 内部クラス

5.3 各構成要素に関する規約

(1) インポート宣言

アスタリスク(“*”)は使用せず、各クラス/インタフェース名まで記述すること。

(2) クラス/インタフェース宣言

原則として 1 ファイル 1 クラスで記述すること。

(3) フィールド

以下の順序で記述すること。

1） public→protected→未指定(package private)→private の順序で記述する。
ただし、上記は原則であり、機能的なまとまりを考慮した方がわかりやすい場合は、そのような順序で記述しても良い。

可能な限り、final でない static フィールドは作成すべきではない。どうしても必要な場合、static は非 static フィールドより前に書く。

(4) メソッド

以下の順序で記述すること(番号が若い方を優先する)。

1） コンストラクタ→非 static メソッド→static メソッド。
2） public→protected→未指定(package private)→private。
ただし、上記は原則であり、機能的なまとまりを考慮した方がわかりやすい場合は、そのような順序で記述しても良い。
3） オーバーロードしているメソッド(引数が少ない順)→オーバーロードされているメソッド。
これは、doSomething(String str) が doSomething(String str, boolean flag) の呼び出しを行うようなオーバーロードの場合、本当の処理が記述されているメソッドを最後に書く、ということを意味する。
4） クラスの get/set メソッドは他の全メソッド宣言の後に書く。膨大な数の自明なメソッドの中に、自明でないメソッドが埋まっているコードを書くと、コードの可読性が著しく下がるため。

(5) 内部クラス

内部クラスの使用は原則禁止する。

内部クラスは再利用性・可読性を損なう可能性が高いため、使用しない。内部クラスは、全て単独のクラスに置き換えることが可能である。

以下のように規模が小さく、かつ再利用を前提としないケースでは、内部クラス（無名内部クラスではない）を使用しても構わない。

- GUI のイベントリスナー実装。（複数のリスナーを実装するような場合は除く）
- 汎用性の無いコールバック処理。

内部クラスを使用する場合は他のクラスに依存しない（他のクラスから生成されない）こと、再利用を前提としないことなどの検討を行い、コメントに記述すること。

6 命名規則

6.1 命名規則に関する共通的な事項

- 名称には英語を利用する(ローマ字表記禁止)。

説明	名称は全て英語を基本とする。 スペルミスを防ぐため、使用する英語は、辞書でスペルを確認すること(xxxer、xxxor などは、特に間違いやすい)。
動機	日本語名をローマ字表記した名前を使用しても、オフショア開発などで意味が分からず、開発者が困惑するため。

違反サンプル	**修正サンプル**
public class Kuruma	public class Car

- 識別子には意味のある名前をつける。

説明	識別子には、その役割が分かるような、意味のある名前をつける。
動機	ID、連番などによる命名は一見してその内容を推測することができず、著しく可読性を落とすため使用してはならない。また、普遍的な名称を使用しない。

違反サンプル	**修正サンプル**
jp.co.acroquest.framework.a0001 int a;	jp.co.acroquest.framework.appmanager int count;

- 大文字・小文字の違いで名前を区別する識別子を定義してはならない。

説明	大文字・小文字の違いで名前を区別する識別子を定義してはならない。
動機	名前で意味を示しているのに、大文字、小文字の違いでは、意味の違いが分からないため。

違反サンプル	**修正サンプル**
private int number; private int Number;	private int minNumber; private int maxNumber;

- 先頭にアンダースコア"_"を用いることを禁止する。

説明	先頭にアンダースコア"_"を用いることを禁止する。
動機	人によって記述形式が異なると可読性が悪いため、記述形式を揃える。

違反サンプル	**修正サンプル**
private int _number;	private int number;

- 任意の位置に"$"を用いることを禁止する。

説明	任意の位置に"$"を用いることを禁止する。
動機	内部クラスと混同するため。 フレームワークによっては、$を使って自動的に生成したクラス名に使用していることがあるため。

違反サンプル	**修正サンプル**
private int column$1;	private int column1;

- パッケージ名プレフィックスを除き、国コード及びドメイン名と重複してはならない。

説明	パッケージ名プレフィックスを除き、国コード及びドメイン名と重複してはならない。
動機	パッケージ構成が分からなくなるため。

違反サンプル	**修正サンプル**
jp.co.acroquest.appmanager.acroquest.tools	jp.co.acroquest.appmanager.common.tools

6.2 ソースコード内の各要素に対する命名規則

6.2.1 パッケージ

- パッケージ名は小文字で構成する。

説明	パッケージ名は全て小文字で構成する。	
動機	クラス名と混同するため。	
違反サンプル		修正サンプル
jp.co.acroquest.SampleSystem.SomePackage		jp.co.acroquest.samplesystem.somepackage

- アンダースコア"_"を使用しない。

説明	パッケージ名にアンダースコア"_"を使用しない。	
動機	可読性向上のため。	
違反サンプル		修正サンプル
jp.co.acroquest.samplesystem.some_package		jp.co.acroquest.samplesystem.somepackage

6.2.2 クラス/インタフェース

(1) 通常クラス

- クラス名は Pascal 形式とする。

説明	クラス名は単語の先頭を大文字にする。名称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)を大文字にして連結する。	
動機	Java 標準に準拠。	
違反サンプル		修正サンプル
public class someobject		public class SomeObject

- クラス名は、役割を果たす名前にする。

説明	クラス名は、役割を果たす名前にする。過度に長くない、名詞又は名詞句であることが望ましい。	
動機	クラス名からその役割が分からない場合、ソースの詳細を見る必要があるため。	
違反サンプル		修正サンプル
public class T001		public class Car

- 名詞に含まれる単語は省略しない。

説明	名詞に含まれる単語は省略しない。ただし、URL や HTTP のように略語が一般的であると認められる場合は略語を用いることが出来る。	
動機	勘違いを極力無くすため。省略すると、意味が分からなくなることがある。	
違反サンプル		修正サンプル
// 「ResponseHandler」なのか、 // 「ResultHandler」なのか分からない。 public class ResHandler		public class ResponseHandler

- JDK 標準クラスライブラリに存在する名前と重複する名前を使用してはならない。

説明	クラス名に既に JDK 標準クラスライブラリに存在する名前を使用しない。
動機	IDE による自動補完などで間違った補完を行い、意図しないソースになることがあるため。

違反サンプル	修正サンプル
package jp.co.acroquest.util; public class List　// java.util.List と同じ	package jp.co.acroquest.util; public class CustomerList

(2) 抽象クラス

- 抽象クラスはそれが抽象クラスであることを示す形容詞を先頭に付ける。

説明	抽象クラスはそれが抽象クラスであることを示す形容詞を先頭に付ける。適当な名前がない場合は、先頭に"Abstract"を付与し、その後にサブクラスを連想させる名前を連結する。
動機	クラス名を見ただけで、そのクラスが抽象クラスであることが分かると効率が良いため。

違反サンプル	修正サンプル
abstract class Sample	abstract class AbstractSample

(3) 例外クラス

- 例外クラスは、"Exception"を接尾辞に付けた名前にする。

説明	例外クラスは、それが例外クラスであることを示すように、最後に”Exception”を付与する。
動機	クラス名を見ただけで、そのクラスが例外クラスであることが分かると効率が良いため。

違反サンプル	修正サンプル
abstract class SampleErr	class SampleException extends Exception

(4) インタフェース

- インタフェース名は、Pascal 形式とする。

説明	インタフェース名は単語の先頭を大文字にする。名称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)を大文字にして連結する。
動機	通常のクラスと同様の命名規則とする。

違反サンプル	修正サンプル
public interface samplereader	public interface SampleReader

- インタフェース名は、役割を抽象化した名称にする。

説明	インタフェース名は、java.io.DataInput 及び java.io.DataOutput のように、役割を抽象化した名称にする。
動機	インタフェース名を見ただけで、その役割が分かるようにするため。

違反サンプル	修正サンプル
public interface SomeInterface	public interface SampleReader

- 能力付加型のインタフェース名は"～able"とする。

説明	能力付加型のインタフェース名は、java.lang.Runnable 及び java.lang.Cloneable のように、"～able"とする。
動機	インタフェース名を見ただけで、その役割が分かるようにするため。

違反サンプル	修正サンプル
public interface ReloadInterface	public interface Reloadable

- インタフェース名にインタフェースとクラスを区別するためのプレフィックス、サフィックスを用いることを禁止する。

説明	インタフェース名にインタフェースとクラスを区別するためのプレフィックス、サフィックスを用いることを禁止する(開発環境が自動生成する Java 言語プログラムは対象外とする)。
動機	Java 標準に準拠。

違反サンプル	修正サンプル
// I～というプレフィックス ISomeInterfaceName // ～IF というサフィックス ConnectionIF	SomeInterfaceName Connection

6.2.3 フィールド

(1) 共通

- boolean 型を表す属性は、true/false の状態がわかる名称にする。

説明	true/false のどちらが、どのような状態を指すのか、分かるようにする。 単語だけで分からない変数の場合は、「is＋形容詞」または「is＋名詞」にする。
動機	変数名を見るだけで、true/false のどちらが、どのような状態を指すのか、分かるようにするため。

違反サンプル	**修正サンプル**
private boolean open;	private boolean enable; private boolean exists; private boolean hasValue; private boolean isOpened;

- 親クラスで定義されているフィールドと同じ名前のフィールドを定義してはならない。

説明	継承をしている場合に、親クラス/子クラスで、同じ名前のフィールドを定義しない。
動機	親クラスの変数なのか、子クラスの変数なのか分からなくなってしまうため。

違反サンプル	**修正サンプル**

(2) 定数

static final とし、すべて大文字で記述し、各単語をアンダースコア 1 つで区切る。

説明	定数名は全て大文字で記述し、各単語をアンダースコア 1 つで区切る。また、定数名は、記述的であって、不必要に省略しないこと。 複数の定数で 1 の群を形成する場合、適切な短いプレフィックスをつけることが望ましい。
動機	名前を見ただけで、定数と分かるようにする。

違反サンプル	**修正サンプル**
public static final String propKey = "sample.key";	public static final String KEY_PROP = "sample.key";

(3) 属性（クラス変数）

- Camel 形式とし、末尾にアンダースコアを 2 つ付加する。

説明	名称は、単語の先頭を小文字にし、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）を大文字にして連結する。 クラス変数には、末尾にアンダースコアを 2 つ付加する。 ただし、プロジェクトによっては、アンダースコアなしで統一しても良い。
動機	変数名を見ただけで、クラス変数ということが分かるようにする。

違反サンプル	**修正サンプル**
private static String classfield;	private static String classField__;

(4) 属性（インスタンス変数）

- Camel 形式とし、末尾にアンダースコアを 1 つ付加する。

説明	名称は、単語の先頭を小文字にし、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）を大文字にして連結する。 インスタンス変数には、末尾にアンダースコアを 1 つ付加する。 ただし、プロジェクトによっては、アンダースコアなしで統一しても良い。
動機	変数名を見ただけで、インスタンス変数ということが分かるようにする。

違反サンプル	**修正サンプル**
private int amountOfStock;	private int amountOfStock_;

6.2.4 メソッド

(1) 共通

- メソッド名は、Camel 形式とする。

説明	メソッド名は、動詞で表す名称とし、先頭を小文字として単語の区切りのみを大文字とする。
動機	Java 標準に準拠。

違反サンプル	修正サンプル
public void sampleMethod();	public void readText();

(2) getter/setter

- getter/setter（属性を取得・設定するメソッド）は"get＋属性名"または"set＋属性名"とする。ただし、boolean 型の属性を取得するメソッドは、"is＋属性名"とする。
- これらのメソッド名に使用する属性名は先頭文字を大文字に変換する。

説明	getter/setter の命名規則を統一することによって、メソッドの役割を明確にする。 逆に、それ以外の目的で、"get＋属性名"、"set＋属性名"という命名規則を利用しないこと。
動機	Java 標準に準拠。

違反サンプル	修正サンプル
	public void setAttribute(int attribute) {} public int getAttribute() {} public void setVisible(boolean visible) {} public boolean isVisible() {}

(3) 判定メソッド

- 判定メソッド（boolean の結果を返すメソッド）は、その戻り値の true/false の状態がわかる名前にする。形式として、yes または no を返す疑問文にすることが望ましい。

説明	true/false のどちらが、どのような状態を指すのか、分かるようにする。 単語だけで分からない変数の場合は、「is＋形容詞」または「is＋名詞」にする。
動機	メソッド名を見るだけで、true/false のどちらが、どのような状態を指すのか、分かるようにするため。

違反サンプル	修正サンプル

public boolean empty() {}
// 動詞と混同するため望ましくない
	public boolean isEmpty() { }
	
	// 有効か
	public boolean isEnabled() { }
	// 削除できるか
	public boolean canRemove() { }
	// 削除されているか
	public boolean hasRemoved() { }
	// 次のトークンを取得できるか
	public boolean hasMoreToken() { }
	// element が含まれているか
	public boolean contains(Object element) { }

(4) ファクトリメソッド

- ファクトリメソッド（オブジェクトを生成するメソッド）は"create＋オブジェクト名"とする。

説明	オブジェクトを生成するメソッドは、メソッド名の先頭に”create”を付加する。
動機	メソッド名を見るだけで、役割が明確になるようにする。

違反サンプル	修正サンプル
public SampleBean makeSampleBean()	public SampleBean createSampleBean()

(5) コンバータメソッド

- コンバータメソッド（オブジェクトを別のオブジェクトに変換するメソッド）は"convertTo＋変換後オブジェクト名"とする。

説明	オブジェクトを別のオブジェクトに変換するメソッドは、メソッドの先頭に”convertTo”を付加する。
動機	メソッド名を見るだけで、役割が明確になるようにする。

違反サンプル	修正サンプル

public SampleBean makeSampleBean()	public SampleBean convertToSampleBean()
	// 特定のオブジェクトに変換する場合
	
	public SampleClass convertTo()
	// 自分自身に変換する場合

(6) CRUD系（Daoなど）メソッド

- データ操作に関するメソッドは、命名規則を統一する。

説明	データの追加は insert、更新は update、削除は delete、検索は”find+条件”とする。
動機	データを操作するメソッドは、複数箇所で利用される場合が多いが、命名規則を統一することで、メソッド名を見るだけで、役割が明確になるようにする。

違反サンプル	修正サンプル
	public void insert() public void update() public void delete() public SampleBean findById(int id) public List findAll() public List findByName(String name)

6.2.5 ローカル変数

(1) 共通

- ローカル変数は Camel 形式とする。
- 変数の役割を簡潔に表す、意味のある短い単語であること。

説明	ローカル変数は先頭が小文字で単語の区切りのみを大文字にした名称にする。属性と明確に区別するため、末尾にアンダースコアを付加してはならない。 変数の役割を簡潔に表す、意味のある短い単語であること。スコープが狭い変数には適度に省略した名称を用いても良い。 この場合の「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指す。
動機	Java 標準に準拠。

違反サンプル	**修正サンプル**
String str_;	String str;

(2) ループ変数

- ループ変数に、i、j、k など一文字の変数を用いてはならない。

説明	ループ変数に、i、j、k など一文字の変数を用いてはならない。 これらの変数名は意味を類推することが困難なため、意味のある適切な名称（略語でも良い）にする。
動機	一文字の変数は、見間違いなどが発生しやすいため。

違反サンプル	**修正サンプル**

// 変数が分かりにくい
for (int i = 0; i < length; i++)
{
　　// 処理
}
	// 変数が分かりやすい
	for (int index = 0; index < length; index++)
	{
	　　// 処理
	}

表 6-1 代表的なループ変数の名称

No.	変数名	用途
1	index	配列を走査する処理。
2	count	カウント処理。

6.3 名称の決定方針

6.3.1 用語の統一

- プログラム全体で統一された名称を用いる。

説明	名称の決定にあたり、プログラム全体で統一された名称を用いることは可読性の向上につながる。このため、プロジェクトで統一された用語を管理する辞書を作成し、メンテナンスされることが望ましい。 ここでいう辞書は、クラス名やインタフェース名、メソッド名そのものではなく、それらを構成する単語と、その用途などである。
動機	可読性向上のため。

違反サンプル	**修正サンプル**
// 同じ処理なのに、名称が異なる public void change() public void modify() public void invoke() public void execute()	// 同じ処理は、名称を統一。 public void change() public void invoke()

6.3.2 名称の対称性

- 英単語の対称性を意識した名称にする。

説明	クラスやインタフェースやメソッド名、変数などの役割や機能が対になる場合は、英単語の対称性を意識した名称にする。これにより、利用者などから対となる構成要素の存在を類推しやすくなる。
動機	可読性向上のため。

違反サンプル	**修正サンプル**
// 名称が対称的でない public void open() public void dispose() public void read() public void output()	// 名称が対称的である public void open() public void close() public void read() public void write()

7 スタイル

スタイルは、原則、コードフォーマッタ（Eclipse 標準機能）や Checkstyle を利用して、自動で整形、チェックを行なう。ルールとしては、以下の内容を標準化する。

- コードの長さ
- 改行位置
- インデント
- 空白/ブランク行の挿入位置

システム全体でスタイルが統一されるよう、標準ルールを作成した上で、メンバが上記ツールを利用することを徹底する。

本章では、そのルールの内、特に理解しておくべき内容のみ、記述する。

7.1 ソースコードのフォーマットに関するスタイル

7.1.1 コード長の基準

表 7-1 コード長の基準

No.	要素	値	説明
1	クラスのステップ数	通常： 400 LOC 最大： 800 LOC	最大値を超える場合は、責務が肥大化している可能性が高いため、処理を分割するように見直す。
2	メソッドのステップ数	通常： 60 LOC 最大： 120 LOC	最大値を超える場合は、責務が肥大化している可能性が高いため、処理を分割するように見直す。
3	ブロックのネスト	1 メソッド内で、 最大 3 階層	最大値以上に深い階層となる場合は、可読性が著しく損なわれる。そのため、以下の点を見直す。 ・ 本当にそのネストが必要かどうか ・ メソッドに切り出せないか

7.1.2 ステートメント/ブロック

(1) ステートメント

- 1 行に 2 つ以上のステートメントを記述しない。

説明	1 行に 2 つ以上のステートメントを記述しない。
動機	Java 標準に準拠。

違反サンプル	**修正サンプル**
int result = top - bottom; return result;	int result = top - bottom; return result;

(2) ブロック

- 開始の中カッコは、宣言文の次行に単独で記述する。

説明	BSD//Allman スタイルに準拠し、開始の中カッコは、宣言文の次行に単独で記述する。 開始中カッコの後にステートメントを記述してはならない。コメントの記述も禁止する。
動機	可読性を考慮。

違反サンプル	**修正サンプル**

private void sampleMethod() {
　　int value = 0;
}
 	
if (args.length > 0) {
　　int value = 0;
}
	private void sampleMethod()
	{
	　　int value = 0;
	}
	
	if (args.length > 0)
	{
	　　int value = 0;
	}

7.1.3 インデント

(1) インデント

インデント（段下げ）は、スペース 4 個分とし、タブは利用しない。

7.2 this/superの指定

オブジェクトのインスタンスを示す、this, super は、以下の方針で指定する。

表 7-2 this/super の指定

No.	対象	指定の有無	備考
1	インスタンス変数	○	
2	クラス変数	×	
3	メソッド	×	継承メソッドなどで、明示的に親クラスのメソッドを呼び出す必要がある場合は、super を利用する。

8 コメント

コメントは自分のみならず、他人が読んだ際に説明として機能することが重要である。保守性、可読性を向上させておくことで、他人が API を呼び出す場合や、ソースコードを改修する場合に役立つ。

コメントに口語などのくだけた文体を用いず、「だ/である」調で記述する。ただし、パッケージ製品など広くユーザに読まれることを意識する場合などでは、「です/ます」調を使用しても良い（プロジェクト内で文体を統一すること）。

また、コメントを記述する際は、以下のことを意識して記述すること。

- コメントは必要十分なボリューム、内容で記述する。
- 冗長なコメントを記述しない。冗長なコメントは、可読性を損ない、また修正時にコードとの相違などミスを誘発する。冗長、あるいは長すぎると判断されるコメントの種類は以下のようなものである。
  1) 要旨を述べていない。
     長くなるコメントは、最初の一文に要旨をまとめるべきである。
  2) 一文毎に記述されている。
     適切な処理ブロックについて説明を記述すればよい。
  3) そもそも対象となるコードが複雑であるため、大量の説明が記述されている。
     このような場合は、対象となるコードが適切な粒度であるかを再検討するべきである。
  4) 前提などを省略されている。
     説明が不足したコメントはコメントとして機能しないため、意味が無い。作成者だけが理解できる（他人が理解できない）コメントも同義である。対象となる要素の意味、目的、前提などを過不足無く記述するように意識すること。

8.1 Javadoc

Javadoc コメントを活用すること。Javadoc コメントを使用することで、ツールを利用して HTML に変換し、API 仕様をドキュメント化することが可能となる。

8.1.1 共通

(1) 記述のポイント

- 文の末尾は全角の句点"。"で終わること。
  - Javadoc ツールは、最初の句点までを最初の一文と判断して、サマリの Javadoc を抽出するため。

  改行、リスト等、HTML タグを使用して記述すること。
  - Javadoc コメントに記述する内容は、実際に Javadoc ツールが生成する結果が読みやすい形にするため。

  Javadoc コメントは、クラスやメソッドの外部仕様として記述すること。つまり、IPO（Input-Process-Output）が明確になるように説明を記述すること。

(2) Javadocコメントを必須とする位置

- public または protected、デフォルトアクセス権のクラス、フィールド、メソッド定義

(3) Javadocコメントを推奨する位置

- private のクラス、フィールド、メソッド定義

8.1.2 記述ルール

(1) クラスのコメント

- クラスの役割について記述する。
- author タグを記述し、クラスの作成者および更新者を記述する。複数記述する場合は一人ずつタグを記述する。ただし、プロジェクトによっては版権が自社にないことがあるため、author タグを記述するかどうかは、プロジェクト毎に決定すること。

(2) フィールドのコメント

- フィールドのコメントは、一文で簡潔に記述することが望ましい。ただし、必要に応じて複数の文に分割し、補足説明を加えても良い。

(3) メソッドのコメント

- メソッドのコメントには、メソッドの入力に対してどのように動作するか？という仕様を記述する。
- メソッドのコメントには、param、return、throws（または exception）タグを必須とする。
  ただし、それぞれがメソッド定義に含まれていない場合は記述しない。
  1) param タグ
     引数の名前とその説明を宣言されている順に記述する。
     複数の引数を記述する場合は、それぞれの引数についてタグを記述する。
  2) return タグ
     戻り値についての説明を記述する。
  3) throws タグ
     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。
     複数の例外を記述する場合は、それぞれの例外についてタグを記述する。
     未捕捉例外（RuntimeException の派生クラス）については、原則記述する必要はない。ただし、そのメソッド内で生成し、throw するものについては記述すること。

8.1.3 Javadocコメントの記述例

```
/**
 * ログイン処理を行なうクラス。<br />
 * このクラスはスレッドセーフではないため、非同期で処理する場合は、
 * 処理毎にインスタンスが生成されるようにする。
 *
 * @author Acro Taro
 */
public class LoginService
{
    /** ログインするホスト名 */
    private String host_;

    /**
     * デフォルトコンストラクタ。
     */
    public LoginService()
    { }

    /**
     * 指定されたユーザ ID、パスワードで、ログインを行なう。<br />
     * ログインの結果を true/false で返す。<br />
     * 以下の場合は、ログインエラーとなる。<br />
     * <ul>
     *   <li>指定されたユーザ ID、パスワードが存在しない場合</li>
     *   <li>指定されたユーザアカウントが失効している場合</li>
     * </ul>
     *
     * @param userId ユーザ ID
     * @param password パスワード
     * @return true の場合はログイン成功、false の場合はログイン失敗。
     * @throws AppException ホストへの接続に失敗した場合。
     */
    public boolean login(String userId, String password) Throws AppException
    {
        // ログイン処理を行なう。
    }
}
```

8.2 通常コメント

Javadoc 以外に、処理が複雑な箇所など、保守性を向上させるために、コメントを適宜記述する。

8.2.1 コメントが必要な箇所

主にコメントが必要となるのは、以下のような箇所である。

- if～else 文や switch 文などで、処理が複数に別れる部分
- for 文や while 文などで、ループが終了する条件の部分
- 処理が複雑な部分
- 処理の順序や並列性が関係する部分
- 状態遷移が発生する部分

8.2.2 コメントが不要な箇所

以下のような箇所は、コメントは不要である。

- ソースコードの変更箇所
  構成管理の変更ログ機能を利用することとし、コード中には、履歴コメントを記述しない。履歴コメントを残すことはコードにとって意味が無く、可読性を落とすためである。
- 不要となったコード
  一時的なコメントアウトを除き、不要なコードは削除する。後になると、コメントアウトされている理由が分からなくなり、削除して良いのかどうか、分からなくなるためである。
- コードを見れば明らかな処理
  見るだけで処理が明確な内容については、コメントは不要である。過度なコメントは、逆に可読性を落とす。例えば、処理の一行一行にコメントを記述するようなことはしない。

8.3 TODOコメント

- TODO コメントは、コーディング終了時には、原則、全て消しておく。
  - コーディング完了の段階で、TODO コメントが残っていないことを確認する。
  - ただし、次期のリリースなどで対応する内容として、明示的に TODO コメントを残している場合を除く。

9 テストコード規約

xUnit を利用することで、一部のテストを自動化することが可能である。本章では、テストコードに関する規約を説明する。

テストコードで重要となる可読性・保守性は実プログラムとは異なる。
テストコードで可読性が高いとは、そのテストコードがどのような試験を行っているのかが分かり、網羅性がチェックしやすい状態を指す。
保守性が高いとは、テストパターンの追加・削除が容易な状態を指す。
可読性・保守性の高いテストコードを作成するための規約を以下に示す。

また、本章では、ソースコードとテストコードは、以下の意味で扱うものとする。

ソースコード	アプリケーションの処理を記述したコード。テストの対象となる。
テストコード	ソースコードに対するテストを記述したコード。

9.1 構成

- ソースコードとテストコードとはルートディレクトリを分け、パッケージ構成を同じにする。

説明	ソースコードとテストコードは、ルートディレクトリで分け、パッケージは同じ構成とする。 src ├──main │ ├──java ・・・ソースコードを配置 │ │ └jp │ │ └co │ └──resources └──test ├──java ・・・テストコードを配置 │ └jp │ └co └──resources
動機	ビルドやリリースの際に、ソースコードとテストコードを簡単に分けられるようにするため。

違反サンプル	**修正サンプル**
src/jp/co/acroquest/app/ Sample.java SampleTest.java	src/main/java/jp/co/acroquest/app/ Sample.java src/test/java/jp/co/acroquest/app/ SampleTest.java

- テストコードのクラスはソースコードのクラスと 1 対 1 となるように作成する。

説明	テストコードのクラスはソースコードのクラスと 1 対 1 となるように作成する。
動機	ファイル名を見ただけで対応付けがすぐ分かるようにするため。

違反サンプル	修正サンプル
src/main/java/jp/co/acroquest/app/ 　　Sample.java src/test/java/jp/co/acroquest/app/ 　　ExampleTest.java	src/main/java/jp/co/acroquest/app/ 　　Sample.java src/test/java/jp/co/acroquest/app/ 　　SampleTest.java

- テストパターン 1 つにつき 1 メソッドを作成する。

説明	テストパターン 1 つにつき、テストメソッドを 1 つ作成する。
動機	何のテストを行うのか、メソッド単位で分かるようにするため。

違反サンプル	修正サンプル

public void testSampleMethod_値のチェック
{
　　// null の場合
　　assertThat(actual, nullValue());
 	
　　// 空文字の場合
　　assertThat(actual, is(""));
}
	public void testSampleMethod_null
	{
	　　// null の場合
	　　assertThat(actual, nullValue());
	}
	
	public void testSampleMethod_空文字
	{
	　　// 空文字の場合
	　　assertThat(actual, is(""));
	}

- テストデータのファイルは、テストクラスと同じパッケージ位置に配置する。

説明	テストデータのファイルは、テストクラスと同じパッケージ位置に配置する。 ただし、他のテストクラスと共通で使うものや、システム全体でマスタデータとなるようなデータファイルは、機能毎のパッケージ配下やルートパッケージに配置して良い。
動機	テストクラスとテストデータを同じパッケージ構成にしておくことで、メンテナンスがしやすくなるため。

違反サンプル	修正サンプル
src/main/java/jp/co/acroquest/app/ 　　Sample.java src/test/java/jp/co/acroquest/app/ 　　SampleTest.java data/SampleTest_method_データ数 10.xls	src/main/java/jp/co/acroquest/app/ 　　Sample.java src/test/java/jp/co/acroquest/app/ 　　SampleTest.java src/test/resouces/jp/co/acroquest/app/ 　　SampleTest_method_データ数 10.xls

9.2 命名規則

- テストコードのクラス名は、ソースコードのクラス名の末尾に”Test”を付与した名前とする。

説明	テストコードのクラス名は、ソースコードのクラス名の末尾に”Test”を付与した名前とする。
動機	ファイル名を見ただけで対応付けがすぐ分かるようにするため。

違反サンプル	修正サンプル
src/main/java/jp/co/acroquest/app/ Sample.java src/test/java/jp/co/acroquest/app/ TestSample.java	src/main/java/jp/co/acroquest/app/ Sample.java src/test/java/jp/co/acroquest/app/ SampleTest.java

- モッククラスは、ソースコードのクラス名の末尾に”Mock”を付与した名前とする。

説明	テストコードを作成する上でモックを作成する場合は、ソースコードのクラス名の末尾に”Mock”を付与した名前とする。
動機	ファイル名を見ただけで、テストコードと区別し、かつ、何のソースコードに対するモックなのかがすぐ分かるようにするため。

違反サンプル	修正サンプル
src/main/java/jp/co/acroquest/app/ Sample.java src/test/java/jp/co/acroquest/app/ Sample1.java	src/main/java/jp/co/acroquest/app/ Sample.java src/test/java/jp/co/acroquest/app/ SampleMock.java

- メソッド名は「テスト対象メソッド名＋”_” ＋ テストケース内容」とし、テストケース内容は日本語で記述する

説明	メソッド名は「テスト対象メソッド名＋”_” ＋ テストケース内容」とし、テストケース内容は日本語で記述する。
動機	連番だけを付けられたテストメソッドでは、何をテストしているのかが分かりにくいため。 Java では、日本語名のメソッドも利用可能であるが、その特性を活かし、テストケースの内容を日本語で書くことで、何のテストをしているのか、すぐ分かるようにする。 ※ソースコードで、日本語メソッドは利用しないこと。

違反サンプル	修正サンプル

public void testMethod_1-1-1()
{
// 処理
}
 	
public void testMethod_1-1-2()
{
// 処理
}
	public void testMethod_パスワードが null()
	{
	// 処理
	}
	
	public void testMethod_パスワード長短い()
	{
	// 処理
	}

- テストデータのファイル名は、「テストクラス名 ＋ ”_” ＋ テストメソッド名＋”_” ＋ テストケース内容」とする。

説明	テストデータのファイル名は、「テストクラス名 ＋ “_” ＋ テストメソッド名＋”_” ＋ テストケース内容」とする。
動機	ファイル名を見ただけで対応付けがすぐ分かるようにするため。

違反サンプル	修正サンプル
src/main/java/jp/co/acroquest/app/ 　　Sample.java src/test/java/jp/co/acroquest/app/ 　　SampleTest.java src/test/resouces/jp/co/acroquest/app/ 　　テストデータ_ケース 10.xls	src/main/java/jp/co/acroquest/app/ 　　Sample.java src/test/java/jp/co/acroquest/app/ 　　SampleTest.java src/test/resouces/jp/co/acroquest/app/ 　　SampleTest_method_データ数 10.xls

9.3 テストケースの作成

- モッククラスが必要な場合は、Mockito 等モックフレームワークを使って作成する。

- (JUnit4.4～) テストケースの判断には、”assertThat”を使用する。
  - “assertThat”第二引数には、”org.hamcrest.Matcher<T>”を指定する。
  - assertEquals と assertThat では、引数へ指定する”期待値”と”実際の値”の順番が変わる点に注意する。

(～ JUnit4.3) assertEquals	(JUnit4.4～) assertThat

// 左が期待値、右が実際の値。
assertEquals(expectedValue, actualValue);
	// 左が実際の値、右が期待値。
	assertThat(actualValue, is(expectedValue));

9.4 テストケースの維持

テストケースは、一度作成して終わりになるものではなく、常にテストがパスする状態を維持することが必要である。以下の点に注意して、テストケースを維持する。

- いつでもテストがパスするようにする。
  - 構成管理リポジトリにコミットする前に、テストケースを実行し、テストがパスすることを確認する。
- どこでもテストがパスするようにする。
  - 構成管理リポジトリからファイルを取得したら、即実行できる状態にする。テストケースを実行する前に、事前準備をしなくてはならない状況は避けること。
  - 環境に依存しないように、テストケースを作成する（絶対パスの利用は避ける、環境変数に依存しない、など）。どうしても環境に依存するような処理がある場合は、テストケース内で、自動でその環境になるように、事前処理を作成する。
  - どのマシンで実行しても、同じ結果が得られるようにする。
  - 乱数を使用したテストは実行毎に結果が変わる可能性があるため、テストがパスすることを十分に確認すること。

9.5 テストケースの作成が難しいもの

テストケースの作成が難しい場合に、無理にテストケースを作成しようとすると、手間がかかり、非効率となる可能性がある。以下のようなものは、注意が必要である。

- 現在の日付が関係する処理
  - テストデータを自動で変更できるようにしておくことを考える。
- マルチスレッドの処理
  - 別途、手動で試験することが必要。
- ファイルのパスなどが影響するもの
  - テストケース内で、ファイル位置の変更や内容の変更を行なう。
- xUnit でのテストが考慮されていないもの
  - リファクタするか、xUnit のテストをあきらめる。
- 画面の処理
  - GUI（Swing、HTML 等）の xUnit フレームワークも多く存在するため、それらを使う。ただし、画面のキャプチャは残せないもの多いため、その場合は、別途手動で試験が必要
- メール送信処理
  - メール送信ライブラリやメールサーバのモックを作成し、送信内容を確認できるようにする。

10 配列/コレクション

10.1 配列

- 配列の操作は、java.lang.System、java.util.Arrays クラスを利用すること。

説明	配列の操作は、java.lang.System クラスや、java.util.Arrays クラスのメソッドを利用すること。 コピー ： System#arraycopy()、Arrays#copyOf() ソート ： Arrays#sort() 検索 ： Arrays#binarySearch() また、Jakarta Commons Collections や Google Collections を利用すると、拡張された便利な API を利用することが可能であるため、必要に応じて導入を検討すること。 Jakarta Commons Collections http://commons.apache.org/collections/ Google Collections http://code.google.com/p/google-collections/
動機	配列に対してコピーや検索を行う場合、自前で行うとパフォーマンスが低下しやすいため、Java が標準で利用している API を利用すること。

違反サンプル

```
// 配列のコピー
for(int index = 0; index < ids.length; index++)
{
	ids2[index] = ids[index];
}

// 要素の検索
int result = -1;
for(int index = 0; index < array.size(); index++)
{
	if (value == array[index])
	{
		result = index;
		break;
	}
}
```

修正サンプル

```
// 配列のコピー
String[] ids2 = Arrays.copyOf(ids,ids.length);

// 要素の検索
// バイナリサーチはソートされていることが前提
Arrays.sort(array);
int result =  Array.binarySearch(array, value);
```

- null ではなく、サイズ 0 の配列を返すこと。

説明	配列を戻り値にする場合、その配列が null となる場合は、サイズ 0 の配列を返すようにする。
動機	呼び出し元で null チェックの実装が不要であり、NullPointerException の発生も予防できるため。

違反サンプル	**修正サンプル**
int[] array = null; return array;	int[] array = new array[0]; return array;

- パフォーマンスが特に重要となる複数要素の取得処理では、配列を利用すること。

説明	パフォーマンスが特に重要な部分には、配列を利用すること。 通常は、ArrayList を利用すれば十分であるが、特に高パフォーマンスを求められる場合には、配列の利用を検討する。
動機	単純に要素を取得するだけの処理では、コレクションフレームワークの List よりも、配列で処理する方が高速であるため。

違反サンプル	**修正サンプル**

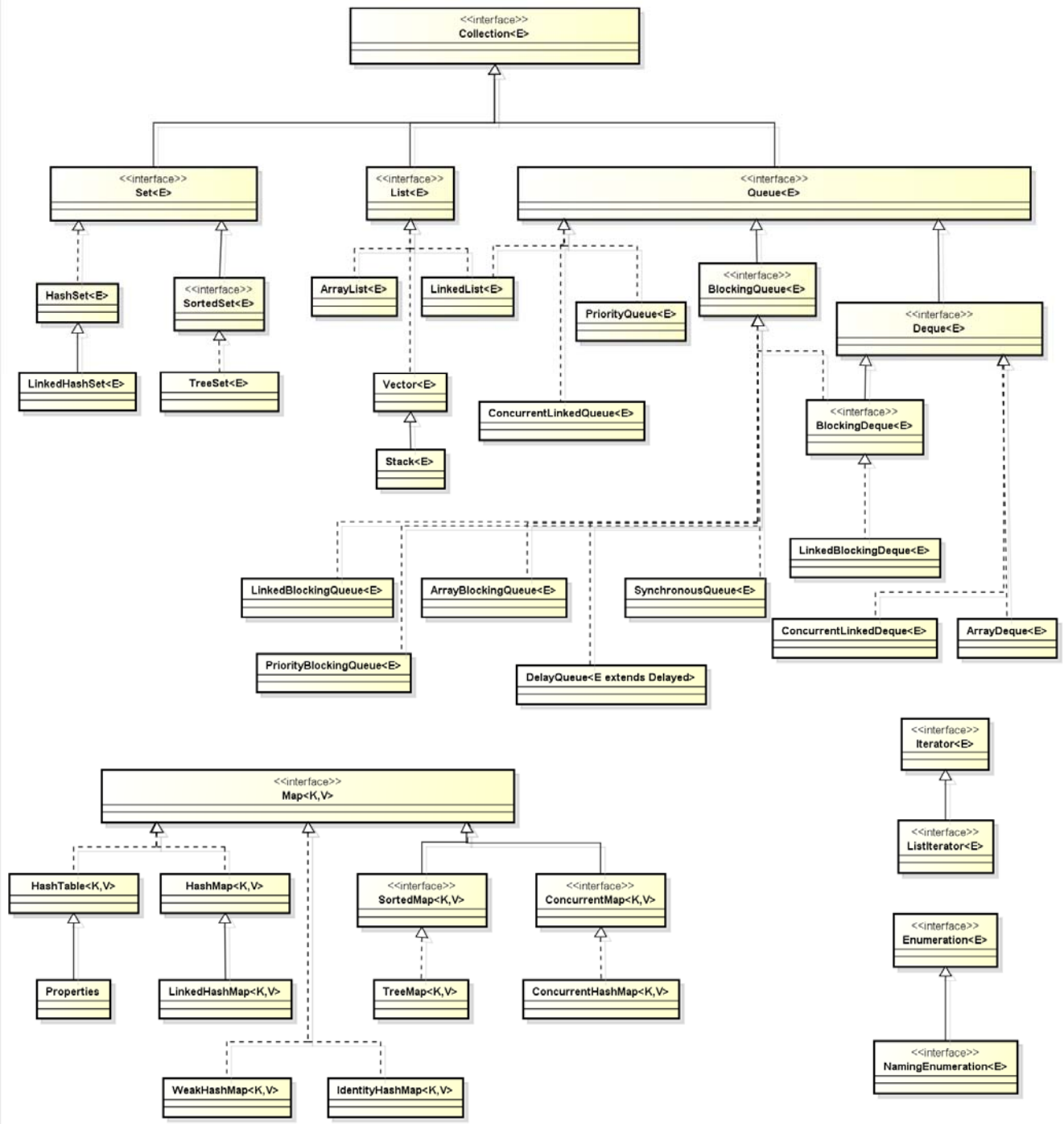
List list = getList();
for(int index = 0; index < list.size(); index++)
{
Object obj = list.get(index);
}
	Object[] array = getArray();
	for(int index = 0; index < array.length; index++)
	{
	Object obj = array[index];
	}

10.2 コレクション

JavaSDK では、複数の要素を保持するために、用途や特性に合わせたコレクションクラスが提供されている。要件に応じて、適切にコレクションクラスを選択することで、効率的なプログラムの作成が容易になる。

それらのコレクションクラス全体をまとめて、コレクションフレームワークと言う。

図 10-1 コレクションフレームワーク



- 原則、Vector、Hashtable クラスは利用しない。

説明	原則、Vector、Hashtable は利用せず、代わりに、ArrayList や HashMap クラスを利用する。
動機	これらのクラスは、JavaSDK1.1 以前で利用されていたものであり、パフォーマンスが低いため。スレッドセーフであるという特徴はあるが、大抵の場合はスレッドセーフを考慮しなくても良い箇所での利用が多く、また、スレッドセーフへの考慮は、メソッドやクラス単位で考慮するべきであるため。

違反サンプル	**修正サンプル**
Vector vector = new Vector(); Hashtable hashTable = new Hashtable();	List list = new ArrayList(); Map map = new HashMap();

- 変数の型には、原則として、インタフェースを利用する。

説明	変数の型には、原則として、インタフェースを利用すること。
動機	具象クラスが差し替えとなった場合でも、変更の影響を最小にするため。

違反サンプル	**修正サンプル**
ArrayList list = new ArrayList(); HashMap map = new HashMap();	List list = new ArrayList(); Map map = new HashMap();

- コレクションの操作は、java.util.Collections クラスを利用すること。

説明	コレクションの操作は、java.util.Collections クラスのメソッドを利用すること。 ソート ： Collections#sort() 検索 ： Collections#binarySearch() また、Jakarta Commons Collections や Google Collections を利用すると、拡張された便利な API を利用することが可能であるため、必要に応じて導入を検討すること。 Jakarta Commons Collections http://commons.apache.org/collections/ Google Collections http://code.google.com/p/google-collections/
動機	コレクションに対してソートや検索を行う場合、自前で行うとパフォーマンスが低下しやすいため、Java が標準で利用している API を利用すること。

違反サンプル

```
// 要素の検索
int result = -1;
for(int index = 0; index < list.size(); index++)
{
    if (value == list.get(index))
    {
        result = index;
        break;
    }
}
```

修正サンプル

```
// 要素の検索
// バイナリサーチはソートされていることが前提
Collections.sort(list);
int result =
    Collections.binarySearch(list, value);
```

- null ではなく、サイズ 0 のコレクションを返すこと。

説明	コレクションを戻り値にする場合、そのコレクションが null となる場合は、サイズ 0 のコレクションを返すようにする。
動機	呼び出し元で null チェックの実装が不要であり、NullPointerException の発生も予防できるため。

違反サンプル

```
List list = null;

return list;
```

修正サンプル

```
List list = new ArrayList();

return list;
```

- Generics
  - クラスやインタフェースをパラメータとして扱うことが可能になり、キャストが不要になるため、型に安全なコーディングが可能となる。
  - Generics に対応したクラスやメソッドを独自に実装することも可能である。
  - Collection フレームワークを使用する際は、必ず Generics を利用する。

手動による型変換	Generics を利用した処理

List list = new ArrayList();
list.add("Hello World!");
 	
// キャストが必要
String str = (String) list.get(0);
	List<String> list = new ArrayList<>();
	list.add("Hello World!");
	
	// キャストが不要
	String str = list.get(0);

パラメータの型に依存する処理	Generics を利用した処理

// これまでは処理が同じでも型が異なれば、
// クラスやメソッドを分ける必要があった。
 	
public class SampleCommand
{
　public void execute(Integer value)
　{
　　System.out.println(value);
　}
 	
　public void execute(Double value)
　{
　　System.out.println(value);
　}
}
	// Generics によりクラスをパラメータ化することで、
	// 処理をまとめることができる。
	
	public class SampleCommand<T>
	{
	　public void execute(T param)
	　{
	　　System.out.println(param);
	　}
	}

- (JavaSE 7.0～) ダイヤモンドオペレータ
  - new する時、型を冗長に記載せずに済むようになった。記載が簡潔化されるため推奨する。

(～JavaSE 6.0) 旧 Generics 記法	(JavaSE 7.0～) 新 Generics 記法
// 両辺に<String>の記述が必要。 List<String> list = new ArrayList<String>();	// 左辺のみ<String>の記述が必要。右辺は<>。 List<String> list = new ArrayList<>();

11 ロギング

ロギング処理については、SLF4J+Logback といった、ロギングライブラリを利用する。これらのライブラリを利用することにより、高度なロギング処理を簡単に実現できるようになる。

- SLF4J
  http://www.slf4j.org/

- Logback
  http://logback.qos.ch/

標準では、上記 2 つを組み合わせて利用することを標準とするが、これらのライブラリをラップするかたちで、独自にロギングライブラリを作成することも可能とする。

11.1 ログ規約

- Logger の変数は、static final とすること。

説明	ログ出力を行うクラス（通常は Logger クラス）のスコープは、static final とする。 ただし、変数名を大文字にする必要は無い。
動機	Logger クラスは、変更することはないため、static final として、高速にアクセスできるようにするが、定数ではないため。

違反サンプル	**修正サンプル**
private static fnal Logger LOGGER =LoggerFactory .getLogger(Sample.class);	private static fnal Logger logger__ =LoggerFactory .getLogger(Sample.class);

- メソッドの IN/OUT の箇所でログ出力は行わないこと。

説明	メソッドの IN/OUT の箇所で、デバッグログを出力することはしない。 ただし、外部インタフェースとなるような箇所で、必ずログを出力させた方が良いと判断した箇所については、その限りではない。
動機	全てのメソッドの IN/OUT でログを出力するようにするのは、実装上非効率であり、かつ、ログも過度に出力されるため。

違反サンプル	**修正サンプル**

private void sampleMethod()
{
logger__.debug(“処理開始”);
 	
// 処理
 	
logger__.debug(“処理終了”);
}
	private void sampleMethod()
	{
	// 削除
	
	// 処理
	
	// 削除
	}

- デバッグログ出力時には isDebugEnabled()を使うこと。

説明	デバッグログの部分には、isDebugEnabled()メソッドを使い、無駄な負荷をかけないようにする。 INFO、WARNING、ERROR などのレベルにも同様のメソッドがあるが、パフォーマンス上問題なければ、isXxxEnabled()メソッドを利用する必要はない。
動機	運用時のログレベルは、通常 DEBUG レベルではないが、デバッグログの部分でログメッセージを生成するために、無駄に負荷がかかってしまうため。

違反サンプル

```
String errorMsg = String.format(
          "データ読み込み :%s",
          obj.toString());
logger__.debug(errorMsg);
```

修正サンプル

```
if (logger_.isDebugEnabled())
{
      String errorMsg = String.format(
            "データ読み込み :%s",
            obj.toString());
      logger__.debug(errorMsg);
}
```

- ログ出力部分で、例外が発生しないように注意すること。

説明	ログ出力部分で、例外が発生しないように注意すること。
動機	ログ出力部分で、例外の発生を考慮していないなかったために、運用で例外が発生してしまい、処理が継続できないことがあるため。

違反サンプル

```
try
{
      string param = obj.getParam();

      // 処理
}
catch(IOException ex)
{
      // obj が null の可能性がある
      logger_.warn(obj.toString(), ex);
}
```

修正サンプル

```
try
{
      string param = obj.getParam();

      // 処理
}
catch(IOException ex)
{
      if (obj != null)
      {
            logger_.warn(obj.toString(), ex);
      }
      else
      {
            logger_.warn("obj = null", ex);
      }
}
```

11.2 ログ出力方針

11.2.1 ログを出力すべき箇所

- 状態遷移を行う処理は、ログを出力すること。
  - 状態遷移を行う処理は問題が発生しやすいため、ログにより問題をデバッグできるようにするため。
- 他システムや装置などと連携する部分となる外部インタフェースでは、ログを出力すること。
  - 通信ログを出力しておくことで、問題が発生した際の原因の切り分けが容易になるため。

11.2.2 ログを出力すべきではない箇所

- for や while など、ループ処理の中では必要以上にログ出力を行わないこと。
  - ループの回数分ログ出力を行うと、パフォーマンスの低下・ログの肥大化・ログ可読性の低下を招くため。
- Utility クラスでのログ出力は行わないようにすること。
  - 利用頻度の高い Utility クラスでログ出力を行うと、パフォーマンスの低下や再利用性の低下を招くことになるため。
  - ログ出力は、呼び出し元のクラスで行うようにする。

11.2.3 オブジェクトの内容のログ出力

- JavaBeans となるデータクラスは、ToString メソッドを実装し、オブジェクトの内容がログに出力されるようにする。
  - Apache Commons Lang に含まれる ToStringBuilder クラスなどを利用すると良い。
    http://commons.apache.org/lang/

11.3 ログレベルの動的更新

- ロギングライブラリ Logback ではログ出力設定内容を一定周期で再読み込みするオプションがある。
  - 例えばアプリケーション動作中、解析用にログレベルを DEBUG 変更するために使用する。logback.xml へ configuration タグの scan 属性を有効にし、scanPeriod で読み込み周期を設定しておく。

定期再読み込みを行わない	定期再読み込みを行う
// 設定の定期再読み込みを行わない。 <configuration> ...(各種設定) </configuration>	// 60 秒毎に設定の再読み込みを行う。 <configuration scan="true" scanPeriod="60 seconds" > (各種設定) </configuration>

12 コーディングテクニック

12.1 ソリッドコーディング

高品質なプログラムを生み出すためには、バグを発生させないための堅牢なコードの記述を、日頃から習慣化しておくことが必要である。

12.1.1 誤った処理の防止

- **オブジェクト、String の比較は、equals メソッドで行う。**
  - オブジェクトは、「==」や「!=」では、同一インスタンスかどうかを比較しており、内容の比較を行っていないため。

違反サンプル	修正サンプル

String str1 = “test”;
String str2 = “test”;
 	
// 実装ミス
// false となる
if (str1 == str2)
{
　　// 処理
}
	String str1 = “test”;
	String str2 = “test”;
	
	// 正しい判定
	// true となる
	if (str1.equals(str2))
	{
	　　// 処理
	}

- **String の空文字チェックは、isEmpty メソッドで行う。**
  - Apache Commons Lang の isEmpty は空文字チェックだけでなく、null チェックも行う（Java 標準の isEmpty は null チェックを行わない）。コードが簡潔になるため、Apache Commons Lang のライブラリを使える場合はそちらを使う。
    http://commons.apache.org/lang/

(JavaSE 6.0～)Java 標準	Apache Commons Lang

// null もしくは空の場合、処理を中断する。
if (inputStr == null || inputStr.isEmpty())
{
　　System.out.println("Null or Empty");
　　return;
}
	// null もしくは空の場合、処理を中断する。
	if (StringUtils.isEmpty(inputStr))
	{
	　　System.out.println("Null or Empty");
	　　return;
	}

- **equals メソッドを実装する場合は、hashCode メソッドも実装する。**
  - HashMap など、ハッシュ値で処理を行う場合に、等価と判断される 2 つのインスタンスでは、同じハッシュ値を返す必要があるため。
  - Apache Commons Lang に含まれる EqualsBuilder/HashCodeBuilder クラスなどを利用すると良い。
    http://commons.apache.org/lang/

自前で実装(equals)

```
@Override
public boolean equals(Object obj)
{
	if (obj == null)
	{
		return false;
	}
	if (obj == this)
	{
		return true;
	}
	else if (obj.getClass() != getClass())
	{
		return false;
	}
	StrCheck target = (StrCheck) obj;

	if (super.equals(obj) == false)
	{
		return false;
	}
	if (this.field1 != target.field1)
	{
		if (this.field1 == null
		   || target.field1 == null)
		{
			return false;
		}
		else if (
		this.field1.equals(target.field1)
		  == false)
		{
			return false;
		}
	}
	if (this.field2 != target.field2)
	{
		if (this.field2 == null
		   || target.field2 == null)
		{
			return false;
		}
		else if (
		this.field2.equals(target.field2)
		  == false)
		{
			return false;
		}
	}

	return true;
}
```

Apache Commons Lang(equals)

```
@Override
public boolean equals(Object obj)
{
	if (obj == null)
	{
		return false;
	}
	if (obj == this)
	{
		return true;
	}
	else if (obj.getClass() != getClass())
	{
		return false;
	}
	StrCheck target = (StrCheck) obj;

	boolean isSame = new EqualsBuilder()
	   .appendSuper(super.equals(obj))
	   .append(this.field1, target.field1)
	   .append(this.field2, target.field2)
	   .isEquals();

	return isSame;
}

// 性能は低下するが、以下のように簡単に記述することもできる。
// @Override
// public boolean equals(Object obj)
// {
//	return EqualsBuilder.reflectionEquals(
//		this, obj);
// }
```

自前で実装(hashCode)	Apache Commons Lang(hashCode)

public int hashCode()
{
// 37 は適当に決めた class ごとの
// ユニーク値。
int resultCode = field1.hashCode();
resultCode = 37 * resultCode
+ field2.hashCode();
return resultCode;
}
	public int hashCode()
	{
	// 17,37 は適当に決めた class ごとの
	// ユニーク値。
	int resultCode =
	new HashCodeBuilder(17, 37).
	append(field1).
	append(field2).
	toHashCode();
	return resultCode;
	}
	
	// 以下のように簡単に記述することもできる。
	// public int hashCode()
	// {
	// return HashCodeBuilder
	// .reflectionHashCode(this);
	// }

- **変数の使いまわしは行わない。**
  - 一度宣言した変数を複数の目的で安易に使いまわすと、変数の値を変更したことによる影響が、想定外の場所で発生するため。

- **Integer#parseInt は、基数を指定するメソッドを利用する。**
  - Integer#parseInt メソッドは、以下の 2 つのメソッドが存在するが、十進数/十六進数などの基数の取り違いを防止するため、基数を指定するメソッドを利用する。

違反サンプル	修正サンプル

// デフォルトでは、
// 十進数として変換するが、
// 十六進数として変換すべきときに、
// 誤る可能性がある
int val = Integer.parseInt(str);
	// 十進数として変換
	int val = Integer.parseInt(str, 10);
	
	// 十六進数として変換
	int val = Integer.parseInt(str, 16);

- **ストリームは、必ず finally で close する。**
  - close されないストリームが存在すると、システムリソースが枯渇していくためである。
  - ストリームの open と close の処理は、同一クラス内で行うような構成にしておくと、実装漏れが発生しにくい。
  - (JavaSE 7.0～) Java7 より追加された表記法 try-with-resources により、try 括弧内で宣言した変数に対し自動で close されるようになった。記述を簡潔にできるため、推奨する。

(～JavaSE 6.0)close 処理

```
File target =
      new File("/opt/target/file/path.txt");

InputStream input = null;
try
{
      input = new FileInputStream(target);
      // input に対する処理
}
catch (IOException ex)
{
      System.err.println(ex.getMessage());
}
finally
{
      try
      {
            if (input != null)
            {
                  input.close();
            }
      }
      catch (IOException ioe)
      {}
}
```

(JavaSE 7.0～)try-with-resources

```
File target =
      new File("/opt/target/file/path.txt");

try (InputStream input = new
FileInputStream(target))
{
      // inputに対する処理
}
catch (IOException ex)
{
      System.err.println(ex.getMessage());
}
```

- **InputStream#read(byte[] b)/read(byte[] b, int off, int len)では、戻り値を利用して、読み込んだバイト数を意識した処理を行う。**
  - 指定したバッファのバイト数分だけ、ストリームからデータを読み込めるとは限らないため。

12.1.2 異常系の実装漏れの防止

- **引数の null チェックや値の範囲チェックを省略しない。**
  - 「ここは null にはならないだろう」と安易に考え、引数の null チェックを怠る場合が多いが、自分が想定している処理漏れや仕様変更により、NullPointerException や IndexOutOfBoundsException などが発生する。
  - 特に、public のメソッドは、きちんと null チェックや値の範囲のチェックを行う。

- **文字列の比較では定数を起点する。**
  - 定数を起点にすることで、NullPointerException の発生を予防できるため。

違反サンプル	修正サンプル

final String CONST = ”test”;
 	
// 変数を起点
if (str.equals(CONST) == true)
{
　　// 処理
}
	final String CONST = ”test”;
	
	// 定数を起点
	if (CONST.equals(str) == true)
	{
	　　// 処理
	}

- **複数の条件判定は null チェックを最初に行う。**
  - null チェックを最初に行えば、その条件が真の場合は、その後の条件は判定されず、NullPointerException の発生を予防できるため。

違反サンプル	修正サンプル

// 例外が発生する可能性がある
if (item.getPrice() < 1000)
　　&& (item != null))
{
　　// 処理
}
	// item が null の場合は、
	// 2 つ目の条件は評価されないため、
	// 例外が発生しない
	if ((item != null)
	　　&& (item.getPrice() < 1000))
	{
	　　// 処理
	}

- **文字列→数値の変換は、例外処理を省略しない。**
  - Integer#parseInt()メソッド等、文字列を数値へ変換する処理をする場合は、RuntimeException のサブクラスである NumberFormatException が発生するため、きちんと例外を try～catch で処理する。
  - null や空文字を変換しようとして、NumberFormatException が発生することが多い。

- **例外を無視しない。**
  - try～catch で、catch した例外に対して、何も処理をせずに、処理を継続しないこと。
  - ログ出力を行うか、例外をスローして、呼び出し元のクラスで例外のハンドリングを行うようにする。

- **例外が発生して、処理が停止しないようにする。**
  - main メソッドからの呼び出し処理や、スレッド処理は、処理の起点となる部分で、try～catch で全ての例外（実行時例外を含む）をハンドリングするようにする。

- **(JavaSE 7.0～) 例外処理が共通となる場合、マルチキャッチする。**
  - JavaSE 7.0 から、一つの Catch 節に"|"区切りで複数の Exception を書くことができるようになった。
  - 簡潔に記述できるため、推奨する。

(～JavaSE 6.0) 通常の例外処理	(JavaSE 7.0～) 例外処理のマルチキャッチ

catch (InterruptedException ex)
{
　　ex.printStackTrace();
}
catch (ExecutionException ex)
{
　　ex.printStackTrace();
}
	catch (InterruptedException
	　　| ExecutionException ex)
	{
	　ex.printStackTrace();
	}

12.1.3 保守性を高めるコーディング

- **変数のスコープはできる限り狭くする。**
  - スコープを限定することは、可読性を上げるだけでなく、実装誤りを防止することにも繋がるため。

- **ローカル変数は、利用する直前で宣言する。**
  - 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と、スコープが必要以上に広くなり、可読性が低くなったり、想定外の場所で誤って値を変更してしまったりするため。

- **ループ処理や条件分岐処理の中で、リターンしない。**
  - 構文上では処理が途中となる部分でリターンすると、必要な事後処理を行っていなかったり、条件分岐を間違ったりするため。
  - ただし、メソッドの開始部分で、null チェックなどの引数の検証を行う場合は、条件分岐の中でもリターンしても良い。ループ処理の場合は、break でループ処理を終了させるようにする。

- **同じ case 条件の switch 文を複数使用しない。**
  - 同じ case 条件の switch が 2 箇所以上現れたら，必ずポリモーフィズム，FactoryMethod，Prototype パターン等でリファクタリングすること。

違反サンプル	修正サンプル

switch (order.getOrderType())
{
　case 1:
　　setup1();
　　break;
　case 2:
　　setup2();
　　break;
}
 	
// do Common
switch (order.getOrderType())
{
　case 1:
　　execute1();
　　break;
　case 2:
　　execute2();
　　break;
}
	Factory cookFactroy =
	　FactoryCreator.getCookFactory();
	Cook cook = cookFactroy.create(order);
	cook.setup();
	
	// do Common
	cook.execute();

- **(JavaSE 7.0～) switch 文で文字列を使う場合、必ず null チェックを行う。**
  - 外部から渡された文字列に対して Switch を行うとき、if-else switch 構文で記載する。

違反サンプル	修正サンプル

switch (input)
{
　case "A":
　　break;
　default:
　　break;
}
	if (input == null)
	{
	　logger_.error("Fatal input");
	}
	else switch (input)
	{
	　case "A":
	　　break;
	　default:
	　　break;
	}

- **アノテーション**
  - クラス、メソッド、パラメータに対してメタデータを定義できる。
  - アノテーションを利用することで、定義ファイルなどを利用しなくても、プログラム内からアノテーションを読み取って、アノテーションの内容に応じた処理をすることが可能となる。
  - 特に、フレームワークやライブラリの開発では、積極的に利用する。
  - (JavaSE 7.0～)インタフェースを実装した際にも@Override アノテーションが利用可能となったため、積極的に使用する。

- **拡張 for 文**
  - 拡張 for 文を利用することで、ループの処理が完結に記述できるようになる。ループインデックスを利用する必要がない場合は、拡張 for 文を利用する。
  - 内部の処理は、Iterator でのアクセスとなる。

通常の for 文	拡張 for 文

List<String> list = new ArrayList<>();
 	
int size = list.size();
for (int index = 0; index < size; index ++)
{
　　String msg = (String)list.get(index);
　　System.out.println(msg);
}
	List<String> list = new ArrayList<>();
	
	for (String msg : list)
	{
	　　System.out.println(msg);
	}

- **enum**
  - パターンを表す定数には、タイプセーフとなるよう enum 型を積極的に利用する。
  - 定数を利用した場合、ハードコーディングで値を入力されたり、定数にない値を指定されたりする可能性があったが、enum 型を利用することで、特定のパターンしか指定できなくなる。

定数による指定	enum 型

public static final int RED = 1;
public static final int BLUE = 2;
public static final int YELLOW = 3;
	public enum Color
	{
	　　RED,
	　　BLUE,
	　　YELLOW,
	　　BLACK,
	　　WHITE
	}

- **Autoboxing/Auto-Unboxing**
  - プリミティブ型とそのラッパークラス型の変換を自動で行う。

手動による型変換	Autoboxing/Auto-Unboxing
List<Integer> list = new ArrayList<>(); list.add(Integer.valueOf(index));	List<Integer> list = new ArrayList<>(); list.add(10);

12.1.4 マルチスレッドに留意したコーディング

- **スレッドアンセーフなクラスを利用している場合は、同期化されていることを確認する。**
  - 利用頻度が高く、注意すべきクラスには以下のようなものがある。
    - ✓ java.text.SimpleDateFormat
    - ✓ java.util.Matcher（java.util.regex.Pattern はスレッドセーフである）
    - ✓ java.util.HashMap
    - ✓ java.util.ArrayList

- **マルチスレッドからアクセスされる Map は、ConcurrentHashMap（Java 1.5 以降）を使用する。**
  - マルチスレッド下の場合、排他制御を行うのは当たり前のことではあるが、排他制御を行わずにHashMapにアクセスした場合（正確には、putとgetを同時に行った場合）、無限ループが発生するため、特に注意が必要である。
  - Java.util.Collections.synchronizedMap を利用して、スレッドセーフなマップを取得することは可能だが、java.util.ConcurrentModificationException の発生は防げない。
  - 安全性、パフォーマンスを考慮すると、java.util.concurrent.ConcurrentHashMap を利用するのが適切である。

- **synchronized の範囲は最小限にする。**
  - パフォーマンスの低下やデッドロックの発生を招くため。

- **スレッド間の待ち/同期にはタイムアウトを設定する。**
  - 検討漏れの結果、無限停止に陥るリスクがあるため。
  - Object#wait や Thread#join などが該当する。

違反サンプル	修正サンプル

try
{
　　// タイムアウトなし
　　wait();
}
catch (InterruptedException ex)
{
　　// 処理
}
	try
	{
	　　// タイムアウトあり
	　　wait(60000);
	}
	catch (InterruptedException ex)
	{
	　　// 処理
	}

- ExecutorService を使っている場合、Future#get へタイムアウトを指定する。
  ExecutorService を使う場合、スレッドプールを再利用することでスレッド生成のコストを抑えられる。

(JavaSE 1.4～) Thread

```
Thread thread = new Thread(new Runnable()
{
	@Override
	public void run()
	{
		// 重い処理
	}
});

thread.start();
try
{
	// タイムアウト 60 秒を指定。
	thread.join(60000);
} catch (InterruptedException ie) {
	ie.printStackTrace();
}
```

(JavaSE 5.0～) Executorservice

```
ExecutorService pool =
Executors.newSingleThreadExecutor();
try
{
	Future<?> future =
	pool.submit(new Runnable()
	{
		@Override
		public void run()
		{
			// 重い処理
		}
	});
	try
	{
		// タイムアウト 60 秒を指定。
		future.get(60, TimeUnit.SECONDS);
	}
	catch (InterruptedException
		| ExecutionException ex)
	{
		ex.printStackTrace();
	}
	catch (TimeoutException timeoute)
	{
		System.err.println(
			timeoute.getMessage());
	}
}
finally
{
	pool.shutdownNow();
}
```

12.2 ハイパフォーマンスコーディング

コーディングを行う際は、常にパフォーマンスに留意したコードを作成するべきである。ひとつの処理では問題が発生しないものでも、パフォーマンスの低下が積み重なり、大きな問題を招くためである。

12.2.1 パフォーマンスに留意したコーディング

- **ハイパフォーマンスを求められる箇所では Autoboxing/Auto-Unboxing を使用しない。**
  - Autoboxing/Auto-Unboxing の導入により、プリミティブ型（int など）とそのラッパークラス型（Integer など）の変換が自動で行われるようになったが、内部では変換が行われるため、パフォーマンス低下を引き起こす可能性がある。ハイパフォーマンスを求められる箇所では Autoboxing/Auto-Unboxing を使用しないこと。
  - 特に、List や Map に安易に格納しないよう注意する。

違反サンプル	修正サンプル

int value = 1;
List<Integer> list
= new ArrayList<Integer>();
 	
// 内部では変換が行われている
list.add(value);
int temp = list.get(0);
	int value = 1;
	int[] array = new int[10];
	
	// 配列で処理する
	array[0] = value;
	int temp = array[0];

- **文字列から数値への変換には「parseXxx」メソッドを使用する。**
  - 「new Integer(“123”)」など変換とするよりも、変換が高速に処理できるため。

- **スレッド数を管理する。**
  - スレッドの生成・起動処理はコストが高いため、スレッドプールを利用する。

- **コレクションの初期サイズはデータ量を予測して設定する。**
  - コレクションは、オブジェクトを生成した時点で一定サイズの要素数を配列として確保する（ArrayList は 10 要素分）。そして、その要素数を超える際に、元のサイズの倍のサイズとしてオブジェクトを生成し直して、全要素の値をコピーする。これらの処理はコストが大きく、パフォーマンス低下を招きやすい
  - コレクションに大量の要素を追加することが分かっている場合は、パフォーマンス低下を防ぐために、あらかじめデータ量を予測して設定する。

- **フレームワーク/ライブラリを利用する際に、一度だけ、初期化やオブジェクトの取得を行えば良い処理を、複数箇所で行わない。**
  - 注意すべき処理には以下のようなものがある。
    - ✓ JNDI の lookup
    - ✓ Log4j の初期化
    - ✓ DI コンテナの初期化（Spring, Seasar2 など）

- StringBuffer の代わりに StringBuilder を使う。
  - StringBuffer はすべてのメソッドが synchronized になっており、同期化されている。そのため、コストの高い処理になっていた。
  - StringBuilder は同期化されないが高速であるため、ローカル変数として利用する場合は StringBuffer でなく、代わりに StringBuilder を利用する。利用方法は、StringBuffer とほぼ同じである。

12.2.2 ループ

ループ処理は、パフォーマンス低下の問題を引き起こしやすい箇所である。ループ処理を可能な限り最適化すべきである。

- **ループの階層はできるだけ少なくする。**
  - 3 階層以上のネストは、パフォーマンス問題に繋がる可能性が高いため、できるだけネストが深くならないように処理を整理する。
  - 1 つのメソッド内だけでなく、メソッドの呼び出し階層でループの階層が深くなっていないかにも注意する。

- **ループ内で、「+」での文字列連結を行わない（コストの高い処理）。**
  - 「”abc” + “xyz”」というような「+」での文字列連結は、ループ内では行わないこと。
  - StringBuffer（Java 1.4 以前）/StringBuilder（Java 1.5 以降）を利用する。

- **ループ内で、String クラスの split/concat/replace/replace/replaceAll/replaceFirst は利用しない（コストの高い処理）。**
  - 上記の API は、実行速度が低いため、ループ内では利用しないこと。
  - ループ外で処理する（実行回数が少ない）分には、あまり問題はないが、ループ内で処理する必要がある場合は、代替方法として以下のようなものがある。
    - ✓ String#split → java.util.regex.Pattern#split を利用する
    - ✓ String#concat → StringBuilder#append を利用する
    - ✓ String#replace → java.util.regex.Pattern#compile と java.util.regex.Matcher#matcher を組み合わせる
      ※replaceAll/replaceFirst も同様

- **ループ内で、キャスト処理を行わない（コストの高い処理）。**
  - instanceof を利用して、クラスの型をチェックした後、必要な箇所だけキャストする。

- **ループ内で、List#contains は利用しない（コストの高い処理）。**
  - リスト内の要素を線形検索することがあり、要素数が多い場合に時間がかかるため。
  - ループ内で処理する必要がある場合は、一度、Map に置き換えて検索する。

- **ループ内で、I/O 処理（ファイル/ネットワーク等の入出力）の open/close は極力行わない（コストの高い処理）。**
  - ストリームを open/close する処理はコストが高いため。

- **ループ内で、try～catch 処理は極力行わない（コストの高い処理）。**
  - if 文でチェックできないか、ループ処理外で try～catch できないかを検討する。

- **ループ内で、不要なオブジェクトの生成を行わない（コストの高い処理）。**
  - 本当にループ内で行う必要があるのかどうかを確認し、整理する。

- **ループ内で、不要なインスタンス変数の呼び出しを行わない（コストの高い処理）。**
  - インスタンス変数へのアクセスはコストが高いため、特にパフォーマンスが要求される処理では、ループ外で一度ローカル変数に格納することを検討する。

12.2.3 I/O

ディスクやネットワーク越しのデータアクセスは、メモリアクセスに比べて1万倍以上は遅い。バッファリング等の仕組みを積極利用すべきである。

- **バッファリングを行う。**
  - バッファリングを行っていない場合、パフォーマンス低下を引き起こす可能性がある。
  - BufferedInputStream/BufferedOutputStream を使用する。

- **無駄な直列化（シリアライズ）は行わない。**
  - シリアライズ対象のクラスのメンバはプリミティブ型を積極的に利用する。
  - シリアライズしなくてもよいメンバは transient を付け、シリアライズ対象外とする。

- **DB アクセスにはコネクションプーリングを利用する。**
  - コネクションプーリングは、フレームワークでサポートしていたり、ライブラリが存在したりするため、それらを活用する。

- **DB アクセスには、Statement ではなく、PreparedStatement を利用する。**
  - Statement よりも、PreparedStatement の方が高速に動作する。

13 Javaの新機能

13.1 JavaSE 6.0

本節では、JavaSE 6.0 で、新規に追加されたり、変更された API について説明する。
ただし、掲載しているものは、その全てではない。特に重要なものについてのみ、記述している。

13.1.1 新規API

JavaSE6.0 からの新規 API 一覧については、Oracle のリリースノートに記載されている。
http://www.oracle.com/technetwork/java/javase/features-141434.html

- java.util.Arrays#copyOf
  - 配列のコピーを行う。配列なので System より Arrays を使用した方が直観的に分かりやすい為、推奨する。
  - Arrays#copyOf では型が明示的に指定されるため、型違いによる Exception が発生しない。

(～JavaSE 5.0) System.arraycopy	(JavaSE 6.0～) Arrays.copyOf
String[] names = 　new String[] { "ringo", "mikan" }; // 事前にコピー先配列を宣言する必要がある。 String[] copyNames = new String[names.length]; System.arraycopy(　names, 0, copyNames, 0, names.length);	String[] names = 　new String[] { "ringo", "mikan" }; String[] copyNames = Arrays.copyOf(　names, names.length);

- java.io.File#getXXXSpace
  - File クラスへディスク容量を確認するメソッドが追加された。
    1. getTotalSpace()　:そのファイルの存在するパーティションのサイズを返却する。
    2. getFreeSpace()　:そのファイルの存在するパーティションの空きサイズを返却する。
    3. getUsableSpace():そのファイルの存在するパーティションの利用可能な空きサイズを返却する。
       ※getFreeSpace に比べ権限確認などを行い利用可能なサイズを返す。単位は Byte。

  - 権限を確認するメソッドも追加されている。
    1. setWritable(boolean writable)　　:書き込み権限を付与する。
    2. setReadable(boolean readable)　　:読み取り権限を付与する。
    3. setExecutable(boolean executable):実行権限を付与する。
    4. canExecute()　　:実行可能か確認する。
       ※Windows では setReadable, setExecutable は機能しない(必ず true)。

13.2 JavaSE 7.0

本節では、JavaSE 7.0 で、新規に追加されたり、変更された API について説明する。
ただし、掲載しているものは、その全てではない。特に重要なものについてのみ、記述している。

13.2.1 新規API

JavaSE7.0 からの新規 API 一覧については、Oracle のリリースノートに記載されている。
http://www.oracle.com/technetwork/java/javase/jdk7-relnotes-418459.html

- java.nio.file.Path/Files
  - JDK5.0 以前では File クラスで操作していた内容が、Path/Files クラスによりより便利に使えるようになった。テキストデータの全読み込み/書き込みの例を示す。

(～JavaSE 6.0) File

```
File inputFile =
      new File("/opt/input/file/path.txt");
File outputFile =
      new File("/opt/output/file/path.txt");

      InputStream input = null;
      Reader reader = null;
      BufferedReader br = null;

      OutputStream output = null;
      Writer writer = null;
      BufferedWriter bw = null;

      try
      {
            input = new FileInputStream(inputFile);
            reader    = new
            InputStreamReader(input, "UTF-8");
            br = new BufferedReader(reader);

            output = new
            FileOutputStream(outputFile);
            writer    = new
            OutputStreamWriter(output, "UTF-8");
            bw = new BufferedWriter(writer);

            String line = br.readLine();

            while(line != null) {
                  bw.write(line);
                  bw.newLine();
                  line = br.readLine();
            }
      }
      catch (IOException ioe)
      {
            System.err.println(ioe.getMessage());
      }
      finally
      {
            try
            {
                  if (input != null)
                  {
                        input.close();
                  }
            //各リソースへのcloseを行う(省略)。
            }
            catch (IOException ioe)
            {}
      }
```

(JavaSE 7.0～) Path/Files

```
Path inputFile =
      Paths.get("/opt/input/file/path.txt");
// テキストファイル全読み込み
List<String> lines = Files.readAllLines(
      inputFile, Charset.defaultCharset());

Path outputFile =
      Paths.get("/opt/output/file/path.txt");
// テキストファイル全書き込み
Files.write(outputFile,
                  lines,
                  Charset.forName("UTF-8"));
```

※バイナリ/ストリームの読み書きについても新しい API が提供されているが、ここでは割愛する。

- java.util.Objects#equals/deepEquals/hashCode/toString
  - オブジェクト操作に関する Utility クラスとして、Objects クラスが追加された。以下の主要 4 static メソッドを説明する。
    1. equals(Object a, Object b) : nullCheck を行った後に a.equals(b)を行う。
    2. deepEquals(Object a,Object b) : nullCheck を行った後、配列であれば Arrays.deepEquals を、そうでなければ a.equals(b)を行う。
    3. hashCode(Object o) : 引数が null ならば 0、それ以外ならば hashCode を返す。
    4. toString(Object o) : 引数が null ならば"null"、それ以外ならば toString を返す。
- java.util.concurrent.ThreadLocalRandom
  - java.util.Random はスレッドセーフでない。マルチスレッドで乱数生成する場合には、ThreadLocalRandom を利用する。

乱数生成	マルチスレッドでの乱数生成

// 乱数を生成する。
double randomDouble = Math.random();
	// スレッドごとに乱数を生成する。
	// 必ず current()を実行してから生成を行う。
	double threadRandomDouble = ThreadLocalRandom.current().nextDouble();

14 コードサンプル

```
/**
 * Copyright (c) Acroquest Technology Co, Ltd. All Rights Reserved.
 * Please read the associated COPYRIGHTS file for more details.
 *
 * THE SOFTWARE IS PROVIDED BY Acroquest Technolog Co., Ltd.,
 * WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING
 * BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
 * IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDER BE LIABLE FOR ANY
 * CLAIM, DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING
 * OR DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.
 */
package jp.co.acroquest.sample;

import java.util.LinkedList;

/**
 * int 値を格納するスタッククラス。<br>
 * int 型の値をスタックに格納し、LIFO を実現する。<br>
 *  このスタックは、スレッドセーフではない。
 *
 * @author Acro Taro
 * @version 1.0
 */
public class IntStack implements Stack
{
    /**  スタックの値を保持する連結リスト。  */
    private final LinkedList stack_ = new LinkedList();

    /**
     *  デフォルトコンストラクタ。
     */
    public IntStack()
    { }

    /**
     *  スタックから値を取得する。<br>
     *  スタックが空の場合は、EmptyStackException をスローする。
     *
     * @return  スタックから取得した値。
     */
    public int pop()
    {
        Integer valueObj = (Integer) this.stack_.removeLast();

        //  値が取得できなかった(スタックが空)場合は、EmptyStackException をスローする
        if (valueObj == null)
        {
            throw new EmptyStackException("stack is empty.");
        }

        int value = valueObj.intValue();

        return value;
    }
```

```
    /**
     * スタックに値を追加する。
     *
     * @param value  スタックに追加する値。
     */
    public void push(int value)
    {
        Integer valueObj = Integer.valueOf(value);
        this.stack_.addLast(valueObj);
    }

    /**
     * スタックに指定した値が含まれているか検索する。<br>
     * 見つかった場合は、最初の要素を 1 とした深さを返す。
     *
     * @param value  検索する値。
     * @return  見つかった場合は深さ。見つからなかった場合は-1。
     */
    public int search(int value)
    {
        Integer valueObj    = Integer.valueOf(value);
        int         findIndex = this.stack_.lastIndexOf(valueObj);
        int         depth         = -1;

        if (findIndex >= 0)
        {
            depth = findIndex + 1;
        }

        return depth;
    }

    /**
     * スタックが空であるかどうかを検査する。
     *
     * @return  スタックが空ならば true、値を持っていれば false。
     */
    public boolean isEmpty()
    {
        return this.stack_.isEmpty();
    }
}
```

15 特記事項

本書内で利用している単語の意味を以下に示す。

- Pascal 形式
  識別子の最初の文字と、後に続いて連結されている各単語の最初の文字を大文字にする。
  例）BackColor
- Camel 形式
  識別子の最初の文字は小文字にし、後に続いて連結されている各単語の最初の文字を大文字にする。
  例）backColor

16 謝辞

本書を作成するにあたり、以下のサイトで公開されているコーディング規約を参考にさせて頂きました。
ありがとうございます。

コーディング規約の会（オブジェクト倶楽部）
http://www.objectclub.jp/community/codingstandard/