Java コーディング規約 2004

履歴
初版 2004 年 9 月

このドキュメントは、(株)電通国際情報サービスが作成した Java コーディング規約です。
現在、PDF 版を用意しています。

このドキュメントは現状有姿にて無償で提供され、記載内容およびその利用と結果について一切の保証は適用されず、著作者は何ら責任を問われないものとします。

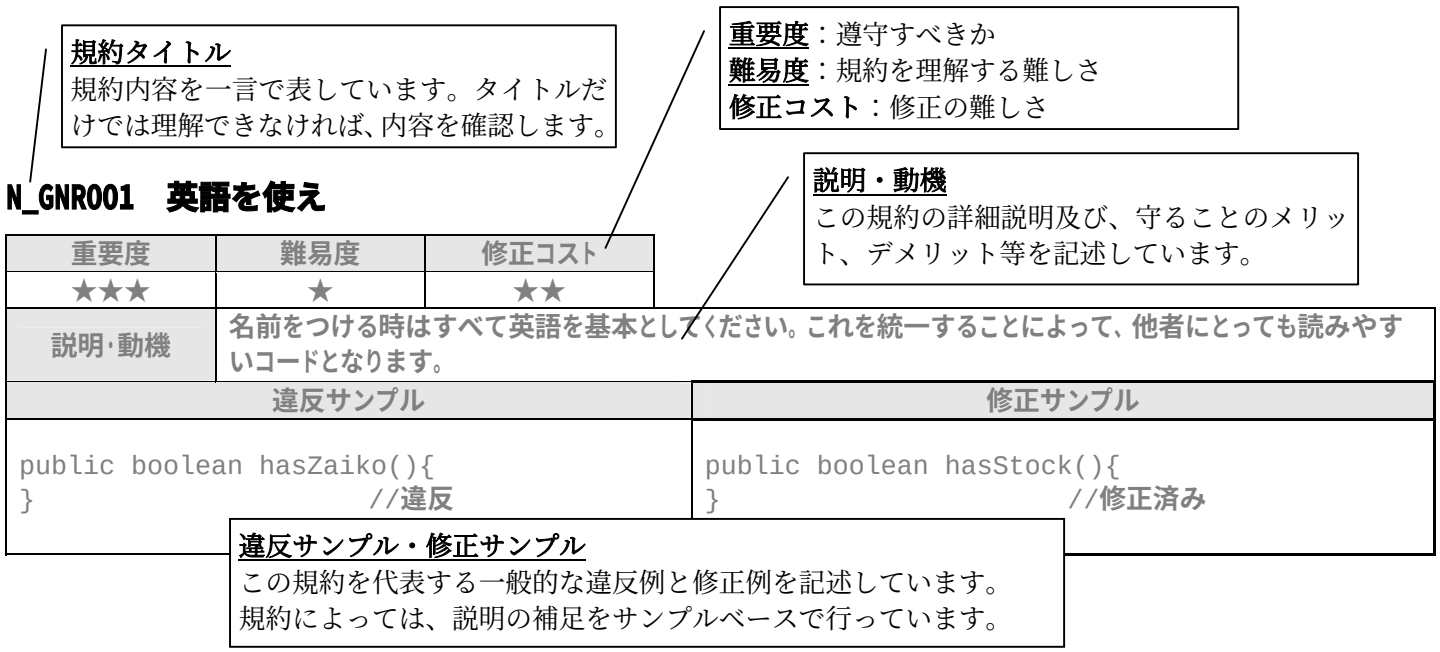
皆さんの Java を用いたシステム開発プロジェクトにおいて、開発効率化・システム運用効率化に資する目的や学究の目的でこれをコピー、修正、配布してかまいません。

強制ではありませんが、フッタの Copyright 表示を維持していただくことを希望します。ご意見などを以下のアドレスへ頂けるとさらに嬉しく思います。
java-coding-info@objectclub.jp

目次

0. 規約の見方

各規約は、下記のようなフォーマットで記されています。サンプルを元に見方を説明します。



N_GNR001　英語を使え

重要度	難易度	修正コスト
★★★	★	★★

説明･動機	名前をつける時はすべて英語を基本としてください。これを統一することによって、他者にとっても読みやすいコードとなります。

違反サンプル	修正サンプル
public boolean hasZaiko(){ }　　//違反	public boolean hasStock(){ }　　//修正済み

重要度：遵守すべきか

★★★：必須規約。守らないことにより、バグを内包する可能性がある規約。または、一般的な認知度が高い常識的な規約

★★：推奨規約。守らないことにより、保守性が下がる可能性のある規約。

★：エチケット規約。統一されていないことにより、特に可読性が下がる規約

難易度：規約を理解する難しさ

★★★：Java を使いこなしていないと理解しにくいレベルの規約

★★：Java の基礎を理解していれば理解できるレベルの規約。

★：Java 初心者でも容易に理解できるレベルの規約。

修正コスト：修正の難しさ

★★★：一定の Java のスキルを持ったプログラマが、他の個所への影響調査やロジックの変更も含めて一つ一つ修正する必要がある規約

★★：Eclipse 等のリファクタリング機能を用いて修正できる規約。または、一括置換はできないが、他の個所への影響が少なく、修正方法も明らかな規約。

★：テキストエディタ等を用いて一括変換すれば修正できる規約

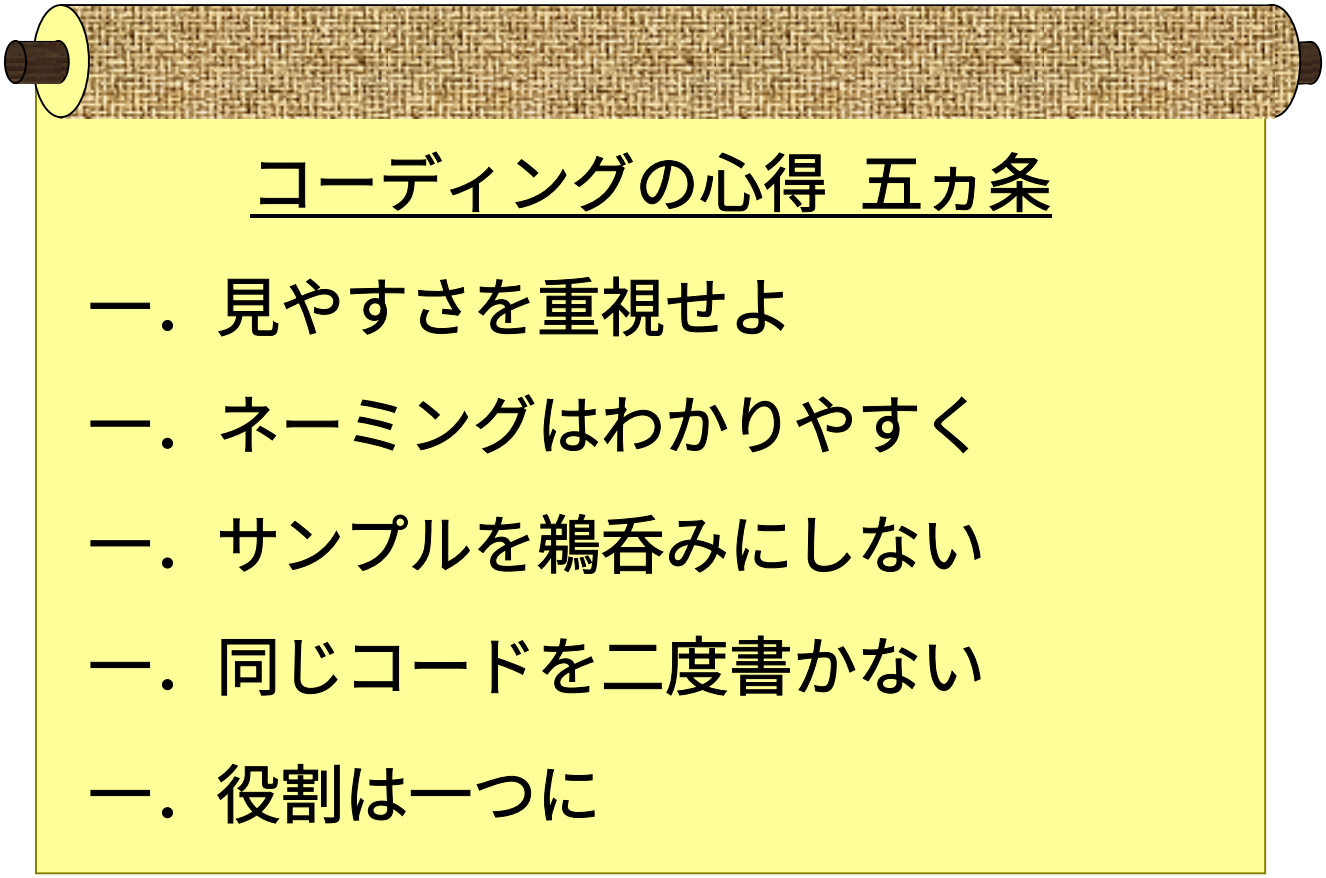
1. コーディングの心得 五か条

コーディングは、本来、非常に知的な作業です。

長いプログラムを記述すること(ステップ数)によって生産性が評価されたのは、過去の時代の出来事です。現在は、クラスやメソッドの役割が明確で、ロジックが読みやすく、保守性に優れたプログラムを記述することが評価されます。

コーディング規約は、コードの書き方に関する一種のパターンと言うこともでき、コードの保守性を向上させる具体的な方法を示しています。したがって、規約の一つ一つの意図を理解し、守ることが重要です。しかし、保守性に優れたコードを作成するためには、コーディング規約を守ることに加えて、良いコードを記述するための基本的な心構えをしっかり心に留めておく必要があります。

本章では、この心構えを「コーディングの心得 5 か条」として紹介します。
どの心得もごく当たり前に思えるかもしれません。しかし、これらの心得と照らして既存のコードを参照すると、実践されていないコードが意外と多いことに気づかれる方も多いのではないでしょうか。
特に、コーディング経験の浅いプログラマの方は、この「コーディングの心得 5 か条」を理解し、コーディング時に自分で考えて実践することをお勧めします。



◇見やすさを重視せよ

「良いコード」の基本は、「他の人が読んでもわかりやすいと感じられるコード」と言えます。コードの見やすさは、フォーマットはもちろん、ロジックの簡潔さや API の常識的な使い方などから生まれます。コーディングにあたっては、常に他の人の視点を意識しながら、見やすさに気を配って記述しましょう。
また、自分で記述したコードであっても、しばらくたってから読み返してみると理解に時間がかかった経験はないでしょうか。「3 日前に書いたコードは他人のコードと同じ」ということも言われます。見やすさを重視することは、他の人のためだけでなく、自分のためにもなります。

◇ネーミングはわかりやすく

コーディングでは、様々な変数やメソッドなどにネーミング(名前付け)しなければなりません。ネーミングとは、本来、その対象の本質を表すような名前を考える作業です。大変難易度の高い作業ですが、一方で適当に行ってもコードの動作は変わらないため、人によっては手を抜きがちです。しかし、ネーミングの良し悪しは、コードの可読性に非常に大きな影響を及ぼします。
例えば、「C0001」というクラス名があるとします。これでは、何を表すクラスなのかすぐにわかりません。また、「int p = 5000;」という記述があるとします。プログラマに聞くと、変数名 p は価格(Price)の略だと言ったとします。であれば略さずに、「int price = 5000;」としたほうがわかりやすいと思いませんか？
「ネーミングはわかりやすく」の背景には、読んで内容が理解できるという意味で、文章のようなプログラミングを行うという考え方があります。

◇サンプルを鵜呑みにしない

サンプルコードを活用すること自体は、著作権等を侵害しなければ問題ないでしょう。問題なのは、その内容や背景を理解しないまま、サンプルコードだけを鵜呑みにして、「おまじない」として表面的に適用してしまうことです。
コードを「おまじない」ととらえていては、サンプルコードの間違いを気づかないまま適用してしまうこともあります。
例えば、ストリームのクローズ処理を行っていないサンプルコードであっても、それに気づかずに自分のコードに適用してしまい、後で思わぬ障害を引き起こすということもありえます。サンプルコードは、そこで説明する内容に絞ったコードが多いため、このような例はよく見られます。
また、サンプルコードをそのまま適用した結果、自分が記述すべきコードには必要のないコードが含まれてしまうこともあります。その場合、コードの可読性を下げる原因になります。
自分のコードは、自分で深く理解して記述しましょう。

◇同じコードは二度書かない

コードをコピー・ペーストしていませんか？コピー・ペーストしてしまうと、何らかの修正をする際に、全ての個所に同じ修正をする羽目になります。同じコードが現れるようならまとめて一つにし、外に出してコールするような書き方をすべきです。
同じコードをまとめる作業は、どちらかといえば、コーディング時よりリファクタリング(ソフトウェアの外部的振る舞いを変更せずに内部構造を改善する作業)で行われることが多いでしょう。しかし、コーディング時からできるだけ気をつけておきたいことです。

◇役割は一つに

メソッドの役割が明確で、かつ 1 つであれば単体テストが行いやすくなります。つまり、コードの「試験性」が高まります。また、役割が一つであれば、後でコードを変更する際に修正箇所がわかりやすいため、障害修正に要する時間が短くなります。つまり、コードの「保守性」があがります。
例えば、「チェックをして実行する」機能を実現するために、checkAndDo()メソッドが存在したとします。このメソッドは check()メソッドと do()メソッドに分割すべきです。なぜなら、checkAndDo()メソッドのチェックロジックに誤りがあった場合、本来は do()メソッドに書かれる内容まで把握する必要があるからです。分割してあれば、check()メソッドだけの変更で済みます。
このことはクラスの設計にも言えることです。

2. ネーミング規約

2.1. 全般

N_GNR001 英語を使え

重要度	難易度	修正コスト
★★★	★	★★
説明･動機	名前をつける時はすべて英語を基本としてください。これを統一することによって、他者にとっても読みやすいコードとなります。	

違反サンプル	修正サンプル
`public boolean hasZaiko(){` `}  //違反`	`public boolean hasStock(){` `}  //修正済み`

N_GNR002 大文字･小文字の違いで名前を区別しない

重要度	難易度	修正コスト
★★★	★	★★
説明･動機	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。これを統一することによって、他者にとっても読みやすいコードとなります。	

違反サンプル	修正サンプル
`private int number;` `private int Number;  //違反`	`private int carNumber;` `private int trainNumber;  //修正済み`

2.2. パッケージ

N_PKG001 パッケージ名はすべて小文字に

重要度	難易度	修正コスト
★★★	★	★★
説明･動機	特に取り決めがない限り、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。	

違反サンプル	修正サンプル
`jp.co.isid.FrameWork.Banking.ACCOUNT` `//違反`	`jp.co.isid.framework.banking.account` `//修正済み`

N_PKG002 パッケージ名は意味のある名前に

重要度	難易度	修正コスト
★★★	★	★★

説明・動機	パッケージ名は、できるだけパッケージの内容が連想可能な名前にしてください。パッケージ名に機能 ID、連番等を用いると、すぐに意味がわかりません。この規約を守ることによって、他者にとっても読みやすいコードとなります。

違反サンプル	修正サンプル
`jp.co.isid.framework.banking.a00001 //違反`	`jp.co.isid.framework.banking.account //修正済み`

N_PKG003 パッケージ名は省略しない

重要度	難易度	修正コスト
★	★	★★

説明・動機	パッケージ名は、多少長くなってもできるだけ省略せず、わかりやすい名前を使用するようにしてください。

違反サンプル	修正サンプル
`jp.co.isid.fw.bkg.acc //違反`	`jp.co.isid.framework.banking.account //修正済み`

N_PKG004 サブパッケージ名の重複は可能

重要度	難易度	修正コスト
★★	★	★★

説明・動機	親パッケージ名が異なっているのであれば（FQDN が一意になるのであれば）、同一サブパッケージ名が存在してもかまいません。 FQDN（Fully Qualified Domain Name）：完全修飾ドメイン名

サンプル

```
jp.co.isid.framework.banking.account
jp.co.isid.framework.trading.account     //親パッケージ名が異なるので OK。
```

2.3. クラス

N_CLS007 クラス名は役割を表す名前にする

重要度	難易度	修正コスト
★★★	★	★★

説明・動機	クラス名には、機能 ID、連番等の意味のわかりにくい名前を使用しないでください。クラス名に意味のある文字列を使用することによって、クラスの内容が連想できるため、他者にとっても読みやすいコードとなります。

違反サンプル	修正サンプル
`public class S0001 {  //違反` `}`	`public class Sample {  //修正済み` `}`

N_CLS001 クラス名は単語の先頭を大文字にする

重要度	難易度	修正コスト
★★★	★	★★

説明・動機	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	修正サンプル
`public class sampleclass {  //違反` `}`	`public class SampleClass {  //修正済み` `}`

N_CLS002 例外クラス名には名前の最後に"Exception"をつける

重要度	難易度	修正コスト
★★★	★	★★

説明・動機	例外クラス名には、名前の最後に”Exception”をつけてください。これを統一することによって、他者にとっても読みやすいコードとなります。

違反サンプル	修正サンプル

`public class Sample extends Exception{`
`  //違反`
`}`
	`public class SampleException`
	`  extends Exception{`
	`  //修正済み`
	`}`

N_CLS003 インターフェース名はクラス名に準ずる

重要度	難易度	修正コスト
★★★	★	★★

説明・動機	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。これを統一することによって、他者にとっても読みやすいコードとなります。 例外：クラス名との区別が必要であれば、先頭に”I”をつけてください。

サンプル
`public class Sample implements ISample {` `}`

N_CLS006 抽象クラス名はクラス名に準ずる

重要度	難易度	修正コスト
★★★	★	★★

説明・動機	抽象クラス名の命名規則は基本的にクラス名に準じます。これを統一することによって、他者にとっても読みやすいコードとなります。 例外：適当な名前がないときは、”Abstract”から始め、その後にサブクラス名を連想させる名前を付けてください。

サンプル

```
abstract class AbstractSample {
}
```

N_CLS005 実装クラス名はクラス名に準ずる

重要度	難易度	修正コスト
★★★	★	★★

説明・動機	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。これを統一することによって、他者にとっても読みやすいコードとなります。 例外：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 {
}
```

N_CLS004 能力付加型のインターフェース名は最後に”able”をつける

重要度	難易度	修正コスト
★★	★	★★

説明・動機	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には、例えば Runnable, Cloneable 等があります。能力付加型のインターフェースを定義する場合は、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。これを統一することによって、他者にとっても読みやすいコードとなります。

サンプル

```
public class Sample implements Pluggable {
}
```

2.4. テストクラス

N_TCL001 テストクラス名は「テスト対象クラス名 + Test」にする

重要度	難易度	修正コスト
★★★	★	★★

説明・動機	テストクラス名は、「テスト対象クラス名 ＋ Test」としてください。これを統一することによって、他者にとっても読みやすいコードとなります。 以下のサンプルは、JUnit を意識したものです。

サンプル

```
public class SampleClassTest extends TestCase{
}
```

N_TCL002 すべてのテストやパッケージテストを行うクラス名は「AllTest」or「パッケージ名 + Test」にする

重要度	難易度	修正コスト
★★	★	★★

説明・動機	すべてのテストやパッケージテストを行うクラス名は、「AllTest」もしくは「パッケージ名 ＋ Test」としてください。これを統一することによって、他者にとっても読みやすいコードとなります。

サンプル

```
public class AllTest {
}
-----------------------------------------
package jp.co.isid.banking;
public class BankingTest {
}
```

2.5. メソッド

N_MTD001 コンストラクタと同じ名前のメソッドはつくらない

重要度	難易度	修正コスト
★★★	★	★★

説明・動機	コンストラクタと同じ名前のメソッドをつくってはいけません。これを統一することによって、他者にとっても読みやすいコードとなります。

違反サンプル

```
public class BadSample {
    public BadSample() {
    }
    public void BadSample(int number){
    }   //違反
}
```

修正サンプル

```
public class FixedSample {
    public FixedSample() {
    }
    public void setNumber(int number){
    }   //修正済み
}
```

N_MTD002　メソッド名は区切りのみ大文字にする

重要度	難易度	修正コスト
★★★	★	★★

説明・動機	メソッド名は、ひとつ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	修正サンプル
`public void Samplemethod() {` `}　　//違反`	`public void sampleMethod() {` `}　　//修正済み`

N_MTD003　オブジェクトを生成するメソッド名は「"create"+オブジェクト名」にする

重要度	難易度	修正コスト
★★★	★	★★

説明・動機	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。これを統一することによって、他者にとっても読みやすいコードとなります。

サンプル

```
public Sample createSample(){
}
```

N_MTD004　変換メソッド名は「"to"+オブジェクト名」にする

重要度	難易度	修正コスト
★★★	★	★★

説明・動機	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。これを統一することによって、他者にとっても読みやすいコードとなります。

サンプル

```
public Another toAnother(){
}
```

N_MTD005 ゲッターメソッド名は「"get"+属性名」にする

重要度	難易度	修正コスト
★★★	★	★★

説明・動機	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。これを統一することによって、他者にとっても読みやすいコードとなります。これは、JavaBeans の規約です。

サンプル

```
private String sampleName;
public String getSampleName(){
    return sampleName;
}
```

N_MTD006 セッターメソッド名は「"set"+属性名」にする

重要度	難易度	修正コスト
★★★	★	★★

説明・動機	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。これを統一することによって、他者にとっても読みやすいコードとなります。これは、JavaBeans の規約です。

サンプル

```
private String sampleName;
public void setSampleName(String name){
    sampleName = name;
}
```

N_MTD007 boolean 変数を返すメソッド名は true/false の状態がわかるようにする

重要度	難易度	修正コスト
★★★	★	★★

説明・動機	boolean 変数を返すメソッド名は、その返り値の 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。これを統一することによって、他者にとっても読みやすいコードとなります。

サンプル

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

N_MTD008 英語の対義語を意識せよ

重要度	難易度	修正コスト
★★★	★	★★

説明･動機	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。これによって、他者にとっても読みやすいコードとなります。 例えば、send と receive、top と bottom、width と height 等があります。

サンプル

```
public void send(String message){
}
public String receive(){
}
```

対になっている

2.6. 引数

N_PRM001 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

重要度	難易度	修正コスト
★★★	★	★★

説明･動機	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。同じ名前になってしまう場合は、”_”を始めにつけて区別する、this を使用する、等の対策をとってください。これによって、他者にとっても読みやすいコードとなります。

違反サンプル

```
private String name;
public void setName(String name){
}                              //違反
```

修正サンプル

```
private String name;
public void setName(String _name){
}                               //修正済み
```

2.7. 変数全般

N_VAL001 boolean 変数は true/false の状態がわかるようにする

重要度	難易度	修正コスト
★★★	★	★★

説明･動機	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。これを統一することによって、他者にとっても読みやすいコードとなります。 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。 (例) 在庫があるかどうかのフラグとして使用する変数 〇hasStock ×isStock

サンプル

```
private boolean isAsleep;
private boolean canSpeak;
private boolean hasExpired;
private boolean exists;
private boolean hasValue;
```

N_VAL002 定数は全て static final とし、すべて大文字、区切りは"_"

重要度	難易度	修正コスト
★★★	★	★★

説明･動機	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。これを統一することによって、他者にとっても読みやすいコードとなります。

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
```

N_VAL003 変数名には役割の反映を

重要度	難易度	修正コスト
★★★	★	★★

説明･動機	変数にはその役割が反映された名前を付けてください。これを統一することによって、他者にとっても読みやすいコードとなります。

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
```

2.8. ローカル変数

N_LVL001 スコープが狭い変数名は省略した名前でもよい

重要度	難易度	修正コスト
★★	★	★★

説明･動機	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。 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

```
mport java.io.*;

public class Sample {
    public static void main(String[] args){
      String index = “一行:”;//修正済み
      try{
            BufferedReader br = new
BufferedReader(new FileReader(“test.txt”));
            String str = br.readLine();
            System.out.println(str);
            br.close();
        }catch(IOException e){
            e.printStackTrace();
        }
    }
}
```

N_LVL002 for 文のループカウンタは、ネストごとに”i”,”j”,”k”・・・を使う

重要度	難易度	修正コスト
★★	★	★★

説明･動機	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。これを統一することによって、他者にとっても読みやすいコードとなります。

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++){
        }
    }
}
                                    //違反
```

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++){
        }
    }
}
                                    //修正済み
```

3. コーディング規約

3.1. 全般

C_GNR000 オブジェクトの参照にはインターフェースを利用せよ

重要度	難易度	修正コスト
★★★	★★	★★

説明･動機	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){
    Vector 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。
    }
}
```

C_GNR001 推奨されない API を使用しない

重要度	難易度	修正コスト
★★★	★★	★★

説明･動機	“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();
        System.out.println(
                    sampleDate.getYear());
    }                              //違反
}
```

修正サンプル

```
import java.util.Calendar;

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

C_GNR002 使われないコードは書かない

重要度	難易度	修正コスト
★★★	★	★★

説明･動機	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は記述されていませんか？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。必要のあるものについてはそれが使用されるようコードを見直してください。

違反サンプル

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

修正サンプル

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

C_GNR003 宣言は適切な権限で

重要度	難易度	修正コスト
★★★	★	★★★

説明･動機	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

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

修正サンプル

```
private void internalMethod{
}                    //修正済み
```

C_GNR004 final を適切に利用する

重要度	難易度	修正コスト
★★	★	★★★

説明・動機

継承されないクラス、オーバーライドされないメソッド、値の変わらない変数（つまり定数）等、変化のないもの/変化させたくないものについては final で宣言するようにしてください。変化のないことが明示され、コードの可読性も向上します。

違反サンプル

```
public class BadSample {
    public static int DEFAULT_INTERVAL = 60;
                                    //違反
}
```

修正サンプル

```
public class FixedSample {
    public static final int DEFAULT_INTERVAL =
60;
                                    //修正済み
}
```

C_GNR005 プリミティブ型と参照型の違いを意識する（定義編）

重要度	難易度	修正コスト
★★★	★★	★★

説明・動機

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

説明・動機

◇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
・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）。

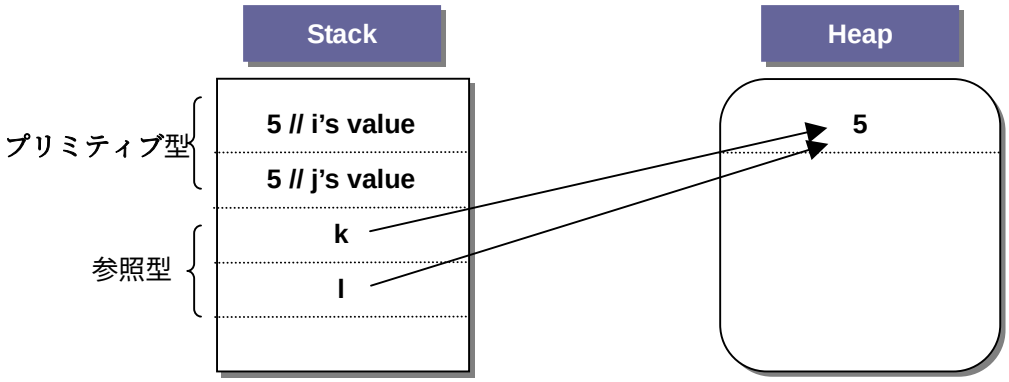
・参照型：値への参照が代入される

（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）。

メモリ上でのイメージ



*メソッドの引数として渡されるときも、同様のことが起こります。
（メソッドパラメータ編を参照してください）

C_GNR006 プリミティブ型と参照型の違いを意識する(引数編)

重要度	難易度	修正コスト
★★★	★★	★★

説明・動機

メソッドのパラメータとして渡される引数が、プリミティブ型である場合と参照型である場合の挙動の違いを理解していますか？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);
        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。

3.2. メトリクス

C_MTR001 1 メソッドの行数は約 20 行以下

重要度	難易度	修正コスト
★★	★	★★
説明･動機	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。	

C_MTR002 1 クラスの行数は約 600 行以下

重要度	難易度	修正コスト
★★	★	★★★
説明･動機	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。	

C_MTR003 1 クラス内の public メソッド数は 30 個以下

重要度	難易度	修正コスト
★★	★	★★★
説明･動機	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。これ以上メソッド数が増えてしまう場合は、クラスを分割する等、設計の見直しを行ってください。	

C_MTR004 1 パッケージ内のクラス数は 10 個以下

重要度	難易度	修正コスト
★★	★	★★★
説明･動機	1 パッケージに含まれるクラス数は 10 個程度までが理想です。多くても 20 個程度にはとどめてください。これ以上クラス数が増えてしまう場合は、パッケージを分割する等、設計の見直しを行って下さい。	

C_MTR005 循環的複雑さを大きくしない

重要度	難易度	修正コスト
★★	★★★	★★★
説明･動機	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。	

3.3. フォーマット

C_FMT001 タブを利用せず、空白文字を利用する

重要度	難易度	修正コスト
★★	★	★

説明・動機	コードのインデントにタブを使っていませんか？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。いちいち空白文字を入れるのが面倒な場合はエディタの置換機能を使って最後に一括変換するのも手です。

違反サンプル

```
public class BadSample {
       private int bottom = 5;
    private int height = 10;
       private int constant = 2;
    int getArea() {
       int area = (bottom * height) /
                                  constant;
       return area;
    }
}
/*違反：タブと空白文字が混在しており、別エディタでの表示が崩れてしまっている*/
```

修正サンプル

```
public class FixedSample {
    private int bottom = 5;
    private int height = 10;
    private int constant = 2;
    int getArea() {
       int area = (bottom * height) / constant;
       return area;
    }
}
//修正済み
```

C_FMT002 インデントは空白文字 4 文字分

重要度	難易度	修正コスト
★★	★	★

説明・動機	特に取り決めがない限り、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。

違反サンプル

```
public class BadSample {
     private int bottom = 5;
     private int height = 10;
     private int constant = 2;
     public int getArea() {
      int area = (bottom * height) / constant;
      return area;
     }
}
/*違反：インデントの空白文字数が統一されておらず読みづらい*/
```

修正サンプル

```
public class FixedSample {
    private int bottom = 5;
    private int height = 10;
    private int constant = 2;
    public int getArea() {
       int area = (bottom * height) / constant;
       return area;
    }
}
//修正済み
```

C_FMT003 長すぎる行は避ける

重要度	難易度	修正コスト
★★★	★	★★

説明・動機

長いコードを改行せずにだらだら続けていませんか？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。
改行場所の目安として、臨機応変に対処してください：

1. カンマのあとで改行する
2. 優先度の低い演算子の前で改行する
3. 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②のように見やすい位置であわせるようにしてください。

サンプル

【違反サンプル】

```
// 違反 長すぎる！！
    public void longLineMethod( boolean booleanValue, String stringValue, int intValue){
    }
```

【修正サンプル】

① カンマのあとで改行

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

② 演算子の前で改行

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

③ 変数に代入

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

C_FMT004 "{"の後にステートメントを記述しない

重要度	難易度	修正コスト
★★★	★	★★

説明・動機

“{“の後に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、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
}
```

C_FMT005 1 行に 2 つ以上のステートメントを記述しない

重要度	難易度	修正コスト
★★★	★	★★

説明･動機	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。

違反サンプル

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

修正サンプル

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

C_FMT006 カンマの後には空白文字を

重要度	難易度	修正コスト
★	★	★★

説明･動機	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

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

修正サンプル

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

C_FMT007 代入演算子 (=, +=, -=, ...) の前後には空白文字を

重要度	難易度	修正コスト
★	★	★★

説明･動機	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
```

C_FMT008 for 文内のセミコロンの後には空白文字を

重要度	難易度	修正コスト
★	★	★★

説明･動機	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
}
```

C_FMT009 “++”や”--“とオペランドの間には空白文字を入れない

重要度	難易度	修正コスト
★	★	★★

説明･動機	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	修正サンプル

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

C_FMT010 ビット演算子（”|”、”&”、”^”、”<<”、”>>”）の前後には空白文字を

重要度	難易度	修正コスト
★	★	★★

説明･動機	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	修正サンプル

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

C_FMT011 論理演算子（”||”、”&&”）の前後には空白文字を

重要度	難易度	修正コスト
★	★	★★

説明･動機	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	修正サンプル

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

C_FMT012 関係演算子（"<"、">"、 ">="、 "<="、"=="、"!="）の前後には空白文字を

重要度	難易度	修正コスト
★	★	★★

説明･動機	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	修正サンプル

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

C_FMT016 算術演算子（"＋"、"ー"、 "＊"、 "/"、"%"）の前後には空白文字を

重要度	難易度	修正コスト
★	★	★★

説明･動機	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	修正サンプル

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

C_FMT013 return 文ではカッコを使わない

重要度	難易度	修正コスト
★	★	★★

説明･動機	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

C_FMT014 boolean 変数は既に比較済み

重要度	難易度	修正コスト
★★	★	★★

説明･動機	boolean 変数をわざわざ true と比較していませんか？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
	}

C_FMT015 不等号の向きは左向き ("<"、"<=") にする

重要度	難易度	修正コスト
★★	★	★★

説明･動機	不等号の向きがばらばらになっていませんか？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{
	
	}

3.4. コメント

C_CMT001 Javadoc コメントには、少なくとも author と version(クラス),param と return と exception(メソッド)を記述する

重要度	難易度	修正コスト
★★	★	★★

説明・動機

クラス、メソッド、フィールドについては下記のように Javadoc コメントを記述してください。特に public 及びデフォルトで宣言されたものについては必須です。下記以外のタグについても、必要に応じて記述してください。

【Javadoc コメントの様式】
・「/**」で始める
・2 行目以降は「*」で始め、下記の Javadoc タグで本文を記述する
・「*/」で終わる

【クラス】
@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。

【メソッド】
@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。

【フィールド】
フィールドについての説明
できるだけ変数名で説明する。どうしてもあらわしきれない場合は、Javadoc コメントとして説明を記述する。

注意：本コーディング規約のサンプルコードは、規約の説明を目的としたコードです。規約のポイントをよりはっきりさせるために、Javadoc コメントは記述していません。

サンプル

```
/**
 * Javadoc コメントの記述について説明するサンプルクラスです。
 * @version 1.0
 * @author Tsuyoshi Wakahara
 */
public class JavadocSample {

    /**
     * sampleMethod の説明
     * @param value 引数の説明
     * @return 返り値の説明
     * @exception SampleException 例外の説明
     */
    public String samleMethod(String value) throws SampleException{
        String returnValue = “SAMPLE”;
        return returnValue;
```

```
    }

}
```

C_CMT002 コメントは必要なものだけを簡潔に

重要度	難易度	修正コスト
★★★	★	★★

説明・動機

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。

以下の違反サンプルのように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。

下記のサンプルでは、クラスの Javadoc コメントは省略しています。

違反サンプル

```
public class BadCommentSample {
   /**
    * 和を求めるメソッド
    *
    * @param elements 和を求めたい数値の配列
    * @return 引数の配列に対して求めた和
    */
    public int sum(int[] elements){
        //和の結果を格納する変数
      int sum = 0;
       //配列の和を計算するループ
      for(int i = 0; i < elements.length; i
++){
                //要素をひとつ取り出して和を計算
         sum += elements[i];
      }
       //結果を返す
      return sum;
    }
}
```

修正サンプル

```
public class FixedCommentSample {
   /**
    * 和を求めるメソッド
    *
    * @param elements 和を求めたい数値の配列
    * @return 引数の配列に対して求めた和
    */
    public int sum(int[] elements){
      int sum = 0;
      for(int i = 0; i < elements.length; i ++){
         sum += elements[i];
      }
      return sum;
    }
}
```

3.5. import

C_IMP002 java.lang パッケージはインポートしない

重要度	難易度	修正コスト
★	★	★★

説明・動機	java.lang パッケージは非明示的にインポートされることを知っていますか？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。

違反サンプル	修正サンプル

`import java.lang.*;  //違反`
 	
`public class BadSample {`
`}`
	`//修正済み`
	`public class FixedSample {`
	`}`

3.6. クラス

C_CLS001 継承させたくないクラスは final 宣言する

重要度	難易度	修正コスト
★★★	★★	★★

説明・動機	継承されないクラスは final を宣言してください。下記のメリットがあります。 ・ クラスの継承及びメソッドのオーバーライドのないことがコンパイラに通知されるため、コードの最適化がより適切に行われます。 ・ final を宣言することでそのクラスは継承されないことが明示され、可読性が向上します。

違反サンプル	修正サンプル
`public class BadSample{  //違反` `}`	`public final class FixedSample{  //修正済み` `}`

C_CLS003 フィールドを宣言する順序は public (+)、protected (#)、デフォルト (~)、private (-) の順

重要度	難易度	修正コスト
★	★	★★

説明・動機	フィールドを宣言する際、その順序は考慮に入れていますか？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	修正サンプル

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

3.7. コンストラクタ

C_CRT001 public 宣言していないクラスには public 権限のコンストラクタを作らない

重要度	難易度	修正コスト
★★★	★	★★

説明･動機	public でないクラスにコンストラクタを作成する際、うっかり public で宣言していませんか？public でないクラスのコンストラクタにはどこからでもアクセスできるわけではありません。例えば、下記の違反サンプルの場合、このコンストラクタを別パッケージから呼び出すクラスはコンパイルが通りません。このようなことを防ぐために、コンストラクタから public 修飾子を除去するか、クラスに public 修飾子をつけられるように設計を見直してください。

違反サンプル

```
class BadSample {   //非 public なクラス
    public BadSample(){   //違反
    }
}
```

修正サンプル

```
class FixedSample {
    FixedSample(){    //修正済み
    }
}
```

C_CRT002 デフォルトコンストラクタは、原則作成する

重要度	難易度	修正コスト
★★	★	★★

説明･動機	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：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。。

違反サンプル

```
public class BadSample {
    private int number;
    public BadSample(int num){   //違反
        number = num;
    }
}
```

修正サンプル

```
public class FixedSample {
    private int number;
    public FixedSample(){  //修正済み
        this(10);
    }
    public FixedSample(int num){
        number = num;
    }
}
```

3.8. メソッド

C_MTD001 オーバーライドさせたくないメソッドは final を利用する

重要度	難易度	修正コスト
★★★	★	★★

説明・動機	そのメソッドは、サブクラスによってオーバーライドさせてもよいでしょうか？オーバーライドされないことがわかっている場合には final 宣言をしてください。これにより、誰もメソッドをオーバーライドしていないことがコンパイラに通知されるため、コンパイラは最適化されたコードを生成することができパフォーマンスが向上します。

違反サンプル	修正サンプル

C_MTD002 サイズが 0 の配列を利用する

重要度	難易度	修正コスト
★★	★★	★★★

説明・動機	戻り値が配列のメソッドで、null が戻される可能性はありませんか？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。

違反サンプル

```
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;
       }
    }
}
```

修正サンプル

```
import java.util.List;

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

C_MTD003 public メソッドはクラスの整合性を壊さないような設計を

重要度	難易度	修正コスト
★★★	★★	★★★

説明・動機	public メソッドを作成する際は、クラスの内部的な整合性を壊さないよう、慎重に設計してください。

違反サンプル

下記のクラスは、stockRightSpeaker と stockLeftSpeaker の２つの数値を属性として保持しています。Speaker は常に左右セットで売られなければならないとしたときに、以下の設計では問題があります：

```
public class BadSample {
    private int stockRightSpeaker;
    private int stockLeftSpeaker;

    public void buyRightSpeaker(
                    int quantityRight){
        stockRightSpeaker =
          stockRightSpeaker - quantityRight;
    }  //違反

    public void buyLeftSpeaker(
                    int quantityLeft){
        stockLeftSpeaker =
          stockLeftSpeaker - quantityLeft;
    }  //違反
}
```

ここでは、それぞれの buy メソッドによって stockRightSpeaker と stockLeftSpeaker の両方の値をそれぞれ外部から変更することができます。しかし、万が一どちらか一方の値のみが変更された場合、Speaker は左右セットで販売という前提は容易に崩れてしまいます。

修正サンプル

修正後は、stockRightSpeaker と stockLeftSpeaker に別々にアクセスするのではなく、ひとつのメソッドで一度に変更を行うようにしています。これで、常に Speaker は左右セットで販売という前提を保持することになります。

```
Public class FixedSample {
    private int stockRightSpeaker;
    private int stockLeftSpeaker;

    public void buySpeaker(int quantity){
        stockRightSpeaker =
                stockRightSpeaker - quantity;
        stockLeftSpeaker =
                stockLeftSpeaker - quantity;
    }  //修正済み
}
```

C_MTD004 メソッドは 1 つの役割にする

重要度	難易度	修正コスト
★★★	★	★★★

説明･動機	1つのメソッド内で複数の処理が行われていませんか？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);
    }

}
```

C_MTD005 メソッドの返り値として this は利用しない

重要度	難易度	修正コスト
★★	★★	★★★

説明・動機	何故 this を return するのか、もう一度考えてみてください。何か特別な理由は見当たりますか？this を return することによって得られるメリットはほとんどありません。メソッドの返り値を void にするか、this 以外の返り値にしてください。

解説

メソッドが呼び出されるということは、メソッドの呼び出し元はオブジェクトに対する参照を持っています。this を return することは無意味です。また、便宜上の都合で以下①のようなメソッド呼び出しを、②のように記述できるようにするために this を return するという場合があるかもしれません：

①
```
Sample sample = new Sample();
sample.sampleMethod1();
sample.sampleMethod2();
sample.sampleMethod3();
```

②
```
sample.sampleMethod1().sampleMethod2().sampleMethod3();
```

しかし、下記のような可読性上の問題が発生しますのでこれも避けてください。

- Javadoc を見ただけでは同じ型のオブジェクトが返されることが、上記のような便宜的な理由によるものなのか、他の意味があるのかということが、コードを見て this が return されているのを確認するまでわからない。
- ②のように書かれていると、sampleMethod1()と sampleMethod2()の戻り値が何かということがわからず、sampleMethod2()と sampleMethod3()がどの型のメソッドなのかがわからない。
- 明示的に同期化されていない場合、各メソッドの処理間で別のスレッドによる処理が入る可能性があるが、②の書き方ではその一文を実行する間は異なるスレッドに実行が移ることはないと勘違いしてしまう可能性がある。

C_MTD006 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

重要度	難易度	修正コスト
★★★	★★★	★★★

説明･動機

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。

解説

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)); ①'
       System.out.println(badSample.classify(sample2)); ②'
       System.out.println(badSample.classify(sample3)); ③'
   }
}
```

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、①'、②'、③'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。

3.9. クラスメソッド

C_CMD001 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

重要度	難易度	修正コスト
★★★	★	★★

説明･動機	クラスメソッドを使用する際、オブジェクト名を用いて呼びだしていませんか？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
    }
}
```

3.10. 変数全般

C_VAL001　1 つのステートメントには 1 つの変数宣言

重要度	難易度	修正コスト
★★	★	★★

説明･動機	面倒だからといって、1 つのステートメントで複数の変数を宣言するようなことはしていませんか？これでは、結果としてコードの可読性･保守性を下げることになってしまいます。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。

違反サンプル

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

修正サンプル

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

C_VAL002　リテラルは使用しない

重要度	難易度	修正コスト
★★	★	★★

説明･動機	リテラルを使用している個所はありませんか？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。 例外：-1,0,1 等をカウント値としてループ処理等で使用するような場合 ＊Java コーディング規約 2004 内のサンプルの中には、わかりやすさを重視したため、本規約「リテラルは使用しない」を意図的に守っていないものも存在します。

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 static final int ARRAY_SIZE = 10;

    private int[] sampleArray = new int
[ARRAY_SIZE];
                                    //修正済み
}
```

C_VAL003 配列宣言は「型名[]」にする

重要度	難易度	修正コスト
★★	★	★★

説明・動機	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
}
```

C_VAL004 できるだけローカル変数を利用する

重要度	難易度	修正コスト
★★	★	★★

説明・動機	変数へのアクセスが頻繁に起こる場合、パフォーマンスは考慮に入れられていますか？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;
    }
}
```

C_VAL005 定数は final で宣言する

重要度	難易度	修正コスト
★★★	★	★★

説明・動機	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 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 final int CONSTANT = 5;
                                        //修正済み
    private int getSize(int number) {
        return number * CONSTANT;
    }
}
```

C_VAL006 ローカル変数とインスタンス変数を使いわける

重要度	難易度	修正コスト
★★	★	★★

説明･動機	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いませんか。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...
        }
    }

    // ...
}
```

3.11. インスタンス変数

C_IVL001 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

重要度	難易度	修正コスト
★★★	★★	★★

説明･動機	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 ・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;
    }
}
```

C_IVL002 インスタンス変数は private にする

重要度	難易度	修正コスト
★★★	★	★★

説明・動機

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いませんか？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();
        result += 10;
        System.out.println(result);
    }
}
```

3.12. クラス変数

C_CVL001 public static final 宣言した配列を利用しない

重要度	難易度	修正コスト
★★★	★★	★★

説明・動機

final 宣言を行った配列の要素は変更できないと思っていませんか？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 {
    public static final int[] SAMPLE_ARRAY
                = {0, 1, 2, 3};    //違反

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

C_CVL002 クラス変数にはクラス名を使用してアクセス

重要度	難易度	修正コスト
★★★	★	★★

説明･動機	クラス変数を使用する際、オブジェクト名を用いてアクセスしていませんか？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();
        int localValue
             =object.STATIC_VALUE;   //違反
    }
}
```

修正サンプル

```
public class FixedSample {
    public static final int STATIC_VALUE = 10;

    public void sampleMethod(){
        FixedSample object = new FixedSample();
        int localValue
          = FixedSample.STATIC_VALUE;  //修正済
み
    }
}
```

3.13. ローカル変数

C_LVL001 ローカル変数は利用する直前で宣言する

重要度	難易度	修正コスト
★★	★	★★

説明・動機

変数のスコープは必要最小限に留めましょう。ローカル変数をとりあえずコードのはじめにまとめて宣言しておく、というようなことはありませんか？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。

違反サンプル

```
public class BadSample {
    public static void main(String[] args){
        int localValue1 = 10;
        int localValue2 = 20;
        int localValue3 = 30;    //違反

        TestClass test = new TestClass();

        test.sampleMethod1(localValue1);
        ・・・
        test.sampleMethod2(localValue2);
        ・・・
        test.sampleMethod3(localValue3);
        ・・・
    }
}
```

修正サンプル

```
public class FixedSample {
    public static void main(String[] args){
        TestClass test = new TestClass();

        int localValue1 = 10;
        test.sampleMethod1(localValue1);
        ・・・

        int localValue2 = 20;
        test.sampleMethod2(localValue2);
        ・・・

        int localValue3 = 30;
        test.sampleMethod3(localValue3);
        ・・・                        //修正済み
    }
}
```

C_LVL002 ローカル変数は安易に再利用しない

重要度	難易度	修正コスト
★★★	★	★★

説明・動機

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いませんか？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;
    }
}
```

3.14. 継承

C_IHT001 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

重要度	難易度	修正コスト
★★★	★★	★★

説明・動機

スーパークラスと同じ名前のフィールドをサブクラスで宣言していませんか？ 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。

違反サンプル

```
public class BadSample{
    protected int number = 4;
}

public class ChildBadSample extends
BadSample{
    protected int number = 5; // 違反

    public static void main(String[] args){
        new BadSample().print();
    }

    private void print(){
        System.out.println(
                    "number is " + number );
        System.out.println(
          "super.number is " + super.number );
    }
}
```

上記プログラムの実行結果は以下のとおりです：

```
number is 5
super.number is 4
```

修正サンプル

```
public class FixedSample{
    protected int number = 4;
}

public class ChildFixedSample extends
FixedSample{
    protected int childNumber = 5; // 修正済み

    public static void main(String[] args){
        new FixedSample().print();
    }

    private void print(){
        System.out.println(
                    "number is " + number );
        System.out.println(
             "super.number is " + super.number );
        System.out.println(
             "childNumber is " + childNumber );
    }
}
```

上記プログラムの実行結果は以下のとおりです：

```
number is 4
super.number is 4
childNumber is 5
```

C_IHT002 abstract メソッドを利用する

重要度	難易度	修正コスト
★★★	★	★★

説明・動機

何も処理をしない空のメソッドはありませんか？実装のないメソッドは空にせず、必要であれば abstract として宣言してください。そうすることによってサブクラスでの実装漏れを防ぐことができます。

違反サンプル

```
public abstract class BadSample
                    implements ISample {

    public void sampleMethod(){}
                              // 違反
 }

public interface ISample {
    public void sampleMethod();
 }
```

修正サンプル

```
public abstract class FixedSample
                         implements ISample {

    public abstract void sampleMethod();
                                   //修正済み
 }

public interface ISample {
    public void sampleMethod();
 }
```

C_IHT003 スーパークラスで private 宣言されているメソッドと同じ名前のメソッドをサブクラスで定義しない

重要度	難易度	修正コスト
★★★	★★	★★

説明・動機

スーパークラスにある private メソッドと同じ名前のメソッドをサブクラスで定義しないでください。private メソッドはオーバーライドされず全く別のメソッドとして扱われます。他の人の混乱を招き、バグにつながる恐れもあります。
また、スーパークラスの public メソッドをみだりに private メソッドに変更してはいけません。もしもそのメソッドがサブクラスでオーバーライドされていた場合、サブクラスのメソッドの意味が変わってしまいます。エラーにもならないので、気づかれないままバグとなってしまう場合があります。
下記解説に具体的な例を記します。

解説

まずはスーパークラスの private ではないメソッドを通常にオーバーライドした場合を見てみましょう:

```
// スーパークラス
 public class Parent {
    public void sampleMethod(){
       System.out.println("Parent: method");
    }

    public static void main(String[] args){
       Child childInstance = new Child ();
       test(childInstance);
    }

    public static void test(Parent parentInstance){  //スーパークラスの型の引数
       parentInstance.sampleMethod ();  // ☆
    }
 }

// スーパークラスを継承したクラス
 public class Child extends Parent {
    public void sampleMethod(){  // スーパークラスと同名のメソッド
       System.out.println("Child: method");
    }
 }
```

オーバーライドが正しくされている場合、引数がスーパークラスの型で定義されたメソッドが呼ばれたとき、自動的にサブクラスでオーバーライドされたメソッドが呼ばれます。
従って、上記のコードを実行した場合、Parent クラスの sampleMethod()メソッドは Child クラスでオーバーライドされているので、☆印の部分で Child クラスのインスタンスの sampleMethod()メソッドが呼ばれます。結果は以下のように表示されます:

```
  Child: method
```

一方、スーパークラスで private として宣言されたメソッドはオーバーライドすることはできません。しかし、もし同じ名前のメソッドを定義したとしてもエラーになるわけではなく、全く別のメソッドとして扱われます。
従って、網掛け部分の public を private に変更して実行すると結果は以下のようになります:

```
  Parent: method
```

メソッドはオーバーライドされませんので、Parent クラスの sampleMethod()メソッドが呼ばれます。気づかずに変更をしてしまった場合バグの原因となる場合もあります。

C_IHT004　equals()メソッドを実装した場合は、コレクションクラスに格納することを想定して hashCode()メソッドも実装する

重要度	難易度	修正コスト
★★★	★★	★★

説明･動機

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ましたか？
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。

解説

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。
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。

C_IHT005 可能な限り toString()メソッドを実装する

重要度	難易度	修正コスト
★★	★	★★

説明・動機

toString()メソッドであなたが期待する値が得られることを確認しましたか？デフォルトの Object.toString()メソッドは hashCode()で得た値を返すだけです。ハッシュコードよりもわかりやすい値を出力するためにはメソッドをオーバーライドする必要があります。

違反サンプル

```
public class BadSample{
    private int sampleValue;

    public BadSample(int value){
       sampleValue = value;
    }

   public static void main(String[] args){
     System.out.println(
                  new BadSample(100));
   }
}
```

上記を実行した場合、以下のような結果が得られます。

```
BadSample@5224ee
```

修正サンプル

```
public class FixedSample{
   private int sampleValue;

   public FixedSample(int value){
      sampleValue = value;
   }

   public String toString(){
      StringBuffer result =
         new StringBuffer("FixedSample[");

      result.append(sampleValue);
      result.append("]");

      return result.toString();
   }    //修正済み

   public static void main(String[] args){
     System.out.println(
                    new FixedSample(100));
   }
}
```

toString()をオーバーライドした上記を実行した場合、結果は以下のようになります。

```
FixedSample[100]
```

3.15. インスタンス

C_IST001 オブジェクトどうしは equals()メソッドで比較する

重要度	難易度	修正コスト
★★★	★	★★

説明・動機

オブジェクトの値の等値比較を"=="で行っていませんか？"=="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。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.");
       }
    }
}
```

C_IST002 Class 名を利用した比較をおこなわない

重要度	難易度	修正コスト
★★★	★★	★★

説明・動機

Class オブジェクトを名前（Class クラスの getName()メソッドで取得される名前）で比較していませんか？JavaVM によっては、getName()メソッドの返り値にパッケージ名がつかないため、複数の異なるクラスが同じ値となる可能性があります。必ず Class クラスのオブジェクト同士を直接比較するようにしてください。（JDK1.4 の getName()メソッドはパッケージ名がつくため名前は一意です。）

違反サンプル

```
import java.util.*;

public class BadSample{
   private List sampleList =
                          new ArrayList();

   public boolean hasSameList(List list){
      Class classA = sampleList.getClass();
      Class classB = list.getClass();

      String classAName = classA.getName();
      String classBName = classB.getName();

      return classAName.equals(classBName);
                                  // 違反
   }
}
```

修正サンプル

```
import java.util.*;

public class FixedSample{
   private List sampleList =
                          new ArrayList();

   public boolean hasSameList(List list){
      Class classA=sampleList.getClass();
      Class classB = list.getClass();
      // 修正済み
      return classA.equals(classB);
   }
}
```

3.16. 制御構造

C_CTR001 制御文 (if, else, while, for, do while) の“{ }”は省略しない

重要度	難易度	修正コスト
★★★	★	★★

説明・動機	制御文の”{}”を省略していませんか？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");
        }
    }
}
```

C_CTR002 ステートメントが無い{}ブロックを利用しない

重要度	難易度	修正コスト
★★	★	★★

説明・動機	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

違反サンプル

```
for(int i = 0; i < 10; i++ ){
        // 後で実装する！！
}
```

C_CTR003 if/while の条件式で"="は利用しない

重要度	難易度	修正コスト
★★★	★	★★

説明・動機

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。

違反サンプル

```
public class BadSample{
    public String badSampleMethod(
                                boolean flag){
        String value = "0";
        if (flag = true) {  // 違反
            value = "a";
        }
        return value;
    }
}
```

上記の場合、flag に true が代入されるので条件の結果は常に true になってしまいます。

修正サンプル

```
public class FixedSample{
    public String fixedSampleMethod(
                                boolean flag){
        String value = "0";
        if (flag == true) {  // 修正
//違反 正しくは if(flag)と書きます(規約 C_FMT014 参照)
            value = "a";
        }
        return value;
    }
}
```

C_CTR004 for と while の使い分けを意識する

重要度	難易度	修正コスト
★★	★★	★★

説明･動機

for と while の違いを意識して使っていますか？プログラマの好みにもよりますが、基本的に配列や数列を処理する場合やループ回数が明確なものに関しては for 文を使用してください。使い分けを間違えると、著しく可読性が低下します。
for 文の括弧の中は、カウンタの宣言、ループ条件、次のカウンタを求める式の 3 つの条件を記述しています。これらの条件を省略する時は、while 文の適用も考えられるはずです。while 文ではない明確な理由があるときだけ、条件式を省略した for 文を利用してください。

違反サンプル

for 文のインクリメントの省略はしないでください:

```
public class BadSample{
    void method(int x){
        // 違反
        // カウンタ定義とインクリメントの省略は避ける
        int i = 0;
        for (; i < x; ) {
            // 処理
            i++;
        }
    }
}
```

用途によっては例外もありますが、数値をインクリメントさせる場合は原則 for 文を使ってください:

```
public class AnotherBadSample{
    void method(int x){
        int i = 0;
        while(i < x){  // 違反 for 文が適当
            // 処理
            i++;
        }
    }
}
```

また次のような場合は、特にインクリメントが必要ではありませんので、修正サンプルのように、原則 while 文を使ってください:

```
void sampleMethod(String target){
       StringTokenizer strToken =
               new StringTokenizer(target)
     for(int  i=0;
          strToken.hasMoreTokens();i++){
             System.out.println(
                 strToken.nextToken());
       }
}
```

修正サンプル

カウンタをインクリメントさせて行うループ処理には原則 for 文を使うようにしましょう:

```
public class FixedSample{
    void method(String[] arrayStr){
        int size = arrayStr.length;
        for(int i = 0; i < size; i++){
            System.out.println(arrayStr[i]);
        }
    }
}
```

特にインクリメントが必要でない場合は while ループを使います:

```
import java.util.StringTokenizer;

public class WhileLoopSample{
    void sampleMethod(String target){
        StringTokenizer strToken
                    = new StringTokenizer(target)
        while(strToken.hasMoreTokens()){
            System.out.println(
                        strToken.nextToken());
        }
    }
}
```

C_CTR005 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

重要度	難易度	修正コスト
★★★	★	★★

説明･動機	for 文のカウンタをループ内のステートメントで変更していませんか？ 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;
    }
 }
```

C_CTR006 for 文のカウンタは 0 から始める

重要度	難易度	修正コスト
★★★	★	★★

説明･動機	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

```
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

```
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;
        }
    }
}
```

C_CTR007 break や continue は使わないほうがわかりやすい

重要度	難易度	修正コスト
★★	★★	★★

説明・動機

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。

違反サンプル

```
public class BadSample{
    public static void main(String[] args){
        BadSample sample = new BadSample();

        sample.useContinue();
        sample.useBreak();
    }

    private void useContinue(){
        int skipNumber = 5;

        for (int i = 0; i < 10; i++){
            if ((i + 1) == skipNumber){
                continue;       //違反
            }
            System.out.println(
                    (i + 1) + "回目です。");
        }
    }

    private void useBreak(){
        int limitNumber = 5;

        for (int i = 0; i < 10; i++){
            if (limitNumber < (i + 1) ){
                break;          //違反
            }
            System.out.println(
                        (i + 1) + "回目です。");
        }
    }
}
```

[実行結果]

```
1 回目です。
2 回目です。
3 回目です。
4 回目です。
6 回目です。
7 回目です。
8 回目です。
9 回目です。
10 回目です。
1 回目です。
2 回目です。
3 回目です。
4 回目です。
5 回目です。
```

修正サンプル

```
public class FixedSample{
    public static void main(String[] args) {
        FixedSample sample = new FixedSample();

        sample.unUseContinue();
        sample.unUseBreak();
    }

    private void unUseContinue(){
        int skipNumber = 5;

        for (int i = 0; i < 10; i++){
            if ((i + 1) != skipNumber){
                                        //修正済み
                System.out.println(
                        (i + 1) + "回目です。");
            }
        }
    }

    private void unUseBreak(){
        int limitNumber = 5;

        for (int i = 0; (i + 1) <= limitNumber;
                                i++){ //修正済み
            System.out.println(
                        (i + 1) + "回目です。");
        }
    }
}
```

[実行結果]

```
1 回目です。
2 回目です。
3 回目です。
4 回目です。
6 回目です。
7 回目です。
8 回目です。
9 回目です。
10 回目です。
1 回目です。
2 回目です。
3 回目です。
4 回目です。
5 回目です。
```

C_CTR008 配列をコピーするときは System.arraycopy ()メソッドを利用する

重要度	難易度	修正コスト
★★★	★★	★★

説明･動機

配列をコピーするのにわざわざループ文などで処理を書いていませんか？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。

違反サンプル

```
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

```
public class FixedSample{
    String[] copy(String[] orgArray){
        if(orgArray == null){
            return null;
        }

        int length = orgArray.length;
        String[] copyArray = new String[length];
        System.arraycopy(orgArray, 0,
                    copyArray, 0, length);
                                    // 修正済み
        return copyArray;
    }
 }
```

* System.arraycopy の使用方法については Javadoc を参照してください。

C_CTR009 繰り返し処理中のオブジェクトの生成は最小限にする

重要度	難易度	修正コスト
★★★	★★	★★★

説明･動機

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。

違反サンプル

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

修正サンプル

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

            sampleBuffer.setLength(0);
                                    // 初期状態に戻す
        }
    }
 }
```

C_CTR010 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

重要度	難易度	修正コスト
★★	★★	★★★

説明・動機

多くの分岐がメソッドの中に含まれていませんか？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");
    }
}
```

C_CTR011 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)

重要度	難易度	修正コスト
★★★	★★	★★

説明・動機

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。

違反サンプル

```
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
    }
}
```

修正サンプル

```
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
    }
}
```

3.17. スレッド

C_TRD001 スレッドは原則 Runnable を実装

重要度	難易度	修正コスト
★★	★★	★★★

説明･動機

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。

違反サンプル

以下の場合、クラスは Thread クラスのサブクラス:

```
public class BadSampleController
                extends Thread{  // 違反
    private Car car;

    public BadSampleController(Car _car){
        car = _car;
    }

    public void run(){
        car.startEngine();
        // ...
    }

    // その他の Controller としての機能...
}
```

修正サンプル

以下の場合、クラスはスレッド機能を持つ Controller のサブクラス(あるいは自身が Controller スーパークラス)となることができる:

```
public class FixedSampleController
                extends Controller
                implements Runnable{  // 修正済み
    private Car car;

    public FixedSampleController (Car _car){
        car = _car;
    }

    public void run(){
        car.startEngine();
        // ...
    }

    // その他の Controller としての機能...
}
```

C_TRD002 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

重要度	難易度	修正コスト
★★★	★★★	★★

説明・動機

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）

違反サンプル

```
public class BadSample {
    // ...

    public void refuel(int _fuel){
        synchronized(car){
            fuel = _fuel;
            car.notify();  // 違反
        }

        // ...
    }
}
```

修正サンプル

```
public class FixedSample {
    // ...

    public void refuel(int _fuel){
        synchronized(car){
            fuel = _fuel;
            car.notifyAll();  // 修正済み
        }

        // ...
    }
}
```

C_TRD003 Thread クラスの yield()メソッドは利用しない

重要度	難易度	修正コスト
★★★	★★★	★★

説明・動機

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。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。

違反サンプル

```
public class ThreadBadSample {
    private LockObject lock;

    public void longProcess(){
        synchronized(lock){
            Thread.yield();    //違反
    }
}
```

修正サンプル

```
public class ThreadFixedSample {
    private LockObject lock;
     public void longProcess(){
         synchronized(lock){
             try {
                 lock.wait();   //修正済み
             } catch (InterruptedException e) {
                 e.printStackTrace();
             }
         }
    }
}
```

C_TRD004 synchronized ブロックから synchronized ブロックのあるメソッドを呼び出さない

重要度	難易度	修正コスト
★★★	★★★	★★

説明・動機

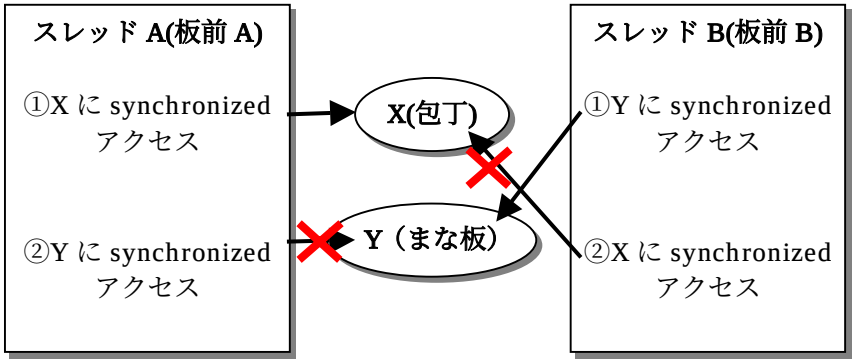
synchronized ブロックの中から synchronized ブロックのあるメソッド(synchronized メソッドを含む)を呼び出していませんか？それらのデッドロック発生の可能性を考慮しましたか？
デッドロックは以下の条件が満たされたときに発生する可能性があります：

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。

C_TRD006 wait()、notify()、notifyAll()メソッドは、synchronized ブロック内から利用する

重要度	難易度	修正コスト
★★	★★★	★★

説明･動機

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。

違反サンプル

```
public class BadSample{
   public static void main(String[] args){
      Test test = new Test();
      try{
         test.wait();  // 違反
      }catch(InterruptedException e){
         e.printStackTrace();
      }
   }
}
```

上記サンプルを実行すると、以下のような例外が発生します。notify()、notifyAll()を呼んだ場合にも同じ結果となります:

```
Exception in thread "main"
java.lang.IllegalMonitorStateException:
current thread not owner
```

修正サンプル

```
public class FixedSample{
   public static void main(String[] args){
      Test test = new Test();
      synchronized(test){  // 修正済み
         try{
            test.wait();
         }catch(InterruptedException e){
            e.printStackTrace();
         }
      }
   }
}
```

修正済みサンプルでは例外は発生しません。

C_TRD007 wait()メソッドは while ブロック内から利用する

重要度	難易度	修正コスト
★★★	★★★	★★

説明･動機	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。

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

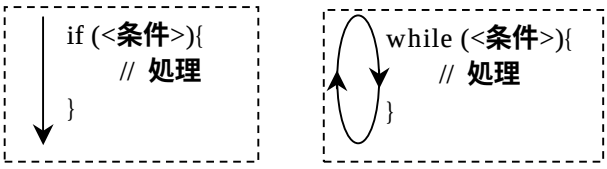
このようなことが起こった場合、fuel が null でないときに car.drive()が正常に呼ばれる(車が走る)、という保証はありません。

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:



*while 文は条件が true である間処理が繰り返される

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。

C_TRD008 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

重要度	難易度	修正コスト
★★★	★★★	★★★

説明・動機

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){
        // …
    }
}

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 ait()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：

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

C_TRD009 同期化(synchronized)の適用は必要な部分だけにする

重要度	難易度	修正コスト
★★★	★★	★★★

説明・動機

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)

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;
    }
}
```

3.18. 文字操作

C_STR001 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

重要度	難易度	修正コスト
★★★	★	★★

説明・動機

文字列どうしの比較に"=="や"!="演算子を用いていませんか？これらの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
    }
}
```

C_STR002 文字列リテラルは new しない

重要度	難易度	修正コスト
★★	★	★★

説明・動機

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 を使用していませんか？このような場合に 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 {
    private String sampleString =
                    new String ("Literal");
    //違反

    public void print() {
        System.out.println(sampleString);
    }
}
```

修正サンプル

```
public class FixedSample {
    private String sampleString = "Literal";
    //修正済み

    public void print() {
        System.out.println(sampleString);
    }
}
```

C_STR003 更新される文字列には StringBuffer クラスを利用する

重要度	難易度	修正コスト
★★★	★	★★

説明・動機

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するようなことはしていませんか？ 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 StringBuffer で定義し、文字列連結の際には 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();
}
```

C_STR004 更新されない文字列には String クラスを利用する

重要度	難易度	修正コスト
★★★	★	★★

説明・動機

変更されない文字列にもかかわらず、StringBuffer オブジェクトとして定義していませんか？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() {
        StringBuffer buffer
             = new StringBuffer(“Message”);
                                    //違反
        String output = buffer.toString();
        System.out.println(output);
    }
}
```

修正サンプル

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

C_STR005 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

重要度	難易度	修正コスト
★★★	★	★★

説明・動機

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) {
        return (userInput != null
               && userInput.equals("y");
                                    //違反
    }
}
```

修正サンプル

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

C_STR006 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

重要度	難易度	修正コスト
★★★	★	★★

説明・動機

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);
```

C_STR007 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

重要度	難易度	修正コスト
★★	★★	★★

説明・動機

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
    }
}
```

C_STR008 システム依存記号 (¥n、¥r など) は使用しない

重要度	難易度	修正コスト
★★	★★	★★★

説明・動機

「\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いませんか？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。

違反サンプル

```
public class BadSample {
    public void displayMessage(
                        String message) {
        StringBuffer result =
            new StringBuffer("Message:¥n");
                                        //違反
        result.append(message);

        System.out.println(result);
    }
}
```

修正サンプル

```
public class FixedSample {
    public void displayMessage(
                        String message) {
        StringBuffer result =
                new StringBuffer("Message:");

        result.append(System.getProperty(
                        "line.separator"));
                                        //修正済み
        result.append(message);

        System.out.println(result);
    }
}
```

3.19. 数値

C_NUM001 誤差の無い計算をするときは、BigDecimal クラスを使う

重要度	難易度	修正コスト
★★★	★★	★★

説明・動機	想定された誤差内の計算結果が得られていますか？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。

解説

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);
        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:9.99999999999998
  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。

C_NUM002 数値の比較は精度に気をつける

重要度	難易度	修正コスト
★★★	★★	★★

説明・動機

数値を比較する際、その数値の精度を把握していますか？前規約（『誤差の無い計算を必要とする時は 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。

C_NUM003 低精度なプリミティブ型にキャストしない

重要度	難易度	修正コスト
★★★	★★	★★

説明・動機

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。

解説

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。

3.20. 日付

C_DATE001 日付を表す配列には、long の配列を利用する

重要度	難易度	修正コスト
★★	★	★★

説明･動機	日付の配列を扱う際、とりあえず Date の配列型を使用する、という認識を持っていませんか？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();
```

3.21. コレクション

C_CLT001　Java2 以降のコレクションクラスを好め

重要度	難易度	修正コスト
★★	★	★★★

説明･動機	Vector クラス、Hashtable クラス、Enumeration クラスを使っていませんか？特にこれらを利用する理由がなければ、インターフェースを統一する目的で、これらの代わりに List(ArrayList クラス)、Map(HashMap クラス)、Iterator を使用するようにしてください。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。

違反サンプル

```
import java.util.*;

public class BadSample {
  public static void main(String[] args){
    Vector sampleVector = new Vector();
                                    //違反

    sampleVector.addElement(“1”);
    sampleVector.addElement(“2”);
    sampleVector.addElement(“3”);

    Enumeration sampleEnumeration =
                 sampleVector.elements();
    while (sampleEnumeration.
                      hasMoreElements()) {
       System.out.println(
         sampleEnumeration.nextElement());
    }

    sampleVector.removeElement(“1”);

    int length = sampleVector.size();
    for (int i = 0; i < length; i++) {
       System.out.println(
              sampleVector.elementAt(i));
    }
  }
}
```

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

    sampleList.add(“1”);
    sampleList.add(“2”);
    sampleList.add(“3”);

    Iterator sampleIterator =
                       sampleList.iterator();
    while (sampleIterator.hasNext()) {
       System.out.println(
                     sampleIterator.next());
    }

    sampleList.remove(“1”);

    int length = sampleList.size();
    for (int i = 0; i < length; i++) {
       System.out.println(
                       sampleList.get(i));
    }
  }
}
```

C_CLT002 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

重要度	難易度	修正コスト
★	★★★	★★★

説明・動機

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。

解説

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 ){
        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。）

3.22. ストリーム

C_STM001 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

重要度	難易度	修正コスト
★★★	★★	★★

説明･動機	ストリームに対する close()メソッドは確実に実行されますか？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();
            }
        }
    }
}
```

C_STM002 ObjectOutputStream では reset()を利用する

重要度	難易度	修正コスト
★★★	★★★	★★

説明・動機

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。

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。

違反サンプル

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

修正サンプル

```
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
    }
}
```

3.23. 例外

C_EXT001 catch 文で受け取る例外は、詳細な例外クラスで受け取る

重要度	難易度	修正コスト
★★★	★★	★★★

説明・動機

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いませんか？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) {
            e.printStackTrace();  //違反
        } finally {
            try{
                reader.close();
            }catch(Exception e2){
                e2.printStackTrace(); //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();
        } catch (IOException ioe1) {
            ioe1.printStackTrace(); //修正済み
        } finally {
            try{
                reader.close();
            }catch(IOException ioe2){
                ioe2.printStackTrace();
                                    //修正済み
            }
        }
    }
}
```

C_EXT002 Exception クラスのオブジェクトを生成してスローしない

重要度	難易度	修正コスト
★★★	★★	★★★

説明・動機

Exception クラスのオブジェクトを生成して、例外をスローしていませんか？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
    }
}
```

C_EXT003 catch ブロックでは必ず処理をする

重要度	難易度	修正コスト
★★★	★★	★★

説明・動機

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
           }
       }
    }
}
```

C_EXT004 Error、Throwable クラスを継承しない

重要度	難易度	修正コスト
★★★	★★	★★★

説明・動機

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。

違反サンプル

```
public class BadSampleException
                  extends Error {  // 違反
}

public class AnotherBadSampleException
                extends Throwable {  // 違反
}
```

修正サンプル

```
public class FixedSampleException
                extends Exception {  // 修正済み
}
```

C_EXT005 例外クラスは無駄に定義しない

重要度	難易度	修正コスト
★★	★★	★★★

説明・動機

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。

違反サンプル

```
// 違反例①
public class InputOutputException
                        extends Exception {
    public InputOutputException(
                                String sample) {
        super(sample);
    }
}

// 違反例②
public class SampleClassABCMethodException
                           extends Exception {
    public SampleClassABCMethodException(
                                    String sample)
{
        super(sample);
    }
}
```

修正サンプル

左記で定義されている違反例①の InputOutputException クラスは JDK 標準パッケージに含まれる IOException クラスを利用すればよいので必要ありません。
また、違反例②の SampleClassABCMethodException クラスはある特定の機能のみに対応しており汎用的ではありませんので、このような定義もしないようにしましょう。

C_EXT006　finally ブロックは必ず実行される

重要度	難易度	修正コスト
★★★	★★	★

説明･動機

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。

違反サンプル

```
public class BadSample {
   public int sampleMethod(int value){
       int a = value;
       try{
            if( a == 1 ){
                return a;
            }
       }catch(SampleException e){
             e.printStackTrace();
       }finally{
             a = 2;
       }
       return a;        //違反：このメソッドの必ず返り値は 2 になる。
   }
```

3.24. ガーベッジコレクション

C_GC001　finalize() をオーバーライドした場合は super.finalize() を呼び出す

重要度	難易度	修正コスト
★★★	★★	★★

説明･動機

finalize()をオーバーライドしたときに、super.finalize()を呼びましたか？ 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。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。

違反サンプル

```
public class BadSample {
    protected void finalize()
                        throws Throwable {
                                  // 違反
    }
 }
```

修正サンプル

```
public class FixedSample {
    protected void finalize()
                           throws Throwable {
        super.finalize();  // 修正済み
    }
 }
```

C_GC002 アプリケーションから finalize()を呼び出さない

重要度	難易度	修正コスト
★★★	★★	★★

説明･動機

わざわざ finalize()を呼んでいませんか？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。

違反サンプル

```
import java.io.*;

public class BadSample {
    private FileInputStream stream = null;
    public void finalize() throws Throwable
{
        if(stream != null){
            stream.close();
        }
        super.finalize();
    }
    private FileInputStream stream = null;
}

public class BadTest {
    private BadSample sample = new
BadSample();
    void cleanup() throws Throwable {
        sample.finalize();  // 違反
        sample = null;
    }
}
```

修正サンプル

```
public class FixedSample {
    private boolean released = false;
    private FileInputStream stream = null;

    public synchronized void release()
                                throws Throwable{
        if (!released) {
            if(stream != null){
                stream.close();
            }
            released = true;
        }
    }

    public void finalize() throws Throwable {
        release();
        super.finalize();
    }
}

public class FixedTest {
    private FixedSample sample = new FixedSample
();

    void closeTest() throws Throwable {
        sample.release();  // 修正済み
        sample = null;
    }
}
```

3.25. その他

C_MISC001 Cloneable を明示的に実装

重要度	難易度	修正コスト
★★★	★★	★
説明･動機	clone()メソッドを使用する際、Cloneable インターフェースを実装せずに使おうとしていませんか？clone メソッドは java.lang.Object クラスで定義されているため、メソッドとしてはすべてのクラスでサポートされているのですが、Cloneable を実装していないクラスでは java.lang.CloneNotSupportedException がスローされ、使用することができないようになっています。	

違反サンプル

```
public class BadSample {    //違反
    public Object clone(){
        try{
            return super.clone();
        }catch(
            CloneNotSupportedException e){
            throw new InternalError(
                        e.getMessage());
        }
    }
    public static void main(
                            String[] args) {
        CloneSample sample =
                     new CloneSample();
        sample.clone();
    }
}
```

修正サンプル

```
public class FixedSample implements Cloneable
{
                                        //修正済み
        public Object clone(){
        try{
            return super.clone();
        }catch(
            CloneNotSupportedException e){
            throw new InternalError(
                        e.getMessage());
        }
    }
    public static void main(
                            String[] args) {
        CloneSample sample =
                        new CloneSample();
        sample.clone();
    }
}
```

C_MISC002 キャスト処理は instanceof で囲む

重要度	難易度	修正コスト
★★★	★★	★★
説明･動機	キャスト処理は継承関係にあるもの同士しか行えません。継承関係にないものもキャスト処理の対象となりうる場合は、instanceof を用いたチェックを必ず行うようにしてください。	

違反サンプル

```
public class BadSample {
    public void badSampleMethod(
                            Object object){
        Parent parent = (Parent)obj;  //違反
    }
}
```

修正サンプル

```
public class FixedSample {
    public void fixedSampleMethod(
                                Object object){
        if (object instanceof Parent) {
                                        //修正済み
            Parent parent = (Parent)object;
        }else {
        //別処理
        }
    }
}
```