Java编码规范

百度在线网络技术（北京）有限公司

(版权所有,翻版必究)

修改记录

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No | 修改后  版本号 | 修改内容简介 | 修改日期 | 修改人 |
| 1 | V1.0 | 全文 |  |  |
|  |  | 如果系统已编写MRD，写明对应的MRD文档名称 |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

目 录

[1 概述 1](#_Toc383772877)

[1.1 目的 1](#_Toc383772878)

[1.2 适用范围 1](#_Toc383772879)

[2 总体原则 1](#_Toc383772880)

[3 编码规范 1](#_Toc383772881)

[3.1 命名规范 1](#_Toc383772882)

[3.1.1基本约定 1](#_Toc383772883)

[3.1.2包命名 2](#_Toc383772884)

[3.1.3文件命名 3](#_Toc383772885)

[3.1.4类/接口命名 3](#_Toc383772886)

[3.1.5变量命名 4](#_Toc383772887)

[3.1.6方法命名 5](#_Toc383772888)

[3.1.8 JSP/HTML命名规范 6](#_Toc383772889)

[3.2 书写风格规范 7](#_Toc383772890)

[3.3 java类代码编写规范 10](#_Toc383772891)

[3.4潜在BUG隐患 25](#_Toc383772892)

# 概述

## 目的

为了统一部门的编程风格，提高代码的可读性，可维护性，可用性以及作为code review的参考。减少软件维护成本，最终提高软件产品的生产力。特制定Java编码规范。

## 适用范围

本规范对Java源代码有约束力，对于开发工具自动生成的源代码可以不约束。

# 总体原则

* 编码风格的一致性，团队开发过程中，每个成员在编码过程中应当依据编码规范的约定进行，使最终的代码风格一致。由此带来易于互相理解、交流和维护。
* 规范的约束性，作为编码规范，在其适用范围内，一旦采纳，每个成员就应严格遵守，并做为代码质量的一个衡量标准

# 编码规范

## 命名规范

### 3.1.1基本约定

命名应当清晰、明了，有明确含义，使用完整的英文单词或大家可以理解的缩写，能够表达出命名的具体含义，命名应当符合“min-length && max-information”原则，谨慎使用缩写。

说明：

命名应当符合“最小长度且最大信息量”的原则。较好的命名应该象firstName、grandTotal、CorporateCustomer， 而诸如x1、y1等命名反映任何命名含义，因而造成代码难以理解、维护和改进。

命名中应该尽量避免缩写命名，对于标识符应当使用完整的英文进行描述。但对于整个描述较长的，可对单词进行缩略。对单词的缩略可按较短的单词可通过去掉“元音”形成缩写，较长的单词可取单词的头几个字母形成缩写，一些单词有大家公认的缩写，常用单词的缩写必须统一。协议中的单词的缩写与协议保持一致。对于某个系统使用的专用缩写应该在某处做统一说明。设计命名中应该慎用缩写命名。如要采用，则应采用统一的缩略规则，并且在代码的相应部分统一采用缩写。例如，采用num作为number的缩写，那么在整个代码中应该始终使用该缩写。

例如：如下单词的缩写能够被大家基本认可。

　　 temp 可缩写为 tmp ;

flag 可缩写为 flg　；

　 statistic 可缩写为 stat ;

　　 increment 可缩写为 inc　；

　　 message 可缩写为 msg　；

命名最好采用领域的术语

说明：

软件开发人员应注意软件用户的一些约定术语，要尽可能的使用领域里已经存在的术语，不应当随意的创造术语，这会降低软件的易维护性。如果用户称他们的“客户” (clients) 为“顾客” (customers)，那么就采用术语Customer 来命名这个类，而不用 Client。

避免使用太长的命名，最好不超过15个字母

说明：

命名的长度一般小于30个字母，最好不要超过15个字母。名字太长时，谨慎使用缩写。

避免使用相似或者仅在大小写上有区别的名字。

说明：

比如不应同时使用变量名 anSqlDatabase 和 anSQLDatabase。

### 3.1.2包命名

Sun标准java分配包用java.开头，我们的包名以com.baidu.bae开头。

说明：

包名可以采用一个顶级域名（如当前的com、edu、gov、mil、net、org等）开头。包名称的随后成份根据组织自身的命名规范变化。这样的规范或许通过指定分公司、部门、项目、机器、帐户等来确定目录名成分。

包名必须全部采用小写字母。

说明：

例如： com.baidu.bae.processmanager

com.baidu.bae.taskmanager

### 3.1.3文件命名

源文件名必须和文件中的唯一的public类或接口的名称一致，大小写也应该相同。

说明：

Sun的规定，需要特别注意大小写。

### 3.1.4类/接口命名

类名/接口名使用完整的英文单词来命名。

说明：

基本约定，清晰的表达类/接口的用途。

类所有单词的首字母要大写，其他字母小写。

**说明：**

例如 FrontTaskManager

接口命名以大写I开头或者以interface结尾，所有单词的首字母要大写，其他字母小写。

**说明：**

例如 ITaskManager 或者 TaskManagerInterface

继承自Exception类的类名的最后一个单词采用Exception。

说明：

加强命名习惯，通过类名明显知道该类的作用。

例如 ProcessRunException

采用设计模式的类名尽可能反映出模式的意义。

说明：

这种命名易读，而且能通过类名更多的知道类的含义。

例如 ProcessFactory

### 3.1.5变量命名

常量用完整的英文大写单词命名，单词之间用下划线连接。

说明：

用于明显区分常量和变量。

例如 private static final long MAX\_LENGTH = 100000

变量命名应采用完整的英文单词，首字母小写，其他单词的首字母大写。

说明：

基本约定，便于阅读和理解。

数组（array）和vector的命名方法用所存对象名称的复数（加s）表示

说明：

这种约定的主要优点是有助于区分表示复数值（集合）和单值（非集合）的变量。

例如 Customer[] customers;

String[] columnNames;



方法内的局部变量不要与成员变量重名。

说明：

尽管局部变量和全局变量的作用域不同而不会发生语法错误，但容易使人误解。

数据流的命名以in或out或inOut打头。

说明：

这是Sun公司的约定。

**输入流：**命名用in +“流名称”；

**输出流：**命名用out +“流名称”；

**输入/输出流：**命名用inOut +“流名称”;

循环计数器可以采用 i， j 或 k 作为循环计数器。

说明：

习惯用法，方便编写。

异常对象：字母e/ex作为一般的异常符被广泛地接受，建议使用。

说明：

习惯用法，方便编写。

### 3.1.6方法命名

成员方法名称应采用完整的英文单词组成；首字母小写，除第一个单词外其余的成员单词的首字母用大写；成员方法取名采用动宾结构，名称中的第一个词为动词。

说明：

方法名力求清晰、明了，通过方法名就能够判断方法的主要功能。方法名中不同意义成员变量之间不要用下划线连接，而要把每个成员变量的首字母大写以示区分。方法命名采用大小写字母结合的形式，但专有名词不受限制。

获取器方法以get\*\*\*（成员变量名）命名，设置器方法以set\*\*\*（成员变量名）命名。

说明：

获取器方法作为一个成员方法，返回一个成员变量的值。设置器方法，也叫变值方法，是可以修改一个成员变量值的成员方法。

获取器方法返回值是boolean型的要以”is”打头，提高代码可读性。

说明：

例如

public class GETBFixed{

private boolean ready = false;

public boolean isReady () { //正确

return ready;

}

}

用正确的反义词组命名具有互斥意义的相反动作的方法。

说明：

下面是一些在软件中常用的反义词组：

add / remove ； begin / end ； create / destroy ； insert / delete

first / last ； get / set ； increment / decrement ； put / get

add / delete ； lock / unlock ；open / close ； min / max

old / new ； start / stop ；next / previous ；source / target

### 3.1.8 JSP/HTML命名规范

jsp和html文件名全部小写，并遵循如下规范: 数据/内容显示页: 名词要求能说明显示内容的信息，如：newmessage.html , myfile.jsp 操作处理页：命名为名次动词形式，如filedelete.jsp

## 书写风格规范

文件的每个主要部分需增加注释，各个部分之间用空行分隔。

说明：

源文件有头注释，注明作者、版权、版本、完成日期等信息，提高代码可维护性。



源代码文件(.java)的布局顺序是：文件注释、包（package语句）、import语句、public类注释、public类声明和定义、other private类注释、private类定义。

说明：

* **文件注释：**

文件的最开始应该是文件注释，文件注释的内容和格式参见对文件注释的说明。

* **package:**

package语句是文件注释以外的第一句，包名为小写字母，通过.分级。

* **imports:**

import 中的包的顺序：

* + 标准包
  + 第三方的包
  + 本地包

并且包名按照字母顺序排列。如果 import 行中包含了同一个包中的不同子目录，则应该用 .\* 来处理。

package和import 之间空一行。

标准包、第三方包和本地包的import之间有空行。

文件的长度一般不应该超过2000行。每个函数不超过200行，每行不超过80个字符

说明：

为了增强代码可读和可维护性，对于文件，函数，以及每行的代码规模都应做一定限度，如果函数逻辑复杂应该可以调用子函数，不应将所有代码在一个函数中实现

用空格(4个空格)替代tab键，4个空格为一缩进单位。

说明：

保证在不同的编译和查看工具下，代码格式保持统一。避免不同编辑器之间的差异而导致代码布局混乱。

长表达式（超过80列）要在低优先级操作符处拆分成新行，操作符放在新行之首（以便突出操作符）。拆分出的新行要进行适当的缩进，使排版整齐。

说明：

断行方法：

* 在逗号后断行。
* 在操作符前断行。
* 低优先级操作符断行优于高优先级操作符断行。
* 较高级别断行优于较低级别的断行。

对齐方法：

* 将新行与同一级别的先前行的表达式的起始端对齐。
* 条件表达式的续行在第一个条件处对齐。
* for循环语句的续行在初始化条件语句处对齐。
* 方法调用和方法声明的续行在第一个参数处对齐。
* 赋值语句的续行应在赋值号处对齐。

如果上述规则导致代码排列混乱或代码左边界少于两个缩进，可用两倍缩进替代。下面是一些断行方法调用的示例：

someMethod(longExpression1, longExpression2, longExpression3,

longExpression4, longExpression5);

var = someMethod1(longExpression1,

someMethod2(longExpression2,

longExpression3));

方法之间要有一空行，不同的逻辑块之间也要有空行，在定义类和接口之前要有空行，提高代码可读性。

定义方法时，方法名字和”( )”之间没有空格。

说明：

方法名和紧跟其后的（之间不加空格，这样可以帮助区分关键字和方法调用。

例如：

public void method(){

}

public void method2(){

}

在一元(“++”,”--”)运算符和变量之间不允许有空格，提高代码可读性。

在位运算符(“>>”,”<<”,”|”,”^ ”，“&”)和变量之间要有空格。 赋值操作符（=, \*=, &=, +=, %=, -=, /=, <<=, >>=, ^=, |=.）前后都要有空格

每行只有一个语句，提高代码可读性。

说明：

反例：

public class OSPL{

private int method (int a, int b) {

int i = a + b; return i; //违例

}

正例：

public class OSPL{

private int method (int a, int b) {

int i = a + b; //正确

return i; //正确

}

if/if-else/if else-if else 语句书写永远加{}。



确保标识符的顺序如下：public/protected/private、abstract、static、final、transient、volatile、synchronized、native、strictfp

## java类代码编写规范

成员变量、成员方法应当恰当的采用public/private/protected/default关键字声明，尽量将信息的暴露降到最低。

说明：

不必要的数据暴露容易造成数据操作的错误，因此，成员变量、成员方法应该使用恰当的访问权限修饰符，保证类的封装性，避免数据不必要的暴露。

一般来说实例变量应该定义为private，如果需要外部访问，可以通过提供相应的读写方法来实现。

反例：

public class AF {

public int a = 10; // 违例。 a是类AF的内部实例变量，不应该向外部暴露

private int c = 14;

}

**正例：**

public class AFFixed {

private int a = 10; //正确

private int c = 14;

public void setA (int a) { // 正确

this.a = a;

}

public int getA () { //正确

return a;

}

}

工具类应该只声明private构造方法，且要声明默认的构造方法。

说明：

由于工具类中包含的都是静态变量和方法，不会被实例化，所以不需要声明非private构造方法。

由于隐含的默认构造方法是public的，所以需要显式声明默认的构造方法，且声明为private。

如果一个类只有private构造方法，则应该将其声明为final类型以便避免其他类的继承。

说明：

因为一个类只有private构造方法时子类无法继承它的构造函数，这样造成子类的构造函数会出现问题，因此应该避免该类被继承。

对于字符串常量，使用String而不是StringBuffer来表达。

说明：

对于可变长度的字符串使用StringBuffer效率更高，对于不变的字符串使用String更好。

反例：

public class USC {

private String method() {

StringBuffer s = new StringBuffer ("Hello"); // 违例，常量字符串

String t = s + " World!";

return t;

}

}

正例：

public class USCFixed {

private String method() {

String s = "Hello"; // 修正

String t = s + " World!";

return t;

}

}

对于可以简化的赋值，应该使用简化方式。

反例：

public class AAS {

private void method() {

int i = 0;

i = i - 1; // 违例:可以简化为"-="

}

}

正例：

public class AASFixed {

private void method() {

int i = 0;

i -= 1;

}

}

类的成员变量如果只被一个方法使用，则应该将成员变量声明为局部变量。

将类内部使用的函数、变量全部定义为private类型，以免外部错误调用。

对于String对象避免调用toString()方法，因为没有必要。

对于continue和break语句避免使用标签跳转。

说明：

由于这种跳转会使程序变得复杂，容易出现错误。

在catch语句块中避免使用instanceof来判断异常类型，对于每一类异常，应该分别用不同catch块处理。

说明：

这样的处理程序清晰且不易遗漏异常。

优先通过接口引用对象，将参数、返回值、变量等声明为接口以便降低耦合，提高程序的灵活性。

说明：

在声明参数、返回值、变量等时，将其声明为接口类型可以减少代码间的依赖，降低程序代码之间的耦合，提高代码的灵活性。

反例：

public class AUVT {

private static final MAX\_CAPACITY = 10;

private ArrayList list = new ArrayList(); //违例，list应该声明为List类型来降低耦合

public void addElement(Object element) {

if (list.size() < MAX\_CAPACITY){

list.add(element);

}else{

System.out.println("list is filled.");

}

}

}

正例：

public class AUVT {

private static final MAX\_CAPACITY = 10;

private List list = new ArrayList(); //正例

public void addElement(Object element) {

if (list.size()< MAX\_CAPACITY){

list.add(element);

}else{

System.out.println("list is filled.");

}

}

}

用大写字母命名常量，并且将其声明为static final。

在表达式中尽量避免使用自增（++）/自减（--）操作符以提高代码可读性。

说明：

在表达式特别是在那些复杂的表达式中使用自增（++）/自减（--）操作符会使代码变得复杂，不易理解。这些表达式包括条件表达式、赋值表达式等。

反例：

public class A {

private int count = 0;

public void calculate(int mod) {

count = (mod++ \* 2 ) / 9;

}

}

正例：

public class A {

private int count = 0;

public void calculate(int mod) {

int tmp = mod++;

count = (tmp \* 2 ) / 9;

}

}

如果一个类有多个构造函数，则构造函数间尽量复用避免重复代码。

反例：

public class CHAIN {

private String streetName;

private String streetName2;

private String aptNum;

private String city;

private String state;

private int zip;

private String country;

public CHAIN(String streetNameL, String streetName2L, String aptNumL, String cityL, String stateL,

int zipL, String countryL ) {

streetName = streetNameL,

streetName2 = streetName2L,

aptNum = aptNumL;

city = cityL;

state = stateL;

zip = zipL;

country = countryL;

}

public CHAIN(String streetNameL, String aptNumL, String cityL, String stateL, int zipL ) { // 违例，没有复用已有代码

streetName = streetNameL,

streetName2 = "",

aptNum = aptNumL;

city = cityL;

state = stateL;

zip = zipL;

country = "USA";

}

}

正例：

public class CHAINFixed {

private String streetName;

private String streetName2;

private String aptNum;

private String city;

private String state;

private int zip;

private String country;

public CHAIN(String streetNameL, String streetName2L, String aptNumL, String cityL, String stateL,

int zipL, String countryL ) {

streetName = streetNameL,

streetName2 = streetName2L,

aptNum = aptNumL;

city = cityL;

state = stateL;

zip = zipL;

country = countryL;

}

public CHAIN(String streetNameL, String aptNumL, String cityL, String stateL, int zipL ) {

this(streetNameL,"", aptNumL, cityL, stateL, zipL, "USA"); // 正例

}

}

避免在一个声明语句中声明多个不同类型变量，比如一个整型变量，一个整型变量数组；避免在一个声明语句中声明多个变量。这样清晰且方便注释。

反例：

public class VDT{

private int index, index1[]; //违例：一个整型，一个整型数组

public void method() {

int aaa, bbb[]; //违例：一个整型，一个整型数组

int ccc;

int ddd;

}

}

public class MVOS {

private String s1，s2; //违例：同一个声明语句声明两个变量

public MVOS() {

s1 = "hello";

s2 = "world";

}

}

正例：

public class VDTFixed {

private int index;

private int[] index1; //正确

public void method() {

int aaa;

int[] bbb; //正确

int ccc;

int ddd;

}

}

public class MVOSFixed {

private String s1; //正确

private String s2; //正确

public MVOSFixed (){

s1 = "hello";

s2 = "world";

}

}

避免直接使用数字常量，用大写单词命名的静态常量替代数字常量，提高代码可读性。

说明：

指定的常量的采用可以使得代码更容易被理解与维护。

反例：

public class USN {

public int[] getArray () {

return new int [1000]; //违例

}

}

正例：

public class USNFixed {

private static final int ARRAY\_SIZE = 1000;

public int[] getArray() {

return new int [ARRAY\_SIZE]; //正确

}

}

赋值语句：避免嵌套赋值，避免在一条语句中将一个数值赋给多个变量，提高代码可读性。

反例：

public class NEA{

private int method(int i, int j) {

int k = (i = i + j) + j; //违例

return k;

}

}

正例：

public class NEAFixed {

private int method(int i, int j) {

i = i + j; //正确

int k = i + j; //正确

return k;

}

}

如果一个方法没有使用任何实例类成员（例如类的非静态方法和成员变量）,那么就把方法声明为静态的，提高代码的封装性。

反例：

public class ASI {

public int add(int i, int j) { //违例

return i + j;

}

}

正例：

public class ASIFixed {

public static int method(int i, int j) { //正确

return i + j;

}

}

如果类声明中有main方法，则把其放在类的最后; main(String[] str)仅仅用于程序入口点方法。

说明：

类中避免存在除main(String[] str)之外的其他main方法，否则容易引起混淆。

反例：

public class Main {

public static void main (String[] args) {

System.out.println ("This is the main method.");

}

public static void main () { //违例

System.out.println ("This is another method named main.");

}

}

正例：

public class Mainfixed {

public static void otherMain () { //正确

System.out.println ("This is another method, now named otherMain.");

}

public static void main (String[] args) {

System.out.println ("This is the main method.");

}

}

在初始化过程中，避免使用非常量静态成员变量初始化其他变量，因为静态成员变量的生存期是整个系统的过程，在这一过程中其值可能被修改。

反例：

public class NFS {

static int max = 10;

int size = max; //违例

}

正例：

public class NFS{

public static final int MAX = 10; //正确

int size = NFS.MAX; //正确

}

在所有的静态变量被赋值以前尽量避免在静态初始化中对当前类进行实例化。

反例：

public class SICUI {

static {

MyClass obj= new SICUI(); //违例，因为此时会调用该类的构造函数，而此时Vector还没有被初始化

}

private static final Vector emptyVector= new Vector();

SICUI() {

System.out.println(emptyVector.size());//此处会抛出NullPointerException，因为emptyVector还没有被初始化

}

}

正例：

public class SICUIFixed {

private static final Vector emptyVector= new Vector();

static {

MyClass obj = new SICUI(); //正确

}

SICUIFixed() {

System.out.println(emptyVector.size());

}

}

避免对成员变量的循环初始化，因为这样的初始化具有不确定性且难于理解。

反例：

public class IC {

public static void main(String args[]) {

IC2 ref2 = new IC2(); //如果代码顺序改变，结果将会不同

IC1 ref1 = new IC1();

System.out.println(IC1.j + " "+ IC2.i);

}

}

public class IC1 {

public static int j= IC2.i+ 1; //违例

}

public class IC2 {

public static int i= IC1.j; //违例

}

正例：

public class ICFixed {

public static void main(String args[]) {

IC2 ref2= new IC2();

IC1 ref1= new IC1();

System.out.println(IC1.j + " "+ IC2.i);

}

}

public class IC1 {

public static int j= IC2.i+ 1; //正确

}

public class IC2 {

public static int i= 0; //正确

}

将公共和保护接口最小化，尽量声明方法为private。

说明：

面向对象程序设计的基本点之一是最小化一个类的公共接口。

对于以值传递的方法参数，避免对其进行赋值。

说明：

当参数值在某个方法中用多次时，给该方法参数赋值会产生潜在的bug。

反例：

public class AFP {

private int method (int low, int high) {

int count = 0;

while (low++ < high) { //违例

count += low;

}

return count;

}

}

正例：

public class AFPFixed {

private int avg (int x) {

int count = 0;

int i = x;

while (i++ < 10) {

count += i;

}

return count % i;

}

}

不要使用异常处理来作为程序的流程控制。

反例：

public class AEFC {

public void myMethod() {

try {

while (true) {

if (i< 0){

throw new PrevertedException(); //违例

}

//do something

}

} catch (PrevertedException e) {

//do something

}

}

}

正例：

public class AEFCFixed {

public void myMethod() {

while (i >= 0) { //正确

//do something

}

}

}

对于private的常量应该声明为final。

反例：

public class FF {

private int size = 5; // 常量没有定义成final

private int method (int x) {

return x + size;

}

}

**正例：**

public class FFFixed {

private static final int SIZE = 5; // 定义为static final

private int method (int x) {

return x + SIZE;

}

}

对于一个子系统的代码，在一个接口或工具类中定义所有常量，提高代码的可维护性。

反例：

public class DCI {

public static final int MAX = 1000; //违例

public int getMax() {

return MAX;

}

}

正例：

public class DCI {

public int getMax() {

return Constants.MAX;

}

}

public interface Constants {

public static final int MAX = 1000;

}

尽可能的应用设计模式。

说明：

已有的设计模式总结出了不少代表性的模型，应用这些模式来设计自己的类和接口不但提高效率，而且程序的质量会有较大的提高。有关设计模式的详细介绍可参考相关文章。

建议每个类都定义一个toString()方法。

说明：

虽然java.lang.Object提供了toString方法的实现，但是它返回的字符串通常并不是类用户所期望看到的，所以一般建议在写每个类时都提供一个toString方法，以便将类用户感兴趣的信息返回。

在一个方法中，“!”操作符使用最好不要超过三次。除非逻辑需要，尽量不要使用技巧性太高的代码，不易懂，维护容易出现问题

说明：

非逻辑的代码不易读，所以建议避免使用次数太多。

反例：

public class DUN {

boolean method(boolean a, boolean b) { // VIOLATION

if (!a){

return (!a && !b);

}else{

return !b;

}

}

}

正例：

public class DUNFixed {

boolean method(boolean a, boolean b) {

if (a){

return !b;

}else {

return (! (a || b) );

}

}

}

不要编写多种功能集于一身的函数，为了对参数进行更强的确认，要编写功能单一的小函数。使类尽可能短小精悍，而且只解决一个特定的问题。

对于可能出现异常的地方要加注解。

## 3.4潜在BUG隐患

try、catch、finally语句块不能为空，如果try..catch块有返回值，则****finally块避免使用返回值****。

说明：

如果catch语句块为空，那么在try语句产生异常时，会把异常给屏蔽掉，导致异常捕获不到，产生意想不到的后果。

因为finally块总是被执行，所以如果try-catch块有返回值，则finally块的返回值会将其覆盖。

反例：

public class AECB {

public void method () {

try {

System.in.read ();

} catch (java.io.IOException e) {

//违例

}

}

}

正例：

public class AECBFixed {

public void method () {

try {

System.in.read ();

} catch (java.io.IOException e) {

System.out.println("Descriptive error"); //正确

}

}

}

避免在”if”条件中赋值。

说明：

Java编译器对boolean型变量在if条件语句中的赋值是合法的，对整型变量的赋值是不合法的，所以下面的反例在语法上是正确的，但不提倡这样写代码。

反例：

public class ASI {

public int foo (boolean b) {

int ret = 0;

if (b = true) { //违例，赋值语句

ret = 3;

}

return ret;

}

}

正例：

public class ASIFixed {

public int foo (boolean b) {

int ret = 0;

if (b) { //正确

ret = 3;

}

return ret;

}

}

在for循环语句体中，避免对循环控制变量赋值。

反例：

public class FLVA {

private int method() {

int sum = 0;

for (int i = 0; i < 100; i++) {

i += 3; //违例

sum += i;

}

return sum;

}

}

正例：

public class FLVA {

private int method() {

int sum = 0;

for (int i = 0; i < 100; i += 4) { //正确

sum += i;

}

return sum;

}

}

避免控制语句没有对应的语句。

反例:

public class EB {

private void method (int i) {

if ( i < 0) ; // 违例

System.out.println("negative i"); // 反例，由于if 后面跟了空语句，所以该语句始终被执行

}

}

避免强制转换高精度类型为低精度类型。

说明：

存在精度丢失的风险。

反例：

public class CLP {

private void method() {

double d = 4.25;

int i;

i = this.square ((int) d); //违例

}

private int square(int i) {

return (i \* i);

}

}

避免使用浮点型变量做”等于”比较。

反例：

public class DCF {

private int method (double d) {

if (d == 1) { //违例

return 1;

}else {

return 0;

}

}

}

正例：

public class DCFFixed {

private int method (int d) { //正确

if (d == 1) {

return 1;

}else {

return 0;

}

}

}

在进行对象比较时，尽量将常量或者明确知道不能为null的量放在比较的左边。

说明：

这样可以避免空指针异常，同时也能避免在比较语句中赋值。

反例：

public class CLS {

private String v = “”;

public void testMethod (int something) {

if (something == 5) { // 违例，容易出现something = 5的错误

}

}

public void testMethod1 (String something) {

if (something.equals(v)) {// 违例，容易出现空指针异常

}

}

}

正例：

public class CLS {

private String v = “”;

public void testMethod (int something) {

if (5 == something) { // 正例

}

}

public void testMethod1 (String something) {

if (v.equals(something)) { //正例，因为v明确知道非空，不会出现空指针异常

}

}

}

避免类成员的名字和方法中的参数冲突，提高代码可读性。

反例：

public class MPC {

private int i = 0;

private void method (int i, int j) { //违例

}

void j() {

}

}

正例：

public class MPCFixed {

private int i = 0;

private void method (int first， int second) { //正确

}

private void j() {

}

}

避免类方法和构造方法同名，提高代码可读性。

反例：

public class NAMING {

public NAMING() {} //构造方法

public void NAMING(int size) { //违例，非构造方法和构造方法重名

}

}

正例：

public class NAMINGFixed {

public NAMINGFixed () {} //构造方法

public anotherNaming (int size) { //正确

}

}

避免直接或间接的定义扩展Error或Throwable或RuntimeException异常，用户定义的异常应该继承自java.lang.Exception。

说明：

用户定义的异常应该继承自java.lang.Exception。

反例：

public class NDCException extends Error { //违例

public NDCException(String s) {

super(s);

}

}

public class NXRE extends RunTimeException { //违例

public NXRE(String s) {

super(s);

}

}

正例：

public class NDCExceptionFixed extends Exception { //正确

public NDCExceptionFixed(String s) {

super(s);

}

}

public class NXREFixed extends Exception { //正确

public NXREFixed(String s) {

super(s);

}

}

避免直接抛出Exception异常，避免抛出或捕捉Error或NullPointerException。

**说明：**

Error/Exception：包含的范围太大，处理时容易出现问题，因此不建议这样使用。

NullPointerException：VM会抛出该异常，往往出现这种异常的时候程序可能会存在问题，建议用户不要直接使用该异常以免将可能的问题蒙蔽掉。

反例：

public class NTX {

public void lostDetailsTest() throws Exception { //违例

throw new java.io.IOException ("IO exception");

}

}

正例：

public class NTXFixed {

public void lostDetailsTest() throws java.io.IOException { //正确

throw new java.io.IOException ("IO exception");

}

}

对每个switch语句都要有default分支，每个分支语句都要包括break语句（除非设计需要可以不包括break，比如两个case分支处理相同时），以便提高代码可读性，防止潜在的bug。

反例：

public class PDS {

public void method (int i) {

switch (i) { //违例

case 0:

System.out.println ("Zero");

break;

}

}

}

public class SBC {

public int method (int i) {

int x = 0;

switch (i) {

case 1:

x = 10;

break;

case 2: //违例

x = 20;

default:

x = 40;

break;

}

}

}

正例：

public class PDSFixed {

public void method (int i) {

switch (i) {

case 0:

System.out.println ("Zero");

break;

default: //正确

break;

}

}

}

public class SBCFixed {

public void method (int i) {

int x = 0;

switch (i) {

case 1:

x = 10;

break;

case 2: //正确

x = 20;

break;

default:

x = 40;

break;

}

}

}

在switch语句中，避免把case分支写成文本标签。

反例：

public class TLS {

public static int method (int i) {

switch (i) {

case 4:

case3: //违例:

i++;

break;

case 25:

unusedlabel: //违例

break;

}

return i;

}

}

正例：

public class TLSFixed {

public static int method (int i) {

switch (i) {

case 4:

case 3: //正确

i++;

break;

case 25: //正确

break;

default:

break;

}

return i;

}

}

对象比较时，使用equals ()方法，而不是 ==

反例：

public class UEI {

private Choice co = null;

public boolean notEqual (Choice other) {

boolean monthly = co.getSelectedItem () != other; //违例

return monthly;

}

}

正例：

public class UEIFixed {

private Choice co = null;

public boolean notEqual (Choice other) {

boolean monthly = !(co.getSelectedItem ().equals (other)); //正确

return monthly;

}

}

避免使用'System.getenv ()'方法。

说明：

因为并不是所有的平台都有环境变量，而且该方法用起来也不方便。

反例：

public class ENV {

public void method (String name) {

System.getenv (name); //违例

}

}

正例：

public class ENVFixed {

public void method (String name) {

System.getProperty (name); //正确

}

}

避免使用字符串拼接操作符“+”来操作数字，以免出现异常。

说明：

在对字符串进行“+”操作时会调用相应对象的toString方法，操作数字可能会产生意想不到的结果。

反例：

public class DCP {

public static void main (String args []) {

System.err.println ("2+9 = " + 2 + 9); // 违例，这将输出2+9=29，和预期不符

}

}

正例：

public class DCPFixed {

public static void main (String args []) {

System.err.println ("2+9 = " + (2 + 9)); // 修正，将2+9放到()里

}

}

返回值以零长度数组替代null以便避免异常。

反例：

public class ARN{

private List slist = new ArrayList();

public String[] getStringList () {

if (slist.size() == 0){

return null; //违例

}

return (String[])slist.toArray (new String[slist.size ()]);

}

public void find () {

ARN a = new ARN ();

String[] list = a.getStringList ();

if (list != null) { //需要判空

if (Arrays.asList (list).contains ("Hello")) {

System.out.println ("find");

}

}

}

}

正例：

public class ARNFixed {

private List slist = new ArrayList();

public String[] getStringList () {

return (String[]) slist.toArray (new String[slist.size()]); //正确

}

public void find () {

ARNFixed a = new ARNFixed ();

String[] list = a.getStringList ();

if (Arrays.asList (list).contains ("Hello")) {

System.out.println ("find");

}

}

}

在构造函数调用前或者调用中避免使用未初始化的变量。

反例：

public class UIRC {

Vector v; //未初始化的变量

UIRC(Object singleElement) {

v.add(singleElement); //违例，会出现空指针

}

}

**正例 ：**

public class UIRCFixed {

Vector v;

UIRC(Object singleElement) {

v= new Vector(); //正确

v.add(singleElement); //正确

}

}

避免使用public static final数组成员变量，因为这样的定义只是说明数组本身不能被修改。

说明：

public static final数组是Object，其自身引用不能被修改，但是其数组内容可能会被修改。

反例：

public class PSFA {

public static final int[] MUTABLE = { 0 }; //违例

public static void main (String[] args) {

MUTABLE[0] = 1;

System.out.println ("MUTABLE[0]=" + MUTABLE[0]);

}

}

正例：

public class PSFAFixed {

private static final int[] MUTABLE = { 0 };

public static final List unmutable =

Collections.unmodifiableList (Arrays.asList(MUTABLE));

}

避免在嵌套的循环内增加/减少同一个变量。

在声明处显式初始化所有的局部变量，这样可以避免潜在的bug。

反例：

public class LV {

public boolean method (int size) {

int max; // 违例，没有进行初始化

return size > 0;

}

}

正例：

public class LVFixed {

public boolean method (int size) {

int max = 0; // 正确

return size > max;

}

}

建议在equals()方法处理中使用getClass()方法。

说明：

对象equals的首要条件是：类型相同。否则强制类型转换将产生无法预计的后果。

反例：

public class EQL {

public int value;

public boolean equals (Object o) {

EQL temp = (EQL)o;

return temp.value == value; //违例

}

}

正例：

public class EQLFixed {

public int value;

public boolean equals (Object o) {

if (getClass () == o.getClass ()) { //正确

EQLFixed temp = (EQLFixed)o;

return temp.value == value;

}else {

return false;

}

}

}

使用静态变量时，谨慎考虑多线程情况下的异常情况。

不能假定传入的参数是合法的。各函数在进行处理前要检查参数的合法性。不能假定获得的对象引用是合法的，要检查方法的返回的对象句柄是否为null。