

# Java编程规范

## 1. 概述

### 1.1 编写目的

本规范的目的是使开发人员能以标准的、规范的方式设计和编码。通过建立编码规范，使每个开发人员养成良好的编码风格和习惯，规范开发人员行为，从而提高程序的可靠性、可读性、可修改性、可维护性和一致性等，增进团队间的交流，并保证软件产品的质量。

### 1.2 声明

本规范并不是终稿，本规范会伴随着开发人员的成长而变化，任何开发人员认为不合理或需要添加的规范都会经过讨论之后，添加到本规范中或从本规范中删除。

### 1.3 参考资料

Java开发规范 Java Development Specification 2.0

Java Programming Style Guidelines

## 2. 代码风格

### 2.1 基本原则

便于自己开发，增加代码的可读性，易于与他人的交流，代码风格前后一致，并且在不同的编辑器下风格一致。

### 2.2 缩进

子功能块当在其父功能块后缩进。

当功能块过多而导致缩进过深时当将子功能块提取出来做为子函数。

代码中以TAB（4个字符）缩进，在编辑器中请将TAB设置为以空格替代，否则在不同编辑器设置下会导致TAB长度不等而影响整个程序代码的格式。

### 2.3 长度

为了便于阅读和理解，单个函数的行数不宜超过一个屏幕，单个类的行数不宜超过1500行。

### 2.4 行长度

尽量避免一行的长度超过80个字符。当一个表达式无法容纳在一行内时，可以依据如下一般规则断开：

- 在一个逗号后面断开
- 在一个操作符前面断开
- 宁可选择较高级别(higher-level)的断开，而非较低级别(lower-level)的断开
- 新的一行应该与上一行同一级别表达式的开头处对齐
- 如果以上规则导致你的代码混乱或者使你的代码都堆挤在右边，那就代之以缩进8个空格。

以下是断开方法调用的一些例子：

```

someMethod(longExpression1, longExpression2, longExpression3,
           longExpression4, longExpression5);
var = someMethod1(longExpression1,
                 someMethod2(longExpression2,
                             longExpression3));

```

以下是两个断开算术表达式的例子。前者更好，因为断开处位于括号表达式的外边，这是个较高级别的断开。

```

longName1 = longName2 * (longName3 + longName4 - longName5)
                + 4 * longname6; //PREFER

longName1 = longName2 * (longName3 + longName4
                        - longName5) + 4 * longname6; //AVOID

```

以下是两个缩进方法声明的例子。前者是常规情形。后者若使用常规的缩进方式将会使第二行和第三行移得很靠右，所以代之以缩进8个空格

```

//CONVENTIONAL INDENTATION
someMethod(int anArg, Object anotherArg, String yetAnotherArg,
           Object andStillAnother) {
    ...
}

//INDENT 8 SPACES TO AVOID VERY DEEP INDENTS
private static synchronized horkingLongMethodName(int anArg,
           Object anotherArg, String yetAnotherArg,
           Object andStillAnother) {
    ...
}

```

if语句的换行通常使用8个空格的规则，因为常规缩进(4个空格)会使语句体看起来比较费劲。比如：

```

//DON'T USE THIS INDENTATION
if ((condition1 && condition2)
    || (condition3 && condition4)
    ||!(condition5 && condition6)) { //BAD WRAPS
doSomethingAboutIt();                //MAKE THIS LINE EASY TO MISS
}

//USE THIS INDENTATION INSTEAD
if ((condition1 && condition2)
    || (condition3 && condition4)
    ||!(condition5 && condition6)) {
doSomethingAboutIt();
}

//OR USE THIS
if ((condition1 && condition2) || (condition3 && condition4)
    ||!(condition5 && condition6)) {
doSomethingAboutIt();
}

```

## 2.5 配置代码风格模板

### 2.5.1 设置页宽

在eclipse中选择“Window”中的“Preferences”。在弹出的窗口中选择“General—Editors—TextEditors”。在右侧弹出窗口中选中“show print margin”，并且设置“print margin column”为140，然后点击右下方的“Apply”按钮！

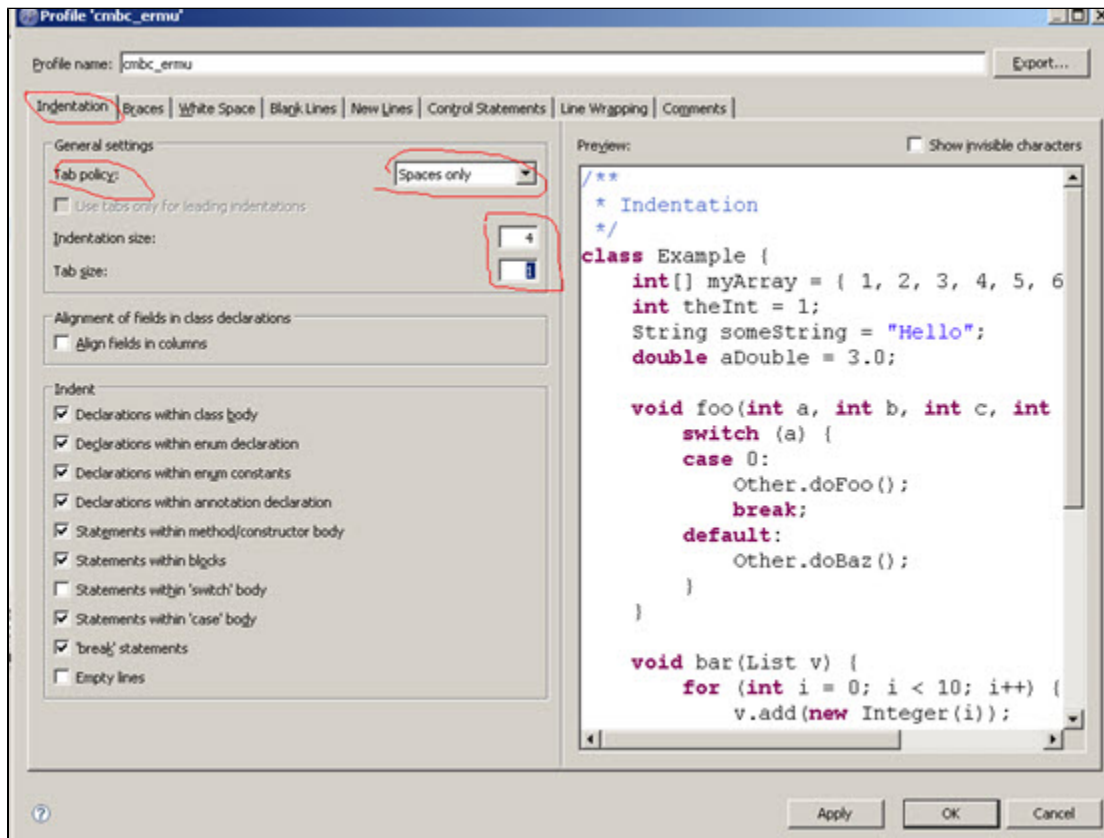
### 2.5.2 代码风格设置

选择选择Windows->Preference。在弹出的窗口中选择Java->Code Style->Formatter。

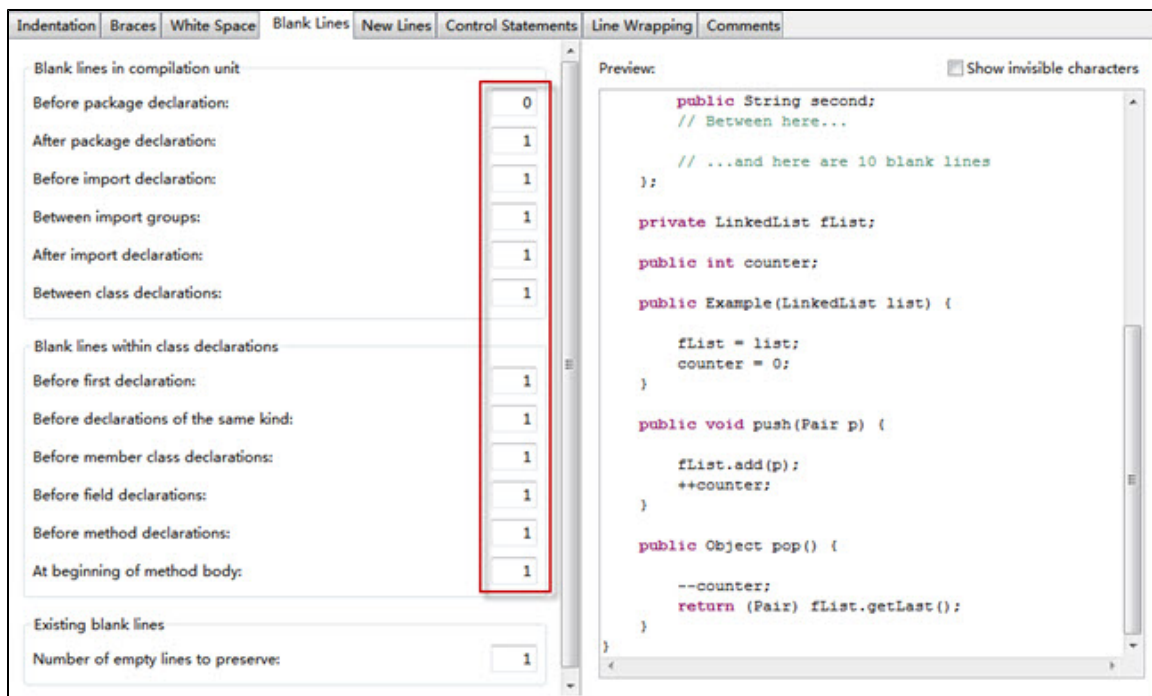
在右侧显示区域中，选中” new ”按钮创建新规范的名字。

在此可以设置代码” 缩进 ”风格：

将Tab policy 设置为Space only，将Indentation size 设置为4，把Tab size设置为1



点击” Apply ”按钮。



设置完成之后，选择Blank Lines空行设置：

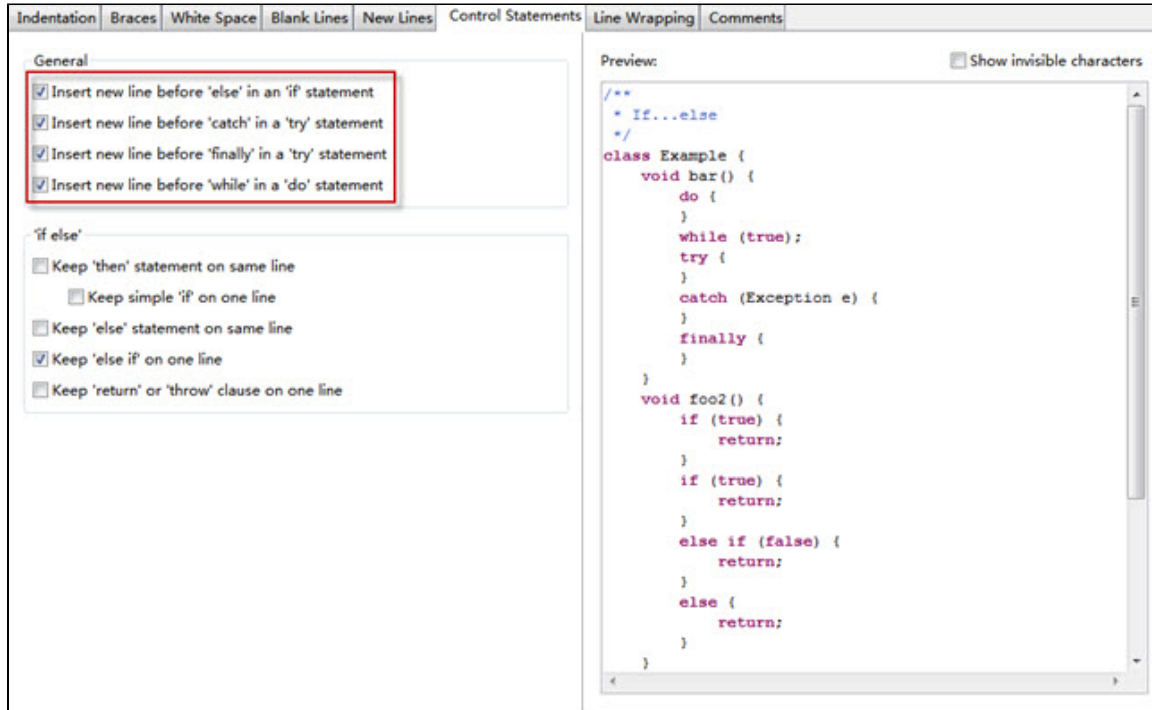
选择Control Statements, 勾选以下四项

Insert new line before 'else' in an 'if' statement

Insert new line before 'catch' in a 'try' statement

Insert new line before 'finally' in a 'try' statement

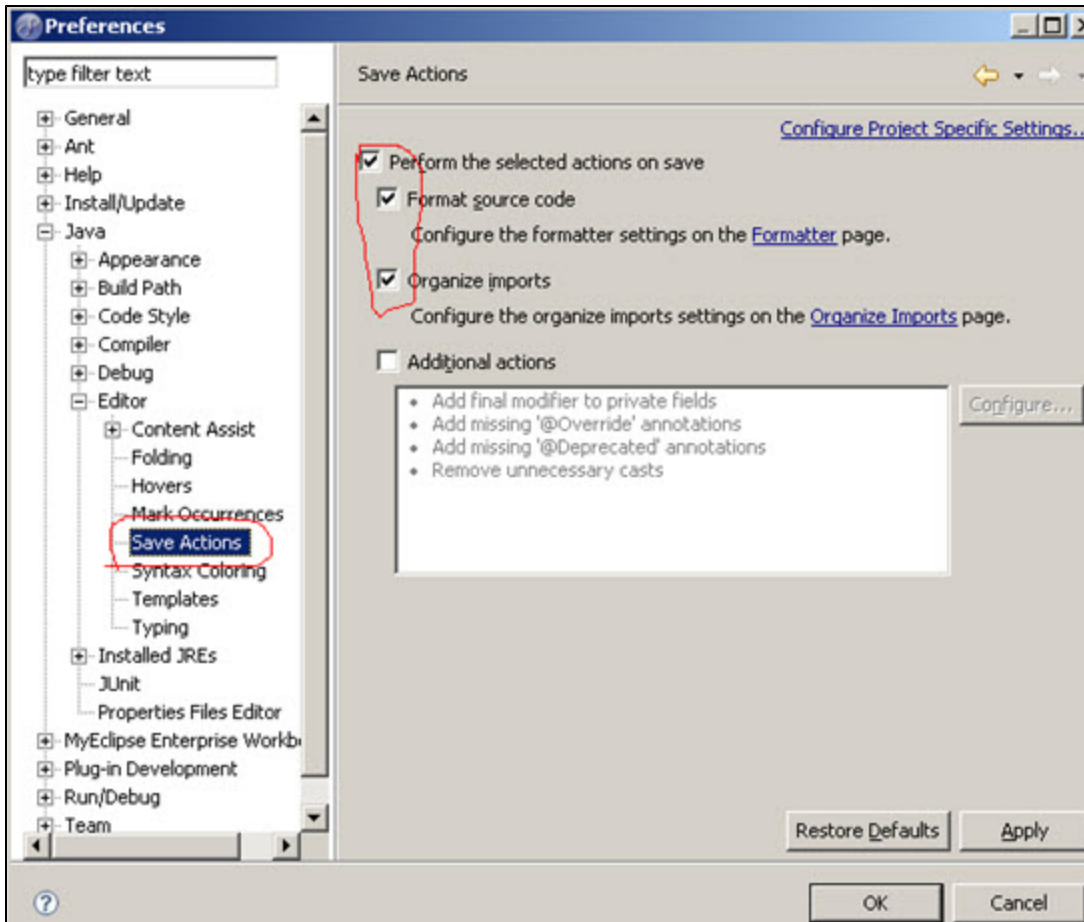
Insert new line before 'while' in a 'do' statement



### 2.5.3 格式化代码

手动格式化代码: 编写完程序之后, 可以用快捷键” Ctrl+Shfit+F”, 或者点击菜单中Source中的子菜单Format, 则可以格式化代码。

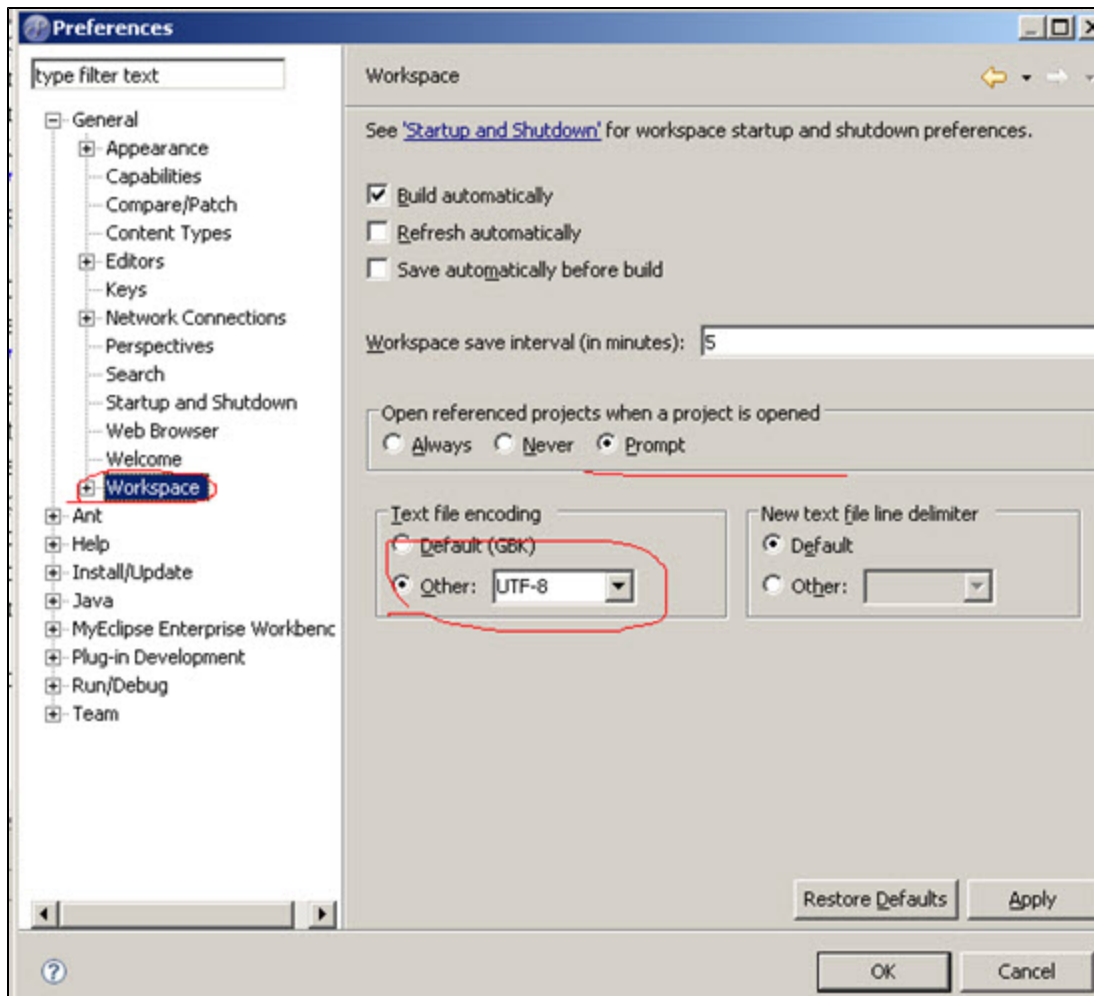
自动格式化代码: 选择菜单window中首选项Preferrences; 在弹出的窗口中, 选择节点java, Editor, Save Actions。在右侧显示中, 选中” Perform the selected actions on save ”项及子项” Format source code ”和” Organize imports”, 然后选中” Apply” 和” OK” 按钮即可! 如下图:



## 2.5.4 将eclipse设置为UTF-8格式

必须将所有的java代码的编码格式都用UTF-8格式

选择菜单window中首选项Preferences，在弹出的框中选中右侧树形结构中的General，然后选择下方的Workspace，在右侧区域中找到Text file encoding，修改该配置项：选择other，并选中下拉列表中的”UTF-8”，点击”Apply”按钮。如下图：



### 3. 命名规范

#### 3.1 一般性规范

- a) 在名称中使用首字母缩写词时，只有缩写词的首字母缩写，其他部分小写，比如：

```
exportHtmlSource(); // NOT: exportHTMLSource();  
openDvdPlayer();    // NOT: openDVDPlayer();
```

- b) 在很大范围内有效的变量使用长名字，在小范围内有效的变量使用短名字（比如l/j/k/m/n/c/d等）  
c) 在调用一个类的方法时，类的名称是隐含的，因此类的名称不应该出现在方法名中。

#### 3.2 包命名

一个唯一包名的前缀总是全部小写字母并且是一个顶级域名。

#### 3.3 类命名

类名必须是名词，并且每个单词的首字母必须大写，其余小写。

#### 3.4 变量命名

首字母小写，其他单词的首字母大写。

### 3.5 常量命名

- a) 常量所有字母大写，并且使用下划线分割单词。
- b) 尽量避免使用常量，对常量的访问可以替换为使用方法访问，比如：

```
int getMaxIterations() // NOT: MAX_ITERATIONS = 25{    return MAX_ITERATIONS;
}
```

### 3.6 方法命名

方法名必须是动词，并且以小写字母作为首字母。

### 3.7 私有变量

类私有变量以下划线结尾，比如：

```
class Person{
    private String name_; ...
}
```

### 3.8 特殊命名规范

- a) 必须使用get/set方法访问类变量。
- b) Boolean变量和方法使用is前缀，也可以使用has/can/should，但是注意前后需要一致。
- c) 提供计算功能的方法可以使用compute单词开头
- d) 提供查找功能的方法可以使用find单词开头
- e) 对象初始化的方法可以使用init单词开头
- f) 声明集合对象时使用复数形式。
- g) 表示对象个数的变量可以使用n前缀
- h) 遍历使用的变量叫i/j/k，并且j/k必须在嵌套的循环里使用，比如

```
for (Iterator i = points.iterator(); i.hasNext(); ) {
    :
}
for (int i = 0; i < nTables; i++) {
    :
}
```

对应的名称需要成对使用。比如get/set, add/remove, create/destroy, start/stop, insert/delete, increment/decrement, old/new, begin/end, first/last, up/down, min/max, next/previous, old/new, open/close, show/hide, suspend/resume等。

- i) 常量声明时必须以一种类型作为前缀，比如

```
final int COLOR_RED = 1;
```

- j) 异常类必须以Exception结尾。

## 4. 注释(Comments)

Java程序有两类注释：实现注释(implementation comments)和文档注释(document comments)。实现注释是那些在C++中见过的，使用//和/\*界

定的注释。文档注释(被称为“doc comments”)是Java独有的,并由/\*.../界定。文档注释可以通过javadoc工具转换成HTML文件。

实现注释用以注释代码或者实现细节。文档注释从实现自由(implementation-free)的角度描述代码的规范。它可以被那些手头没有源码的开发人员读懂。

注释应被用来给出代码的总括,并提供代码自身没有提供的附加信息。注释应该仅包含与阅读和理解程序有关的信息。例如,相应的包如何被建立或位于哪个目录下之类的信息不应包括在注释中。

在注释里,对设计决策中重要的或者不是显而易见的地方进行说明是可以的,但应避免提供代码中已清晰表达出来的重复信息。多余的注释很容易过时。通常应避免那些代码更新就可能过时的注释。

注意:频繁的注释有时反映出代码的低质量。当你觉得被迫要加注释的时候,考虑一下重写代码使其更清晰。

注释不应写在用星号或其他字符画出来的大框里。注释不应包括诸如制表符和回退符之类的特殊字符。

## 4.1 实现注释的格式(Implementation Comment Formats)

程序可以有4种实现注释的风格:块(block)、单行(single-line)、尾端(trailing)和行末(end-of-line)。

### 4.1.1 块注释(Block Comments)

块注释通常用于提供对文件,方法,数据结构和算法的描述。块注释被置于每个文件的开始处以及每个方法之前。它们也可以被用于其他地方,比如方法内部。在功能和方法内部的块注释应该和它们所描述的代码具有一样的缩进格式。

块注释之首应该有一个空行,用于把块注释和代码分割开来,比如:

```
/*
 * Here is a block comment.
 */
```

块注释可以以/\*-开头,这样indent(1)就可以将之识别为一个代码块的开始,而不会重排它。

```
/*-
 * Here is a block comment with some very special
 * formatting that I want indent(1) to ignore.
 *
 *     one
 *         two
 *             three
 */
```

注意:如果你不使用indent(1),就不必在代码中使用/\*-,或为他人可能对你的代码运行indent(1)作让步。

### 4.1.2 单行注释(Single-Line Comments)

短注释可以显示在一行内,并与其后的代码具有一样的缩进层级。如果一个注释不能在一行内写完,就该采用块注释(参见“块注释”)。单行注释之前应该有一个空行。以下是一个Java代码中单行注释的例子:

```
if (condition) {

    /* Handle the condition. */
    ...

}
```

### 4.1.3 尾端注释(Trailing Comments)

极短的注释可以与它们所要描述的代码位于同一行,但是应该有足够的空白来分开代码和注释。若有多个短注释出现于大段代码中,它们应该具有相同的缩进。



以下是一个Java代码中尾端注释的例子：

```
if (a == 2) {  
    return TRUE;           /* special case */  
}  
else {  
    return isPrime(a);     /* works only for odd a */  
}
```

#### 4.1.4 行末注释(End-Of-Line Comments)

注释界定符“//”，可以注释掉整行或者一行中的一部分。它一般不用于连续多行的注释文本；然而，它可以用来注释掉连续多行的代码段。以下是所有三种风格的例子：

```
if (foo > 1) {  
    // Do a double-flip.  
    ...  
}  
else {  
    return false;         // Explain why here.  
}  
  
//if (bar > 1) {  
//  
//    // Do a triple-flip.  
//    ...  
//}  
//else {  
//    return false;  
//}
```

#### 4.1.5 文档注释(Documentation Comments)

注意：此处描述的注释格式之范例，参见“Java源文件范例”

若想了解更多，参见“How to Write Doc Comments for Javadoc”，其中包含了有关文档注释标记的信息(@return, @param, @see)：

<http://java.sun.com/javadoc/writingdoccomments/index.html>

若想了解更多有关文档注释和javadoc的详细资料，参见javadoc的主页：

<http://java.sun.com/javadoc/index.html>

文档注释描述Java的类、接口、构造器，方法，以及字段(field)。每个文档注释都会被置于注释界定符/\*.../之中，一个注释对应一个类、接口或成员。该注释应位于声明之前：

```
/**  
 * The Example class provides ...  
 */  
public class Example { ...
```

注意顶层(top-level)的类和接口是不缩进的，而其成员是缩进的。描述类和接口的文档注释的第一行(/\*\*)不需缩进；随后的文档注释每行都缩进1格(使星号纵向对齐)。成员，包括构造函数在内，其文档注释的第一行缩进4格，随后每行都缩进5格。

若你想给出有关类、接口、变量或方法的信息，而这些信息又不适合写在文档中，则可使用实现块注释或紧跟在声明后面的单行注释。例如，有关一个类实现的细节，应放入紧跟在类声明后面的实现块注释中，而不是放在文档注释中。

文档注释不能放在一个方法或构造器的定义块中，因为Java会将位于文档注释之后的第一个声明与其相关联。

类注释使用如下格式：

```

/**
 *
 *
 * Copyright:
 * Company:
 * @author
 * @author
 * @version 1.00 9999/99/99
 *          9.99 9999/99/99
 *          9.99 9999/99/99
 * @see     1
 * @see     2
 */

```

例子:

```

/**
 * Title:111
 * Description: 222
 * Copyright: Copyright (c) 2010
 * Company: cmbc
 * @author 123
 * @version 1.00
 */

```

方法使用如下方式的注释:

```

/**
 *
 *
 * @param 1 1
 * @param 2 2
 * @param 3 3
 * @return
 * @throws .
 * @throws
 * @see 1
 * @see 2#
 */

```

例子

```

/**
 * Return lateral location of the specified position.
 * If the position is unset, NaN is returned.
 *
 * @param x X coordinate of position.
 * @param y Y coordinate of position.
 * @param zone Zone of position.
 * @return Lateral location.
 * @throws IllegalArgumentException If zone is <= 0.
 */
public double computeLocation(double x, double y, int zone)
throws IllegalArgumentException
{ ... }

```

注意:

- a) 开头在独立的一行以/\*\*开头
- b) 下面行中的\*要与第一个\*对齐
- c) 每个\*后面要有一个空格

- d) 在方法描述部分和参数部分要有一个空白行
- e) 每个参数的描述部分要对齐
- f) 每个参数描述部分以标点符号结尾

## 4.2 注释规范

- a) 尽量使用明了的逻辑结构以及适当的名称使代码能够自解释来减少注释的使用。
- b) 在注释分割符后添加一个空格。
- c) 如果是集合类型没有指定对象类型，那么要在声明的后面使用注释说明装载的对象类型，比如：

```
private Vector points_; // of Point
private Set shapes_;    // of Shape
```

- d) 任何时候使用集合类型，都要尽可能地在声明时说明装载对象的类型。
- e) 所有public类以及public类内的public和protected方法都要使用Javadoc进行注释。
- f) 所有非javadoc注释都要使用//的方式进行注释，包括多行注释。

## 5. 语句

### 5.1 package和import

在多数Java源文件中，第一个非注释行是包语句。在它之后可以跟import语句。Package和import语句之间以空行分开。Import语句以基本包开始，将相关的一类包归为一组，每组之间以空行隔开，且引入的类必须显示的指明（不能用import java.util.\*这样的）。例如：

```
package java.awt;

import java.io.IOException;
import java.net.URL;

import java.rmi.RmiServer;
import java.rmi.server.Server;

import javax.swing.JPanel;
import javax.swing.event.ActionEvent;

import org.apache.xmlrpc.server.SoapServer;
```

### 5.2 类和接口

类和接口的声明顺序如下：

1. 类/接口文档
2. class或者interface语句
3. 类（static）变量以public，protected，package（包可见），private的顺序声明
4. 实例变量以public，protected，package（包可见），private的顺序声明
5. 构造函数
6. 方法

### 5.3 方法

方法声明应该遵循以下顺序：

<access> static abstract synchronized <unusual> final native

Access指方法的可见性，有public/protected/package/private四种，必须放在最前面

<unusual>包括volatile和transient

## 5.4 变量

尽可能在变量声明的时候将变量初始化，并且保证变量的有效范围尽可能地小。

## 5.5 循环

a) 只有循环控制语句能够在for（）构造语句中出现，比如：

```
sum = 0; // NOT: for (i = 0, sum = 0; i < 100; i++) sum += value[i];
for (i = 0; i < 100; i++){
    sum += value[i];
}
```

b) 循环变量应该在循环前初始化。

c) 避免使用do-while

## 5.6 条件

避免使用复杂的条件表达式，可以使用临时变量替代，增加代码的可读性，例如：

```
bool isFinished = (elementNo < 0) || (elementNo > maxElement);
bool isRepeatedEntry = elementNo == lastElement;
if (isFinished || isRepeatedEntry){
    :
}
// NOT:
if ((elementNo < 0) || (elementNo > maxElement) || elementNo == lastElement) {
    :
}
```

不要条件中包含执行语句，例如：

```
InputStream stream = File.open(fileName, "w");
if (stream != null) {
    :
}
// NOT:
if (File.open(fileName, "w") != null) {
    :
}
```

## 5.7 switch

一个switch语句应该具有如下格式：

```

switch (condition) {
    case ABC:
        statements;
        // Fallthrough
    case DEF:
        statements;
        break;
    case XYZ:
        statements;
        break;
    default:
        statements;
        break;
}

```

每当一个case顺着往下执行时(因为没有break语句)，通常应在break语句的位置添加注释。上面的示例代码中就包含注释// Fallthrough。

## 5.7 其他

a) 为防止维护人员错误理解以及在维护时修改错误，在语句块中，无论语句有多少，都不能省略”{}”。例如：

```

if (null == obj) {
    return;
}

```

b) 属于一个逻辑单元内的语句应该与其他逻辑单元用空行分开。比如：

```

// Create a new identity matrix
Matrix4x4 matrix = new Matrix4x4();

// Precompute angles for efficiency
double cosAngle = Math.cos(angle);
double sinAngle = Math.sin(angle);

// Specify matrix as a rotation transformation
matrix.setElement(1, 1, cosAngle);
matrix.setElement(1, 2, sinAngle);
matrix.setElement(2, 1, -sinAngle);
matrix.setElement(2, 2, cosAngle);

// Apply rotation
transformation.multiply(matrix);

```

c) 避免使用魔法数字（magic number），常量数字应该在常量中声明，比如

```

private static final int TEAM_SIZE = 11;
Player[] players = new Player[TEAM_SIZE];    // NOT: Player[] players = new Player[11];

```

d) 浮点数字应该永远包含一个小数点和小数，比如：

```

double total = 0.0;    // NOT: double total = 0;
double speed = 3.0e8;  // NOT: double speed = 3e8;
double sum;
sum = (a + b) * 10.0;

```

e) 浮点数字的小数点前应该添加一个数字，比如：

```

double total = 0.5;    // NOT: double total = .5;

```

## 6. 其他

### 6.1 程序中出现的打印语句必须去掉

有时为了调试方便，进行输出某一些部分的值。而采用

```
System.out.println("");
```

这种打印语句的，在程序完成后必须去掉。

### 6.2 金额的数学运算规范

禁止使用float和double进行金额加减乘除运算，必须使用java.math.BigDecimal对金额进行运算。两个构造函数：

BigDecimal(double val)和BigDecimal(String val)

强烈要求使用后者，因为前者的结果存在一些不可预测的结果，仍然没有达到精确计算的精度。在我们EMU的交易中，输入金额往往为double类型，需要使用BigDecimal(String val)

来实现。进行加减乘除运算对应的方法为：add, subtract, multiply, divide.

如果需要对运算的结果按照约定的小数位数进行舍入，必须使用：

Java.math.BigDecimal.setScale(约定小数位数, RoundingMode.HALF\_UP)

## 7. 异常

### 7.1 异常定义

异常用于通知系统出现了不正常的情况，这种不正常的情况需要传递给相应的处理者。异常的目的是为了对异常的处理。只有在程序处理范围之外出现的不正常情况才属于异常，逻辑判断不属于异常范畴。

异常仅指从Exception和RuntimeException继承的类，程序通过异常类的不同来区分不同的异常。

使用Runtime异常，而不要使用Checked异常。这样只需要在方法注释中说明可能产生什么异常，但不需要在方法定义的时候声明，调用者也不需要强制捕获异常。

不要使用Error，Error用于JDK出现的错误，也不要实现Throwable接口。

如果可以通过返回null来说明不正常的情况，尽量采用此种方式，API中应有说明。

### 7.2 异常类定义

异常提供出的信息包括：异常的类型（类名）、异常信息、异常现场信息。

异常类型定义应恰如其分，异常信息应当准确而简约，异常现场信息是对异常信息的补充，用于描述异常发生的上下文信息，帮助通过日志进行排错。如：某个渠道某个交易拆包出错，那么“拆包出错”属于异常信息，而渠道和交易属于现场信息。

### 7.3 异常的声明

如果方法中会抛出异常，要在方法声明的注释中添加异常注释，RuntimeException不需要在方法声明中添加throws部分。

### 7.4 异常的处理

异常的处理通常会有如下几种情况：

- 记录日志。
- 对异常情况做恢复处理。
- 把异常包装为另一个异常抛出，参见异常转换。

d) 忽略，永远不要忽略异常，如有特殊情况，至少要说明忽略的原因。

## 7.5 异常转换

类的调用是有层次的，那么在高层方法调用底层方法时，底层方法抛出底层的异常，高层方法经常需要做异常的转换，即将底层的异常用高层的异常进行包装，必须将底层异常作为高层异常构造参数，然后抛出高层的异常。

捕获到底层异常的方法，可以选择处理并记录日志也可现在异常转换，如选择异常转换则不需要记录日志。

## 7.6 其他

尽可能地使用JDK中的已有异常类，如：`IllegalArgumentException`、`IllegalStateException`等。

任何异常都不应该改变对象调用方法之前的状态，如果有这种情况，则必须在API文档中说明。

## 8. 单元测试

除了`get/set`（有一定逻辑的除外）方法之外，其他`public`方法均须编写测试案例。

每次提交代码之前先进行单元测试，测试不通过的代码不得提交。

测试应当对数据库等资源不留或少留痕迹，例如，当测试添加一个用户时，在其成功后应及时从数据库中删除该记录，以避免脏数据的产生（由此衍生的一个经验是将添加、获取、删除一起测试）。