# Java编程风格指南

前言

这份文档是 Java编程风格规范的完整定义。当且仅当一个Java源文件符合此文档中的规则， 我们才认为它符合的Java编程风格。

与其它的编程风格指南一样，这里所讨论的不仅仅是编码格式美不美观的问题， 同时也讨论一些约定及编码标准。然而，这份文档主要侧重于我们所普遍遵循的规则， 对于那些不是明确强制要求的，我们尽量避免提供意见。

# 1.1 术语说明

在本文档中，除非另有说明：

术语class可表示一个普通类，枚举类，接口或是annotation类型(@interface)

术语comment只用来指代实现的注释(implementation comments)，我们不使用“documentation comments”一词，而是用Javadoc。

其他的术语说明会偶尔在后面的文档出现。

## 1.2 指南说明

本文档中的示例代码并不作为规范。也就是说，虽然示例代码是遵循编程风格，但并不意味着这是展现这些代码的唯一方式。 示例中的格式选择不应该被强制定为规则。

源文件基础

# 2.1 文件名

源文件以其最顶层的类名来命名，大小写敏感，文件扩展名为.java。

# 2.2 文件编码：UTF-8

源文件编码格式为UTF-8。

# 2.3 特殊字符

## 2.3.1 空白字符

除了行结束符序列，ASCII水平空格字符(0x20，即空格)是源文件中唯一允许出现的空白字符，这意味着：

所有其它字符串中的空白字符都要进行转义。

制表符Tab不用于缩进

Tab键大小是4个空格

## 2.3.2 特殊转义序列

对于具有特殊转义序列的任何字符(\b, \t, \n, \f, \r, \“, \‘及\)，我们使用它的转义序列，而不是相应的八进制(比如\012)或Unicode(比如\u000a)转义。

## 2.3.3 非ASCII字符

对于剩余的非ASCII字符，是使用实际的Unicode字符(比如∞)，还是使用等价的Unicode转义符(比如\u221e)，取决于哪个能让代码更易于阅读和理解。

Tip: 在使用Unicode转义符或是一些实际的Unicode字符时，建议做些注释给出解释，这有助于别人阅读和理解。

例如：

String unitAbbrev = "μs"; | 赞，即使没有注释也非常清晰

String unitAbbrev = "\u03bcs"; // "μs" | 允许，但没有理由要这样做

String unitAbbrev = "\u03bcs"; // Greek letter mu, "s" | 允许，但这样做显得笨拙还容易出错

String unitAbbrev = "\u03bcs"; | 很糟，读者根本看不出这是什么

return '\ufeff' + content; // byte order mark | Good，对于非打印字符，使用转义，并在必要时写上注释

Tip: 永远不要由于害怕某些程序可能无法正确处理非ASCII字符而让你的代码可读性变差。当程序无法正确处理非ASCII字符时，它自然无法正确运行， 你就会去fix这些问题的了。(言下之意就是大胆去用非ASCII字符，如果真的有需要的话)

源文件结构

一个源文件包含(按顺序地)：

许可证或版权信息(如有需要)

package语句

import语句

一个顶级类(只有一个)

以上每个部分之间用一个空行隔开。

# 3.1 许可证或版权信息

如果一个文件包含许可证或版权信息，那么它应当被放在文件最前面。

/\*

\* 项目: spi

\* 文件: CleanCodeDemo.java

\* 包名: clean.code

\* 日期: 2015年8月24日下午7:12:29

\* 版权：Copyright (c) 2015, qjdchina.com All Rights Reserved.

\*/

**package** clean.code;

# 3.2 package语句

package语句不换行，列限制(4.4节)并不适用于package语句。(即package语句写在一行里)

同一组的要放到一起

**package** com.qjdchina.account.biz.impl.test.integrate;

**import** **static** org.junit.Assert.*assertFalse*;

**import** java.io.IOException;

**import** java.math.BigDecimal;

**import** java.net.InetAddress;

**import** java.net.UnknownHostException;

**import** java.sql.ResultSet;

**import** java.sql.SQLException;

**import** java.text.SimpleDateFormat;

**import** java.util.Calendar;

**import** java.util.Date;

**import** java.util.HashMap;

**import** java.util.List;

**import** java.util.Map;

**import** java.util.concurrent.atomic.AtomicInteger;

**import** org.junit.Assert;

**import** org.junit.Before;

**import** org.junit.FixMethodOrder;

**import** org.junit.Test;

**import** org.junit.runner.RunWith;

**import** org.junit.runners.MethodSorters;

**import** org.slf4j.Logger;

**import** org.slf4j.LoggerFactory;

**import** org.springframework.batch.core.BatchStatus;

**import** org.springframework.batch.core.Job;

**import** org.springframework.batch.core.JobExecution;

**import** org.springframework.batch.core.JobParameter;

**import** org.springframework.batch.core.JobParameters;

**import** org.springframework.batch.core.launch.JobLauncher;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.beans.factory.annotation.Qualifier;

**import** org.springframework.jdbc.core.JdbcTemplate;

**import** org.springframework.jdbc.core.RowCallbackHandler;

**import** org.springframework.test.context.ContextConfiguration;

**import** org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

**import** com.qjdchina.account.biz.AccountManage;

**import** com.qjdchina.account.biz.LoanInfoManagement;

**import** com.qjdchina.account.biz.NewLoanManagement;

**import** com.qjdchina.account.biz.RepayManagement;

**import** com.qjdchina.account.biz.RepaymentPlanManagement;

**import** com.qjdchina.account.biz.batchs.services.InterestAccrual;

**import** com.qjdchina.account.biz.cover.LoanInfoCover;

**import** com.qjdchina.account.biz.meta.BizDateMetaManagement;

**import** com.qjdchina.account.biz.meta.BizDateRedisCache;

**import** com.qjdchina.account.biz.meta.CreditAccountMetaManager;

**import** com.qjdchina.account.biz.meta.CreditRunningAccountMetaManager;

**import** com.qjdchina.account.biz.meta.DebitAccountMetaManager;

**import** com.qjdchina.account.biz.meta.DebitRunningAccountMetaManager;

**import** com.qjdchina.account.biz.meta.RepaymentPlanMetaManager;

**import** com.qjdchina.account.biz.test.tools.DBUtils;

**import** com.qjdchina.account.biz.test.tools.DateUtil;

**import** com.qjdchina.account.biz.test.tools.XlsDataSetBeanFactory;

**import** com.qjdchina.account.dao.mapper.BizDateMapper;

**import** com.qjdchina.account.model.BizDate;

**import** com.qjdchina.account.model.LoanInfo;

**import** com.qjdchina.account.service.model.AccountVO;

**import** com.qjdchina.account.service.model.LoanInfoVO;

**import** com.qjdchina.account.service.model.RepayDetailVO;

**import** com.qjdchina.account.service.model.mode.AccountMode;

**import** com.qjdchina.account.service.status.AccountStatus;

**import** com.qjdchina.account.service.status.RepaymentPlanStatus;

**import** com.qjdchina.commons.result.Result;

# 3.3 import语句

## 3.3.1 import不要使用通配符

即，不要出现类似这样的import语句：import java.util.\*;

## 3.3.2 不要换行

import语句不换行，列限制(4.4节)并不适用于import语句。(每个import语句独立成行)

## 3.3.3 顺序和间距

import语句可分为以下几组，按照这个顺序，每组由一个空行分隔：

所有的静态导入独立成组

com. imports(仅当这个源文件是在com.包下)

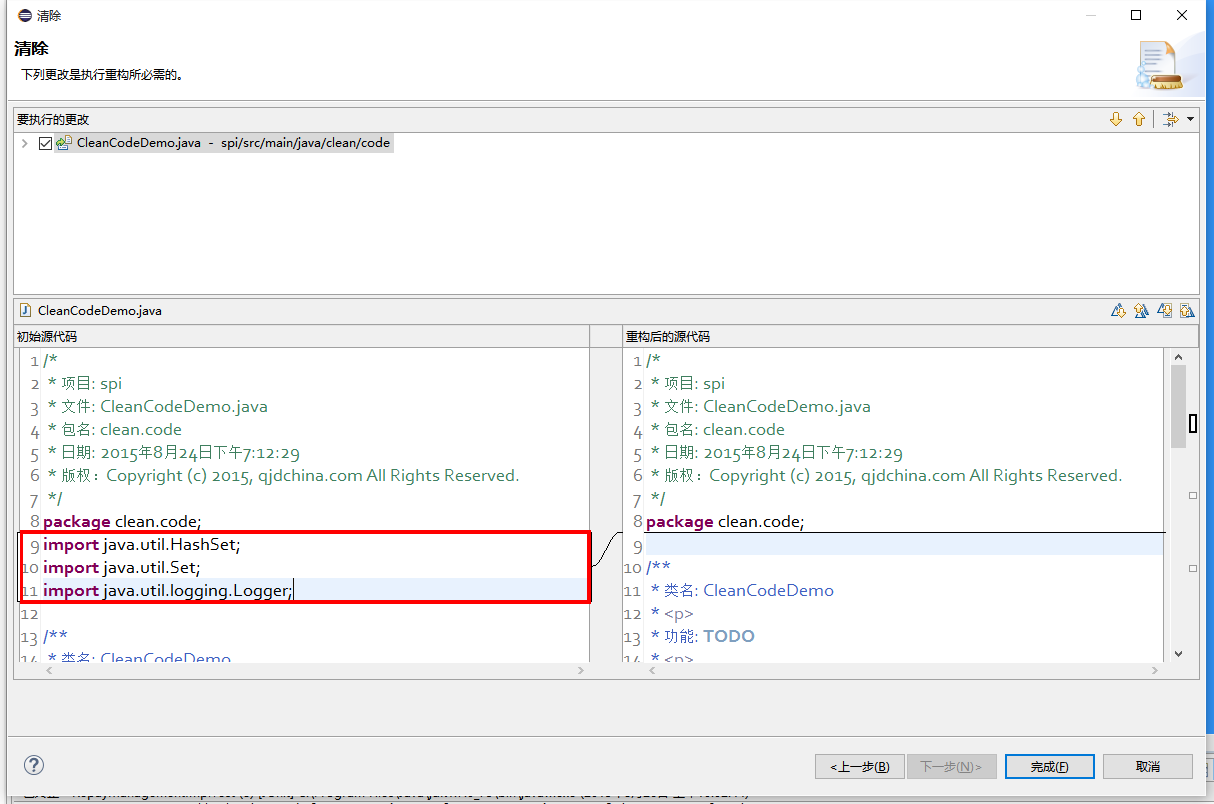
第三方的包。每个顶级包为一组，字典序。例如：android, com, junit, org, sun

java imports

javax imports

组内不空行，按字典序排列。

## 3.3.4 清除掉没用到的import



# 3.4 类声明

## 3.4.1 只有一个顶级类声明

每个顶级类都在一个与它同名的源文件中(当然，还包含.java后缀)。

例外：package-info.java，该文件中可没有package-info类。

## 3.4.2 类成员顺序

类的成员顺序对易学性有很大的影响，但这也不存在唯一的通用法则。不同的类对成员的排序可能是不同的。 最重要的一点，每个类应该以某种逻辑去排序它的成员，维护者应该要能解释这种排序逻辑。比如， 新的方法不能总是习惯性地添加到类的结尾，因为这样就是按时间顺序而非某种逻辑来排序的。

### 3.4.2.1 重载：永不分离

当一个类有多个构造函数，或是多个同名方法，这些函数/方法应该按顺序出现在一起，中间不要放进其它函数/方法。

格式

术语说明：块状结构(block-like construct)指的是一个类，方法或构造函数的主体。需要注意的是，数组初始化中的初始值可被选择性地视为块状结构(4.8.3.1节)。

# 4.1 大括号

## 4.1.1 使用大括号(即使是可选的)

大括号与if, else, for, do, while语句一起使用，即使只有一条语句(或是空)，也应该把大括号写上。

## 4.1.2 非空块：K & R 风格

对于非空块和块状结构，大括号遵循Kernighan和Ritchie风格 (Egyptian brackets):

左大括号前不换行

左大括号后换行

右大括号前换行

如果右大括号是一个语句、函数体或类的终止，则右大括号后换行; 否则不换行。例如，如果右大括号后面是else或逗号，则不换行。

示例：

return new MyClass() {

@Override public void method() {

if (condition()) {

try {

something();

} catch (ProblemException e) {

recover();

}

}

}

};

4.8.1节给出了enum类的一些例外。

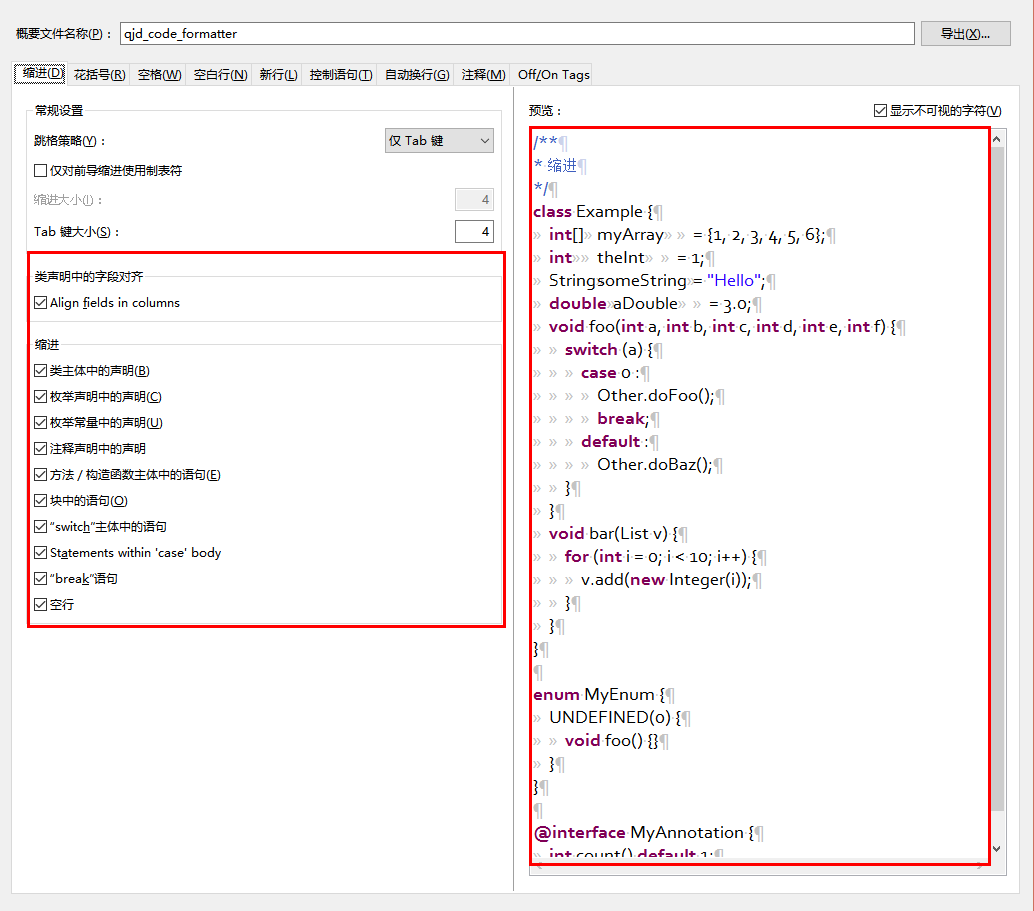
## 4.1.3 空块：可以用简洁版本

一个空的块状结构里什么也不包含，大括号可以简洁地写成{}，不需要换行。例外：如果它是一个多块语句的一部分(if/else 或 try/catch/finally) ，即使大括号内没内容，右大括号也要换行。

示例：

void doNothing() {}

# 4.2 块缩进：2个空格



每当开始一个新的块，缩进增加2个空格，当块结束时，缩进返回先前的缩进级别。缩进级别适用于代码和注释。代码示例:

/\*\*

\* 缩进

\*/

**class** Example {

**int**[] myArray = {1, 2, 3, 4, 5, 6};

**int** theInt = 1;

String someString = "Hello";

**double** aDouble = 3.0;

**void** foo(**int** a, **int** b, **int** c, **int** d, **int** e, **int** f) {

**switch** (a) {

**case** 0 :

Other.doFoo();

**break**;

**default** :

Other.doBaz();

}

}

**void** bar(List v) {

**for** (**int** i = 0; i < 10; i++) {

v.add(**new** Integer(i));

}

}

}

**enum** MyEnum {

UNDEFINED(0) {

**void** foo() {}

}

}

**@interface** MyAnnotation {

**int** count() **default** 1;

}

# 4.3 一行一个语句

每个语句后要换行。

# 4.4 列限制：80或100

一个项目可以选择一行80个字符或100个字符的列限制，除了下述例外，任何一行如果超过这个字符数限制，必须自动换行。

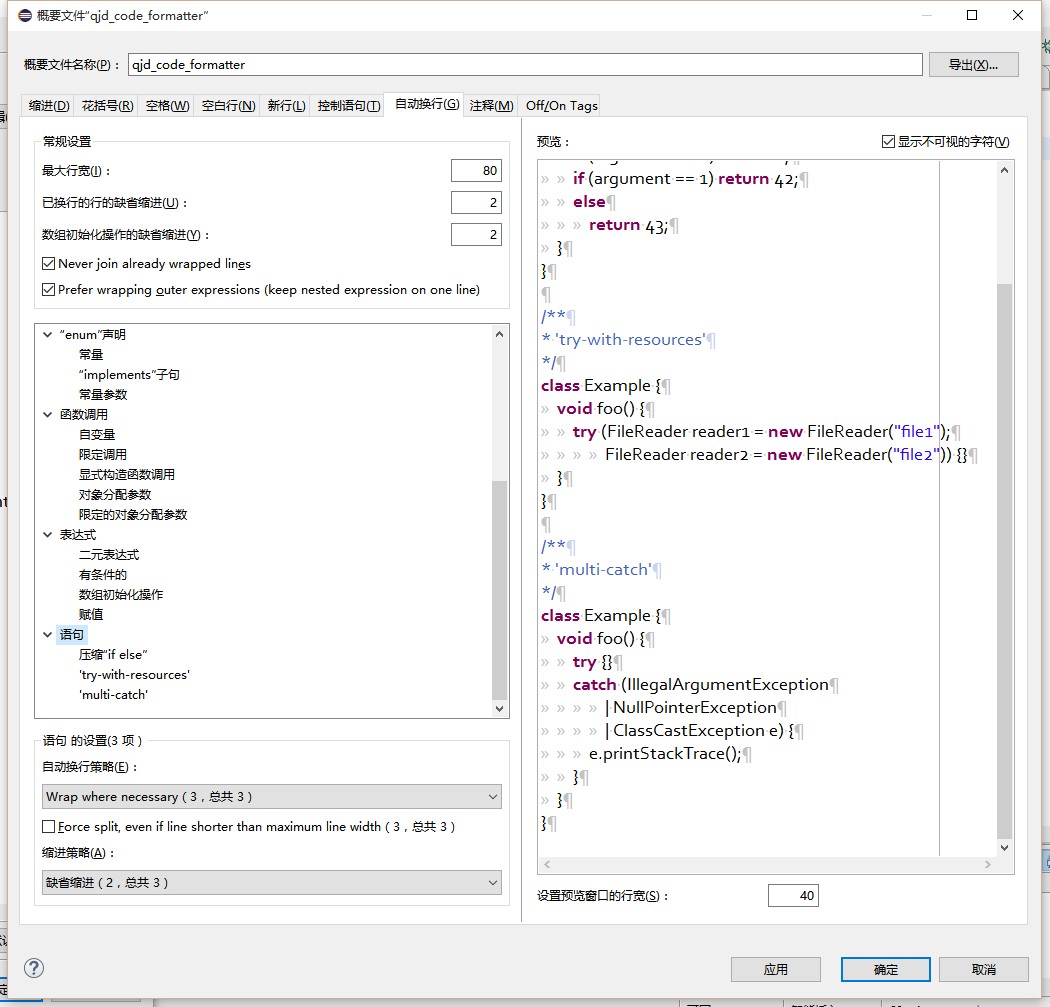
例外：

不可能满足列限制的行(例如，Javadoc中的一个长URL，或是一个长的JSNI方法参考)。

package和import语句。

# 4.5 自动换行

术语说明：一般情况下，一行长代码为了避免超出列限制(80或100个字符)而被分为多行，我们称之为自动换行(line-wrapping)。



我们并没有全面，确定性的准则来决定在每一种情况下如何自动换行。很多时候，对于同一段代码会有好几种有效的自动换行方式。

Tip: 提取方法或局部变量可以在不换行的情况下解决代码过长的问题(是合理缩短命名长度吧)

## 4.5.1 从哪里断开

自动换行的基本准则是：更倾向于在更高的语法级别处断开。

如果在非赋值运算符处断开，那么在该符号前断开(比如+，它将位于下一行)。注意：这一点与其它语言的编程风格不同(如C++和JavaScript)。 这条规则也适用于以下“类运算符”符号：点分隔符(.)，类型界限中的&（<T extends Foo & Bar>)，catch块中的管道符号(catch (FooException | BarException e)

如果在赋值运算符处断开，通常的做法是在该符号后断开(比如=，它与前面的内容留在同一行)。这条规则也适用于foreach语句中的分号。

方法名或构造函数名与左括号留在同一行。

逗号(,)与其前面的内容留在同一行。

## 4.5.2 自动换行时缩进至少+4个空格

自动换行时，第一行后的每一行至少比第一行多缩进4个空格(注意：制表符不用于缩进。见2.3.1节)。

当存在连续自动换行时，缩进可能会多缩进不只4个空格(语法元素存在多级时)。一般而言，两个连续行使用相同的缩进当且仅当它们开始于同级语法元素。

不鼓励使用可变数目的空格来对齐前面行的符号。

# 4.6 空白

## 4.6.1 垂直空白

以下情况需要使用一个空行：

类内连续的成员之间：字段，构造函数，方法，嵌套类，静态初始化块，实例初始化块。

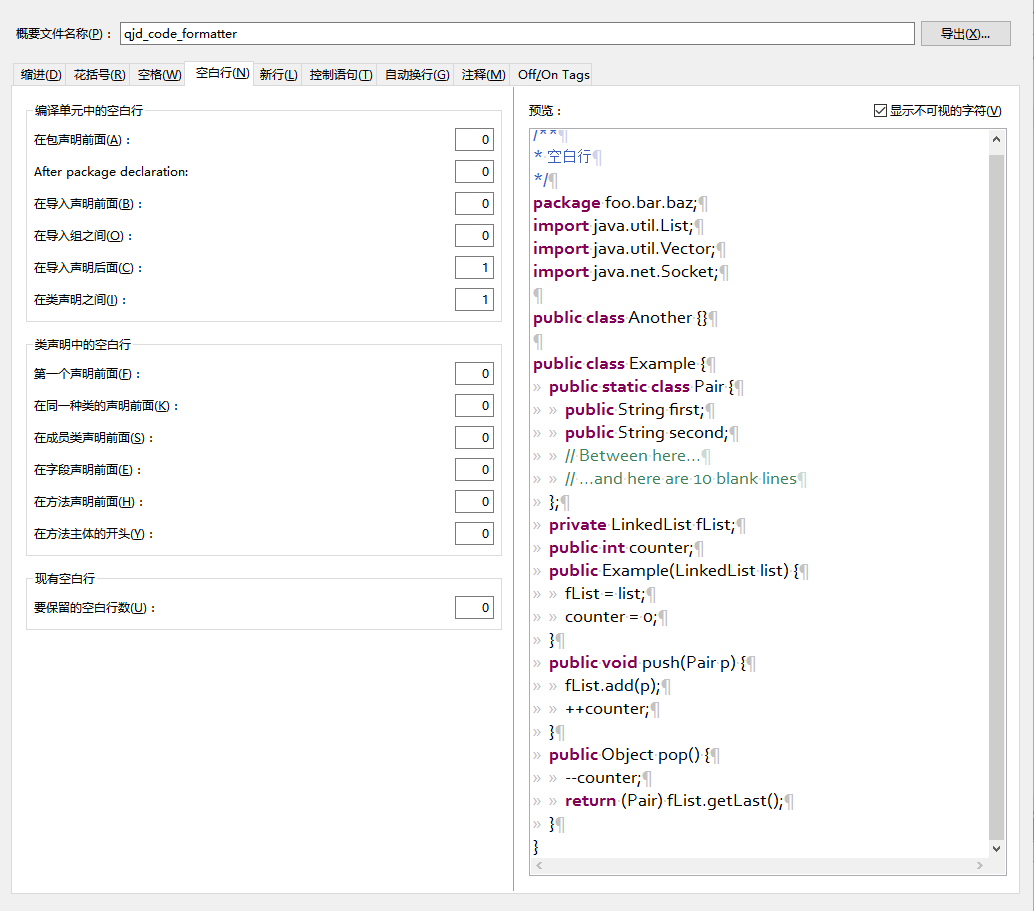
例外：两个连续字段之间的空行是可选的，用于字段的空行主要用来对字段进行逻辑分组。

在函数体内，语句的逻辑分组间使用空行。

类内的第一个成员前或最后一个成员后的空行是可选的(既不鼓励也不反对这样做，视个人喜好而定)。

要满足本文档中其他节的空行要求.

多个连续的空行是允许的，但没有必要这样做(我们也不鼓励这样做)。



代码示例:

/\*\*

\* 空白行

\*/

**package** foo.bar.baz;

**import** java.util.List;

**import** java.util.Vector;

**import** java.net.Socket;

**public** **class** Another {}

**public** **class** Example {

**public** **static** **class** Pair {

**public** String first;

**public** String second;

// Between here...

// ...and here are 10 blank lines

};

**private** LinkedList fList;

**public** **int** counter;

**public** Example(LinkedList list) {

fList = list;

counter = 0;

}

**public** **void** push(Pair p) {

fList.add(p);

++counter;

}

**public** Object pop() {

--counter;

**return** (Pair) fList.getLast();

}

}

## 4.6.2 水平空白

除了语言需求和其它规则，并且除了文字，注释和Javadoc用到单个空格.

## 4.6.3 水平对齐：不做要求

术语说明：水平对齐指的是通过增加可变数量的空格来使某一行的字符与上一行的相应字符对齐。

这是允许的(而且在不少地方可以看到这样的代码)，但编程风格对此不做要求。即使对于已经使用水平对齐的代码，我们也不需要去保持这种风格。

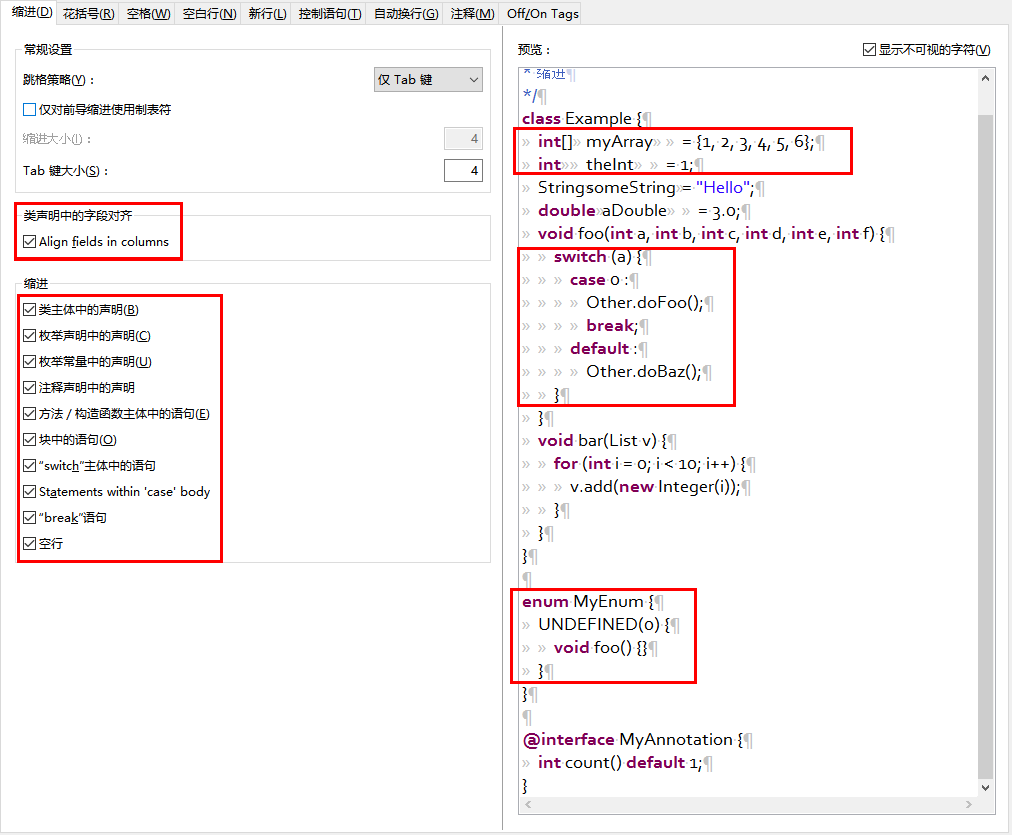
以下示例先展示未对齐的代码，然后是对齐的代码：

private int x; // this is fine

private Color color; // this too

private int x; // permitted, but future edits

private Color color; // may leave it unaligned



Tip：对齐可增加代码可读性，但它为日后的维护带来问题。考虑未来某个时候，我们需要修改一堆对齐的代码中的一行。 这可能导致原本很漂亮的对齐代码变得错位。很可能它会提示你调整周围代码的空白来使这一堆代码重新水平对齐(比如程序员想保持这种水平对齐的风格)， 这就会让你做许多的无用功，增加了reviewer的工作并且可能导致更多的合并冲突。

代码示例:

/\*\*

\* 缩进

\*/

**class** Example {

**int**[] myArray = {1, 2, 3, 4, 5, 6};

**int** theInt = 1;

String someString = "Hello";

**double** aDouble = 3.0;

**void** foo(**int** a, **int** b, **int** c, **int** d, **int** e, **int** f) {

**switch** (a) {

**case** 0 :

Other.doFoo();

**break**;

**default** :

Other.doBaz();

}

}

**void** bar(List v) {

**for** (**int** i = 0; i < 10; i++) {

v.add(**new** Integer(i));

}

}

}

**enum** MyEnum {

UNDEFINED(0) {

**void** foo() {}

}

}

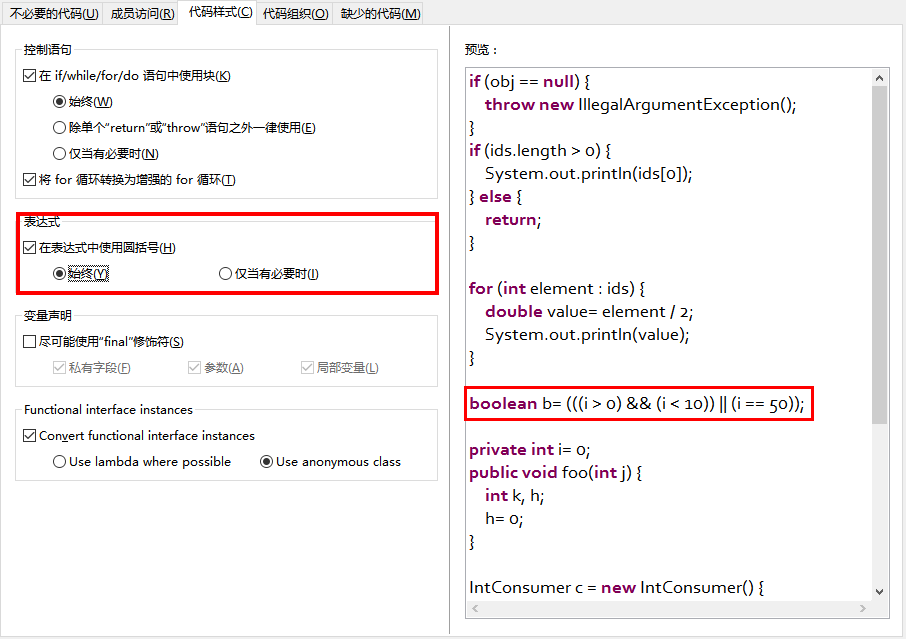
**@interface** MyAnnotation {

**int** count() **default** 1;

}

# 4.7 用小括号来限定组

除非作者和reviewer都认为去掉小括号也不会使代码被误解，或是去掉小括号能让代码更易于阅读，否则我们不应该去掉小括号。 我们没有理由假设读者能记住整个Java运算符优先级表。



**if** (obj == **null**) {

**throw** **new** IllegalArgumentException();

}

**if** (ids.length > 0) {

System.out.println(ids[0]);

} **else** {

**return**;

}

**for** (**int** element : ids) {

**double** value= element / 2;

System.out.println(value);

}

**boolean** b= (((i > 0) && (i < 10)) || (i == 50));

**private** **int** i= 0;

**public** **void** foo(**int** j) {

**int** k, h;

h= 0;

}

IntConsumer c = **new** IntConsumer() {

@Override **public** **void** accept(**int** value) {

System.out.println(i);

}

};

Runnable r = **new** Runnable() {

@Override **public** **void** run() {

//do something

}

};

# 4.8 具体结构

## 4.8.1 枚举类

枚举常量间用逗号隔开，换行可选。

没有方法和文档的枚举类可写成数组初始化的格式：

private enum Suit { CLUBS, HEARTS, SPADES, DIAMONDS }

由于枚举类也是一个类，因此所有适用于其它类的格式规则也适用于枚举类。

## 4.8.2 变量声明

### 4.8.2.1 每次只声明一个变量

不要使用组合声明，比如int a, b;。

### 4.8.2.2 需要时才声明，并尽快进行初始化

不要在一个代码块的开头把局部变量一次性都声明了(这是c语言的做法)，而是在第一次需要使用它时才声明。 局部变量在声明时最好就进行初始化，或者声明后尽快进行初始化。

## 4.8.3 数组

### 4.8.3.1 数组初始化：可写成块状结构

数组初始化可以写成块状结构，比如，下面的写法都是OK的：

new int[] {

0, 1, 2, 3

}

new int[] {

0,

1,

2,

3

}

new int[] {

0, 1,

2, 3

}

new int[]

{0, 1, 2, 3}

### 4.8.3.2 非C风格的数组声明

中括号是类型的一部分：String[] args， 而非String args[]。

## 4.8.4 switch语句

术语说明：switch块的大括号内是一个或多个语句组。每个语句组包含一个或多个switch标签(case FOO:或default:)，后面跟着一条或多条语句。

### 4.8.4.1 缩进

与其它块状结构一致，switch块中的内容缩进为2个空格。

每个switch标签后新起一行，再缩进2个空格，写下一条或多条语句。

### 4.8.4.2 Fall-through：注释

在一个switch块内，每个语句组要么通过break, continue, return或抛出异常来终止，要么通过一条注释来说明程序将继续执行到下一个语句组， 任何能表达这个意思的注释都是OK的(典型的是用// fall through)。这个特殊的注释并不需要在最后一个语句组(一般是default)中出现。示例：

switch (input) {

case 1:

case 2:

prepareOneOrTwo();

// fall through

case 3:

handleOneTwoOrThree();

break;

default:

handleLargeNumber(input);

}

### 4.8.4.3 default的情况要写出来

每个switch语句都包含一个default语句组，即使它什么代码也不包含。

## 4.8.5 注解(Annotations)

注解紧跟在文档块后面，应用于类、方法和构造函数，一个注解独占一行。这些换行不属于自动换行(第4.5节，自动换行)，因此缩进级别不变。例如：

@Override

@Nullable

public String getNameIfPresent() { ... }

例外：单个的注解可以和签名的第一行出现在同一行。例如：

@Override public int hashCode() { ... }

应用于字段的注解紧随文档块出现，应用于字段的多个注解允许与字段出现在同一行。例如：

@Partial @Mock DataLoader loader;

参数和局部变量注解没有特定规则。

## 4.8.6 注释

### 4.8.6.1 块注释风格

块注释与其周围的代码在同一缩进级别。它们可以是/\* ... \*/风格，也可以是// ...风格。对于多行的/\* ... \*/注释，后续行必须从\*开始， 并且与前一行的\*对齐。以下示例注释都是OK的。

/\*

\* This is // And so /\* Or you can

\* okay. // is this. \* even do this. \*/

\*/

注释不要封闭在由星号或其它字符绘制的框架里。

Tip：在写多行注释时，如果你希望在必要时能重新换行(即注释像段落风格一样)，那么使用/\* ... \*/。

## 4.8.7 Modifiers

类和成员的modifier顺序按Java语言规范中推荐的顺序出现。

@Annotations public protected private abstract static final transient volatile synchronized native strictfp

strictfp, 即 strict float point (精确浮点),

**命名约定**

# 5.1 对所有标识符都通用的规则

标识符只能使用ASCII字母和数字，因此每个有效的标识符名称都能匹配正则表达式\w+。

在其它编程语言风格中使用的特殊前缀或后缀，如name\_, mName, s\_name和kName，在Java编程风格中都不再使用。

# 5.2 标识符类型的规则

## 5.2.1 包名

包名全部小写，连续的单词只是简单地连接起来，不使用下划线。

## 5.2.2 类名

类名都以UpperCamelCase风格编写。

类名通常是名词或名词短语，接口名称有时可能是形容词或形容词短语。现在还没有特定的规则或行之有效的约定来命名注解类型。

测试类的命名以它要测试的类的名称开始，以Test结束。例如，HashTest或HashIntegrationTest。

## 5.2.3 方法名

方法名都以lowerCamelCase风格编写。

方法名通常是动词或动词短语。

下划线可能出现在JUnit测试方法名称中用以分隔名称的逻辑组件。一个典型的模式是：test<MethodUnderTest>\_<state>，例如testPop\_emptyStack。 并不存在唯一正确的方式来命名测试方法。

## 5.2.4 常量名

常量名命名模式为CONSTANT\_CASE，全部字母大写，用下划线分隔单词。

每个常量都是一个静态final字段，但不是所有静态final字段都是常量。在决定一个字段是否是一个常量时， 考虑它是否真的感觉像是一个常量。例如，如果任何一个该实例的观测状态是可变的，则它几乎肯定不会是一个常量。 只是永远不打算改变对象一般是不够的，它要真的一直不变才能将它示为常量。

// Constants

**static** **final** **int** NUMBER = 5;

**static** **final** ImmutableList<String> NAMES = ImmutableList.of("Ed", "Ann");

**static** **final** Joiner COMMA\_JOINER = Joiner.on(','); // because Joiner is immutable

**static** **final** SomeMutableType[] EMPTY\_ARRAY = {};

**enum** SomeEnum { ENUM\_CONSTANT }

// Not constants

**static** String nonFinal = "non-final";

**final** String nonStatic = "non-static";

**static** **final** Set<String> mutableCollection = **new** HashSet<String>();

**static** **final** ImmutableSet<SomeMutableType> mutableElements = ImmutableSet.of(mutable);

**static** **final** Logger logger = Logger.*getLogger*(MyClass.getName());

**static** **final** String[] nonEmptyArray = {"these", "can", "change"};

这些名字通常是名词或名词短语。

## 5.2.5 非常量字段名

非常量字段名以lowerCamelCase风格编写。

这些名字通常是名词或名词短语。

## 5.2.6 参数名

参数名以lowerCamelCase风格编写。

参数应该避免用单个字符命名。

## 5.2.7 局部变量名

局部变量名以lowerCamelCase风格编写，比起其它类型的名称，局部变量名可以有更为宽松的缩写。

虽然缩写更宽松，但还是要避免用单字符进行命名，除了临时变量和循环变量。

即使局部变量是final和不可改变的，也不应该把它示为常量，自然也不能用常量的规则去命名它。

## 5.2.8 类型变量名

类型变量可用以下两种风格之一进行命名：

单个的大写字母，后面可以跟一个数字(如：E, T, X, T2)。

以类命名方式(5.2.2节)，后面加个大写的T(如：RequestT, FooBarT)。

# 5.3 驼峰式命名法(CamelCase)

驼峰式命名法分大驼峰式命名法(UpperCamelCase)和小驼峰式命名法(lowerCamelCase)。 有时，我们有不只一种合理的方式将一个英语词组转换成驼峰形式，如缩略语或不寻常的结构(例如"IPv6"或"iOS")。指定了以下的转换方案。

名字从散文形式(prose form)开始:

把短语转换为纯ASCII码，并且移除任何单引号。例如："Müller’s algorithm"将变成"Muellers algorithm"。

把这个结果切分成单词，在空格或其它标点符号(通常是连字符)处分割开。

推荐：如果某个单词已经有了常用的驼峰表示形式，按它的组成将它分割开(如"AdWords"将分割成"ad words")。 需要注意的是"iOS"并不是一个真正的驼峰表示形式，因此该推荐对它并不适用。

现在将所有字母都小写(包括缩写)，然后将单词的第一个字母大写：

(1) 每个单词的第一个字母都大写，来得到大驼峰式命名。

(2) 除了第一个单词，每个单词的第一个字母都大写，来得到小驼峰式命名。

最后将所有的单词连接起来得到一个标识符。

示例：

Prose form Correct Incorrect

------------------------------------------------------------------

"XML HTTP request" XmlHttpRequest XMLHTTPRequest

"new customer ID" newCustomerId newCustomerID

"inner stopwatch" innerStopwatch innerStopWatch

"supports IPv6 on iOS" supportsIpv6OnIos supportsIPv6OnIOS

"YouTube importer" YouTubeImporter YoutubeImporter

Note：在英语中，某些带有连字符的单词形式不唯一。例如："nonempty"和"non-empty"都是正确的，因此方法名checkNonempty和checkNonEmpty也都是正确的。

**编程实践**

# 6.1 @Override：能用则用

只要是合法的，就把@Override注解给用上。

# 6.2 捕获的异常：不能忽视

除了下面的例子，对捕获的异常不做响应是极少正确的。(典型的响应方式是打印日志，或者如果它被认为是不可能的，则把它当作一个AssertionError重新抛出。)

如果它确实是不需要在catch块中做任何响应，需要做注释加以说明(如下面的例子)。

try {

int i = Integer.parseInt(response);

return handleNumericResponse(i);

} catch (NumberFormatException ok) {

// it's not numeric; that's fine, just continue

}

return handleTextResponse(response);

例外：在测试中，如果一个捕获的异常被命名为expected，则它可以被不加注释地忽略。下面是一种非常常见的情形，用以确保所测试的方法会抛出一个期望中的异常， 因此在这里就没有必要加注释。

try {

emptyStack.pop();

fail();

} catch (NoSuchElementException expected) {

}

# 6.3 静态成员：使用类进行调用

使用类名调用静态的类成员，而不是具体某个对象或表达式。

Foo aFoo = ...;

Foo.aStaticMethod(); // good

aFoo.aStaticMethod(); // bad

somethingThatYieldsAFoo().aStaticMethod(); // very bad

# 6.4 Finalizers: 禁用

极少会去重载Object.finalize。

Tip：不要使用finalize。如果你非要使用它，请先仔细阅读和理解Effective Java 第7条款：“Avoid Finalizers”，然后不要使用它。

# 7.1 Javadoc格式

## 7.1.1 一般形式

Javadoc块的基本格式如下所示：

/\*\*

\* Multiple lines of Javadoc text are written here,

\* wrapped normally...

\*/

public int method(String p1) { ... }

或者是以下单行形式：

/\*\* An especially short bit of Javadoc. \*/

基本格式总是OK的。当整个Javadoc块能容纳于一行时(且没有Javadoc标记@XXX)，可以使用单行形式。

## 7.1.2 段落

空行(即，只包含最左侧星号的行)会出现在段落之间和Javadoc标记(@XXX)之前(如果有的话)。 除了第一个段落，每个段落第一个单词前都有标签<p>，并且它和第一个单词间没有空格。

## 7.1.3 Javadoc标记

标准的Javadoc标记按以下顺序出现：@param, @return, @throws, @deprecated, 前面这4种标记如果出现，描述都不能为空。 当描述无法在一行中容纳，连续行需要至少再缩进4个空格。

# 7.2 摘要片段

每个类或成员的Javadoc以一个简短的摘要片段开始。这个片段是非常重要的，在某些情况下，它是唯一出现的文本，比如在类和方法索引中。

这只是一个小片段，可以是一个名词短语或动词短语，但不是一个完整的句子。它不会以A {@code Foo} is a...或This method returns...开头, 它也不会是一个完整的祈使句，如Save the record...。然而，由于开头大写及被加了标点，它看起来就像是个完整的句子。

Tip：一个常见的错误是把简单的Javadoc写成/\*\* @return the customer ID \*/，这是不正确的。它应该写成/\*\* Returns the customer ID. \*/。

# 7.3 哪里需要使用Javadoc

至少在每个public类及它的每个public和protected成员处使用Javadoc，以下是一些例外：

## 7.3.1 例外：不言自明的方法

对于简单明显的方法如getFoo，Javadoc是可选的(即，是可以不写的)。这种情况下除了写“Returns the foo”，确实也没有什么值得写了。

单元测试类中的测试方法可能是不言自明的最常见例子了，我们通常可以从这些方法的描述性命名中知道它是干什么的，因此不需要额外的文档说明。

Tip：如果有一些相关信息是需要读者了解的，那么以上的例外不应作为忽视这些信息的理由。例如，对于方法名getCanonicalName， 就不应该忽视文档说明，因为读者很可能不知道词语canonical name指的是什么。

## 7.3.2 例外：重载

如果一个方法重载了超类中的方法，那么Javadoc并非必需的。

## 7.3.3 可选的Javadoc

对于包外不可见的类和方法，如有需要，也是要使用Javadoc的。如果一个注释是用来定义一个类，方法，字段的整体目的或行为， 那么这个注释应该写成Javadoc，这样更统一更友好。

# 附:英文原文

# Google Java Style

Last changed: March 21, 2014

|  |  |  |
| --- | --- | --- |
| [1 Introduction](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s1-introduction)  [1.1 Terminology notes](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s1.1-terminology)  [1.2 Guide notes](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s1.2-guide-notes)  [2 Source file basics](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s2-source-file-basics)  [2.1 File name](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s2.1-file-name)  [2.2 File encoding: UTF-8](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s2.2-file-encoding)  [2.3 Special characters](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s2.3-special-characters)  [2.3.1 Whitespace characters](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s2.3.1-whitespace-characters)  [2.3.2 Special escape sequences](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s2.3.2-special-escape-sequences)  [2.3.3 Non-ASCII characters](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s2.3.3-non-ascii-characters)  [3 Source file structure](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s3-source-file-structure)  [3.1 License or copyright information, if present](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s3.1-copyright-statement)  [3.2 Package statement](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s3.2-package-statement)  [3.3 Import statements](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s3.3-import-statements)  [3.3.1 No wildcard imports](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s3.3.1-wildcard-imports)  [3.3.2 No line-wrapping](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s3.3.2-import-line-wrapping)  [3.3.3 Ordering and spacing](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s3.3.3-import-ordering-and-spacing)  [3.4 Class declaration](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s3.4-class-declaration)  [3.4.1 Exactly one top-level class declaration](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s3.4.1-one-top-level-class)  [3.4.2 Class member ordering](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s3.4.2-class-member-ordering) | [4 Formatting](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s4-formatting)  [4.1 Braces](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s4.1-braces)  [4.1.1 Braces are used where optional](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s4.1.1-braces-always-used)  [4.1.2 Nonempty blocks: K & R style](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s4.1.2-blocks-k-r-style)  [4.1.3 Empty blocks: may be concise](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s4.1.3-braces-empty-blocks)  [4.2 Block indentation: +2 spaces](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s4.2-block-indentation)  [4.3 One statement per line](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s4.3-one-statement-per-line)  [4.4 Column limit: 80 or 100](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s4.4-column-limit)  [4.5 Line-wrapping](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s4.5-line-wrapping)  [4.5.1 Where to break](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s4.5.1-line-wrapping-where-to-break)  [4.5.2 Indent continuation lines at least +4 spaces](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s4.5.2-line-wrapping-indent)  [4.6 Whitespace](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s4.6-whitespace)  [4.6.1 Vertical Whitespace](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s4.6.1-vertical-whitespace)  [4.6.2 Horizontal whitespace](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s4.6.2-horizontal-whitespace)  [4.6.3 Horizontal alignment: never required](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s4.6.3-horizontal-alignment)  [4.7 Grouping parentheses: recommended](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s4.7-grouping-parentheses)  [4.8 Specific constructs](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s4.8-specific-constructs)  [4.8.1 Enum classes](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s4.8.1-enum-classes)  [4.8.2 Variable declarations](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s4.8.2-variable-declarations)  [4.8.3 Arrays](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s4.8.3-arrays)  [4.8.4 Switch statements](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s4.8.4-switch)  [4.8.5 Annotations](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s4.8.5-annotations)  [4.8.6 Comments](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s4.8.6-comments)  [4.8.7 Modifiers](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s4.8.7-modifiers)  [4.8.8 Numeric Literals](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s4.8.8-numeric-literals) | [5 Naming](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s5-naming)  [5.1 Rules common to all identifiers](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s5.1-identifier-names)  [5.2 Rules by identifier type](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s5.2-specific-identifier-names)  [5.2.1 Package names](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s5.2.1-package-names)  [5.2.2 Class names](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s5.2.2-class-names)  [5.2.3 Method names](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s5.2.3-method-names)  [5.2.4 Constant names](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s5.2.4-constant-names)  [5.2.5 Non-constant field names](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s5.2.5-non-constant-field-names)  [5.2.6 Parameter names](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s5.2.6-parameter-names)  [5.2.7 Local variable names](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s5.2.7-local-variable-names)  [5.2.8 Type variable names](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s5.2.8-type-variable-names)  [5.3 Camel case: defined](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s5.3-camel-case)  [6 Programming Practices](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s6-programming-practices)  [6.1 @Override: always used](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s6.1-override-annotation)  [6.2 Caught exceptions: not ignored](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s6.2-caught-exceptions)  [6.3 Static members: qualified using class](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s6.3-static-members)  [6.4 Finalizers: not used](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s6.4-finalizers)  [7 Javadoc](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s7-javadoc)  [7.1 Formatting](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s7.1-javadoc-formatting)  [7.1.1 General form](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s7.1.1-javadoc-multi-line)  [7.1.2 Paragraphs](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s7.1.2-javadoc-paragraphs)  [7.1.3 At-clauses](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s7.1.3-javadoc-at-clauses)  [7.2 The summary fragment](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s7.2-summary-fragment)  [7.3 Where Javadoc is used](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s7.3-javadoc-where-required)  [7.3.1 Exception: self-explanatory methods](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s7.3.1-javadoc-exception-self-explanatory)  [7.3.2 Exception: overrides](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s7.3.2-javadoc-exception-overrides) |

## 1 Introduction

This document serves as the **complete** definition of Google's coding standards for source code in the Java™ Programming Language. A Java source file is described as being in Google Style if and only if it adheres to the rules herein.

Like other programming style guides, the issues covered span not only aesthetic issues of formatting, but other types of conventions or coding standards as well. However, this document focuses primarily on the **hard-and-fast rules**that we follow universally, and avoids giving advice that isn't clearly enforceable (whether by human or tool).

### 1.1 Terminology notes

In this document, unless otherwise clarified:

1. The term class is used inclusively to mean an "ordinary" class, enum class, interface or annotation type (@interface).
2. The term comment always refers to implementation comments. We do not use the phrase "documentation comments", instead using the common term "Javadoc."

Other "terminology notes" will appear occasionally throughout the document.

### 1.2 Guide notes

Example code in this document is **non-normative**. That is, while the examples are in Google Style, they may not illustrate the only stylish way to represent the code. Optional formatting choices made in examples should not be enforced as rules.

## 2 Source file basics

### 2.1 File name

The source file name consists of the case-sensitive name of the top-level class it contains, plus the .java extension.

### 2.2 File encoding: UTF-8

Source files are encoded in **UTF-8**.

### 2.3 Special characters

#### 2.3.1 Whitespace characters

Aside from the line terminator sequence, the **ASCII horizontal space character** (**0x20**) is the only whitespace character that appears anywhere in a source file. This implies that:

1. All other whitespace characters in string and character literals are escaped.
2. Tab characters are **not** used for indentation.

#### 2.3.2 Special escape sequences

For any character that has a special escape sequence (\b, \t, \n, \f, \r, \", \' and \\), that sequence is used rather than the corresponding octal (e.g. \012) or Unicode (e.g. \u000a) escape.

#### 2.3.3 Non-ASCII characters

For the remaining non-ASCII characters, either the actual Unicode character (e.g. ∞) or the equivalent Unicode escape (e.g. \u221e) is used, depending only on which makes the code **easier to read and understand**.

**Tip:** In the Unicode escape case, and occasionally even when actual Unicode characters are used, an explanatory comment can be very helpful.

Examples:

|  |  |
| --- | --- |
| **Example** | **Discussion** |
| String unitAbbrev = "μs"; | Best: perfectly clear even without a comment. |
| String unitAbbrev = "\u03bcs"; // "μs" | Allowed, but there's no reason to do this. |
| String unitAbbrev = "\u03bcs"; // Greek letter mu, "s" | Allowed, but awkward and prone to mistakes. |
| String unitAbbrev = "\u03bcs"; | Poor: the reader has no idea what this is. |
| return '\ufeff' + content; // byte order mark | Good: use escapes for non-printable characters, and comment if necessary. |

**Tip:** Never make your code less readable simply out of fear that some programs might not handle non-ASCII characters properly. If that should happen, those programs are **broken** and they must be **fixed**.

## 3 Source file structure

A source file consists of, **in order**:

1. License or copyright information, if present
2. Package statement
3. Import statements
4. Exactly one top-level class

**Exactly one blank line** separates each section that is present.

### 3.1 License or copyright information, if present

If license or copyright information belongs in a file, it belongs here.

### 3.2 Package statement

The package statement is **not line-wrapped**. The column limit (Section 4.4, [Column limit: 80 or 100](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s4.4-column-limit)) does not apply to package statements.

### 3.3 Import statements

#### 3.3.1 No wildcard imports

**Wildcard imports**, static or otherwise, **are not used**.

#### 3.3.2 No line-wrapping

Import statements are **not line-wrapped**. The column limit (Section 4.4, [Column limit: 80 or 100](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s4.4-column-limit)) does not apply to import statements.

#### 3.3.3 Ordering and spacing

Import statements are divided into the following groups, in this order, with each group separated by a single blank line:

1. All static imports in a single group
2. com.google imports (only if this source file is in the com.google package space)
3. Third-party imports, one group per top-level package, in ASCII sort order

for example: android, com, junit, org, sun

1. java imports
2. javax imports

Within a group there are no blank lines, and the imported names appear in ASCII sort order. (**Note:** this is not the same as the import statements being in ASCII sort order; the presence of semicolons warps the result.)

### 3.4 Class declaration

#### 3.4.1 Exactly one top-level class declaration

Each top-level class resides in a source file of its own.

#### 3.4.2 Class member ordering

The ordering of the members of a class can have a great effect on learnability, but there is no single correct recipe for how to do it. Different classes may order their members differently.

What is important is that each class order its members in some **logical order**, which its maintainer could explain if asked. For example, new methods are not just habitually added to the end of the class, as that would yield "chronological by date added" ordering, which is not a logical ordering.

##### 3.4.2.1 Overloads: never split

When a class has multiple constructors, or multiple methods with the same name, these appear sequentially, with no intervening members.

## 4 Formatting

**Terminology Note:** block-like construct refers to the body of a class, method or constructor. Note that, by Section 4.8.3.1 on [array initializers](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s4.8.3.1-array-initializers), any array initializer may optionally be treated as if it were a block-like construct.

### 4.1 Braces

#### 4.1.1 Braces are used where optional

Braces are used with if, else, for, do and while statements, even when the body is empty or contains only a single statement.

#### 4.1.2 Nonempty blocks: K & R style

Braces follow the Kernighan and Ritchie style ("[Egyptian brackets](http://www.codinghorror.com/blog/2012/07/new-programming-jargon.html)") for nonempty blocks and block-like constructs:

* No line break before the opening brace.
* Line break after the opening brace.
* Line break before the closing brace.
* Line break after the closing brace if that brace terminates a statement or the body of a method, constructor or named class. For example, there is no line break after the brace if it is followed by else or a comma.

Example:

**public** MyClass **getMyClass**() {

**return** **new** MyClass() {

***@Override***

**public** **void** **method**() {

**if** (condition()) {

**try** {

something();

}

**catch** (**ProblemException** **e**) {

recover();

}

}

}

};

}

A few exceptions for enum classes are given in Section 4.8.1, [Enum classes](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s4.8.1-enum-classes).

#### 4.1.3 Empty blocks: may be concise

An empty block or block-like construct may be closed immediately after it is opened, with no characters or line break in between ({}), **unless** it is part of a multi-block statement (one that directly contains multiple blocks: if/else-if/else or try/catch/finally).

Example:

void doNothing() {}

### 4.2 Block indentation: +2 spaces

Each time a new block or block-like construct is opened, the indent increases by two spaces. When the block ends, the indent returns to the previous indent level. The indent level applies to both code and comments throughout the block. (See the example in Section 4.1.2, [Nonempty blocks: K & R Style](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s4.1.2-blocks-k-r-style).)

### 4.3 One statement per line

Each statement is followed by a line-break.

### 4.4 Column limit: 80 or 100

Projects are free to choose a column limit of either 80 or 100 characters. Except as noted below, any line that would exceed this limit must be line-wrapped, as explained in Section 4.5, [Line-wrapping](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s4.5-line-wrapping).

**Exceptions:**

1. Lines where obeying the column limit is not possible (for example, a long URL in Javadoc, or a long JSNI method reference).
2. package and import statements (see Sections 3.2 [Package statement](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s3.2-package-statement) and 3.3 [Import statements](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s3.3-import-statements)).
3. Command lines in a comment that may be cut-and-pasted into a shell.

### 4.5 Line-wrapping

**Terminology Note:** When code that might otherwise legally occupy a single line is divided into multiple lines, typically to avoid overflowing the column limit, this activity is called line-wrapping.

There is no comprehensive, deterministic formula showing exactly how to line-wrap in every situation. Very often there are several valid ways to line-wrap the same piece of code.

**Tip:** Extracting a method or local variable may solve the problem without the need to line-wrap.

#### 4.5.1 Where to break

The prime directive of line-wrapping is: prefer to break at a **higher syntactic level**. Also:

1. When a line is broken at a non-assignment operator the break comes before the symbol. (Note that this is not the same practice used in Google style for other languages, such as C++ and JavaScript.)
   * This also applies to the following "operator-like" symbols: the dot separator (.), the ampersand in type bounds (<T extends Foo & Bar>), and the pipe in catch blocks (catch (FooException | BarException e)).
2. When a line is broken at an assignment operator the break typically comes after the symbol, but either way is acceptable.
   * This also applies to the "assignment-operator-like" colon in an enhanced for ("foreach") statement.
3. A method or constructor name stays attached to the open parenthesis (() that follows it.
4. A comma (,) stays attached to the token that precedes it.

#### 4.5.2 Indent continuation lines at least +4 spaces

When line-wrapping, each line after the first (each continuation line) is indented at least +4 from the original line.

When there are multiple continuation lines, indentation may be varied beyond +4 as desired. In general, two continuation lines use the same indentation level if and only if they begin with syntactically parallel elements.

Section 4.6.3 on [Horizontal alignment](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s4.6.3-horizontal-alignment) addresses the discouraged practice of using a variable number of spaces to align certain tokens with previous lines.

### 4.6 Whitespace

#### 4.6.1 Vertical Whitespace

A single blank line appears:

1. Between consecutive members (or initializers) of a class: fields, constructors, methods, nested classes, static initializers, instance initializers.
   * **Exception:** A blank line between two consecutive fields (having no other code between them) is optional. Such blank lines are used as needed to create logical groupings of fields.
2. Within method bodies, as needed to create logical groupings of statements.
3. Optionally before the first member or after the last member of the class (neither encouraged nor discouraged).
4. As required by other sections of this document (such as Section 3.3, [Import statements](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s3.3-import-statements)).

Multiple consecutive blank lines are permitted, but never required (or encouraged).

#### 4.6.2 Horizontal whitespace

Beyond where required by the language or other style rules, and apart from literals, comments and Javadoc, a single ASCII space also appears in the following places **only**.

1. Separating any reserved word, such as if, for or catch, from an open parenthesis (() that follows it on that line
2. Separating any reserved word, such as else or catch, from a closing curly brace (}) that precedes it on that line
3. Before any open curly brace ({), with two exceptions:
   * @SomeAnnotation({a, b}) (no space is used)
   * String[][] x = {{"foo"}}; (no space is required between {{, by item 8 below)
4. On both sides of any binary or ternary operator. This also applies to the following "operator-like" symbols:
   * the ampersand in a conjunctive type bound: <T extends Foo & Bar>
   * the pipe for a catch block that handles multiple exceptions: catch (FooException | BarException e)
   * the colon (:) in an enhanced for ("foreach") statement
5. After ,:; or the closing parenthesis ()) of a cast
6. On both sides of the double slash (//) that begins an end-of-line comment. Here, multiple spaces are allowed, but not required.
7. Between the type and variable of a declaration: List<String> list
8. Optional just inside both braces of an array initializer
   * new int[] {5, 6} and new int[] { 5, 6 } are both valid

**Note:** This rule never requires or forbids additional space at the start or end of a line, only interior space.

#### 4.6.3 Horizontal alignment: never required

**Terminology Note:** Horizontal alignment is the practice of adding a variable number of additional spaces in your code with the goal of making certain tokens appear directly below certain other tokens on previous lines.

This practice is permitted, but is **never required** by Google Style. It is not even required to maintain horizontal alignment in places where it was already used.

Here is an example without alignment, then using alignment:

private int x; // this is fine

private Color color; // this too

private int x; // permitted, but future edits

private Color color; // may leave it unaligned

**Tip:** Alignment can aid readability, but it creates problems for future maintenance. Consider a future change that needs to touch just one line. This change may leave the formerly-pleasing formatting mangled, and that is **allowed**. More often it prompts the coder (perhaps you) to adjust whitespace on nearby lines as well, possibly triggering a cascading series of reformattings. That one-line change now has a "blast radius." This can at worst result in pointless busywork, but at best it still corrupts version history information, slows down reviewers and exacerbates merge conflicts.

### 4.7 Grouping parentheses: recommended

Optional grouping parentheses are omitted only when author and reviewer agree that there is no reasonable chance the code will be misinterpreted without them, nor would they have made the code easier to read. It is not reasonable to assume that every reader has the entire Java operator precedence table memorized.

### 4.8 Specific constructs

#### 4.8.1 Enum classes

After each comma that follows an enum constant, a line-break is optional.

An enum class with no methods and no documentation on its constants may optionally be formatted as if it were an array initializer (see Section 4.8.3.1 on [array initializers](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s4.8.3.1-array-initializers)).

private enum Suit { CLUBS, HEARTS, SPADES, DIAMONDS }

Since enum classes are classes, all other rules for formatting classes apply.

#### 4.8.2 Variable declarations

##### 4.8.2.1 One variable per declaration

Every variable declaration (field or local) declares only one variable: declarations such as int a, b; are not used.

##### 4.8.2.2 Declared when needed, initialized as soon as possible

Local variables are **not** habitually declared at the start of their containing block or block-like construct. Instead, local variables are declared close to the point they are first used (within reason), to minimize their scope. Local variable declarations typically have initializers, or are initialized immediately after declaration.

#### 4.8.3 Arrays

##### 4.8.3.1 Array initializers: can be "block-like"

Any array initializer may optionally be formatted as if it were a "block-like construct." For example, the following are all valid (**not** an exhaustive list):

new int[] { new int[] { 0, 1, 2, 3 0,} 1, 2,new int[] { 3, 0, 1, } 2, 3} new int[] {0, 1, 2, 3}

##### 4.8.3.2 No C-style array declarations

The square brackets form a part of the type, not the variable: String[] args, not String args[].

#### 4.8.4 Switch statements

**Terminology Note:** Inside the braces of a switch block are one or more statement groups. Each statement group consists of one or more switch labels (either case FOO: or default:), followed by one or more statements.

##### 4.8.4.1 Indentation

As with any other block, the contents of a switch block are indented +2.

After a switch label, a newline appears, and the indentation level is increased +2, exactly as if a block were being opened. The following switch label returns to the previous indentation level, as if a block had been closed.

##### 4.8.4.2 Fall-through: commented

Within a switch block, each statement group either terminates abruptly (with a break, continue, return or thrown exception), or is marked with a comment to indicate that execution will or might continue into the next statement group. Any comment that communicates the idea of fall-through is sufficient (typically // fall through). This special comment is not required in the last statement group of the switch block. Example:

switch (input) {

case 1:

case 2:

prepareOneOrTwo(); // fall through

case 3:

handleOneTwoOrThree();

break;

default:

handleLargeNumber(input);

}

##### 4.8.4.3 The default case is present

Each switch statement includes a default statement group, even if it contains no code.

#### 4.8.5 Annotations

Annotations applying to a class, method or constructor appear immediately after the documentation block, and each annotation is listed on a line of its own (that is, one annotation per line). These line breaks do not constitute line-wrapping (Section 4.5, [Line-wrapping](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s4.5-line-wrapping)), so the indentation level is not increased. Example:

@Override@Nullablepublic String getNameIfPresent() { ... }

**Exception:** A single parameterless annotation may instead appear together with the first line of the signature, for example:

@Override public int hashCode() { ... }

Annotations applying to a field also appear immediately after the documentation block, but in this case, multiple annotations (possibly parameterized) may be listed on the same line; for example:

@Partial @Mock DataLoader loader;

There are no specific rules for formatting parameter and local variable annotations.

#### 4.8.6 Comments

##### 4.8.6.1 Block comment style

Block comments are indented at the same level as the surrounding code. They may be in /\* ... \*/ style or // ... style. For multi-line /\* ... \*/ comments, subsequent lines must start with \* aligned with the \* on the previous line.

/\*

\* This is

\* okay

\*/

// And so

// is this.

/\* Or you can

\* even do this. \*/

Comments are not enclosed in boxes drawn with asterisks or other characters.

**Tip:** When writing multi-line comments, use the /\* ... \*/ style if you want automatic code formatters to re-wrap the lines when necessary (paragraph-style). Most formatters don't re-wrap lines in // ... style comment blocks.

#### 4.8.7 Modifiers

Class and member modifiers, when present, appear in the order recommended by the Java Language Specification:

public protected private abstract static final transient volatile synchronized native strictfp

#### 4.8.8 Numeric Literals

long-valued integer literals use an uppercase L suffix, never lowercase (to avoid confusion with the digit 1). For example, 3000000000L rather than 3000000000l.

## 5 Naming

### 5.1 Rules common to all identifiers

Identifiers use only ASCII letters and digits, and in two cases noted below, underscores. Thus each valid identifier name is matched by the regular expression \w+ .

In Google Style special prefixes or suffixes, like those seen in the examples name\_, mName, s\_name and kName, are **not** used.

### 5.2 Rules by identifier type

#### 5.2.1 Package names

Package names are all lowercase, with consecutive words simply concatenated together (no underscores). For example, com.example.deepspace, not com.example.deepSpace or com.example.deep\_space.

#### 5.2.2 Class names

Class names are written in [UpperCamelCase](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s5.3-camel-case).

Class names are typically nouns or noun phrases. For example, Character or ImmutableList. Interface names may also be nouns or noun phrases (for example, List), but may sometimes be adjectives or adjective phrases instead (for example,Readable).

There are no specific rules or even well-established conventions for naming annotation types.

Test classes are named starting with the name of the class they are testing, and ending with Test. For example, HashTest or HashIntegrationTest.

#### 5.2.3 Method names

Method names are written in [lowerCamelCase](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s5.3-camel-case).

Method names are typically verbs or verb phrases. For example, sendMessage or stop.

Underscores may appear in JUnit test method names to separate logical components of the name. One typical pattern is test*<MethodUnderTest>*\_*<state>*, for example testPop\_emptyStack. There is no One Correct Way to name test methods.

#### 5.2.4 Constant names

Constant names use CONSTANT\_CASE: all uppercase letters, with words separated by underscores. But what is a constant, exactly?

Every constant is a static final field, but not all static final fields are constants. Before choosing constant case, consider whether the field really feels like a constant. For example, if any of that instance's observable state can change, it is almost certainly not a constant. Merely intending to never mutate the object is generally not enough. Examples:

// Constants

**static** **final** **int** ***NUMBER*** = 5;

**static** **final** **ImmutableList**<String> ***NAMES*** = ImmutableList.of("Ed", "Ann");

**static** **final** **Joiner** ***COMMA\_JOINER*** = Joiner.on(','); // because Joiner is immutable

**static** **final** **SomeMutableType**[] ***EMPTY\_ARRAY*** = {};

**enum** *SomeEnum* { ***ENUM\_CONSTANT*** }

// Not constants

**static** **String** *nonFinal* = "non-final";

**final** **String** nonStatic = "non-static";

**static** **final** Set<String> ***mutableCollection*** = **new** HashSet<String>();

**static** **final** **ImmutableSet**<SomeMutableType> ***mutableElements*** = ImmutableSet.of(mutable);

**static** **final** **Logger** ***logger*** = **Logger**.*getLogger*(MyClass.getName());

**static** **final** **String**[] ***nonEmptyArray*** = {"these", "can", "change"};

These names are typically nouns or noun phrases.

#### 5.2.5 Non-constant field names

Non-constant field names (static or otherwise) are written in [lowerCamelCase](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s5.3-camel-case).

These names are typically nouns or noun phrases. For example, computedValues or index.

#### 5.2.6 Parameter names

Parameter names are written in [lowerCamelCase](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s5.3-camel-case).

One-character parameter names should be avoided.

#### 5.2.7 Local variable names

Local variable names are written in [lowerCamelCase](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s5.3-camel-case), and can be abbreviated more liberally than other types of names.

However, one-character names should be avoided, except for temporary and looping variables.

Even when final and immutable, local variables are not considered to be constants, and should not be styled as constants.

#### 5.2.8 Type variable names

Each type variable is named in one of two styles:

* A single capital letter, optionally followed by a single numeral (such as E, T, X, T2)
* A name in the form used for classes (see Section 5.2.2, [Class names](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html" \l "s5.2.2-class-names)), followed by the capital letter T (examples: RequestT, FooBarT).

### 5.3 Camel case: defined

Sometimes there is more than one reasonable way to convert an English phrase into camel case, such as when acronyms or unusual constructs like "IPv6" or "iOS" are present. To improve predictability, Google Style specifies the following (nearly) deterministic scheme.

Beginning with the prose form of the name:

1. Convert the phrase to plain ASCII and remove any apostrophes. For example, "Müller's algorithm" might become "Muellers algorithm".
2. Divide this result into words, splitting on spaces and any remaining punctuation (typically hyphens).
   * Recommended: if any word already has a conventional camel-case appearance in common usage, split this into its constituent parts (e.g., "AdWords" becomes "ad words"). Note that a word such as "iOS" is not really in camel caseper se; it defies any convention, so this recommendation does not apply.
3. Now lowercase everything (including acronyms), then uppercase only the first character of:
   * ... each word, to yield upper camel case, or
   * ... each word except the first, to yield lower camel case
4. Finally, join all the words into a single identifier.

Note that the casing of the original words is almost entirely disregarded. Examples:

|  |  |  |
| --- | --- | --- |
| **Prose form** | **Correct** | **Incorrect** |
| "XML HTTP request" | XmlHttpRequest | XMLHTTPRequest |
| "new customer ID" | newCustomerId | newCustomerID |
| "inner stopwatch" | innerStopwatch | innerStopWatch |
| "supports IPv6 on iOS?" | supportsIpv6OnIos | supportsIPv6OnIOS |
| "YouTube importer" | YouTubeImporter YoutubeImporter\* |  |

\*Acceptable, but not recommended.

**Note:** Some words are ambiguously hyphenated in the English language: for example "nonempty" and "non-empty" are both correct, so the method names checkNonempty and checkNonEmpty are likewise both correct.

## 6 Programming Practices

### 6.1 @Override: always used

A method is marked with the @Override annotation whenever it is legal. This includes a class method overriding a superclass method, a class method implementing an interface method, and an interface method respecifying a superinterface method.

**Exception:**@Override may be omitted when the parent method is @Deprecated.

### 6.2 Caught exceptions: not ignored

Except as noted below, it is very rarely correct to do nothing in response to a caught exception. (Typical responses are to log it, or if it is considered "impossible", rethrow it as an AssertionError.)

When it truly is appropriate to take no action whatsoever in a catch block, the reason this is justified is explained in a comment.

try { int i = Integer.parseInt(response); return handleNumericResponse(i);} catch (NumberFormatException ok) { // it's not numeric; that's fine, just continue}return handleTextResponse(response);

**Exception:** In tests, a caught exception may be ignored without comment if it is named expected. The following is a very common idiom for ensuring that the method under test does throw an exception of the expected type, so a comment is unnecessary here.

try { emptyStack.pop(); fail();} catch (NoSuchElementException expected) {}

### 6.3 Static members: qualified using class

When a reference to a static class member must be qualified, it is qualified with that class's name, not with a reference or expression of that class's type.

Foo aFoo = ...;Foo.aStaticMethod(); // goodaFoo.aStaticMethod(); // badsomethingThatYieldsAFoo().aStaticMethod(); // very bad

### 6.4 Finalizers: not used

It is **extremely rare** to override Object.finalize.

**Tip:** Don't do it. If you absolutely must, first read and understand [Effective Java](http://books.google.com/books?isbn=8131726592) Item 7, "Avoid Finalizers," very carefully, and then don't do it.

## 7 Javadoc

### 7.1 Formatting

#### 7.1.1 General form

The basic formatting of Javadoc blocks is as seen in this example:

/\*\* \* Multiple lines of Javadoc text are written here, \* wrapped normally... \*/public int method(String p1) { ... }

... or in this single-line example:

/\*\* An especially short bit of Javadoc. \*/

The basic form is always acceptable. The single-line form may be substituted when there are no at-clauses present, and the entirety of the Javadoc block (including comment markers) can fit on a single line.

#### 7.1.2 Paragraphs

One blank line—that is, a line containing only the aligned leading asterisk (\*)—appears between paragraphs, and before the group of "at-clauses" if present. Each paragraph but the first has <p> immediately before the first word, with no space after.

#### 7.1.3 At-clauses

Any of the standard "at-clauses" that are used appear in the order @param, @return, @throws, @deprecated, and these four types never appear with an empty description. When an at-clause doesn't fit on a single line, continuation lines are indented four (or more) spaces from the position of the @.

### 7.2 The summary fragment

The Javadoc for each class and member begins with a brief **summary fragment**. This fragment is very important: it is the only part of the text that appears in certain contexts such as class and method indexes.

This is a fragment—a noun phrase or verb phrase, not a complete sentence. It does **not** begin with A {@code Foo} is a..., or This method returns..., nor does it form a complete imperative sentence like Save the record.. However, the fragment is capitalized and punctuated as if it were a complete sentence.

**Tip:** A common mistake is to write simple Javadoc in the form /\*\* @return the customer ID \*/. This is incorrect, and should be changed to /\*\* Returns the customer ID. \*/.

### 7.3 Where Javadoc is used

At the minimum, Javadoc is present for every public class, and every public or protected member of such a class, with a few exceptions noted below.

Other classes and members still have Javadoc as needed. Whenever an implementation comment would be used to define the overall purpose or behavior of a class, method or field, that comment is written as Javadoc instead. (It's more uniform, and more tool-friendly.)

#### 7.3.1 Exception: self-explanatory methods

Javadoc is optional for "simple, obvious" methods like getFoo, in cases where there really and truly is nothing else worthwhile to say but "Returns the foo".

**Important:** it is not appropriate to cite this exception to justify omitting relevant information that a typical reader might need to know. For example, for a method named getCanonicalName, don't omit its documentation (with the rationale that it would say only /\*\* Returns the canonical name. \*/) if a typical reader may have no idea what the term "canonical name" means!

#### 7.3.2 Exception: overrides

Javadoc is not always present on a method that overrides a supertype method.

Last changed: March 21, 2014