

NPU BUS

软件编码规范

编写人员： 刘龙涛、李云飞、王琪、刘峥、李宗明

校对人员：刘嘉林、康浩然、苏畅、王泽堃、史明明

编写日期： 2019年7月2日

文档版本： 1.0版

**目 录**

[1引言 6](#_Toc301520687)

[1.1文档标识 6](#_Toc301520688)

[1.2项目概述 6](#_Toc301520689)

[1.3文档概述 7](#_Toc301520690)

[1.4参考文档 7](#_Toc301520691)

[2 语句规范 8](#_Toc301520692)

[2.1 简单语句 8](#_Toc301520693)

[2.2 复合语句 8](#_Toc301520694)

[2.3 返回语句 8](#_Toc301520695)

[2.4 if, if-else, if else-if else语句 9](#_Toc301520696)

[2.5 for语句 10](#_Toc301520697)

[2.6 while语句 10](#_Toc301520698)

[2.7 do-while语句 11](#_Toc301520699)

[2.8 switch语句 11](#_Toc301520700)

[2.9 try-catch语句 12](#_Toc301520701)

[2.10 异常处理 13](#_Toc301520702)

[2.10.1不要在没有说明的情况下捕捉和忽略它们。 13](#_Toc301520703)

[2.10.2不要捕捉一般异常 15](#_Toc301520704)

[2.10.3 Finalizers：一般不使用。 15](#_Toc301520705)

[3命名规范 16](#_Toc301520706)

[3.1 包 16](#_Toc301520707)

[3.2 类 17](#_Toc301520708)

[3.3 接口 17](#_Toc301520709)

[3.4 方法 17](#_Toc301520710)

[3.4.1 JAVATESTS测试方法的命名 18](#_Toc301520711)

[3.5 实例变量 18](#_Toc301520712)

[3.6 常量 18](#_Toc301520713)

[3.7文件命名 19](#_Toc301520714)

[3.7.1代码文件命名 19](#_Toc301520715)

[3.7.2 res/layout文件命名 19](#_Toc301520716)

[3.7.3 raw/drawable文件命名 20](#_Toc301520717)

[4注释规范 20](#_Toc301520718)

[4.1 注释的格式 21](#_Toc301520719)

[4.1.1块注释 21](#_Toc301520720)

[4.1.2 单行注释(Single-Line Comments) 22](#_Toc301520721)

[4.1.3 尾端注释(Trailing Comments) 23](#_Toc301520722)

[4.1.4 行末注释(End-Of-Line Comments) 23](#_Toc301520723)

[4.2 开头注释 24](#_Toc301520724)

[4.3 类和接口的文档注释 25](#_Toc301520725)

[4.3 方法的文档注释 27](#_Toc301520726)

[5样式规范 28](#_Toc301520727)

[5.1文件组织 28](#_Toc301520728)

[5.1.1开头注释语句 29](#_Toc301520729)

[5.1.2包和引入语句 29](#_Toc301520730)

[5.1.3类和接口语句 29](#_Toc301520731)

[5.2 与声明相关的 31](#_Toc301520732)

[5.2.1每行声明变量的数量 31](#_Toc301520733)

[5.2.2初始化 31](#_Toc301520734)

[5.2.3布局 32](#_Toc301520735)

[5.2.4类和接口的声明 33](#_Toc301520736)

[5.3缩进排版 34](#_Toc301520737)

[5.3.1行长度 34](#_Toc301520738)

[5.3.2换行 34](#_Toc301520739)

[5.4空白 37](#_Toc301520740)

[5.4.1空行 37](#_Toc301520741)

[5.4.2空格 38](#_Toc301520742)

[5.5编程惯例 39](#_Toc301520743)

[5.5.1提供对实例以及类变量的访问控制 39](#_Toc301520744)

[5.5.2引用类变量和类方法 39](#_Toc301520745)

[5.5.3常量 40](#_Toc301520746)

[5.5.4变量赋值 40](#_Toc301520747)

[5.5.5其它惯例 41](#_Toc301520748)

[6代码检查工具Checkstyle介绍 43](#_Toc301520749)

[6.1 Windows7系统下Eclipse + Checkstyle环境配置 43](#_Toc301520750)

[6. 标准应用 44](#_Toc301520751)

[6.3 高级应用 44](#_Toc301520752)

[7示例代码 45](#_Toc301520753)

# 1引言

## 1.1文档标识

中文名称：《软件编码规范》。

英文名称：“Software Coding Standard（SCS）”。

文档版本：“1.0”。

文档编号：“NPU-BUS-SCS-1.0”。

## 1.2项目概述

本文档适用于“西北工业大学校车管理开发”项目（以下简称“NPUBUS项目”）的开发过程。NPUBUS项目，由本小组负责实施，该项目标识号为“NPU-BUS”，其软件产品版本号为“1.0”，包括三个内部版本，分别是0.1版、0.2版和0.3版。

项目内容为：

本产品是一款校车服务平台，分为管理层面和用户层面。目的是为了解决近年来，随着

新老校区间的课程安排调度以及每年的学生数量不断增加，乘坐校车在新老校区来往的人数也相应有所增多，对校车工作方的校车调度和发车数量控制有了更高的要求。鉴于校车工作方的数据来源仅仅是根据个人的经验去判断，尚未实现数字化规范化程序化，导致经常有学生等待数量及校车数量之间出现了矛盾，资源未能充分利用好、学生和老师权利不得已正常落实的情况等现实问题。这个平台可以实现给校车工作方提供准确的人数大数据，校车工作方能根据实时情况智能调度学校班车；能够让学生免去苦苦等待校车却上不了校车的情况。

## 1.3文档概述

本文档依据国家标准[《GB/T 8567-2006计算机软件文档编制规范》](../../资料/GBT%208567-2006%20计算机软件文档编制规范.pdf)制定，属于技术文档，仅限于联合实验室和中兴通讯的项目相关人员阅读。

编码规范对于程序员而言尤为重要。该文档作为本项目开发过程中编码指导文档，规范程序员的编码习惯，提高编码效率，提升工程质量。

为了执行规范，每个软件开发人员必须一致遵守编码规范。

## 1.4参考文档

* [《GB/T 8567-2006计算机软件文档编制规范》](../../资料/GBT%208567-2006%20计算机软件文档编制规范.pdf)，国家标准

# 2 语句规范

## 2.1 简单语句

每行至多包含一条语句，例如：

argv++; // Correct

argc--; // Correct

argv++; argc--; // AVOID!

## 2.2 复合语句

复合语句是包含在大括号中的语句序列，形如“{ 语句 }”。例如下面各段。

◆ 被括其中的语句应该较之复合语句缩进一个层次。

◆ 左大括号“{”应位于复合语句起始行的行尾；右大括号“}”应另起一行并与复合语句首行对齐。

◆ 大括号可以被用于所有语句，包括单个语句，只要这些语句是诸如if – else或for控制结构的一部分。这样便于添加语句而无需担心由于忘了加括号而引入bug。

## 2.3 返回语句

一个带返回值的return语句不使用小括号“()”，除非它们以某种方式使返回值更显见。例如：

return;

return myDisk.size();

return (size ? size ; defaultSize);

## 2.4 if, if-else, if else-if else语句

if-else语句格式如下：

if (condition) {

statements;

}

或

if (condition) {

statements;

} else {

statements;

}

或

if (condition) {

statements;

} else if (condition) {

statements;

} else if (condition) {

statements;

}

注意：if语句问题用“{”和“}”括起来，避免使用如下容易引起错误的格式：

if (condition) // AVOI! THIS OMITS THE BRACES {}!

statement;

## 2.5 for语句

一个for语句格式如下：

for (initialization; condition; update) {

statements;

}

一个空的语句(所有工作都在初始化，条件判断，更新子句中完成)应该具有如下格式：

for (initialization; condition; update);

当在for语句的初始化或更新子句中使用逗号时，避免因使用三个以上变量，而导致复杂度提高。若需要，可以在for循环之前(为初始化子句)或for循环末尾(为更新子句)使用单独的语句。

## 2.6 while语句

一个while语句应该具有如下格式：

while (condition) {

statements;

}

一个空的while语句应该具有如下格式：

while (condition);

## 2.7 do-while语句

一个do-while语句应该具有如下格式：

do {

statements;

} while (condition);

## 2.8 switch语句

一个switch语句格式如下：

switch (condition) {

case ABC:

statements;

/ \* falls through \*/

case DEF:

statements;

break;

case XYZ:

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/ \* falls through \*/。

## 2.9 try-catch语句

一个try-catch语句格式如下：

try {

statements;

} catch (ExceptionClass e) {

statements;

}

一个try-catch语句后面也可能跟着一个finally语句，不论try代码块是否顺利执行完，它都会被执行。

try {

statements;

} catch (ExceptionClass e) {

statements;

} finally {

statements;

}

## 2.10 异常处理

### 2.10.1不要在没有说明的情况下捕捉和忽略它们。

有时很容易编写完全忽略异常的代码，比如：

void setServerPort(String value) {

try {

serverPort = Integer.parseInt(value);

} catch (NumberFormatException e) {

}

}

原则上，你必须在你代码中处理每一个异常。特殊的处理要视情况而定。可接受的一个方案（为了性能考虑）是：

样式一： 将异常抛出给方法的调用者。

void setServerPort(String value) throws NumberFormatException {

serverPort = Integer.parseInt(value);

}

样式二：抛出一个新的适用于抽象等级的异常。

void setServerPort(String value) throws ConfigurationException {

try {

serverPort = Integer.parseInt(value);

} catch (NumberFormatException e) {

throw new ConfigurationException("Port " + value + " is not valid.");

}

样式三：在catch {}块中用合适的代码块来处理异常。

/\*\* Set port. If value is not a valid number, 80 is substituted. \*/

void setServerPort(String value) {

try {

serverPort = Integer.parseInt(value);

} catch (NumberFormatException e) {

serverPort = 80; // default port for server

}

样式四：捕获异常并抛出一个新的运行时异常。这是非常危险的。

/\*\* Set port. If value is not a valid number, die. \*/

void setServerPort(String value) {

try {

serverPort = Integer.parseInt(value);

} catch (NumberFormatException e) {

throw new RuntimeException("port " + value " is invalid, ", e);

}

### 2.10.2不要捕捉一般异常

避免以下操作：

try {

someComplicatedIOFunction(); // may throw IOException

someComplicatedParsingFunction(); // may throw ParsingException

someComplicatedSecurityFunction(); // may throw SecurityException

// phew, made it all the way

} catch (Exception e) { // I'll just catch all exceptions

handleError(); // with one generic handler!

}

### 2.10.3 Finalizers：一般不使用。

当一个对象被垃圾回收时，Finalizers能够去执行某段代码。

优点: 可以方便清理，特别是外部资源。

缺点: 不能保证finalizer什么时候会调用，或甚至根本上没被调用。

# 3命名规范

命名规范的目的是使程序更易读。它们也可以提供一些有关标识符功能的信息。以助于理解代码，不论它是一个常量、包、还是类。

命名过程中应做到以下几点：使用完整的英文描述来命名；避免命名超长（控制在15个字符以内）；避免相似的命名，例如：persistentObj和persistentObjs不要一起使用；anSqlStmt和anSQLStmt不要一起使用；慎用缩写，如果要用到缩写，则按照缩写规则使用缩写，例如：No.代表number数字，ID代表identification标示，下面分类介绍命名规范。

## 3.1 包

由于Android开发应用中，各个应用的包名一定要区分，否则安装后的应用程序会覆盖先前安装的庆用，所以要求各个应用的包名命名需特别关注。

所有项目包名以android.customui开头。此目录只允许建立文件夹，不直接存放具体文件。后面是程序所在项目的英文名称（小写），再下为子系统的名称，每个子系统内按照类别区分。如：

android.customui.app

android.customui.app.weathereport

## 3.2 类

类名是一个名词，采用大小写混合方式，每个单词的首字母大写。尽量使你的类名简洁而富于描述。使用完整单词，避免缩写词(除非该缩写词被广泛的使用，如URL，HTML)。

例如：

class Raster

class ImageSprite

自定义控件的类名前面统一加上“C”,如果是继承重写控件，则C+控件类名。如重写android.widget.Button则命名为Cbutton，其他的类名，如适配器或服务类，可不加C。

## 3.3 接口

命名规则：大小写规则与类名相似，常以“able”、“ible”结尾。如：

interface RasterDelegatible

interface Storingible

## 3.4 方法

方法名是一个动词，采用大小写混合方式，第一个单词的首字母小写，其后单词的首字母大写。

例如：

run()

runFase()

getBackground()

### 3.4.1 JAVATESTS测试方法的命名

示例：testMethod\_specificCase( )。

## 3.5 实例变量

实例变量名应简短且富于描述。变量名的选用应该易于记忆，即，能够指出其用途。尽量避免单个字符的变量名，除非是一个性临时变量。临时变量通常被取名为i，j，k，m和n，它们一般用于整形；c，d，e，它们一般用于字符型。另外：

a. 变量命名采用全小写。

b. 应用名字以app为前缀。

c. 网址以url为前缀。

d. 文字说明或参照内容以string为前缀。

e. 颜色以color为前缀。

f. 密码文字password为前缀。

g. 全数字以digit为前缀。

## 3.6 常量

大小写规则和变量名相似，除了前面需要一个下划线隔开。尽量避免ANSI常量，容易引起错误。

例如：

static final int MIN\_WIDTH = 4

static final int MAX\_WIDTH = 999

static final int GET\_THE\_CPU = 1

注意：日志标记统一定义为静态常量类名：例如：

private static final String TAG = CmdInter.class.getSimplName();

## 3.7文件命名

### 3.7.1代码文件命名

a. 一般的代码文件命名以继承名类名为前缀，例：继承自Activity的文件可写成ActivityMain.java。

b. 一些纯粹的类名命名遵守以大的归类为前缀，可自定，如InfoRank.java。

c. 首字母及新单词开始必须大写。

### 3.7.2 res/layout文件命名

要求xml文件加个前缀以便区别。

a. 命名以全小写，单词不缩写。

b. 布局文件的命名可参照代码文件的命名，一般来说一个页面一个xml,对应也会有一个java代码，对应的java和xml命名可相同。

c. 对于一些属性文件命名以css为前缀。

### 3.7.3 raw/drawable文件命名

要求每个资源文件加个前缀以便区别。

a. 命名全以小写和下划线“\_”组合命名，单词不缩写。

b. 图片的命名必须唯一，整套图片直接存放于drawable文件夹下，以命名分类。

c. 按钮类图片命名以button为前缀，如button\_ok, button\_cancel。

d. 背景类图片命名以back为前缀。

e. 显示类图片命名以view为前缀。

f. 展示类图片命名以dis为前缀，如幻灯片播放。

g. 临时用图命名以tmp为前缀。

h. 其他图上命名以other为前缀。

# 4注释规范

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。

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。注释不应包括诸如制表符和回退符之类 的特殊字符。

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。

## 4.1 注释的格式

程序有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

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，比如方法的内部。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

\*/

### 4.1.2 单行注释(Single-Line Comments)

单行注释可以显示一行内，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)

尾端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，它可以用来注释掉多行的代码段。

if(foo > 1) {

// Do a double-filp.

……

}

else {

return false;

}

// if (bar > 1) {

//

// // Do a triple-filp.

// ……

// }

// else {

// return false;

// }

## 4.2 开头注释

所有的源文件都应该在开头有一个C语言风格的注释，其中列出数出类名、版本信息，日期和版权声明，所有的文件应该有一个文档注释。 然后紧接着是包和引入语气，每一块以空白行分隔。再下来是类或接口的声明。示例代码如下：

/ \*

\* Classname

\*

\* Version information

\*

\* Date

\*

\* Copyright notice

\* /

package com.android.internal.foo;

import android.os.Blah;

import android.view.Yada;

import java.sql.ResultSet;

import java.sql.SQLException;

/\*\*

\* Does X and Y and provides an abstraction for Z.

\*/

public class Foo {

...

}

## 4.3 类和接口的文档注释

类、接口的注释使用Javadoc标准注释，描述类或接口的用途。

其包含如下信息：

1. 用途。开发人员使用某个类和接口之前，需要知道采用该类和接口的用途。

2. 如何使用。开发人员需要知道该类和接口应如何使用，如果必要的话还需要注明不应该如何使用。

3. 开发维护的日志。一个有关于该类和接口的维护记录：时间、作者、摘要。

示例代码如下：

/\*\*

\* Provides a UPnP stack with Android configuration (WiFi network only) as an application service component.

\* <p>

\* Sends a search for all UPnP devices on instantiation. See the {@link org.teleal.cling.android.AndroidUpnpService}

\* interface for a usage example.

\* </p>

\* <p/>

\* Override the {@link #createRouter(org.teleal.cling.UpnpServiceConfiguration, org.teleal.cling.protocol.ProtocolFactory, android.net.wifi.WifiManager, android.net.ConnectivityManager)}

\* and {@link #createConfiguration(android.net.wifi.WifiManager)} methods to customize the service.

\*

\* @author Christian Bauer

\*/

public class AndroidUpnpServiceImpl extends Service {

… …

}

注意：若想了解有关文档注释和javadoc的详细资料，参见javadoc的主页：

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

## 4.3 方法的文档注释

方法注释包含如下信息：

1.类该方法作用是什么？

2.@param 各个参数说明

3.异常处理。@throws

4.@return 方法返回值、代表的意义（如数值，1：成功，0：失败）。例如：

/\*\*

\* 类方法注释

\* @param param1 数据1

\* @param param1 数据2

\* return 返回测试值、意义

\*/

public String getBossId(int param1, int param2){

… …

return "bob";

}

注意：顶层(top-level)的类和接口是不缩进的，而其成员是缩进的。描述类和接口的注释的第一行会被置于注释的第一行(/ \*\*)不需要缩进；随后的注释每行都缩进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、变量或方法的实现信息，而这些信息又不适合写在文档注释中，则可使用块注释（见4.1.1）或紧跟在声明后面的单行注释（见4.1.2）。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。

# 5样式规范

## 5.1文件组织

一个文件由被空行分割而成的段落以及标识每个段落的可选注释共同组成。超过2000行的程序难以阅读，应该尽量避免。“Java源文件范例”提供了一个页面布局合理的Java程序范例。

每个Java源文件都包含一个单一的公共类或接口。若私有类和接口与一个公共类相关联。可以将它们和公共类放入同个源文件。公共类必须是这个文件中的第一个类和接口。

Java源文件还遵循以下规则：

◆ 开头注释语句(参见“开头注释”)

◆ 包和引入语句(参见“包和引入语句)

◆ 类和接口语句(参见“类和接口声明)

## 5.1.1开头注释语句

参见4.2。

## 5.1.2包和引入语句

在多数Java源文件中，第一个非注释行是包语句行。在它之后可以跟引入语句。例如：

package java.awt;

import java.awt.peer.CanvasPeer;

注意：当你想使用foo包中的Bar类，这有两种可能的方式导入它：

import foo.\*;

import foo.Bar;

分析：样式一的优点 ：大大减少了import声明的数目。样式二的优点 ：很明显的看到哪些类实际在用，对于维护者而言，代码更具有可读性。

决定：使用样式二import 所有的Android代码。一个明确的例外是用于Java标准库（使用java.util .\*，java.io. \*，等等）和单元测试代码（junit.framework .\*）。

## 5.1.3类和接口语句

下表描述了类和接口语句部分以及它们出现的先后次序。参见“Java源文件范例”中一个包含注释的例子。

|  |  |  |
| --- | --- | --- |
|  | 类/接口声明的各部分 | 注 解 |
| 1 | 类/接口文档注释  (/ \*\*…\* /) | 该注释中所包含的信息，参见“文档注释” |
| 2 | 类/接口的声明 |  |
| 3 | 类/接口实现的注释  (/ \*…\* /)  如果有必要的话 | 该注释应包含任何有关整个类或接口的信息，而这些信息又适合作为类/接口文档注释。 |
| 4 | 类的(静态)变量 | 首先是类的public变量，随后是protected变量，再后是包一级别的变量(没有访问修饰符)，最后是private变量。 |
| 5 | 实例变量 | 首先是public变量，随后是protected变量，再后是包一级别的变量(没有访问修饰符)，最后是private变量。 |
| 6 | 构造器 |  |
| 7 | 方法 | 这些方法应该按功能，而非作用域或访问权限，分组。 |

## 5.2 与声明相关的

### 5.2.1每行声明变量的数量

推荐一行一个声明，因为这样以利于写注释。亦即，

int level; // indentation level

int size; // size of table

要优于：

int level, size;

不要将不同类型变量的声明放在同一行，例如：

int foo, fooarry[]; // WRONG!

注意：上面的例子中，在类型和标识之间放了一个空格，另一种被允许的替代方法是使用制表符：

int level; // indentation level

int size; // size of table

Object currentEntry; // currently selected table entry

### 5.2.2初始化

尽量在声明局部变量的同时进行初始化。唯一 不这么做理由是变量的初始值依赖于某些先前发生的计算。

### 5.2.3布局

只在代码块的开始处声明变量(一个块是指任何被包含在大括号“{”和“}”中间的代码)。不要在首次用于该变量时才声明之，这会把注意力不集中的程序员搞糊涂，同时会妨碍代码在该作用域内的可移植性。

void myMethod() {

int int1 = 0;

if (condition) {

int int2 = 0;

…

}

}

该规则的一个例外是for循环的索引变量

for (int i = 0; i < maxLoops; i++) { … }

避免声明的局部变量覆盖上一级声明的变量。例如，不要在内部代码块中声明相同的变量名：

int count;

…

myMethod() {

if (condition) {

int count = 0; // AVOID

…

}

…

}

### 5.2.4类和接口的声明

当编写类和接口时，应该遵守以下格式规则：

◆ 在方法名与其参数列表之前的左括号“(”间不要有空格。

◆ 左大括号“{”位于声明语句同行的末尾。

◆ 右大括号“}”另起一行，与相应的声明语句对齐，除非是一个空语句，“}”应紧跟在“{”之后。

class Sample extends Object {

int ivar1;

int ivar2;

Sample(int i, int j) {

ivar1 = i;

ivar2 = j;

}

int emptyMethod() {}

…

}

方法与方法之间 以空行分隔。

## 5.3缩进排版

4个空格常被作为缩进排版的一个单位。缩进的确切解释并未详细指定(空格vs.制表符)。一个制表符等于8个空格(而非4个)。

### 5.3.1行长度

尽量避免一行长度超过80个字符，因为很多终端和工具不能很好处理之。

### 5.3.2换行

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之：

◆ 在一个逗号后面断开。

◆ 在一个操作符前面断开。

◆ 宁可选择较高级别的(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。

以下是断开方法的一些例子：

someMethod(longExpression1, longExpression2, longExpression3,

longExpression4, longExpression5);

var = someMethod1(Expression1,

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- longNeme5)

+ 4 \* longName6); //PREFFER

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，所以代这以缩进8个空格。

// CONVENTION 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 LING 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)) {

doSomthingAoutIt();

}

这里有三种可行的方法用于处理三元运算表达式：

alpha = (aLongBooleanExpression) ? beta : gamma;

alpha = (aLongBooleanExpression) ? beta

: gamma;

alpha = (aLongBooleanExpression)

? beta

: gamma;

## 5.4空白

### 5.4.1空行

空行将逻辑相关的代码段分隔开，以提高可读性。

下列情况应总是使用两个空行：

◆ 一个源文件的两个片段(section)之间。

◆ 类声明和接口声明之间。

下列情况应该总是使用一个空行：

◆ 两个方法之间。

◆ 方法内的局部变量和方法的第一条语句之间。

◆ 块注释或单行注释之前。

◆ 一个方法内的两个逻辑段之间，用以提高可读性。

### 5.4.2空格

下列情况应该使用空格：

◆ 一个紧跟着括号的关键应该被空格分开，例如：

while (true) {

….

}

注意：空格不应该置于方法名与其左括号之间，这将有助于区分关键字和方法调用。

◆ 空白应该位于参数列表中逗号的后面。

◆ 所有的二元运算符，除了“.”，应该使用空格将之与操作数分开。一元操作符和操作数之间不应该加空格，比如：负号(“-”)，自增(“++”)和自减(“--”)。例如：

a += c + d;

a = (a + b) / (c \* d);

while (d++ + s++) {

n++;

}

printSize(“size is ” + foo + “\n”);

◆ for语句中的表达式应该被空格分开，例如：

for (expr1; expr2; expr3)

◆ 强制转型后应该跟一个空格，例如：

myMethod((byte) aNum, (Object) x);

myMethod((int) (cp + 5), ((int) (i + 3)) + 1);

## 5.5编程惯例

### 5.5.1提供对实例以及类变量的访问控制

不要把实例或类类变量声明为public。通常，实例变量无需显式的设置(set)和获取(gotten)，通常这作为方法调用的边缘效应(side effect)而产生。

一个具有public实例变量的恰当例子，是类仅作为数据结构，没有行为。亦即，若你要使用一个结构(struct)而非一个类(如果Java支持结构的话)，那么把类的实例变量声明为public是合适的。

### 5.5.2引用类变量和类方法

不要用一个对象访问一个类的静态变量和方法。要用类名替代。例如：

classMethod(); // OK

AClass.classMethod(); // OK

anObject.classMethod(); // AVOID!

### 5.5.3常量

位于for循环中作为计数器值的数字常量，除了-1，0和1之外，不应被直接写入代码。

### 5.5.4变量赋值

不要在一个语句中给多个变量赋相同的值。它很难读懂。例如：

fooBar.fChar = barFoo.lchar = ‘c’; // AVOID!

不要将赋值运算符用在容易与相等关系运算符混淆的地方。例如：

if (c++ = d++) { // AVOID! (Java disallows)

….

}

应该写成

if ((c++ = d++) ! = 0) {

…

}

不要使用内嵌(embedded)赋值运算符试图提高运行时效率，这是编译器的工作。例如：

d = (a = b +c) + r; // AOVID!

要写成

a = b + c;

d = a + r;

### 5.5.5其它惯例

#### 5.5.5.1 圆括号(Parentheses)

一般而言，在含有多种运算符的表达式中使用括号来避免运算符优先级问题，是个好方法。即便运算符的优先级对你而言可能很清楚，但对其他人未必如此。你不能假设别的程序员和你一样清楚运算符的优先级。

if (a == b && c ==d) // AVOID!

if ((a == b) && (c == d)) // RIGHT

#### 5.5.5.2返回值(Returning Values)

要让你的程序结构符合目的。例如：

if (booleanException) {

return true;

else {

return false;

}

应该代之以如下方法：

return booleanException;

类似地：

if (condition) {

return x;

}

return y;

应该写为：

return (condition ? x : y);

#### **5.5.5.3条件运算符“?”前的表达式**

如果一个包含二元运算符表达式出现在三元运算符“ ? : ”之前，那么要给表达式添上一对圆括号。例如：

(x >= 0) ? x : -x;

#### 5.5.5.4特殊注释(Special Comments)

在注释中使用XXX来标识某些方法未实现(bogus)的但可以工作的内容。用FIXME来标识某些假的和错误的内容。

#### 5.5.5.5 TODO的风格

“TODO：在这里写描述”。

#### 5.5.5.6日志记录

日志总共分五个等级，平时在调试时不要贪图方便随便定义日志的等级。要求不重要的信息定义在debug等级或info等级，较为严重的情况把日志定义为warn等级和error等级。正常情况下不要使用System.println();作为日志输出。为了查看方便，建议多增加几个日志的查看规则，以方便查看日志。但是，小心日志记录，它开销很大。

假如某个类中要使用到日志的输出或调试，统一在类的声明后面（即第一个属性的位置）加上“Private static final String TAG = “类名”;”语句，输出日志的时候便可使用Log.e(TAG,”content”)进行输出。

# 6代码检查工具Checkstyle介绍

Checkstyle是一款检查Java程序源代码样式的工具，它可以有效的帮助我们检视代码以便更好的遵循代码编写规范，特别适用于小组开发时彼此间的样式规范和统一。Checkstyle提供了高可配置性，以便适用于各种代码规范，所以除了使用它提供的几种常见标准之外，你也可以定制自己的标准。

Checkstyle的官方网站：<http://checkstyle.sourceforge.net/>。

## 6.1 Windows7系统下Eclipse + Checkstyle环境配置

**方法一：**

1. SVN的\*\*\*里有Eclipse-cs插件的源文件，把里面的两个文件夹plugins和 features下面的文件分别放到eclipse 目录下面对应的plugins和features目录下。

2. 必须在启动eclipse的时候加上一个参数 -clean. 这样eclipse就可以更新它当前安装的许多插件信息。比如你得eclipse在d盘 那么就是 d:\eclipse\eclipse.exe –clean。

3. 接下来就可以正常使用Eclipse-cs插件进行代码检查了。

**方法二：**

1. 在Eclipse中，选择Help->Software Updates->Find and Install

2. 选择 Search for new features to install 选择Next

3. 选择 New Remote Site

4. 在名称中输入 Checkstyle Plug-in，在URL中输入:

http://eclipse-cs.sourceforge.net/update

5. 然后继续安装就行了。

建议执行第一种方法，统一标准。

## 6. 标准应用

在你需要进行Checkstyle的项目上单击右键，选择properties。在弹出的对话框中选择Checkstyle项目，如果需要Checkstyle的时候就勾选上左上角的复选框，不需要的时候反之即可。在此界面你还可以选择代码规范或使用其默认的编码规范，进行代码检查。

另外，你也可以定制自己的检查标准，进行高级代码检查。

## 6.3 高级应用

通过编写自定义的编码标准，进行高级代码检查。

例如：

# 7示例代码

源代码范例如下所示

/\*

\* @(#) Blah.java 11/08/18

\*

\* Copyright (c) 2011-2012 SSM-ZTE, Lab.

\* west of friend Road, Xi’an, Shanxi, 710000 China

\* All rights reserved.

\*

\* This software is the confidential and proprietary information of Sun

\* Microsystems, Inc. ("Confidential Information"). You shall not

\* disclose such Confidential Information and shall use it only in

\* accordance with the terms of the license agreement you entered into

\* with Sun.

\*/

package android.demo.ui;

import java.util.\*;

/\*\*

\* 该类的基本功能描述

\*

\* @version 1.0 08-18-2011

\* @author Firstname Lastname

\*/

public class Blah extends SomeClass {

/\* 该类的实现细节描述 \*/

/\*\* classVar1 的javaDoc说明 \*/

public static int classVar1;

/\*\*

\* classVar2 的javaDoc说明

\* 多于一行的情况

\*/

private static Object classVar2;

/\*\* instanceVar1 的javaDoc说明 \*/

public Object instanceVar1;

/\*\* instanceVar2 的javaDoc说明 \*/

protected int instanceVar2;

/\*\* instanceVar3 的javaDoc说明 \*/

private Object[] instanceVar3;

/\*\*

\* constructor Blah的javaDoc说明

\*/

public Blah() {

// 实现细节描述

}

/\*\*

\* method doSomething 的javaDoc说明

\*/

public void doSomething() {

// 实现细节描述

}

/\*\*

\* method doSomethingElse 的javaDoc说明

\* @param someParam 参数描述

\*/

public void doSomethingElse(Object someParam) {

// ...implementation goes here...

}

}