Java&Android编码规范

目 录

前言 4

1 代码布局 5

1.1 基本格式 5

1.2 对齐 6

1.3 空行空格 7

1.4 断行 9

2 注释 12

2.1 实现注释 12

2.2 文档注释 17

3 命名规则 20

4 声明 24

5 表达式与语句 25

6 类和接口 29

7 Android 31

2. 前 言

编写本规范的目的是为了进一步规范JAVA软件编程风格，提高软件源程序的可读性、可靠性和可重用性，确保在开发成员或开发团队之间的工作可以顺利交接，不必花很大的力气便能理解已编写的代码，以便继续维护和改进以前的工作，提高软件源程序的质量和可维护性，减少软件维护成本。

本规范的内容包括：代码布局、注释、命名规则、声名、表达式与语句、类与接口等。

本规范分成规则性和建议性两种：对于规则性规范，要求所有软件开发人员严格执行；对于建议性规范，开发人员可以根据实际情况选择执行。

自本规范实施之日起，以后新编写的和修改的代码均应执行本规范。

本规范用到的术语解释如下：

**规则：**编程时必须遵守的规则。

**建议：**编程时必须加以考虑的原则。

**说明：**对相应规则的必要的解释。

**正例：**对此规则或建议给出的修正后的例子。

**反例：**对此规则或建议给出的反面的例子。

# 代码布局

代码布局的目的是显示出程序良好的逻辑结构，提高程序的准确性、连续性、可读性、可维护性。更重要的是，统一的代码布局和编程风格，有助于提高整个项目的开发质量，提高开发效率，降低开发成本。同时，对于普通程序员来说，养成良好的编程习惯有助于提高自己的编程水平，提高编程效率。因此，统一的、良好的程序布局和编程风格不仅仅是个人主观美学上的或是形式上的问题，而且涉及到产品质量，涉及到个人编程能力的提高，必须要引起重视。

# 基本格式

|  |
| --- |
| 【规则1-1-1】源代码文件（.java）的布局顺序是：包、import语句、类。 |

**正例:**

packagecom.sunsharp;

importjava.awt.peer.CanvasPeer;

import java.io.\*;

public class ClassName

{

}

|  |
| --- |
| 【规则1-1-2】遵循统一的布局顺序来书写import语句，不同类别的import语句之间用空行分隔。 |

**说明：**package语句其后可跟import语句，而且与package间隔一个空行。import包的排列顺序为java开头的包在最前面，接下来是引自外部的包，再接下来是应用程序自身的包，即import 中标准的包名要在本地的包名之前，而且按照字母顺序排列。

**正例:**

packagecom.sunsharp;

import java.util.ArrayList; //java自身的包

importjava.util.List;

importorg.apache.http.\*; //第三方的包

importcom.snusharp.common.\*; //程序自身的包

|  |
| --- |
| 【规则1-1-3】if、else、else if、for、while、do,switch等语句独占一行，执行语句不得紧跟其后。不论执行语句有多少都要加{}。 |

**说明：**这样可以防止书写失误，也易于阅读。

**正例：**

if(varible1 < varible2)

{

varible1 = varible2;

}

**反例：**

下面的代码执行语句紧跟if的条件之后，而且没有加{}，违反规则。

if (varible1 < varible2) varible1 = varible2;

|  |
| --- |
| 〖建议1-1-1〗源程序中关系较为紧密的代码应尽可能相邻。 |

**说明：**这样便于程序阅读和查找。

**正例：**

length = 10;

width = 5; // 矩形的长与宽关系较密切，放在一起。

strCaption = “Test”;

**反例：**

length = 10;

strCaption = “Test”;

width = 5;

# 对齐

|  |
| --- |
| 【规则1-2-1】一般禁止使用制表符，必须使用空格进行缩排。缩进为4个空格。 |

**说明：**对于利用Eclipse等编程工具的，可以设置TAB键为4个空格代替。消除不同编辑器对制表符处理的差异。

|  |
| --- |
| 【规则1-2-2】程序的分界符‘{’和‘}’应独占一行，‘}’同时与引用它们的语句左对齐。{ }之内的代码块使用缩进规则对齐。 |

**说明：**这样使代码便于阅读，并且方便注释。

do… while语句可以例外，while条件可与 } 在同一行。

**正例：**

voidfunction(intvar)

{

while (condition)

{

doSomething(); // 与{ }缩进4格

} //与引用它们的模块左对齐

}

**反例：**

voidfunction(intvar){

while (condition)

{

doSomething();

}

}

|  |
| --- |
| 【规则1-2-3】多维的数组如果在定义时初始化，按照数组的矩阵结构分行书写。 |

**正例：**

int[][] number=

{

{1,1, 1},

{2, 4, 8},

{3, 9, 27},

{4, 16, 64}

};

|  |
| --- |
| 【建议1-2-1】相关的赋值语句等号对齐。 |

**正例：**

width = 50;

length = 20;

height = 40;

# 空行空格

|  |
| --- |
| 【规则1-3-1】不同逻辑程序块之间要使用空行分隔。 |

**说明：**空行起着分隔程序段落的作用。适当的空行可以使程序的布局更加清晰。

下列情况应该总是使用空行：

* 一个源文件的两个片段(section)之间
* 类声明和接口声明之间
* 常量声明区域之后
* 方法声明之前
* 方法内的局部变量和方法的第一条语句之间
* 一个方法内的两个逻辑段之间，用以提高可读性

**正例：**

voiddoSomething()

{

Connection con = null; //数据库连接

booleanreturnParameter = false; //返回

//空一行

//if代码的注释

if (reconsign == null)

return false;

}

**反例：**

voiddoSomething()

{

　　　 Connection con = null; //数据库连接

booleanreturnParameter = false; //返回

//if代码的注释

if (reconsign == null)

return false;

}

|  |
| --- |
| 【规则1-3-2】一元操作符如“++”、“--”、“!”、“~”、（类型）等前后不加空格。“[]”“.”这类操作符前后不加空格。 |

**正例：**

!value

~value

++count

number[i] = 5;

box.getWidth();

|  |
| --- |
| 【规则1-3-3】多元运算符和它们的操作数之间至少需要一个空格。 |

**说明：**空格的多少根据上下文调整。

**正例：**

value = oldValue;

total + value;

number += 2;

|  |
| --- |
| 【规则1-3-4】方法名之后不要留空格。 |

**说明：**方法名后紧跟左括号‘(’。

|  |
| --- |
| 【规则1-3-5】‘(’向后紧跟，‘)’、‘,’、‘;’向前紧跟，紧跟处不留空格。‘,’之后要留空格。‘;’不是行结束符号时其后要留空格。 |

**正例：**

例子中的**凵**代表空格。

for(i**凵**=**凵**0;**凵**i**凵**<**凵**MAX\_BSC\_NUM;**凵**i++)

{

doSomething(width,**凵**height);

}

# 断行

|  |
| --- |
| 【规则1-4-1】长表达式（超过120列）要在低优先级操作符处拆分成新行，操作符放在新行之首（以便突出操作符）。拆分出的新行要进行适当的缩进，使排版整齐。 |

**说明：**

断行方法：

1. 在逗号后断行
2. 在操作符前断行
3. 在低优先级操作符处断行

对齐方法：

1. 将新行与同一级别的先前行的表达式的起始端对齐。
2. 条件表达式的续行在第一个条件处对齐。
3. for循环语句的续行在初始化条件语句处对齐。
4. 函数调用和函数声明的续行在第一个参数处对齐。
5. 赋值语句的续行应在赋值号处对齐。
6. 如果上述规则导致代码排列混乱或代码左边界少于两个缩进，可用两倍缩进替代。

下面是一些断行方法调用的示例：

**正例：**

someMethod(longExpression1,longExpression2, longExpression3,

longExpression4, longExpression5);

var = someMethod1(longExpression1,

someMethod2(longExpression2,

longExpression3));

下面是两个断行算术表达式例子，第一个是优选方法，因为断行出现在括号表达式之外，属于较高级别的断行。

**正例：**

longName1 = longName2 \* (longName3 + longName4 - longName5)

+ 4 \* longname6; //允许的断行方法

**反例:**

longName1 = longName2 \* (longName3 + longName4

-longName5) + 4 \* longname6; //应该避免的断行方法

下面是两个缩排方法的例子，第一个是传统的方式，第二个例子中如果采用传统方式缩排将导致第二行和第三行右边出现太多空白，因此，采用8个空格符替代。

//传统的缩排方法，第二行与第一行的括号对齐。

**正例：**

void someMethod(intanArg, Object anotherArg, String yetAnotherArg,

Object andStillAnother)

{

...//你代码的位置

}

//由8个空格符来替代与括号对齐的方法，以避免第二行、第三行出现太多的空格符

**正例：**

private static synchronized horkingLongMethodName(int anArg,

Object anotherArg, String yetAnotherArg,

Object andStillAnother)

{

...//你代码的位置

}

对于if语句的行封装通常使用8空格规则，因为传统的4空格缩排方式使得有些语句容易被忽略掉，使if语句体难以理解。例如：

**反例：**

//不允许使用下面的缩进方法

if ((condition1 && condition2)

|| (condition3&& condition4)

||!(condition5 && condition6))

{ //不好的缩进

doSomethingAboutIt(); //这样对齐的缩进方式很容易让阅读的人忽略掉这一行

}

**正例：**

//宜采用下面的缩进方法（分成三行的情况）

if ((condition1 && condition2)

| (condition3&& condition4)

||!(condition5 && condition6))

{|

doSomethingAboutIt();

}

//或使用下面的缩进方法（分成二行的情况）

**正例：**

if ((condition1 && condition2) || (condition3 && condition4)

||!(condition5 && condition6))

{

doSomethingAboutIt();

}

对于三重表达式，有三种方式可以对它进行换行缩排:

**正例：**

//单行的情况

alpha = (aLongBooleanExpression) ? beta : gamma;

//分成两行的情况，第二行的冒号与第一行的问号对齐。

alpha = (aLongBooleanExpression) ? beta

: gamma;

//分成三行的情况，第二行的问号和第三行的冒号都与第一行的括号对齐

alpha = (aLongBooleanExpression)

? beta

: gamma;

|  |
| --- |
| 【规则1-4-2】方法声明时，修饰符、类型与名称不允许分行书写。 |

**正例：**

public static double calculateArea(double width, double height);

**反例：**

public static double

calculateArea(double width, double height);

# 注释

注释有助于理解代码，有效的注释是指在代码的功能、意图层次上进行注释，提供有用、额外的信息，而不是代码表面意义的简单重复。

Java程序有两类注释：实现注释，文档注释。

# 实现注释

实现注释是那些在C++中见过的，使用/\*…\*/和//界定的注释。

实现注释有4种实现注释的风格：

* 块注释(BlockComments)

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。

* 单行注释(single-line Comments)

短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. \*/

……

}

* 尾端(trailing)

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。以下是一个Java代码中单行注释的例子：

if (a ==2)

{

return TRUE; / \* special case \*/

}

* 行末(end-of-line)。

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。

|  |
| --- |
| 【规则2-1-1】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 |

**正例：**

/\*

\* Here is a block comment.

\*/

**反例：**

/\*Here is a block comment.

\*/

|  |
| --- |
| 【规则2-1-2】注释符与注释内容之间要用一个空格进行分隔。 |

**正例：**

/\* 注释内容 \*/

// 注释内容

**反例：**

/\*注释内容\*/

//注释内容

|  |
| --- |
| 【规则2-1-3】注释应与其描述的代码相近，对代码的注释应放在其上方（需与其上面的代码用空行隔开）或右方（对单条语句的注释）相邻位置，不可放在下面。 |

**说明：**在使用缩写时或之前，应对缩写进行必要的说明。避免在代码行的末尾添加注释；行尾注释使代码更难阅读。不过在批注变量声明时，行尾注释是合适的；在这种情况下，将所有行尾注释在公共制表位处对齐。

**正例：**

如下书写结构比较清晰

// 获得子系统索引

subSysIndex = data.getSysIndex;

// 代码段1注释

[ 代码段1 ]

/\* 代码段2注释 \*/

[ 代码段2 ]

**反例1**：

如下例子注释与描述的代码相隔太远。

/\* 获得子系统索引 \*/

subSysIndex = subSys.getSysIndex();

**反例2**：

如下例子注释不应放在所描述的代码下面。

subSysIndex = subSys.getSysIndex();

/\* 获得子系统索引 \*/

**反例3**：

如下例子，显得代码与注释过于紧凑。

/\* 代码段1注释 \*/

[ 代码段1 ]

/\* 代码段2注释 \*/

[ 代码段2 ]

|  |
| --- |
| 【规则2-1-4】单行注释与所描述内容进行同样的缩进。 |

**说明：**这样可使程序排版整齐，并方便注释的阅读与理解。

**正例：**

如下注释结构比较清晰

intdoSomething()

{

/\* 代码段1注释 \*/

[ 代码段1 ]

/\* 代码段2注释 \*/

[ 代码段2 ]

}

**反例**：

如下例子，排版不整齐，阅读不方便；

intdoSomething()

{

/\* 代码段1注释 \*/

[ 代码段1 ]

/\* 代码段2注释 \*/

[ 代码段2 ]

}

|  |
| --- |
| 【规则2-1-5】若有多个尾端注释出现于大段代码中，它们应该具有相同的缩进。 |

**说明：**这样可使程序排版整齐，并方便注释的阅读与理解。

**正例：**

if( a==2 )

{

return true; /\* special case \*/

}

else

{

returnisPrime(a); /\* works only for odd a \*/

}

**反例**：

如下例子，排版不整齐，阅读不方便；

if( a==2 )

{

return true;/\* special case \*/

}

else

{

returnisPrime(a); /\* works only for odd a \*/

}

|  |
| --- |
| 〖规则2-1-6〗包含在{ }中代码块的结束处要加注释，便于阅读。特别是多分支、多重嵌套的条件语句或循环语句。 |

**说明：**此时注释可以用英文，方便查找对应的语句。

**正例：**

voiddoSomething()

{

if (…)

{

…

while (…)

{

…

} // end of while(…) // 指明该条while语句结束

…

} // end ofif (…) // 指明是哪条语句结束

} // end of void main（） // 指明函数的结束

|  |
| --- |
| 〖规则2-1-7〗对分支语句（条件分支、循环语句等）必须编写注释。 |

**说明：**这些语句往往是程序实现某一特殊功能的关键，对于维护人员来说，良好的注释有助于更好的理解程序，有时甚至优于看设计文档。

# 文档注释

文档注释(被称为“doc comments”)是Java独有的，并由/\*\*…\*/界定，注释内容里包含标签。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或者实现细节。文档注释从实现自由(implemtentation-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。

|  |
| --- |
| 【规则2-2-1】注释使用中文注释。与doc有关的标准英文单词标签保留。 |

**说明：**文档型注释描述了Java类(Java classes)，接口(interfaces)，构造函数(constructors)、方法(methods)和域(fields)。每一个文档注释都包含在/\*\*…\*/分隔符中，每一个类（class）、接口(interface)或成员(member)都有一个注释。这些注释应该只出现在声明(declaration)前。

|  |  |  |
| --- | --- | --- |
| 标签 | 用处 | 用途 |
| @author作者的名称 | 类、接口 | 说明特定某一段程序代码的作者。每一个作者各有一个标记。 |
| @deprecated | 类、方法 | 说明该类的应用程序编程接口 (API) 已被废弃，因此应不再使用。 |
| @exceptionname description | 方法 | 说明由方法发出的异常。一个异常采用一个标记，并要给出异常的完整类名。 |
| @paramname 参数名的描述 | 方法 | 用来说明传递给一个方法的参数，其中包括参数的类型/类和用法。每个参数各有一个标记。 |
| @return方法返回值的描述 | 方法 | 若方法有返回值，对该返回值进行说明。应说明返回值的类型/类和可能的用途。 |
| @since | 类、方法 | 例如： sinceJDK 1.1：说明自从有 JDK 1.1 以来，该项已存在了多长时间。 |
| @see 类名 | 类、接口、方法、字段 | 在文档中生成指向特定类的超文本链接。可以并且应该采用完全合法的类名。 |
| @seeClassName#memberfunctionName | 类、接口、方法、字段 | 在文档中生成指向特定方法的超文本链接。可以并且应该采用完全合法的类名。 |
| @version版本号 | 类、接口 | 说明特定一段代码的版本信息。 |

|  |
| --- |
| 【规则2-2-2】类、接口头部必须进行注释。 |

**说明：**注释必须列出：类、接口编号、名称、内容摘要等。

类编号由功能模块编号和类名两部分组成，中间用“\_”隔开，功能模块编号使用该类所在

的功能模块的编号，类名用类的名称。例如：M01\_Employee。

**正例：**

下面是类头部的中文注释：

/\*\*

\* 类编号：

\* 类名称：

\* 内容摘要：//说明主要功能。

\*@author

\*/

|  |
| --- |
| 【规则2-2-3】公共方法前面应进行文档型注释。 |

**说明：**注释必须列出：方法编号、主要功能、参数类型、输入参数、返回值、调用的前置条件和后置条件、异常说明、关键算法、可见性决策等。

**正例：**

下面是公共方法头部的注释：

/\*\*

\* 方法名称：

\* 内容摘要：＜列出主要功能、调用说明、异常说明、业务逻辑等＞

\* @param

\* @return

\* ＠throws

\*/

public String getName(String name)

{

return name;

}

|  |
| --- |
| 〖建议2-1〗Java语言中，公共的属性采用单行文档注释，对于需要比较多的声明的，可进行多行注释。 |

**说明：**如果是Javadoc注释，属性可以采用文档型注释。

**正例：**

对于public型变量的单行声明：

/\*\* classVar1 对classVar1的声明 \*/

public static int classVar1;

对于public型变量的多行声明：

/\*\*

\*classVar1 对classVar1的声明第一行

\*对classVar1的声明第二行（继续对classVar1的声明）

\*/

public static int classVar1;

对于public型变量的多行声明：

/\*\*

\*classVar1 对classVar1的声明第一行

\*对classVar1的声明第二行（继续对classVar1的声明）

\*/

public static int classVar1;

|  |
| --- |
| 〖建议2-2〗通过对方法、变量等正确的命名以及合理地组织代码结构，使代码成为自注释的。 |

**说明：**清晰准确的方法、变量的命名，可增加代码的可读性。

|  |
| --- |
| 〖建议2-3〗尽量避免在注释中使用缩写，特别是不常用缩写。 |

**说明：**在使用缩写时，应对缩写进行必要的说明。

# 命名规则

好的命名规则能极大地增加可读性和可维护性。同时，对于一个有上百个人共同完成的大项目来说，统一命名约定也是一项必不可少的内容。本章对程序中的所有标识符（包括包、变量名、控件名、常量名、方法名、类名、接口等）的命名做出约定。

三种命名规范说明：

Pascal规范：第1个字符大写，目标名中的每个单词的第1个字母也大写，比如InvoiceNumber或者PrintInvoice。其他的所有字符都小写。

Camel规范：第1个字符不大写，但目标名中的每个单词的第1个字母大写，比如，invoiceNumber。其他的所有字符都小写。

匈牙利规范：在目标名中加入表示类型的前缀，如strName。

|  |
| --- |
| 【规则3-1】标识符要采用英文单词或其组合，便于记忆和阅读，切忌使用汉语拼音来命名。 |

**说明：**标识符应当直观且可以拼读，可望文知义，避免使人产生误解。程序中的英文单词一般不要太复杂，用词应当准确。

|  |
| --- |
| 【规则3-2】标识符只能由26个英文字母，10个数字及下划线的一个子集来组成，并严格禁止使用连续的下划线。用户定义的标识符下划线不能出现在标识符的头尾，数字也不能出现在标识符的头部。 |

**说明：**这样做的目的是为了使程序易读。因为variable\_name和variable\_\_name很难区分，下划线符号‘\_’若出现在标识符头或结尾，容易与不带下划线‘\_’的标识符混淆。

|  |
| --- |
| 【规则3-3】标识符应当使用完整的英文描述，标识符的命名应当符合“min-length && max-information”原则，谨慎使用缩写。 |

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

**正例：**如下单词的缩写能够被大家认可：

temp 可缩写为tmp；

flag 可缩写为flg；

statistic 可缩写为 stat ；

increment 可缩写为inc；

message 可缩写为msg；

以下是一些常用缩写：

|  |  |
| --- | --- |
| 常用词 | 缩写 |
| Argument | arg |
| Buffer | buf |
| Clear | clr |
| Clock | clk |
| Compare | cmp |
| Configuration | cfg |
| Context | ctx |
| Delay | dly |
| Device | dev |
| Disable | dis |
| Display | disp |
| Enable | en |
| Error | err |
| Function | fnct |
| Hexadecimal | hex |
| Initialize | init |
| Mailbox | mbox |
| Manager | mgr |
| Maximum | max |
| Message | msg |
| Minimum | min |
| Multiplex | mux |
| operatingsystem | OS |
| Parameter | param |
| Previous | prev |
| Priority | prio |
| Read | rd |
| Ready | rdy |
| Register | reg |
| Schedule | sched |
| Semaphore | sem |
| Stack | stk |
| Synchronize | sync |
| Timer | tmr |
| Trigger | trig |
| Write | wr |

|  |
| --- |
| 【规则3-4】采用应用领域相关的术语来命名。 |

**说明：**软件开发人员应注意软件用户的一些约定术语，不应当随意的创造术语，这会降低软件的易用性。

|  |
| --- |
| 【规则3-5】程序中不要出现仅靠大小写区分的相似的标识符。 |

**说明：**命名时应避免采用几乎相同的名称。例如，变量名称persistentObject和persistentObjects不应当同时运用；anSqlDatabase和anSQLDatabase也不应同时使用。

|  |
| --- |
| 【规则3-6】用正确的反义词组命名具有互斥意义的变量或相反动作的函数等。 |

**说明：**下面是一些在软件中常用的反义词组。

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

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

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

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

show / hide ； send / receive ；source / destination ；cut / paste ；

up / down

|  |
| --- |
| 【规则3-7】类名和接口使用类意义完整的英文描述，每个英文单词的首字母使用大写、其余字母使用小写的大小写混合法。 |

**正例：**

OrderInformation,CustomerList

|  |
| --- |
| 【规则3-8】属性名使用意义完整的英文描述，第一个单词的字母使用小写，剩余单词首字母大写其余字母小写的大小写混合法。属性名不能与方法名相同。 |

**正例：**

private String customerName;

private int orderNumber;

|  |
| --- |
| 【规则3-9】方法名使用类意义完整的英文描述：第一个单词的字母使用小写、剩余单词首字母大写其余字母小写的大小写混合法。 |

**正例：**

privatevoid insertCustomer();

|  |
| --- |
| 【规则3-10】方法中，存取属性的方法采用set和 get方法，动作方法采用动词和动宾结构。 |

**格式**：

get + 非布尔属性名()

is + 布尔属性名()

set + 属性名()

动词()

动词 + 宾语()

**正例：**

public String getType();

publicbooleanisFinished();

public void setVisible(boolean);

public void show();

public void addKeyListener(Listener);

|  |
| --- |
| 【规则3-11】常量名使用全大写的英文描述，英文单词之间用下划线分隔开，并且使用 static final修饰。 |

**正例：**

publicstatic final int MAX\_VALUE = 1000;

|  |
| --- |
| 〖建议3-1〗非静态，非公开成员变量 添加m前缀，以便于同参数，变量相区分。 |

**正例：**

private String mCustomerName;

private int mOrderNumber;

|  |
| --- |
| 〖建议3-2〗静态成员变量 添加s前缀。 |

**正例：**

private static String sCustomerName;

private static int sOrderNumber;

|  |
| --- |
| 〖建议3-3〗常用组件类的命名以组件名加上组件类型名结尾。 |

**正例：**

Application 类型的，命名以App 结尾——MainApp

Frame 类型的，命名以Frame 结尾——TopoFrame

|  |
| --- |
| 〖建议3-4〗含有集合意义的属性命名，尽量包含其复数的意义。 |

**正例：**

customers, orderItems

# 声明

|  |
| --- |
| 【建议4-1】一行只声明一个变量。 |

**正例：**

int level;

int size;

**反例：**

int level, size;

|  |
| --- |
| 【规则4-2】一个变量有且只有一个功能，不能把一个变量用于多种用途。 |

**反例：**

voidmyMethod()

{

booleanmyFlag = false;

myFlag = isVisible(myView) //作为可见性判断标志

//使用myFlag做其它操作

……

myFlag = isEnable(myComponent) //作为可用性判断标志

}

|  |
| --- |
| 〖建议4-1〗变量声明应该只放在代码段的开始部分。最好不要到使用时才声明变量。对象类变量在函数体结束后，手工设置为null值，以利于资源的回收。 |

**正例：**

voidmyMethod()

{

intmyInt = 0; //方法块的开始

//其它语句

}

|  |
| --- |
| 〖建议4-2〗接口应当在名字后面加上able, ible或者er。 |

**正例：**

Contactable，Prompter

# 表达式与语句

表达式是语句的一部分，它们是不可分割的。表达式和语句虽然看起来比较简单，但使用时隐患比较多。本章归纳了正确使用表达式和if、for、while、switch等基本语句的一些规则与建议。

|  |
| --- |
| 【规则5-1】每一行应该只包括一个语句。 |

**说明：**复杂的语句阅读起来难于理解，并容易隐含错误。

**正例：**

argv++;

**反例：**

argv++; argc--;

if(s4 == null) s4 = new String(“Joy”);

|  |
| --- |
| 【规则5-2】在表达式中使用括号，使表达式的运算顺序更清晰。在逻辑上简单的情况下可以不加。 |

**说明：**由于将运算符的优先级与结合律熟记是比较困难的，为了防止产生歧义并提高可读性，即使不加括号时运算顺序不会改变，也应当用括号确定表达式的操作顺序。

**正例：**

if (((iYear % 4 == 0)&& (iYear % 100 != 0)) || (iYear % 400 == 0))

**反例：**

if (iYear % 4 == 0 &&iYear % 100 != 0 || iYear % 400 == 0)

|  |
| --- |
| 【规则5-3】当复合语句作为控制流程的一部分时，应该用‘{ }’把所有的复合语句括起来，即使只有一句简单语句。 |

**说明：**这样可以更方便地加入语句而不会由于忘掉加括号而引起的偶然性的错误。

正例：

int count;

...

myMethod()

{

if (condition)

{

count = 1;

}

}

|  |
| --- |
| 【规则5-4】不可将浮点变量用“==”或“!=”与任何数字比较。 |

**说明：**无论是float还是double类型的变量，都有精度限制。所以一定要避免将浮点变量用“==”或“!=”与数字比较，应该转化成“>=”或“<=”形式。

**正例：**

if ((fResult>= -EPSINON) && (fResult<= EPSINON))

**反例：**

if (fResult == 0.0) //隐含错误的比较

其中EPSINON是允许的误差（即精度）。

|  |
| --- |
| 【规则5-5】在switch语句中，每一个case分支必须使用break结尾，最后一个分支必须是default分支。 |

**说明：**避免漏掉break语句造成程序错误。有时编程逻辑需要，可以几个case语句共用一个语句块，对这种情况必须加注释进行说明。

**正例：**

switch (iMessage)

{

case SPAN\_ON:

[处理语句]

break;

case SPAN\_OFF:

[处理语句]

break;

default:

[处理语句]

break;

}

|  |
| --- |
| 〖建议5-1〗避免在“if”条件中赋值。 |

**说明：**Java编译器对boolean型变量在if条件语句中的赋值时合法的，对整型变量的赋值是不合法的。

|  |
| --- |
| 〖建议5-2〗带值的返回语句不需要用括号‘（）’。 |

**说明：**除非有时不得不用括号使返回结构更加明显。

正例：

return;

return myDisk.size();

return (size ? size : defaultSize); //用括号使返回结构更加明显。

|  |
| --- |
| 〖建议5-3〗循环嵌套次数不大于3次。 |

|  |
| --- |
| 〖建议5-4〗do while语句和while语句仅使用一个条件。 |

**说明：**保持程序简洁。如果需要判断的条件较多，建议用临时布尔变量先计算是否满足条件。

**正例：**

Boolean bCondition;

do

{

……..

bCondition = ((tAp[iPortNo].bStateAcpActivity != PASSIVE)

|| (tAp[iPortNo].bStateLacpActivity != PASSIVE))

&& (abLacpEnabled[iPortNo])

&& (abPortEenabled[iPortNo])

}while (bCondition);

|  |
| --- |
| 〖建议5-5〗在多重循环中，如果有可能，应当将最长的循环放在最内层，最短的循环放在最外层，以减少CPU跨切循环层的次数。 |

**正例：**

for (iCol = 0; iCol< 5; iCol++)

{

for (iRow = 0; iRow< 100; iRow++)

{

iSum = iSum + aiDate[iRow][iCol];

}

}

**反例：**

for (iRow = 0; iRow< 100; iRow++)

{

for (iCol = 0; iCol< 5; iCol++)

{

iSum = iSum + aiDate[iRow][iCol];

}

}

|  |
| --- |
| 〖建议5-6〗不可在for 循环体内修改循环变量，防止for 循环失去控制。 |

# 类和接口

|  |
| --- |
| 【规则6-1】类内部的代码布局顺序：类变量、属性、构造函数、方法。 |

**正例：**

class Test

{

private int exampleVar;

public string name;

public Test()

{

…;

}

public void setName(string name)

{

…;

}

private void method()

{

…

}

}

|  |
| --- |
| 〖建议6-1〗功能相关的方法放在一起。 |

**说明：**如接口中关系较紧密的的几个方法，类属性的get和set 方法，有调用关系的方法，重载的方法等有相近或相关的方法尽可能放在一起，方便阅读。

正例：

class Sample extends Object

{

int ivar1;

int ivar2;

String name;

Sample(inti,int j)

{

ivar1 = i;

ivar2 = j;

}

public void setName(String name)

{

this.name = name;

}

public String getName()

{

returnthis.name;

}

int method1 ()

{

…

method2();

}

void method2()

{

…

}

}

|  |
| --- |
| 〖建议6-5〗函数的参数个数不宜超过4个。 |

**说明：**过多的函数参数会导致性能降低。

|  |
| --- |
| 〖建议6-6〗使程序结构体现程序的目的。 |

**正例：**

return booleanExpression;

**反例：**

if (booleanExpression)

{

return true;

}

else

{

return false;

}

|  |
| --- |
| 〖建议6-2〗保证内部类定义成private，提高类的封装性。 |

|  |
| --- |
| 〖建议6-3〗嵌套内部类不能超过两层。 |

|  |
| --- |
| 〖建议6-4〗一个接口可以有多个实现类，实现类共同的变量在接口里声明。 |

# Android

Android下的开发基本可沿用Java下的编码规范，这里只是一些命令规范和日志的使用做一些补充。

|  |
| --- |
| 〖建议7-1〗res/layout下的xml文件统一用小写和下划线"\_"组合命名，并加上前缀以便区分 |

**正例：**

对话框的xml配置文件:dlg\_name.xml

|  |
| --- |
| 〖建议7-2〗layout中的id采用以下命名模式：  view缩写\_模块名称\_view的逻辑名称 |

**说明：**view的缩写详情如下

ListView： lv

RelativeView: rv

TextView: tv

ImageView: iv

ImageButton: ib

Button: btn

**正例：**

@＋id/lv\_appstore\_applist

**反例：**

@+id/ListView01

|  |
| --- |
| 〖建议7-3〗activity文中的view变量采用以下命名模式：  逻辑名称\_view缩写 |

**正例：**

ListView applistLv

|  |
| --- |
| 〖建议7-4〗res/drawable下的资源文件采用以下命名模式：  activity名称\_逻辑名称/common\_逻辑名称 |

**正例：**

main\_default.png,main\_pressed.png

|  |
| --- |
| 〖建议7-5〗strings.xml中的id采用以下命名模式：  activity名称\_功能模块名称\_逻辑名称/activity名称\_逻辑名称/common\_逻辑名称 |

**正例：**

<string name="main\_downloading">正在下载…</string>

|  |
| --- |
| 〖建议7-6〗字符串信息应统一在strings.xml中定义，调试信息除外 |

|  |
| --- |
| 〖建议7-7〗使用日志时，不重要的信息定义在debug等级或者info等级，较为严重的情况把日志定义的warn等级和error等级。正常情况下尽量不使用System.out.println();作为日志的输出 |

|  |
| --- |
| 〖建议7-8〗使用日志时，统一在类的声明后面加上：  Private static finnal String TAG = “类名”;  输出日志的时候使用Log.e(TAG, “content”)进行输出。 |