- 1 简介
  - 1.1 术语说明
  - 1.2 指南说明
- 2 源文件基础
  - 2.1 文件名
  - 2.2 文件编码：UTF-8
  - 2.3 特殊字符
    - 2.3.1 空格字符
    - 2.3.2 特殊转义序列
    - 2.3.3 非ASCII字符
- 3 源文件结构
  - 3.1 许可证或版权信息
  - 3.2 Package语句
  - 3.3 Import语句
    - 3.3.1 Import不要使用通配符
    - 3.3.2 不要换行
    - 3.3.3 顺序和间距
    - 3.3.4 非静态导入的类
  - 3.4 类声明
    - 3.4.1 只声明唯一一个顶级类
    - 3.4.2 类内容顺序
      - 3.4.2.1 重载：永不分离
- 4 格式
  - 4.1 大括号
    - 4.1.1 在需要的地方使用大括号
    - 4.1.2 非空块：K & R 风格
    - 4.1.3 空块：可以用简洁版本
  - 4.2 块缩进：2个空格
  - 4.3 一行一个语句
  - 4.4 列限制：100
  - 4.5 自动换行
    - 4.5.1 从哪里断开
    - 4.5.2 自动换行时缩进至少+4个空格
  - 4.6 空格
    - 4.6.1 垂直空格
    - 4.6.2 水平空格
    - 4.6.3 水平对齐：不做要求
  - 4.7 用小括号来限定组：推荐
  - 4.8 具体结构
    - 4.8.1 枚举类
    - 4.8.2 声明变量
      - 4.8.2.1 每次只声明一个变量
      - 4.8.2.2 需要时才声明
    - 4.8.3 数组

- 4.8.3.1 数组初始化：可写成块状结构
- 4.8.3.2 非C风格的数组声明
- 4.8.4 switch语句
- 4.8.4.1 缩进
- 4.8.4.2 Fall-through 注释
- 4.8.4.3 default的情况要写出来
- 4.8.5 注解
- 4.8.6 注释
- 4.8.6.1 块注释风格
- 4.8.7 修饰符
- 4.8.8 数值型字面值
- 5 命名
- 5.1 所有标识符共有的规则
- 5.2 标识符类型的规则
- 5.2.1 包名
- 5.2.2 类名
- 5.2.3 方法名
- 5.2.4 常量名
- 5.2.5 非常量字段
- 5.2.6 参数名
- 5.2.7 局部变量名
- 5.2.8 类型变量名
- 5.3 驼峰情况：定义
- 6 编程实践。
- 6.1 @Override：总是使用
- 6.2 捕获的异常：不能忽视
- 6.3 静态成员：使用类进行调用
- 6.4 Finalizers: 禁用
- 7 Javadoc
- 7.1 格式
- 7.1.1 一般形式
- 7.1.2 段落
- 7.1.3 块标记
- 7.2 摘要片段
- 7.3 哪里需要使用Javadoc
- 7.3.1 例外：不言自明的方法
- 7.3.2 例外：重载
- 7.3.4 可选的Javadoc

1 简介

本文是Google Java编程风格规范的完整定义。当且仅当一个Java源文件符合本文中的规范，我们才认为它符合Google Java编程风格。

和其他编程风格指南一样，这里不仅涉及编程格式的美观问题，还涉及其他类型的约定或编码标准。然而，本文主要关注的是我们普遍遵循的硬性规则，对于那些不是明确要求的，我们避免提供的意见(无论是人力还是工具)

1.1 术语说明

在本文中，除非另有说明：

1. 术语类用来表示一个普通类，枚举类，接口和注解类型(@interface)
2. 术语成员(类的)用来表示一个嵌套类、属性、方法或构造函数；即除了初始化器和注解之外的类的所有顶级内容
3. 术语注解指实现注释。不使用短语“文档注释”，而是使用通用术语“Javadoc”

其他“术语说明”将偶尔出现在本文中。

1.2 指南说明

本文中的示例代码是非标准的。也就是说，虽然示例代码是遵循Google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。

2 源文件基础

2.1 文件名

源文件以其顶层的类名来命名，大小写敏感，文件扩展名为.java。

2.2 文件编码：UTF-8

源文件以UTF-8编码

2.3 特殊字符

2.3.1 空格字符

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

1. 所有其他字符串中的空白字符都需要进行转义
2. 制表符不用于缩进

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，尽管在字符串和注释之外使用Unicode转义是非常不鼓励的。

> 提示：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。

例子：

例子	讨论
String unitAbbrev = "μs";	最好：即使没有注释也非常清晰
String unitAbbrev = "\u03bcs"; // "μs"	允许：但是没有理由要这样做
String unitAbbrev = "\u03bcs"; // Greek letter mu, "s"	允许：但是这样显得笨拙还容易出错
String unitAbbrev = "\u03bcs";	很差：读者根本看不出这是什么
return '\ufeff' + content; // byte order mark	很好：对于非打印字符，使用转义，并在必要时写上注释

> 提示：不要因为某些程序可能无法正确处理非ASCII字符而使您的代码变得不可读。当程序无法正确处理非ASSII字符时，它自然无法正确运行，你就会去fix这些问题了。

3 源文件结构

一个源文件包含，按顺序地：

1. 许可证或版权信息（如有需要）
2. package语句
3. import语句
4. 一个顶级类

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，那么它应该被放在这里

3.2 Package语句

package语句不换行。列限制（4.4 节，列限制：100）并不适用于package语句。

3.3 Import语句

3.3.1 Import不要使用通配符

不应使用通配符import，不管是静态导入还是其他。

3.3.2 不要换行

import语句不换行，列限制（4.4节，列限制：100）并不适用于import语句。

3.3.3 顺序和间距

import导入的顺序如下：

1. 所有静态导入为一组
2. 所有非静态导入为一组

如果同时存在静态导入与非静态导入，则以一个空白行分割。 import语句之间没有其他空白行。

每一个组中，导入的名称以ASCII排序显示。

3.3.4 非静态导入的类

静态导入不能用于静态嵌套类。

3.4 类声明

3.4.1 只声明唯一一个顶级类

每个顶级类都再一个与它同名的源文件中。

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

3.4.2 类内容顺序

类的成员和初始化块顺序对易读性有很大的影响。但是没有一个统一正确的标准。不同的类可以以不同的方式对其内容进行排序。

重要的是，每个类都要按照一定的逻辑规律排序。维护者应该能解释这种排序逻辑。比如，新增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/方法应该按照顺序出现在一起，中间不要放进其他代码。

4 格式

术语说明：块状结构指的时一个类，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 在需要的地方使用大括号

大括号用在if，else，for，do和while语句。甚至当它的实现为空或者只有一句话时，也要使用。

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：

- 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、方法，构造方法或者类的终止，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。

例子：

```
return () -> {
  while (condition()) {
    method();
  }
};

return new MyClass() {
  @Override public void method() {
    if (condition()) {
      try {
        something();
      } catch (ProblemException e) {
        recover();
      }
    } else if (otherCondition()) {
      somethingElse();
    } else {
      lastThing();
    }
  }
};
```

4.8.1节给出了枚举类的一些例外。

4.1.3 空块：可以用简洁版本

一个空块或者空的构造方法，可以在大括号开始之后直接结束大括号，中间不需要空格或换行。但是当一个由几个语句块联合组成的语句块时，则需要换行。（例如：if/else或try/catch/finally）

例子:

```
 // This is acceptable
  void doNothing() {}

  // This is equally acceptable
  void doNothingElse() {
  }



// This is not acceptable: No concise empty blocks in a multi-block statement
  try {
    doSomething();
  } catch (Exception e) {}
```

4.2 块缩进：2个空格

每当开始一个新的快，缩进就增加两个空格。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（见4.1.2节中的代码示例）

4.3 一行一个语句

每个语句后要换行

4.4 列限制：100

Java代码单行限制长度为100个字符，一个字符是指任何Unicode编码点。除了以下情况，超出此上限的行必须进行换行，如4.5节所解释的

> 每个Unicode编码点都算作一个字符，即使其显示宽度大于或小于。例如，如果使用全宽度字符，您可以选择比此规则严格要求的更早地换行。

例外：

1. 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）
2. package和import语句（见3.2节和3.3节）
3. 注释中那些可能被剪切并粘贴到shell中的命令行。

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而被分为多行，我们称之为自动换行。

这里并没有全面，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。

> 注：虽然换行的典型原因是为了避免溢出列限制，即使在事实上符合列限制的代码也可能由作者自行决定是否换行。

> 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

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。

1. 如果在非赋值运算符处断开，那么在该符号前断开。（注意：这一点与Google其它语言的变成风格不同，如C++和JavaScript）。
   - 这也适用于类似以下的运算符：
     - 点分隔符（.）
     - 方法引用的两个冒号
     - 类型边界中的符号(<T extends Foo & Bar>)
     - 在catch块中的一个管道(catch (FooException | BarException e))
2. 如果在赋值运算符处断开，通常的做法是在该符号后断开，但是无论哪种方式都可以接受。
   - 这也适用于增强型for("foreach")语句中的赋值运算符冒号。
3. 方法名或构造函数名与左括号留在同一行。
4. 逗号(,)与其前面的内容留在同一行。
5. 在lambda表达式中，不能断开，除了lambda的主体由单个非支持表达式组成，则可以在箭头之后断开。

```
MyLambda<String, Long, Object> lambda =
    (String label, Long value, Object obj) -> {
        ...
    };

Predicate<String> predicate = str ->
    longExpressionInvolving(str);
```

> 注意：换行的主要目标是要有清晰的代码，而不是只包含最少行数的代码。

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四个空格。

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

第4.6.3水平对齐一节中指出，不鼓励使用可变数目的空格来对齐前面的符号。

4.6 空格

4.6.1 垂直空格

以下情况需要使用一个空行：

1. 类的连续成员或初始化器之间：字段，构造函数，方法，嵌套类，静态初始化块和实例初始化块。
    - 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。
    - 例外：在4.8.1节中介绍的，枚举常量之间的空行。
2. 要满足本文档中其他节的空行要求（例如 3节，源文件结构，3.3节，import语句）

一个空行也可能出现在任何可以提高可读性的地方，例如在语句之间将代码组织成逻辑子部分。在第一个成员或初始化器之前或在类的最后一个成员或初始化器之后的空白行既不鼓励也不劝阻

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）

4.6.2 水平空格

除了语言或其他样式规则需要的地方之外，除了文字、注释和Javadoc用到单个空格之外，下面的地方也只显示一个ASCII空格。

1. 分隔任何保留字与紧随其后的左括号(()(如if, for , catch)。
2. 分隔任何保留字与其前面的右大括号(})(如else, catch)。
3. 在任何左大括号前({)，两个例外：
    - @SomeAnnotation({a, b})(不使用空格)
    - String[][] x = {{foo}};(大括号间没有空格)
4. 在任何二元或三元运算符两侧。这也适用于以下"类运算符"符号：
    - 类型界限中的&：<T extends Foo & Bar>
    - catch块中的管道符号：catch (FooException | BarException)
    - foreach语句中的分号
    - lambda表达式中的箭头：(String str) -> str.length() 不能有空格的情况
    - 一个方法引用的两个冒号(::)，它写作Object::toString
    - 点分隔符(.)，它写作object.toString()
5. 在,;;及右括号())后
6. 如果在一条语句后做注释，则双斜杠(//)两边都要空格。这里可以允许多个空格，但没有必要。
7. 类型和变量之间：List<String> list
8. 数组初始化中，大括号内的空格是可选的。
    - new int[] {5, 6}和new int[] { 5, 6 }都是可以的。
9. 在类型注释和[]之间

这个规则并不要求或禁止一行的开始或结束处需要额外的空格，只对内部空格做要求。

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Google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
```

> 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分格），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。

4.7 用小括号来限定组：推荐

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。

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。

```
private enum Answer {
  YES {
    @Override public String toString() {
      return "yes";
    }
  },

  NO,
  MAYBE
}
```

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，因此所有适用于其他类的格式规则也适用于枚举类。

4.8.2 声明变量

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

例外：在循环头上，可以声明多个变量。

4.8.2.2 需要时才声明

不要在一个代码块的开头把局部变量一次性都声明了，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，下面的写法都是可以的：

```
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 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(或者，最后一个语句组，后面跟着0条或多条语句)

4.8.4.1 缩进

和其它块状结构一致，switch块中的内容缩进为2个空格。

switch标签后，出现换行，缩进级别增加+2，就像打开了一个块。下面的switch标签返回到前一个缩进级别，就像一个块已经关闭一样

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);
}
```

注意，在case1: 之后不需要注释，只在语句组的末尾加上注释。

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。

例外：如果枚举类型的switch语句包含该类型的所有可能值的显式用例，那么它可能会省略默认语句组。这使得ide或其他静态分析工具在遗漏任何案例时能够发出警告

4.8.5 注解

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

本节讨论实现注释。Javadoc在第7节Javadoc中单独讲解。

任何换行符之前可以有任意空格，然后是实现注释。这样的注释使该行非空白。

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。以下示例注释都是可以的。

```
/*
 * 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，如果您想要自动的代码格式器在必要时重新换行（即注释像段落风格一样），那么请使用/.../。大多数格式化程序不会使// ...风格的注释块重新换行。

4.8.7 修饰符

类和成员的修饰符，如果存在，则按Java语言规范中推荐的顺序出现。

```
public protected private abstract default static final transient volatile synchronized native
strictfp
```

4.8.8 数值型字面值

long整型的字面值使用大写L作为后缀，从不小写（以避免与数字1混淆）。例如，3000000000L 而不是3000000000l

5 命名

5.1 所有标识符共有的规则

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Google其它编程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。例如，com.example.deepspace，而不是com.example.deepSpace 或者 com.example.deep_space

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，例如，Character 或 ImmutableList。接口名称有时可能是名词或名词短语（例如：List），有时又是形容词或形容词短语（例如Readable）。

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。例如，sengMessage或stop。

下划线可能出现在JUnit测试方法名称中用以分割名称的逻辑组件，每个组件使用lowerCamelCase编写。一个典型的模式时：<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，用下划线分割单词。那到底什么算是一个常量？

每个常量都有一个静态final字段，其内容是不可变的，并且其方法没有可检测的副作用。这包括基元，字符串，不可变类型和不可变类型的不可变集合。如果任何实例的observable状态可以改变，它不是一个常量。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视为常量。

```
// Constants
static final int NUMBER = 5;
static final ImmutableList<String> NAMES = ImmutableList.of("Ed", "Ann");
static final ImmutableMap<String, Integer> AGES = ImmutableMap.of("Ed", 35, "Ann", 32);
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 ImmutableMap<String, SomeMutableType> mutableValues =
    ImmutableMap.of("Ed", mutableInstance, "Ann", mutableInstance2);
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

非常量字段以lowerCamelCase风格编写

这些名字通常是名词或者名词短语。例如，computedValues 或 index

5.2.6 参数名

参数名以lowerCamelCase风格编写。

在公共方法中参数应该避免使用单个字符名明

5.2.7 局部变量名

局部变量名以lowerCamelCase风格编写。

即使局部变量是final和不可变的，也不应该把它视为常量，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 驼峰情况：定义

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"）。Google指定了以下的转换方案。

名字从散文形式开始：

1. 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"
2. 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"）。需要注意的时"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。
3. 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：
    - 每个单词的第一个字母都大写，来得到大驼峰式命名。
    - 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。
4. 最后将所有的单词连接起来得到一个标识符。

注意，原词的大小写几乎被完全忽略了。例如：

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*	

*可接受，但不推荐

> 注意：在英语中，某些带有连字符的单词形式不唯一：例如"nonempty"和"non-emtpy"都是正确的，所以方法名称checkNonempty和chechNonEmpty也是正确的。

6 编程实践。

6.1 @Override：总是使用

只要是合法的，就把@Override注解给用上。这包括重写超类方法的类方法，实现接口方法的类方法和重定义超接口方法的接口方法。

例外：当父方法未@Deprecated时，可以省略@Override

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，则把它当做一个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

使用类名调用类的静态成员，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 Javadoc

7.1 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. */
```

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

7.1.2 段落

空行会出现在段落之间和和Javadoc标记之前（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

，并且它和第一个单词间没有空格。

7.1.3 块标记

标准的块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。当描述无法再一行中容纳，连续行需要至少缩进4个空格。

7.2 摘要片段

每个Javadoc块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。

这是一个小片段，可以是一个名字短语或动词短语，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，以下是一些列外：

还可能存在其他Javadoc内容，如第7.3.4节"非必需Javadoc"中所述。

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。在这种情况下除了写"Returns the foo"，确实也没什么值得写了。

> 重要：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并非必须的。

7.3.4 可选的Javadoc

其他类和成员根据需要和期望使用Javadoc

每当实现注释用于定义类或成员的总体目的或行为时，该注释将改为Javadoc。

非必需的Javadoc不是严格要求遵守第7.1.2，7.1.3和7.2节的格式化规则，但当然是建议。