java开发规范

1. 前言

编写本规范的目的在于提高软件开发的规范性，提高程序代码可读性、可维护性、健壮性、可移植性、重用性等，以达到提高软件质量及开发效率的

这份文档是项目组内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，我们才认为它符合项目组要求的Java编程风格。

1.1 基本要求

1. 首要要求是它必须正确，能够按照程序员的真实思想去运行；
2. 第二个的要求是代码必须清晰易懂，使别的程序员能够容易理解代码所进行的实际工作；
3. 第三个要求是保持源程序的风格统一。

1.2 术语说明

在本文档中，除非另有说明：

1. 术语 class 可表示一个普通类，枚举类，接口或是annotation类型(`@interface`)
2. 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。

1.3 参考资料

1. Google Java Style
2. Sun's Coding Conventions
3. The Elements of Java Style
4. 代码检测工具的规则： checkstyle，pmd，findbugs

2. 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

源文件编码格式统一为：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：

1. 所有其它字符串中的空白字符都要进行转义。
2. 制表符(Tab) 不能 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。

> 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：

非ASCII字符

```
String unitAbbrev = "s";                                  //
String unitAbbrev = "\u03bcs"; // "s"                     //
String unitAbbrev = "\u03bcs"; // Greek letter mu, "s"     //
String unitAbbrev = "\u03bcs";                             //
return '\ufeff' + content; // byte order mark             // Good
```

> 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

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(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。

3.2 package语句

package语句 不换行，列限制(4.4 列限制：80或100)并不适用于package语句。(即package语句写在一行里，就算它很长。)

3.3 import语句

3.3.1 import不要使用通配符

即，不管是静态导入还是变通导入，不要出现类似这样的import语句：

- `import java.util.*`;
- `import static java.lang.Math.*`;

3.3.2 不要换行

import语句不换行，列限制(4.4 列限制：80或100)并不适用于import语句。(每个import语句独立成行，就算它很长。)

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：

1. 所有的静态导入 static imports 独立成组
2. `com.yy` imports (仅当这个源文件是在 `com.yy` 包下)
3. 第三方的包。每个顶级包为一组，字典序。
    - 例如： `android` , `com` , `junit` , `org` , `sun`
4. `java` imports
5. `javax` imports

组内不空行，按字典序排列。

ImportOrder.java

```
package com.yy.ent.spec;

import static java.lang.Math.PI;
import static org.junit.Assert.fail;

import com.yy.ent.clients.daemon.DaemonService;
import com.yy.ent.commons.base.http.HttpUtils;

import android.app.Application;
import android.content.Context;

import com.google.common.collect.Lists;

import org.apache.thrift.transport.TServerSocket;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Value;

import java.net.InetSocketAddress;
import java.util.List;

import javax.annotation.Resource;

public class ImportOrder {
    // codes here
}
```

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。

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对

3.4.2.1 修饰符顺序

类和成员的修饰符(modifiers)如果存在，则按Java语言规范中推荐的顺序出现：

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

3.4.2.2 类声明顺序

1. 静态成员变量 Static Fields
2. 静态初始化块 Static Initializers
3. 成员变量 Fields
4. 初始化块 Initializers
5. 构造器 Constructors
6. 静态成员方法 Static Methods
7. 成员方法 Methods
8. 重载自Object的方法如 toString(), hashCode() 和 main 方法
9. 类型(内部类) Types(Inner Classes)

同等的类型，按3.4.2 类成员顺序里描述的原则进行，不管 `public, protected, private` 的顺序如何。

3.4.2.3 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。

4. 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

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

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();
      }
    }
  }
};
```

4.1.3 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：4个空格

每当开始一个新的块，缩进增加4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。

> Note: 在IDE中，使用Tab时，需设置Tab键输出的是4个空格！

4.3 一行一个语句

每个语句后要换行。

4.4 列限制：120或150

一个项目可以选择一行120个字符或150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：

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。
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(100或120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。

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。也就是说：

1. 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)`）。
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（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。
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。

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。

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

4.6 空白

4.6.1 垂直空白

以下情况需要使用一个空行：

1. 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。
2. 在函数体内，语句的逻辑分组间使用空行。
3. 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）。
4. 要满足本文档中其他节的空行要求（比如3.3 import语句）

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，单个ASCII空格也出现在以下几个地方：

1. 分隔任何保留字与紧随其后的左括号（`(`）（如`if`, `for`, `catch`等）。
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，见下面的Note）。
4. 在任何二元或三元运算符的两侧。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 e)`）。
    - `foreach`语句中的分号。
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 list`。
8. 数组初始化中，大括号内的空格是可选的
    - 即`new int[] {5, 6}`和`new int[] { 5, 6 }`都是可以的。

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。

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 4 个空格。

每个 switch 标签后新起一行，再缩进 4 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

```
@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。

> Note： 下面的小节里将对 `/* ... */` 风格和 `// ...` 风格的使用场景进行了具体的说明。

4.8.6.2 注释基本原则

- 注释中的第一个句子要以（英文）句号、问号或者感叹号结束。Javadoc生成工具会将注释中的第一个句子放在方法汇总表和索引中。
- 为了在JavaDoc和IDE中能快速链接跳转到相关联的类与方法，尽量多的使用 `@see xxx.MyClass，@see xx.MyClass#find(String)` 。
- Class必须以 `@author` 作者名声明作者，不需要声明 `@version` 与 `@date` ，由版本管理系统保留此信息。
- 如果注释中有超过一个段落，用<p>分隔。
- 示例代码以 `<pre></pre>` 包裹。
- 标识(java keyword, class/method/field/argument名，Constants) 以 `<code></code>` 包裹。
- 标识在第一次出现时以 `{@linkxxx.Myclass}` 注解以便JavaDoc与IDE中可以链接。

4.8.6.3 失效代码注释

由 `/.../` 界定。专用于注释已失效的代码。

```
/*
 * Comment out the code
 * String s = "hello";
 * System.out.println(s);
 */
```

4.8.6.4 代码细节注释

由 `//` 界定，专用于注释代码细节，即使有多行注释也仍然使用 `//` ，以便与用 `/**/` 注释的失效代码分开

除了私有变量外，不推荐使用行末注释。

```
class MyClass {
    private int myField; // An end-line comment.
public void myMethod {
       //a very very long
//comment.
if (condition1) {
          //condition1 comment
...
        } else {
          //elses condition comment
...
        }
    }
}
```

4.8.6.5 可精简的注释内容

注释中的每一个单词都要有其不可缺少的意义，注释里不写" `@param name -名字` "这样的废话。

如果该注释是废话，连同标签删掉它，而不是自动生成一堆空的标签，如空的 `@param name` ，空的 `@return` 。

4.8.6.6 推荐的注释内容

- 对于API函数如果存在契约，必须写明它的前置条件(precondition)，后置条件(postcondition)，及不变式(invariant)。
- 对于调用复杂的API尽量提供代码示例。
- 对于已知的Bug需要声明。
- 在本函数中抛出的unchecked exception尽量用 `@throws` 说明。

4.8.6.7 Null规约

如果方法允许Null作为参数，或者允许返回值为Null，必须在JavaDoc中说明。

如果没有说明，方法的调用者不允许使用Null作为参数，并认为返回值是Null Safe的。

```
/**
 * .
 *
 * @ return the object to found or null if not found.
 */
Object get(Integer id){
    ...
}
```

4.8.6.8 特殊代码注释

- 代码质量不好但能正常运行，或者还没有实现的代码用 `//TODO:` 或 `//XXX:` 声明
- 存在错误隐患的代码用 `//FIXME:` 声明

4.8.7 Long型后缀

`long` 型的数字直接量，使用大写的L为后缀，决不能采用小写的l作为后缀。比如： `3000000000L` 是正确的， `3000000000l` 却是不允许的。

5. 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。

不允许使用汉语拼音命名。

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；尽量使用单个单词。

比如： `com.example.deepspace` ，而不是 `com.example.deepSpace` 或 `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，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` 。

interface名可以是一个名词或形容词(加上 `'able','ible', or 'er'` 后缀)，如 `Runnable, Accessible` 。

为了基于接口编程，不采用 首字母为 `I` 或加上 `IF` 后缀的命名方式，如 `IBookDao,BookDaoIF` 。

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。

那，到底什么算是一个常量？

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

```
// 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。

局部变量及输入参数尽可能地不要与类成员变量同名(get/set方法与构造函数除外)。

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")。这里指定了以下的转换方案。

名字从散文形式(prose form)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。
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。

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*	

加星号处表示可以，但不推荐。

遇到缩写如XML时，仅首字母大写，即 `loadXmlDocument()` 而不是 `loadXMLDocument()` 。

> 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

6. 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，可避免父类方法改变时导致重载函数失效。

例外： 当父方法声明为 `@Deprecated` 时， `@Override` 声明变得无效。

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AssertionE

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
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。

6.5 静态工具类

隐藏工具类的构造器，确保只有static方法和变量的类不能被构造实例。

6.6 使用接口进行变量类型

变量，参数和返回值定义尽量基于接口而不是具体实现类，如 `Map map = new HashMap()` ，而不是 `HashMap map = new HashMap()` ；

6.7 不使用控制台输出信息

代码中不能使用 `System.out.println()，e.printStackTrace()` ，必须使用 `logger` 打印信息

6.8 代码度量

6.8.1 耦合度度量

- DAC度量值不要不大于7
    - 解释： DAC(Data Abstraction Coupling)数据抽象耦合度是描述对象之间的耦合度的一种代码度量。DAC度量值表示一个类中有实例化的其它类的个数。
- CFO度量值不要不大于20
    - 解释： CFO(Class Fan Out)类扇出是描述类之间的耦合度的一种代码度量。CFO度量值表示一个类依赖的其他类的个数。

6.8.2 方法度量

- 方法（构造器）参数在5个以内
    - 太多的方法（构造器）参数影响代码可读性。考虑用值对象代替这些参数或重新设计。
- 方法长度150行以内
- CC 度量值不大于10
    - 解释：CC(CyclomaticComplexity)圈复杂度指一个方法的独立路径的数量，可以用一个方法内if,while,do,for,catch,switch,case,?:语
- NPath度量值不大于200
    - 解释： NPath度量值表示一个方法内可能的执行路径的条数。

6.8.3 其他度量

- 布尔表达式中的布尔运算符(&&,||)的个数不超过3个
- if语句的嵌套层数3层以内
- 文件长度2000行以内
- 匿名内部类20行以内

太长的匿名内部类影响代码可读性，建议重构为命名的（普通）内部类。

6.9 @SuppressWarnings：去除多余warning

不需要关心的warning信息用@SuppressWarnings("unused"), @SuppressWarnings("unchecked"), @SuppressWarnings("serial") 注释。

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。

8. 附录

8.1 intellij IDEA的配置

intellij IDEA中的codestyles配置文件：intellij-java-yyent-style.xml

- 复制设置的XML文件INTELLIJ_SETTINGS_DIR/config/codestyles（其中INTELLIJ_SETTINGS_DIR是包含您的IntelliJ设置的文件夹，通常它

  在我的情况的完整路径是“%Userprofile%\.IntelliJIdea14\config\codestyles”）
- 重启IntelliJ
- 进入“File” - “Settings” - “Code Style”，选择“使用全局设置”，然后从下拉框中选择以前导入的样式（YYEntStyle）
- 可选只有通过点击“复制到项目”，并选择“使用每个项目设置”事后应用样式到一个项目中

8.2 eclipse的配置

eclipse中的codestyles配置文件：eclipse-java-yyent-style.xml

- 打开Window->Preferences->Java->Code Style->Code Templates
- 点击"Import"，导入模板xml文件