AM ESD JAVA 开发规范

拟制	季虎	日期	10 年 4 月 7 日
评审人	杨昕吉，严波，曾光荣	日期	
批准		日期	
签发		日期	

ESD JAVA 开发规范

1 开发规范及约束

具体代码规范见后续，本章节将只进行引用，不再重复其内容，本规范将侧重于上述规范中没有明确规定的其他重要原则或建议，已更适用于本项目的规范开发和管理。

1.1 基本原则

1、 必须遵循公司《软件编程规范总则》。

2、 必须遵循《Java 语言代码规范》。

3、 必须遵循公司、产品线及技术框架组规定的其他相关开发规范。

1.2 工程路径规范

遵循 Maven 对于各类 Java 工程路径的约定

1.2.1 web 工程路径

1.3 版本控制规范

1、 未通过编译的代码不准上传至版本库。

2、 自己没有通过简单验证（单元测试）或无法运行的代码不准上传至版本库。

3、 建议使用 findbugs 工具进行自检后再上传至版本库。

4、 IDE 中有警告标识（黄色提示）的代码不建议上传至版本库，请消除后再上传。

5、 每天开发的代码应及时上传到版本库，以保证可持续集成特性。

6、 一些公共的配置如需修改应及时上传到版本库，否则请自己保证版本合并。

7、 上传时如遇版本冲突问题，请自行合并，合并时请仔细检查已防止覆盖别人的修改。

8、 建议对于公共类和公共接口，如果有更新请发通知通知相关项目组，避免大量重复功能的开发以及代码不同步造成的集成测试或日建造失败。

9、 转测试后的代码修改，需要明确填写 checkin 的注释，并且注释的格式也要统一例如:

问题单号为：xxxxx

产品名称为：AM *****

路标版本为：V100R001

基线版本-子版本为：V100R001C02

问题简述：C ：功能测试：OT-COUT-COM-NLSM-09-01-C002：静态工作模式，注销后重新注册用户，服务器端用户被删掉

1.4 命名及编码规范

1.4.1 Java 命名规范

1.4.1.1 Java 文件命名规范

1、 Java 接口名称按普通类格式命名。

接口命名 AccountManager.java

2、 测试类名命名规则为：

单元测试：被测试类类名+Test.java

系统测试：被测试类类名+ST.java

类 AccountManagerImpl.java

单元测试类 AccountManagerImplTest.java

自动测试类　　AccountManagerST.java

3、 业务类接口文件命名规则为：***Manager.java，实现类命名规则为：***ManagerImpl.java。

业务接口类　　AccountManager.java

业务接口实现类　　AccountManagerImpl.java

4、 Naming-sql 对应的 DAO 对象：*DAO.java，例如：

DAO 对象　　ProjectDAO.java

5、 域模型： *.java，不做特殊命名，例如：

DAO 对象　　Project.java

6、 Action 类命名规则： *Action.java，名字较长，则用动词+名词前缀：

Action 对象　　CreateProjectAction.java

1.4.1.2 Java 方法命名规范

为了方便统一进行的声明式事务配置和可扩展特性，基本的 CRUD 建议命名规范为：create/update/delete/get/query。其中 get 用于获取指定对象和查询条件固定的情况；而 query 用于查询条件不固定的情况。对于批量操作的情况，最好在方法名后加上 List 以示区别。

1.4.1.3 Java 变量命名规范

Java 变量命名应该按照编程规范，首字母小写，其余部分采用驼峰命名法，并且使用能够明确表示其意义的名称。

专有对象变量建议使用其类名第一字母小写的方式命名，如：

业务对象变量 MenuService 可以命名为 menuService，而不要命名为 menu

对于数组类型的变量，建议以 s 为后缀；集合类型的变量后面加上 List、Set、Map 后缀。

```
MenuBean[] menus = new MenuBean[10];
List menuList = new ArrayList();
Set menuSet = new HashSet();
Map menuMap = new HashMap();
```

1.4.2 JSP 编码规范

1.4.2.1 JSP 文件命名规范

JSP 文件名采用**名词+动词**的词组命名，文件名的首字母小写，后面的单词首字母大写，后缀名小写。使用名字加动词而不是使用动词加名词的目的主要是方便分类和查找，比如人员操作信息的页面会排列在一起。

列表页面	menuList.jsp
新增页面	menuCreate.jsp
修改页面	menuUpdate.jsp
详细页面	menuDetail.jsp

1.4.2.2 JSP 编码规范

1、采用和 java 代码一样的规范要求，包括注释和排版风格。

2、应保持 JSP 页面的整洁，不随意嵌入 java 脚本。

3、页面模板（jsp）应仅负责对 action 传递的数据模型进行展示，不应主动调用业务对象。

1.4.3 JavaScript 编码规范

1.4.3.1 JavaScript 文件命名规范

JavaScript 文件命名按照编程规范，首字母小写，其余部分采用驼峰命名法，并且使用能够明确表示其意义的名称。

1.4.3.2 JavaScript 变量命名规范

因为 JavaScript 是弱类型的语言，变量声明、方法的参数、返回值都没有类型声明。如果 JavaScript 规模很大时，非常容易遇到一个变量不能立即反应出它的类型。因此对于 JavaScript 变量名采用 Hungarian Type Notation 命名法，变量名需加上类型缩写的前缀。

Type	Prefix	Example
Array	a	aValues
Boolean	b	bFound
Float	f	fValue
Function	fn	fnMethod
Integer	i	iValue
Object	o	oType
Regular Expression	re	rePattern
String	s	sValue
Variant	v	vValue

1.4.3.3 JavaScript 编码规范

1、Javascript 代码采用和 java 代码一样的规范要求，包括注释和排版风格。

2、建议不要使用 javascript 来生成大量的页面标签（HTML），javascript 最好只用于控制页面标签的状态和属性。使用 ajax 的请特别注意。

3、建议不仅在客户端使用 javascript 进行业务逻辑验证，应考虑移植到服务端验证框架或业务逻辑中实现。

4、控制 ajax 的使用范围

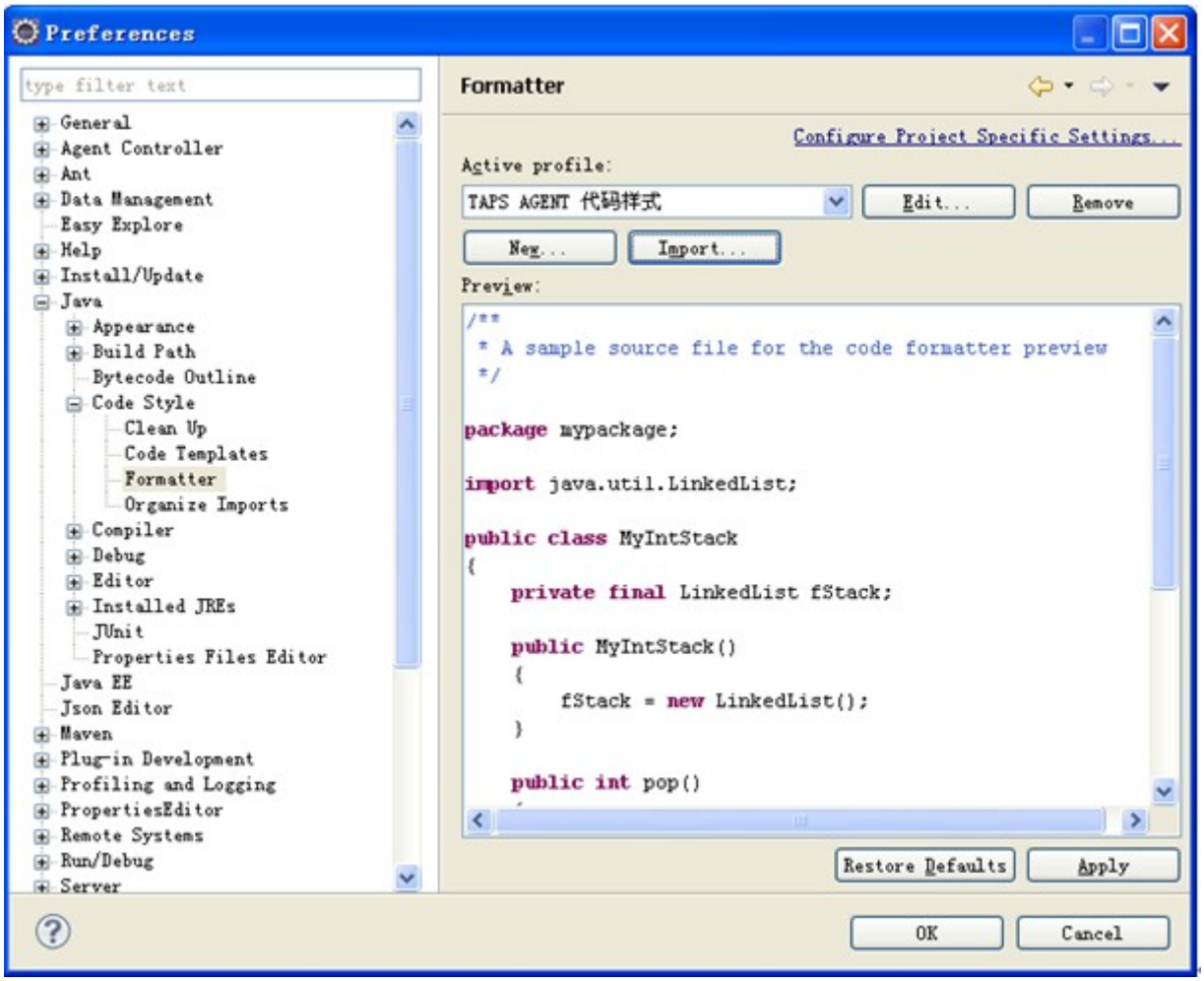
[说明]：能使用普通 form 提交方式简单解决的问题就最好不要使用 ajax。在没有一些基础框架的支持下大量使用 ajax 只会增加客户端 js 和页面的混乱，增加调试和维护的成本。ajax 是一种简单的技术，直接的后果是容易被滥用。建议轻量级的使用 AJAX，就是那种交互简单，数据较少的操作。

1.4.4 源代码风格和注释

为了适应公司的代码编程规范，并进一步统一 Java 代码的格式，需导入如下模板以统一代码样式。 统一提供基于 Eclipse 的样式配置

1.4.4.1 导入 Formatter.xml

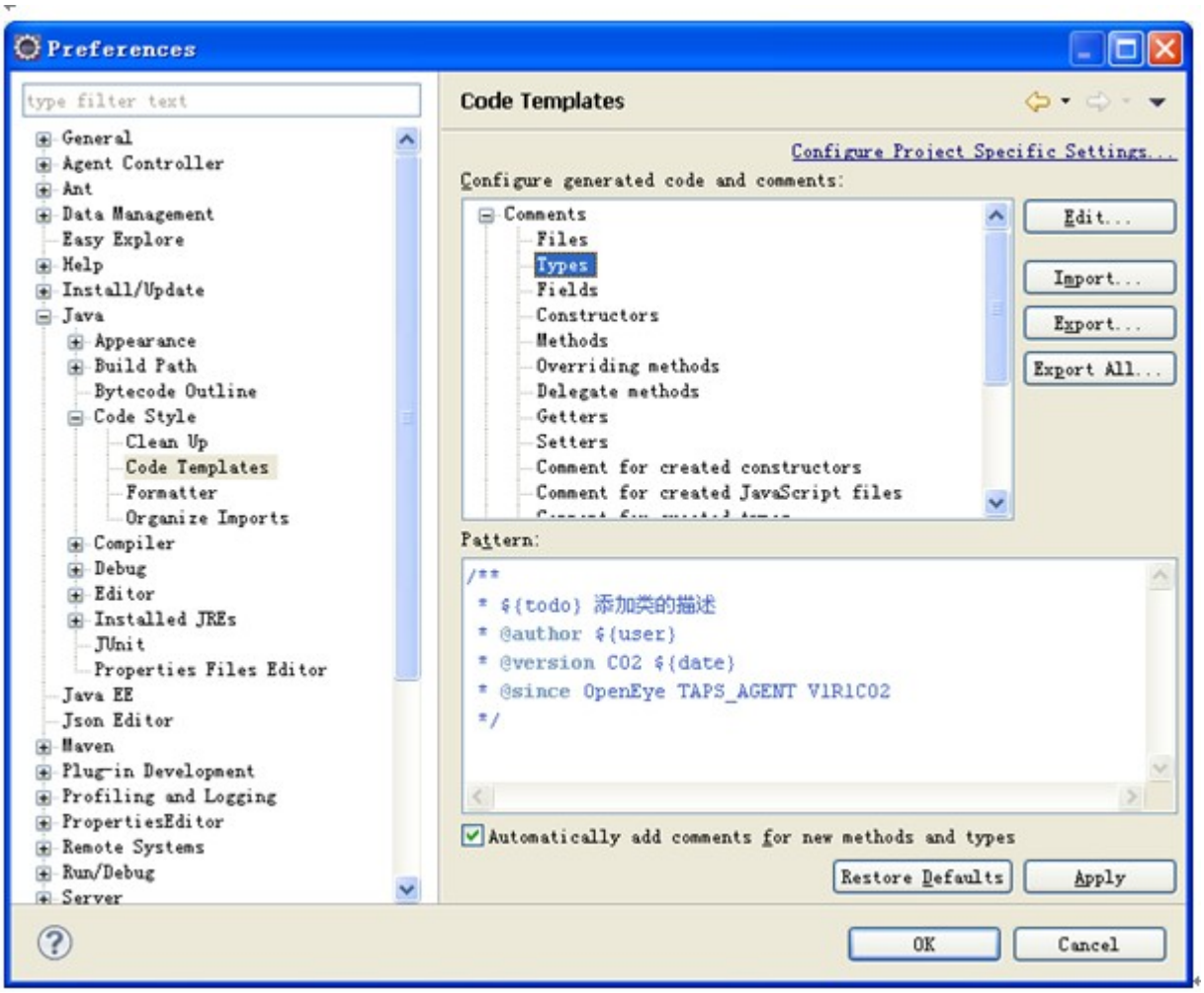
进入 Eclipse 菜单 Windows->Preferences->Java->Code Style->Formattter，点击 Import 按钮导入 Formatter.xml。



附件：Formatter.xml

Formatter.XML

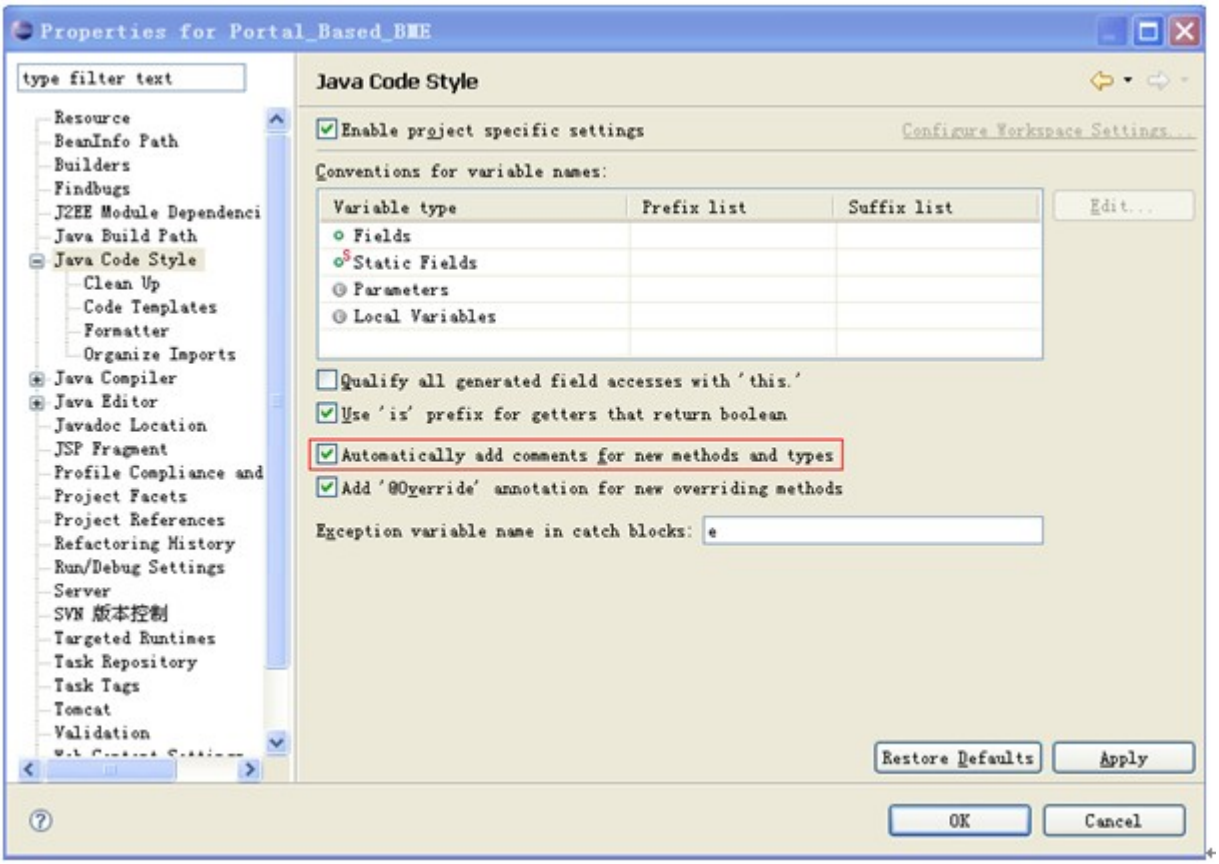
1.4.4.2 导入 Code Templates.xml



附件：Code Templates.xml

Code Templates.xml

在 Eclipse3.2 版本中新增类文件时，在向导页面必须要勾选如下选项，否则生成的代码中不会出现定义好的注释。



1.4.5 源代码编码约定

所有源代码编码约定为 UTF-8 。

1.5 注释规范

1.5.1 注释范围及内容要求

- 所有文件头必须有文件注释，注释必须包含**作者姓名、版权说明、功能描述、修改记录。**
- 所有类、接口必须有注释，注释必须包含**作者姓名、类功能描述、修改记录。**
- 类、接口中所有属性成员必须有注释，注释必须包含**属性描述、取值说明。**
- 所有函数必须有注释，注释必须包含**作者姓名、函数功能描述、参数和返回值含义描述、其他特殊约束的描述、修改记录。**
- 对于关键的代码块、关键处理逻辑必须添加注释说明各代码块的功能及注意事项。

1.5.2 注释率

有效注释率要求 40%以上。

JAVA 代码规范

2 介绍

2.1 为什么要有编码规范

编码规范对于程序员而言尤为重要，有以下几个原因：

- 一个软件的生命周期中，80%的花费在于维护
- 几乎没有任何一个软件，在其整个生命周期中，均由最初的开发人员来维护
- 编码规范可以改善软件的可读性，可以让程序员尽快而彻底地理解新的代码
- 如果你将源码作为产品发布，就需要确任它是否被很好的打包并且清晰无误，一如你已构建的其它任何产品

为了执行规范，每个软件开发人员必须一致遵守编码规范。每个人、所有人。

3 文件名

这部分列出了常用的文件名及其后缀。

3.1 文件后缀

Java 程序使用下列文件后缀：

文件类别	文件后缀
Java 源文件	.java
Java 字节码文件	.class

3.2 常用文件名

常用的文件名包括：

文件名	用途
Maven pom **文件**（pom.xml）	Maven 的默认工程文件

Ant build 文件（build.xml)	Ant 的编译文件
README	概述特定目录下所含内容的文件的首选文件名

4 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"提供了一个布局合理的 Java 程序范例。

4.1 Java 源文件

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一个源文件。公共类必须是这个文件中的第一个类或接口。

Java 源文件还遵循以下规则：

- 开头注释（参见"开头注释"）
- 包和引入语句（参见"包和引入语句"）
- 类和接口声明（参见"类和接口声明"）

4.1.1 开头注释 (Beginning Comments)

所有的源文件都应该在开头有一个 C 语言风格的注释，其中列出类名、版本信息、日期和版权声明：

```
/*
 * Classname
 *
 * Version information
 *
 * Date
 *
 * Copyright notice
 */
```

4.1.2 包和引入语句(Package and Import Statements)

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

```
package java.awt;

import java.awt.peer.CanvasPeer;
```

4.1.3 类和接口声明(Class and Interface Declarations)

下表描述了类和接口声明的各个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(/**……*/)	该注释中所需包含的信息，参见"文档注释"
2	类或接口的声明	
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，而这些信息又不适合作为类/接口文档注释。
4	类的(静态)变量	首先是类的公共变量，随后是保护变量，再后是包一级别的变量(没有访问修饰符，access modifier)，最后是私有变量。
5	实例变量	首先是公共级别的，随后是保护级别的，再后是包一级别的(没有访问修饰符)，最后是私有级别的。
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。例如，一个私有的类方法可以置于两个公有的实例方法之间。其目的是为了更便于阅读和理解代码。

5 缩进排版(Indentation)

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)。

5.1 行长度(Line Length)

尽量避免一行的长度超过 80 个字符，因为很多终端和工具不能很好处理之。

注意：用于文档中的例子应该使用更短的行长，长度一般不超过 70 个字符。

5.2 换行(Wrapping Lines)

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(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。

以下是断开方法调用的一些例子：

```
someMethod(longExpression1, longExpression2, longExpression3,
          longExpression4, longExpression5);

var = someMethod1(longExpression1,
                  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 - longName5)
           + 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，所以代之以缩进 8 个空格

```
//CONVENTIONAL 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)会使语句体看起来比较费劲。比如：

```
//DON'T USE THIS INDENTATION
if ((condition1 && condition2)
   || (condition3 && condition4)
   ||!(condition5 && condition6)) { //BAD WRAPS
   doSomethingAboutIt();             //MAKE THIS LINE 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)) {
  doSomethingAboutIt();
}
```

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;
```

6 注释(Comments)

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。

实现注释用以注释代码或者实现细节。文档注释从实现自由(implementation-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。

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。

注释不应写在用星号或其他字符画出来的大框里。注释不应包括诸如制表符和回退符之类的特殊字符。

6.1 实现注释的格式(Implementation Comment Formats)

程序可以有 4 种实现注释的风格：块(block)、单行(single-line)、尾端(trailing)和行末(end-of-line)。

6.1.1 块注释(Block Comments)

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.
 */
```

块注释可以以/*-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
 */
```

注意：如果你不使用 indent(1)，就不必在代码中使用/*-，或为他人可能对你的代码运行 indent(1)作让步。

参见"文档注释"

6.1.2 单行注释(Single-Line Comments)

短注释可以显示在一行内，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，就该采用块注释(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. */
 ...
}
```

6.1.3 尾端注释(Trailing Comments)

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。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 */
}
```

6.1.4 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，它可以用来注释掉连续多行的代码段。以下是所有三种风格的例子：

```
if (foo > 1) {

  // Do a double-flip.
  ...
```

```
}
else {
  return false;          // Explain why here.
}

//if (bar > 1) {
//
//    // Do a triple-flip.
//    ...
//}
//else {
//    return false;
//}
```

6.2 文档注释(Documentation Comments)

文档注释描述 Java 的类、接口、构造器，方法，以及字段(field)。每个文档注释都会被置于注释定界符/**...*/之中，一个注释对应一个类、接口或成员。该注释应位于声明之前：

```
/**
 * The Example class provides ...
 */
public class Example { ...
```

注意顶层(top-level)的类和接口是不缩进的，而其成员是缩进的。描述类和接口的文档注释的第一行(/**)不需缩进；随后的文档注释每行都缩进 1 格(使星号纵向对齐)。成员，包括构造函数在内，其文档注释的第一行缩进 4 格，随后每行都缩进 5 格。

若你想给出有关类、接口、变量或方法的信息，而这些信息又不适合写在文档中，则可使用实现块注释(见 5.1.1)或紧跟在声明后面的单行注释(见 5.1.2)。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。

文档注释不能放在一个方法或构造器的定义块中，因为 Java 会将位于文档注释之后的第一个声明与其相关联。

7 声明(Declarations)

7.1 每行声明变量的数量(Number Per Line)

规定一行一个声明，因为这样以利于写注释。亦即，

```
int level;  // indentation level
int size;   // size of table
```

避免

```
int level, size;
```

不要将不同类型变量的声明放在同一行，例如：

```
int foo,  fooarray[];   //WRONG!
```

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

```
int           level;         // indentation level
int           size;          // size of table
Object        currentEntry;  // currently selected table entry
```

7.2 初始化(Initialization)

尽量在声明局部变量的同时初始化。唯一不这么做的理由是变量的初始值依赖于某些先前发生的计算。

7.3 布局(Placement)

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。）不要在首次用到该变量时才声明之。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;         // beginning of method block

   if (condition) {
     int int2 = 0;     // beginning of "if" block
     ...
   }
 }
```

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!
     ...
   }
   ...
}
```

7.4 类和接口的声明(Class and Interface Declarations)

当编写类和接口是，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

8 语句(Statements)

8.1 简单语句(Simple Statements)

每行至多包含一条语句，例如：

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

8.2 复合语句(Compound Statements)

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。

8.3 返回语句(return Statements)

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为显见。例如：

```
 return;

return myDisk.size();

return (size ? size : defaultSize);
```

8.4 if，if-else，if else-if else 语句

if-else 语句应该具有如下格式：

```
 if (condition) {
   statements;
}
```

```
if (condition) {
   statements;
} else {
   statements;
}

if (condition) {
   statements;
} else if (condition) {
   statements;
} else{
   statements;
}
```

注意：if 语句总是用"{"和"}"括起来，避免使用如下容易引起错误的格式：

```
 if (condition) //AVOID! THIS OMITS THE BRACES {}!
   statement;
```

8.5 for 语句

一个 for 语句应该具有如下格式：

```
 for (initialization; condition; update) {
   statements;
}
```

一个空的 for 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。

8.6 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);
```

8.7 do-while 语句

一个 do-while 语句应该具有如下格式：

```
 do {
   statements;
} while (condition);
```

8.8 switch 语句

一个 switch 语句应该具有如下格式：

```
 switch (condition) {
case ABC:
   statements;
   /* falls through */
case DEF:
   statements;
   break;

case XYZ:
   statements;
   break;

default:
   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 */。

8.9 try-catch 语句

一个 try-catch 语句应该具有如下格式：

```
 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;
}
```

9 空白

9.1 空行

空行将逻辑相关的代码段分隔开，以提高可读性。

下列情况应该总是使用两个空行：

- 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（参见"5.1.1"）或单行注释（参见"5.1.2"）之前
- 一个方法内的两个逻辑段之间，用以提高可读性

9.2 空格

下列情况应该使用空格：

- 一个紧跟着括号的关键字应该被空格分开，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因该加空格，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);
```

10 命名规范(Naming Conventions)

命名规范使程序更易读，从而更易于理解。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，例如，不论它是一个常量，包，还是类。

标识符类型	命名规则	例子
包 (Packages)	一个唯一包名的前缀总是全部小写的 ASCII 字母并且是一个顶级域名，通常是 com，edu，gov，mil，net，org，或 1981 年 ISO 3166 标准所指定的标识国家的英文双字符代码。包名的后续部分	com.sun.eng com.apple.quicktime.v2 edu.cmu.cs.bovik.cheese

	根据不同机构各自内部的命名规范而不尽相同。这类命名规范可能以特定目录名的组成来区分部门(department)，项目(project)，机器(machine)，或注册名(login names)。	
类(Classes)	命名规则：类名是个一名词，采用大小写混合的方式，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(除非该缩写词被更广泛使用，像 URL，HTML)	class Raster; class ImageSprite;
接口(Interfaces)	命名规则：大小写规则与类名相似	interface RasterDelegate; interface Storing;

方法(Methods)	方法名是一个动词，采用大小写混合的方式，第一个单词的首字母小写，其后单词的首字母大写。	run();
		runFast();
		getBackground();

变量(Variables)	除了变量名外，所有实例，包括类，类常量，均采用大小写混合的方式，第一个单词的首字母小写，其后单词的首字母大写。变量名不应以下划线或美元符号开头，尽管这在语法上是允许的。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，除非是一次性的临时变量。临时变量通常被取名为 i，j，k，m 和 n，它们一般用于整型；c，d，e，它们一般用于字符型。	char c; int i; float myWidth;
实例变量(Instance Variables)	大小写规则和变量名相似，除了前面需要一个下划线	int _employeeId; String _name; Customer _customer;

常量(Constants)	类常量和 ANSI 常量的声明，应该全部大写，单词间用下划线隔开。(尽量避免 ANSI 常量，容易引起错误)	static final int MIN_WIDTH = 4;
		static final int MAX_WIDTH = 999;
		static final int GET_THE_CPU = 1;

11 编程惯例(Programming Practices)

11.1 提供对实例以及类变量的访问控制

若没有足够理由，不要把实例或类变量声明为公有。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。

一个具有公有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公有是合适的。

11.2 引用类变量和类方法

避免用一个对象访问一个类的静态变量和方法。应该用类名替代。例如：

```
classMethod();              //OK
AClass.classMethod();       //OK
anObject.classMethod();     //AVOID!
```

11.3 常量(Constants)

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。

11.4 变量赋值(Variable Assignments)

避免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的效率，这是编译器的工作。例如：

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

应该写成

```
a = b + c;
d = a + r;
```

11.5 其它惯例(Miscellaneous Practices)

11.5.1 圆括号(Parentheses)

一般而言，在含有多种运算符的表达式中使用圆括号来避免运算符优先级问题，是个好方法。即使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
```

11.5.2 返回值(Returning Values)

设法让你的程序结构符合目的。例如：

```
if (booleanExpression) {
```

```
    return true;
  } else {
    return false;
  }
```

应该代之以如下方法：

```
return booleanExpression;
```

类似地：

```
  if (condition) {
    return x;
  }
  return y;
```

应该写做：

```
return (condition ? x : y);
```

11.5.3 条件运算符"?"前的表达式

如果一个包含二元运算符的表达式出现在三元运算符" ？ : "的"?"之前，那么应该给表达式添上一对圆括号。例如：

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

11.5.4 特殊注释(Special Comments)

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