**java编程规范手册**

## 文件

### 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 |
| Java资源束 | .properties |

java相关其他文件经常使用如下文件后缀

| 文件类别 | 文件后缀 |
| --- | --- |
| JSP源文件 | .jsp |
| XML文档 | .xml |
| XSLT文档 | .xsl |
| XML文档类型定义 | .dtd |
| XML Schema定义 | .xsd |
| HTML文档 | .html  .htm |

### 常见文件名

常用的文件名包括：

|  |  |
| --- | --- |
| 文件名 | 用途 |
| build.xml | Apache-Ant build脚本的首选文件名 |
| build.properties | Apache-Ant build 环境配置的文件名 |
| README | 概述特定目录下所含内容的文件的首选文件名 |
| pom.xml | Maven 构件工程的属性文件 |

## 工程命名

所属项目名称和单一名词组合而成，中间以“-”分割，如：framework-core、pay-web、pay-service。

## 目录结构

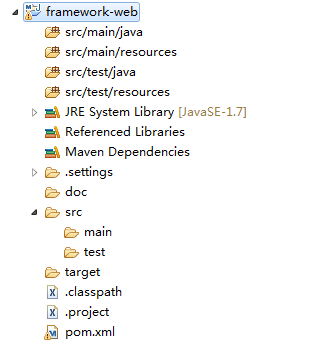
### 编写目的

本章描述了工程框架的目录结构，用于框架搭建时和编程过程中目录结构和命名的规范。

### 框架结构分层

采用maven构建的标准目录结构，目录下存在doc目录，用于存放开发者记录工作笔记；src目录，用于存放源代码和相关配置；以及target用于存放编译后相关文件。

以web程序为例结构如下图：



1. main文件夹存放项目业务代码

* java为业务源程序目录，包名规范详见本章第5节；
* resources为工程配置文件目录；
* webapp为页面元素文件目录。

1. test文件夹存放单元测试文件

* java为单元测试源程序目录；
* resources为单元测试配置文件目录；

1. resources配置文件目录结构

* property为属性文件的配置；
* sqlmap为ibatis、mybatis等mapper配置文件目录；
* xml为spring等配置文件；
* .vm目录为autoconfig模版文件目录；

### 逻辑结构分层

我们约定包结构分层为一般为两种

1. 组织标识.产品标识|系统标识.模块标识

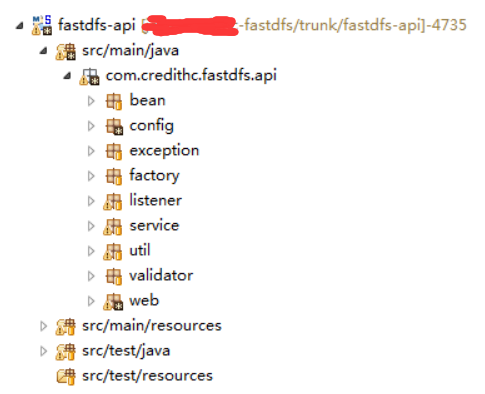
如：com.\*.fastdfs.api，其中com.\*为组织标识，fastdfs为产品标识，api为模块标识

1. 组织标识.产品标识|系统标识.业务分层标识.模块标识

如：com.\*.pay.protocol.chinapnr，其中com.\*为组织标识，pay为产品标识，protocol为业务标识

注：框架类型以此格式为准：组织标识.framework.core，如com.\*.framework.core

一般来说包名应以小写字符或者单词组成，第一种适用于大部分系统；第二种适用于web管理系统，框架部分与业务部分分层的应用。业务简单的工程，可以只按照逻辑分层，如图：



业务相对比较复杂的工程中，一般先按照业务分层（可多级）再进行逻辑分层。

## 命名规范

### 编写目的

本章描述了JAVA开发中的有关包、类、接口、方法、实例变量、变量和常量的命名规则，用于规范JAVA编程过程中的命名和代码书写规范。

### 规范细则

在以下章节提到的命名都应遵循示意的基本原则。

### Package 包的命名

开发组将基于JAVA开发中产生的包分为两类，一是与各业务系统相关的包，一是与业务系统无关的、可公用的包。它们的命名规则除要遵守“包名应全部是小写字母，包名中不能出现下划线，并且第一个字母不能是数字”的规则。

|  |
| --- |
| package com.\*.framework.web |

### Class、Interface类、接口的命名

类和接口的名称应是一个名词，采用大小写混和的方式，禁止使用拼音进行命名，所有单词都应紧靠在一起，其中每个单词的首字母应大写，接口名称应以大写I开始，接口的实现要以Impl结尾。

|  |
| --- |
| class ClassName { }  interface IClassDemo{ }  public class ClassOneImpl implements IClassDemo{ } |

### Methods方法的命名

方法名应是一个动词或动名结构，采用大小写混和的方式，禁止使用拼音进行命名，其中第一个单词的首字母用小写，其后单词的首字母大写。

|  |
| --- |
| public ClassName getClassName () {  return null;  } |

每个方法前必须加说明包括：参数说明、返回值说明、异常说明。如果方法名实在是太长可以对变量名缩写，但是必须添加相应的说明。

注：获取单个方法用get做前缀。

获取多个对象的方法用list做前缀

获取统计值的方法用count做前缀

插入的方法用save（推荐）或insert做前缀

删除的方法用remove（逻辑删除方法命名）或delete（数据删除）做前缀

修改的方法用update做前缀

**领域命名规则**

数据对象：xxxDO，xxx即为数据表名。

数据传输对象：xxxDTO，xxx为业务领域相关的名称。

展示对象：xxxVO，xxx一般为网页名称。

POJO是DO/DTO/BO/VO的统称，禁止命名成xxxPOJO

### Variables变量的命名

变量，包括所有局部变量，实例变量，类变量，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，一般用于整型；c，d，e，一般用于字符型。

|  |
| --- |
| （1）类的实例对象定义如下：  Person person;  （2）同一个类的多个对象可以采用一下定义方式：  Person  person1；  Person  person2；  （3）集合类的实例命名使用集合包含元素的英文名称的复数表示，例如：  Vector  persons;  （4）如果变量名实在是太长可以对变量名缩写，但是必须在类说明或方法说明部分（视缩写的范围而定）进行说明。  （5）数组的声明要用"int[] packets"的形式，而不要用"int  packets[]"。 |

### Constants常量的命名

常量定义为static final，类常量声明，只能包含字母、数字和下划线，并且应该全部大写，第一个字符必须是字母，单词间用下划线隔开。

|  |
| --- |
| public static final String DB\_CONFIG\_FILE\_PATH = "property/datasource-oracle.properties";  public static final int SERVER\_PORT = 8080; |

### 变量定义规范

1. 去掉没必要的公共变量。

2. 构造仅有一个模块或函数可以修改、创建，而其余有关模块或函数只访问的公共变量，防止多个不同模块或函数都可以修改、创建同一公共变量的现象。

3. 仔细定义并明确公共变量的含义、作用、取值范围及公共变量间的关系。

4. 明确公共变量与操作此公共变量的函数或过程的关系，如访问、修改及创建等。

5. 当向公共变量传递数据时，要十分小心，防止赋于不合理的值或越界等现象发生。

6. 防止局部变量与公共变量同名。

7. 仔细设计结构中元素的布局与排列顺序，使结构容易理解、节省占用空间，并减少引起误用现象。

8. 结构的设计要尽量考虑向前兼容和以后的版本升级，并为某些未来可能的应用保 留余地（如预留一些空间等）。

9. 留心具体语言及编译器处理不同数据类型的原则及有关细节。

10. 严禁使用未经初始化的变量。声明变量的同时对变量进行初始化。

11. 编程时，要注意数据类型的强制转换。

#### 配置文件命名

##### properties文件

1. 文件名

所有的properties文件均以单词加“-”的格式命名。注意文件的命名应该遵循（尽量将经常修改的配置归类在一起） ：

文件分类-文件意义.properties

文件分类：

* 数据源配置: datasource-数据库类型
* 消息配置：notice-消息类型
* 系统配置：system
* 监控配置：monitor
* 日志配置：log
* 文件配置：file-文件类型
* 请求地址配置：url
* 路径配置：path
* 数据库相关配置：db
* 阈值配置：[threshold](app:ds:threshold" \t "_self)
* 内容格式配置：message

注：第三方插件或者依赖的其他配置文件如果有固定格式不可修改沿用插件固定格式如：quartz. properties等。

文件意义：

如果前两级已经可以标识文件意义，此级可以省略，如果不能释义，需要使用此级关系。文件意义应该是简短的英文单词或缩写字母。

|  |
| --- |
| datasource-oracle.properties  notice-mail.properties |

一般来说文件名中字母均使用小写格式，最多不得超出4个单词（除变动标识）。

1. Key值

Key值为单词、小写字母、数字的形式命名，单词间用“.”分割，最多不得使用4个以上单词。Key值的名为应该为：

属性分类.意义，除特殊规定外不允许出现无属性分类的情况。

|  |
| --- |
| datasource.url=xxxxxxxxxxxx  datasource.username=xxxx  datasource.password=xxxxx  或者  ftp.path=XXXXXXXX |

1. 注释

注释用“#”来实现，注释必须在每行配置代码之上，注释不可进行转码，编码采用UTF-8字符集。

##### xml文件

1. 组成部分

* xml声明语句
* DOCTYPE 声明语句
* 处理指令

(<?处理指令是指包括在一对学有尖括号的问号的内容?>)

* 元素
* 注释
* CDATA区

(注:以上几个部分可以不必全部出现)

1. 编写规范

* 标签不允许交叉
* 最外层(根标记)只能有一个,不允许有和它并列的标记

1. 标记命名规则

可以用数字,字母及其它一些可见字符,但是

* 不能以字母或下划线开头;
* 中间不能有空格.
* 不能以字符组合xml(或xML或XML等)打头.
* 大小写敏感.
* 名称中间不能有冒号.

1. 命名

* 不要使用".",因为在很多程序中,"."用于引用对象属性.
* 最好不要使用减号"-",而是使用下划线"\_"代替,以免与表达式中的"-"运算符发生冲突.
* 名称尽量简短,以免xml文件过大.
* 名称的大小写最好统一,即要大写都大写,要小写都小写.
* 名称可以使用非英文字符,如中文,但是有些软件可能不支持除了英文以外的字符,所以在使用中文时要注意这点.

1. 属性

* 一个标签可以有一个或多个属性,每个属性都有 它自己的名称和取值.
* 属性值一定要用双引号或单引号引起来.
* 属性名称遵循和标签一样的命名规范.
* 属性其实可以被改写成用子标签来描述同样的信息.

1. 注释

* 注释格式:<!-- 注释-->
* xml声明之前不能有注释
* 注释必须在每行配置代码之上。

##### 其它文件

遵循以上原则

### SQL编码规范

Sql编码需遵循如下原则：

1. 所有SQL关键字小写，表名、字段名、序列名需全部大写。
2. 序列变量名定义，主键字段名 + ” \_SEQ”，变量值为“主键字段名”
3. insert，select操作必须指定相应的字段\*不允许出现。
4. SQL中需使用变量绑定。
5. 使用exists 代替 in, not exists代替not in
6. 尽可能减少对表的查询次数。
7. 多表查询时，选择记录少的表(或者交叉表)作为基础表，放在from子句的最后面。
8. 多表查询时，过滤掉最大记录数的条件放在where子句的最后面。

### 字符集规范

统一采用UTF-8，包括jsp、java文件。

### 日志规范

1. 不允许出现System.out.println(……)的形式。
2. 日志采用slf4j作为日志的接口。
3. 推荐采用logback异步输出作为最终的日志输出。
4. 日志按照级别高低分为DEBUG、INFO、WARN、ERROR、FATAL，在实际使用中使用前四种级别的输出。系统上线时应该被配置成只记录生产中的错误（ERROR），以避免过度的系统开销和过大日志的生成。在开发阶段可以使用调试级别（DEBUG）。

**日志使用示例参考如下：**

1. Slf4j的引入：

|  |
| --- |
| import org.slf4j.Logger;  import org.slf4j.LoggerFactory; |

1. 普通日志输出：

|  |
| --- |
| logger.info("......开始重新初始化数据库表缓存......"); |

1. 参数型日志输出：

|  |
| --- |
| logger.info("......初始化缓存【{}】，按照配置信息创建缓存成功。", config.getCacheName());  logger.error("......初始化缓存【{}】失败：{}", config.getCacheName(), e.getMessage()); |

## 注释

### 编写目的

本章描述了JAVA开发中的有关注释的规则与规范。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己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。

源代码中强制要求添加注释的位置有：类、接口、方法、构造方法、成员变量。并且注意注释的书写风格。

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。

### 规范细则

#### 文件的注释

|  |
| --- |
| /\*\*  \* <br>项目名: pay-service  \* <br>文件名: PayConfig.java  \* <br>Copyright 2015 恒昌互联网运营中心  \*/ |

#### 类的注释

|  |
| --- |
| /\*\*  \*  \* <br>类 名: ApplicationFactory  \* <br>描 述: spring加载配置文件  \* <br>作 者: xxx  \* <br>创 建： 2015年5月5日  \* <br>版 本：v1.0.0  \* <br>  \* <br>历 史: (版本) 作者 时间 注释  \*/ |

#### 方法的注释

|  |
| --- |
| /\*\*  \* 描 述：  \* 作 者：xxx  \* 历 史: (版本) 作者 时间 注释  \* **@param** date  \* **@return**  \*/ |

#### 单行注释

|  |
| --- |
| if (condition) {  /\* 注释 \*/  ...  } |

#### 尾端和行末注释

|  |
| --- |
| if (a == 2) {  return TRUE; /\* special case \*/  } else {  return isPrime(a); /\* works only for odd a \*/  } |

注意注释的缩进。

#### Map类型的注释

|  |
| --- |
| Map<String /\*consumerId\*/, String /\*updateTime\*/> |

Map的key、value为String类型的，可以加上注释，方便理解含义。

#### Getter/Setter注释

|  |
| --- |
| **public** **class** MessageRes {  /\*\* 处理状态 \*/  **private** String status;  /\*\* 处理结果信息 \*/  **private** String message;  /\*\*  \* 获取 处理状态  \*  \* **@return** status  \*/  **public** String getStatus() {  **return** status;  }  /\*\*  \* 设置 处理状态  \*  \* **@param** status  \*/  **public** **void** setStatus(String status) {  **this**.status = status;  }  /\*\*  \* 获取 处理结果信息  \*  \* **@return** message  \*/  **public** String getMessage() {  **return** message;  }  /\*\*  \* 设置 处理结果信息  \*  \* **@param** message  \*/  **public** **void** setMessage(String message) {  **this**.message = message;  }  } |

在每个成员变量、get、set方法上用/\*\* XXX \*/格式写清楚注释，这样别人在看代码时可以很方便知道该字段的含义。如果有状态值的，也可以在注释上一并写清楚，例如：

/\*\* 启用状态 0:未启用 1:启用\*/

## 编程惯例

### 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（get），通常这作为方法调用的边缘效应副作用的结果 （side effect:The change in the value of a variable within a routine resulting from assignments outside the routine.美国传统双解词典）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。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。

### 引用静态变量和静态方法

避免通过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! |

### 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。

|  |
| --- |
| for (int i=0;i<max;i++){}//这样是可以的  for (int i=3;i<max;i++){}//不要这样写，应该写成：  int start = 3;  for (int i=start; i<max; i++){} |

### 变量赋值

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; |

### 其它惯例

#### 圆括号

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 |

#### 返回值

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：

|  |
| --- |
| if (condition) {  return x;  }else{  return y;  } |

不推荐写作：return (condition ? x : y);

#### 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; |

#### 特殊注释

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。

### Import包规范

#### import语句中最好不包含'\*'字符。

例如：com.\*.framework.core.\*

#### 删除没有使用的导入语句。

#### 避免多余的导入语句。

例如：重复的导入；导入java.lang 中的类；导入同一个包中的类。建议使用Eclipse中的“Organize Imports”功能，或其他风格化工具帮助管理import。

### 长度规范

#### 单个源文件的长度不超过2000行。

建议将超过2000行的类，重新分析结构，拆分为结构松散、耦合性低、复用率高、体积小巧、功能单一的一组类。

#### 每行代码的长度不超过80个字符。

#### 方法和构造方法的长度不超过150行。

建议将超过150行的方法重构为功能更加单一的几个方法。

### 修饰关键字

#### 修饰符的顺序。

正确的顺序为：public，protected，private，abstract，static，final，transient，volatile， synchronized，native，strictfp。Eclipse (3.1)默认不检查修饰符的顺序，请使用checkstyle检查，或使用风格化工具纠正顺序。

#### 删除多余的关键字。

例如：

a、JLS强烈建议不要在接口的方法名前使用“public”或“abstract”修饰符。

b、接口变量自动为public, static , final的，不用再使用一次。

c、定义为Final 的类，其中的方法自动为final 的，不用再使用一次。

### 代码块区域

#### 源代码中不应有“多余的、嵌套的语句块”。

比如，在代码中随意嵌套的代码块，和不必要的嵌套。例如：

|  |
| --- |
| public void guessTheOutput() {  int whichIsWich = 0;  {  int whichIsWhich = 2;  }  System.out.println("value = " + whichIsWhich);  } |

#### 源代码中不应有“空的代码块区”

#### 左花括号和右花括号的放置遵循以下风格：

|  |
| --- |
| if (condition) {  i = 0;  } else {  i = 1;  } |

而不是：

|  |
| --- |
| if (condition)  {  i = 0;  }  else  {  i = 1;  } |

### 一般的编码问题

#### 不使用行内条件语句

例如:

|  |
| --- |
| String a = getParameter("a");  String b = (a==null || a.length<1) ? null : a.substring(1); |

这样的代码有些时候不容易理解, 也容易导致很难发现的bug。尽可能用独立的条件语句来控制逻辑。

#### 为了避免同步时的运行开销，程序应当避免“双重检测锁定（DCL）”。

一个使用DCL的例子如下：

|  |
| --- |
| public class MySingleton {  private static MySingleton theInstance = null;  private MySingleton() {  }  public MySingleton getInstance() {  if (theInstance == null) { // synchronize only if necessary  synchronized (MySingleton.class) {  if (theInstance == null) {  theInstance = new MySingleton();  }  }  }  return theInstance;  }  } |

实现单例推荐使用以下方式：

|  |
| --- |
| **private** **static** **class** SingletonHolder {  **static** **final** MySingleton ***instance*** = **new** MySingleton();  }  **public** **static** MySingleton getInstance() {  **return** SingletonHolder.***instance***;  } |

#### 源代码中不应有空语句(也就是单独的分号";")

#### 如果覆盖了equals()方法的类，同时必须覆盖了hashCode()方法。

推荐使用eclipse插件快速生成toString()，hashCode()，compareTo()，equals()。

#### 尽量使用基本类型的封装类的工厂方法实例化类。

一个简单的例子是 java.lang.Boolean类。为了节约内存和CPU资源，该类一般选用预定义的常量 TRUE和FALSE。应当使用Boolean.valueOf()方法替代建构器方法。

#### 避免行内赋值。

检查子表达式，例如在String s=Integer.toString(i=2)当中的赋值。基本原理：除了for循环以外，所有的赋值都应该发生在自己的顶层声明那来增加可读性，在上面例子当中的内部赋值当中，很难看出来该变量所有放置的位置.

#### switch语句必须有一个默认的default子句

#### 多余抛出的异常。

例如：Throws语句当中声明重复的异常、非受检查的异常、或者一个已经声明的异常的子类.

方法显示的抛出自定义异常时，不可再抛出此异常以外的异常。

#### 简化过于复杂的boolean表达式

目前发现的代码比如 if (b == true), b || true, !false, 等等.

基本原理:复杂的boolean逻辑表达式使程序难于理解和维护。

#### 简化过于复杂的boolean返回语句。

比如下面的代码

|  |
| --- |
| if (valid()){  return false;  }else{  return true;  } |

应该写为

|  |
| --- |
| return !valid(); |

#### 关于String的拼接。

推荐使用String.format来进行拼接，容易解读。比如：

|  |
| --- |
| //下面的%s对应参数str； %i对应参数int  String str = string.format(“字符串：%s，数字：%i”, str, int); |

而不是：

|  |
| --- |
| String s = “字符串：”+str+”，数字：”+int; |

非必要时也减少使用：

|  |
| --- |
| StringBuilder/Buffer sb = new StringBuilder/Buffer();  sb.append(“字符串：”);  sb.append(str);  sb.append(“，数字：”);  sb.append(int);  String str = sb.toString(); |

#### 关于计时器的使用。

推荐使用Spring的工具类：org.springframework.util.StopWatch

|  |
| --- |
| StopWatch stopWatch = **new** StopWatch();  stopWatch.start("task01");  /\*\*处理业务\*/  stopWatch.stop();  stopWatch.start("task02");  /\*\*处理业务\*/  stopWatch.stop();  System.***out***.println(stopWatch.prettyPrint()); |

## 类设计问题

### 工具类（类仅仅包含静态的方法）定义为final类，私有其构造函数。

另外，建议命名以“Util”或“Utils”结尾。

基本原理：实例化应用工具类是没有意义的事情，因此构建方法应该要么是私有（private）的，要么是保护（protected－－如果你想它是可继承的）的，一个通常犯的错误就是忘记隐藏默认的构建方法。

### 最小化类成员的可视性。

仅仅static的final类型的成员应该为public类型的；其他的类成员必须为private类型。除非这个类仅仅代表一个数据结构。

## 其他规范

### 数组类型变量的声明。

要使用JAVA风格的声明：int[] someArray; 避免使用C风格的声明：int someArray[];

### 源代码中避免使用

e.printStacktrace(), System.out.println(), System.exit()等语句。

### 集合建议使用

Collection.toArray(Object a[])和Arrays.asList(String [] a)，进行数组和集合之间的转换。

### 实现java.io.Serializable接口的类，必须定义serialVersionUID（Eclipse可以生成）。

例如private static final long serialVersionUID = -3949507908848305518L;

如果此类中的成员变量如果是另一类，那么这个类也必须实现java.io.Serializable接口或者定义为transient。