JAVA 编码规范指南

二〇二〇年五月二十五日

目录

第1章　命名规范

1.1　目的

让项目中所有的文档都看起来像一个人写的，增加可读性，减少项目组中因为换人而带来的损失所有的命名使用英文单词或常用的缩写，而不要使用拼音

1.2　概述

JAVA 命名的总的规则是这样，类和接口名称以及非开头单词的首字母用大写，其它字母用小写。

基本原则：

- 用全英文单词描述
- 采用相应领域的术语来参与命名
- 采用大小写混合的方式提高可读性
- 避免命名长度超过 15 个字母
- 避免多个命名之间十分相似，仅大小写不同
- 避免下划线命名

命名项	命名规范	举例
Arguments/ parameters	采用全英文描述。为了同变量名称进行区别，可以用“a”和“an”作为前缀	customer, account, 或者 aCustomer, anAccount
Fields/ properties	用全英文描述，首字母小写，非开头单词的首字母大写	firstName, lastName, warpSpeed
取 boolean 对象的成员函数	用 is 作名称的前缀	isPersistent(), isString, isCharacter(),
类 classes	用全英文描述，所有单词的首字母大写	Customer, SavingsAccount

源文件 compilation unit file	和主 class 的名称相同	Customer.java, SavingsAccount.java, Singleton.java
组件（components/widgets）	用全英文来描述组件的用途，结尾的单词表示组件的类型	OkBotton, CustomerList, FileMenu,
构造函数 constructors	与类的命名相同	Customer(), SavingsAccount()
析构函数 destructors	一般 JAVA 没有析构函数，通常在系统的垃圾回收之前，调用 finalize()来实现	finalize()
异常处理对象 Exceptions	一般用字母“e”来表示	e
常量	用大写字母，单词之间用下划线相连	MIN_BALANCE, DEFAULT_DATE
Get./set 成员函数	用 get/set 作名称的前缀	getFirstName(), getWarpSpeed(), setFistName(), setWarpSpeed(),
接口 interfaces	用全英文描述，所有单词的首字母大写	Runnable, Contactable, Prompter, Singleton
包 packages	用全英文描述，首字母小写，非开头单词的首字母大写	java.awt, com.ambysoft.www.persistence.mapping
局部变量	用全英文描述，非开头单词的首字母大写。	grandTotal, customer, newAccount,
循环变量	通常用 i, j, k ，也可用其他有意义的单词	i, j, k, counter,

成员函数	用全英文描述，非开头单词的首字母大写， 命名时采用动宾结构	openFile(), addAccount(),

1.3 包（Package）的命名

包的标示符由使用句号（.）分隔的小写单词或缩写组成，标示符的前两个单词统一为”com.easipass”， “com.easipass.library”为技术部 Java 模版库的保留前缀，用于存放可重用的组件，非通用的类不要以此标示作为前缀。

示例：

package com.easipass.edi.enhancement;

package com.easipass.library.pool.connection;

1.4 类（Class）和接口（Interface）的命名

类的名字由大写字母开头而其它字母都小写的单词组成。

示例：

ConnectionPool、Runnable

1.5 变量的命名

- 变量使用有意义的名字，使用缩写也要让人一目了然（否则应加注释）
- 变量命名不要使用匈牙利命名法。
- 变量的名字必须用一个小写字母的单词开头，后面的单词用大写字母开头。
  示例：

  String className;

1.6 常量（Static Final、Final）的命名

常量的名字由大写字母的单词或缩写组成，单词与单词之间用下划线分隔。

示例：

MAX_SIZE、CONN_STRING

1.7 参数的命名

参数的名字必须和变量的命名规范一致。（由于 JAVA 中参数对象都是传地址的（除了 String，和基本数据类型），所以将参数与变量同样对待。）

1.8 方法的命名

原则（对于非构造器函数）：方法名应能准确描述对象的行为和功能，首单词小写，后面的单词首字母大写其余字母小写。方法名应包含一个动词描述其行为，如'load','initialize'或'is', 'has'， 'set' ， 'get'作为名称的一部分。

进行设置：set 作为前缀

```
public void setFieldValue(String fieldName,String fieldValue) {
        ……
}
```

取值：get 作为前缀

```
public String getFieldValue(String fieldName) {
                ……
}
```

1.9 数组的申明

数组应该总是用下面的方式来命名：

```
byte[] buffer;
```

而不是：

```
byte buffer[];
```

1.10 几个注意点

1.10.1 括号

通常建议在有多个操作符的表达式中借助于括号来更加清楚地表达一条语句。这能帮助你和其他程序员更好的理解代码。

```
if (a == b && c == d)                    // 避免
if ((a == b) && (c == d))        // 较好
```

1.10.2 返回值

写代码时应该尽量让你的程序结构很好地体现意图。如，

```
if (booleanExpression) {
return true;
} else (
```

```
    return false;
  )
```

应该改为如下语句。

```
    return booleanExpression;
```

类似的，

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

应该写为，

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

1.10.3 条件运算中的表达式

如果在三重操作符“?:”中的“?”前面还有二元操作，那么必须采用括号。如，

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

1.10.4 警惕内存或其它资源的泄漏

由于 Java 中有内存垃圾收集的机制，并且没有指针类型，开发者就可以从 C/C++的内存噩梦中解脱出来。正因为如此，很多开发者就不再关注内存的使用问题。小心，这里存在陷阱。

请注意，Java 的这个机制是内存垃圾收集而非内存收集！所以，你需要把使用的对象变成 JVM 可以识别的垃圾。一般而言，在方法中声明使用的变量在方法外面时就会成为垃圾。但是有些却不是，例如：

```
private HashMap dialogMap = new HashMap();
…
{
 JDialog dialog = new JDialog(“XXX”);
 …
 dialogMap.put(dialog.getTitle(),dialog);
 }
…
```

如果你在整个代码期间都没有调用如下语句：

```
    Object ref =dialogMap.remove(key);
    …（释放资源，如：((JDialog)ref).dispose()）
```

ref = null;

或者

dialogMap.clear()（有时，仅仅这个语句还不行）

那么你的代码就存在内存泄漏问题！

几乎所有存储对象的结构都要引起注意，看看是否有资源没有释放，没有成为垃圾而无法回收。下面的对象要严格释放：

java.sql.Connection;

java.sql.Statement;

java.sql.PreparedStatement

java.sql.CallableStatement

java.sql.ResultSet

这些对象如果不释放其资源，不仅仅是内存问题；还将引起数据库问题。如果不释放 Connection，那么很快就用尽连接或是数据库巨慢（数据库连接数目还受到 Licence 的限制）。如果不释放后面的对象资源，那么很快就会用尽数据库游标，因为每打开一次后面的资源就要使用一个游标，即使语句中没有使用游标（其实每个 DDL 语句都使用一个缺省游标）。而数据库的游标一般是有限制的，Oracle8.1.6 中缺省为 100，Oracle8.1.7 中缺省为 300。

常见的需要进行资源释放的对象还有：

java.io.InputStream

java.io.OutputStream

java.io.Reader

java.io.Writer

java.net.Socket

java.nio.channels.Channel

… …

千万注意！

可能在一段时间内永远不会碰到内存不足的问题，是不是就不用警惕上面提到的内容呢？

No！

JVM 中垃圾内存的收集还受到内存容量的影响。当还有可用内存时，常常不会主动去垃圾收集。这就是为什么常常没有碰到内存不足的问题，因为你机器的内存足够大。

为了 Java 程序有序的运行，你可以为它设定一个最大最小内存使用量。就像 Weblogic 中的一样，如：

-hotspot -ms64m -mx64m。

这样有利于更好的利用垃圾收集机制。

第2章　JAVA 编码规范

有许多规定和标准对提高 JAVA 代码的可维护性是非常重要的，但其中，如何使你的代码更易于他人的理解是最重要的。

代码部分	相应的编码规定
属性（fields）	● Field 应该设为私有 ● 不要直接访问属性，应该定义存取成员函数 ● 不要用常量静态属性（final static fields），应使用存取成员函数 ● 不要重名 ● 静态属性要初始化
Classes	● 减少 publice 和 protected 接口的数量 ● 在编码之前为类定义公共接口 ● 按照如下顺序声明类属性和成员函数 a) constructors b) finalize() c) public member functions d) protected member functions e) private member functions f) private field
局部变量	● 不要命名重复 ● 每行只能声明一个变量 ● 局部变量采用行内注释 ● 紧跟在使用之前定义局部变量
成员函数	● 先进行总体说明注释(document comment） ● 将代码安逻辑关系分段 ● 合理使用空行 ● 好的成员函数应能够在 30 秒钟内被其他人理解 ● 写单行的短注释 ● 尽可能控制对成员函数内部成员的访问

	● 说明操作顺序

第3章　J2EE 命名规范

在开发 J2EE 系统的过程中，随着系统开发的深入，如何控制各种对象和资源的命名就成为一个使系统成功的一个重要因素。任何随意性都将导致系统在后期产生混乱，结构不清晰，导致更多的维护成本。

3.1　J2EE 包结构

实体对象

com.company.projectname.model.EjbnameModel

例如：用户对象

com.jiewen.posp.model.UserModel

DAO 对象

DAO 接口：com.company.projectname.dao.EjbnameDAO

DAO 实现：com.company.projectname.dao.EjbnameDAOImpl

例如：用户对象

DAO 接口：com.jiewen.posp.dao.UserDAO

DAO 实现：com.jiewen.posp.dao.UserDAOImpl

DAOOracle 实现：com.jiewen.posp.dao.UserDAOImplOracle

DAOMySQL 实现：com.jiewen.posp.dao.UserDAOImplMySQL

异常

com.company.projectname.exception.XXXException

例如：用户关键字已经存在的异常

com.jiewen.posp.exception.UserDupKeyException

Handler 处理类

com.company.projectname.handler.ObjectHandler

例如：线路对象

com.jiewen.posp. handler.SaleHandler

工具类

com.company.projectname.util.Xxxx

例如：

com.jiewen.posp.util.Dom4jUtils

3.2 WEB 应用包结构

Servlet 类

com.company.projectname[.subsystem].web.XxxServlet

例如：POS 管理系统

com.jiewen.pms.web.MainServlet

Action 类

com.company.projectname[.subsystem].action.XxxAction

例如：POS 管理系统

com.jiewen.pms.sys.action.UserAction

Web 工具类

com.company.projectname[.subsystem].util.XXX

例如：

com.jiewen.pms.util.DataUtil

Web 异常类

com.company.projectname[.subsystem].exception.XXXException

例如：通用错误类

com.jiewen.pms.exception.GeneralFailureException

3.3 JNDI 名称

引用数据源(可以使用 DatabaseType)

jdbc/DatabaseType/DataSources

例如：jdbc/oracle/ResTxDataSource

第4章　编码风格

1. 缩进
   用 Tab 键（四格）缩进；

   例:

```
import java.util.*;

public class TestMain {
      public static void main(String argv[]) {
            String Test="Test 测试";
                  int x=0;
           try {
              if (x==0) {
                        System.out.println(Test);
                         }else {
                             System.out.println(Test);
                         }
                      }catch(Exception e) {
                     System.out.println("Error:" + e.toString());
              }
      }
}
```

   package，import，class 定义顶格，其余开始缩进。

   Try,if,while，for，synchronized ，switch……等均需要缩进。

2. { }的使用
   见上例。“{”前有一个空格。

3. 在需要的地方空行。
4. 一行程序不超过 80 个字符。
5. 测试代码中使用 System.out.println();测试完成后一定要删除。推荐使用自己的 Debug 类中的 println 方法。
6. 类中常量、变量、方法定义时，定义顺序：
   1、public 的常量
   2、protected 的常量
   3、private 的常量
   4、public 的变量
   5、protected 的变量
   6、private 的变量
   7、构造器
   8、public 的方法
   9、protected 的方法
   10、private 的方法
7. 方法重载时，书写顺序按参数个数递增的顺序书写。

8. 让一切东西都尽可能地“私有”——private
9. 避免使用“数字”，应尽量将其创建一个常数，并为其使用具有说服力的描述性名称，并在整个程序中都采用常数标识符。这样可使程序更易理解以及更易维护。
10. 涉及构建器和异常的时候，通常希望重新丢弃在构建器中捕获的任何异常——如果它造成了那个对象的创建失败。这样一来，调用者就不会以为那个对象已正确地创建，从而盲目地继续

第5章　注释

原则：每个文件、类、public 的方法和变量均需要加注释，采用 JavaDoc 可以提取出来的风格。

（使用/**

* ……

* ……

*/这种风格）

其它注释在需要的时候使用，如一个技巧，一个功能块，release 后的修改，不满足编码规范的地方，等等。

1、文件注释：

```
/*
  File:
  copyright 2001-2002 Shanghai E&P International, INC.All Rights Reserved.
  Date    Author    Changes
  XX      XX        XX
*/
```

说明：

File 后填文件名

Copyright 如上

Date　Author　Changes 为修改记录，这三个字符串保留在第三行作为列名，每次修改就在下面加上一条记录

2、类注释：

```
/**
 * Description
* @see classname or fully-qualified-classname or fully-qualified-classname#method-name
 * @author author
```

```
 * @version version
 */
```

说明：

Description 用类的描述代替

如果该类需要有关联类或相关类的方法，可以使用@see 标签，在@see 后加上相关的类名或类的方法

@author 后面为类的作者

@version 后面为类的版本

3、函数注释

```
/**
 * Description
 * @see classname or fully-qualified-classname or fully-qualified-classname#method-name
 * @param paramName paramDescription
 * @return returnValueDescription
 * @throws Exception exceptionDescription
 */
```

说明：

Description 用函数的功能描述代替

如果该函数需要有关联类或相关类的方法，可以使用@see 标签，在@see 后加上相关的类名或类的方法

如果该函数具有参数，则使用@param 标签对参数进行说明

如果该函数具有返回值，则使用@return 标签对返回值进行说明

如果该函数会抛出异常，则使用@throws 标签对抛出的异常进行说明

4、公共变量注释

```
/**
 * Description
 */
```

说明：

Description 用变量的功能描述代替

第6章　指导方针

指导方针是一些经验和教训，它告诉新手在实际编程中的注意点，遵循这些方针将会对提高程序质量和编程水平带来帮助

1、在 import 语句中尽量少用通配符'*'，明确要引用的类，检查所有 import 申明的引用类是否都真正被使用
理由：过多的使用通配符将使代码的阅读者难以确定代码的上下文（Context）和依赖性。

2、考虑为每个类写一个 main 方法，main 方法中应包含类的单元测试代码或是演示程序
理由：形成一个基本的测试用例，同时也给使用者提供了类使用方法的实例。

3、对于自启动的 Java 应用程序，包含 main 方法（这里的 main 方法是启动函数）的类应与其它正常类分离
理由：将应用程序嵌入它的组件类中将影响组件的重用性。

4、如果你设想别人可能和你写同样一个类但是以不同的实现方法，考虑定义一个接口（interface），而不要使用抽象类（abstract class）。通常，当一个类是部分抽象的时候才使用抽象类，如：它们实现了一些函数以供它们的子类共享
理由：接口比抽象类更灵活。它们支持多重继承并且能被用于不相关的类中。

5、从来不要把类内的实例变量申明为公用的（public）
理由：这是一个标准的面向对象的原则，把变量公用化将放弃对类的内部结构的控制，这样做也无法对变量的有效性进行验证。

6、少用静态（static）标示符（除了静态常量：static final constants）
理由：静态变量就象非面向对象语言中的全局变量，它们使得方法更依赖于上下文（context-dependent），隐藏了可能的幅面影响，并且的方法和变量都不能在子类中重载（override）。

7、通常使用 long 代替 int，用 double 代替 float，为了保证兼容性，int 用于一些标准的 java 结构和类中（主要的例子就是数组 array，数组索引以及数组隐含的所有东西，例如数组的最大尺寸 maximum sizes）
理由：使用 long 代替 int，double 代替 float 可能使算法上溢（overflow）或下溢（underflow）的可能性减少 4billion 倍。

8、当比较对象时，使用 equals 方法代替操作符==。特别是比较字符串（string）时，不要用==
理由：方法可以重载，操作符无法重载。

第7章　附录一 Java 源文件样例

```
/*
   File: Hello.java
   copyright 2001-2002 Shanghai E&P International, INC.All Rights Reserved.
   Date            Author          Changes
   2001-12-31   Bin Wu          Created
*/

import java.awt.*;

/**
 * A class just shows to you how to use javadoc.
 * @see                  java.lang.Object
 * @version      2001.12.31
 * @author       Joker Wu
 */
public class Hello {

        /**
         *      My Name
         */
        public String myName;

        /**
         *      Just say hello to you
         *      @param word1 firstGreeting
         *      @param word2 secondGreeting
         *      @return GreetingWords
         *      @throws Exception All Abnormal Situation
         */
        public String helloWorld(String word1,String word2) {
                System.out.println(word1+word2);
                return word1+word2;
        }
}
```

第8章　附录二 JavaDoc 注释说明

8.1 Comment documentation

One of the thoughtful parts of the Java language is that the designers didn't consider writing code to be the only important activity—they also thought about documenting it. Possibly the biggest problem with documenting code has been maintaining that documentation. If the documentation and the code are separate, it becomes a hassle to change the documentation every time you change the code. The solution seems simple: link the code to the documentation. The easiest way to do this is to put everything in the same file. To complete the picture, however, you need a special comment syntax to mark special documentation, and a tool to extract those comments and put them in a useful form. This is what Java has done.

The tool to extract the comments is called *javadoc*. It uses some of the technology from the Java compiler to look for special comment tags you put in your programs. It not only extracts the information marked by these tags, but it also pulls out the class name or method name that adjoins the comment. This way you can get away with the minimal amount of work to generate decent program documentation.

The output of javadoc is an HTML file that you can view with your Web browser. This tool allows you to create and maintain a single source file and automatically generate useful documentation. Because of javadoc we have a standard for creating documentation, and it's easy enough that we can expect or even demand documentation with all Java libraries.

8.2 Syntax

All of the javadoc commands occur only within /** comments. The comments end with */ as usual. There are two primary ways to use javadoc: embed HTML, or use "doc tags." Doc tags are commands that start with a '@' and are placed at the beginning of a comment line. (A leading '*', however, is ignored.)

There are three "types" of comment documentation, which correspond to the element the comment precedes: class, variable, or method. That is, a class comment appears right before the definition of a class; a variable comment appears right in

front of the definition of a variable, and a method comment appears right in front of the definition of a method. As a simple example:

```
/** A class comment */
public class docTest {
  /** A variable comment */
  public int i;
  /** A method comment */
  public void f() {}
}
```

Note that javadoc will process comment documentation for only **public** and **protected** members. Comments for **private** and "friendly" members (see Chapter 5) are ignored and you'll see no output. (However, you can use the **-private** flag to include **private** members as well.) This makes sense, since only **public** and **protected** members are available outside the file, which is the client programmer's perspective. However, all **class** comments are included in the output.

The output for the above code is an HTML file that has the same standard format as all the rest of the Java documentation, so users will be comfortable with the format and can easily navigate your classes. It's worth entering the above code, sending it through javadoc and viewing the resulting HTML file to see the results.

8.3 Embedded HTML

Javadoc passes HTML commands through to the generated HTML document. This allows you full use of HTML; however, the primary motive is to let you format code, such as:

```
/**
* <pre>
* System.out.println(new Date());
* </pre>
*/
```

You can also use HTML just as you would in any other Web document to format the regular text in your descriptions:

```
/**
* You can <em>even</em> insert a list:
* <ol>
```

```
* <li> Item one
* <li> Item two
* <li> Item three
* </ol>
*/
```

Note that within the documentation comment, asterisks at the beginning of a line are thrown away by javadoc, along with leading spaces. Javadoc reformats everything so that it conforms to the standard documentation appearance. Don't use headings such as **<h1>** or **<hr>** as embedded HTML because javadoc inserts its own headings and yours will interfere with them.

All types of comment documentation—class, variable, and method—can support embedded HTML.

8.4 @see: referring to other classes

All three types of comment documentation (class, variable, and method) can contain **@see** tags, which allow you to refer to the documentation in other classes. Javadoc will generate HTML with the **@see** tags hyperlinked to the other documentation. The forms are:

```
@see classname
@see fully-qualified-classname
@see fully-qualified-classname#method-name
```

Each one adds a hyperlinked "See Also" entry to the generated documentation. Javadoc will not check the hyperlinks you give it to make sure they are valid.

8.5 Class documentation tags

Along with embedded HTML and **@see** references, class documentation can include tags for version information and the author's name. Class documentation can also be used for *interfaces* (see Chapter 8).

8.5.1 @version

This is of the form:

```
@version version-information
```

in which **version-information** is any significant information you see fit to include. When the **-version** flag is placed on the javadoc command line, the version information will be called out specially in the generated HTML documentation.

8.5.2 @author

This is of the form:

```
@author author-information
```

in which **author-information** is, presumably, your name, but it could also include your email address or any other appropriate information. When the **-author** flag is placed on the javadoc command line, the author information will be called out specially in the generated HTML documentation.

You can have multiple author tags for a list of authors, but they must be placed consecutively. All the author information will be lumped together into a single paragraph in the generated HTML.

8.5.3 @since

This tag allows you to indicate the version of this code that began using a particular feature. You'll see it appearing in the HTML Java documentation to indicate what version of the JDK is used.

8.6 Variable documentation tags

Variable documentation can include only embedded HTML and **@see** references.

8.7 Method documentation tags

As well as embedded documentation and **@see** references, methods allow documentation tags for parameters, return values, and exceptions.

8.7.1 @param

This is of the form:

```
@param parameter-name description
```

in which **parameter-name** is the identifier in the parameter list, and **description** is text that can continue on subsequent lines. The description is considered finished when a new documentation tag is encountered. You can have any number of these, presumably one for each parameter.

8.7.2 @return

This is of the form:

```
@return description
```

in which **description** gives you the meaning of the return value. It can continue on subsequent lines.

8.7.3 @throws

Exceptions will be demonstrated in Chapter 10, but briefly they are objects that can be "thrown" out of a method if that method fails. Although only one exception object can emerge when you call a method, a particular method might produce any number of different types of exceptions, all of which need descriptions. So the form for the exception tag is:

```
@throws fully-qualified-class-name description
```

in which **fully-qualified-class-name** gives an unambiguous name of an exception class that's defined somewhere, and **description** (which can continue on subsequent lines) tells you why this particular type of exception can emerge from the method call.

8.7.4 @deprecated

This is used to tag features that were superseded by an improved feature. The deprecated tag is a suggestion that you no longer use this particular feature, since sometime in the future it is likely to be removed. A method that is marked **@deprecated** causes the compiler to issue a warning if it is used.

8.8 Documentation example

Here is the first Java program again, this time with documentation comments added:

```
//: c02:HelloDate.java
import java.util.*;

/** The first Thinking in Java example program.
 * Displays a string and today's date.
 * @author Bruce Eckel
 * @author www.BruceEckel.com
 * @version 2.0
```

```
*/
public class HelloDate {
  /** Sole entry point to class & application
   * @param args array of string arguments
   * @return No return value
   * @exception exceptions No exceptions thrown
  */
  public static void main(String[] args) {
    System.out.println("Hello, it's: ");
    System.out.println(new Date());
  }
} ///:~
```

The first line of the file uses my own technique of putting a '**:**' as a special marker for the comment line containing the source file name. That line contains the path information to the file (in this case, **c02** indicates Chapter 2) followed by the file name[24]. The last line also finishes with a comment, and this one indicates the end of the source code listing, which allows it to be automatically extracted from the text of this book and checked with a compiler.

第9章　文档修订记录

版本号	修改点说明	变更日期	变更人	审批人

修改点说明的内容有如下几种：创建、修改、删除