**XXX智慧项目**

**代码编写规范**

**拟制：**

**审核：**

**会签：**

**XXX科技有限公司**

**目 录**

[一、 概述 5](#_Toc17025)

[（一） 背景 5](#_Toc15858)

[（二） 读者对象 5](#_Toc15350)

[（三） 商讨 5](#_Toc16017)

[二、 基本原则 6](#_Toc6070)

[（一） 一开始就必须正确的使用规范 6](#_Toc9369)

[（二） 简易性原则 6](#_Toc17262)

[（三） 清晰性原则 6](#_Toc11045)

[（四） 健壮性原则 6](#_Toc10571)

[（五） 效率原则 6](#_Toc18734)

[三、 格式规范 7](#_Toc11251)

[（一） 代码缩进 7](#_Toc18440)

[（二） 分解较长的行 8](#_Toc17296)

[（三） 空白区域的使用 11](#_Toc19149)

[（四） SQL语句书写规范 16](#_Toc19459)

[1． SELECT语句书写规则 16](#_Toc13900)

[2． insert语句书写规则 18](#_Toc21926)

[3． update语句的编写规则 21](#_Toc9513)

[4． delete语句的编写规则。 22](#_Toc5463)

[四、 命名规范 23](#_Toc12890)

[（一） 使用有意义的命名 23](#_Toc13725)

[（二） 使用术语进行命名 24](#_Toc29729)

[（三） 超长命名处理 24](#_Toc6689)

[（四） 保留元音 24](#_Toc5164)

[（五） 缩写后的首字母大写 25](#_Toc28633)

[（六） 不要用字符的大小写来区分命名 25](#_Toc5524)

[（七） 包命名 26](#_Toc30610)

[1． 包命名 26](#_Toc8125)

[2． 使用单一的、小写单词来命名每一个包 26](#_Toc9658)

[（八） 类命名 26](#_Toc7541)

[1． 类或接口的每个单词的首字母大写 26](#_Toc18847)

[2． 使用名词来命名类 27](#_Toc14782)

[3． 一组有关系的属性、静态服务或常量的类命名采用复数 27](#_Toc16252)

[（九） 接口命名 30](#_Toc20514)

[1． 使用名词或形容词来命名接口 30](#_Toc3077)

[（十） 方法的命名 30](#_Toc21643)

[1． 方法的首单词的首字母小写，后面的单词的首字母大写 30](#_Toc30043)

[2． 使用动词来命名方法 31](#_Toc5873)

[3． 遵从JavaBeans规则来命名访问方法 32](#_Toc3210)

[（十一） 变量命名 33](#_Toc10589)

[1． 在变量的命名中，变量的首单词的首字母小写，后面的单词的首字母大写 33](#_Toc29691)

[2． 使用名词来命名变量 34](#_Toc8972)

[3． 集合含义的变量采用复数形式进行命名 35](#_Toc21126)

[（十二） 属性命名 35](#_Toc4545)

[1． 使用“this”关键字限定属性变量，用于区分逻辑变量 36](#_Toc8200)

[（十三） 参数命名 36](#_Toc12928)

[1． 当一个构件器或一个“set”方法通过参数赋值给一个属性时，将参数和属性采用同样的名字来命名 36](#_Toc8099)

[（十四） 常量命名 37](#_Toc5449)

[五、 注释规范 38](#_Toc15641)

[（一） 保持注释和代码同步 38](#_Toc11152)

[（二） 使用主动语态并尽量忽略不需要的用词 38](#_Toc28129)

[（三） 注释类型 38](#_Toc32641)

[（四） 版权信息注释 38](#_Toc31707)

[（五） 使用文档注释来描述程序接口 39](#_Toc591)

[（六） 使用标准代码去隐藏代码而不要删除它 41](#_Toc3805)

[（七） 使用行注释来解释实现细节 42](#_Toc4644)

[（八） 文档注释 43](#_Toc13163)

[1． 在代码编写前就描述程序接口 43](#_Toc7278)

[2． 文档中应描述公有、保护、包和私有的成员 43](#_Toc20562)

[3． 对每个包提供概要描述信息和总体的功能 43](#_Toc10128)

[4． 对每个应用或一组包提供概要描述信息和总体的功能 43](#_Toc18947)

[（九） 注释格式 44](#_Toc10066)

[1． 在所有的文档注释中，使用一致的格式和结构 44](#_Toc18912)

[（十） 使用第三人称代词 45](#_Toc22904)

[（十一） 使用“this”而不是“the”指向当前类的实例 45](#_Toc4313)

[（十二） 不要在方法或构造函数名后添加“（）”，除非做一个特殊的标识 46](#_Toc5225)

[（十三） 注释内容 47](#_Toc14348)

[1． 为每一个类、接口、属性和方法提供一个概要描述 47](#_Toc32729)

[2． 充分描述每个方法提供的操作 47](#_Toc5956)

[3． 解释和理解你的软件的使用，最简单的方法是提供例子。 47](#_Toc12223)

[（十四） 以文档方式记录已知的缺点和不足 48](#_Toc2185)

[（十五） 记录同步的语义 49](#_Toc7215)

[（十六） 代码内部注释 49](#_Toc14625)

[1． 需要帮助他人理解代码时才增加内部注释 49](#_Toc1228)

[2． 描述代码应该做的事情，而不是描述代码正在做什么 49](#_Toc31670)

[3． 避免使用行末注释的方法 50](#_Toc1256)

[4． 采用行末注释的方式解释局域变量 51](#_Toc27765)

[5． 创建并使用一套关键字集来标注未解决的问题 51](#_Toc1425)

[6． 在多重嵌套控制结构中标注结束标志 51](#_Toc32615)

[7． 如果在switch语句中没有break，则在相应的两个case之间添加一行“fall-through”注释 52](#_Toc25881)

[8． 标注empty语句 53](#_Toc4692)

[六、 编程规范 53](#_Toc10211)

[（一） 将代表基础数据类型的类声明为final 53](#_Toc9960)

[（二） 定义类和方法尽可能短小精悍 54](#_Toc25712)

[（三） 将所有成员变量声明为private类型 54](#_Toc20225)

[（四） 使用多态(polymorphism)，不使用isntanceOf 54](#_Toc27910)

[（五） 类型安全 55](#_Toc25663)

[1． 用Object对象代替一般目的类可以提供静态类型检验 55](#_Toc2922)

[2． 将枚举类型封装为类 55](#_Toc11606)

[（六） 语句与表达式 57](#_Toc1917)

[1． 用等价的方法替换重复的表达式 57](#_Toc26290)

[2． 在控制结构中尽量使用代码块来代替表达式 57](#_Toc14277)

[3． 用圆括号清晰地标明运算的顺序 59](#_Toc4454)

[4． 在switch语句的最后一个case语句块结束的地方加上一个break语句。 59](#_Toc4897)

[5． 判断两个对象的值是否相等，使用equals()，而不是使用== 61](#_Toc17033)

[（七） 构造函数 62](#_Toc13625)

[1． 保证对象的构造总是有效 62](#_Toc15970)

[2． 在构造函数中不要引用非常量方法 62](#_Toc4977)

[3． 使用嵌套构造器来消除冗余的代码 63](#_Toc17941)

[（八） 异常处理 64](#_Toc20067)

[1． 使用未检测到的、运行时异常来报告程序中未预测到的逻辑错误 64](#_Toc29625)

[2． 使用可检验的异常处理来报告可能发生的程序错误，但是，在正常的程序运行中，这种是错误是很少发生的。 65](#_Toc11305)

[3． 使用return代码报告可预测性的状态变化 65](#_Toc8177)

[4． 转移异常信息的处理 65](#_Toc27138)

[5． 不要对运行时异常或者错误不做任何处理 65](#_Toc25987)

[6． 用一个finally语句块释放资源 66](#_Toc18454)

[七、 编程技巧 68](#_Toc14984)

[（一） byte 数组转换到 characters 68](#_Toc2149)

[（二） Utility 类 68](#_Toc5152)

[1.1 初始化 68](#_Toc8744)

[（三） 对象构造 68](#_Toc28887)

[1． 不要在循环中构造和释放对象 68](#_Toc13604)

[2． 处理 String 的时候要尽量使用 StringBuffer 类。 68](#_Toc18204)

[（四） 避免太多的使用 synchronized 关键字 69](#_Toc25801)

[（五） 通过对系统数据结构的划分与组织的改进，以及对程序算法的优化来提高空间效率。 69](#_Toc117)

[（六） 循环体内工作量最小化。 70](#_Toc16655)

[（七） 不应花过多的时间提高调用不很频繁的函数代码效率。 71](#_Toc12118)

[（八） 在保证程序质量的前提下，通过压缩代码量、去掉不必要代码以及减少不必要的局部和全局变量，来提高空间效率。 72](#_Toc21357)

[（九） 在多重循环中，应将最忙的循环放在最内层。 72](#_Toc23760)

[（十） 尽量用乘法或其它方法代替除法，特别是浮点运算中的除法。 72](#_Toc22628)

# 概述

## 背景

程序的编码是一个创造性极强的工作，虽然要奇思妙想，但也必须要遵守一定的规则和限制，编码风格的重要性对软件项目开发来说是不言而喻的。

本系统开发基于Java语言J2EE技术路线，为了规范Java编码、提高编码质量、增强代码的可读性和可维护性特制定本规范。

开发工程师在开发过程中必须遵守本规范，规范是代码编写及代码验收等管理环节中必须执行的标准。

## 读者对象

本手册的读者对象为Java开发工程师、测试工程师、数据库设计工程师。

## 商讨

对规范如有疑问请及时和技术平台小组联系。

# 基本原则

以下是开发过程中必须遵循的基本原则，可能缺乏操作性，却是后续规范的概括和总结。对暂未能列入本编码规范的问题，作一般性要求。

## 一开始就必须正确的使用规范

为保证整个产品编码风格的一致性，要求在产品的第一个模块开始，必须严格使用本规范开始编码。

## 简易性原则

代码的编写力求简易性，构建类和方法力求简单，提高代码的可读性和可维护性。

## 清晰性原则

确保每一个类、接口、方法、变量和对象都有清楚的功能，并解释在何时、何地使用。设计人员应确保此原则的贯彻执行。

## 健壮性原则

程序要健壮，要为错误和异常提供可预见的处理，不能隐藏错误和异常。

## 效率原则

编码要兼顾程序运行期效率。

# 格式规范

## 代码缩进

所有的语句块内部需要缩进，语句块通常被{}包围，它包括但不仅包括类定义，函数定义，条件分支，循环体

采用Tab来进行代码缩进，增强可读性，不要使用太多的空格。

class MyClass {

..void function(int arg) {

....if (arg < 0) {

......for (int index = 0; index <= arg; index++) {

........// ......

......}

....}

..}

}

另外，在程序块中，对于一个用于处理循环的结构，也要使用缩进，以突出结构显示：

void funcition(int arg) {

..loop:

....for (int index = 0; index <= arg; index++) {

......switch (index) {

........case 0:

..........//..

..........break loop; // 退出语句

........default;

..........//..

..........break; // 退出switch

......}

....}

}

在每一个代码块中，对于最后字符为 ”{” 的一行，在”{”前加上空格； ”}” 独占一行，同时首字符与这个代码块对齐。下面的例子阐明运用这条规则是如何来进行JAVA的定义与结构控制的：

## 分解较长的行

为了确保源代码在打印时仍然可读，限制源代码中行的最大长度，以满足打印的要求，尽量控制在每行不超过80个字符，每行不允许超过100字符。

1.不允许在一行上编写多个表达式：

错误：double x = Math.random(); double y = Math.random(); // 太长!

正确

double x = Math.random();

double y = Math.random();

2.如果存在复杂的表达式：

double length = Math.sqrt(Math.pow(Math.random),2.0) + Math.pow(Math

.random(),2.0));//太长

不要为了断行将非公共的变量提取出来，这样将增加执行的时间，对这种情况，可以在+号附近将它断开成两行,如Double length = Math.sqrt(Math.pow(Math.random), 2.0)

+ Math.pow(Math.random(), 2.0));

对于这样一种情况，将复杂的表达式分解成小的表达式，并将小的表达式的值用临时变量来进行处理。

double xSquared = Math.pow(Math.randow(),2.0);

double ySquared = Math.pow(Math.randow(),2.0);

double length = Math.sqrt(xSquared+ySquared);

最后，对于不能按照上面两种方式进行处理的较长的行，通过回车换行来进行处理：

方法一

在一行中，存在一个或多个逗号：

double length = Math.sqrt(Math.pow(x,2.0),Math.pow(y,2.0)); //太长

采用这样的方式来进行处理：按照逗号来进行换行处理，并与第一个逗号表达式对齐。

double length = Math.sqrt ( Math.pow(x,2.0),

Math.pow(y,2.0));

方法二

如不存在逗号：

class Myclass {

private int field;

...

Boolean equals(Object obj) {

return this == obj || (obj instanceof MyClass && this.field

== obj.field); //太长!

}

...

}

这种情况下，建议采用最低优先级的操作来进行划分，或是存在一个比“等于”高的优先级操作来进行划分：

class Myclass {

private int field;

...

Boolean equals(Object obj) {

return this == obj

|| (obj instanceof MyClass && this.field == obj.field);

}

...

}

方法三

在需要的时候，重复应用方法一和方法二，直到每一行代码都小于最大的允许的长度。

对于函数的定义，太长时折行，优先将throws子句断行，其次将类似参数尽量位于同一行，如

Public void drawRect(int x, int y, Int width, int height, int color)

throws Exception {

...

}

## 空白区域的使用

代码中空白区域使用不当，会造成很难阅读和理解，适当的使用空白区域，会使得编写的方法、备注、代码块和表达式清晰易懂。

空格的使用在如下情况下:

定义常量时每一组同类型的常量=号上下对齐

在 “)” 或“}” 后紧随代码；在代码后紧随 “(” 或“{”；在 “)” 后紧随“{” 时使用空格：

for.(...).{

...

}

while.(...).{

...

}

do.{

...

}.while.(...);

switch.( ...).{

...

}

if.(...).{

...

}

else.if.(...).{

...

}

else.{

...

}

try.{

...

}

catch.(...).{

...

}

finally.{

...

}

对于二元操作，除了“.”操作符外，使用空格

double length = Math.sqrt(x \* x + y \* y);

double xNorm = length > 0.0 ? (x / length) : x;

空白行使用在下列情况下：

对于负号的情况，如果之后为长表达式，按二元操作符的要求补空格，如

int a = -3(没有空格)

int a = - a + 3(补一个空格)

方法实现中的每一个逻辑块：

void handleMessage(Message message) {

DataInput content = message.getDateInput();

Int messageType = content.readInt();

Switch (messageType) {

case WARNING:

...处理 ...

break;

case ERROR:

... 处理...

break;

default:

... 处理...

break;

}

}

类和接口的成员定义

public class Foo {

/\*\*

\* 定义一个内部类

\*/

class InnerFoo {

...

}

/\*\*

\* 成员 Bar.

\*/

private Bar bar;

/\*\*\*

\* 用指定的Bar构建一个 Foo /

\*/

Foo(Bar bar) {

this.bar = bar;

}

}

源代码文件中每个类和接口的定义

/ \*

\* ...文件描述 ...

\*/

/\*\*\*

\* ...接口描述 ...

\*/

interface FooInterface {

...

}

/\*\*\*

\* ...类描述 ...

\*/

public class Foo implements FooInterface {

...

}

## SQL语句书写规范

### SELECT语句书写规则

分为两种情况：

关键字用大写，一目了然，如：SELECT count(\*) as counter FROM ctrmas，select语句的选择字段需要详细，原则上不鼓励使用select \* from，而应该书写具体的字段名，以免表结构变更带来兼容性问题。

对于简单的Select语句可以使用一个变量来写完：

strSql = "select count(\*) as counter from ctrmas";

对于复杂的select语句使用连个变量完成一条语句，要求每个字段、每个where中的条件作为一行方便阅读，

在每行的前面加一个空格。同时要求每行的字段对齐，SQL关键字对齐。

例1：

strFrom = "select distinct(asgmas.empnum)," +

" empmas.shtnam " +

"from asgmas, outer empmas ";

不允许下面的写法，上面的语句能被编译器优化为常量字符串，下面的语句会被处理成4条语句，要构造3次StringBuffer。

strFrom = "";

strFrom = strFrom + “select distinct(asgmas.empnum),";

strFrom = strFrom + " empmas.shtnam ";

strFrom = strFrom + " from asgmas, outer empmas ";

例2：

strWhere = " where asgmas.crewno = '" + mstrPKey + "'" +

" and asgmas.empnum is not null " +

" and empmas.empnum = asgmas.empnum " +

" order by shtnam";

或用下面写法

strWhere = "";

strWhere = strWhere + " where asgmas.crewno = '" + mstrPKey + "'";

strWhere = strWhere + " and asgmas.empnum is not null ";

strWhere = strWhere + " and empmas.empnum = asgmas.empnum ";

strWhere = strWhere + " order by shtnam";

3)

strSQL = strFrom + strWhere;

### insert语句书写规则

insert语句的编写规则，必须列出所有要插入的字段。每个字段的值占用一行，方便阅读与检查。

strSQL = "insert into rtamas " +

"(rtanum, avltyp, effdat, expdat, begtim, endtim, enttyp, plancd, crewno, skilcd, empnum, unvcod, shtnam, qtyval, lstusr, lstdtm, fstusr, fstdtm) " +

"values (" +

mlngSeqnum + ",'" +

mstrAvltyp + "','" +

FormatDBDate(mdtmDatefr) + "','" +

FormatDBDate(mdtmDateto) + "','" +

Format(mdtmTimefr, "hh:mm:ss") + "','" +

Format(mdtmTimeto, "hh:mm:ss") + "'," +

"'EM','" +

mstrPlancd + "','" +

mstrCrewno + "','" +

mstrSkilcd + "','" +

mstrEmpnum + "','" +

mstrUnvcod + "','" +

Trim(ReplaceChar(mstrDescription, "'", " ")) + "'," +

"1," +

"'WB','" +

FormatDBDateTime(Now) + "'," +

"'WB','" +

FormatDBDateTime(Now) + "')";

也可以每行分开写，如：

strSQL = "";

strSQL = strSQL + " insert into rtamas " ;

strSQL = strSQL + " (rtanum, avltyp, effdat, expdat, begtim, endtim, enttyp, plancd, crewno, skilcd, empnum, unvcod, shtnam, qtyval, lstusr, lstdtm, fstusr, fstdtm) ";

strSQL = strSQL + " values (";

strSQL = strSQL + mlngSeqnum + ",'";

strSQL = strSQL + mstrAvltyp + "','";

strSQL = strSQL + FormatDBDate(mdtmDatefr) + "','";

strSQL = strSQL + FormatDBDate(mdtmDateto) + "','";

strSQL = strSQL + Format(mdtmTimefr, "hh:mm:ss") + "','";

strSQL = strSQL + Format(mdtmTimeto, "hh:mm:ss") + "',";

strSQL = strSQL + "'EM','";

strSQL = strSQL + mstrPlancd + "','";

strSQL = strSQL + mstrCrewno + "','";

strSQL = strSQL + mstrSkilcd + "','";

strSQL = strSQL + mstrEmpnum + "','";

strSQL = strSQL + mstrUnvcod + "','";

strSQL = strSQL + Trim(ReplaceChar(mstrDescription, "'", " ")) + "',";

strSQL = strSQL + "1,";

strSQL = strSQL + "'WB','";

strSQL = strSQL + FormatDBDateTime(Now) + "',";

strSQL = strSQL + "'WB','";

strSQL = strSQL + FormatDBDateTime(Now) + "')";

### update语句的编写规则

与insert语句相似，对于一行必须只能给一个字段的赋值。但是当where条件复杂时可以用单独的变量来处理，同样也可以采用分行写的方法。

strSQL = "update rtamas set avltyp = '" + mstrAvltyp + "'," +

"effdat = '" + FormatDBDate(mdtmDatefr) + "'," +

"expdat = '" + FormatDBDate(mdtmDateto) + "'," +

"begtim = '" + Format(mdtmTimefr, "hh:mm:ss") + "'," +

"endtim = '" + Format(mdtmTimeto, "hh:mm:ss") + "'," +

"plancd = '" + mstrPlancd + "'," +

"crewno = '" + mstrCrewno + "'," +

"skilcd = '" + mstrSkilcd + "'," +

"empnum = '" + mstrEmpnum + "'," +

"unvcod = '" + mstrUnvcod + "'," +

"shtnam = '" + mstrDescription + "'," +

"lstusr = 'WB'," +

"lstdtm = '" + FormatDBDateTime(Now) + "' " +

"where rtanum = " + mlngSeqnum;

### delete语句的编写规则。

与select语句相似。当where条件复杂时可以用单独的变量来处理。每个条件占用一行。可以采用分行写的方法。

strSQL = "delete from rtamas " +

"where rtanum = " + mlngSeqnum + " " +

"and plancd = '" + mstrPlancd + "'," +

"and crewno = '" + mstrCrewno + "'";

# 命名规范

## 使用有意义的命名

在命名一个类、变量、方法或常量时，使用有意义的命名，可提高程序的可读性，避免使用单个字符或通用名字来定义类、变量、方法或常量。

下面例中的变量“a”和常量“65”的意义就不清楚。

if (a < 65) { //“a”描述的意义是什么

y = 65 – a;

}

else {

y = 0;

}

通过定义有意义的命名，修改后的代码就比较易懂。

if (age < RETIREMENT\_AGE) {

yearsToRetriement = RETIREMENT\_AGE – age;

}

else {

yearToRetirement = 0;

}

对于那些能够提供足够的信息来决定其目的的临时变量，可以不使用这种规则，如下面代码中的用于循环计数或索引的变量：

for (int i = 0; i < numberOfStudents; ++i) {

enrollStudent(i);

}

对于经常使用的这种类型的变量，也可以作为标准。

## 使用术语进行命名

使用在特定领域的术语来进行命名，例如：比如使用电力行业术语来命名。

## 超长命名处理

对于一个对象的命名必须描述它的使用目的，如果一个类、接口、变量或方法的名字非常长。那么对应的实体将很难被实现。面对这种情况，在不减少实际意义的前提下简化实体的命名，首先考虑它的设计目的，分解这个实体，通过更简单的命名来生成新的类、接口、方法和变量。

## 保留元音

如果多个有含义的命名被缩写后相同，这样的缩写将影响代码的可读性，并使得代码容易混淆，因此在缩写时，保留元音。但是这样的处理也会造成临时阅读代码的用户感到困难。对比下面两段程序：

public Message appendSignature(Message message,String signature) {

...

}

public Msg appndSgntr(Msg msg,String sgntr) {

...

}

如果需要移出元音来简化长命名，那么就应该考虑初始的命名是不是适当的。

## 缩写后的首字母大写

对于那些以单词作为分割的命名，这条规则可以帮助我们排除混淆。

setDSToffset setDstOffset

loadXMLDocument loadXmlDocument

但不要在以下的处理中使用这个规则：

对于那些只包含大写字母的常量（常量的命名规则参见4.14）

static final String XML\_DOCUMENT = “text/XML”

方法、变量、参数命名时，首单词的缩写用小写（参见4.11）

private Document xmlDocument

## 不要用字符的大小写来区分命名

JAVA编译器能够通过大小写来区分只存在大小写差异的两个不同命名。但对于阅读代码的人来说，在读取的时候就容易造成混淆。

例如：在同一个代码范围内，存在两个变量theSQLInputStream和theSqlInputStream,这样的代码就会给阅读的人带来很大的不便。

## 包命名

### 包命名

使用公司颠倒的Internet域名作为包的根限定。例如定义一个包为server，那么这个包的命名为server.luculent.net

提示：在命名的过程中，不要使用java和javax来命名包。

### 使用单一的、小写单词来命名每一个包

包在命名时，采用有含义的单个小写单词，例如：java.io和java.net

## 类命名

### 类或接口的每个单词的首字母大写

类或接口的每个单词的首字母大写。在每个命名中使用首字母大写来区分每个单词，用这种方式来区分类或接口与变量的命名(变量的命名规则参见4.11)

public class PrintStream

extends FilterOutputStream {

...

}

public interface ActionListener

extends EventListener {

...

}

### 使用名词来命名类

类用来定义对象或相关的事物，故而使用名词标识符来命名类。

class CustomerAccount {

...

}

public abstract class KeyAdapter implements KeyListener {

...

}

### 一组有关系的属性、静态服务或常量的类命名采用复数

使用类来命名的一组有关系的属性、静态服务或变量，使用复数来进行类的命名。

/\*\*

\* The <code>LineMetrics</code> class give

\*access to the metrics needed to layout

\*characters along a line ant to layout of

\*a set of lines.

\*/

public class LineMetrics {

public LineMetrics()

public abstract int getNumChars();

public abstract float getAscent();

public abstract float getDescent();

public abstract float getLeading();

public abstract float getHeight();

...

}

下面示例中，在类java.bean.Beans中定义了一组静态服务：

/\*\*

\* The <code>Beans</code> class provides some

\*general purpose beans control methods.

\*/

public class Beans {

public static Object instantiate(...) {...};

public static Object getInstanceOf(...) {...};

public static boolean isInstanceOf(...) {...};

public static boolean isDesignTime() {...};

public static boolean isGuiAvailable() {...};

public static void setDesignTime(...) {...};

public static void setGuiAvailable(...) {...};

...

}

下面示例中，在类java.sql.Types中定义了一组静态常量：

/\*\*

\*The <code>Types</codes> class defines constants

\*that are used to identify SQL types

\*/

public class Types {

public final static int BIT = -7;

public final static int TINYINT = -6;

public final static int SMALLINT = 5;

public final static int INTEGER = 4;

public final static int BIGINT = -5;

public final static int FLOAT = 6;

public final static int REAL = 7;

public final static int DOUBLE = 8;

public final static int NUMERIC = 2;

public final static int DECIMAL = 3;

public final static int CHAR = 1;

...

}

## 接口命名

### 使用名词或形容词来命名接口

一个接口用来描述一个提供的服务，或用来描述一个服务提供的功能，故而用名词或形容词来描述一个接口。

public interface ActionListener {

public void actionPerformed(ActionEvent e);

}

使用形容词来命名接口的功能和行为，大多数的接口命名采用动词后加“able”或“ible”后缀的形容词来描述接口的功能：

public interface Runable {

public void run();

}

public interface Accessible {

public Context getContext();

}

## 方法的命名

### 方法的首单词的首字母小写，后面的单词的首字母大写

大写字母用于分割方法名中使用的单词，首字母小写用来区分方法和构造器。

class MyImage extends Image {

public MyImage() {

...

}

public void flush() {

...

}

public Image getScaledInstance() {

...

}

}

### 使用动词来命名方法

方法和操作通常使用动词来定义其行为：

class Accout {

private int balance;

...

public void withdraw(int amount) {

deposit(-1 \* amount);

}

public void deposit(int amount) {

this.balance += amount;

}

}

### 遵从JavaBeans规则来命名访问方法

JavaBeans为访问javaBeans实例的属性方法建立了一套标准的规范，那么在类中命名一个方法时就必须遵从这个规范，不管是否实现一个bean。

JavaBean中定义一个布尔类型的方法的命名时，以“is”开始

boolean isValid() {

return this.isValid;

}

JavaBean提供一个读取的方法时，命名采用“get”开始

String getName() {

return this.name;

}

访问方法读取一个索引属性时，使用”int index”

String getAlias(int index) {

return this.aliases(index);

}

JavaBean中设定一个布尔或其他类型的属性时，命名采用“set”开始

void setValid(Boolean isValid) {

this.isValid = isValid;

}

void setName(String name) {

this.name = name;

}

访问方法设定一个索引属性时，使用”int index”和“set”

void setAlias(int index,String alias) {

this.aliases[index] = alias;

}

请在JDK中，坚持使用这些规则，用is/get/set符号来处理Bean的属性。

## 变量命名

### 在变量的命名中，变量的首单词的首字母小写，后面的单词的首字母大写

大写字母用于分割变量名中使用的单词，首字母小写用来区分变量和类（类定义规则参见规则4.8）

class Customer {

...

private Address address;

private Phone daytimePhone;

...

public Address setAddress(Address address) {

Address oldAddress = this.address;

this.address = address;

return oldAddress;

}

...

public void setDaytimePhone(Phone daytimePhone) {

...

}

...

}

### 使用名词来命名变量

变量用来表示一个对象或事物，故而采用名词来标识变量:

class Customer {

...

private Address billingAddress;

private Address shippingAddress;

private Phone daytimePhone;

private Vector openOrders;

...

}

### 集合含义的变量采用复数形式进行命名

对于代表一个集合的变量或属性，使用复数形式来进行命名，这种命名方式可将代表集合值的的变量或属性与代码单个值的变量或属性分离开:

Customer[] customers = newCustomer[MAX\_CUSTOMERS];

void addCustomer(int index, Customer customer) {

this.customers[index] = customer

}

vector orderItems = new Vector();

void addOrderItem(OrderItem orderItem) {

this.orderItems.addElement(orderItem);

}

## 属性命名

### 使用“this”关键字限定属性变量，用于区分逻辑变量

class Dude {

private String name;

public Dude(String name) {

this.name = name;

}

public setName(String name) {

this.name = name;

}

}

## 参数命名

### 当一个构件器或一个“set”方法通过参数赋值给一个属性时，将参数和属性采用同样的名字来命名

使用逻辑变量来隐藏实例变量的名称，这是一种”poor”风格，但这种规则也有些有益的地方，使用相同的命名能比使用不同的命名更能让人意识到其中的重要性，同时使用相同的命名能让你准确地了解参数是赋值给属性的。

class Dude {

private String name;

public Dude(String name) {

this.name = name;

}

public setName(String name) {

this.name = name;

}

}

## 常量命名

在命名常数时，每一个单词中的所有字母大写，单词之间采用下划线来进行分割

使用大写来命名常量。使得他们能够很好与变量区分。

class Byte {

public static final byte MAX\_VALUE = 255;

public static final byte MIN\_VALUE = 0;

public static final Class TYPE = Byte.class;

}

# 注释规范

## 保持注释和代码同步

当修改代码时，必须同时更新对应的注释，代码和文档同时为一个软件产品，故而同样重要。

## 使用主动语态并尽量忽略不需要的用词

对于代码来说，注释是应该是有力的、清楚的并且是简明的语言，故在编写注释时尽量采用主动语态并尽量忽略不需要的用词。

## 注释类型

Java提供三种注释类型：

文档注释用“/\*\*”开始，并由”\*/”结束

/\*\*

\*文档注释.

\*/

单行或行尾注释，采用”//”开始

//单行注释

class Myclass {

int myField; //行尾注释

...

}

每一种注释类型均有特有的使用场合，合理地使用这三种规则。

## 版权信息注释

版权信息必须在 java 文件的开头。/\*\*

\* <p>Title: 主题</p>

\* <p>Description: 描述</p>

\* <p>Copyright: Copyright (c) 2004</p>

\* <p>Company: luculent</p>

\* @author:

\* @version 1.0

\*/

## 使用文档注释来描述程序接口

在代码中，应该把文档注释放在编写的类、接口、方法、构件器或者是属性前，这对于使用javadoc工具来生成对应的HTML格式的文档很有好处，javadoc功能只认可在类、接口、构造器、方法或属性前编写的文档，javadoc会忽略在方法内部的注释，javadoc只认可语句前的一段注释，所以不要在语句前使用多个注释段。

文档注释的主要目的是约定客户端和服务器端之间的连接，一个与方法关联的文档，应该描述所有的可供调用的的功能，以及返回值，没有必要阐述具体的实现细节。

下面实例中，使用文档注释一个类中包含内部类、属性、方法、和构造器：

/\*\*

\* The <code> Rectangle2D</code> class describles

\*a rectangle defined by location(x,y) and

\*dimensions(w,h)

\*...

\*/

public abstract class Rectangle2D

extends RectangularShape {

/\*\*

\* The <code>Double</code> class defines a

\* rectangle specified in double coordinate...

\*/

static class Double extends Rectangle2D {.....}

...

/\*\*

\*The bitmask that indicates that a point lies

\*below this Rectangle2D...

\*/

static int OUT\_BOTTOM;

...

/\*\*

\*Adds a Rectangle2D to this Rectangle2D...

\*/

public void add(Rectangle2D r) {...}

...

/\*\*

\* This is an abstract class that cannot be

\* instantiated derectly...

\*/

protected Rectangle2D() {...}

...

}

## 使用标准代码去隐藏代码而不要删除它

当希望临时隐藏一段代码时在编译时不使用，那么使用C类型的注释，使用这种注释时，要确保在这段注释里没有嵌套其它类型的注释段。

下面代码片断描述了怎样使用这种注释类型来隐藏一个成员函数的定义：

/\*\*

\*...

\*@deprecated

\*/

/\*

我想临时删除这个方法，想知道其它包是否在用它。

public void thisOldFunction() {

//There has got to be a better way!

...

}

\*/

## 使用行注释来解释实现细节

在文档中使用一个或多个行注释：

指出变量或表达式的目的

实现层次的实际要求

复杂的运算规则

缺陷修复

优化代码时

注释问题、缺陷以及不足

在编写代码文档时，尽量减少需要的嵌套注释的使用，对于简单重复出现的代码不要使用注释、注释中只描写其有用的信息。

double totalCost; //发票汇总

...

// over $1000发票的折扣

if (totalCost > 1000.0) { //:TODO:用常量 ?

//此处折扣固定，因为所有的的顾客使用相同的折扣

//我们将来会用一个变量替换，以提高灵活性

totalCost = totalCost \* DISCOUNT;

}

## 文档注释

### 在代码编写前就描述程序接口

创建API参考文档最好的时间是在开发过程前，使用文档或“DOC”类型注释定义每个类或接口。

### 文档中应描述公有、保护、包和私有的成员

提供所有成员的文档注释，包括包、保护的、私有的入口，实现层的文档注释应尽可能的详细，这对于要使用程序或修改程序的开发者来说是很重要的，他需要了解所有的实现细节，不只是查看公共的成员。

### 对每个包提供概要描述信息和总体的功能

javadoc工具能够将包描述生成对应的文档，所以在你的创建包时，需要将包的概要信息和总体功能描述清楚。

创建一个包的描述，你应该创建一个包注释文件，并命名为package.html,且与包的源代码文件存在在同一目录下，javadoc工具会自动在这个目录下寻找这个文件。

包注释文件包含HTML，而不包含java源代码。包的描述信息，应该包含在HTML文档的<body>标签中，javadoc工具将HTML文件中的<body>...</body>标签中的内容作为包的描述。

### 对每个应用或一组包提供概要描述信息和总体的功能

javadoc工具能够将一组互不相干的包的总体功能描述生成文档，使用这个功能为每一个应用或有关联的包提供总体描述。

为了创建一个总体的描述，你必须创建一个总体的注释文件，注释问题扩展名为“.html”,例如：overview.html。为了在文档中包括这个文件，必须告诉javadoc工具，通过-overview选项来设定文件路径

包注释文件包含HTML，而不包含java源代码。总体功能信息，应该包含在HTML文档的<body>标签中，javadoc工具将HTML文件中的<body>...</body>标签中的内容作为一个应用或一组包的概要描述。

## 注释格式

### 在所有的文档注释中，使用一致的格式和结构

一个适当的格式文档注释，依据一个或多个javadoc的标签来进行描述，每个一个文档注释依据下面的格式：

首行注释用“/\*\*”，并与其关联的源代码的缩进相同

其余行使用“\*”开始，并与首行注释的第一个星号对齐

使用单个空格来分割在同一行中每个星号后的描述文档，或者标签描述文档和标签之间使用空行

使用“\*/”做为结束，并与其他行中的“\*”对齐

/\*\*

\* Descriptive text for this entity

\*

\* @tag Descriptive text for this tag

\*/

## 使用第三人称代词

当描述类、接口、方法时,使用第三人称代词(如”they”和”it”),第三人称动词形式(“sets”和“gets”)，不使用第二人称动词形式(如“set”和“get”).

下面是API文档里一些第三人称动词形式：

adds deallocates removes

allocates destroys returns

computes gets sets

constructs provides tests

converts reads writes

## 使用“this”而不是“the”指向当前类的实例

在描述方法的目的或行为时,定义方法用“this”而不是“the”指向类的实例：

/\*\*

\* Return a <code>String</code> representing the

\* value of the <code>Flag</code> object

\* ...

\*/

public String toString() {...}

/\*\*

\* Return a <code>String</code> representing the

\* value of this <code>Flag</code> object

\* ...

\*/

public String toString() {...}

## 不要在方法或构造函数名后添加“（）”，除非做一个特殊的标识

方法或构造函数的参照不要包含括号,除非参照区分重载的方法或构造函数并且希望指向一个重载操作的单个域。不要通过一对圆括号来指出相关方法的名称。如果这个名称和一个重载的方法相关连，而且这个重载的某个方法没有参数,就会引起混淆。

考虑下面重载方法,他们是java.lang.String类里定义的：

public class String {

…

public String toLowerCase() {...}

public String toLowerCase(Locale locale) {...}

…

}

如果使用“toLowerCase()”标识符去指示一个或所有的toLowerCase方法，那么在读取文档时，将会造成混淆，建议使用不带括号的方式进行处理：

toLowerCase 用于指示所有的方法

toLowerCase() 用于指示上述方法中的方法1

toLowerCase(Locale) 用于指示上述方法中的方法2

## 注释内容

### 为每一个类、接口、属性和方法提供一个概要描述

对于每一个类、接口、属性和方法，在文档注释中，至少需包含其概要信息。

### 充分描述每个方法提供的操作

在每一个方法注释中，应该对每个参数、每个检测异常、可能出现的异常以及返回值做详细的描述。

### 解释和理解你的软件的使用，最简单的方法是提供例子。

在每个非凡的类和方法的描述中包含一段简单的例子，并使用<pre>...</pre>标签来维护每个例子的格式。

/\*\*

\* ...

\* If you are formatting multiple numbers.it is

\* more efficient to get the format just once so

\* the system does not have to fetch the

\* information about the local language and

\* country conventions multiple times:

\* <pre>

\* DateFormat df = DataFormat.getDateInstance();

\* for(int i = 0; i < a.length; ++i) {

\* out.println(df.format(myDate[i]) + “;”);

\* }

\* </pre>

\* To format a number for a different Locale,

\* specify the locale in the call to

\* <code> getDateInstance</code>

\* <pre>

\* DateFormat df;

\* df = DateFormat.getDateInstance(Locale.Us);

\* </pre>

\* ...

\*/

public abstract class DateFormat extends Format {

...

}

## 以文档方式记录已知的缺点和不足

该方式不仅可以标识和描述与一个类或方法相关的任何重大问题，而且还可以表明改进代码或这些缺点或不足所在的工作区。如果可能，还可以表示问题什么时候可以被解决。

虽然没有一个程序员愿意在自己的代码中注明问题所在，但是他的同事都希望他能在注明，因为这些备注信息将有助于实现一个工作区，或者将这些问题独立出来以降低对未来变化的冲击。

## 记录同步的语义

在一个方法前面的synchronized修饰符一般表示为了保护对象的状态，使用该方法调用多个线程时是否应该按照顺序调用。用户可以通过查看Javadoc产生的文档来知道某个方法是否定义为synchronized，因为javadoc生成文档时将声明为synchronized的每个方法的修饰符都作为定义的一部分。

除了上述方法外，Java还提供了一种同步机制，这种机制可以适用于一个代码块，而不是整个方法。事实上，使用第二中同步机制的方法同样可以达到线程访问安全的目的，但包含这些同步机制的方法在声明时却无需标明同步标识。这种情况下，你就需要用文档记录所有使用这种内部同步机制的方法。

## 代码内部注释

### 需要帮助他人理解代码时才增加内部注释

假如所增加的内部注释不能增加任何有价值的信息，最好还是不要写注释信息，让代码自身来表示自己好了。

### 描述代码应该做的事情，而不是描述代码正在做什么

其实，对于一份书写格式良好的代码，其他的开发人员只要看一下就能知道它是用来做什么。例如：快速地浏览一下下面的代码就可以知道这个程序实现的功能是：当一张发票总额超过1000美元时可享受5%的折扣。

If(this.invoiceTotal > 1000.0) {

this.invoiceTotal = this.invoiceTotal \* 0.95;

}

下面的注释几乎没有提供任何额外的信息，因此无需注释：

// 发票金额超过1000美圆以上给出 5% 折扣

If(this.invoiceTotal > 1000.0) {

this.invoiceTotal = this.invoiceTotal \* 0.95;

}

### 避免使用行末注释的方法

行末注释，即在一行工作代码结束的地方添加注释，应该慎重使用。因为这些注释很容易干扰代码的可视化结构。当修改行末注释时可能会使行的宽度超过显示区域的宽度导致换行或者需要移动水平滚动条才能看到右边的注释。还有一些程序员试图通过左对齐的方式来改变行末注释的外观，只有当修改代码时才会重新对齐这些注释，其实这只是在浪费时间而已。

有时喜欢将一行注释直接放在被注释的代码前面的注释方式，对于这种方式也有一种例外就是：对于局域变量的描述说明是可以采取行末注释的方式，因为这些行的代码不可能会很长。

### 采用行末注释的方式解释局域变量

假如对一个局域变量的描述非常短，可以考虑将这些注释直接放在代码行的后面。比如：

int cur = 0; //Index of current pattern element

int prev = 0; Index of previous pattern element

不用考虑注释的定位问题，因为它会自动与其他行末注释对齐。

### 创建并使用一套关键字集来标注未解决的问题

创建一系列关键字集用于创建特殊的注释，以便于其他的开发人员在可以用它来发现未解决的问题，这些问题必须在宣布开发结束前彻底得到解决。这种注释内应该包含注释日期和提出问题的人。在同一段代码中应该选择重复率最小的词，比如在下面的例子中，采用两个冒号前后修饰单词“UNRESOLVED”的方式来提高该关键词是唯一的可能性。

// :UNRESOLVED: EBW, 11 July 1999

// Tlhis still does not handle the case where the input overflows the internal buffer!!

While(everMoreInput){

…

}

### 在多重嵌套控制结构中标注结束标志

总的来说，应该尽量避免使用多重嵌套结构，当然，也可以在每重嵌套结束的花括号后面加上行末注释，以增加多重嵌套的可读性。

For(i…) {

For(j…) {

While(…) {

If(…) {

Switch(…) {

…

} //end switch

} // end if

} // end while

} // end for j

} //end for i

### 如果在switch语句中没有break，则在相应的两个case之间添加一行“fall-through”注释

当switch结构中某个case语句后面不需要包含break语句时，可以在下一个case出现之前加一行注释“fall through”表明你的目的，否则别的开发人员可能会认为此处应该是break跳出，或者怀疑你是否忘记写break了。

Switch(command) {

case FASE\_FORWARD:

isFastForward = true;

// Fall through!

case PLAY:

case FORWARD:

isForward = true;

break;

case FAST\_REWIND:

isFastRewind = true;

// Fall through!

case REWIND:

isRewind = true;

break;

…

}

注：两个直接相邻的标签直接无需插入注释！

### 标注empty语句

在循环控制结构中，如while或for循环中，如果循环体为空，则加一行empty注释已表明用意为空。

// Strip leading spaces

while((c = reader.read()) == SPACE);

// Empty!

# 编程规范

## 将代表基础数据类型的类声明为final

一些代表基础类型的简单类，如工程包中定义的复数（ComplexNumber），在工程这个目标领域中会被广泛使用，正因如此，效率可能会成为一个很重要的问题。而将类声明成final类型，则可以直接调用它的成员函数，这样可以大大提高效率。

当然，将类声明成final类型将限制它成为超类（superclass）。然而，很少有人会去派生一个基础数据类。在很多这种例子中，对象组合是重用的一种更有效机制。

## 定义类和方法尽可能短小精悍

短小精悍的类和方法更易于设计、编码、测试、归档、阅读、理解和使用，总的来说，短小精悍的类包含较少的方法，并且代表较简单的概念，它们的接口倾向于展示更好的聚合力。

如果发现一个类或方法看起来太庞大，则考虑重新将这个类或方法写成一些附加的类或方法。

## 将所有成员变量声明为private类型

这样做可以保证成员数据的一致性，因为只有拥有这些成员数据的类才能改变它。访问所有成员变量都需要通过它的成员函数来访问。这样可以减小对象之间的耦合度，从而提高程序的可维护性。

## 使用多态(polymorphism)，不使用isntanceOf

不要根据对象的类型使用instanceof方法来选择对象的行为，因为每次当选择对象的类型发生变化时，都必须修改选择代码。

采用在每个派生类中都实现指定对象的行为，这样，每个客户都可以与不知派生类的情况的基类发生交互。

## 类型安全

### 用Object对象代替一般目的类可以提供静态类型检验

public class Queue {

public void enqueue( Object object) { … }

public Object dequeue() { … }

}

public class OrderQueue {

private Queue queue;

public OrderQueue() {

this.queue = new Queue();

}

public void enqueue( Order order) {

this.queue.enqueue(order);

}

public Order dequeue() {

return (Order)this.queue.dequeue();

}

}

### 将枚举类型封装为类

将枚举类型封装为类可以提供各个枚举子的类型安全检验

public class Color {

private static int count = 0;

public static final Color RED = new Color(count++);

public static final Color GREEN = new Color(count++);

public static final Color BLUE = new Color(count++);

private int value;

private Color(int value){

this.value = value;

}

public Boolean equals(Color other){

return this.value == other.value;

}

public static int Color.count() {

return count;

}

}

Color aColor = Color.RED;

If(anotherColor.equals(aColor)){

…

}

## 语句与表达式

### 用等价的方法替换重复的表达式

写一次代码而且保证只写一次，挑出一些公共的功能并重新打包成一个方法或类。这样可以使得代码更易于学习和理解，使得维护更容易，同时也大大减少了测试工作。

### 在控制结构中尽量使用代码块来代替表达式

java代码块语句提供了一种机制：可以将任何数量的代码块作为一个单一的复合语句对待。代码块语句可以用在任何常规语句使用的地方，包括可以用于java控制结构体类的表达式语句。

当语言控制结构中能使用简单、非块化语句的时候，在这种情况下，应该使用块语句。

当控制结构中使用嵌套语句的时候，使用块语句可以减少常见的模糊性，它提供了一种机制来提升代码的可读性。

下面这段代码就容易引起混淆，因为其缩进排版使得代码看起来else语句应该与第一个if语句关联，但是编译器却将它与第二个if语句关联。在java语言规范中这种情况被称之为“摇摆else问题”，使用块语句可以消除这种问题：

if(x >=0)

if( x > 0) positives();

else //Oops! Actually matches most recent if!

negativeX();

if(x >=0) {

if( x > 0) positives();

}

else {

negativeX();

}

在下面的例子中，上面的代码比下面的代码难于修改，这是因为在不改变现有结构的情况下，你无法新增其它的语句，而下面的代码使用了块语句，修改就非常容易了。

for (int I = n; I >= 0; i--)

for (int j = n; j >= 0 j--)

f(i, j);

// g(i, j); cannot add here!

for (int I = n; I >= 0; i--) {

for (int j = n; j >= 0 j--) {

f(i, j);

g(i, j); // Can add here!

}

}

如果一个控制语句只有一个单一的、平常的语句作为控制体，为了提高可读性，可以把控制体语句与控制语句直接写在一行。把这种情况视为是一种异常。

### 用圆括号清晰地标明运算的顺序

在数学表达式中的运算顺序并不总是很明显的，即便可以肯定这个运算顺序，为了安全起见，最好还是假定别人看起来不一定是很明显的。

// Extraneous but useful parentheses.

Int width = ((buffer \* offset) / pixelWidth) + gap;

### 在switch语句的最后一个case语句块结束的地方加上一个break语句。

下面的switch语句假设不再会有其他的case语句，因此在Y case结束的时候就不需要。break语句：

switch( … ) {

case X:

…

break;

case Y:

…

}

但是，如果需要增加一个新的case语句，而这个增加的人又简单地在最后一个case语句的后面加上新的case语句，而没去注意这个case后面是否有break语句，这样会产生很难察觉的“fall through”错误，举例如下：

switch (…) {

case X:

…

break;

case Y:

… // Oops! Unintended fall- through!

case Z:

…

}

为了避免出现这样的问题，应该总是在最后一个case语句结束的地方写一个break语句，即便是在default语句后边也写上一个break语句：

switch (…) {

case X:

…

break;

case Y:

… // OK! No more fall - through!

break;

case Z:

…

break;

default:

… // Complain about value!

break;

}

注：如果确实某个case后面不需要break语句，应该在相应的位置加上“fall-through”

### 判断两个对象的值是否相等，使用equals()，而不是使用==

许多C++程序员写java程序处理日期何字符串对象时都可能犯这样的错误：

Date today = new Date();

while (date != today) {

…

}

String name;

…

if (name == “Bob”) {

hiBob();

}

在java语言中，“！=”和“==”操作符是用来比较两个对象是否相等，而不是比较两个对象的值是否相等，要比较两个对象的值是否相等，应该使用equals方法：

Date today = new Date();

while (!date.equals(today)) {

…

}

String name;

…

if (“Bob”.equals(name)) {

hiBob();

}

注：与name.equals(“Bob”)表达式不一样，表达式”Bob”.equals(name)在name为null时不会抛出异常。

## 构造函数

### 保证对象的构造总是有效

不允许无效的对象构造。如果一个对象的构造无效，在其有效之前还必须进行进一步的初始化，使用协调多级构造的静态方法。构造函数应该构造一个对象，这样在构造函数执行完后，新的对象就处于有效状态了，将无效构造的构造函数设置为protected或private隐藏起来。

### 在构造函数中不要引用非常量方法

子类可以覆盖非final方法，java在执行派生类的构造函数之前，会根据被构造对象实际的类型调用相应的方法，这就意味着构造器在调用派生类的方法时，派生类可能还处于无效状态，因此，为了防止这样的事件发生，构造器中只能调用final方法。

### 使用嵌套构造器来消除冗余的代码

为了避免写过多冗余构造代码，可以在高级的构造函数中调用低级的构造函数。

下面的代码在两处不同的地方都实现了相同的低级初始化：

class Account {

String name;

double balance;

final static double DEFAULT\_BALANCE = 0.0d;

Account (String name, double balance) {

this.name = name;

this.balance = balance;

}

Account (String name) {

this.name = name;

this.balance = DEFAULT\_BALANCE;

}

}

下面的代码就只在一处进行了低级初始化：

class Account {

String name;

double balance;

final static double DEFAULT\_BALANCE = 0.0d;

Account (String name, double balance) {

this.name = name;

this.balance = balance;

}

Account (String name) {

this (name, DEFAULT\_BALANCE);

}

}

假如在使用断定时，这种方法同样是有用的，因为这样可以减少指定构造函数的参数个数，因此也就减少了对这些参数的有效性检验。

## 异常处理

### 使用未检测到的、运行时异常来报告程序中未预测到的逻辑错误

捕捉和处理运行时异常是可能的，但是，接下来一个严重的结果就是程序终止运行。由于程序的错误经常会抛出运行时异常，如错误的断定、索引出解、除数为零或者引用空指针等。

### 使用可检验的异常处理来报告可能发生的程序错误，但是，在正常的程序运行中，这种是错误是很少发生的。

在正常条件下，一个表明严重问题的可检测异常是很少发生的。调用者必须捕捉这些异常。

### 使用return代码报告可预测性的状态变化

对于可预测的状态变化，可以使用一个return代码，或者调用能返回一个状态的特定方法。这使得代码更具可读性，使控制流更加直截了当。比如，在读取一个文件的时候，在某个点到文件的末尾是可以预测的。

### 转移异常信息的处理

保留所有的异常信息，

try {

for( int i = v.size(); --i >= 0;) {

ostream.println(v.elementAt(i));

}

}catch(ArrayOutOfBounds e) {

// Should never get here

throw new UnexpectedExceptionError(e);

}

### 不要对运行时异常或者错误不做任何处理

打断这种规则将使调试变得很难进行，因为异常信息已经丢失了：

try {

for(int i= v.size(); --i >=0;) {

ostream.println(v.elementAt(i));

}catch( ArrayOutOfBounds e) {

// Oops! We shoule never get here …

// … but if we do, nobody will ever know!

}

}

即使已经写了一个简单的catch处理代码来捕捉未知的异常，至少还是应该打印出栈跟踪错误信息，因为你并不知道在你的软件会发生什么。

try {

for(int i= v.size(); --i >=0;) {

ostream.println(v.elementAt(i));

}catch( ArrayOutOfBounds e) {

// Oops! We shoule never get here …

// … but print a stack trace just in case …

e.printStackTrace();

}

}

### 用一个finally语句块释放资源

程序执行一旦进入try语句块，与之对应的finally语句块肯定也会被执行。可以在finally语句块中释放资源。

public void logSomeStuff() {

OutputStream log = new FileOutputStream(“log”);

…

// could get exception here!

…

log.close();

}

在这个例子中，我们使用一个finally语句块保证当执行结束退出try语句块的时候对象会被关闭。即便是这个线程发生异常退出或者执行了返回语句，finally语句块中的代码还是会被执行。

public void logSomeStuff() {

OutputStream log = null;

try {

OutputStream log = new FileOutputStream(“log”);

}

finally {

if( log != null) {

try {

log.close();

}

catch (IOException ex) {

}

}

}

# 编程技巧

## byte 数组转换到 characters

将 byte 数组转换到 characters：

"Hello world!".getBytes();

## Utility 类

Utility 类（仅仅提供方法的类）应该被申明为抽象的来防止被继承或被初始化。

* 1. **初始化**

下面的代码是一种很好的初始化数组的方法：

objectArguments = new Object[] { arguments };

## 对象构造

### 不要在循环中构造和释放对象

### 处理 String 的时候要尽量使用 StringBuffer 类。

如果是在jdk 1.5的环境下，线程安全变量应该使用StringBuilder来代替StringBuffer

StringBuffer 类是构成 String 类的基础。String 类将 StringBuffer 类封装了起来，（以花费更多时间为代价）为开发人员提供了一个安全的接口。当我们在构造字符串的时候，我们应该用 StringBuffer 来实现大部分的工作，当工作完成后将 StringBuffer 对象再转换为需要的 String 对象。比如：如果有一个字符串必须不断地在其后添加许多字符来完成构造，那么我们应该使用 StringBuffer 对象和她的 append() 方法。如果我们用 String 对象代替 StringBuffer 对象的话，会花费许多不必要的创建和释放对象的 CPU 时间。

## 避免太多的使用 synchronized 关键字

避免不必要的使用关键字 synchronized，应该在必要的时候使用，这是一个避免死锁的好方法。 原则上禁止synchronized关键字嵌套使用。

## 通过对系统数据结构的划分与组织的改进，以及对程序算法的优化来提高空间效率。

说明：这种方式是解决软件空间效率的根本办法。 示例：如下记录学生学习成绩的结构不合理。

typedef unsigned char BYTE;

typedef unsigned short WORD;

typedef struct STUDENT\_SCORE\_STRU BYTE name[8];

BYTE age;

BYTE sex;

BYTE class;

BYTE subject;

float score; } STUDENT\_SCORE;

因为每位学生都有多科学习成绩，故以上结构将占用较大空间。应如下改进（分为两个结构），总的存贮空间将变小，操作也变得更方便。

typedef struct STUDENT\_STRU

{

BYTE name[8];

BYTE age;

BYTE sex;

BYTE class; } STUDENT;

typedef struct STUDENT\_SCORE\_STRU

{

WORD student\_index;

BYTE subject;

float score;

} STUDENT\_SCORE;

## 循环体内工作量最小化。

应仔细考虑循环体内的语句是否可以放在循环体之外，使循环体内工作量最小，从而提高程序的时间效率。 示例：如下代码效率不高。

for (ind = 0; ind < MAX\_ADD\_NUMBER; ind++)

{ sum += ind;

back\_sum = sum; /\* backup sum \*/ }

语句“back\_sum = sum;”完全可以放在for语句之后，如下：

for (ind = 0; ind < MAX\_ADD\_NUMBER; ind++)

{ sum += ind; }

back\_sum = sum; /\* backup sum \*/

变量生存周期最小化，如

int i;

for (i = 0; i < 10; i++) {

…

}

一般应该写成for (int i = 0; i < 10; i++)，如果for循环之后仍然需要使用i变量，如果位置不远，将i变量的生存域外增加一个语句块。如

{

int i;

for (int i = 0; i < 10; i++) {

…

}

if (i == 10) {

…

}

}

## 不应花过多的时间提高调用不很频繁的函数代码效率。

说明：对代码优化可提高效率，但若考虑不周很有可能引起严重后果。

## 在保证程序质量的前提下，通过压缩代码量、去掉不必要代码以及减少不必要的局部和全局变量，来提高空间效率。

## 在多重循环中，应将最忙的循环放在最内层。

说明：减少CPU切入循环层的次数。 示例：如下代码效率不高。

for (row = 0; row < 100; row++)

{

for (col = 0; col < 5; col++)

{

sum += a[row][col];

}

}

可以改为如下方式，以提高效率。

for (col = 0; col < 5; col++)

{

for (row = 0; row < 100; row++)

{

sum += a[row][col];

}

}

## 尽量用乘法或其它方法代替除法，特别是浮点运算中的除法。

说明：浮点运算除法要占用较多CPU资源。 示例：如下表达式运算可能要占较多CPU资源。

#define PAI 3.1416 radius = circle\_length / (2 \* PAI);

应如下把浮点除法改为浮点乘法。

#define PAI\_RECIPROCAL (1 / 3.1416 ) // 编译器编译时，将生成具体浮点数 radius = circle\_length \* PAI\_RECIPROCAL / 2;