[公司名称] | [公司地址]

[文档标题]

[文档副标题]

|  |
| --- |
| **<公司名称>** |

<规范文档>

Java语言编程规范

版本 <1.0>

修订历史记录

|  |  |  |  |
| --- | --- | --- | --- |
| **日期** | **版本** | **说明** | **作者** |
| 2016-11-02 | 1.0 | 新建文档 | 蒙斌 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**Java语言编程规范**

# 1. 简介

本文提供一整套编写高效可靠的 Java 代码的标准、约定和指南，作为本次开发Java编码规范，以指导今后的Java编程工作，JavaScript编程也采用该规范。它们以安全可靠的软件工程原则为基础，使代码易于理解、维护和增强。而且，通过遵循这些程序设计标准，显著提高 Java 软件开发者的生产效率会。经验证明，若从一开始就花时间编写高质量的代码，则在软件开发阶段，对代码的修改要容易很多。最后，遵循一套通用的程序设计标准将带来更大的一致性，使软件开发团队的效率明显提高。所以我们必须遵守并完善它。

# 2. 程序设计标准

Java 的程序设计标准很重要，原因在于它将提高开发团队各成员的代码的一致性。一致性的提高会使代码更易理解，这意味着它更易开发和维护。从而降低了应用程序的总开发成本。

## 2.1 代码格式约定

### 2.1.1 {}使用约定

在Java编程中{}常用的格式有两种

public static void main(String[] args)

{

……

}

public static void main(String[] args){

……

}

第一种风格的可读性是显然的，所以我们选择第一种作为编码约定。如果有编辑环境自动生成的代码，格式应该设置为第一种方式，如果不能设置（后续新增编码模板），我们应该保证我们手写的代码为第一种风格。

### 2.1.2 缩进约定

代码缩进我们约定为一个tab键(降tab长度设置为4个空格长度)，注意并非4个空格。

### 2.1.3 空格的使用约定

下列情况应该使用空格

**1． 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。

**2．空白应该位于参数列表中逗号的后面。**

**3．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");

**4．for语句中的表达式应该被空格分开，例如：**

for (expr1; expr2; expr3)

**5．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);

## 2.2 命名约定

我们将在整个标准中讨论命名约定，所以让我们先讨论几个基本点：

1. **使用可以准确说明变量/字段/类的完整的英文描述符。**

例如，采用类似 firstName，grandTotal 或 CorporateCustomer 这样的名字。虽然象 x1，y1 或 fn 这样的名字很简短，输入起来容易，但是我们难以知道它们代表什么、结果是什么含义，因而使代码难以理解、维护和改进。

1. **采用该领域的术语。**

如果用户称他们的“客户” (clients) 为“顾客” (customers)，那么就采用术语 Customer 来命名这个类，而不用 Client。许多程序开发者会犯的一个错误是，不去使用工业或领域里已经存在着很完美的术语时，却生造出一些普通词汇。

1. **采用大小写混合，提高名字的可读性。**

一般应该采用小写字母，但是类和接口的名字的首字母，以及任何中间单词的首字母应该大写。

1. **尽量少用缩写，但如果一定要使用，就要谨慎地使用。**

这意味着应该保留一个标准缩写的列表，明智地从中选取，并且在使用时保持一致。例如，想对单词“number”采用缩写，那么可从 nbr，no 或者 num 中选取一个，说明一下采用了哪一个（具体是哪个倒无所谓），并且只使用这一种形式。

1. **避免使用长名字（最好不超过 15 个字母）。**

虽然 PhysicalOrVirtualProductOrService 看起来似乎是个不错的类名，但是这个名字太长了，应该考虑重新给它起个短一点的名字，比如象 Offering。

1. **避免使用相似或者仅在大小写上有区别的名字。**

例如，不应同时使用变量名 persistentObject 和 persistentObjects，以及 anSqlDatabase 和 anSQLDatabase。

1. **避免使用下划线作为名字的首末字母。**

以下划线为首末字母的名字通常为系统保留，除预处理定义之外，一般不用作用户命名。更重要的是，下划线经常造成麻烦而且难输入，所以尽量避免使用。

## 2.3 注释约定

我们还会对注释约定进行讨论，所以，我们先谈谈一些基本点：

1. **注释应该增加代码的清晰度。**

代码注释的目的是要使代码更易于被同时参与程序设计的开发人员以及其他后继开发人员理解。

**2． 如果你的程序不值得注释，那么它也很可能也不值得运行。**

**3． 避免使用装饰性内容，也就是说，不要使用象广告横幅那样的注释语句。**

二十世纪六十年代和七十年代，COBOL 程序员们养成了画框的习惯，典型的是用星号将他们的内部注释圈起来。当然，这给他们的艺术创造欲一个发泄方式，但是坦白地说，这只是在大量浪费时间，并不能给最终的产品增加丝毫价值。要写的是清晰的代码，不是外表可爱的代码。此外，由于有些字体的显示和打印是成比例的，但有些又不是，所以无法将那些框排整齐。

**4． 保持注释的简洁。**

最好的注释应该是简单明了的注释。注释不必洋洋洒洒，只需提供足够的信息，使别人能够理解你的代码。

**5． 先写注释，后写代码。**

写代码注释的最好方法是在写代码之前就写注释。这使你在写代码之前可以想想代码的功能和运行。而且这样确保不会遗漏注释。另一种方法是边写代码边写注释。因为注释可以使代码更易理解，所以在程序开发的过程中，也可以利用这一点。如果打算花些时间写注释，那么至少你应从这个过程中获得些什么。

**6． 注释信息不仅要包括代码的功能，还应给出原因。**

例如，下面例 1 中的代码显示金额在 $1,000 以上（包括 $1,000）的定单可给予 5% 的折扣。为什么要这样做呢？难道有一个商业法则规定大额定单可以得到折扣吗？这种给大额定单的特殊是有时限的呢，还是一直都这样？最初的程序设计者是否只是由于慷慨大度才这样做呢？除非它们在某个地方（或者是在源代码本身，或者是在一个外部文档里）被注释出来，否则你不可能知道这些。

例 1.1

if (grandTotal >= 1000.00)

{

grandTotal = grandTotal \* 0.95;

}

### 2.3.1 Java 注释语句类型

Java 有三种注释语句风格：

以 /\*\* 开始， \*/ 结束的文档注释；以 /\* 开始，以 \*/ 结束的C语言风格注释；以及以// 开始，代码行末尾结束的单行注释。

下表是对各类注释语句建议用法的一个概括，也给出了几个例子。

|  |  |  |
| --- | --- | --- |
| 注释语句类型 | 用法 | 示例 |
| 文档注释 | 文档注释 在接口、类、成员函数和字段声明之前紧靠它们的位置用文档注释进行说明。文档注释由 javadoc 处理，为一个类生成外部注释文档. | /\*\*  \* <p>标题： </p>  \* <p>功能： </p>  \* <p>版权： Copyright (c) 2003</p>  \* <p>公司: </p>  \* 作者：  \* <p>版本： 1.0</p>  \*/ |
| C语言风格注释 | 采用 C 语言风格的注释语句将无用的代码注释掉。保留这些代码是因为用户可能改变想法，或者只是想在调试中暂时不执行这些代码。 | /\*  这部分代码已被它前面的代码替代，所以于 1999 年6 月 4 日被 B. Gustafsson 注释掉。如果两年之后仍未用这些代码，将其删除。  . . . （源代码）  \*/ |
| 单行注释 | 在成员函数内部采用单行注释语句对业务逻辑、代码片段和临时变量声明进行说明。 | // 因为让利活动  // 从 1995 年 2 月开始，  // 所以给所有超过 $1000 的  // 发货单 5% 的折扣。 |

### 2.3.2 避免行末注释

强烈反对采用行内注释，即在一行的末尾加上注释。

这种注释必须在代码的右端对齐，这样才能避免代码结构看起来混乱。结果，这些注释的格式难以划一。“如果你使用了很多这样的注释，则要花时间去将它们排列整齐。这些时间并没有花在更多地了解代码上，而完全花在了敲击空格键和制表符这种冗长乏味的工作上。”

行末注释也难以维护。因为当该行程序的代码加长时，它们会将这些注释挤出该行，如果你将它们排齐了，你不得不对余下的注释做同样的工作。

### 2.3.3 快速浏览 javadoc

Sun 公司的 Java Development Kit (JDK) 中有一个名为 javadoc 的程序。它可以处理 Java 的源代码文件，并且为 Java 程序产生 HTML 文件形式的外部注释文档。Javadoc 支持一定数目的标记，标识注释文档中各段起始位置的保留字。详情请参考 JDK javadoc 文档。

|  |  |  |
| --- | --- | --- |
| **标记** | **用于** | **目的** |
| @author name | 类、 接口 | 说明特定某一段程序代码的作者。每一个作者各有一个标记。 |
| @deprecated | 类、 成员函数 | 说明该类的应用程序编程接口 (API) 已被废弃，因此应不再使用。 |
| @exception name description | 成员函数 | 说明由成员函数发出的异常。一个异常采用一个标记，并要给出异常的完整类名。 |
| @param name description | 成员函数 | 用来说明传递给一个成员函数的参数，其中包括参数的类型/类和用法。每个参数各有一个标记。 |
| @return description | 成员函数 | 若成员函数有返回值，对该返回值进行说明。应说明返回值的类型/类和可能的用途。 |
| @since | 类、成员函数 | 说明自从有 JDK 1.1 以来，该项已存在了多长时间。 |
| @see ClassName | 类、接口、成员函数、字段 | 在文档中生成指向特定类的超文本链接。可以并且应该采用完全合法的类名。 |
| @see ClassName#member functionName | 类、接口、成员函数、字段 | 在文档中生成指向特定成员函数的超文本链接。可以并且应该采用完全合法的类名。 |
| @version text | 类、接口 | 说明特定一段代码的版本信息。 |

你注释代码的方式很大地影响着你的工作效率以及所有维护改进代码的后继开发者的工作效率。在软件开发过程中及早注释代码，会促使你在开始撰写代码之前仔细考虑这些代码，从而带来更高的工作效率。而且，当你重新阅读数天前或者数星期前所写的代码时，你可以很容易地判断出当时你是怎么想的，因为这一切都有记录。

# 3. 成员函数标准

## 3.1 命名成员函数

成员函数的命名应采用完整的英文描述符，大小写混合使用：所有中间单词的第一个字母大写。成员函数名称的第一个单词常常采用一个有强烈动作色彩的动词。

示例：

openAccount()

printMailingLabel()

save()

delete()

这种约定常常使人一看到成员函数的名称就能判断它的功能。虽然这种约定要使开发者多做一些输入的工作，因为函数名常常较长，但是回报是提高代码的可理解性。

### 3.1.1 命名存取成员函数

在后续章节中，我们将更详细地讨论获取和存放字段值（字段/属性）的存取成员函数。下面将概括一下存取函数的命名约定。

#### 3.1.1.1 获取函数

获取函数作为一个成员函数，返回一个字段的值。除了布尔字段之外，应采用 get 作为字段的前缀；布尔字段采用 is 作为前缀。

示例：

getFirstName()

getAccountNumber()

isPersistent()

isAtEnd()

遵循这个命名约定，显然，成员函数将返回对象的字段，布尔型的获取函数将返回布尔值“真”或者“假”。

这个标准的另一个优点是：它遵循 beans development kit (BDK) 对获取成员函数采用的命名约定 。它的一个主要的缺点是 get 是多余的，需要额外的录入工作。

获取函数的另一种命名约定。Has 和 Can基于正规英文约定的一个可行的取代方法是，用 has 或者 can 来代替布尔型获取函数的 is 前缀。

例如，形如 hasDependents() 和 canPrint() 的获取函数，读起来就发现它们意义更加明确。这种方法存在的问题是 BDK 目前还不支持这种命名方法。三可以将 isBurdenedWithDependents() 和 isPrintable() 这些成员函数重新命名。

#### 3.1.1.2 设置函数

设置函数，也叫变值函数，是可以修改一个字段值的成员函数。无论何种字段类型，都要在字段名的前面加上 set 前缀。

示例：

setFirstName(String aName)

setAccountNumber(int anAccountNumber)

setReasonableGoals(Vector newGoals)

setPersistent(boolean isPersistent)

setAtEnd(boolean isAtEnd)

按照这种命名约定，显然是一个成员函数设定一个对象的字段值。这个标准的另一个优点是：它遵循 beans development kit (BDK) 对设置函数采用的命名约定。它的一个主要的缺点是 set 是多余的，需要额外的录入。

## 3.2 命名构造函数

构造函数是在一个对象初次生成时，完成所有必需的初始化的成员函数。构造函数与它所属类的名字总是相同的。例如，类 Customer 的构造函数是 Customer()。注意大小写一致。

示例：

Customer()

SavingsAccount()

PersistenceBroker()

这个命名约定由 Sun 公司设定，必须严格遵守。

## 3.3 成员函数的可见性

良好的程序设计应该尽可能减小类与类之间耦合，所遵循的经验法则是：尽量限制成员函数的可见性。如果成员函数没必要公有 (public)，就定义为保护 (protected)；没必要保护 (protected)，就定义为私有 (private)。

|  |  |  |
| --- | --- | --- |
| **可见性** | **说明** | **正确用法** |
| **public** | 公有成员函数可被任何其它对象和类的成员函数调用。 | 当该成员函数必须被该函数所在的层次结构之外的其他对象和类在访问时。 |
| **protected** | 被保护的成员函数可被它所在的类或该类的子类的任何成员函数调用。 | 当该成员函数提供的行为被它所在类的层次结构内部而非外部需要时。 |
| **private** | 私有成员函数只可以被该类所在的其它成员函数调用，该类的子类不可以调用。 | 当该成员函数所提供的行为明确针对定义它的类时。私有成员函数常常是重新分配要素的结果。重新分配要素又叫“重组”，指类内其它成员函数封装某一个特定行为的做法。 |

## 3.4 注释成员函数

如何注释一个成员函数常常成为判断函数是否可被理解，进而可维护和可扩展的决定性因素。

### 3.4.1 成员函数的函数头

每一个 Java 成员函数都应包含某种称之为“成员函数文档”的函数头。这些函数头在源代码的前面，用来记录所有重要的有助于理解函数的信息。这些信息包含但不仅仅局限于以下内容：

1. **成员函数做什么以及它为什么做这个。**

通过给一个成员函数加注释，让别人更加容易判断他们是否可以复用代码。注释出函数为什么做这个可以让其他人更容易将你的代码放到程序的上下文中去。也使其他人更容易判断是否应该对你的某一段代码加以修改（有可能他要做的修改与你最初为什么要写这一段代码是相互冲突的）。

1. **哪些参数必须传递给一个成员函数。**

还必须说明，如果带参数，那么什么样的参数必须传给成员函数，以及成员函数将怎样使用它们。这个信息使其他程序员了解应将怎样的信息传递给一个成员函数。在 (第 1.4.2 节“快速浏览 javadoc”) 中讨论的 javadoc @param 标识便用于该目的。

1. **成员函数返回什么。**

如果成员函数有返回值，则应注释出来，这样可以使其他程序员正确地使用返回值/对象。在 (第 1.4.2节“快速浏览 javadoc”) 里讨论的 javadoc @return 标识便用于此目的。

1. **已知的问题。**

成员函数中的任何突出的问题都应说明，以便让其他程序开发者了解该成员函数的弱点和难点。如果在一个类的多个成员函数中都存在着同样的问题，那么这个问题应该写在类的说明里。

1. **任何由某个成员函数抛出的异常。**

应说明成员函数抛出的所有异常，以便使其他程序员明白他们的代码应该捕获些什么。在 (第 1.4.2节“快速浏览 javadoc”) 中讨论的 javadoc @exception 标识便用于此目的。

1. **可见性决策。**

如果你觉得你对于一个成员函数可见性的选择会遭到别人的质疑，例如可能你将一个成员函数设为公共的，但是却没有任何对象调用该成员函数，那么应说明你的决定。这将帮助其他开发者了解你的想法，使他们不必浪费时间操心考虑你为什么要选择一种有疑问的东西。

1. **成员函数是如何改变对象的。**

若一个成员函数修改了一个对象，例如一个银行帐户的成员函数 withdraw() 修改了帐户余额，那么就需要说明。这种信息必须给出，使其他 Java 程序员能准确地知道一个成员函数调用将如何影响目标对象。

1. **避免使用含有信息的函数头。**

比如说作者、电话、创建和修改日期、单元（或者文件名）的位置，因为这些信息很快就会过时。将版权所有者信息放到单元的最后。例如，读者不会想要翻过两三页诸如“版权所有”等对理解程序毫无帮助且（或）不提供任何编程信息的文本。避免使用垂直滚动条或者关闭的文本框或对话框，这些东西只会增加视觉干扰，而且较难保持一致。采用一个配置管理工具来保存单元历史。

1. **如何在适当情况下调用成员函数的例子。**

最简单的确定一段代码如何工作的方法是看一个例子。考虑包含一到两个如何调用成员函数的例子。

**10．可用的前提条件和后置条件。**

前提条件是指一个成员函数可正确运行的限制条件；后置条件是指一个成员函数执行完以后的属性或声明。前提条件和后置条件以多种方式说明了在编写成员函数过程中所做的假设，精确定义了一个成员函数的应用范围。

**11．所有并行事件。**

对众多程序开发者来说，并行性是一个新而复杂的概念；对有经验的并行性程序开发者来说，并行性也是一个老但却复杂的课题。最终结果是，如果应用了 Java 的并行编程特性，那么应在程序中详细地将其注释出来。建议，当一个类既包含了同步也包含了非同步的成员函数时，必须注释出成员函数依赖的执行上下文，尤其是当函数可被无约束访问时。这样可以让其他开发者安全地使用你的成员函数。当一个采用了 Runnable 接口的类的设置函数（即可更新一个字段的成员函数）没有同步时，应说明这样做的理由。最后，如果覆盖或重载一个成员函数，并且修改了它的同步性时，也应说明理由。

仅当注释增加代码的清晰度时，才应加上注释。对于每个成员函数，并非要说明以上所有部分，因为对于每一个成员函数来说，并不是以上所有的部分都适用。但是，对于所写的每个成员函数要说明以上的部分内容。

### 3.4.2 内部注释

除成员函数注释以外，在成员函数内部还需加上注释语句来说明你的工作。目的是使成员函数更易理解、维护和增强。

内部注释应采用两种方式：C 语言风格的注释 (/\* 和 \*/) 和单行注释 (//)。

正如上述所讨论的，应认真考虑给代码的业务逻辑采用一种风格的注释，给要注释掉的无用代码采用另外一种风格的注释。建议对业务逻辑采用单行注释，因为它可用于整行注释和行末注释。采用 C 语言风格的注释语句去掉无用的代码，因为这样仅用一个语句就可以容易地去掉几行代码。此外，因为 C 语言风格的注释语句很象文档注释符。它们之间的用法易混淆，这样会使代码的可理解性降低。所以，应尽量减少使用它们。

在函数内，一定要说明：

1. **控制结构。**

说明每个控制结构，例如比较语句和循环。你无须读完整个控制结构内的代码才判断它的功能，而仅需看看紧靠它之前的一到两行注释即可。

1. **代码做了些什么以及为什么这样做。**

通常你常能看懂一段代码做了什么，但对于那些不明显的代码，你很少能判断出它为什么要那样做。例如，看完一行代码，你很容易地就可以断定它是在定单总额上打了 5% 的折扣。这很容易。不容易的是为什么要打这个折扣。显然，肯定有一条商业法则说应打折扣，那么在代码中至少应该提到那条商业法则，这样才能使其他开发者理解你的代码为什么会是这样。

1. **局部变量。**

虽然我们在第 4 章将仔细讨论这一点，在一个成员函数内定义的每一个局部变量都应在它代码的所在行声明，并且应采用一个行内注释说明它的用法。

1. **难或复杂的代码。**

若发现不能或者没有时间重写代码，那么应将成员函数中的复杂代码详细地注释出来。一般性的经验法则是，如果代码并非显而易见的，则应说明。

1. **处理顺序。**

如果代码中有的语句必须在一个特定的顺序下执行，则应保证将这一点注释出来。没有比下面更糟糕的事了：你对一段代码做一点简单的改动，却发现它不工作，于是花了几个小时查找问题，最后发现原来是搞错了代码的执行顺序。

在闭括号后加上注释。常常会发现你的控制结构内套了一个控制结构，而在这个控制结构内还套了一个控制结构。虽然应该尽量避免写出这样的代码，但有时你发现最好还是要这样写。问题是闭括号 } 应该属于哪一个控制结构这一点就变得混淆了。一个好消息是，有一些编辑器支持一种特性：当选用了一个开括号后，它会自动地使相应得闭括号高亮显示；一个坏消息是，并非所有的编辑器都支持这种属性。

我发现通过将类似 //end if，//end for，//end switch，& 这样的注释加在闭括号所在行的行后，可以使代码更易理解。

## 3.5 编写清晰整洁的代码的技巧

这一部分讲述几个技巧，这些技巧有助于区分专业软件开发者和蹩脚代码编写者。这些技巧是：

给代码加上注释

给代码分段

使用空白

遵循 30 秒条规则

说明消息发送的顺序

写短小单独的命令行

### 3.5.1 给代码加上注释

记住：如果你的代码不值得注释，那么它就不值得保留。当正确地使用了本文提到的注释标准和方针，就可以大幅度地提高代码质量。

### 3.5.2 让代码分段/缩进

一种提高代码可读性的方法是给代码分段，换句话说，就是在代码块内让代码缩进。所有在括号 { 和 } 之内的代码，构成一个块。基本思想是，块内的代码都应统一地缩进去一个单位。这个单位在这里定义为长度为4的一个tab;

Java 的约定似乎是开括号放在块的所有者所在行的后面，闭括号应缩进一级。在指出的很重要的一点是，你所在的机构应选取一个缩进风格并始终使用这种风格。采用与你的 Java 开发环境所生成的代码一样的缩进风格。

### 3.5.3 在代码中使用空白。

在 Java 代码中加入几个空行，也叫空白，将代码分为一些小的、容易理解的部分，可以使它更加可读。建议采用一个空行来分隔代码的逻辑组，例如控制结构，采用两个空行来分隔成员函数定义。没有空白的代码很难读，很难理解。

### 3.5.4 遵循 30 秒条法则

其他的程序员应能在少于 30 秒钟的时间内完全理解你的成员函数，理解它做什么，为什么这样做，它是怎样做的。如果他们做不到，说明你的代码太难维护，应加以改进。30 秒钟，明明白白。一个好的经验法则是：如果一个成员函数一个屏幕装不下，那么它就很可能太长了。

### 3.5.5 写短小单独的命令行

每一行代码只做一件事情。在依赖于穿孔卡片的计算机发展的早期，想让一行代码完成尽量多的功能的想法是可以理解的。若想在一行里做多件事情，就会使代码难于理解。为什么要这样呢？我们应使代码尽量容易理解，从而更容易维护和改进。正如同一个成员函数应该并且只能做一件事一样，一行代码也只应做一件事情。

此外，应让代码在一个屏幕内可见。也不应向右滚动编辑窗口来读取一整行代码，包括含有行内注释语句的代码。

### 3.5.6 说明运行顺序

提高代码可读性的一个相当简单的方法是使用圆括号 (parenthesis，又叫“round brackets”) 来说明 Java 代码运行的准确顺序。如果为了理解你的源码而必须了解编程语言的操作顺序，那么这说明源码中一定有什么重要的东西做的不对。这大多是在 AND 或者 OR 其它几个比较关系处产生的逻辑比较上的问题。注意：如果你象前文所建议的那样，采用短小单独的命令行，那么就不会产生这个问题。

# 4. 字段/属性的标准

field 这个词在这里指的是字段，Beans Development Kit (BDK) 叫它“属性”。字段是说明一个对象或者一个类的一段数据。字段可以是象字符串或者浮点数这样的基本数据类型，也可以是一个对象

例如一个消费者或者一个银行帐户。

## 4.1 命名字段

应采用完整的英文描述符来命名字段，以便使字段所表达的意思一目了然。象数组或者矢量这样是集合的字段，命名时应使用复数来表示它们代表多值。

示例：

firstName

zipCode

unitPrice

discountRate

orderItems

## 4.2 命名组件（部件）

应采用完整的英文描述符命名组件（接口部件），名字的后缀是组件类型名。这让你容易区分一个组件的目的和它的类型，容易在一个表里找到各个组件（许多可视化的编程环境在一个 Applet 程序或者一个应用程序中提供了一个所有组件的列表。如果所有名字都是类似于 button1， button2, & 这样的话，很容易混淆）。

示例：

okButton

customerList

fileMenu

newFileMenuItem

命名组件的另一种方法。匈牙利符号

“匈牙利符号” 是基于字段应按照以下方法命名的原则：xEeeeeeEeeeee，其中 x 指组件类型，EeeeeEeeeee 是完整的英文描述符。

示例：

pbOk

lbCustomer

mFile

miNewFile

这个方法的主要优点是，这是一个通用于 C++ 代码中的工业标准，已经有许多人在遵守它。此外，开发者能快速地从变量的名字上来判断它的类型和用法。主要的缺点是，当有许多同一类型的部件时，前缀符号变得很笨重；而且，违背了采用完整的英文描述符的命名约定。

命名组件的另一种方法。匈牙利符号后缀

这基本上是其他两种方式的组合，生成的名字如 okPb，customerLb，fileM 和 newFileMi。主要优点是组件名说明了组件类型，并且同一类的组件不在一起，而按照字母顺序排列。主要的缺点仍是未采用完整的英文描述符，它偏离了规范，使标准难以被记住。

设定组件命名标准。无论使用哪种约定，都要生成一个“正式”部件名列表。例如，当命名按钮时，是用 Button 或是 PushButton，b 或是 pb？生成一个列表，让组里的每一个 Java 开发者都得到它。

### 4.2.1 命名常量

Java 中，常量，即不变的值，一般用类的静态常量字段来实现。公认的约定是，采用完整的英文大写单词，在词与词之间用下划线连接。

示例：

MINIMUM\_BALANCE

MAX\_VALUE

DEFAULT\_START\_DATE

这个约定的主要优点是，它有助于区分常量和变量。在本文后面的章节中我们将看到，如果用定义获取函数返回常量值的方法来取代定义常量，代码的可适应性和可维护性都会大大提高。

### 4.2.2 命名集合

一个集合，例如数组和矢量，应采用复数命名来表示队列中存放的对象类型。命名应采用完整的英文描述符，名字中所有非开头的单词的第一个字母应大写。

示例：

customers

orderItems

aliases

这种约定的主要优点是有助于区分表示复数值（集合）和单值（非集合）的字段。

## 4.3 字段可见性

当字段被声明为 protected 类型时，子类中的成员函数可能会直接访问它，有效地提高了类内层次结构的耦合性。这使类更难维护和加强，所以应该尽量避免。字段不应被直接访问，而应采用存取成员函数（参见下文）访问。

|  |  |  |
| --- | --- | --- |
| **可见性** | **说明** | **正确用法** |
| **public** | 一个公共字段可被任何其他对象或者类中的成员函数访问。 | 不要让字段公有。 |
| **protected** | 被保护的字段可被它声明时所在的类及该类的子类的所有成员函数访问。 | 不要让字段被保护。 |
| **private** | 私有字段只可以被它声明时所在的类的其它成员函数调用，该类子类中的函数不可以调用。 | 所有的字段都应置为私有，由获取和设置成员函数（存取函数）访问。 |

对于那些非长期性的字段（它们不被永久保留），应将它们标注为 static 或 transient。使它们与 BDK 的约定一致。

**不要“隐藏”名字**

名字隐藏是指给局部变量、参数或者字段所取的名字，与另一个更大范围内定义的变量、参数或者字段的名字相同（或相似）。

例如，如果把一个字段叫做 firstName ，就不要再生成一个局部变量或者参数叫做 firstName，或者任何类似的名字，如 firstNames 或 fistName。名字隐藏会使代码难于理解，并容易产生问题。因为你或者其他开发者在修改代码时，会误读代码，而错误又很难发现。

## 4.4 注释一个字段

所有的字段都应很好地加以注释，以便其他开发者理解它。要想有效地注释，以下的部分需要说明：

1. **字段的说明。**

需说明一个字段，才能使人了解如何使用它。

1. **注释出所有采用的不变量。**

字段中的不变量是指永远为“真”的条件。例如，字段 dayOfMonth 的不变量可能是它的值只能在 1 到 31 之间（显然，可以用基于某一年里的某个月份来限制这个字段值，使其变的更加复杂）。通过说明字段值的限制条件，有助于定义重要的业务规则，使代码更易理解。

**3. 示例。**

对于那些有复杂业务规则与之相关联的字段，应提供几个例子，使它们容易理解。一个例子常象一幅画：它抵得上几千个词语。

**4． 并行事件。**

对众多程序开发者来说，并行性是一个新而复杂的概念；事实上，即使对有经验的并行程序开发者来说，并行性也是一个老但却复杂的课题。最终结果是，如果应用了 Java 的并行编程特性，那么你应在程序中详细地注释出来。

1. **可见性决策。**

如果声明了一个非私有字段，则应说明为什么要这样做。字段的可见性在上文中（第 3.2 节“字段的可见性”）讨论了，支持封装的存取成员函数的用法将在下文（第 3.4 节“存取成员函数的使用”）中讨论。总的来说，最好能有一个好的理由解释为什么不将变量声明为私有类型。

## 4.5 使用存取成员函数

除了满足命名约定之外，适当地使用存取成员函数，即提供更新或访问字段值的成员函数，可以实现字段的可维护性。存取成员函数有两种：设置函数（也叫变化函数）和获取函数。设置函数修改变量的值，获取函数取到变量的值。

虽然存取成员函数往往会增加代码费用，但是现在 Java 编译器使用时已被优化，所以存取成员函数会增加代码费用这一点已不再正确。存取函数帮助隐藏了类的具体实现细节。一个变量仅能访问两个控制点：设置函数和获取函数。让需修改的点最小化，增加了类的可维护性。Java 代码的优化将在（第 7.3 节“优化 Java 代码”）中讨论。

* **你的机构能改进的最重要的标准之一是存取函数的使用。**

一些开发者不愿使用存取成员函数的原因是，他们不想多输入几个键（例如，对于一个获取函数，除了字段名之外，还必须输入 in， get 和()）。最主要的是，采用存取函数后增加的可维护性和可扩展性，远远超过输入这些字符所做的工作。

* **存取函数是唯一可以访问字段的地方。**

正确使用存取成员函数的关键概念是：成员函数中，只有存取成员函数可以直接访问字段。的确，定义字段的类内的其它成员函数可能可以直接访问该私有字段，但是这样做会增加类内的耦合，所以不应这样。

### 4.5.1 为什么使用存取函数?

“好的程序设计试图将程序部件与不必要、未计划或者不需的外部影响分隔开来。访问修饰语句（存取函数）给编程语言控制这一类的接触提供了一种清晰并可检验的方法。”

存取成员函数通过以下方法提高类的可维护性：

1. **更新字段。**

每个字段只有几个单点要更新，这使得修改和检测都很容易。换句话说，字段已被封装。

1. **获得字段的值。**

你完全控制着字段应怎样被访问以及被谁访问。

1. **获取常量名和类名。**

在获取函数中封装常量值和类名，当这些值或名字改变时，只需更新获取函数内的值，而并非常量或者名字被使用处的每一行代码。

1. **初始化字段。**

采用滞后初始化 (lazy initialization) 保证字段总能被初始化，并且只在需要时才初始化。

1. **减少类与子类之间的耦合。**

当子类通过它们相应的存取成员函数访问被继承的字段时，它可以不影响它的任何子类，而只修改超类字段的实现方式，这样有效地减少了超类与子类之间的耦合。存取函数减少了那种一旦超类被修改就会波及子类的“脆弱基类”的风险。

1. **将变化封装到字段中。**

如果一个或者多个字段的业务规则变化了，可以只潜在地修改存取函数，就同样可以提供规则变化之前的功能。这一点使你很容易响应新的业务规则。

简化并行事件。指出，如果采用了基于字段值的 waits 语句，那么设置成员函数提供了一个位置可包含 notifyAll。这让转向并行解决方案更加容易。

名字隐藏不再是一个大问题。虽然应该避免名字隐藏，即避免赋给局部变量一个与字段相同的名字，但是始终通过存取函数访问字段意味着可以给局部变量任何你想取的名字。不必担心字段名的隐藏，因为你无论如何都不会直接访问它们。

**当不用存取函数时：**唯一可能你不想用存取函数的时候是当执行时间最重要时。但是，这实际上很少见，因为应用程序耦合性的增加会平衡掉这样做之后赢得的好处。

### 4.5.2 命名存取函数

获取成员函数应在名字中加上 get + 字段名，除非字段表示的是一个布尔值（“真”或者“假”），这时获取函数名中应加上 is + 字段名。无论何种字段类型，设置成员函数应在名字中加上 set + 字段名 [GOS96]，[DES97]。

注意字段名始终采用大小写混合，所有单词的第一个字母要大写。

命名约定在 JDK 中被始终使用，在 beans development 中也必须使用。

**Examples:**

|  |  |  |  |
| --- | --- | --- | --- |
| **字段** | **类型** | **获取函数名** | **设置函数名** |
| firstName | 字符串 | getFirstName() | setFirstName() |
| address | 地址对象 | getAddress() | setAddress() |
| persistent | 布尔值 | isPersistent() | setPersistent() |
| customerNo | 整型 | getCustomerNo() | setCustomerNo() |
| orderItems | OrderItem的对象数组 | getOrderItems() | setOrderItems() |

### 4.5.3 存取函数的高级技术

存取函数不仅仅局限在获取和设置实例的字段值时使用。这一节讨论如何将存取函数应用于以下方面，以提高代码的适应性：

* 初始化字段值
* 访问常量值
* 访问集合
* 同时访问几个字段

**滞后初始化**

在访问变量前需要将其初始化。对于初始化有两种想法：当对象生成时初始化所有的变量（传统方法）或者在第一次使用变量时进行初始化。第一种方式在对象初次创建时调用特殊的成员函数，这些成员函数叫做构造函数。虽然这种方法可行，但是它被证明常常产生错误。当增加一个新的变量时，你很容易就会忘记更新构造函数（另一种称为滞后初始化的方法是，字段由它们的获取函数初始化。如下文所示（注意在获取函数内是如何采用一个设置函数的）。

注意：成员函数检查支行个数是否为零，若是则将个数设置为适当的默认值。

/\*\* 返回支行号，支行号

\* 是整个帐号的最左边四位数字。

\* 帐号的格式是 BBBBAAAAAA。

\*/

protected int getBranchNumber()

{

if( branchNumber == 0)

{

// 默认支行号是 1000，

// 它是 Bedrock 城市中心的一个主要支行

setBranchNumber(1000);

}

return branchNumber;

}

对于在数据库中实际存放的是其它对象的字段，常采用滞后初始化。

例如，当生成一个新的存货清单项时，不必从默认的数据库中获得存货项的类型。而仅仅是在第一次访问该项时，才需用滞后初始化方式设定它的值。这样当需要时，只需从数据库中读出存货项类型对象即可。这种方法对于那些有不常被访问的字段的对象有好处。

如果你不打算用这些东西，为什么花精力从固定存取中获取它们呢？

只要获取函数采用了滞后初始化方式，就应该象上文例子那样说明为什么采用那个默认值。这样做除去了代码中字段是如何被应用的那层神秘，使代码的可**维护性和可扩展性**都得到了提高。

### 4.5.4 常量存取函数

Java 中的代码模式是将常量值作为静态常量字段。这种方式对于“常量”是可以理解的，因为它们是固定的。例如，布尔类使用“真”和“假”这两个静态最终字段，它们代表这个类的两个实例。DAYS\_IN\_A\_WEEK 这个常量的值可能永不会改变，这一点也可以理解。

但是，随时间流逝，许多事物“常量”因为业务规则的变化而发生变化。

例如：Archon Bank of Cardassia (ABC) 一直坚持若要获取利息，一个帐户至少应有 $500 的余额。要实现这一点，我们可以在计算利息的成员函数的类 Account 中增加一个叫做 MINIMUM\_BALANCE 的静态字段。这虽然可行，但却不灵活。如果业务规则变化为不同种类的帐户有不同的最小余额限制，比如说，储蓄帐户的最小余额为 $500 ，支票帐号的最小余额为 $200，那么又该怎么办呢？又如果业务规则变化为第一年要保持 $500 的最小余额，第二年要保持 $400 的最小余额，第三年要保持 $300 的最小余额，这又应该怎么办呢？要是规则变为夏季应有 $500 ，但冬季应有 $250 呢？可能所有这些规则的组合在不久的将来就会采用。

要指出的一点是，将常量作为字段来使用不够灵活，一个好得多的办法是将常量作为获取函数来处理。上例中，采用一个叫做 getMinimumBalance() 的静态成员函数比一个叫做 MINIMUM\_BALANCE 的静态字段要灵活得多，因为在成员函数中我们可以实现不同的业务规则，并针对不同的帐号生成不同的子类。

/\*\* 得到帐号值。帐号号码的格式如下：

\* BBBBAAAAAA，BBBB 是支行号码，

\* AAAAAA 是支行帐号号码

\*/

public long getAccountNumber()

{

return ( ( getBranchNumber() \* 100000 ) + getBranchAccountNumber() );

}

/\*\*

\* 设定帐号号码。帐号号码的格式如下：

\* BBBBAAAAAA，BBBB 是支行号码，

\* AAAAAA 是支行帐号号码

\*/

public void setAccountNumber(int newNumber)

{

setBranchAccountNumber( newNumber % 1000000 );

setBranchNumber( newNumber / 1000000 );

}

常量获取函数的另一个优点是，它们有助于提高代码的一致性。考虑上述代码，它们不能正常工作。一个帐户号码是支行号和支行帐号号码的结合。测试我们的代码，我们发现设置成员函数 setAccountNumber() 不能正确更新支行帐号（它取了最左边的三位数，而不是四位）。那是因为我们采用了 1,000,000，而不是 100,000 来提取字段 branchAccountNumber。若如下面所示，对这个值采用一个单独的来源，即常量获取函数 getAccountNumberDivisor()，我们的代码将更一致，并能正常工作。

/\*\*

\* 返回要求的除数，用来将整个帐户号分割

\* 为支行号和支行帐户号。

\* 整个帐号的格式是 BBBBAAAAAA。

\*/

public int getAccountNumberDivisor()

{

return ( (long) 1000000);

}

/\*\*

\* 得到帐号值。帐号号码的格式如下：

\* BBBBAAAAAA，BBBB 是支行号码，

\* AAAAAA 是支行帐号号码

\*/

public long getAccountNumber()

{

return ( ( getBranchNumber() \* getAccountNumberDivisor() ) +

getBranchAccountNumber() );

}

/\*\*

\* 设定帐号号码。帐号号码的格式如下：

\* BBBBAAAAAA，BBBB 是支行号码，

\* AAAAAA 是支行帐号号码

\*/

public void setAccountNumber(int newNumber)

{

setBranchAccountNumber( newNumber % getAccountNumberDivisor() );

setBranchNumber( newNumber / getAccountNumberDivisor() );

}

通过对常量使用获取函数，我们减少了代码出现问题的可能性，同时，提高了系统的可维护性。当帐号的格式变化时，我们知道它肯定会变化的，我们的代码很容易修改，因为它既有隐含的也有集中的供我们构造或分解帐号的必须信息。

### 4.5.5 集合存取函数

存取函数的主要目的是将访问封装到字段，以减少代码的耦合。集合，如数组和矢量，要比单值复杂，实现起来自然不只是需要获取和设置成员函数。特别是因为要对集合进行增减，所以需使用存取成员函数。在集合字段的适当处加入如下存取成员函数：

|  |  |  |
| --- | --- | --- |
| **成员函数类型** | **命名约定** | **示例** |
| 集合获取函数 | getCollection() | getOrderItems() |
| 集合设置函数 | setCollection() | setOrderItems() |
| 在集合中插入一个对象 | insertObject() | insertOrderItem() |
| 从集合中删除一个对象 | deleteObject() | deleteOrderItem() |
| 生成并且插入一个新对象到集合中 | newObject() | newOrderItem() |

这种方法的优点是集合被完全封装了，允许你以后用另外一个结构，可能是链表或是 B 树来取代它。

### 4.5.6 同时访问几个字段

存取成员函数的一个优点是，它使你能有效地执行业务规则。

考虑如下一个有关形状 (Shape) 的类的层次结构。Shape 的每一个子类通过 xPosition 和 yPosition 这两个字段表示位置，并且可以通过调用成员函数 move(Float xMovement, Float yMovement) 在屏幕上的二维坐标下移动。为达到我们的目的，图形在任何时刻都不可以只在一个坐标轴方向上移动，而应同时沿 x 和 y 轴移动（成员函数 move() 的两个参数中的任何一个参数都可以是 0.0）。这就意味着 move() 成员函数应该是公有的，但是 setXPosition() 和 setYPosition() 应该是私有的,被 move() 成员函数正确调用。

另一个实现方法是，引入一个可以同时更新两个字段的设置成员函数，如下文所示。成员函数 setXPosition() 和 setYPosition() 应该仍然是私有的，这样它们不会被外部类和子类直接调用（要加入一些如下文所示的注释来说明它们不应被直接调用）。

/\*\* 设定图形位置 \*/

protected void setPosition(Float x, Float y)

{

setXPosition(x);

setYPosition(y);

}

/\*\* 设置 x 坐标。重要：调用 setPosition(),不是这个成员函数。\*/

private void setXPosition(Float x)

{

xPosition = x;

}

/\*\* 设置图形的 y 坐标

\* 重要：调用 setPosition(),不是这个成员函数。

\*/

private void setYPosition(Float y)

{

yPosition = y;

}

### 4.5.7 存取函数的可见性

尽可能地让字段成为被保护 (protected) 类型，这样只有子类可以访问它们。仅当一个外部类需要访问一个字段时，才将相应的获取函数或设置函数置为公有。

注意：获取函数是公有而设置函数是私有的情况经常发生。

有时需要将设置函数设为私有以保证某个常量不变。

例如，Order 类可能含有一个字段表示 OrderItem 实例的集合，含有另一个叫 orderTotal 的字段表示整个定单 (order) 的总和。orderTotal 是一个表示订购项目子类总和的有用字段。唯一可以更新 orderTotal 值的成员函数是那些处理订购项目集合的函数。假设那些成员函数都在 Order 中实现，那么即使 getOrderTotal() 很可能是公有，也应设 setOrderTotal() 为私有。

## 4.6 一定要初始化静态字段

静态字段，也叫类字段，应被赋予有效值，因为不能假定类的实例将在一个静态字段被访问之前生成。

# 5. 局部变量标准

局部变量是指在一个块（通常是一个成员函数）内定义的对象或者数据项。一个局部变量的作用范围是定义它的块。局部变量的一些重要的程序设计标准集中在：

命名约定

注释约定

声明

## 5.1 命名局部变量

一般说来，命名局部变量遵循与命名字段一样的约定，即使用完整的英文描述符，任何非开头的单词的第一个字母要大写。

但是为方便起见，对于如下几个特殊的局部变量类型，这个约定可以放宽：

流

循环计数器

异常

### 5.1.1 命名流

当有一个单输入和/或单输出流在一个成员函数中被打开、使用和关闭时，通常的约定是对这些流分别采用 in 和 out 来命名。对于既用于输入又用于输出的流，采用 inOut 来命名。

一个常用的取代这种约定的方法是分别采用 inputStream，outputStream 和 ioStream 这样的名字，而不是 in，out 和 inOut，虽然这与 Sun 公司的建议相抵触。

### 5.1.2 命名循环计数器

因为局部变量常用作循环计数器，并且它为 C/C++ 所接受，所以在 Java 编程中，可以采用 i， j 或 k 作为循环计数器。若采用这些名字作为循环计数器，要始终使用它们。

一个常用的取代方法是，使用如 loopCounter 或只是 counter 这样的名字，但是这种方法的问题是，在一个需要多个计数器的成员函数中，常常发现象 counter1 和 counter2 这样的名字。概括起来说，i，j，k 作为计数器时， 它们可以很快被输入，它们被广泛的接受。

### 5.1.3 命名异常对象

因为在 Java 代码中异常处理也非常普遍，所以字母 e 作为一般的异常符被广泛地接受。如果异常嵌套第二层用ex作为第二层异常符号。

## 5.2 声明并注释局部变量

在 Java 中声明和注释局部变量有几种约定。这些约定是：

1. **一行代码只声明一个局部变量。**

这与一行代码应只有一个语句相一致，并使得对每个变量采用一个行内注释成为可能。

1. **用一个行内注释语句说明局部变量。**

行内注释是一种紧接在同一行的命令代码后，用符号 // 标注出来的单行注释风格（它也叫“行末注释”）。应注释出一个局部变量用于做什么、在哪里适用、为什么要用等等，使代码易读。

1. **仅将局部变量用于一件事。**

一旦将一个局部变量用于多个原因，就明显降低了它的一致性，使它难于理解。同时也增加了代码因为局部变量旧值的意外负面影响而产生问题的可能性，这些旧值来源于前面的代码。的确，局部变量的重新利用需要较少的内存，因而更高效，但是复用局部变量降低了代码的可维护性，使代码脆弱。这常常让由于不必分配更多内存而带来的小节省变得不值得。

**关于声明的一般注释**

在代码行间，如在一个 if 语句作用域内，声明的局部变量对于不熟悉你的代码的人来说是难于找到的。

一种取代在第一次使用局部变量之前声明它们的方法是在代码的前部声明它们。函数应该简短，参见（第 2.4.5 节“写出简短单独的命令行”），所以要去代码顶部判断局部变量用途的工作并不是很糟。

# 6. 成员函数的参数标准

有关成员函数参数的重要标准集中在参数应如何命名和说明。参数指成员函数的实参。

## 6.1 命名参数

参数命名遵循与局部变量命名完全一样的约定。对于局部变量，名字隐藏是一个问题。

示例：

customer

inventoryItem

photonTorpedo

in

e

一个可行的取代方法，

例如 Smalltalk，是采用局部变量的命名约定，但在名字之前加入“a”或“an”。加上“a”或“an”有助于让参数与局部变量和字段区分开来，避免名字隐藏的问题。这种方法较好。

示例：

aCustomer

anInventoryItem

aPhotonTorpedo

anInputStream

anException

## 6.2 注释参数

成员函数的参数在采用 javadoc @param 标识的头文件中注释。应说明：

1. **参数用来做什么。**

需要注释出参数用来做什么，以便其他开发者了解使用参数的上下文。

1. **任何约束或前提条件.**

如果一个参数的值域不能被成员函数接收，则应让调用者知道。可能一个成员函数只接收正数，或者字符数小于五的字符串。

1. **示例。**

如果应传递什么样的参数不明显，那么应该在注释中给出一个或多个例子。

**采用参数类型接口。**

若合适的话，不要只说明参数类型属于哪一类，如 Object，而应说明属于哪个接口，例如 Runnable。这样的好处是，这种有赖于环境的方法更具体 （Runnable 比 Object 更具体），或者在支持多态性（不坚持一个参数是一个类的层次结构中某个类的实例，而说明它支持一个特定的接口，这意味着它只用多态地适应你的需要即可）上是一个更好的方法。

# 7. 类、接口、包和编译单元的标准

这一章集中讲述类、接口、包和编译单元的标准和指南。

* 类是一个可以让对象创建的模板。
* 类包含字段和成员函数的声明。
* 接口是公共标识的定义，包括成员函数和字段。
* 使用接口的类必须支持这些函数和字段。
* 包是一个相关类的集合。
* 编译单元是声明类和接口的源码文件。

因为 Java 允许编译单元放在数据库中，所以一个单独的编译单元可能不与一个源码文件物理上直接相关。

## 7.1 类的标准

类的一些重要标准基于：

* 命名约定
* 注释约定
* 声明约定
* 公共和保护接口

### 7.1.1 命名类

标准 Java 约定是使用完全的英文描述符，所有单词的第一个字母要大写，并且单词中大小写混合。

类名应是单数形式。示例：

Customer

Employee

Order

OrderItem

FileStream

String

### 7.1.2 注释类

以下的信息应写在文档注释中紧靠类的定义的前面：

1. **类的目的。**

开发者需要了解一个类的一般目的，以判断这个类是否满足他们的需求。养成一个注释与类有关的任何好东西的习惯，例如：它是否是一个模式的一部分，或是使用它时有什么要引起注意的限制。

1. **已知的问题。**

如果一个类有任何突出的问题，应说明出来，让其他的开发者了解这个类的缺点/难点。此外，还应注明为什么不解决问题的原因。注意：如果问题仅仅针对一个成员函数，那么它应直接与那个成员函数相联系。

1. **类的开发/维护历史。**

通常要包含一个历史记录表，列出日期、类的作者和修改概要。这样做的目的是让进行维护的程序员了解过去曾对一个类所做的修改，是谁做了什么样的修改。

1. **注释出采用的不变量。**

不变量是指一套有关实例或类在所有“稳定”时间片内为“真”的声明。“稳定时间片”是指在一个成员函数被对象/类调用之前和立刻调用之后的时间。通过说明一个类的不变量，你让其他的开发者了解应该如何使用一个类。

1. **并行策略。**

任何采用 Runnable 接口的类应充分说明它的并行策略。对许多程序员来说，并行编程是一个新的而且复杂的题目，所以需要投入一些额外的时间来确保人们能够读懂你的东西。说明你的并行策略以及为什么选取这个策略而不是其它策略这很重要。

常用的并行策略包括下面一些内容：

* 同步对象
* 停滞 (balking) 对象
* 警戒 (guarded) 对象
* 版本 (versioned) 对象
* 同步策略控制器
* 接收器

### 7.1.3 类的声明

一种让你的类容易被理解的方法是用一致的方式来声明它们。Java 中常用的方法是按如下顺序声明一个类：

* 公共成员函数
* 公共字段
* 被保护成员函数
* 被保护字段
* 私有成员函数
* 私有字段

指出，构造函数和 finalize() 应该首先列出，可能是因为这些将会是另一个开发者为了解如何使用一个类而首先查看的成员函数。此外，因为我们有一个要求将所有字段声明为私有的标准，所以声明顺序实际应变为：

* 构造函数
* finalize()
* 公共成员函数
* 被保护成员函数
* 私有成员函数
* 私有字段

在每一个成员函数分组内，常将函数按照字母表的顺序列出来。许多开发者在每个组内有选择性地列出了静态成员函数，然后列出实例成员函数，再在这两个子分组内按照字母表的顺序列出成员函数。这两种方法都可用，你只需选用一种并且一直用它。

### 7.1.4 将公共和保护接口最小化

面向对象程序设计的基本点之一是最小化一个类的公共接口。这样做有几个理由：

1. **可学习性。**

要了解如何使用一个类，只需了解它的公共接口即可。公共接口越小，类越容易学习。

1. **减少耦合。**

当一个类的实例向另一个类的实例或者直接向这个类发送一条消息时，这两个类变得耦合起来。最小化公共接口意味着将耦合的可能降到最低。

1. **更大的灵活性。**

这直接与耦合相联系。一旦想改变一个公共接口的成员函数的实现方法，如你可能想修改成员函数的返回值，那么你很可能不得不修改所有调用了该成员函数的代码。公共接口越小，封装性就越大，代码的灵活性也越大。

尽力使公共接口最小化这一点明显地很值得你的努力，但通常不明显的是也应使被保护接口最小化。

基本思想是，从一个子类的角度来看，它所有超类的被保护接口是公共的。任何在被保护接口内的成员函数可被一个子类调用。所以，出于与最小化公共接口同样的理由，应最小化类的被保护接口。

**首先定义公共接口。**

大多数有经验的开发者在开始编写类的代码之前就先定义类的公共接口。

第一，如果你不知道一个类要完成怎样的服务/行为，你仍有一些设计工作要做。

第二，这样做使这个类很快地初具雏形，以便其他有赖于该类的开发者在“真正的”类被开发出来以前至少可以用这个雏形开始工作。

第三，这种方法给你提供了一个初始框架，围绕着这个框架你构造类。

## 7.2 接口标准

接口的一些重要标准基于：

* 命名约定
* 注释约定

### 7.2.1 命名接口

Java 的约定是用大小写混合的方式命名接口，并且每个单词的第一个字母要大写。虽然象 Singleton 或 DataInput 这样的描述性的名词经常用来命名一个接口，但是较受欢迎的 Java 约定是用象 Runnable 或 Cloneable 这样的描述性的形容词来命名 。

**可替代的其它办法**

**在接口名前加前缀”I”**

建议在一个接口名的前面附加上字母“I”，结果使名字变为如 ISingleton 或 IRunnable 这样。这种方法有助于将接口名与类和包的名字区分开来。仅仅只是因为它使类的图表，有时也叫对象模型，变得易读。

这种方法的一个主要的缺点是一些现有的接口，如 Runnable，未采用此种方法。这种接口命名约定在 Microsoft 的 COM/DCOM 体系结构中也很流行。

### 7.2.2 注释接口

以下的信息应写在文档注释中紧靠接口定义的前面：

1. **目的。**

在其他开发者应用一个接口之前，他们需要理解接口封装的概念。换句话说，他们需要了解接口的目的。一个好的检测是否有必要定义一个接口的方法是你是否可以很容易地说明它的目的。如果说明起来有困难，很可能从一开始起就不需要这个接口。在 Java 中接口的概念较新，所以人们对如何正确使用它们还不是很有经验，常常滥用它们。

1. **它应如何被使用以及如何不被使用。**

开发者需要了解接口应如何使用以及如何不被使用。

因为成员函数的标识在接口中定义，所以对于每个成员函数的标识应遵循第二章中所讨论的注释约定。

## 7.3 包的标准

包的一些重要标准基于：

* **命名约定**
* **注释约定**

### 7.3.1 命名包

关于包的命名有几条规则。按顺序来说，这些规则是：

1. **标识符用点号分隔开来。**

为了使包的名字更易读，Sun 公司建议包名中的标识符用点号来分隔。例如，包名 java.awt 含有两个标识符 java 和 awt。

1. **Sun 公司的标准 java 分配包用标识符 .java 开头。**

Sun 保有这种权利，使得无论你的 Java 开发环境的零售商是怎样的，标准 java 包的命名始终一致。

1. **局部包的名字中的第一个标识符不能都是大写。**

所谓局部包是指那些在你的机构内部使用，不会应用到其他机构中去的包。这样的包的名字的例子有 persistence.mapping.relational 和interface.screens。

1. **全局包的名字用你的机构的 Internet 保留域名开头。**

一个要应用到多个机构的包应包含创建机构的域名，并且最高层的域名类型要大写。

例如，要应用前面的包，它们应被命名为:

com.rational.www.persistence.mapping.relational

com.rational.www.interface.screens。

### 7.3.2 注释包

应保有一个或者多个外部文档以说明你的机构所创建的包的用途。对于每个包，应说明：

1. **包的基本原理。**

其他开发者需要了解一个包是用来做什么的，这样他们才能判断是否可以用它，如果是一个共享包，他们可以判断是否需要改进或是扩展它。

1. **包中的类。**

在包中要包含一个类和接口的列表，每个类或接口用简短的一行文字来说明，以便让其他的开发者了解这个包中含有什么。

技巧：生成一个以包名命名的 HTML 文件，将它放到包的适当的目录中去。这个文件应具有后缀 .html。

## 7.4 编译单元标准

编译单元的标准和指南基于：

* 命名约定
* 注释约定

### 7.4.1 命名编译单元

编译单元，在这个情况下是一个源码文件，应被赋予文件内定义的主要的类或接口的名字。用与包或类相同的名字命名文件，大小写也应相同。扩展名 .java 应作为文件名的后缀

示例：

Customer.java

Singleton.java

SavingsAccount.java

### 7.4.2 注释编译单元

虽然应努力使一个文件中只包含一个类或接口，但是有时在一个文件中定义数个类（或者接口）也可理解。一般的经验规则是，如果类 B 的唯一用途是封装只被类 A 需要的功能，那么可以理解类 B 出现在与类 A 相同的源码文件中。结果下面的文档约定应用于一个源码文件，而不具体到类：

1. **对于有几个类的文件，列出每一个类。**

如果一个文件包含多个类，则应给出一个类的列表，并且简要地说明每个类。

1. **文件名和/或标识信息。**

文件名应包含在它的顶端。好处是如果代码被打印出来了，你会知道源码文件是什么。

1. **版权信息。**

若可能，应说明文件的所有版权信息。通常的做法是说明版权的年份和版权持有个人/机构的名字。注意：代码的作者可能不是版权持有者。

# 8. 错误处理和异常

通常的思想是只对错误采用异常处理：逻辑和编程错误，设置错误，被破坏的数据，资源耗尽，等等。通常的法则是系统在正常状态下以及无重载和硬件失效状态下，不应产生任何异常。

**用异常处理逻辑和编程错误，设置错误，被破坏的数据，资源耗尽**。尽早采用适当的日志机制来报告异常，包括在异常发生的时刻。

最小化从一个给定的抽象类中导出的异常的个数。

在大型系统中，不得不在每一级处理大量的异常使得代码难于理解和维护。有时，异常阻碍了正常的处理。

有以下几种方式最小化异常的数目：

仅导出几个异常，但提供“diagnosis”原语，用以查询错误抽象类或对象，以得到有关产生出的问题本质的更多信息。

 在对象中加入“exceptional”语句，并且提供用来明确检查对象可用性的原语。

**对于经常发生的可预计事件不要采用异常。**

用异常来表达并非一定是错误的状态有几个不便之处：

* 它易混淆。
* 它通常在控制流中强制性地产生一些中断，而这些中断更难理解和维护。
* 它使得代码调试起来更令人痛苦，因为大多数源码级的调试器在缺省值状态下标志出所有的异常。

例如，不要将异常以某种额外值的形式由函数返回（象查询中的 Value\_Not\_Found）。使用一个含有“out”参数的进程，或者引入一个意思为 Not\_Found 的特殊值，或者在一个含有区别量 Not\_Found 的记录中包装一个返回类型。

**不要使用异常实现控制结构。**

这是前面规则的一个特例：异常不应作为“goto”语句的一种形式来使用。

**确保状态码有一个正确值。**

当用一个子程序返回的状态码作为一个“out”参数时，一定要确保“out”参数被赋了值，这可以通过将赋值语句作为子程序体的第一个可执行语句来实现。系统化地使所有状态的缺省值为 “success” 或是 failure”。考虑子程序的所有可能出口，包括异常处理。

**在本地进行安全性检查，不要指望你的客户会去做这件事。**

也就是说，如果一个子程序被给予错误的输入时可能给出错误的输出，则应在子程序中通过控制方式加入检测和报告非法输入的代码。不要依赖于注释来告诉客户输入正确值。如果不检测无效参数，注定早晚有一天，那条注释会被忽略，导致难以调试的错误。

# 9. 其他标准和版本

这一章说明几个重要的标准/指南，但因为它们涉及很广，所以足以单独列出一章。

## 9.1 复用

任何你从外部源所购买/复用的 Java 类库或者包应是 100% 纯粹的 Java 语言。通过强调这一标准，确保了你所用的东西将在你想设置它的所有平台上工作。你可以从各种途径获得 Java 的类、包或者 Applets，或是第三方的专门开发 Java 库的开发公司，或是你的机构中的另一个部门或项目组。

## 9.2 导入类

在说明类名时，import 语句允许使用通配符。例如，

**import java.awt.\* 语句**

一次性地引入了包 java.awt 中的所有类。实际上，这并不完全正确。实际情况是每一个所使用的取自于 java.awt 包中的类，在编译时被引入代码，而其中未使用的类不被引入。虽然这听起来象是一个好的特点，但是它却降低了代码的可读性。一个更好的方法是完全限制代码所使用的类的名字。一个较好的导入类的方法如下面的例子所示：

import java.awt.Color;

import java.awt.Button;

import java.awt.Container;

## 9.3 优化 Java 代码

优化 Java 代码是程序员最后而不是最先应考虑的事。将优化放到最后是因为只要优化那些需要优化的代码。代码的一小部分常常占用了处理时间的大部分，这样的代码就应该优化。缺乏经验的程序员会犯的一个经典性的错误是，想优化他们所有的代码，甚至那些运行起来已经很快的代码。

### 9.3.1 不要浪费时间去优化那些没人会在意的代码！

当优化代码时应该寻找什么？

**指出最重要的因素是固定费用和大输入量时的性能。**

理由很简单：固定费用决定了程序在小输入量时的运行速度，算法决定了大输入量时的运行速度。他的基本规则是，一个程序若在小输入量和大输入量时都运行得很好，那么在中等输入量的情况下很可能也会运行得很好。

**开发可在几种硬件平台和/或操作系统上运行的软件的开发者需意识到不同平台的特性。**

那些可能要消耗相当数量时间的操作，如处理内存和缓冲区的方式，在不同的平台之间常常区别较大。常常会发现要根据平台来对代码进行不同的优化。

**优化代码时要注意的另外一点是用户的优先权问题，因为依赖于上下文，人们会对一些特定的延迟敏感**。

例如，用户可能更喜欢那种立刻显示自身然后再等 8 秒钟再调出数据的屏幕，而不是那种在 5 秒钟内调出数据再显示的屏幕。换句话说，只要能有即刻的响应，大多数的用户乐意多等一点时间。在优化你的程序时，这是一个重要知识。

### 9.3.2 无需总是从用户的观点来优化代码。

虽然优化意味着你的应用程序成功与失败的区别，但不要忘了让代码正确运行更重要。切勿忘记，运行起来慢但却正确的软件永远比运行起来快但却不正确的软件要更受欢迎。

## 9.4 编写 Java 测试集

面向对象的测试是一个不会被对象开发团队忽视的重要课题。事实上，无论用何种语言书写代码，或者你或者某个人必须对你所写的软件进行测试。测试集指的是成员函数的集合，这些函数中的一部分嵌在类本身中（这叫做嵌入式测试），另一些在专门的用于检测应用程序的测试类中。

1. **在所有的测试成员函数名前加上前缀 test。**

这可以使你迅速地找到代码中所有的测试函数。这种在测试函数名字前加前缀 test 方法的优点是，它让你在编译代码的产品版之前，能很容易地从代码中去掉测试函数。

1. **一致地命名所有成员函数的测试函数。**

方法测试是证实单个成员函数能如所定义的那样运行的操作。所有的成员函数应按照 testMemberFunctionNameForTestName 的格式命名。例如，用来测试 withdrawFunds() 的测试集的成员函数包含 testWithdrawFundsForInsufficientFunds() 和 testWithdrawFundsForSmallWithdrawal()。如果要对 withdrawFunds() 做一系列的测试，可以选择写一个叫 testWithdrawFunds() 的成员函数来调用所有函数。

1. **一致地命名所有类的测试成员函数。**

类测试是证实单个类能象所定义的那样的操作。所有的类应按照 testSelForTestName 的格式来命名。例如，检测 Account 类的测试集成员函数包括 testSelfForSimultaneousAccess() 和 testSelfForReporting()。

生成单点来调用类的检测函数。开发一个叫 testSelf() 的静态成员函数来调用所有类测试和方法测试成员函数。

注释测试集成员函数。注释测试集成员函数。注释应包含测试描述和测试的期待结果。

# 10. 成功的模式

作为一个软件开发者，在你的所有品中含有一份有关标准的文档并不会自动地使你更加地有效率。要成功，你必须选择变得更有效率，这意味着你必须将这些标准有效地应用起来。

## 10.1 有效地使用这些标准

以下的建议将帮助你更有效地使用本文所描述的 Java 编程规范：

1. **理解标准。**

花些时间去理解为什么每个标准和指南会使开发效率提高。比如说，不要仅仅是因为指南中要求你才在一行仅声明一个局部变量，而应该是因为你明白它能使你的代码更易懂你才这样做。

1. **信任这些标准。**

理解每个标准是一个开始，但你还需要信任这些标准。遵守标准不应仅仅是当你有时间才做的事，而你应该一直遵守，因为你相信这是最好的程序设计方法。

1. **当你写代码时就应该遵守标准，而不应是一个事后的想法。**

加了注释的代码不仅在你写程序时，而且在你写完程序时，都更容易理解。在程序开发阶段和维护阶段，一致性地命名成员函数和字段都使工作更加容易。在开发和维护阶段，整洁的代码让工作更加容易。概括起来说，遵守标准将提高你开发过程中的生产率，并且使你的代码更易维护（因此也使维护者的生产率提高了）。如果从一开始你就写出整洁的代码，你将在撰写过程中受益。

1. **使它们成为你的质量保证的过程。**

代码检查的一部分应该是确保源码遵守你的机构所采用的标准。将标准作为你训练和指导开发员更有效率的基础。

## 10.2 其它导向成功代码的因素

**1. 面向人而不是面向机器编程。**

你的开发努力的主要目的应该是你的代码易被其它人理解。如果没人能理解它，它就一点儿优点也没有。使用命名约定。注释代码。给代码分段。

**2. 首先设计，然后编写代码。**

你是否曾遇到过这样的情况：一些你的程序所倚靠的代码需要修改？可能是要传一个新的参数给一个成员函数，或者是需要将一个类拆成几个类。为了确信你的代码与被重新设置修改的代码还能一起工作，你必须做多大的额外工作呢？你有多么乐意？你是否曾经问过自己，为什么有些人在开始写代码时不先停下来考虑一下，以避免这一切的发生？他们为什么不首先设计程序呢？当然你做了。如果你在实际开始动手写代码之前花时间想清楚你打算怎样写你的代码，你很可能可以少花些时间编写它。此外，仅通过开始时就将它们想好，你将潜在地减少将来修改代码所带来的影响。

**3. 一小步一小步地开发。**

一小步一小步地开发，先写几个成员函数，检测它们，再多写几个，这样开发比一次性地写完所有代码然后修改它要有效得多。检测和修改十行代码远比检修一百行代码要容易得多。实际上可以很有把握地说，同样是编写、测试和修改 100 行代码，十行十行地做所花的时间少于一口气做 100 行所花的时间的一半。理由很简单。当测试你的代码并发现问题时，问题几乎总是在刚写完的新代码中，当然这假定在剩下的那些旧代码上开始写是很可靠的。在一小部分代码中寻找问题会比在一大段代码中找问题要快得多。通过一小步一小步逐步地开发，减少了查找错误所需的平均时间，这转而又减少了整个的开发时间。

**4. 让代码简洁。**

复杂的代码或许让人在智力上获得满足，但是如果别人读不懂， 那就不好了。如果要求某人，甚至是你，第一次修改一段复杂的代码以纠正其中的错误或对它进行增强，那么很可能代码会被重写。实际上，你很可能已经因为代码难懂而重写过别人的代码。当你在重写代码时，你是怎样认为代码的最初开发者的呢？你认为那个人是天才还是怪物？写出那种后来要被重写的代码没什么可骄傲的，所以应该遵循 KISS 法则：要使代码简单直白。

**5. 学习常用的模式、反模式和代码模式。**

有大量的分析、设计和处理的模式和反模式以及编程代码模式供你提高开发效率。

# 11. 总结及检查点

为方便起见，本章总结到目前为止已给出的规范，并指出检查点。

在我们开始概括和总结这个规范前，我想重申主要的要求：

当你违背一个标准时，将其注释出来。除了这个标准之外的所有标准都可以违背。如果违背了，你必须说明为什么你要违背这个标准，违背这个标准可能产生的影响，以及标准被应用于这个场合之前可能/必须出现的任何条件。

## 11.1 代码格式约定

**检查点:**

* {}使用约定
* 缩进约定
* 空格的使用约定

## 11.2 Java 命名约定

除了以下几个特例之外，命名时应始终采用完整的英文描述符。此外，一般应采用小写字母，但类名、接口名以及任何非初始单词的第一个字母要大写。

检查点:

* 使用完整的英文描述符
* 采用适用于该领域的术语
* 采用大小写混合使名字可读
* 尽量少用缩写，但如果用了，要明智地使用，易于理解。
* 避免使用长的名字（小于 15 个字母是个好主意）
* 避免使用类似的名字，或者仅仅是大小写不同的名字
* 避免使用下划线

|  |  |  |
| --- | --- | --- |
| **操作项** | **命名约定** | **示例** |
| 实参/参数 | 使用传递值/对象的完整的英文描述符，可能要在名字之前加上 a 或 an 前缀。重要的是选择一种并坚持用它。 | customer, account, (推荐使用) - 或者 -  aCustomer,anAccount |
| 字段/属性 | 字段采用完整的英文描述，第一个字母小写，任何中间单词的首字母大写。 | firstName  lastName,warpSpeed |
| 布尔型的获取成员函数 | 所有的布尔型获取函数必须用单词 is 做前缀。如果你遵守前文所说的布尔字段的命名标准，那么你只需将字段名赋给它即可。 | isPersistent()  isString()  isCharacter() |
| 类 | 采用完整的英文描述符，所有单词的第一个字母大写。 | Customer  SavingsAccount |
| 编译单元文件 | 使用类或接口的名字，或者如果文件中除了主类之外还有多个类时，加上前缀 java 来说明它是一个源码文件。 | Customer.java  SavingsAccount.java  Singleton.java |
| 组件/ 部件 | 使用完整的英文描述来说明组件的用途，末端应接上组件类型。 | okButton, fileMenu  customerList, |
| 构造函数 | 使用类名 | Customer(),  SavingsAccount() |
| 析构函数 | Java 没有析构函数，但一个对象在垃圾收集时，调用成员函数 finalize() 。 | finalize() |
| 异常 | 通常采用字母 e 表示异常。第二层用ex | e, ex |
| 静态常量字段(常量) | 全部采用大写字母，单词之间用下划线分隔。一个较好的方法是采用静态常量获取成员函数，因为它很大地提高了灵活性。 | MIN\_BALANCE,  DEFAULT\_DATE |
| 获取成员函数 | 被访问字段名的前面加上前缀 get。 | getFirstName()  getLastName()  getWarpSpeeed() |
| 接口 | 采用完整的英文描述符说明接口封装，所有单词的第一个字母大写。习惯上，名字后面加上后缀 able,. ible 或者 er，但这不是必需的。 | Runnable  Contactable  Prompter  Singleton |
| 局部变量 | 采用完整的英文描述符，第一个字母小写，但不要隐藏已有字段。例如，如果有一个字段叫 firstName，不要让一个局部变量叫firstName。 | grandTotal  customer  newAccount |
| 循环计数器 | 通常采用字母 i，j，k 或者 counter 都可以接受。 | i, j, k, counter |
| 包 | 采用完整的英文描述符，大小写混合，所有单词的第一个字母大写，其它都小写。对于全局包，将你的 Internet 域名反转并接上包名。 | java.awt  com.ambysoft.www.  persistence.mapping |
| 成员函数 | 采用完整的英文描述说明成员函数功能，第一个单词尽可能采用一个生动的动词，第一个字母小写。 | openFile()  addAccount() |
| 设置成员函数 | 被访问字段名的前面加上前缀 set。 | setFirstName()  setLastName()  setWarpSpeed() |

## 11.3 Java 注释约定

一个很好的可遵循的有关注释的经验法则是：问问你自己，你如果从未见过这段代码，要在合理的时间内有效地明白这段代码，你需要哪些信息。

检查点:

* 注释应该增加代码的清晰度
* 如果你的程序不值得注释，那么它也很可能不值得运行
* 避免使用装饰物，也就是说，不要使用象小旗子那样的注释
* 保持注释的简洁
* 在写代码之前写注释
* 注释出为什么做了一些事，而不仅仅是做了什么

### 11.3.1 Java 注释类型

下面的图表说明 Java 注释的三种类型，并给出使用方法。

|  |  |  |
| --- | --- | --- |
| **注释语句类型** | **用法** | **示例** |
| 文档注释 | 文档注释 在接口、类、成员函数和字段声明之前紧靠它们的位置用文档注释进行说明。文档注释由 javadoc 处理，为一个类生成外部注释文档. | /\*\*  \* <p>标题： </p>  \* <p>功能： </p>  \* <p>版权： Copyright (c) 2003</p>  \* <p>公司: </p>  \* 作者：  \* <p>版本： 1.0</p>  \*/ |
| C 语言风格注释 | 采用 C 语言风格的注释语句将无用的代码注释掉。保留这些代码是因为用户可能改变想法，或者只是想在调试中暂时不执行这些代码。 | /\*  这部分代码已被它前面的代码替代，所以于 1999 年6 月 4 日被 B. Gustafsson 注释掉。如果两年之后仍未用这些代码，将其删除。  . . . （源代码）  \*/ |
| 单行注释 | 在成员函数内部采用单行注释语句对业务逻辑、代码片段和临时变量声明进行说明。 | // 因为让利活动  // 从 1995 年 2 月开始，  // 所以给所有超过 $1000 的  // 发货单 5% 的折扣。 |

### 11.3.2 注释哪些部分

下表概括了所写 Java 代码中的每一部分哪些需要注释说明，也是注释的各检查点。

|  |  |
| --- | --- |
| **项目** | **注释哪些部分** |
| 实参/ 参数 | 参数类型  参数用来做什么  任何约束或前提条件  示例 |
| 字段/属性 | 字段描述  注释所有使用的不变量  示例  并行事件  可见性决策 |
| 类 | 类的目的  已知的问题  类的开发/维护历史  注释出采用的不变量  并行策略 |
| 编译单元 | 每一个类/类内定义的接口，含简单的说明  文件名和/或标识信息  版权信息 |
| 获取成员函数 | 若可能，说明为什么使用滞后初始化 |
| 接口 | 目的  它应如何被使用以及如何不被使用 |
| 局部变量 | 用处/目的 |
| 成员函数注释 | 成员函数做什么以及它为什么做这个  哪些参数必须传递给一个成员函数  成员函数返回什么  已知的问题  任何由某个成员函数抛出的异常  可见性决策  成员函数是如何改变对象的  包含任何修改代码的历史  如何在适当情况下调用成员函数的例子  适用的前提条件和 后置条件 |
| 成员函数内部注释 | 控制结构  代码做了些什么以及为什么这样做  局部变量  难或复杂的代码  处理顺序 |
| 包 | 包的基本原理  包中的类 |

## 11.4 Java 程序设计约定

有许多的有关 Java 代码可维护性和可改进性的重要约定和标准。99.9% 的时间里，面向他人，面向你的开发同事编程要比面向机器编程重要得多。使你的代码为别人所理解是最重要的。

|  |  |
| --- | --- |
| **约定目标** | **约定** |
| 存取成员函数 | 考虑对数据库中的字段使用滞后初始化  使用存取函数获得和修改所有字段  对常量采用存取函数  对于集合，加入成员函数来插入和删除项  一旦可能，将存取函数置为被保护类型，不是公共类型 |
| 字段 | 字段永远为私有类型  不要直接访问字段，应使用存取成员函数  不要使用静态常量字段（常量），应使用存取成员函数  不要隐藏名字  一定要初始化静态字段 |
| 类 | 最小化公共和保护接口  在开始写代码之前定义一个类的公共接口  按以下顺序声明一个类的字段和成员函数：  构造函数  finalize()  公共成员函数  被保护成员函数  私有成员函数  私有字段 |
| 局部变量 | 不要隐藏名字  一行代码只声明一个局部变量  用一个行内注释说明局部变量  在使用局部变量之前声明它  仅将局部变量用于一件事 |
| 成员函数 | 给代码加上注释  给代码分段  使用空白，控制结构之前用一个空行，成员函数之前用两个空行  一个成员函数应能在 30 秒内让人理解  写短小单独的命令行  尽量限制成员函数的可见性  说明操作的顺序 |

# 12. 词汇表

|  |  |
| --- | --- |
| **100% 纯粹** | Sun 公司的一个“许可标志”，表明一个 Java 的 Applet、应用程序或者包可在任何支持 Java 虚拟机的平台上运行。 |
| **存取函数** | 修改或者返回字段值的成员函数。也叫访问修改程序。参见获取函数和设置函数。 |
| **分析模式** | 描述一个业务/领域解决方案的建模模式。 |
| **反模式** | 一种解决一般问题的方法，可及时地证明是否错误或非常无效。 |
| **实参** | 参见“参数”。 |
| **字段** | 用来描述一个类或者类的实例的文字数据类型或其它对象的变量。实例字段描述对象（实例），静态字段描述类。字段也被称为字段变量和属性。 |
| **BDK** | Beans 开发工具 (Beans development kit)。 |
| **块** | 包含在大括号内的零个或多个语句的集合。 |
| **括号** | 即字符 **{** 和 **}**，分别叫开括号和闭括号，用来定义一个块的开始和结束。 |
| **类** | 创建对象的定义或模板。 |
| **类测试** | 确信一个类及其实例能如所定义的那样工作的行为。 |
| **CMVC** | 配置和版本控制。 |
| **编译单元** | 存在于磁盘上的物理源码文件或是存放在一个数据库中的“虚拟”源码文件，类和接口在其中定义. |
| **组件** | 象列表、按钮或者窗口那样的接口部件。 |
| **常量获取函数** | 返回“常量”值的获取函数，该常量在需要使用时可能难于程序设计或者计算。 |
| **构造函数** | 完成对象创建时任何所需的初始化的成员函数。 |
| **容器** | 包含有其它对象的对象，它与其它对象合作来完成自身的行为。这可以通过使用内部类 (JDK 1.1+) 或者一个对象内其它类的实例集合 (JDK 1.0+) 来实现。 |
| **CPU** | 中央处理器。 |
| **C 语言风格的注释** | 一种 Java 注释语句格式 /\* 和 \*/，取自 C/C++ 语言，它可用于生成多行注释。通常用来“注释掉”测试中不需或者不想要的代码行。 |
| **设计模式** | 描述一个设计问题的解决方案的建模形式。 |
| **析构函数** | 一个用来当一个对象不再需要时从内存中删去该对象的 C++ 类成员函数。因为 Java 管理自身的内存，所以这种函数不需要。但 Java 支持一个功能上类似的叫做 **finalize()** 的成员函数。 |
| **文档注释** | 一个 Java 注释格式 /\*\* 和 \*/，可由 *javadoc* 处理，来给一个类文件提供外部说明。对于接口、类、成员函数和字段的主要说明应用文档注释来写。 |
| **字段** | 参见“属性”。 |
| **finalize()** | 在一个对象从内存里删除之前的“垃圾收集”过程中被自动调用的成员函数。这个成员函数的目的是做所有必需的清扫工作，如关闭打开文件。 |
| **垃圾收集** | 不再使用的对象被自动地从内存中删除的自动内存管理。 |
| **获取函数** | 一种返回字段值的存取成员函数。获取函数可用来回答常量值，实现时常常最好是将常量作为静态字段，因为这种方法更加灵活。 |
| **HTML** | 超文本生成语言，用于生成网页的工业标准形式。 |
| **缩进** | 参见“分段”。 |
| **行内注释**  **(Inline comments)** | 用一行紧接在同一行代码之后来说明这行源码的注释方法。虽然也可采用 C 语言风格的注释，但此种情况下，单行注释更被典型应用。 |
| **接口** | 包含成员函数和字段的公共标识定义，使用接口的类必须支持接口。通过组合，促成了接口的多态性。 |
| **I/O** | 输入/输出。 |
| **不变量** | 一套有关实例或类在所有“稳定”时间片内为“真”的声明。“稳定时间片”是指在一个成员函数被对象/类调用之前和调用之后的时间。 |
| **Java** | 一种工业标准的面向对象的开发语言。它非常适合于开发 Internet 的应用程序以及必须在多种计算平台上运行的应用程序。 |
| **javadoc** | 含在 JDK 中的一个工具，用于处理 Java 源码文件并生成一个 HTML 格式的外部文件，该文件基于源码文件中的注释语句说明源码文件的内容。 |
| **JDK** | Java 开发工具 (Java Development Kit)。 |
| **滞后初始化 (Lazy initialization)** | 一种在字段第一次被需要时才在相应的获取成员函数中初始化该字段的技巧。滞后初始化用于当一个字段并非经常使用时，以及字段需要大量的内存来存放或者需要从永久存取中读出时。 |
| **局部变量** | 在一个块的作用域内，通常时一个成员函数内，定义的变量。一个局部变量的作用范围是定义它的块。 |
| **成员函数** | 与一个类或者类的实例相关的一段可执行代码。将一个成员函数作为一个函数面向对象的等价物。 |
| **成员函数标识** | 参见“标识”。 |
| **方案测试** | 确信一个成员函数能如所定义的那样工作的行为。 |
| **名字隐藏** | 给一个字段/变量/参数取的名字与另一个更高作用范围内同类的名字相同，或者至少相似的做法。名字隐藏最经常的滥用方式是将一个局部变量取了一个与实例字段相同的名字。应避免名字隐藏，因为它使你的代码难于理解并且容易出错。 |
| **重载** | 成员函数过载是指在一个类（或者子类）中成员函数被不止一次地定义，而每次定义的唯一区别只是标识的不同。 |
| **覆盖** | 成员函数覆盖是指它在一个子类中被重新定义，其标识与原先相同。 |
| **包** | 相关类的集合。 |
| **分段** | 一种在代码块的范围内让代码缩进一个单位，使其与块外的代码区别开来的技巧，通常是缩进一个水平制表符的距离。分段有助于提高代码的可读性。 |
| **参数** | 传给一个成员函数的实数。一个参数可以是一个已定义的类型，例如字符串、整型、或者对象。 |
| **后置条件** | 在一个成员函数运行结束时为“真”的属性或者声明。 |
| **前置条件** | 让一个函数可以正确运行的限制条件。 |
| **属性** | 参见“字段”。 |
| **设置函数** | 给一个字段设值的存取成员函数。 |
| **标识** | 若存在，则为参数类型的结合，其顺序必须传给一个成员函数。也叫成员函数标识。 |
| **单行注释** | 一种 Java 注释格式 //，取自 C/C++ 语言，常用做内部成员函数的业务逻辑注释。 |
| **标记** | 用于标注文档注释特定部分的一个约定，文档注释将被 *javadoc* 处理，生成看上去专业化的注释。标记的例子，如 *@see* 和 *@author*。 |
| **测试集** | 用于检测代码的成员函数集合。 |
| **UML** | 统一建模语言，是一种工业标准建模符号。 |
| **可见性** | 用来描述类、成员函数或字段的封装程度的技术。关键字 public，protected 和 private 可被用来定义可见性。 |
| **空白** | 为提高代码可读性而加入的空行、空格和制表符。 |
| **部件** | 参见“组件”。 |