

# 中软国际公司内部技术规范

## Java语言编程规范



中软国际科技服务有限公司

版权所有 侵权必究

## 修订声明

参考:华为 Java 语言编码规范。

0.

| 日期         | 修订版本 | 修订描述           | 作者  |
|------------|------|----------------|-----|
| 2017-07-28 | 1.0  | 整理创建初稿         | 陈丽佳 |
| 2017-08-02 | 1.1  | 统一整理格式，并修正部分错误 | 陈丽佳 |

# 目录

|                       |   |    |
|-----------------------|---|----|
| 1                     | 说明  | 7  |
| 1.1                   | 前言  | 7  |
| 1.2                   | 代码总体原则  | 7  |
| 1.3                   | 规范实施、解释   | 7  |
| 1.4                   | 术语定义  | 7  |
| 2                     | 代码风格  | 8  |
| 2.1                   | 命名  | 8  |
| 原则 1                  | 为包、类、方法、变量取一个好名字，使代码易于理解  | 8  |
| 规则 1                  | 禁止使用魔鬼数字  | 8  |
| 规则 2                  | 常量命名，由全大写单词组成，单词间用下划线分隔，且使用 <code>static</code>   | 9  |
| <code>final</code> 修饰 |   | 9  |
| 规则 3                  | 变量、属性命名，使用名词，并采用首字母小写的驼峰命名法   | 9  |
| 规则 4                  | 方法的命名，用动词和动宾结构，并采用首字母小写的驼峰命名法   | 10 |
| 规则 5                  | 类和接口的命名，采用首字母大写的驼峰命名法   | 10 |
| 规则 6                  | 包的命名，由一个或若干个单词组成，所有的字母均为小写  | 11 |
| 建议 1                  | 数组声明的时候使用 <code>int[] index</code> ，而不要使用 <code>int index[]</code>  | 11 |
| 2.2                   | 注释  | 11 |
| 原则 2                  | 尽量用代码来解释自己  | 11 |
| 规则 7                  | 注释应解释代码的意图，而不是描述代码怎么做的  | 11 |
| 规则 8                  | 保证注释与代码一致，避免产生误导  | 12 |
| 规则 9                  | 注释应与其描述代码位置相邻，放在所注释代码上方或右方，并与代码采用同样缩进   | 12 |
| 建议 2                  | 不要用注释保留废弃代码   | 12 |
| 建议 3                  | 不要用注释记录修改日志   | 13 |
| 建议 4                  | 一般单行注释用 <code>//</code> ，块注释用 <code>/* */</code> ，JavaDoc 注释用 <code>/** */</code>   | 13 |
| 2.3                   | 排版  | 14 |
| 原则 3                  | 团队应遵守一致的排版风格  | 14 |
| 规则 10                 | 将排版风格固化到 IDE 的代码格式化配置文件中，并让整个团队使用   | 14 |
| 规则 11                 | 在不同的概念之间，增加空行   | 14 |
| 规则 12                 | 将逻辑紧密相关的代码放在一起  | 16 |
| 规则 13                 | 控制一行的宽度，不要超过 120 个字符  | 17 |
| 规则 14                 | 在不同的概念间（关键字、变量、操作符等）增加空格，以便清楚区分概念   | 18 |
| 规则 15                 | 采用缩进来区分不同层次的概念  | 18 |
| 建议 5                  | 将局部变量的作用域最小化  | 20 |
| 建议 6                  | 给 <code>if</code> 、 <code>for</code> 、 <code>do</code> 、 <code>while</code> 、 <code>switch</code> 等语句的执行体加大括号 <code>{}</code> | 22 |
| 建议 7                  | 控制文件的长度，最好不要超过 500 行  | 22 |
| 3                     | 变量和类型   | 23 |
| 原则 4                  | 谨慎使用静态成员变量  | 23 |

|                |  |    |
|----------------|--|----|
| 规则 16          | 避免随意进行类型强制转换，应改善设计，或在转换前用 <code>instanceof</code>                      |    |
| 进行判断           | 23   |    |
| 规则 17          | 需要精确计算时不要使用 <code>float</code> 和 <code>double</code> .....             | 24 |
| 规则 18          | 不能用浮点数作为循环变量 .....   | 25 |
| 规则 19          | 浮点型数据判断相等不能直接使用 <code>==</code> .....                                  | 25 |
| 规则 20          | 避免同一个局部变量在前后表达不同的含义 .....  | 25 |
| 规则 21          | 不要在单个的表达式中对相同的变量赋值超过一次 .....   | 26 |
| 建议 8           | 基本类型优于包装类型，注意合理使用包装类型 .....  | 27 |
| 4 方法 .....     |  | 29 |
| 原则 5           | 方法设计的第一原则是要短小 .....  | 29 |
| 原则 6           | 方法设计应遵循单一职责原则（SRP），一个方法仅完成一个功能 .....                                   | 29 |
| 原则 7           | 方法设计应遵循单一抽象层次原则（SLAP） .....  | 30 |
| 原则 8           | 方法设计应遵循命令与查询职责分离原则（CQRS） .....   | 30 |
| 规则 22          | 不要把方法的入参当做工作变量/临时变量，除非特别需要 .....                                       | 31 |
| 规则 23          | 使用类名调用静态方法，而不要使用实例或表达式来调用 .....  | 32 |
| 建议 9           | 应明确规定对接口方法参数的合法性检查由调用者负责还是由接口方法本身负责 .....                              | 33 |
| 建议 10          | 方法的参数个数不宜过多 .....  | 33 |
| 建议 11          | 谨慎使用可变数量参数的方法 .....  | 34 |
| 5 包、类和接口 ..... |  | 35 |
| 原则 9           | 类和接口的设计应遵循面向对象 SOLID 设计原则 .....  | 35 |
| 原则 10          | 类的设计应遵循迪米特法则 .....   | 36 |
| 原则 11          | 类的设计应遵循“Tell, Don't ask”原则 .....                                       | 36 |
| 原则 12          | 类设计时优选组合而不是继承 .....  | 37 |
| 规则 24          | 除提供给外部使用的全局常量外，应尽量避免类成员变量被外部直接访问 .....                                 | 37 |
| 规则 25          | 避免在无关的变量或无关的概念之间重用名字，避免隐藏（hide）、遮蔽（shadow）和遮掩（obscure） .....           | 38 |
| 规则 26          | 不要在父类的构造方法中调用可能被子类覆写的方法 .....  | 40 |
| 规则 27          | 覆写 <code>equals</code> 方法时，应同时覆写 <code>hashCode</code> 方法 .....        | 41 |
| 规则 28          | 子类覆写父类方法时应加上 <code>@Override</code> 注解 .....                           | 42 |
| 建议 12          | 接口定义中去掉多余的修饰词 .....  | 42 |
| 建议 13          | 设计时，考虑类的可变性最小化 .....   | 42 |
| 6 异常和日志 .....  |  | 43 |
| 6.1 异常 .....   |  | 43 |
| 原则 13          | 只针对真正异常的情况才使用 <code>exception</code> 机制 .....                          | 43 |
| 规则 29          | 在抛出异常的细节信息中，应包含能捕获失败的信息 .....  | 43 |
| 规则 30          | 对可恢复的情况使用受检异常（checked exception），对编程错误使用运行时异常（runtime exception） ..... | 43 |
| 规则 31          | 不要忽略异常 .....   | 44 |
| 规则 32          | 方法注释和文档中要包含所抛出异常的说明 .....  | 44 |
| 规则 33          | 方法抛出的异常，应该与本身的抽象层次相对应 .....  | 44 |
| 建议 14          | 对第三方 API 抛出大量各类异常进行封装 .....  | 45 |
| 建议 15          | 使用异常来做错误处理，而非错误码 .....   | 45 |

|       |  |    |
|-------|--|----|
| 规则 34 | 在 finally 块中不要使用 return、break 或 continue 使 finally 块非正常结束                    | 46 |
| 规则 35 | 不要直接捕获受检异常的基类 Exception  | 47 |
| 建议 16 | 一个方法不应抛出太多类型的异常  | 47 |
| 建议 17 | 充分利用断言   | 47 |
| 6.2   | 日志   | 48 |
| 原则 14 | 日志信息准确、繁简得当，满足快速定位的需要  | 48 |
| 规则 36 | 日志的记录，不要使用 System.out 与 System.err 进行控制台打印，应该使用专用的日志工具(比如：slf4j+logback)进行处理 | 48 |
| 规则 37 | 日志工具对象 logger 应声明为 private static final                                      | 48 |
| 规则 38 | 日志应分等级   | 48 |
| 规则 39 | 日志中不要记录敏感信息  | 49 |
| 7     | 多线程并发  | 50 |
| 规则 40 | 多线程访问同一个可变变量，需增加同步机制   | 50 |
| 规则 41 | 禁止不加控制地创建新线程   | 51 |
| 规则 42 | 创建新线程时需指定线程名   | 52 |
| 规则 43 | 使用 Thread 对象的 setUncaughtExceptionHandler 方法注册 Runtime 异常的处理者(v1.5+)         | 52 |
| 规则 44 | 不要使用 Thread.stop 方法，因为该方法本质是不安全的，使用它可能会导致数据遭到破坏                              | 53 |
| 规则 45 | 不要依赖线程调度器、线程优先级和 yield() 方法  | 55 |
| 规则 46 | 采用 Java1.5 提供新并发工具代替 wait 和 notify (v1.5+)                                   | 55 |
| 规则 47 | 使用线程安全集合在多线程间共享可变数据  | 55 |
| 建议 18 | 多线程操作同一个字符串相加，应采用 StringBuffer   | 55 |
| 建议 19 | 针对线程安全性，需要进行文档(javadoc)说明  | 56 |
| 8     | 语言特性   | 57 |
| 8.1   | 运算和表达式   | 57 |
| 规则 48 | 不要写复杂的表达式  | 57 |
| 规则 49 | 运算时应避免产生溢出   | 57 |
| 规则 50 |  | 57 |
| 建议 20 | 采用括号明确运算的优先级   | 58 |
| 8.2   | 控制语句   | 58 |
| 规则 51 | 采用 for-each 代替传统的 for 循环 (v1.5+)   | 58 |
| 规则 52 | 在 switch 语句的每一个 case、和 default 中都放置一条 break 语句                               | 59 |
| 8.3   | 序列化  | 60 |
| 原则 15 | 尽量不要实现 Serializable 接口   | 60 |
| 规则 53 | 序列化对象中的 HashMap、HashSet 或 Hashtable 等集合不能包含对象自身的引用                           | 60 |
| 建议 21 | 实现 Serializable 接口的可序列化类应该显式声明 serialVersionUID                              | 61 |
| 8.4   | 泛型   | 62 |
| 规则 54 | 在集合中使用泛型 (v1.5+)   | 62 |
| 建议 22 | 类的设计可优先考虑泛型 (v1.5+)  | 62 |
| 建议 23 | 方法的设计可优先考虑泛型 (v1.5+)   | 63 |
| 建议 24 | 优先使用泛型集合，而不是数组 (v1.5+)   | 63 |

|       |   |    |
|-------|---|----|
| 8.5   | 其他语言特性.....   | 64 |
| 规则 55 | 新代码不要使用已标注为@deprecated 的方法 .....  | 64 |
| 建议 25 | 使用 JDK 自带的 API 或广泛使用的开源库，不要自己写类似的功能。<br>64  |    |
| 建议 26 | 升级到最新稳定版的 Java 平台版本上，以便获取新特性带来的收益<br>64   |    |
| 建议 27 | 充分利用编译器的告警选项 .....  | 64 |
| 建议 28 | 使用字符串 API 时，应注意方法使用的是否是“正则表达式”.....   | 64 |
| 建议 29 | 值类（value classes）的设计，可考虑实现 Comparable 接口，方便在<br>集合中实现对象的搜索、排序、计算极值等 .....   | 64 |
| 9     | 性能与资源管理.....  | 66 |
| 9.1   | 性能.....   | 66 |
| 原则 16 | 谨慎地进行性能优化 .....   | 66 |
| 规则 56 | 使用 System.arraycopy() 进行数组复制 .....  | 66 |
| 规则 57 | 使用集合的 toArray() 方法将集合转为数组（v1.42+） .....   | 66 |
| 建议 30 | 在 Java 的 IO 操作中，尽量使用带缓冲的实现 .....  | 67 |
| 9.2   | 资源管理.....   | 67 |
| 规则 58 | 避免创建不必要的对象 .....  | 67 |
| 规则 59 | 将对象存入 HashSet，或作为 key 存入 HashMap(或 Hashtable)后，必<br>须确保该对象的 hashCode 值不变，避免因为 hashCode 值变化导致不能从集合内删<br>除该对象，进而引起内存泄漏的问题 ..... | 68 |
| 规则 60 | 执行 IO 操作时，应该在 finally 里关闭 IO 资源 .....   | 69 |
| 规则 61 | 消除过期的对象引用 .....   | 70 |
| 10    | 可移植性.....   | 71 |
| 规则 62 | 不要在代码中硬编码“\n”和“\r”作为换行符号 .....  | 71 |
| 建议 31 | 谨慎地使用本地方法 .....   | 71 |
| 建议 32 | 避免对第三方代码的强依赖或陷入第三方代码细节 .....  | 71 |
| 11    | 国际化.....  | 72 |
| 规则 63 | 在所有的输入输出环节，指明正确的编码方式，进行正确的字符到字<br>节，或字节到字符的转换 .....   | 72 |
| 规则 64 | 如果输入源或输出目标直接支持，尽可能直接使用 Unicode 进行输入<br>输出。 72   |    |
| 规则 65 | 不要依赖平台默认的字符编码方式。 .....  | 72 |
| 规则 66 | 对于使用默认编码方式的第三方代码或者遗留代码，可应用适配器模<br>式，将返回的字符串转换成 Unicode 内码 .....   | 73 |
| 建议 33 | 字符串大小写转换时，应加上 Locale.US .....   | 73 |

# 1 说明

## 1.1 前言

随着业务的发展和产品架构的演进，越来越多的产品使用 JAVA 语言，同时 JAVA 语言也在不断发展，增加了很多新语言特性，如泛型、注解等。但目前 JAVA 使用现状是：基础技能薄弱，陷入很多误区，不能很好地发挥 JAVA 的作用。而原公司 JAVA 语言编程规范内容更偏向于对排版的要求，对 JAVA 语言特性使用注意事项上描述较少。为了帮助团队合理使用 JAVA，规避语言陷阱，特修订本规范。

## 1.2 代码总体原则

和其他语言编程一样，JAVA 编程遵循通用原则：

- 1、清晰第一。清晰性是易于维护、易于重构的程序必需具备的特征。
- 2、简洁为美。简洁就是易于理解并且易于实现。
- 3、选择合适的风格，团队保持一致。

除此之外，JAVA 编程还应该注意以下方面：

### 1、正确使用 JAVA

面向对象技术使得程序结构清晰、简单，提高了代码的重用性，但又隐藏了很多内部实现细节，不小心会误入陷阱。使用许多 JAVA 特性时，要注意正确使用。比如：多线程并发、泛型、装箱数据类型、异常处理、对象克隆等。

### 2、安全高效

## 1.3 规范实施、解释

本规范制定了编写 JAVA 语言程序的基本原则、规则和建议。

本规范适用于公司内使用 JAVA 语言编码的所有软件。本规范自发布之日起生效，对以后新编写的和修改的代码应遵守本规范。

在某些情况下需要违反本规范给出的规则时，相关团队必须通过一个正式的流程来评审、决策规则违反的部分，个体程序员不得违反本规范中的相关规则。

## 1.4 术语定义

**原则：**编程时必须坚持的指导思想。

**规则：**编程时强制必须遵守的约定。

**建议：**编程时必须加以考虑的约定。

**说明：**对此原则/规则/建议进行必要的解释。

**示例：**对此原则/规则/建议从好、不好两个方面给出例子。

本规范适用的 JDK 版本，不特别说明均为 1.4 以上版本适用。

## 2 代码风格

### 2.1 命名

**原则1 为包、类、方法、变量取一个好名字，使代码易于理解**

说明：好的命名，有如下特征：

- 1) **能清晰的表达意图**：使用完整的描述性的单词，避免使用单个字母、未形成惯例的缩写来命名。例如 `int elapsedTimeInDays` 要比 `int d` 好得多；
- 2) **避免造成误导**：有误导的名字比表达不清的名字还要有危害性，比如 `String accountList` 其实并不是一个 `List` 类型；`a=1`，是数字 `1` 还是字母 `1`？`TTLCONFUSION` 与 `TTLCONFUSION` 名称太相似；
- 3) **避免不必要的编解码**：代码被人阅读的次数要远远多于计算机，因此要注意可读性，避免不必要的人脑编解码。比如在 `JAVA` 中不建议采用匈牙利命名法：`Java` 是强类型语言，且 `IDE` 已很先进，在编译前就能及早发现类型错误，因此匈牙利命名法已无用武之地，况且它还有可能导致错误信息，比如 `PhoneNumber strPhone` 当类型变更后，名称未变更导致提供错误信息；
- 4) **能区分出意思**：比如以下名称：`product`，`productInfo`，`productData`，表达的意思都差不多，让人从名称并不能区分出它们各自代表的东西到底有什么不同。因此建议不要在变量/类名后加 `info`，`data`，`object` 等一般意义的词。
- 5) **不用或少用缩写**：小于 15 个字母的一般不用缩写。超过 15 个字母的，可采用去掉元音字母的方法或者行业内约定俗成的缩写，且缩写保持驼峰格式，不要编造不符合惯例的缩写。比如 `serviceDataPoint`，可以缩写为 `svcDataPnt`，不可缩写为 `SDP`。如果要用缩写，仅第一个字母大写其余小写，例如 `getHttpRequest` 改为 `getHttpRequest`。

**规则1 禁止使用魔鬼数字**

说明：直接使用数字，造成代码难以理解，也难以维护。应采用有意义的静态变量或枚举来代替。例外情况，有些特殊情况下，如循环或比较时采用数字 `0`，`-1`，`1`，这些情况可采用数字。

示例：

不好：

```
int doorState = 5;
```

推荐：

```
static final int CLOSE = 3;  
int doorState = CLOSE;
```



```
enum Signal
{
    GREEN, YELLOW, RED
}

public class TrafficLights
{
    Signal color = Signal.RED;
    public void change()
    {
        switch (color)
        {
            case GREEN:
                color = Signal.YELLOW;
                break ;
            case RED:
                color = Signal.GREEN;
                break ;
            case YELLOW:
                color = Signal.RED;
                break ;
            default:
                color = Signal.RED;
        }
    }
}
```

**规则2 常量命名**，由全大写单词组成，单词间用下划线分隔，且使用 **static final** 修饰

示例：

不好：

```
static int MAXUSERNUM = 200;
static String s = "Launcher";
```

推荐：

```
static final int MAX_USER_NUM = 200;
static final String APPLICATION_NAME = "Launcher";
```

**规则3 变量、属性命名**，使用名词，并采用首字母小写的驼峰命名法

说明：驼峰命名是指第一个单词字母使用小写，剩余单词首字母大写其余字母小写的大小写混合法。含有集合意义的属性，名称尽量包含复数；

示例：

不好：

```
String customername;  
List<String> u = new ArrayList<String>();
```

推荐：

```
String customerName;  
List<String> users = new ArrayList<String>();
```

#### 规则4 方法的命名，用动词和动宾结构，并采用首字母小写的驼峰命名法

说明：格式如下

get + 非布尔属性名()

is + 布尔属性名()

set + 属性名()

has + 名词/形容词()

动词()

动词 + 宾语()

其中动词()主要用在动作作用在对象自身，如 document.print();

示例：

不好：

```
public String type()  
public boolean Finished()  
public void visible(boolean)  
public void DRAW()  
public void KeyListener(listener)
```

推荐：

```
public String getType()  
public boolean isFinished()  
public void setVisible(boolean)  
public void draw()  
public void addKeyListener(listener)
```

#### 规则5 类和接口的命名，采用首字母大写的驼峰命名法

说明：类的命名，不应用动词，而应使用名词，比如 Customer，WikiPage，Account，避免采用类似 Manager，Processor，Data，Info 这样模糊的词。

示例：

不好：

```
public class info {}
```

推荐：

```
public class OrderInformation {}
```

## 规则6 包的命名，由一个或若干个单词组成，所有的字母均为小写

**说明：**包名采用域后缀倒置的加上自定义的包名，采用小写字母，都应该以 `com.company` 开头（除一些特殊原因），再加上产品名称和模块名称。部门内部应该规划好包名的范围，防止产生冲突。

**示例：**`com.company.mobilecontrol.views;`

## 建议1 数组声明的时候使用 `int[] index`，而不要使用 `int index[]`

**说明：**例如：`int[] index = new index[8];`比`int index[] = new index[8];`看起来更符合人的阅读习惯。

# 2.2 注释

## 原则2 尽量用代码来解释自己

**说明：**我们必须认识到，写注释从某种意义上来说是一种“失败”，是我们无法用代码来解释意图，而必须借助于注释。因此在写注释前，要慎重思考，看能否通过改善代码可读性来避免写注释。

**示例：**

**不好：**

```
// Check to see if the employee is eligible for full benefits
if ((employee.flags & HOURLY_FLAG) &&
    (employee.age > 65))
```

**推荐：**

```
if (employee.isEligibleForFullBenefits())
```

## 规则7 注释应解释代码的意图，而不是描述代码怎么做的

**说明：**如果不得不写注释，那么就要写出好的注释。好的注释能提供有用、额外的信息，能解释代码为什么要这么写，而不是再描述一遍代码是怎么做的。

**示例：**

**不好：**下面的注释只是将代码所做的事又描述了一遍，没有提供额外有用的信息

```
//Utility method that returns when this.closed is true. Throws an exception
//if the timeout is reached.
public synchronized void waitForClose(final long timeout)
    throws Exception
{
    if (!closed)
    {
        wait(timeout);
        if (!closed) throw new Exception("ResponseSender could not be closed");
    }
}
```

**规则8 保证注释与代码一致，避免产生误导**

说明：注释造成误导，危害性很大，还不如不写。很多误导的产生，并不是有意为之，而是在代码修改的同时没有修改对应的注释造成的。因此，如果写了注释，就要保证注释与代码一致，避免产生误导。如果注释不再有用，必须删除。

**规则9 注释应与其描述代码位置相邻，放在所注释代码上方或右方，并与代码采用同样缩进**

说明：大部分注释应放在代码上方，变量、枚举的注释可选择在代码右方；

示例：

```
static final int THREAD_NUMBER = 25000;
...
//This is our best attempt to get a race condition
//by creating large number of threads.
for (int i = 0; i < THREAD_NUMBER; i++)
{
    WidgetBuilderThread widgetBuilderThread =
        new WidgetBuilderThread(widgetBuilder, text, parent, failFlag);
    Thread thread = new Thread(widgetBuilderThread);
    thread.setName("Widget Builder thread");
    thread.start();
}
```

```
Image buffer; //定义图像对象
Graphics gContext; //定义图形上下文
Thread animate; //定义一个线程
```

**建议2 不要用注释保留废弃代码**

说明：用注释保留大量废弃的代码，对阅读是一种干扰。现代的配置管理工具能恢复任意历史时刻的代码，因此不必担心无法恢复。

示例：

不好：

```
mModel.reloadIcons();
/* DTSyyyyyy xxxxx end > */
/* < DTSyyyyyy xxxxx begin */
//if (!mModel.isAllAppsLoaded()) {
//ViewGroup appsCustomizeContentParent = (ViewGroup) mAppsCustomizeContent.getParent();
//mInflater.inflate(R.layout.apps_customize_progressbar, appsCustomizeContentParent);
//}
/* DTSyyyyyy xxxxx end > */

// For handling default keys
mDefaultKeySsb = new SpannableStringBuilder();
```

### 建议3 不要用注释记录修改日志

说明：代码中充斥着大量修改历史信息，使代码难以阅读。现代的配置管理工具能记录每次修改的日志，因此不必将修改日志写在代码中。

示例：

不好：

```
/*< DTSyyyyyy xxxxxx begin*/  
  
private boolean mAlreadyShowFinish = false;  
private boolean mIsClickEmptySpace = false;  
  
/*DTSyyyyyy xxxxxx end >*/  
/*< DTSyyyyyy xxxxxx begin*/  
  
public View itemUnderLongClick ;  
  
/* DTSyyyyyy xxxxxx end>*/  
/*< DTSyyyyyy xxxxxx begin*/  
  
public int DOCK_SIZE_DRAG_BEFORE = 4;  
  
/* DTSyyyyyy xxxxxx end >*/  
/*< DTSyyyyyy xxxxxx begin */  
  
public int mWindowWidth;  
public int mWindowHeigth;  
  
/* DTSyyyyyy xxxxxx end >*/  
/*< DTSyyyyyy xxxxxx begin */
```

### 建议4 一般单行注释用//，块注释用/\* \*/，JavaDoc 注释用/\*\* \*/

说明：单行注释用//，比较方便。块注释一般采用/\* \*/，在 IDE 支持的情况下，如 Eclipse 可以用 ctrl-/快捷键为块增加注释，这种情况下，建议选择//为块增加注释，更方便快捷。JavaDoc 对注释格式要求采用/\*\* \*/

示例：

```
/**  
 * This is the comment for the example interface.  
 */  
  
interface Example  
{  
    // This is a long comment with whitespace that should be split in multiple  
    // line comments in case the line comment formatting is enabled  
    int foo();  
  
    /*  
     * These possibilities include: <ul><li>Formatting of header  
     * comments.</li><li>Formatting of Javadoc tags</li></ul>  
     */  
    int bar(); // This is a short comment  
    ...  
}
```

## 2.3 排版

### 原则3 团队应遵守一致的排版风格

说明：代码排版在细微处，像审美一样是一个较主观的事情，比如大括号是写在右边，还是独立成行，不同的人有不同的倾向，并无优劣之分。但不统一的排版风格对代码的可阅读性影响很大，因此考虑到团队整体的阅读效率，应把个人喜好放一边，而讨论制定出共同遵守的排版风格。

### 规则10 将排版风格固化到 IDE 的代码格式化配置文件中，并让整个团队使用

说明：让团队内遵守一致的排版风格，最切实可行和有效率的方式就是大家使用同一个代码格式化配置，让大家在编写代码的同时，用 IDE 工具自动对代码进行排版。主流的 JAVA IDE，如 Eclipse, NetBeans, IntelliJ IDEA 都支持代码格式化功能。

### 规则11 在不同的概念之间，增加空行

说明：比如方法与方法、类名与 import、import 与包名之间、相对独立的程序块之间、变量说明后需增加空行，来提升可读性。

示例：

不好：不同概念间没有空行

```
public void addChild2(String parentId, String value)
{
    .....
    Entity entity = new entity();
    entity.setId(genEntityId());
    Relation relation = new Relation();
    relation.setParentId(parentId);
    saveChild(entity, relation);
}
```

```
package fitnessse.wikitext.widgets;
import java.util.regex.*;
public class BoldWidget extends ParentWidget
{
    public static final String REGEXP = "''.+?'";
    private static final Pattern pattern = Pattern.compile(REGEXP,
        Pattern.MULTILINE + Pattern.DOTALL);
    public BoldWidget(ParentWidget parent, String text) throws Exception
    {
        super(parent);
        Matcher match = pattern.matcher(text);
        match.find();
        addChildWidgets(match.group(1));
        public String render() throws Exception {
            StringBuffer html = new StringBuffer("<b>");
            html.append(childHtml()).append("</b>");
            return html.toString();
        }
    }
}
```

推荐：用空行区分不同概念

```
public void addChild2(String parentId, String value)
{
    Entity entity = new entity();
    entity.setId(genEntityId());

    Relation relation = new Relation();
    relation.setParentId(parentId);

    saveChild(entity, relation);
}
```

```
package fitnessse.wikitext.widgets;

import java.util.regex.*;

public class BoldWidget extends ParentWidget
{
    public static final String REGEXP = "''.+?'';

    private static final Pattern pattern = Pattern.compile(REGEXP,
        Pattern.MULTILINE + Pattern.DOTALL
    );

    public BoldWidget(ParentWidget parent, String text) throws Exception
    {
        super(parent);
        Matcher match = pattern.matcher(text);
        match.find();
        addChildWidgets(match.group(1));
    }

    public String render() throws Exception
    {
        StringBuffer html = new StringBuffer("<b>");
        html.append(childHtml()).append("</b>");
        return html.toString();
    }
}
```

## 规则12 将逻辑紧密相关的代码放在一起

说明：将相关的代码，放在一起，阅读代码时能一眼获取相关信息，避免概念的频繁切换。其他还比如，将常量的定义都放在一起，将有调用关系的方法尽量放在一起。

示例：

不好：



```
public void addChild(String parentId, String value)
{
    Entity entity = new Entity();
    Relation relation = new Relation();
    entity.setId(genEntityId());
    relation.setChildEntityId(entity.getId());
    saveChild(entity, relation);
}
```

推荐：

```
public void addChild(String parentId, String value)
{
    Entity entity = new Entity();
    entity.setId(genEntityId());

    Relation relation = new Relation();
    relation.setParentId(parentId);
    relation.setChildEntityId(entity.getId());

    saveChild(entity, relation);
}
```

### 规则13 控制一行的宽度，不要超过 120 个字符

说明：代码行长度越短，越容易理解。而且代码超出屏幕后，需要横向滚屏，严重影响阅读代码的效率。建议一行最长不超过 120 个字符。代码行太长，需要折行。折行时应注意语意流畅性，在低优先级操作符处划分新行，划分出的新行要进行适当的缩进，使排版整齐（详见示例）。

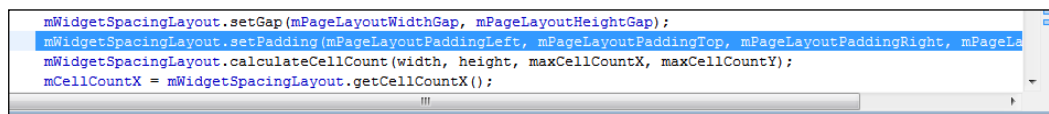
示例：

不好：

一行多个语句：

```
String xmlName = "", nodeVal = "";
```

过长，超过 120 字符，需要滚屏



```
mWidgetSpacingLayout.setGap(mPageLayoutWidthGap, mPageLayoutHeightGap);
mWidgetSpacingLayout.setPadding(mPageLayoutPaddingLeft, mPageLayoutPaddingTop, mPageLayoutPaddingRight, mPageLa
mWidgetSpacingLayout.calculateCellCount(width, height, maxCellCountX, maxCellCountY);
mCellCountX = mWidgetSpacingLayout.getCellCountX();
```

不必要的折行

```
String commandID = Configs.findItemValue(ItemCodes.BIZDEF_HARDCFG,
    bizDef);
```

### 混乱没有规则的折行

```
isLogRecode = null == logisticRecodeLog || "".equals(logisticRecodeLog) ||  
    "true".equals(logisticRecodeLog) || "TRUE".  
    equals(logisticRecodeLog)  
    || "YES".equals(logisticRecodeLog)
```

推荐：

一行一个语句

```
String xmlName = "";  
.....
```

### 过长语句合理折行

```
isLogRecode = null == logisticRecodeLog || "".equals(logisticRecodeLog)  
    || "true".equals(logisticRecodeLog) || "TRUE".equals(logisticRecodeLog)
```

## 规则14 在不同的概念间（关键字、变量、操作符等）增加空格，以便清楚区分概念

说明：增加空格，其本质是区分概念，将逻辑相关紧密的部分凸现出来。具体来说在关键字、变量、常量进行对等操作时，它们之间的操作符之前、之后或者前后要加空格；进行非对等操作时，如果是关系密切的立即操作符（如.）和括号，前后不加空格

示例：

在“=”、“+=”、“,”的两边加空格，“++”、“.”和“(”前后不加空格。

```
private void measureLine(String line)  
{  
    lineCount++;  
    int lineSize = line.length();  
    totalChars += lineSize;  
    .....  
    recordWidestLine(lineSize);  
}
```

## 规则15 采用缩进来区分不同层次的概念

说明：没有缩进的代码几乎不可阅读。我们采用缩进，来区分不同层次，展现代码的层次关系结构，让代码更容易阅读。具体要求，每加一层大括号，就要加一层缩进。

示例：

没有缩进的代码很难阅读

```
public class FitNesseServer implements SocketServer
{
    private FitNesseContext context;
    public FitNesseServer(FitNesseContext context)
    {
        this.context = context;
    }
    public void serve(Socket s)
    {
        serve(s, 10000);
    }
    public void serve(Socket s, long requestTimeout)
    {
        try { FitNesseExpediter sender = new FitNesseExpediter(s, context);
        sender.setRequestParsingTimeLimit(requestTimeout);
        sender.start();
        }
        catch(Exception e) {
        e.printStackTrace();
        }
    }
}
```

改成下面就容易阅读多了

```
public class FitNesseServer implements SocketServer
{
    private FitNesseContext context;

    public FitNesseServer(FitNesseContext context)
    {
        this.context = context;
    }

    public void serve(Socket s)
    {
        serve(s, 10000);
    }

    public void serve(Socket s, long requestTimeout)
    {
        try
        {
            FitNesseExpediter sender = new FitNesseExpediter(s, context);
            sender.setRequestParsingTimeLimit(requestTimeout);
            sender.start();
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

#### 建议5 将局部变量的作用域最小化

说明：变量的声明应该尽可能靠近变量第一次使用的位置。循环变量的定义，应在循环体内部，不应该定义在循环体外。

示例：

不好：变量声明离第一次使用的位置过远

```
public void addChild(String parentId, String value)
{
    Relation relation = new Relation();
    entity entity = new entity();
    entity.setId(genEntityId());
    entity.setValue(value);

    ...
    relation.setParentId(parentId);
    relation.setChildEntityId(entity.getId());

    saveChild(entity, relation);
}
```

推荐：在靠近变量第一次使用处声明变量

```
public void addChild(String parentId, String value)
{
    entity entity = new entity();
    entity.setId(genEntityId());
    entity.setValue(value);

    ...
    Relation relation = new Relation();
    relation.setParentId(parentId);
    relation.setChildEntityId(entity.getId());

    saveChild(entity, relation);
}
```

循环变量在循环体内定义

```
public int countTestCases()
{
    int count = 0;
    for (Test each : tests)
    {
        count += each.countTestCases();
    }
    return count;
}
```

**建议6 给 if、for、do、while、switch 等语句的执行体加大括号{}**

说明：即使执行语句只有一条语句，也应该加括号。执行体只有一条语句时，常诱惑我们在一行内写完，但这并不容易阅读，因为大多数程序员习惯了执行体应该有括号，当执行体无括号，需要大脑做不必要的转换。而且这样做，在修改代码时，如果增加执行体语句，也不容易因忘了增加大括号而搞错。

示例：

不好

```
if (someCondition) doStart();
```

推荐

```
if (someCondition)
{
    doStart();
}
```

**建议7 控制文件的长度，最好不要超过 500 行**

说明：很多开源项目都表明，文件长度越短，越容易理解和维护。

示例：下图是几个开源项目的代码文件长度分布情况，平均值不超过 500 行，绝大多数不超过 200 行。

## 3 变量和类型

### 原则4 谨慎使用静态成员变量

说明：要谨记，静态成员变量是属于类级别的变量，而不是属于某个对象实例。

错误使用静态成员变量可能有以下场景：

1、认为静态变量是属于某个实例，而实际是多个实例操作同一个变量，造成值与预期不一致。

2、没有注意静态变量的初始化顺序，读取还未初始化的静态变量值。例如下面代码的执行顺序是，A、第一行的 INSTANCE 被 main 方法触发，创建一个新的实例；B、类构造方法被触发，beltSize 被计算，此时 CURRENT\_YEAR 还是未初始化状态，默认值为 0，导致 beltSize 被计算为-1930。

```
public class Elvis
{
    public static final Elvis INSTANCE = new Elvis();
    private final int beltSize;
    private static final int CURRENT_YEAR = Calendar.getInstance().get(Calendar.YEAR);

    private Elvis()
    {
        beltSize = CURRENT_YEAR - 1930;
    }

    public int beltSize()
    {
        return beltSize;
    }

    public static void main(String[] args)
    {
        System.out.println("Elvis wears a size " + INSTANCE.beltSize() + " belt.");
    }
}
```

推荐在以下场景中，合理使用静态变量：

1、类的所有实例必须共享同一个变量时，比如，为实现某个任务，该类的所有实例共用的变量，如计数器等；

2、工具类提供的常量，如配置文件中的参数 "映射"到类的变量时，基本上第一次赋值后，数据不再被修改；

3、单例模式中应用。

### 规则16 避免随意进行类型强制转换，应改善设计，或在转换前用 instanceof 进行判断

说明：没有判断直接进行类型转换，可能会因类型不匹配而导致运行期异常 java.lang.ClassCastException。

简单的修改方法即是在强制转换之前使用 `instanceof` 进行判断，确认转换操作可行，但可能带来的问题是修改点过多，工作量巨大，同时维护的工作量也会倍增。最好的方式还是改善设计，使集合中只有同一种类型的对象。

示例：

不好：

```
List datas = new ArrayList();
datas.add("Nicole");
datas.add("Bally");
datas.add(1);
datas.add(3);

for (Object o : datas)
{
    System.out.println((String)o);
}
```

推荐：分为 2 个方法

方法 1：

```
List<String> names = new ArrayList<String>(INIT_SIZE);
names.add("Nicole");
names.add("Bally");

for (String name : names)
{
    System.out.println(name);
}
```

方法 2：

```
List<Integer> prices = new ArrayList<Integer>(INIT_SIZE);
prices.add(1);
prices.add(3);

for (Integer price : prices)
{
    System.out.println(price);
}
```

#### 规则17 需要精确计算时不要使用 `float` 和 `double`

**说明：**浮点数在一个范围很广的值域上提供了很好的近似，但是它不能产生精确的结果。二进制浮点数对于精确计算是非常不适合的，因为它不可能将 0.1，或者 10 的其它任何负次幂表示为一个长度有限的二进制小数。涉及精确的数值计算（货币、金融等），建议使用 `int`、`long`、`BigDecimal` 等



示例：

不好：以下输出结果是 0.6100000000000001

```
System.out.println(1.03 - 0.42);
```

推荐：需要精确计算时

```
BigDecimal income = new BigDecimal("1.03");  
BigDecimal expense = new BigDecimal("0.42");
```

#### 规则18 不能用浮点数作为循环变量

说明：浮点数不能为循环因子，精度问题会导致 `(float)2000000000 == 2000000050` 为 `true`，所以如下的循环不会执行。

```
for (float f = (float) 2000000000; f < 2000000050; f++)  
{  
    ...  
}
```

#### 规则19 浮点型数据判断相等不能直接使用==

说明：由于浮点数在计算机表示中存在精度的问题，因此，判断 2 个浮点数相等不能直接使用等号可以采用如下方式：

```
float a =...;  
float b =...;  
if (Math.abs(a-b) < 1E-6f)  
{  
    ...  
}
```

其中 `1E-6f` 为一个 `float` 极小值，实际使用时请根据情况判断精度，并且提取常量。如果是 `double`,请使用 `1E-6`。

#### 规则20 避免同一个局部变量在前后表达不同的含义

说明：同一个局部变量在前后表达不同的含义，会使代码理解起来比较困难，不利于维护。一个局部变量只应该表达一种含义。

示例：

不好：

```
public void foo()
{
    Entity entity;

    if (CollectionUtils.isNotEmpty(entityList))
    {
        entity = findFirst(entityList);
        //some code

        entity = findLast(entityList);
        //some other code
    }
}
```

推荐：

```
public void foo()
{
    Entity firstEntity;
    Entity lastEntity;

    if (CollectionUtils.isNotEmpty(entityList))
    {
        firstEntity = findFirst(entityList);
        //some code

        lastEntity = findLast(entityList);
        //some other code
    }
}
```

## 规则21 不要在单个的表达式中对相同的变量赋值超过一次

说明：对相同的变量进行多次赋值的表达式会产生混淆，并且很少能够产生你希望的行为。清晰的变量赋值会使代码更易懂，程序也会产生我们预期的行为。

示例：

不好：输出为 0，不符合预期

```
public class Increment
{
    public static void main(String[] args)
    {
        int t = 0;
        for (int i = 0; i < 100; i++)
        {
            t = t++;
        }
        System.out.println(t);
    }
}
```

推荐：输出为 100

```
public class Increment
{
    public static void main(string[] args)
    {
        int t = 0;
        for (int i = 0; i < 100; i++)
        {
            t++;
        }
        System.out.println(t);
    }
}
```

#### 建议8 基本类型优于包装类型，注意合理使用包装类型

**说明：**Java 有两种类型，基本类型（Primitive type）和引用类型（Reference type）。基本类型如 `boolean`, `int`, `double`，引用类型如 `String`, `List`。每一种基本类型都有其对应的包装类型（Wrapper classes），如对应 `int` 的是 `Integer`。

很多情况下基本类型优于装箱基本类型，因为：

1、在 JDK 1.5 以及之后的版本中增加了自动装箱和拆箱的特性。但是，不恰当的并行使用基本类型和包装类型，可能带来大量隐含的装箱和拆箱的操作。如下面的 `for` 语句中，由于循环变量是基本类型，而 `sum` 是包装类型，会导致频繁的自动装箱和拆箱操作，导致性能下降。

```
Long sum = 0L;
for (long i = 0; i < Integer.MAX_VALUE; i++)
{
    sum += i;
}
System.out.println(sum);
```

2、对于包装类型，使用“==”比较可能无法得到预期的结果。如：`new Integer(42) == new Integer(42)` 将返回 `false`。应该使用 `equals` 方法做装箱类型的比较。

使用包装类型合理的场景有：

- 1、作为集合中的元素、键和值
- 2、泛型，必须使用包装类型，如 `List<Integer>`
- 3、进行反射的方法调用时

## 4 方法

### 原则5 方法设计的第一原则是要短小

说明：方法设计的第一原则，是短小，第二原则是还要短小。过长的方法，难以理解和维护。短方法易理解、易维护、易扩展、易测试。根据业界经验，方法的长度建议不超过 50 行。

### 原则6 方法设计应遵循单一职责原则（SRP），一个方法仅完成一个功能

说明：方法应该做一件事，做好这件事，只做这件事。有以下 2 种情况，违反单一职责：1、多段代码重复做同一件事情，那么在方法的划分上可能存在问题，应将重复部分提取为一个方法。2、一个方法完成了多种功能，应将其拆分为多个步骤的子功能。

示例 1：代码重复

```
public boolean save() throws IOException
{
    if (outputFile == null)
    {
        return false;
    }
    FileWriter writer = new FileWriter(outputFile);
    movies.writeTo(writer);
    writer.close();
    return true;
}

public boolean saveAs() throws IOException
{
    outputFile = view.getFile();
    if (outputFile == null)
    {
        return false;
    }
    FileWriter writer = new FileWriter(outputFile);
    movies.writeTo(writer);
    writer.close();
    return true;
}
```

示例 2：下面的方法，不仅判断了用户名和密码，同时也设置了错误信息。可以将设置错误信息独立出来，根据本方法的返回值，在外面设置。

```
public class UserManager
{
    .....
    public boolean checkpassword(String username, String password)
    {
        String foundPassword = find(username);
        if (foundPassword == null || !foundPassword.equals(password))
        {
            errorMessage = "Incorect username or password. Please try again.";
            return false;
        }

        errorMessage = "成功登录";
        return true;
    }
}
```

#### 原则7 方法设计应遵循单一抽象层次原则（SLAP）

说明：SLAP 原则，是指让一个方法中所有的操作处于相同的抽象层。否则跳跃的代码的抽象层次破坏了代码的流畅性，如下所示。

示例 1：方法的操作不在同一个抽象层次，前后是抽象，中间是细节。

```
void compute ()
{
    input ();
    flags = 0x0080;
    output ();
}
```

示例 2：方法的操作在同一个抽象层次上

```
void compute ()
{
    input ();
    process ();
    output ();
}
```

#### 原则8 方法设计应遵循命令与查询职责分离原则（CQRS）

说明：方法应该修改某对象的状态，或者返回该对象的有关信息，但二者不可兼得。也就是说：如果我们要问一个问题，那么就不应该影响到它的答案。实际应用，要视具体情况而定，语义的清晰性和使用的简单性之间需要权衡。将 Command 和 Query 功能合并入一个方法，方便了客户的使用，但是，降低了清晰性，而且，可能不便于基于断言的程序设计并且需要

一个变量来保存查询结果。

示例：

不好：方法设定指定属性的值，如果成功返回 `true`，如果指定属性不存在，则返回 `false`。这样导致了 `if` 语句理解存在歧义，可能会被理解为“`username` 是否已被设置为 `michael`”，而不是“看看 `username` 属性是否存在，如果存在则设置 `username` 为 `michael`”

```
public boolean set(String attribute, String value);

if (set("username", "michael"))
{
    ...
}
```

推荐：将命令和查询的职责分离为 2 个方法，先查询，再命令，就很清晰无歧义了。

```
if (attributeExist("username"))
{
    setAttribute("username", "michael");
    ...
}
```

## 规则22 不要把方法的入参当做工作变量/临时变量，除非特别需要

说明：1、每个变量/参数都有自己独特的功用，让一个变量承担多个职责，变量名无法清晰表达其功能，会使程序难以理解；2、如果参数是传引用方式的，则方法内对参数的更改，会传递到方法外，造成意外的错误。如传引用时，不想方法内修改入参的，建议在参数前加 `final` 关键字；

例外：当以传引入的方式传递参数，而且需要调用者能获取到方法内修改的结果时，可用入参当工作变量；

示例：入参作为工作变量，程序难理解；

```
int sample(int inputVal)
{
    inputVal = inputVal * CurrentMuiniplier(inputVal);
    inputVal = inputVal * CurrentAdder(inputVal);
    return inputVal;
}
```

推荐：采用局部变量，仅为示意，不代表实际中应该命名为 `workingVal`，应该采用业务对应的名称。

```
int sample (int inputVal)
{
    int workingVal = inputVal;
    workingVal = workingVal * CurrentMultiplier(workingVal);
    workingVal = workingVal * CurrentAdder(workingVal);
    ...
    return workingVal;
}
```

### 规则23 使用类名调用静态方法，而不要使用实例或表达式来调用

说明：明确的使用类名调用静态方法不容易造成混淆。使用实例调用静态方法时，调用的静态方法是声明类型的静态方法，与实例的实际类型无关，可能会导致与预期的结果不一致。当父类和子类有同名静态方法时，声明父类变量引用子类实例，调用该静态方法时调用的是父类的静态方法，而非子类的静态方法。

示例：

不好，两次调用的都是 `Dog.bark()` 方法，输出 2 行 woof

```
class Dog
{
    public static void bark()
    {
        System.out.println("woof");
    }
}

class Basenji extends Dog
{
    public static void bark()
    {
        System.out.println("miao");
    }
}

public class Bark
{
    public static void main(String args[])
    {
        Dog woofer = new Dog();
        Dog nipper = new Basenji();
        woofer.bark();
        nipper.bark();
    }
}
```



推荐，用类名来调用静态方法

```
class Dog
{
    public static void bark()
    {
        System.out.print("woof");
    }
}

public class Bark
{
    public static void main(String args[])
    {
        Dog.bark();
    }
}
```

#### 建议9 应明确规定对接口方法参数的合法性检查由调用者负责还是由接口方法本身负责

说明：接口（public）方法调用时，应对参数做合法性检查，以保证程序的健壮性和正确性。对于预计不太可能发生的情况，使用断言检查，对于预计有可能发生情况，采用错误处理检查。

应明确规定参数检查由调用者负责，还是由方法本身负责。同时应避免出现调用者和方法本身都检查的情况，造成过度防御，降低程序性能，并降低代码可读性增大维护成本。

#### 建议10 方法的参数个数不宜过多

说明：如果参数超过 7 个，则维护的难度很大，建议减少参数个数。

如果多个参数同时多次出现在多个方法中，说明这些参数紧密相关，可以将它们封装到一个类中。

示例 1：产品中的真实代码，参数过多

```
private void insertHistory(String workfSeq, String workfType, String recSeq, int
sortOrder, String orderSeq, String workfID, int region, String subSys, String
orderId, String procSystem, String platType, String neID, String telnum, String
imsi, int orderPri, java.util.Date createTime, String preStatus, int maxProcnum,
int procNum, String rspCode, String rspDesc, int maxOvertime, String operType,
String curStatus, String neRetn, String neDesc, int procInterval, String
transid, String oprid, String prepayType, String feedback_cmd)
{
    ...
}
```

示例 2：把参数抽象为 某日志类 SpWorkfLog

```
private void insertHistory(SpWorkfLog spWorkfLog)
{
    ...
}
```

### 建议11 谨慎使用可变数量参数的方法

说明：在 JDK 1.5 版本中初次引入 Varargs（variable number of arguments）可变数量参数，可以接受指定类型的零个到多个参数。不建议使用 varargs 重写使用一个固定长度数组作为参数的方法，而应该在确实需要操作可变长度的值的序列时使用。

## 5 包、类和接口

### 原则9 类和接口的设计应遵循面向对象 SOLID 设计原则

#### 1、单一职责原则（Single Responsibility Principle）

说明：就一个类而言，应该仅有一个引起它变化的原因。如果你能够想到多于一个的动机去改变一个类，那么这个类就具有多于一个的职责。

示例：

职责不单一的类，有多于一个的变化原因：“SuperDashboard 类提供了对最后拥有焦点的组件的访问能力，我们还能通过它跟踪版本号和构建序列号”

```
public class SuperDashboard extends JFrame implements MetaDataUser
{
    public Component getLastFocusedComponent();
    public void setLastFocused(Component lastFocused);
    public int getMajorVersionNumber();
    public int getMinorVersionNumber();
    public int getBuildNumber();
}
```

职责单一的类

```
public class Version
{
    public int getMajorVersionNumber();
    public int getMinorVersionNumber();
    public int getBuildNumber();
}
```

#### 2、开放封闭原则（Open Closed Principle）

说明：开放封闭原则，指的是，新增功能时可以不修改原有的代码。也就是对扩展开放，对修改封闭。比如，可以通过实现一个已有的接口，或者继承一个已有的类，或者替换组合中的一个实现，来扩展新功能，而不修改原有的代码。

#### 3、里氏替换原则（Liskov Substitution Principle）

说明：里氏替换原则是指，子类的实例应该能够替换任何其超类的实例。违反此原则的经典例子是，从长方形中派生一个正方形。因为，正方形要求长宽一起变化，而长方形无此限制，则在长方形出现的地方，无法用正方形来替换。假设 `rectangle.setWidth(4); rectangle.setLength(5); rectangle.getArea();` 应该等于 20，而如果用正方形来替换长方形的话，结果为 25。

#### 4、接口分离原则（Interface Segregation Principle）

说明：使用多个专门的接口比使用单一的总接口要好。不要强迫用户使用他们不用的方法。一个类对另外一个类的依赖性应当是建立在最小的接口上的。一个接口代表一个角色，不应

当将不同的角色都交给一个接口。没有关系的接口合并在一起，形成一个臃肿的大接口，这是对角色和接口的污染。

## 5、依赖倒置原则（Dependency Inversion Principle）

说明：面向过程的开发中，高层直接调用底层，造成高层依赖于底层的具体实现。依赖倒置原则认为，高层应该调用抽象的接口，底层则应实现这个接口，两者都依赖于接口。这样解除了高层和底层的耦合。

### 原则10 类的设计应遵循迪米特法则

说明：迪米特法则（Law of Demeter）又叫最少知识原则（Least Knowledge Principle），初衷在于降低类之间的耦合。

类 C 的方法 f 只应该调用以下对象的方法：

1. 类 C 本身（this 对象）
2. 类 C 实体变量持有的对象（成员）
3. 在方法 f 中创建的对象
4. 作为参数传递给 f 的对象

示例：

迪米特法则说明

```
class C
{
    private A a;
    int func();

    public void f(B b)
    {
        this.func();           //类 C 本身的方法
        a.setActive();         //类 C 成员对象的方法
        b.invert(),            //传入 f 的参数对象的方法
        d = new D();
        d.doSomething();       //它创建的任何对象的方法
    }
}
```

不好：违反迪米特法则，类知道的东西太多，则下面任何一个方法的改变，都会影响到这个类，说明耦合太多。

```
final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();
```

### 原则11 类的设计应遵循“Tell, Don’t ask”原则

说明：你应该尽量告诉对象你希望它们去做的事情；而不要询问它们的状态之后做出决定，最后才告诉它们做什么事情

示例：

违反“Tell, Don’t ask”原则，询问对象状态再做判断

```
...  
if (person.getAddress().getCountry().equals("Australia"))  
...
```

遵循“Tell, Don't Ask”原则的代码，告诉对象去判断

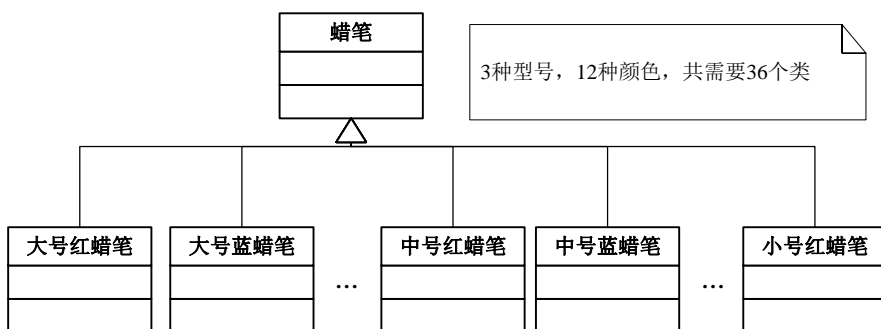
```
...  
if (person.livesIn("Australia"))  
...
```

## 原则12 类设计时优选组合而不是继承

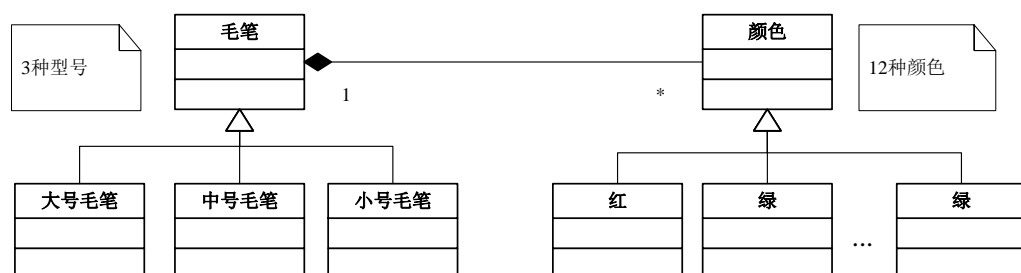
说明：只有当子类真正是超类的子类型时，即两者确实是“IS-A”关系时，才适合用继承。不宜用继承来实现复用，会造成子类数量过多，而采用组合来实现复用，则能实现动态组合，更加灵活；

示例：

继承会造成类爆炸：



采用组合则更加灵活：



## 规则24 除提供给外部使用的全局常量外，应尽量避免类成员变量被外部直接访问

说明：将变量设置为私有（private）的理由是：我们不想其他人依赖这个变量，依赖类内部的实现细节。这样，当内部实现需要变更时，影响面就比较小，变更的成本就比较低。

## 规则25 避免在无关的变量或无关的概念之间重用名字，避免隐藏（hide）、遮蔽（shadow）和遮掩（obscure）

说明：在声明子类的字段、方法或嵌套类型时，除了覆写（override）、重载（overload）之外，要尽量避免重名导致的隐藏（hide）、遮蔽（shadow）和遮掩（obscure）。

我们先来了解这些名字重用的术语列表。

### 覆写（override）——子类与父类间

一个实例方法可以覆写（override）在其超类中可访问到（非 private）的具有相同签名的实例方法（非 static），从而使能了动态分派（dynamic dispatch）；换句话说，VM 将基于实例的运行期类型来选择要调用的覆写方法。

```
class Base
{
    public void f() {}
}

class Derived extends Base
{
    @Override
    public void f() {} // overrides Base.f()
}
```

### 重载（overload）——类内部

在某个类中的方法可以重载（overload）另一个方法，只要它们具有相同的名字和不同的签名。由调用所指定的重载方法是在编译期选定的

```
class CircuitBreaker
{
    public void f(int i) {} // int overloading

    public void f(String s) {} // String overloading
}
```

### 隐藏（hide）——子类与父类间

一个属性、静态方法或内部类可以分别隐藏（hide）在其超类中可访问到的具有相同名字（对方法而言就是相同的方法签名）的所有属性、方法或内部类。上述成员被隐藏后，将阻止其被继承：

```

class Swan
{
    public static void fly()
    {
        System.out.println("swan can fly ...");
    }
}

class UglyDuck extends Swan
{
    public static void fly() //hide Swan.fly
    {
        System.out.println("ugly duck can't fly ...");
    }
}

public class TestFly
{
    public static void main(String[] args)
    {
        Swan swan = new Swan();
        Swan uglyDuck = new UglyDuck();
        swan.fly(); //打印 swan can fly ...
        uglyDuck.fly(); //还是打印 swan can fly ..., hide 让人以为是重载了，其实不是
    }
}

```

## 遮蔽（shadow）——类内部

一个变量、方法或类可以分别遮蔽（shadow）在类内部具有相同名字的变量、方法或类。如果一个实体被遮蔽了，那么就无法用简单名引用到它：

示例：

方法的局部变量遮蔽了类的静态变量

```
class WhoKnows
{
    static String sentence = "I don't know.";

    public static void main(String[] args)
    {
        String sentence = "I know!"; // 遮蔽了类的静态成员 sentence
        System.out.println(sentence); // 打印的是 I know!
    }
}
```

### 遮掩（obscure）——类内部

一个变量可以遮掩具有相同名字的一个类，只要它们都在同一个范围内：如果这个名字被用于变量与类都被许可的范围，那么它将引用到变量上。相似地，一个变量或一个类型可以遮掩一个包。遮掩是唯一一种两个名字位于不同的名字空间的名字重用形式，这些名字空间包括：变量、包、方法或类型。如果一个类或一个包被遮掩了，那么你不能通过其简单名引用到它，除非是在这样一个上下文环境中，即语法只允许在其名字空间中出现一种名字。遵守命名习惯就可以极大地消除产生遮掩的可能性

```
public class Obscure
{
    static String System; // Obscures type java.lang.System

    public static void main(String[] args)
    {
        //Next line won't compile: System refers to static field
        System.out.println("hello, obscure world!");
    }
}
```

### 规则26 不要在父类的构造方法中调用可能被子类覆写的方法

说明：当在父类构造方法中调用可能被子类覆写的方法时，构造方法的表现是不可预知的，很可能会导致异常。而问题出现后，又往往难以快速定位。这个问题是由于在 Java 中，当子类初始化的时候，会调用父类的构造方法，当构造方法调用了被子类覆写的方法，往往会由于子类的初始化未完成而导致异常。

示例：



```
public class SeniorClass
{
    public SeniorClass()
    {
        toString(); //如果被覆写了，可能会导致异常
    }

    @Override
    public String toString()
    {
        return "IAmSeniorClass";
    }
}

public class JuniorClass extends SeniorClass
{
    private String name;

    public JuniorClass()
    {
        super(); //调用父类的构造方法，导致 NullPointerException 异常
        name = "JuniorClass";
    }

    @Override
    public String toString()
    {
        return name.toUpperCase();
    }
}
```

### 规则27 覆写 equals 方法时，应同时覆写 hashCode 方法

说明：因为 Java 对象在存放于基于 Hash 的集合（如 HashMap、HashTable 等）时，会使用其 Hash 码进行索引，如果只覆写了 equals 方法，而没有正确覆写 hashCode 方法，则会导致效率低下甚至出错；Java 对象的 hashCode 方法有如下约定：

1. 同一次运行中，同一个对象如果 equals 方法中用到的信息没有改变，多次调用 hashCode 方法返回值必须相同；
2. 如果两个对象调用 equals 方法时相等，则这两个对象的 hashCode 方法，也必须返回相同的值；
3. 如果两个对象调用 equals 方法时不相等，则这两个对象的 hashCode 方法，不要求其返回值不同。

示例：

不好：覆写 equals 的时候，没有同时覆写 hashCode 方法

```
public class Entity
{
    private String id;
    private String value;

    @Override
    public boolean equals(Object obj)
    {
        if (obj instanceof Entity)
        {
            Entity that = (Entity)obj;

            return StringUtils.equals(this.id, that.id)
                && StringUtils.equals(this.value, that.value);
        }

        return false;
    }
}
```

#### 规则28 子类覆写父类方法时应加上@Override 注解

说明：加上@Override 注解的好处是，如果覆写时因为疏忽，导致子类方法的参数同父类不一致，编译时会报错，使问题在编译期就被发现。如果父类修改了方法定义造成子类不再覆写父类方法，也能使问题在编译期尽早被发现。

#### 建议12 接口定义中去掉多余的修饰词

说明：在接口定义中，属性已缺省具有 public static final 修饰词，方法已缺省具有 public abstract 修饰词。因此在代码中不要再次提供这些修饰词。

#### 建议13 设计时，考虑类的可变性最小化

说明：不可变类是指其实例一旦创建后就不能被修改，如 Java 平台类库的 String、BigInteger 和 BigDecimal。不可变类比可变类更加易于设计、实现和使用。

要使类成为不可变类，遵循以下 5 条规则：

- 1、不要提供任何会修改类状态的方法；
- 2、保证类不会被继承；
- 3、使所有值域都为 final；
- 4、使所有值域都成为私有；
- 5、如果类具有指向可变对象的域，则必须确保该类的使用者无法获得指向这些对象的引用。

## 6 异常和日志

### 6.1 异常

**原则13** 只针对真正异常的情况才使用 **exception** 机制

说明：不要用 **exception** 机制来做流程控制。**exception** 机制只应该用于处理罕见的、意料之外的、导致正常流程无法继续执行的行为，而不是取代正常的业务逻辑判断。

示例：

不好：下列代码企图利用异常来模拟正常的循环操作，是不可取的

```
try
{
    int i = 0;
    while(true)
    {
        range[i++].climb();
    }
}
catch(ArrayIndexOutOfBoundsException e)
{
    ...
}
```

**规则29** 在抛出异常的细节信息中，应包含能捕获失败的信息

说明：在抛出异常的时候，应该同时提供足够信息，以便对分析“异常是如何产生的”有帮助，比如“对该异常有贡献”的参数和变量的值。这样能方便程序员知道应该去查找哪些错误，可以极大地加速诊断过程。但是需要注意符合公司安全红线和法律法规的要求，不要包含敏感信息或者个人信息。

**规则30** 对可恢复的情况使用受检异常（**checked exception**），对编程错误使用运行时异常（**runtime exception**）

说明：Java 存在三种可抛出结构（**throwable**）：受检异常（**checked exception**）、运行时异常（**runtime exception**）和错误（**error**）。使用原则是：

- 1、如果期望调用者能够恢复，则应该使用受检异常。抛出受检异常，可以强迫调用者在一个 **catch** 子句中处理该异常，或者继续向外传播。
- 2、运行时异常是指难以恢复或者不可恢复的程序错误。大多数运行时异常都是表明前提违例（**precondition violation** 指 API 的调用方没有遵循调用约定）。
- 3、错误（**error**）被 JVM 保留用于指示资源不足、约束失败或者其他程序无法继续执行的情况。最好不要实现新的 **Error** 子类，所有抛出的未受检异常都应该是 **RuntimeException** 的子类。

**规则31 不要忽略异常**

说明：通过一个空的 `catch` 块可以很方便的忽略异常，如下所示：

```
try
{
    ...
} catch (SomeException e)
{
    // Empty catch block
}
```

空的 `catch` 块会使异常达不到应有的目的。忽略异常就如同忽略火警信号一样，若把火警信号器关掉了，当真正火灾发生时，就没有人能看到火警信号了。至少，在一个空 `catch` 块中也应该添加注释，解释为什么可以忽略这个异常，对于那些不应该频繁发生的异常，还应该将异常信息记录到日志中。例如关闭 `FileInputStream` 时，可以忽略异常，因为此时并没有改变文件状态，不必执行任何恢复动作，但要将异常信息记录到日志中，当异常频繁发生时就可以调查导致异常的原因。

**规则32 方法注释和文档中要包含所抛出异常的说明**

说明：要正确使用方法，必须对方法所抛出异常有所了解。因此，为保证方法的调用者清楚了解方法所抛出的异常，应该在方法的注释和文档中包含所抛出异常的说明。

**规则33 方法抛出的异常，应该与本身的抽象层次相对应**

说明：当方法把一个异常传给调用方时，请确保异常的抽象层次与方法的抽象层次是一致的。

示例：

不好： `getTaxId` 把更低层的 `IOException` 返回给调用方，暴露了实现细节，而且使调用方代码与底层耦合起来。

```
public class Employee
{
    ...
    public TaxId getTaxId() throws IOException
    {
        ...
    }
    ...
}
```

推荐：抛出 `EmployeeDataNotAvailable` 异常，抽象层次与方法一致

```
public class Employee
{
    ...
    public TaxId getTaxId() throws EmployeeDataNotAvailable
    {
        ...
    }
    ...
}
```

#### 建议14 对第三方 API 抛出大量各类异常进行封装

说明：为了避免与第三方 API 产生太过紧密的耦合，避免因第三方修改了异常之后而导致自身代码的变更，需对第三方抛出的异常进行封装。

#### 建议15 使用异常来做错误处理，而非错误码

说明：使用判断返回错误码来进行错误处理，易导致正常业务流和异常处理流交织在一起，可读性比较差。而且返回错误码，一般要求调用者立刻处理错误，易导致更深层次的嵌套。如下所示：

示例：

不好：使用错误码

```
if (deletePage(page) == E_OK)
{
    if (registry.deleteReference(page.name) == E_OK)
    {
        if (configKeys.deleteKey(page.name.makeKey()) == E_OK)
        {
            logger.log("page deleted");
        } else
        {
            logger.log("configKey not deleted");
        }
    } else
    {
        logger.log("deleteReference from registry failed");
    }
} else
{
    logger.log("delete failed");
    return false;
}
```

推荐：使用异常

```
try
{
    deletePage(page);
    registry.deleteReference(page.name);
    configKeys.deleteKey(page.name.makeKey());
    logger.log("page deleted");
}
catch (XXXException e)
{
    logger.log(e);
    ...
}
```

**规则34 在 finally 块中不要使用 return、break 或 continue 使 finally 块非正常结束**

说明：在 finally 中使用 return、break 或 continue 会使 finally 块非正常结束，造成的影响是，即使在 try 块或 catch 中抛出了异常，也会因为 finally 非正常结束而导致无法抛出。finally 块非正常结束会有编译告警。

如下所示：

下列代码的 main 方法中不会捕获到异常，输出是 test:0

```
public static void main(String[] args)
{
    try
    {
        System.out.println("test:" + test());
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
}

public static int test() throws Exception
{
    int a = 1;
    for (int i = 1; i < 2; i++)
    {
        try
        {
            throw new Exception("bb");
        }
        catch (Exception ex)
        {
            throw ex;
        } finally
        {
            continue;
        }
    }
    return 0;
}
```

### 规则35 不要直接捕获受检异常的基类 **Exception**

说明：捕获受检异常的目的是为了进行恢复，而如果不加区分的捕获所有受检异常，则无法进行对应异常的恢复处理。因此，应该区分并捕获具体的异常。

### 建议16 一个方法不应抛出太多类型的异常

说明：方法抛出过多的异常，会增加客户端异常处理的工作，同时也表明方法承担了过多的职责。

### 建议17 充分利用断言

说明：在 Java 1.4 中增加了 `assert` 关键字，用来测试指定的条件是否满足，当断言被违反时将抛出 `AssertionError`。断言是用来检查代码的 `bug`，即正常执行时永不该发生的情况。契约

式程序设计中，每个方法都有前条件和后条件，可采用断言来验证前置条件和后置条件是否满足。断言的执行是可配置的，在开发阶段，我们使用断言来充分发现程序 bug，而在正式发布后，出于性能考虑可不将断言包含在 release 版本中。

## 6.2 日志

### 原则14 日志信息准确、繁简得当，满足快速定位的需要

说明：日志的目的，就是当问题发生时，帮助程序员快速定位发生问题的原因，因此日志信息应满足以下要求，首先是要提供准确的信息，其次是信息不要太多也不要太少，太多则无法在海量日志中定位出问题，太少则没有足够信息定位，要繁简得当刚好能满足快速定位的需要。

### 规则36 日志的记录，不要使用 `System.out` 与 `System.err` 进行控制台打印，应该使用专用的日志工具(比如：`slf4j+logback`)进行处理

说明：专用日志工具比控制台打印提供了更丰富的日志记录功能，且使用更加简单

示例：

不好：使用控制台打印

```
start = System.currentTimeMillis();  
  
.....//其他加载数据的代码  
  
System.out.println("items loaded,cost " + (System.currentTimeMillis() - start) + "ms.");
```

推荐：采用日志工具（例如 `slf4j+logback`）

```
start = System.currentTimeMillis();  
  
.....//其他加载数据的代码  
  
logger.info("items loaded,cost {}ms.", (System.currentTimeMillis() - start));
```

### 规则37 日志工具对象 `logger` 应声明为 `private static final`

说明：1、声明为 `private` 是出于安全性考虑，防止 `logger` 对象被其他类非法使用

2、声明为 `static` 是为了防止重复 `new` 出 `logger` 对象，造成资源的浪费，同时防止 `logger` 被序列化，造成安全风险；（lib 库设计除外）

3、声明为 `final` 是因为在类的生命周期内无需变更 `logger`;

示例：

```
private static final Logger LOGGER = LoggerFactory.getLogger(this.class);
```

### 规则38 日志应分等级

说明：如果日志不分等级，则定位问题时，无法快速有效屏蔽大量低级别信息，给快速定位带来难度。日志可分为以下级别：`debug`、`info`、`warn`、`error`、`fatal`。推荐与具体实现有关的日志记录 `debug` 级，一般的业务处理日志用 `info` 级，不影响业务进行的错误用 `warn` 级，而记录异常的日志应为 `error` 或 `fatal` 级。

示例：



```
catch (XXXException e)
{
    LOGGER.error("#211#Error when insertNoCCNCMessage!" + e.toString(), e);
}
```

**规则39 日志中不要记录敏感信息**

说明：根据安全编程规范的要求，日志中不应出现敏感信息，如用户的密码、银行卡号等。

## 7 多线程并发

### 规则40 多线程访问同一个可变变量，需增加同步机制

说明：根据Java Language Specification中对Java内存模型的定义，JVM中存在一个主内存(Java Heap Memory)，Java中所有变量都储存在主存中，对于所有线程都是共享的。每个线程都有自己的工作内存(Working Memory)，工作内存中保存的是主存中某些变量的拷贝，线程对所有变量的操作都是在工作内存中进行，线程之间无法相互直接访问，变量传递均需要通过主存完成。根据上述内存模型的定义，要在多个线程间安全的同步共享数据就必须使用锁机制，将某线程中更新的数据从其工作内存中刷新至主内存，并确保其他线程从主内存获取此数据更新后的值再使用。

示例：

不好：下面的代码中，没有对可变数据`stopRequested`的访问做同步。程序期望在一秒钟后线程能停止。但在用java 1.6的server模式运行此程序（Java -server StopThread）时，程序陷入死循环，不能结束。

```
public class StopThread
{
    private static boolean stopRequested;

    public static void main(String[] args) throws InterruptedException
    {
        Thread thread = new Thread(new Runnable()
        {
            public void run()
            {
                int i = 0;
                while (!stopRequested)
                {
                    i++;
                }
            }
        });

        thread.start();

        TimeUnit.SECONDS.sleep(1);
        stopRequested = true;
    }
}
```

增加了`synchronized`同步机制后，程序就能正确地在1秒后终止。另一个方案是在变量前增加`volatile`关键字。

```
public class StopThread
{
    private static boolean stopRequested;

    private static synchronized void requestStop()
    {
        stopRequested = true;
    }

    private static synchronized boolean isStopRequested()
    {
        return stopRequested;
    }

    public static void main(String[] args) throws InterruptedException
    {
        Thread thread = new Thread(new Runnable()
        {
            public void run()
            {
                int i = 0;
                while (!isStopRequested())
                {
                    i++;
                }
            }
        });

        thread.start();

        TimeUnit.SECONDS.sleep(1);
        requestStop();
    }
}
```

#### 规则41 禁止不加控制地创建新线程

说明：Java 虚拟机能够管理的线程数量有限，不加控制的创建新线程可能会导致 Java 虚拟机崩溃。建议用 Java 1.5 之后提供的线程池 `ThreadPoolExecutor` 来管理线程资源。

示例：

不好

```
public void processEntityList1(List<Entity> entityList)
{
    for (Entity entity : entityList)
    {
        new Thread(new EntityProcessor(entity)).start();
    }
}
```

推荐，由线程池来管理线程资源

```
private ThreadPoolExecutor threadPool = ...; // init thread pool

public void processEntityList2(List<Entity> entityList)
{
    for (Entity entity : entityList)
    {
        threadPool.execute(new EntityProcessor(entity));
    }
}
```

#### 规则42 创建新线程时需指定线程名

说明：指定线程名可以给问题定位带来很多方便。日志或者 dump 中会包含线程的名称，而缺省的线程名 Thread-n 无法区分出是哪个线程，给问题定位带来不便。

#### 规则43 使用 Thread 对象的 setUncaughtExceptionHandler 方法注册 Runtime 异常的处理者 (v1.5+)

说明：Java 多线程程序中，所有线程都不允许抛出未捕获的 checked exception，也就是说各个线程需要自己把自己的 checked exception 处理掉。但是无法避免的是 unchecked exception，也就是 RuntimeException，当抛出异常时子线程会结束，但主线程不会知道，因为主线程通过 try catch 是无法捕获子线程异常的。Thread 对象提供了 setUncaughtExceptionHandler 方法来获取线程中产生的异常。而且建议使用 Thread.setDefaultUncaughtExceptionHandler，为所有线程设置默认异常捕获方法。

示例：

```
public class TestUncaughtException
{
    public static void main(String[] args)
    {
        TestThread thread = new TestThread();
        thread.setUncaughtExceptionHandler(new UncaughtExceptionHandler()
        {
            public void uncaughtException(Thread t, Throwable e)
            {
                System.out.println(t.getName() + " : " + e.getMessage());
            }
        });

        thread.start();
    }

    public static class TestThread extends Thread
    {
        public TestThread()
        {
        }

        public void run()
        {
            throw new RuntimeException("just a test");
        }
    }
}
```

**规则44** 不要使用 `Thread.stop` 方法，因为该方法本质是不安全的，使用它可能会导致数据遭到破坏

示例：

不好：不安全做法

```
private Thread blinker;

public void start()
{
    blinker = new Thread(this);
    blinker.start();
}

public void stop()
{
    blinker.stop(); // UNSAFE!
}

public void run()
{
    Thread currentThread = Thread.currentThread();
    while (true)
    {
        try
        {
            currentThread.sleep(interval);
        } catch (InterruptedException e)
        {
            ...
        }
        repaint();
    }
}
```

推荐做法

```
private volatile Thread blinker;

public void stop()
{
    blinker = null;
}

public void run()
{
    Thread currentThread = Thread.currentThread();
    while (blinker == thisThread)
    {
        try
        {
            currentThread.sleep(interval);
        } catch (InterruptedException e)
        {
            ...
        }
        repaint();
    }
}
```

#### 规则45 不要依赖线程调度器、线程优先级和 yield()方法

说明：Java 中的线程调度，是基于操作系统以及 JVM 的实现，在不同的操作系统中，或者不同厂商的 JVM（如 Sun、IBM、Microsoft 等），即使是同一套代码，其多线程的运行也是不一样的。因此，在多线程的程序中，不要依赖于系统的线程调度器来决定程序的逻辑运作，如果程序依赖于线程调度器来达到正确性或者性能要求，会导致不可移植。同理，程序如果依赖 JAVA 的线程优先级来确保正确性，也是不可移植的；而 Thread.yield()对线程调度器仅仅只是个提示，不保证确定的效果，因此代码也不能依赖 Thread.yield()方法。

#### 规则46 采用 Java1.5 提供新并发工具代替 wait 和 notify (v1.5+)

说明：自从 Java1.5 发行版本开始，Java 平台就提供了更高级的并发工具，它们可以完成以前必须在 wait 和 notify 上手写代码来完成的各项工作。java.util.concurrent 更高级的并发工具分成三类：Executor Framework、并发集合(Concurrent Collection)以及同步器(Synchronizer)。

#### 规则47 使用线程安全集合在多线程间共享可变数据

说明：在多个线程间共享可变的集合要使用线程安全的集合类型。在 Java1.5 及以上版本中，应首选 java.util.concurrent 包中提供的集合如 ConcurrentHashMap、ConcurrentSkipListSet 等。不仅线程安全而且性能更好。

#### 建议18 多线程操作同一个字符串相加，应采用 StringBuffer

说明：StringBuffer 相对于 StringBuilder 来说，是线程安全的。

**建议19 针对线程安全性，需要进行文档（javadoc）说明**

说明：当一个类的实例或者静态方法被并发使用的时候，这个类的行为如何，是该类与其客户端程序建立的约定的重要组成部分。如果你没有在一个类的文档中描述其行为的并发性情况，使用这个类的程序员将不得不做出某些假设。如果这些假设是错误的，这样得到的程序就可能缺少足够的同步，或者过度同步。无论属于这其中的哪种情况，都可能会发生严重的错误。



## 8 语言特性

### 8.1 运算和表达式

#### 规则48 不要写复杂的表达式

说明：复杂的表达式难以理解，会增加软件维护的成本和出错的几率。建议将难懂的语句封装到某个方法里，通过方法名来概括该语句的含义。

示例：

不好

```
if ((employee.flags & HOURLEY_FLAG) && (employee.age > 65))
{
    ...
}
```

推

荐

```
if (employee.isEligibleForFullBenefits())
{
    ...
}

private boolean isEligibleForFullBenefits()
{
    return (employee.flags & HOURLEY_FLAG) && (employee.age > RETIRE_AGE) ;
}
```

#### 规则49 运算时应避免产生溢出

#### 规则50

说明：

1、如下列两个 int 型变量 a,b 的运算 `int c=a+b;`或 `c=a-b;` 当结果为很大的正数 (>2147483647), 或为很小的负数时(<-2147483648), 可能会导致溢出, 则 c 不是正确的结果, 建议使用 long 型保存结果。如果是 long 型进行上述计算, 则建议使用 BigInteger。

2、计算一年的毫秒

错误示例：

```
int millisOfYear=1000*60*60*24*365; 或 long millisOfYear=1000*60*60*24*365;
```

正确示例：

```
long millisOfYear=1000L*60*60*24*365;
```

3、使用基于减法的比较器时，需要确认相减的结果不会超出类型的表示范围，如下，假设 T.num 为整型

错误示例：

```
new Comparator()
{
    @Override
    public int compare(T o1, T o2)
    {
        return o1.num - o2.num ;
    }
}
```

正确示例：

```
new Comparator()
{
    @Override
    public int compare(T o1, T o2)
    {
        if (o1.num > o2.num)
        {
            return 1;
        }
        return o1.num < o2.num ? -1 : 0;
    }
}
```

## 建议20 采用括号明确运算的优先级

说明：注意运算符的优先级。在普遍不易理解的复杂表达式中，使用括号说明优先级，以方便阅读和理解。即使代码优先级正确，也应该考虑到后续阅读或者修改这段代码的其他人员的方便，最好加上括号。

例子：

不建议：

```
if (a == b && c == d)
```

推荐：

```
if ((a == b) && (c == d))
```

## 8.2 控制语句

### 规则51 采用 for-each 代替传统的 for 循环（v1.5+）

说明：在 Java 1.5 之前，遍历一个集合的方法是：

```
for (Iterator i = c.iterator(); i.hasNext();)
{
    doSomething((Element)i.next());
}
```

遍历数组的做法是

```
for (int i = 0; i < a.length; i++)
{
    doSomething(a[i]);
}
```

由于需要手工获取数据和递增到下一个元素，在书写时更容易出错，尤其是多层循环嵌套的时候。在 Java1.5 中，引入了 for-each 循环，隐藏了获取数据和递增的操作，避免了混乱和出错的可能。使用 for-each 循环的例子：

```
for (String item : list)
{
    System.out.print(item);
}
```

## 规则52 在 switch 语句的每一个 case、和 default 中都放置一条 break 语句

说明：遗漏 break，可能导致程序误进入下一个 case 分支，执行了预期之外的代码，导致异常。而且，不推荐做出有意不写 break 的设计。

示例：

case ‘-’ 遗漏 break，导致计算结果与预期不一致

```
switch (c)
{
    case '-':
        result = minus(a, b);
        break;
    case '+':
        result = add(a, b); //遗漏 break，导致计算结果与预期不一致
    case '*':
        result = multiply(a, b);
        break;
    case '/':
        result = devide(a, b);
        break;
    default:
        ...
}
```

## 8.3 序列化

### 原则15 尽量不要实现 `Serializable` 接口

使用 Java 内置序列化功能的主要场景是为了在当前程序之外保存对象并在需要的时候重新获得对象。鉴于以下原因，建议除非必须使用的第三方接口要求必须实现 `Serializable` 接口否则请选用其它方式代替。

- 序列化不必要地对外公开了对象的物理实现
- 序列化容易使一个类对其最初的内部表示产生依赖
- 编写正确的反序列化代码有很大的挑战
- 序列化增大了安全风险
- 序列化增加了测试的难度

综上，序列化耦合了对象的逻辑信息和物理实现，使得开发者在面对领域需求之外需要额外关注很多专有的细节知识。在可能的情况下，使用其它替代方案将会减少工作量、减少 bug、降低出现安全漏洞的风险。

### 规则53 序列化对象中的 `HashMap`、`HashSet` 或 `HashTable` 等集合不能包含对象自身的引用

说明：如果一个被序列化的对象中，包含有 `HashMap`、`HashSet` 或 `HashTable` 集合，则这些集合中不允许保存当前被序列化对象的直接或间接引用。因为，这些集合类型在反序列化的时候，会调用到当前序列化对象的 `hashCode` 方法，而此时（序列化对象还未完全加载）计算出的 `hashCode` 有可能不正确，从而导致对象放置位置错误，破坏反序列化的实例。

示例：

```
class Super implements Serializable
{
    final Set<Super> set = new HashSet<Super>();
}

final class Sub extends Super
{
    private int id;

    public Sub(int id)
    {
        this.id = id;
        set.add(this); // 集合中引用了当前对象
    }

    public void checkInvariant()
    {
        if (!set.contains(this))
        {
            throw new AssertionError("invariant violated");
        }
    }

    public int hashCode()
    {
        return id;
    }

    public boolean equals(Object o)
    {
        return (o instanceof Sub) && (id == ((Sub) o).id);
    }
}
```

这个例子中，将当前对象（Sub 对象）放入了对象中的 HashSet 中，在反序列化 set 时，因为 id 属性还未完成初始化，导致 hashCode 的结果为 0，从而导致 Sub 对象在 set 中的位置放置错误，对象被破坏。

#### 建议21 实现 Serializable 接口的可序列化类应该显式声明 serialVersionUID

说明：如果可序列化类未显式声明 serialVersionUID，则序列化运行时将基于该类的各个方面计算该类的默认 serialVersionUID 值，如“Java(TM) 对象序列化规范”中所述。不过，强烈建议所有可序列化类都显式声明 serialVersionUID 值，原因计算默认的 serialVersionUID 对类的详细信息具有较高的敏感性，根据编译器实现的不同可能千差万别，这样在反序列化过程中可能会导致意外的 InvalidClassException。因此，为保证 serialVersionUID 值跨不同 java

编译器实现的一致性，序列化类必须声明一个明确的 `serialVersionUID` 值。还强烈建议使用 `private` 修饰器显式声明 `serialVersionUID`（如果可能），原因是这种声明仅应用于立即声明类 – `serialVersionUID` 字段作为继承成员没有用处。

示例：

```
public class BeanType implements Serializable
{
    private static final long serialVersionUID = -2589766491699675794L;
    ...
}
```

## 8.4 泛型

### 规则54 在集合中使用泛型（v1.5+）

说明：Java 1.5 版本中增加了泛型，在没有泛型之前，从集合中读取到的每一个对象都必须进行转换。如果有人不小心插入类型错误的对象，在运行时的转换处理就会出错。有了泛型之后，可以告诉编译器每个集合中接受哪些对象类型。编译器自动地为你进行转换，并在编译时告知是否插入了类型错误的对象。

示例：

不好：错误的将 `Coin` 对象插入到 `samps` 集合中，直到从集合中获取 `coin` 时才收到错误提示

```
private final Collection stamps = ...;

stamps.add(new Coin(...));
```

推荐：使用泛型，在编译时会提示类型错误

```
private final Collection<Stamp> stamps = ...;

stamps.add(new Coin(...));
```

还有个好处是，从集合中获取元素时，不再需要进行手工类型转换。如下所示：

```
for(Stamp s : stamps)
{
    ...
}
```

### 建议22 类的设计可优先考虑泛型（v1.5+）

说明：使用泛型类型，比使用需要在客户端代码中进行转换的类型来得更加安全，也更加容易。

示例：如下所示，一个设计为泛型的 `stack` 类，在使用时，无需对栈中元素进行类型转换。

```
public static void main(String[] args)
{
    Stack<String> stack = new Stack<String>();
    for (String arg : args)
    {
        stack.push(arg);
    }
    while (!stack.isEmpty())
    {
        System.out.println(stack.pop().toUpperCase());
    }
}
```

### 建议23 方法的设计可优先考虑泛型（v1.5+）

说明：就如类可以从泛型中受益一般，方法也一样。静态工具方法尤其适合于泛型化。

示例：如下所示，泛型方法就像泛型一样，使用起来比要求客户端转换输入参数并返回值的方法来得更加安全，也更加容易。

```
public static <E> Set<E> union(Set<E> s1, Set<E> s2)
{
    Set<E> result = new HashSet<E>(s1);
    result.addAll(s2);
    return result;
}

public static void main(String[] args)
{
    Set<String> guys = new HashSet<String>(Arrays.asList("Tom", "Dick", "Harry"));
    Set<String> stooges = new HashSet<String>(Arrays.asList("Larry", "Moe", "Curly"));
    Set<String> aflCio = union(guys, stooges);
    System.out.println(aflCio);
}
```

### 建议24 优先使用泛型集合，而不是数组（v1.5+）

说明：数组与泛型集合相比，有两个重要不同点。首先，数组是协变的(covariant)，即 Sub 是 Super 的子类型，则 Sub[]也是 Super[]的子类型。相反，泛型则是不可变的(invariant)，对于任意两个类型 Type1 和 Type2, List<Type1>既不是 List<Type2>的子类型，也不是其超类型。其次，数组是具体化的，因此数组在运行时才知道并检查它们的元素类型约束。

示例：如下代码是合法的，但执行时会报错

```
Object[] objectArray = new Long[1];
objectArray[0] = "I don't fit in"; //Throws ArrayStoreException
```

而如下代码则会在编译时报错

```
List<Object> objectList = new ArrayList<Long>(); //Incompatible types
objectList.add("I don't fit in");
```

## 8.5 其他语言特性

### 规则55 新代码不要使用已标注为@deprecated 的方法

说明：标注为@deprecated 的方法，是由于各种原因被 JDK 废弃的方法，为了保持兼容性而没有删除，新写的代码应避免使用这些方法，而应该使用 JDK 推荐的代替方法。

### 建议25 使用 JDK 自带的 API 或广泛使用的开源库，不要自己写类似的功能。

说明：JDK 和开源库（例如 Apache Commons,Google Guava 等）已实现的功能，不要重复实现，避免造成浪费。

### 建议26 升级到最新稳定版的 Java 平台版本上，以便获取新特性带来的收益

说明：较新的版本都包含许多让程序员更轻松的改进，你并不需要费力去学习怎样利用所有的新特性，有些新特性不需要你付出任何努力就可以给你带来实惠。

### 建议27 充分利用编译器的告警选项

在 C/C++中一条被普遍了解的规则是将编译器告警级别调整至最高并保持告警清零。在 Java 中该动作通常由静态检查工具如 findbugs 等完成，并且现在一些 IDE 也增加了很多有价值的告警信息（例如 eclipse 中对 null 指针的分析），将尽可能多的告警选项打开并保持告警正确清零，也有助于代码质量的提高。需要注意的是告警选项缺省情况下未必保存在工程文件中，为了整个团队使用一致的告警选项，需要将之配置到工程中。

### 建议28 使用字符串 API 时，应注意方法使用的是否是“正则表达式”

说明：正则表达式所引发的问题趋向于在运行时刻而不是在编译时刻暴露出来。如，String.replaceAll()方法第一个参数接受的是一个正则表达式，当匹配'.'符号时，需使用\\进行转义。

### 建议29 值类（value classes）的设计，可考虑实现 Comparable 接口，方便在集合中实现对象的搜索、排序、计算极值等

说明：一旦类实现了 Comparable 接口，它就可以跟很多泛型算法以及依赖于该接口的集合实现进行协作。你付出很小的努力就可以获得非常强大的功能。事实上，Java 平台类库中的所有值类（value classes）都实现了 Comparable 接口。如果你正在编写一个值类，它具有非常明显的内在排序关系，比如按字母顺序、按数值顺序或者按年代顺序，那你就应该坚决考虑实现这个接口。



```
public interface Comparable<T>
{
    int compareTo(T t);
}
```

## 9 性能与资源管理

### 9.1 性能

#### 原则16 谨慎地进行性能优化

说明：优化的弊大于利，特别是不成熟的优化。在优化的过程中，产生的软件可能既不快速，也不正确，而且还不容易修正。不要因为性能而牺牲合理的结构。要努力编写好的程序而不是快的程序。如果好的程序不够快，它的良好结构可以使它得到更加便利的优化。好的程序体现了信息隐藏（information hiding）的原则：只要有可能，它们就会把设计决策集中在单个模块中，因此，可以改变单个决策，而不会影响到系统的其他部分。

这并不意味着，在完成程序之前就可以忽略性能问题。实现上的问题可以通过后期的优化而得到修正。但是，遍布全局并且限制性能的结构缺陷几乎是不可能被改正的，除非重写系统。在系统完成之后再改变设计的某个基本方法，会导致系统的结构很不好，从而难以维护和改进。因此，必须在设计过程中考虑性能问题

#### 规则56 使用 System.arraycopy() 进行数组复制

说明：在将一个数组对象复制成另外一个数组对象时，不要自己使用循环复制，可以使用 JAVA 提供的 System.arraycopy() 功能来复制数据对象，这样做可以避免出错，而且效率会更高。

示例：

不好

```
int[] src = { 1, 2, 3, 4, 5 };
int[] dest = new int[5];
for (int i = 0; i < 5; i++)
{
    dest[i] = src[i];
}
```

推荐

```
int[] src = { 1, 2, 3, 4, 5 };
int[] dest = new int[5];
System.arraycopy(src, 0, dest, 0, 5);
```

#### 规则57 使用集合的 toArray() 方法将集合转为数组 (v1.42+)

说明：更好的性能，代码更加简洁

示例：

```
ArrayList list = new ArrayList();
list.add....
```

**建议30 在 Java 的 IO 操作中，尽量使用带缓冲的实现**

说明：在 Java 的 IO 操作，读写操作都有两种实现，一种是没有实现缓冲的，一种是实现了缓冲的，使用带有缓冲功能的 IO 操作，可以降低存储介质的访问次数，从而提高数据读写的效率，提供更好的操作性能。因此，建议尽量使用带有缓冲功能的实现来进行 IO 操作。对于性能要求更高的实现，可以使用 Java NIO

示例：

不好

```
PrintWriter out = null;

try
{
    out = new PrintWriter("file.txt");
    for (int i = 0; i < 100; i++)
    {
        out.println("write content " + i);
    }
} finally
{
    IOUtils.closeQuietly(out);
}
```

推荐

```
PrintWriter out = null;

try
{
    //采用带缓冲的实现
    out = new PrintWriter(new BufferedWriter(new FileWriter("file.txt")));
    for (int i = 0; i < 100; i++)
    {
        out.println("write content " + i);
    }
} finally
{
    IOUtils.closeQuietly(out);
}
```

## 9.2 资源管理

**规则58 避免创建不必要的对象**

说明：重用已经创建的对象比创建一个新的对象要好得多，除非确实需要重新创建。创建重复不必要的对象会导致资源浪费，严重时可能会导致性能问题。

示例：

不好：

```
String s = new String("string"); //建立了2个String对象
Integer i = new Integer(90);
...
Integer j = new Integer(90);
```

推荐：

```
String s = "string";
Integer i = Integer.valueOf(90);
...
Integer j = Integer.valueOf(90); //在-128~127 间，则会重用内存中缓存的对象
```

**规则59** 将对象存入 **HashSet**，或作为 **key** 存入 **HashMap**(或 **HashTable**)后，必须确保该对象的 **hashCode** 值不变，避免因 **hashCode** 值变化导致不能从集合内删除该对象，进而引起内存泄漏的问题

说明：对于 Hash 集合(HashMap, HashSet 等)而言，对象的 hashCode 至关重要，在 hash 集合内查找该对象完全依赖此值。如果一个对象存入 Hash 集合后 hashCode 随即发生变化，结果就是无法在集合内找到该对象，进而不能删除该对象，最终导致内存泄漏。

示例：

错误的示例

```
public class Email
{
    public String address;

    public Email(String address)
    {
        this.address = address;
    }

    public int hashCode()
    {
        int result = address.hashCode();
        return result;
    }

    public static void main(String[] args)
    {
        HashSet<Email> set = new HashSet<Email>();
        Email email = new Email("company.com");
        set.add(email);

        email.address = "silong.com"; //修改地址值，导致 hashCode 值变化 .....

        System.out.println(set.contains(email)); //false
        set.remove(email); //leak
    }
}
```

#### 规则60 执行 IO 操作时，应该在 finally 里关闭 IO 资源

说明：申请的资源不使用时，需要及时释放。而在产生异常时，资源释放常被忽视。因此要求在数据库操作、IO 操作等需要显示调用关闭方法如 close()释放资源时，必须在 try-catch-finally 的 finally 中调用关闭方法。如果有多个 IO 对象需要 close()，需要分别对每个对象的 close()方法进行 try-catch，防止一个 IO 对象关闭失败其他 IO 对象都未关闭。保证产生异常时释放已申请的资源，或使用 apache commons 的 IOUtils.closeQuietly()。JDK1.7 有自动资源管理的特性，不需手动关闭。

示例：

```
try
{
    in = new FileInputStream(inputFileName);
    out = new FileOutputStream(outputFileName);
    copy(in, out);
} finally
{
    IOUtils.closeQuietly(in);
    IOUtils.closeQuietly(out);
}
```

### 规则61 消除过期的对象引用

说明：过期引用是指永远也不会再被解除的引用。在支持垃圾回收的语言中，内存泄漏是很隐蔽的。如果一个对象引用被无意识地保留起来，那么，垃圾回收机制不仅不会处理这个对象，而且也不会处理被这个对象所引用的所有其他对象。

例如：如下是 **Stack** 类的 **pop** 方法。被弹出的对象，不会被垃圾回收机制回收，即使使用 **Stack** 的程序不再引用被弹出的对象，也不会回收。因为，**Stack** 内部仍维护着对这些对象的过期引用。

示例：

不好，会造成内存泄漏

```
public <T> pop()
{
    if (size == 0)
    {
        throw new EmptyStackException();
    }
    return elements[--size];
}
```

改为如下，则可消除过期引用

```
public <T> pop()
{
    if (size == 0)
    {
        throw new EmptyStackException();
    }
    T result = elements[--size];
    elements[size] = null;
    return result;
}
```

## 10 可移植性

### 规则62 不要在代码中硬编码"\n"和"\r"作为换行符号

说明：回车换行符在不同操作系统下是有区别的，如果需要换行，尽量用 `PrintStream`、`PrintWriter` 的 `println` 来代替在字符串中使用硬编码换行符。也可以使用 `System.getProperty("line.separator")` 获取运行时环境的换行符。

示例：

不好

```
System.out.print("Hello,world!\n");
```

推荐

```
System.out.println("Hello,world!");
```

### 建议31 谨慎地使用本地方法

说明：Java 中的本地方法，最早用来解决下面三种场景的使用：

- 1、提供平台特定的能力访问（如 Windows 平台下的注册表或文件锁）；
- 2、提供对遗留库或数据的访问（如遗留库使用 C/C++ 实现等情况）；
- 3、为了提高应用程序部分代码的性能；

但是，随着 Java 技术的发展，以上问题在新的 JDK 版本中都已经得到解决，从 JDK 1.3 开始，已经不再建议使用本地方法；它会导致应用程序与具体的 OS 平台耦合，会降低程序的通用性，并且因为实现方面的问题，通常会导致应用程序阻塞、死锁、崩溃等不稳定现象；而且，代码的可读性及可维护性也将大大降低，总而言之，在使用本地方法之前请务必三思。

### 建议32 避免对第三方代码的强依赖或陷入第三方代码细节

说明：程序中无可避免的会使用第三方提供的代码/jar 包，而这些第三方的代码/jar 包所提供的接口，是我们无法控制的，即使是公司/部门内部提供的接口，因此，当程序中（大量）使用了第三方提供的代码/jar 包时，可以使用 `Adapter` 模式，封装中间层，以避免第三方的变更对已有程序的影响。例如：公司从今年开始取消 `iLog`，使用自研的 `netGo` 替换，而我们在代码中大量使用了 `iLog` 对象进行交互，替换过程难免艰难，也容易引入问题，如果当初在代码中能对 `iLog` 对象进行封装，在替换时就不会影响逻辑代码。

# 11 国际化

**规则63** 在所有的输入输出环节，指明正确的编码方式，进行正确的字符到字节，或字节到字符的转换

说明：使用 `java.nio.charset` 中的类编码解码字符集，见如下示例

```
// Corrupts data on errors
public static byte[] toCodePage_bad(String charset,
    String string) {
    return string.getBytes(charset);
}

// Fails to detect corrupt data
public static String fromCodePage_bad(String charset,
    byte[] bytes) {
    return new String(bytes, charset);
}

// Correct
public static byte[] toCodePage_good(String charset, String string) {
    Charset cs = Charset.forName(charset);
    CharsetEncoder coder = cs.newEncoder();
    ByteBuffer bytebuf = coder.encode(CharBuffer.wrap(string));
    byte[] bytes = new byte[bytebuf.limit()];
    bytebuf.get(bytes);
    return bytes;
}

// Correct
public static String fromCodePage_good(String charset, byte[] bytes) {
    Charset cs = Charset.forName(charset);
    CharsetDecoder coder = cs.newDecoder();
    CharBuffer charbuf = coder.decode(ByteBuffer.wrap(bytes));
    return charbuf.toString();
}
```

**规则64** 如果输入源或输出目标直接支持，尽可能直接使用 Unicode 进行输入输出。

说明：例如，Oracle 数据库直接支持 UTF-8 的文本数据。使用 UTF-8 操作 Oracle，可自动兼容所有的语言文字；反之，使用 ISO-8859-1 或者 ASCII 去操作 Oracle，只能兼容欧美单字节的文字。

**规则65** 不要依赖平台默认的字符编码方式。

说明：例如，中文 Windows 下，默认的编码为 GBK，英文 linux 下，默认编码为 ISO-8859-1。依赖平台默认值意味着同样的程序在不同的平台上可能产生不同的结果。



### 规则66 对于使用默认编码方式的第三方代码或者遗留代码，可应用适配器模式，将返回的字符串转换成 Unicode 内码

说明：例如，我们的数据库错误的使用了 ASCII 编码存储文本，也就是说从数据库返回的中文字，实际上被“拆”成了两个欧洲字符。但是数据库中已经保存了大量数据，全改成 Unicode 表示不容易。我们可以在数据访问层做一个适配器，将欧洲字符重新组合，变成真正的 Unicode 中文。

### 建议33 字符串大小写转换时，应加上 Locale.US

说明：String 类的 toUpperCase() 和 toLowerCase() 方法，如果不输入参数，则会按当前系统默认的编码模式转换，因此转换结果可能并非如你所预期，如下所示：

示例：

不好：如果当前环境是土耳其 Turkish，那最后输出的结果不是预期的大写 I 了，而是另外一个字符（i）

```
String testString = "i";  
System.out.println(testString.toUpperCase());
```

推荐：字符串的大小写转换一般都是在 26 个英文字母，建议显示指定语言为 Local.US

```
String testString = "i";  
System.out.println(testString.toUpperCase(Locale.US));
```