密级：机密

Neusoft

JAVA 编码规范

NSFT_IP03　V1.0

中国 沈阳浑南高新技术产业开发区 东软软件园

www.neusoft.com

更改履历

版本号	更改时间	更改的 图表和章节号	状 态	更改简要描述	更改申请 编号	更改人	批准人

注：状态可以为 N-新建、A-增加、M-更改、D-删除。

JAVA 编码规范

目　录

1. 引言

1.1 简介

所有的程序开发手册都包含了各种规则。一些习惯自由程序的人（例如 Java 程序员）可能对这些规则很不适应，但是在多个开发人员共同协作的情况下，这些规则是必需的。这不仅仅是为了开发效率，而且也为了测试和后期维护。

良好的编码习惯有助于标准化程序的结构和编码风格，使源代码对于自己和别人都易读和易懂。在开发周期中越早使用恰当的编码规定，将会最大程度的提高项目的生产率。良好的编码习惯除了代码格式，详细的注释外，还应该包括使用有助于提高程序效率的编码方式。

规范的开发有助于提高源码的可读性，可维护性，对于提高项目的整体效率更是不可缺少的（尤其是团队开发）。

1.2 目的

本文是一套面向 Java programmer 和 Java developer 进行开发所应遵循的开发规范。按照此规范来开发 Java 程序可带来以下益处：

- 代码的编写保持一致性，
- 提高代码的可读性和可维护性，
- 在团队开发一个项目的情况下，程序员之间可代码共享
- 易于代码的回顾，

2. 源程序

2.1 源程序命名

Java 源程序的名字应该是这种形式：ClassOrInterfaceName.java。ClassOrInterfaceName 应该是在 Java 源程序中定义的 class 或者 interface 的名字（关于 classes 和 interface 的命名规范请参考 3.2）。源程序的文件名后缀通常为 .java。

2.2 供发布的文件

如果文件编译后，需要用打包的形式发布，那么这个包的名字应该是有代表性的（例如应该是这个模块或者这个文件所在单元的名字）。通常包的扩展名有 *.jar(推荐使用)或者 *.zip、*.ear、*.war

等。

2.3 源文件的组织

1) 一个 Java 源文件应该包含如下的元素，并按照以下顺序书写：
2) 版本信息和版权声明
3) 包的声明
4) 引用声明
5) 类或者接口的声明

以上元素之间以至少一个空行来分隔。

2.3.1 版本信息和版权声明

每一个源程序应该以一个包含版本信息和版权声明的块为开始。

例如：

```
/**
 *      <p>application name：       sample1</p>
 *      <p>application describing：  this class handels the request of the client</p>
 *      <p>copyright：              Copyright ® 2002  东软  软件开发事业部版权所有</p>
 *      <p>company：                neusoft</p>
 *      <p>time：                   2002.02.25</p>
 *
 *      @author                 Brunce
 *      @version                 ver 3.1
 */
```

2.3.2 包的声明

每一个源程序若包含一个包的声明，则必须是非注释的第一行，并用一个空格分隔 package 关键字和 package 的名字。

例如：

```
package telmecall.presentation.util;
```

2.3.3 引用声明

import 语句应该从第一列开始，并用一个空格分隔 import 关键字和 import type name。引用时应保证所有引入类都被使用，即减少引入*的使用，因为如果引入不确切的类将很难理解当前类的上下文关

系及相关性。

例如：

```
import    telmecall.presentation.view.bookview//GOOD
import    telmecall.presentartion.*//NOT GOOD
```

2.3.4 类或者接口的声明

每个源程序至少会有一个 class 或者 interface 的声明。

3. 命名规范

3.1 包的命名

包的名字应该都是由小写单词组成。它们全都是小写字母，即便中间的单词亦是如此。

例如：

```
package telmecall.business.console;
package telmecall.business.exception;
package telmecall.presentation.util;
```

其中 telmecall 指项目名称

business/presentation 指业务逻辑和表现层

console/exception/util 指具体模块

3.2 类和接口的命名

类和接口的名字一般由大写字母开头而其他字母都小写的单词组成（但一些专有的缩写词，比如：AWTException 等则除外）。

3.2.1 类的命名

Class 的名字建议使用名词或者名词短语。

例如：

```
//好的类命名：
    LayoutManager，ArrayIndexOutOfBoundsException
  //不好的类命名：
  ManageLayout   //动词短语，建议用名词短语
 awtException   //awt 是专有名词，都应该大写
 array_index_out_of_bounds_exception   //不能有下划线
```

3.2.2 接口的命名

Interface 的名字取决于 Interface 的主要功能和用途。如果 Interface 是使 Object 具有某种特定的功能，则 Interface 的名字建议使用可以描述这种功能的形容词（加上-able 或者-ible 后缀）

例如：

```
Searchable，Sortable，NetworkAccessible 等
```

如果 Interface 不是使 Object 具有某种特定的功能则建议使用名词或者名词短语。

3.3 变量命名

变量的命名采用大小写混合的形式。以小写字母开头，名字中其他单词或者只取首字母的缩写单词以大写字母开头，所有其它的单词都为小写字母，不要使用下划线分隔单词。名字应为名词或者名词短语。

例如：

```
boolean     resizable;
char        recordDelimiter;
```

3.4 常量命名

常量的命名建议都为大写字母，并用下划线分隔单词。

例如：

```
MIN_VALUE，MAX_BUFFER_SIZE，OPTION_FIEL_NAME
```

3.5 方法命名

方法命名采用大小写混合的形式。以小写字母开头，名字中其他单词或者只取首字母的缩写单词以大写字母开头，所有其它的单词都为小写字母，不要使用下划线分隔单词。方法的命名应该能描绘出方法的作用和功能，方法的名字建议使用祈使动词或者动词短语。

例如：

```
    //好的方法命名：
showStatus（），drawCircle（），addLayoutComponent（）
    //不好的方法命名：
mouseButton（）   //名词短语，不能描绘出方法的功能
DrawCircle（）    //首字母不应该大写
```

```
add_layout_component（） //不应该使用下划线
serverRunning（）//动词短语，但不是祈使动词
```

获取或者设置类的某种属性的方法建议显式的命名为 getProperty（）或者

setProperty（），其中 property 是指类的属性的名字。

例如：

```
getHeight（），setHeight（）
```

用于判断类的布尔属性的方法建议显式的命名为 isProperty（），property 是指类的属性的名字。

例如：

```
isResizable（），isVisible（）
```

3.6 标签命名

标签的使用主要是针对 break 和 continue 等语句的。标签的命名应该为小写字母，并使用下划线来分隔单词。虽然语言允许，但也不要在同一个方法中多次使用同一个标签名。

例如：

```
for (int i = 0; i < n; i++)
{
        label_one:
        {
                if (node[i].name == name)
                {
                        break label_one;
                }
        }//label_one
            lable_two:
            {
                if (node[i].name == name)
                {
                        break label_two;
                }
            } //label_two
    }
```

4. 空白的使用

4.1 空行

- 空行的使用有益于将代码按照逻辑分段，以提高代码的可读性。在下列情况下建议使用一个空行：

- 在版权声明块、包声明块、引用声明块之后。
- 在类的声明之间。
- 在方法的声明之间。
- 在类中声明最后一个属性之后，声明第一个方法之前。

例如：

```
package telmecall.presentation.view;

import java.util.Date;

public class BookView
{
    private String bookID;
    private String bookName;
    private String bookAuthor;

    public BookView()
    {
        bookID = null;
        bookName = null;
        bookAuthor = null;
    }

    public void Book()
    {
    }
}
```

4.2 空格

下列情况建议使用单个空格（not tab）：

- 在一个关键字和它所修饰的块之间。可用于如下的关键字：catch，for，if，switch，synchronized，while 。下列关键字的后面请不要跟随空格：super ，this 。
- 在任何关键字和它所用的参数之间。例如：

```
    return true ；
```

- 在一个列表中逗号之后。例如

```
foo(long_expression1, long_expression2, long_expression3);//RIGHT
foo(long_expression1,long_expression2,long_expression3);//NOT GOOD
```

- 下列情况不建议使用空格
- 左括号和后一个字符之间不应该出现空格，例如：

```
if (i== 42)；//RIGHT
if( i==42);   //NOT GOOD
```

- 右括号和前一个字符之间也不应该出现空格，例如：

```
if (i== 42)；//RIGHT
if(i==42 );   //NOT GOOD
```

4.3 缩进

- 行的缩进要求是四个空格。由于在使用不同的源代码管理工具时 Tab 字符将因为用户设置的不同而扩展为不同的宽度，所以请不要使用 tab 键。建议修改各管理工具的设置将 Tab 字符扩展成 4 个空格。

4.4 行的延续

- 代码中的行应该为 80 列，源代码一般不会超过这个宽度，如超过了 80 列应该截成较短的行，建议超长的语句应该在一个逗号或者一个操作符后折行。一条语句换行后，应该比原来的语句有所缩进，缩进的具体格数要依据该语句的类型。

- 例如：

```
// 好的折行方式
foo(long_expression1, long_expression2,
        long_expression3);//缩进后与上一行 long_expression1 对齐
// 好的折行方式
foo(long_expression1,
        long_expression2,// 缩进后与上一行 long_expression1 对齐
        long_expression3);
// 好的折行方式
if (long_logical_test_1 || long_logical_test2
|| long_logical_test_3)//缩进后与上一行 long_logical_test_1 对齐
{
        statements;
  }
while (long_logical_test_1
          || long_logical_test2
           || long_expression3)
{
}
```

5. 注释

- 一般来说，注释的使用应按照以下原则：

- 注释应该能够帮助读者理解代码的功能和作用，应该有助于读者理解程序的流程。

- 注释要言简意赅。

- 错误的注释还不如不做注释。

- 临时的注释用“temporary note”进行标注，以利于将来程序发布时将其删去。

- 例如：

```
// temporary note: Change this to call sort() when the bugs in it are fixed list->mySort();
```

- 对于广大的开发人员只要在源码的编写过程中注意加入适当的注释，则与源码同步的文档也就随之产生。以下将详细介绍注释的编写规范：

5.1 版权信息注释

- 版权信息必须在 java 文件的开头，比如：

```
/**
 * <p>Copyright ® 2002 东软 软件开发事业部版权所有。</p>
 */
```

5.2 类注释

- 类注释通常放在类定义的前面（类实体以外）。通常详细介绍该类的功能，作者，版本，“@deprecated 标志”的使用，以及产生该类的时间和引用其他类等信息。如下：

```
/**
 *  类功能介绍。
 *
 *  <p><code>Method</code> 介绍信息可以使用 HTML 标记。
 *
 *  @see        java.lang.Class
 *  @see        java.lang.Class#getDeclaredMethod(String, Class[])
 *  @author     Frank
 *  @version    1.15, 2001-12-18
 *  @author     Mary
 *  @deprecated
 */
```

解释：

@see：引用其他类

所有三种类型的注释文档都可包含@see 标记，它允许我们引用其他类里的文

档。对于这个标记，javadoc 会生成相应的HTML，将其直接链接到其他文档。格式如下：

@see 类名
@see 完整类名
@see 完整类名#方法名
每一格式都会在生成的文档里自动加入一个超链接的"See Also"（参见）条目。注意javadoc 不会检查我们指定的超链接，不会验证它们是否有效。

@version

格式如下：
@version 版本信息
其中，"版本信息"代表任何适合作为版本说明的资料。若在javadoc 命令行使用了"-version"标记，就会从生成的HTML 文档里提取出版本信息。

@author

格式如下：
@author 作者信息
其中，"作者信息"包括作者的姓名、电子函件地址或者其他任何适宜的资料。若在javadoc 命令行使用了"-author"标记，就会专门从生成的HTML 文档里提取出作者信息。可为一系列作者使用多个这样的标记，但它们必须连续放置。全部作者信息会一起存入最终HTML 代码的单独一个段落里。

@deprecated

这是Java 1.1 的新特性。该标记用于指出一些旧功能已由改进过的新功能取代。该标记的作用是建议用户不必再使用一种特定的功能，因为未来改版时可能摒弃这一功能。若将一个方法标记为@deprecated，则使用该方法时会收到编译器的警告。

5.3 成员注释

只有 public 和 protected 类型的类成员的注释可以被 javadoc 提取到文档中，而 private 型和“友好”（没有限定符）的类成员将被忽略。但是为了提高项目的可读性和可维护性，建议将所有的类成员都注释。如下：

```
    /**
 * <code>double</code> 型数值的最大值。
 * 它等于
 * <blockquote><pre>
 * <code>Double.longBitsToDouble(0x7fefffffffffffffL)</code>
 * </pre></blockquote>
      * 的返回值。
 */
public static final double MAX_VALUE = 1.79769313486231570e+308;
```

5.4 方法注释

● 方法注释的注意内容有方法功能介绍，参数说明，返回类型说明，例外类型，JDK/SDK 的引入版本等。如下：

```
/**
 * 方法的功能描述。
 *
 * @param       s     the string to be parsed.
 * @return      the double value represented by the string argument.
 * @exception   NumberFormatException    if the string does not contain a
 *                  parsable double.
 * @see         java.lang.Double#valueOf(String)
 * @since       1.2
 */
```

解释：

@param

格式如下：

@param 参数名 说明

其中，“参数名”是指参数列表内的标识符，而“说明”代表一些可延续到后续行内的说明文字。一旦遇到一个新文档标记，就认为前一个说明结束。可使用任意数量的说明，每个参数一个。

@return

格式如下：

@return 说明

其中，“说明”是指返回值的含义。它可延续到后面的行内。

@exception

当方法中发生异常时，“扔出”对象。调用一个方法时，尽管只有一个违例对象出现，但一些特殊的方法也许能产生任意数量的、不同类型的违例。所有这些违例都需要说明。所以，违例标记的格式如下：

@exception 完整类名 说明

其中，“完整类名”明确指定了一个违例类的名字，它是在其他某个地方定义好的。而“说明”（同样可以延续到下面的行）告诉我们为什么这种特殊类型的违例会在方法调用中出现。

@since

格式如下：

@since 说明

利用说明内容为文档增加"since"标题，表示从何时开始使用该方法或者类(以正式发布的版本为准)

@see

参考 5.2.1

5.5 内部类的注释

内部类的注释请参考 5.2 类注释。

5.6 其它的注释：

5.6.1 代码修改的注释

若要修改单行代码，请在上一行使用单行注释，写明修改原因，人员姓名和日期。并且用单行注释符号注释原行代码。在下一行添加更新的代码。如下：

```
//初始值设置错误。WangGang  2001-12-18
        //String showMessage = null;
        String showMessage = "欢迎访问！";
```

说明：这种注释主要是针对以下几种情况而言的：

- 严重 bug 的修改
- 版本相对稳定后代码的修改
- 修改别人的代码

5.6.2 冗余代码的注释

若在测试或者维护的过程中发现冗余的程序代码，请注释冗余的代码行。并且在该代码的前一行注明注释原因，注释人员姓名和注释日期。注释符号的选择参照 5.6.1 代码修改的注释。

5.6.3 类体外的注释

类体外主要包括 package 和 import 语句。若在测试或者维护的过程中需要为类添加新的功能，往往需要引入新的类包。在引入新的类包的同时不需要指定引入人及日期等 Log 信息。多余的类包可以直接删除。

6. 类

类的组织顺序建议如下顺序：

- 类的定义
- 类的成员变量(字段/属性)
- 类成员变量(字段/属性)的存取方法

- 构造函数
- 类的方法
- main()方法

6.1 类的定义

包含了在不同的行的 extends 和 implements

```
public class CounterSet
        extends Observable
        implements Cloneable
```

6.2 类的成员变量(字段/属性)

接下来是类的成员变量：

```
/**
* Packet counters
*/
protected int[] packets;
```

6.3 类成员变量(字段/属性)的存取方法

接下来是类变量的存取的方法。

```
/**
  * Get the counters
  * @return an array containing the statistical data.    This array has been
  * freshly allocated and can be modified by the caller.
  */
public int[] getPackets()
    {
  return copyArray(packets, offset);
}
public int[] getPackets()
{
  return packets;
    }
```

```
public void setPackets(int[] packets)
{
this.packets = packets;
}
```

6.4 构造函数

接下来是构造函数，它应该用递增的方式写（比如：参数多的写在后面）。

```
public   CounterSet(int size)
{
   this.size = size;
}
```

6.5 类方法(静态方法)

下面开始写类方法：

```
/**
 * Set the packet counters
 * (such as when restoring from a database)
 */
protected static void setArray(int[] r1, int[] r2, int[] r3, int[] r4)
         throws IllegalArgumentException
{
      // Ensure the arrays are of equal size
       if (r1.length != r2.length || r1.length != r3.length || r1.length != r4.length)
       throw new IllegalArgumentException("Arrays must be of the same size");
       System.arraycopy(r1, 0, r3, 0, r1.length);
       System.arraycopy(r2, 0, r4, 0, r1.length);
}
```

6.6 实例方法

实例方法一般如下定义：

```
/**
 * Set the packet counters
 * (such as when restoring from a database)
 */
protected void setArray(int[] r1, int[] r2, int[] r3, int[] r4)
        throws IllegalArgumentException
{
      // Ensure the arrays are of equal size
       if (r1.length != r2.length || r1.length != r3.length || r1.length != r4.length)
       throw new IllegalArgumentException("Arrays must be of the same size");
       System.arraycopy(r1, 0, r3, 0, r1.length);
       System.arraycopy(r2, 0, r4, 0, r1.length);
}
```

有必要时，须加上 final 修饰符，表示本方法不可被覆盖。

6.7 main 方法

如果 main(String[] args) 方法已经定义了，那么它应该写在类的底部

```
       pulic static void main(String[] args)
    {
          ……
    }
```

7. 接口

接口的定义形式类似于类的定义形式。接口的声明采用如下的格式：

```
[public] interface InterfaceName [extends SuperInterfaces]
{
     InterfaceBody
}
```

SuperInterfaces 是可用逗号分隔的接口列表（如果超过一个接口），按照字母的顺序排列。

接口的声明应从第一列开始，“{” 应该单独占用一行，然后是接口体的定义，开头要有缩进，结束时 “}” 应该单独占一行，且为

第一列。

7.1 接口体的组织

接口体的声明按照建议按如下顺序：

接口的字段声明，接口方法声明。接口字段声明和方法声明同类的字段和方法的声明的形式一样。

8. 语句

8.1 简单语句

8.1.1 赋值和表达式

每行只包含一条语句。

例如：

```
a = b + c; // RIGHT
count++; // RIGHT
a = b + c; count++; // NOT GOOD
```

8.1.2 本地变量声明

本地变量的声明应在不同的行中，如果本地变量声明后不再变动应当声明为 final，使编译器生成更有效率的代码。

例如：

```
int j    = 4; // RIGHT
int i, k; // ACCEPTABLE
int i, j = 4, k; // NOT GOOD
```

8.1.3 数组的声明

数组的[]应该紧跟类型名，而不是数组名。在“[”之前不能有空格。

例如：

```
char buf[]; // NOT GOOD
char[] buf; // GOOD
String[] getNames() { // RIGHT, method return value
```

8.1.4 return语句

return 语句后面的返回值不可以用括号包起来，除非是后面所跟为一个表达式。

例如：

```
return(true); // WRONG
return true; // RIGHT
return (s.length() + s.offset); // RIGHT
```

8.2 复合语句

8.2.1 括号的格式

复合语句是一个语句块被封装在{}中。所有的复合语句建议采用如下的花括号形式：

{} 中的语句应该单独作为一行。例如：

```
if (i>0)
{
    i ++
}       // RIGHT, “{” 单独作为一行
if (i>0) { i ++ }; //NOT GOOD, “{” 和 “}” 在同一行
```

- 在 {} 中的复合语句应该有层次。
- 如下的情况可以不使用花括号：
- 语句块中包含 null 语句，或者只有简单的一行。

注：有些 IDE 工具的默认设置“{”是在行尾，请修改 IDE 工具设置，使“{”换行显示。

8.2.2 If语句

```
if(condition)
{
    statements;
}
if (condition)
{
    statements;
}else
{
    statements;
}
```

8.2.3 for语句

```
for (initialization; condition; update)
{
    statements;
}
```

8.2.4 while语句

```
while (condition)
{
        statements;
}
```

如使用无限循环，那么建议使用：

```
while (true)
{
        statements;
}
```

8.2.5 do-while语句

```
do
{
    statements;
}while (condition);
```

8.2.6 switch语句

```
switch (condition)
{
        case 1:
        case 2:
            statements;
            break;
        default:
            statements;
            break;
   }
```

8.2.7 try语句

```
try
{
        statements;
} catch (exception-declaration)
{
        statements;
}
try
{
        statements;
} finally
{
        statements;
}
try
{
```

```
        statements;
} catch ( exception-declaration)
{
        statements;
} finally
{
        statements;
}
```

8.2.8 synchronized语句

```
synchronized (expression)
{
        statements;
}
```

8.3 标签语句

标签语句通常用花括号封装，而标签本身则应在单独一行，后面跟一个冒号。

例如：

```
            statement-label:
        {
        }// statement-label
            <Body of the document>
```

9. 性能优化常识

9.1 前提

首先让程序运行起来，再考虑变得更快——但只有在自己必须这样做、而且经证实在某部分代码中的确存在一个性能瓶颈的时候，才应进行优化。除非用专门的工具分析瓶颈，否则很有可能是在浪费自己的时间。性能提升的隐含代价是自己的代码变得难于理解，而且难于维护。

对于像字符串的连接操作不使用“＋”而使用专有方法 concat 等其他方法，这类问题，则不能称为性能优化，而只能叫做基本常识。因而这类问题的解决并不能影响程序的可读性和易维护性，所以我们提倡为性能优化打基础。以下将介绍一些常识：

9.2 运算时间

运算	*示例*	*标准时间*
本地赋值	*i=n;*	*1.0*
实例赋值	*this.i=n;*	*1.2*
Int 增值	*i++;*	*1.5*
Byte 增值	*b++;*	*2.0*
Short 增值	*s++;*	*2.0*
Float 增值	*f++;*	*2.0*
Double 增值	*d++;*	*2.0*
空循环	*while(true) n++;*	*2.0*
三元表达式	*(x<0)?-x:x;*	*2.2*
算术调用	*Math.abs(x);*	*2.5*
数组赋值	*a[0]=n;*	*2.7*
Long 增值	*l++;*	*3.5*
方法调用	*funct();*	*5.9*
Throw 或者 catch 违例	*Try{throw e;}*	*320*
同步方法调用	*synchMethod();*	*570*
新建对象	*new Object();*	*980*
新建数组	*new int[10];*	*3100*

标准时间 ＝ 语句执行时间/本地赋值时间

9.3 java.lang.String

字串的开销：字串连接运算符“＋”看似简单，但实际需要消耗大量系统资源。编译器可高效地连接字串，但变量字串却要求可观的处理器时间。该操作要创建并拆除一个 StringBuffer 对象以及一个 String 对象。

上述问题的通常解决方法是新建一个 StringBuffer（字串缓冲），用 append 方法追加自变量，然后用 toString()将结果转换回一个字串。当要追加多个字串，则可考虑直接使用一个字串缓冲——特别是能在一个循环里重复利用它的时候。通过在每次循环里禁止新建一个字串缓冲，可节省 980 单位的对象创建时间（见上表）。

更有效的解决办法是：在构造 StringBuffer 时，应该粗略的估计出它最终的总长度。默认构造函数预设了 16 个字符的缓存容量。append()方法首先计算字符串追加完成后的总长度，如果这个总长度大于 StringBuffer 的存储能力，append()方法调用私有的 expandCapacity()方法。expandCapacity()方法在每次被调用时使 StringBuffer 存储能力加倍，并把现有的字符数组内容复制到新的存储空间。存储能力的扩展，从而导致了两次代价昂贵的复制操作。因此，我们至少有一电可以做得比编译器更好，这就是分配一个初始存储容量大于或者等于最终字符长度 StringBuffer。

因此，使用默认构造函数创建的 StringBuffer 在字符串连接操作上的效率其实和用“＋”是一样的。如果首先估计出整个字符串最终的总长度，才会显著提高效率！

其他的字符串运算操作尽可能使用 String 已经提供的方法。比如，短字符串的连接可以使用 concat；子串的查找可以使用 indexOf，substring 等。

9.4 java.util.Vector

一个 Vector 就是一个 java.lang.Object 实例的数组。Vector 与数组相似，它的元素可以通过整数形式的索引访问。但是，Vector 类型的对象在创建之后，对象的大小能够根据元素的增加或者删除而扩展、缩小。

1) 避免把新元素添加到 Vector 的最前面

除非有绝对充足的理由要求每次都把新元素插入到 Vector 的前面，否则对性能不利。在默认构造函数中，Vector 的初始存储能力是 10 个元素，如果新元素加入时存储能力不足，则以后存储能力每次加倍。Vector 类就象 StringBuffer 类一样，每次扩展存储能力时，所有现有的元素都要复制到新的存储空间之中。

2) 避免从中间删除元素

由于 Vector 中各个元素之间不能含有“空隙”，删除除最后一个元素之外的任意其他元素都导致被删除元素之后的元素向前移动。也就是说，从 Vector 删除最后一个元素要比删除第一个元素“开销”低好几倍。

3) 删除所有元素的最好方法是 removeAllElements()

4) 避免二次搜索

Vector 类型的对象 v 包含字符串“Hello”。考虑下面的代码，它要从这个 Vector 中删除“Hello”字符串：

```
String s = "Hello";
int i = v.indexOf(s);
if(i != -1)
v.remove(s);
```

在这段代码中，indexOf()方法对 v 进行顺序搜索寻找字符串“Hello”，remove(s)方法也要进行同样的顺序搜索。改进之后的版本是：

```
String s = "Hello";
int i = v.indexOf(s);
if(i!= -1) v.remove(i);
```

这个版本中我们直接在 remove()方法中给出待删除元素的精确索引位置，从而避免了第二次搜索。一个更好的版本是：

```
String s = "Hello";
v.remove(s);
```

循环内部的代码不会以任何方式修改 Vector 类型对象 大小时，应该提前取得 Vector.size()

9.5 线程

9.5.1 防止过多的同步

不必要的同步常常会造成程序性能的下降。因此，如果程序是单线程，则一定不要使用同步。

9.5.2 避免同步整个代码段

对某个方法或函数进行同步比对整个代码段进行同步的性能要好。因为代码段的同步牵涉的范围比对某个方法或函数进行同步广。

9.5.3 对每个对象使用多“锁”的机制来增大并发

一般每个对象都只有一个“锁”，这就表明如果两个线程执行一个对象的两个不同的同步方法时，会发生“死锁”。即使这两个方法并不共享任何资源。为了避免这个问题，可以对一个对象实行“多锁”的机制。

9.6 循环

9.6.1 边界

循环的边界是指完成所有循环操作的起点和终点。如果循环体内的操作不影响边界，那么应该在循环体外，计算并且求得边界值。例如：

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

上述代码中每次循环操作，都要计算一次 array.length。

9.6.2 循环体内避免构建新对象

如果在循环体内用到新对象，需要在循环体开始以前构建好该对象。由标准时间表可以看出构建对象有很大的系统消耗，并且在一次循环中还要清除掉该对象，下循环再重新构建。

9.6.3 break

遍历数组、向量或者树结构时，如果满足条件的元素找到，一定要

使用 break 语句退出循环。

10. 日志（Log）

Log 是项目开发中重要的细节，良好的 Log 方式可以加快开发速度、提高测试时 Bug 查找的效率，并且在测试后维护阶段容易跟踪程序的状态。

项目应该编码前就决定 Log（日志）方式，甚至提供相应的类包或者类。所有的开发人员使用相同风格的日志，可以提高项目整体的可读性，更有助于代码的重用。对于调试来讲，在程序中插入 Log 语句不是很困难，也不是很必要，但是在调试器不灵敏时，这显得格外重要。

目前主要使用的有 JDK 1.4 Beta 自带的 Log 包，另外还有一些开放源码组织提供的日志功能，比如 Apache 组织的 Log4J 和 LogKit 等都是很值得借鉴和引用的。

11. 其它

11.1 exit()

exit 除了在 main 中可以被调用外，其他的地方不应该调用。因为这样做不给任何代码机会来截获退出。一个类似后台服务的程序不应该因为某一个库模块决定了要退出就退出。

11.2 异常

声明的错误应该抛出一个 RuntimeException 或者派生的异常。顶层的 main()方法应该截获所有的异常，并且打印（或者记录在日志中）在屏幕上。

将连续的小的“try-catch”块合并到一起。由于这些块将代码分割成小的、各自独立的片断，所以会妨碍编译器进行优化。但是，若过份热衷于删除异常处理模块，也可能造成代码健壮程度的下降，所以在合并的同时需要考虑程序的稳定性。

11.3 类名的唯一性

请保证在自己类路径指到的任何地方，每个名字都仅对应一个类。否则，编译器可能先找到同名的另一个类，并报告出错消息。若怀疑自己碰到了类路径问题，请试试在类路径的每一个起点，搜索一下同名的.class 文件。