1. Java 编程规范
   1. 注释

示例：

/\*\*

\* 根据日志类型和时间读取日志。

\* 分配对应日志类型的LogReader， 指定类型、查询时间段、条件和反复器缓冲数，

\* 读取日志记录。查询条件为null或0的表示没有限制，反复器缓冲数为0读不到日志。

\* 查询时间为左包含原则，即 [startTime, endTime) 。

\* @param logTypeName 日志类型名（在配置文件中定义的）

\* @param startTime 查询日志的开始时间

\* @param endTime 查询日志的结束时间

\* @param logLevel 查询日志的级别

\* @param userName 查询该用户的日志

\* @param bufferNum 日志反复器缓冲记录数

\* @return 结果集，日志反复器

\* @since 1.2

\*/

public static LogIterator read(String logType, Date startTime, Date endTime, int logLevel, String userName, int bufferNum)

1. 对于方法内部用throw语句抛出的异常，必须在方法的注释中标明，对于所调用的其他方法所抛出的异常，选择主要的在注释中说明。 对于非RuntimeException，即throws子句声明会抛出的异常，必须在方法的注释中标明。(1.42+)

说明：异常注释用@exception或@throws表示，在JavaDoc中两者等价，但推荐用@exception标注Runtime异常，@throws标注非Runtime异常。异常的注释必须说明该异常的含义及什么条件下抛出该异常。

1. 注释应与其描述的代码相近，对代码的注释应放在其上方，并与其上面的代码用空行隔开，注释与所描述内容进行同样的缩排。(1.42+)

说明：可使程序排版整齐，并方便注释的阅读与理解。

示例：

/\*

\* 注释

\*/

public void example2() {

// 注释

CodeBlock One

// 注释

CodeBlock Two

}

/\*

\* 注释

\*/

public void example( ) {

// 注释

CodeBlock One

// 注释

CodeBlock Two

}

* 1. 命名
     1. 规则

1. 类名和接口使用类意义完整的英文描述，每个英文单词的首字母使用大写、其余字母使用小写的大小写混合法。(1.42+)

示例：OrderInformation, CustomerList, LogManager, LogConfig, SmpTransaction

1. 方法名使用类意义完整的英文描述：第一个单词的字母使用小写、剩余单词首字母大写其余字母小写的大小写混合法。(1.42+)

示例：

private void calculateRate();

public void addNewOrder();

1. 方法中，存取属性的方法采用setter 和 getter方法，动作方法采用动词和动宾结构。(1.42+)

格式：

get + 非布尔属性名()

is + 布尔属性名()

set + 属性名()

动词()

动词 + 宾语()

示例：

public String getType();

public boolean isFinished();

public void setVisible(boolean);

public void show();

public void addKeyListener(Listener);

1. 属性名使用意义完整的英文描述，第一个单词的字母使用小写，剩余单词首字母大写其余字母小写的大小写混合法。属性名不能与方法名相同。(1.42+)

示例：

private customerName;

private orderNumber;

private smpSession;

1. 常量名使用全大写的英文描述，英文单词之间用下划线分隔开，并且使用 static final修饰。(1.42+)

示例：

public static final int MAX\_VALUE = 1000;

public static final String DEFAULT\_START\_DATE = "2001-12-08";

* + 1. 建议

1. 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.pachira开头(不包括一些特殊原因)。在部门内部应该规划好包名的范围，防止产生冲突。部门内部产品使用部门的名称加上模块名称。产品线的产品使用产品的名称加上模块的名称。(1.42+)

说明：除特殊原因包结构都必须以com.pachira开头，如果因为OEM合作等关系，可以不做要求。

格式：

com.pachira.产品名.模块名称

示例：

融合WEBSMAP包名 com.pachira.iin.websmap

1. 通过对函数或过程、变量、结构等正确的命名以及合理地组织代码的结构，使代码成为自注释的。(1.42+)

说明：清晰准确的函数、变量等的命名，可增加代码可读性，并减少不必要的注释。

1. 常用组件类的命名以组件名加上组件类型名结尾。(1.42+)

示例：

Application 类型的，命名以App 结尾——MainApp

Frame 类型的，命名以Frame 结尾——TopoFrame

Panel 类型的，建议命名以Panel 结尾——CreateCircuitPanel

Bean 类型的，建议命名以Bean 结尾——DataAccessBean

EJB 类型的，建议命名以EJB 结尾——DBProxyEJB

Applet 类型的，建议命名以Applet 结尾——PictureShowApplet

1. 如果函数名超过15 个字母，可采用以去掉元音字母的方法或者以行业内约定俗成的缩写方式缩写函数名。(1.42+)

示例：

getCustomerInformation() 改为 getCustomerInfo()

1. 准确地确定成员函数的存取控制符号:只是该类内部调用的函数使用 private 属性，继承类可以使用的使用protected属性，同包类可以调用的使用默认属性(不加属性控制符号)，对外公开的函数使用public属性(1.42+)

示例：

protected void getUserName() {

。。。。。。

}

private void calculateRate() {

。。。。。。

}

1. 含有集合意义的属性命名，尽量包含其复数的意义。(1.42+)

示例：

customers, orderItems

* 1. 编码
     1. 规则

1. 数据库操作、IO操作等需要使用结束close()的对象必须在try-catch-finally 的finally中close()，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。(1.42+)

示例：

try {

// ... ...

} catch(IOException ioe) {

//... ...

} finally {

try {

out.close();

} catch (IOException ioe) {

//... ...

}

try {

in.close();

} catch (IOException ioe) {

//... ...

}

}

1. 系统非正常运行产生的异常捕获后，如果不对该异常进行处理，则应该记录日志。 (1.42+)

说明：此规则指通常的系统非正常运行产生的异常，不包括一些基于异常的设计。若有特殊原因必须用注释加以说明。

示例：

try {

//.... ...

} catch (IOException ioe) {

logger.error(ioe);

}

1. 自己抛出的异常必须要填写详细的描述信息。(1.42+)

说明：便于问题定位。

示例：

throw new IOException("Writing data error! Data: " + data.toString());

1. 运行时异常使用RuntimeException的子类来表示，不用在可能抛出异常的方法声明上加throws子句。非运行期异常是从Exception继承而来的，必须在方法声明上加throws子句。(1.42+)

说明：非运行期异常是由外界运行环境决定异常抛出条件的异常，例如文件操作，可能受权限、磁盘空间大小的影响而失败，这种异常是程序本身无法避免的，需要调用者明确考虑该异常出现时该如何处理方法，因此非运行期异常必须有throws子句标出，不标出或者调用者不捕获该类型异常都会导致编译失败，从而防止程序员本身疏忽。

运行期异常是程序在运行过程中本身考虑不周导致的异常，例如传入错误的参数等。抛出运行期异常的目的是防止异常扩散，导致定位困难。因此在做异常体系设计时要根据错误的性质合理选择自定义异常的继承关系。

还有一种异常是Error 继承而来的，这种异常由虚拟机自己维护，表示发生了致命错误，程序无法继续运行例如内存不足。我们自己的程序不应该捕获这种异常，并且也不应该创建该种类型的异常。

1. 在程序中使用异常处理还是使用错误返回码处理，根据是否有利于程序结构来确定，并且异常和错误码不应该混合使用，推荐使用异常。(1.42+)

说明：

一个系统或者模块应该统一规划异常类型和返回码的含义。

但是不能用异常来做一般流程处理的方式，不要过多地使用异常，异常的处理效率比条件分支低，而且异常的跳转流程难以预测。

注意：Java 5.0 程序内部的错误码可以使用枚举来表示。

1. 注意运算符的优先级，并用括号明确表达式的操作顺序，避免使用默认优先级。(1.42+)

说明：防止阅读程序时产生误解，防止因默认的优先级与设计思想不符而导致程序出错。

示例：

下列语句中的表达式

word = (high << 8) | low (1)

if ((a | b) && (a & c)) (2)

if ((a | b) < (c & d)) (3)

如果书写为

high << 8 | low

a | b && a & c

a | b < c & d

（1）（2）虽然不会出错，但语句不易理解；（3）造成了判断条件出错。

1. 避免使用不易理解的数字，用有意义的标识来替代。涉及物理状态或者含有物理意义的常量，不应直接使用数字，必须用有意义的静态变量或者枚举来代替。使用异常来表示方法执行错误，而不是使用C++的错误返回码方式。(1.42+)

示例：如下的程序可读性差。

if (state == 0) {

state = 1;

... // program code

}

应改为如下形式：

private final static int TRUNK\_IDLE = 0;

private final static int TRUNK\_BUSY = 1;

private final static int TRUNK\_UNKNOWN = -1;

if (state == TRUNK\_IDLE) {

state = TRUNK\_BUSY;

... // program code

}

注意：Java 5.0 下建议使用枚举来表示。

异常：

public void function() {

...

throw new RuntimeException(“。。。”);

}

1. 数组声明的时候使用 int[] index ，而不要使用 int index[] 。(1.42+)

说明：使用int index[] 格式使程序的可读性较差，int [] index 表示声明了一个int数组(int [])叫做index

示例：

如下程序可读性差：

public int getIndex()[] {

....

}

如下程序可读性好：

public int[] getIndex() {

....

}

1. 不要使用 System.out 与 System.err 进行控制台打印，应该使用工具类(如：日志工具)进行统一记录或者打印。(1.42+)

说明：代码发布的时候可以统一关闭控制台打印，代码调试的时候又可以打开控制台打印，方便调试。

1. 用调测开关来切换软件的DEBUG版和正式版，而不要同时存在正式版本和DEBUG版本的不同源文件，以减少维护的难度。 (1.42+)
2. 集合必须指定模板类型(5.0+)

说明：方便程序阅读，除去强制转换代码

示例：

Map<String,MyObject> map = new HashMap<String,MyObject>();

1. 一个文件不要定义两个类(并非指内部类)。(1.42+)

说明：方便程序的阅读与代码的维护

1. 所有的数据类必须覆写toString()、hashCode()、equals() 方法，toString()方法返回该类有意义的内容。(1.42+)

说明：方便数据类的比较，父类如果实现了比较合理的toString() ，子类可以继承不必再重写。

hashCode与equals可以使用eclipse自动生成。

示例：

public TopoNode {

private String nodeName;

public String toString() {

return "NodeName : " + nodeName;

}

}

1. 判断语句不要使用”\* == true”来判断为真

说明：方便阅读，减少没有必要的计算

以下错误：

if (ok == true) {

……

}

以下正确：

if (ok) {

……

}

1. 不要写没有必要的向上强制转型。(1.42+)

说明：没必要写的向上强制转型会浪费性能，增加代码阅读难度

示例：

以下错误：

FileInputStream fis = new FileInputStream(f);

InputStream is = (InputStream)fis;

* + 1. 建议

1. 记录异常不要保存exception.getMessage()，而要记录exception.toString()，一般可通过日志工具记录完整的异常堆栈信息。(1.42+)

说明：NullPointException抛出时常常描述为空，这样往往看不出是出了什么错。

示例：

try {

...

} catch (FileNotFoundException e) {

logger.error(e);

}

1. 一个方法不应抛出太多类型的异常。(1.42+)

说明： 如果程序中需要分类处理，则将异常根据分类组织成继承关系。如果确实有很多异常类型首先考虑用异常描述来区别，throws/exception子句标明的异常最好不要超过三个。

1. 异常捕获尽量不要直接 catch (Exception ex)，应该把异常细分处理。(1.42+)

说明：可以设计更合理异常处理分支

1. 如果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。(1.42+)

说明：若此段代码各语句之间有实质性关联并且是完成同一件功能的，那么可考虑把此段代码构造成一个新的方法。

1. 集合中的数据如果不使用了应该及时释放，尤其是可重复使用的集合。(1.42+)

说明：由于集合保存了对象的引用，虚拟机的垃圾收集器就不会回收。

1. 源程序中关系较为紧密的代码应尽可能相邻。(1.42+)

说明：便于程序阅读和查找。

示例：矩形的长与宽关系较密切，放在一起。

rect.length = 10;

rect.width = 5;

1. 不要使用难懂的技巧性很高的语句，除非很有必要时。(1.42+)

说明：高技巧语句不等于高效率的程序，实际上程序的效率关键在于设计与算法。

1. 明确方法功能，精确（而不是近似）地实现方法设计。一个函数仅完成一件功能，即使简单功能也编写方法实现。 (1.42+)

说明：虽然为仅用一两行就可完成的功能去编方法好象没有必要，但用方法可使功能明确化，增加程序可读性，亦可方便维护、测试。

1. 应明确规定对接口方法参数的合法性检查应由方法的调用者负责还是由接口方法本身负责，缺省是由方法调用者负责。(1.42+)

说明：对于模块间接口方法的参数的合法性检查这一问题，往往有两个极端现象，即：要么是调用者和被调用者对参数均不作合法性检查，结果就遗漏了合法性检查这一必要的处理过程，造成问题隐患；要么就是调用者和被调用者均对参数进行合法性检查，这种情况虽不会造成问题，但产生了冗余代码，降低了效率。

1. 尽量使用Java 5.0新循环写法。(5.0+)

说明：代码更加简洁

示例：

ArrayList<String> list = new ArrayList<String>();

list.add...

for(String str:list) {

System.out.println(str);

}

1. 使用Java 5.0枚举来替代以前用数字与字符串的同等目的的操作。(5.0+)

说明：Java 5.0以前没有枚举，大家都用数字或者字符串做枚举同样功能的事情

示例：

public enum EnumDemo {

ERROR,INFO,DEBUG

}

In other function：

EnumDemo t = EnumDemo.DEBUG;

if (t == EnumDemo.ERROR) {

。。。。。。

}

1. interface 中定义的常量不要写public、static、final的修饰词，方法不要写public修饰词。(1.42+)

说明：更加简洁

示例：

public interface InterfaceT {

String TT = "abcl";

void doStart();

}

1. 新起一个线程，都要使用Thread.setName(“…”)设置线程名。

说明：性能测试时可对线程状态进行监控，异常时也可以知道异常发生在哪个线程中

* 1. 性能与可靠性
     1. 规则

1. 数组复制使用System.arraycopy(\*) 。(1.42+)

说明：更好的性能

1. 不要使用循环将集合转为数组，可以使用集合的toArray()方法。(1.42+)

说明：更好的性能，代码更加简洁

示例：

ArrayList list = new ArrayList();

list.add....

String [] array = new String[list.size()];

list.toArray(array);

1. 大量字符串的相加等于处理应该使用StringBuffer。(1.42+)

说明：大量的String相加等于处理性能消耗较多。“大量”一般指5次“+=”以上或者在循环中进行字符串+=操作。

示例：

不推荐：

String str = “”;

str += ”a”;

str += ”b”;

推荐：

StringBuffer sb = new StringBuffer();

sb.append(“aa”);

sb.append(“bb”);

sb.append(“cc”);

1. 对类中日志工具对象logger应声明为static. (1.42+)

说明：防止重复new 出logger对象(logger指各种日志工具类，可以是log4j，common log，jdk logger，内部API等，尽管一些logger对LogFactory工厂有一些优化，但是我们也必须防止代码没有必要的运行)。

* + 1. 建议

1. 对于频繁执行的代码，其Debug，Info级别日志输出前须对当前的调试等级先进行判断。(1.42+)

说明：日志一般都会有不少字符串的处理，如果不是Debug级别就没有必要进行处理

示例：

if (logger.debugEnable()) {

logger.debug(“request : ” + request.getMethod());

}

1. public类型的底层函数需对输入参数进行判断，参数不合法应该主动抛出RuntimeException。(1.42+)

说明：底层函数必须保证输入参数正确性再进行其他处理(防止后面的代码抛出错误，减少没有必要的后续代码运行)。使用RuntimeException 减少了try catch满天飞，并有利于快速定于异常代码。

示例：

public void doDivide(int a,int b) {

if (b == 0) {

throw new IllegalArgumentException(“denominator can’t be zero”);

}

...

}

1. 尽量使用JDK自带的API函数，不要自己写类似功能的函数. (1.42+)

说明：JDK自身的函数在可靠性，性能方面一般有更好的表现，大家必须熟练掌握，特别是算法方面的。

1. IO操作流使用有Buffer功能的class. (1.42+)

说明：更好的性能，没有Buffer的输出流频繁IO操作，效率反倒低。

示例：

FileOutputStream file= new FileOutputStream("test.txt");

BufferedOutputStream out = new BufferedOutputStream(file);

for (int i = 0; i < bytes.length; i++ ) {

out.write(…);

}

out.flush();

* 1. 日志
     1. 规则
  2. 日志输出时如有相关错误码或Exception信息，则必须同时输出

说明：这有助于加速定位错误原因

* 1. 严禁滥用日志等级

说明： 合理的日志等级有利于日志分析人员根据问题严重性，筛选信息进行针对性分析。

等级输出说明：

FATAL为严重错误，用于输出系统严重中问题，无法预估的且产生严重影响的错误，该类日志需要重点关注；

ERROR为执行错误，用于输出系统基本错误信息，该类错误需要重点关注；

WARN为警告错误

INFO为正常执行状态、进度等信息输出，注意INFO等级避免输出太多重复信息。

DEBUG为调试等级，用于输出开发专注的调试信息，用于辅助调试

TRACE为跟踪等级，用户跟踪代码的详细执行状态

* 1. 在循环内部，注意使用if语句在必要时输出Log，也可以在循环开始、结束位置使用Log

说明：频繁的Log输出会对性能造成影响，尤其在算法复杂度很高的情况下，

* + 1. 建议

1. 组件级别的输入、输出信息，如参数，尽量写入到DEBUG等级Log中

说明：有助于快速定位问题，代码工作异常与输入输出往往关系很大。

1. 尽量使用str1 + str2的方式拼接字符串，这比format方式更快
2. 默认日志等级为INFO，确保正常执行情况下，Log输出处于可用范围内，且不丢失关键信息，根据问题分析需要再调整等级到DEBUG、TRACE