### 判断

#### case语句块结束时如果不加break，需要有注释说明(fall-through)

对

错

#### case语句应该尽可能短，不影响整体条件分支处理逻辑；如果长了，应该考虑封装方法

对

错

#### Java8中可使用Optional代替null作为返回值或者可能的缺省值

对

错

#### Java平台类库的String、BigInteger和BigDecimal属于不可变类

对

错

#### String s = new String("string");存在对象重复创建，资源浪费情况。

对

错

#### 避免对第三方代码的强依赖或陷入第三方代码细节

对

错

#### 避免枚举常量序号的产生依赖于ordinal()方法

对

错

#### 必须在代码中硬编码"\r"和"\n"作为换行符号

对

错

#### 不建议Java程序访问Windows注册表。如果必须，也不建议使用Native Method访问。

对

错

#### 不要把方法的入参当做工作变量/临时变量

对

错

#### 不应该随意进行类型强制转换，应改善设计，或在转换前用instanceof

对

错

#### 不用的代码段不删除而注释保留，可以为后续可能的使用提供便利。

对

错

#### 创建线程是指定线程名的好处之一是方便问题定位

对

错

#### 代码List objectList = new ArrayList(); //Incompatible types objectList.add("I don't fit in");会产生ClassCastException异常

对

错

#### 代码中的注释、字符串、日志等，如果是中文应该无错别字、语句通顺，如果是英文应该拼写无误、语法正确

对

错

#### 代码注释放于对应代码的上方或右边或下方

对

错

#### 当方法只会抛出运行时异常（runtime exception）时，没有必要在方法注释中说明该异常

对

错

#### 当抛出异常时，应该提供尽肯能多的信息，以便分析“异常是如何产生的”

对

错

#### 调用重载的方法是在编译期确定的

对

错

#### 调用静态方法，可以使用实例或者表达式来调用

对

错

#### 多线程操作同一个字符串相加，应采用StringBuilder

对

错

#### 多线程访问同一个可变变量，需增加同步机制

对

错

#### 对可恢复的情况使用受检异常（checked exception），对编译错误使用运行异常（runtime exception）

对

错

#### 对于返回数组或者容器的方法,应返回null。再在上层调用代码里对返回值判空。

对

错

#### 对于类的访问控制块的声明顺序应该是private,producted,public

对

错

#### 方法命名无法表达的信息，必须加方法头注释辅助说明；禁止空有格式的方法头。

对

错

#### 方法尽量通过方法名自注释，按需写方法头注释。

对

错

#### 浮点型数据可以使用==判断相等

对

错

#### 根据Java Language Specification 中对Java内存模型的定义,每个线程都有自己的工作内存(Working Memory),线程间的变量访问需要通过主内存完成

对

错

#### 记录异常的日志应为error或fatal级

对

错

#### 建议通过Java1.5提供的新并发工具代替wait和notify

对

错

#### 建议通过Thread.stop方法终止线程运行

对

错

#### 将变量设置为私有（private）的原因在于封装类内部实现细节。

对

错

#### 禁止使用主动GC（除非在密码、RMI等方面），尤其是在频繁/周期性的逻辑中

对

错

#### 静态成员变量是属于某个对象实例，而不是属于类级别的变量

对

错

#### 静态成员方法可以在子类与父类之间覆写（override）

对

错

#### 可以不对第三方抛出的异常进行封装，这样可以直接利用第三方的异常来处理。

对

错

#### 可以使用错误码来代替异常来做错误处理

对

错

#### 可以通过使用JAVA的线程优先级来实现业务的优先级调度

对

错

#### 可以在单个的表达式中对相同的变量赋值超过一次

对

错

#### 类和方法的设计可优先考虑泛型。

对

错

#### 每个switch都必须要有default

对

错

#### 每个方法都必须要有方法头注释

对

错

#### 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。

对

错

#### 声明一个带泛型的类或接口的时候，建议限制可以用的泛型类型，避免接口使用者乱用。

对

错

#### 使用第三方提供的代码包时，建议使用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。

对

错

#### 使用泛型，当从集合类中读取一个对象时，就不用手工进行类型转换

对

错

#### 使用线程安全集合可以在多线程间共享可变数据，替代传统的synchronized关键字

对

错

#### 所有方法都要加方法头注释。

对

错

#### 推荐使用本地方法（native method），以提升系统性能。

对

错

#### 为方便排错，在抛出异常的细节信息中包含能捕获失败的信息，但是不要包含敏感信息或者个人信息

对

错

#### 为了保证每个模块都是单一职责，文件不要超过2000行

对

错

#### 为了适配采用系统默认字符编码的遗留系统，建议直接改用默认字符编码

对

错

#### 为了提高代码的简洁性，请尽量使用缩写

对

错

#### 为了提高代码的可读性，一行代码越长越好，这样不用换行阅读

对

错

#### 为了提升性能，可以不加控制的创建线程

对

错

#### 相比匈牙利命名风格，驼峰将类型与命名解耦合，功能更单一

对

错

#### 像写普通代码注释一样去写方法头注释，写真正需要读者阅读的方法头

对

错

#### 写注释时要换位思考，用注释去表达此时读者真正需要的信息。在代码的功能、意图层次上进行注释，即注释解释代码难以表达的意图，不要重复代码信息。

对

错

#### 新代码不要使用已标注为@Deprecated的方法。

对

错

#### 修改代码时，也要保证其相关注释的一致性。只改代码，不改注释是一种不文明行为，破坏了代码与注释的一致性，让阅读者迷惑、费解，甚至误解

对

错

#### 序列化对象中的HashMap、HashSet或者HashTable等集合可以包含对象自身的引用

对

错

#### 异常可以减少代码嵌套的层次，而错误码会增加代码的嵌套

对

错

#### 一个局部变量只应该表达一种含义

对

错

#### 一般的，尽量通过清晰的架构逻辑，好的符号命名来提高代码可读性；需要的时候，才能辅以注释说明。

对

错

#### 英文字符串的大小写转换，必须使用Locais.ENGLISH。

对

错

#### 优秀的代码尽量通过符号命名自我解释，然后按需辅以注释

对

错

#### 右置注释与前面代码至少1空格

对

错

#### 与C++相同，在构造方法中调用的成员方法只可能是当前类的实现，不可能被子类覆盖

对

错

#### 在Java代码中，只能使用//方式的注释

对

错

#### 在程序中，允许空的catch块来忽略异常

try

{

…

}catch(SomeException e)

{

}

对

错

#### 在集合类中我们推荐使用泛型，这样在编译的时候就会检查元素类型

对

错

#### 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。

对

错

#### 在业务流程中，可以使用异常机制来做流程机制

对

错

#### 在中文Windows环境下，配置文件应采用默认GBK编码

对

错

#### 字符串与字节数组之间的转换，应使用系统默认的编码方式

对

错

#### 正式交付给客户的代码不能包含TODO/TBD/FIXME注释

对

错

#### 直接捕获基类异常Exception可以减少代码量，方便以后维护

对

错

#### 注释符与注释内容间不要用空格。

对

错

#### 注释内容要简洁、明了、无二义性，信息全面且不冗余

对

错

#### 注释是为了帮助阅读者快速读懂代码，所以要从读者的角度出发，按需注释

对

错

#### 注释一定要以满足“代码注释率”为目标

对

错

### 单选题

#### PECS指"Producer Extends，Consumer Super"。如果参数化类型表示一个生产者，就是用（）；如果它表示一个消费者，就使用（）。

<? super T>，<? extends T>

<? extends T>，<? super T>

< T1>，<T2>

#### 避免方法的代码块嵌套过深，不要超过()层

5

4

3

2

#### 不影响业务进行的错误用()级

1. error级
2. debug级
3. warn级
4. info级

#### 代码private final Collection starnps=…;stamps.add(new Coin(…));会在()报错

1. 加载时
2. 编译时
3. 执行时

#### 对于常量和枚举的命名，说法错误的是

枚举的值和类型一样，应该都是大驼峰

static final常量应该是全大写

对于方法局部常量，使用小驼峰

常量和枚举的值全大写时，应该使用下划线进行单词连接

#### 对待第三方API抛出的各类异常时，以下说法正确的是

1. 不做处理，继续抛出
2. 捕获并转换成相应的业务异常
3. 捕获后不处理

#### 对于类型命名，下面写法错误的是

enum FileType

class ParkingLot

interface file

abstract class PropertiesMap

#### 对于如下代码，描述正确的是？

void onRequest(int opr,byte[]msg) {

if (opr== NetConst.ADD){

……

}else if (opr == NetConst.DELETE){

……

}else if (opr == NetConst.MODIFY){

……

}

}

1. 缺少一个default:子句
2. 没毛病
3. 缺少一个收尾的else语句

#### 方法长度建议不超过()行

1. 120
2. 100
3. 80
4. 50

#### 父类的构造方法中调用可能被子类覆写的方法造成的后果可能是

子类对象还未生成，不可能调用到子类方法

只可能调用到当前类的成员方法，没有问题

子类包含的对象还未生成，导致NullPointerException异常

父子类方法都无法调用到

#### 关于JAVA线程调度器，说法正确的是

1. 不要依赖线程调度器来决定程序的逻辑运行
2. JAVA的优先级调度在不同操作系统上行为一致
3. 只要遵循相同的JVM规范，各个实现的线程调度器功能都完全相同
4. JAVA线程调度器是平台无关的，所以可以跨平台使用

#### 关于Thread.stop方法，说法正确的是：

主要是跟应用场景有关，在特定场景下可以使用

调用Thread.stop是安全的

它可能会导致数据不一致或遭到破坏

推荐使用

#### 关于volatile的说法，正确的是：

1. 被volatile修饰的变量多线程并发修改是安全的
2. volatile关键字经常用在多个线程并发写、某个线程制度的场景
3. 可以完全代synchronize关键字
4. 对volatile修饰的变量的修改，其他线程可见

#### 关于变量初始化，说法正确的是

任何情况变量定义时都要初始化

变量按需初始化

可以使用未经初始化的变量

冗余初始化更安全

#### 关于方法命名，说法错误的是

1. 方法采用大驼峰
2. 方法采用动词
3. 方法采用小驼峰
4. 方法采用动宾结构

#### 关于方法命名描述错误的是：

方法命名禁止使用单词缩写

方法的命名要符合英语阅读习惯

方法的命名要准确描述方法功能

避免使用无意义或含义不清的单词为方法名

#### 关于魔鬼数字，说法正确的是：

1. 0、1、-1肯定不是魔鬼数字
2. 所有的数字都是魔鬼数字
3. 难以理解的数字就是魔鬼数字
4. 100肯定是魔鬼数字

#### 关于受检异常(check exception)，以下说法正确的是

1. 在难以恢复的情况下应该使用受检异常
2. 受检异常应该是RuntimeException的子类
3. 可恢复的场景应该使用受检异常

#### 关于线程安全性的文档说明，说法错误的是：

不提供文档说明容易导致过度同步或者没有同步，引起严重的功能或者性能问题

它是与其调用者程序建立的约定的重要组成部分

主要作用是为了以后自己维护代码方便

不需要使用者对这个类的线程安全性做假设

#### 关于线程异常，说法正确的是：

1. 线程不应该捕获unchecked exception
2. 线程异常由JVM虚拟机捕获的处理，开发者不需要关心
3. 可以通过Thread对象提供的setUncaughtExceptionHandler方法获取线程中产生的异常
4. 只要抛出unchecked exception，线程就必须结束运行

#### 建议方法的参数个数不超过()个

1. 7
2. 8
3. 6
4. 5

#### 建议使用()来做错误处理

异常

包装类

错误码

日志

#### 如果一个对象存入Hash集合后hashcode随即发送变化，会导致什么结果？

无任何问题

内存溢出

未知异常

内存泄露

#### 什么时候需要增加空行

1. 方法调用之后
2. 指示函之后
3. 相对独立的程序块之间
4. 变量定义之前

#### 为了保证代码缩进统一，建议每次缩进是几个空格

1. 1个Tab符
2. 4个空格
3. 2个空格
4. 8个空格

#### 下面哪段代码不会导致内存泄露

1. public <T> pop(){

if(size==0){

throw new EmptyStackException();

}

T result = elements[--size];

elements[size]=null;

return result;

}

B. public <T> pop(){

if(size==0){

throw new EmptyStackException();

}

return elements[--size];

}

#### 下面哪种是不规范的做法?

1. System.out.print("Hello,World!\n");
2. System.out.println("Hello,World!");
3. System.out.println();
4. System.out.print(System.lineSeparator());

#### 下面的代码运行结果是：()

public static final void main(String[] args){

try{

System.out.println("aa"+aa());

}catch(Exception e){

e.printStrackTrace();

}

}

Public static int aa() throws Exception {

int a=1;

for (int i = 1;i < 2; i++){

try{

throw new Exception ("bb");

}catch(Exception ex){

throw ex;

}finally{

return 1;

}

}

return 0;

}

1. 打出异常堆栈
2. aa:bb
3. aa:0
4. aa:1

#### 需要精确计算时建议使用类型

都可以

double

float

都不可以

#### 一般的业务处理日志用()级

1. debug级
2. warn级
3. info级
4. error级

#### 异常机制不应用于处理以下哪种情况？

意料之外的情况

业务流程控制

罕见的错误

正常流程无法正常运行

#### 以下代码可能存在什么问题：

public pop(){

if(size == 0){

throw new EmptyStackException();

}

Return elements[--size];

}

数组越界

内存泄露

没毛病

内存溢出

#### 以下关于equals方法和hashCode方法描述错误的是

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

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

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

"覆写equals方法时，应同时覆写hashCode方法"的好处是为了提升查询效率

#### 在Java1.5+，在不需要索引的场景，我们应使用()替代传统的for循环?

1. iterator
2. do{}while()
3. for-each
4. 索引for()

#### 在保证代码风格一致性的基础上，下面那些风格适用于成员变量

1. int m\_fileSize;
2. int filesize;
3. int fileSize;
4. int fileSize\_;

#### 正式交付给客户的代码，针对”TODO”注释，正确的做法是：

将所有”TODO”报例外处理

逐个评估”TODO”，按需要进行处理

直接删除”TODO”自验，防止工具上报

将”TODO”改成”遗留”，防止工具上报又不会丢失这些遗留点

### 多选题

#### 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。使用可变数量参数的方法时注意

1. 不应该在确实需要操作可变长度的值的序列时使用
2. 应该再确实需要操作可变长度的值的序列时使用
3. 建议使用varargs重写使用一个固定长度数组作为参数的方法
4. 不建议使用varargs重写使用一个固定长度组作为参数的方法

#### 不可变类是指其实力一旦创建后就不能被修改，要使类成为不可变类，需遵循的规则有：

1. 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
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. 不要提供任何会修改类状态的方法

#### 不用的代码直接删除掉，而不是注释掉，是因为：

注释掉的代码影响其他有用代码的阅读

单元测试、自动化测试等无法覆盖注释掉的代码，其中有可能隐藏缺陷

注释掉的代码无法被正常维护，不可信任

#### 对于类型命名，下面说法正确的是

1. 枚举的命名应该是大驼峰
2. 类的命名应该是大驼峰
3. 接口的命名应该是大驼峰
4. 注解类型应该是大驼峰

#### 关于大括号的使用，说法正确的是：

1. 左大括号放行末时，其前方要保持1空格
2. if/while/switch/for语句必须要用大括号
3. if/while/switch/for语句的左大括号跟随放行末
4. 方法左大括号另起并独占一行

#### 关于行宽,描述正确的是

1. 使用高分辨率的显示器时,可以放松对行高的要求
2. 比如代码中包括较长字符串,不宜截断,则可以例外,超过行宽
3. 通过简短的方法,变量命名,控制嵌套层数来努力让每行代码达到行宽要求
4. 代码行过长,则不利于阅读

#### 关于空行的使用，说法正确的是：

1. 根据上下内容的相关程度，合理安排空行
2. 相邻的两个方法定义之间的间隔使用空行
3. 大括号内的代码块行首之前和行尾之后不要加空行

#### 关于受检异常(checked exception)、运行时异常（runtime exception）、错误（Error），以下说法正确的是

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

运行时异常是指难以恢复或者不可恢复的程序错误

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

除非必要时，最好不要实现新的 Error子类

如果期望调用者能够恢复，则应该使用受检异常

#### 关于异常，以下说法错误的是

1. 方法抛出的异常数量可以不受限制
2. 方法抛出的异常，应该与本身的抽象层次相对应
3. 不用处理的异常可以使用空catch块进行忽略
4. finally块中可以使用return、continue或break进行流程控制

#### 关于注释，如下描述正确的有？

1. 写注释时要换位思考，用注释去表达此时读者真正需要的信息。在代码的功能、意图层次上进行注释。
2. 注释内容要简洁、明了、无二义性，信息全面且不冗余。
3. 一般的尽量通过清晰的架构逻辑，好的符号命名来提高代码可读性；需要的时候，才辅以注释说明
4. 注释是为了帮助阅读者快速读懂代码，所以要从读者的角度出发，按需注释。
5. 修改代码时，也要保证其相关注释的一致性。

#### 尽量不要实现Serializable接口的原因是？

1. 编写正确的反序列化代码有很大的挑战
2. 序列化增加了测试的难度
3. 序列化容易使一个类对其最初的内部表示产生依赖
4. 序列化增大了安全风险
5. Serializable接口在Java8中已经被标注为deprecated
6. 序列化不必要地对外公开了对象的物理实现

#### 请问如下哪些序列化的实现类，不是最优的？

1. public class OhMyGod implements Serializable

{

public static final long serialVersionUID = -2589766491699675794L;

}

1. public class OhMyGod implements Serializable

{

private static final long serialVersionUID = -2589766491699675794L;

}

1. public class OhMyGod implements Serializable

{

private final long serialVersionUID = -2589766491699675794L;

}

1. public class OhMyGod implements Serializable

{

private static long serialVersionUID = -2589766491699675794L;

}

#### 如下例子,当name为()时,会可能出现性能问题而非可靠性问题?

void doSomething(Lastnames){

for(int i=0;i<names.size();i++){

String name = names.get(i);

decorate(name);

}

}

1. names为Vector实例
2. names为null
3. names的size为0
4. names为LinkedList实例

#### 什么时候需要增加空格

1. 逗号，分号与后续内容之间
2. 变量与自增运算符(++)之间
3. 注释符号注释内容之间
4. If关键字后面

#### 使用包装类型合理的场景有

定义常量时

作为集合中的元素、键和值

进行反射的方法调用时

泛型，必须使用包装类型

#### 我们推荐（）实现常用的功能或者算法.

使用广泛使用的开源库

使用JDK自带的API

自己编写代码

#### 下面命名中适合大驼峰的有哪些

1. 类
2. static final常量
3. 接口
4. 方法

#### 下面命名中适合小驼峰的有哪些

类的字段

方法

方法参数

局部变量

#### 下面哪些变量命名是推荐的？

1. int a;
2. int bookId;
3. int udpConnectionCount;
4. int lineCount;

#### 关于受检异常(checked exception)、运行时异常（runtime exception）、错误(Error) ，以下说法正确的是

1. 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
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
4. 如果期望调用者能够恢复，则应该使用受检异常
5. 除非必要是，最好不要实现新的Error子类

#### 子类覆写父类方法时应加上@Override注解的好处是

方便运行期检查

如果覆写时因为忽略，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

只有加了@Override注解才能达到覆盖父类方法的目的

如果父类修改了方法定义造成子类不再覆写父类方法，能使问题在编译期尽早被发现

#### 正确使用异常机制事半功倍，以下说法正确的是

1. 异常信息中可以包含敏感信息和个人信息
2. 方法抛出的异常，应该与本身的抽象层次相对应
3. 异常机制不能取代正常的业务逻辑
4. 注释和文档汇总应该包含所抛出的异常说明