**一、多线程作业**

**package** com.zhu.thread;

**import** java.util.concurrent.atomic.AtomicInteger;

**public** **class** ThreadControlTest\_2 {

**public** **static** **void** main(String[] args) **throws** InterruptedException {

AtomicInteger obj = **new** AtomicInteger(0);//线程安全

Thread a = **new** Thread(**new** RunThreadName\_2(obj, 0));

a.setName("A");

Thread b = **new** Thread(**new** RunThreadName\_2(obj, 1));

b.setName("B");

Thread c = **new** Thread(**new** RunThreadName\_2(obj, 2));

c.setName("C");

a.start();

b.start();

c.start();

}

}

**class** RunThreadName\_2 **implements** Runnable {

**private** AtomicInteger obj;

**private** **int** flag;//标志 0：A 1：B 2：C

**private** **int** count = 0;//次数

**public** RunThreadName\_2(AtomicInteger obj, **int** flag) {

**this**.obj = obj;

**this**.flag = flag;

}

@Override

**public** **void** run() {

**while**(**true**) {

**synchronized** (obj) {//锁对象唯一

**if**(obj.get() % 3 == flag) {

System.***out***.print(Thread.*currentThread*().getName()+"");

obj.set(obj.get()+1);

count++;

obj.notifyAll();

**if**(count>=10) {

**break**;

}

} **else** {

**try** {

obj.wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

}

}

}

## New 对象

1、考虑用静态工厂方法替代构造器

2、遇到多个构造器参数时要考虑用构建器

静态工厂和构造器有一个共同的局限性，它们不能很好的扩展到大量的可选参数。

重叠的构造器方法可行，但是当有许多参数的时候，代码很难编写，并且难以阅读。

builder方式：用一个中间对象来赋值目标对象

**package** com.zhu.thread;

**public** **class** People {

**private** String name;

**private** **int** age;

**private** String sex;

**public** **static** **class** Builder{

**private** String name;

**private** **int** age;

**private** String sex = "man";

**public** Builder(String name, **int** age) {

**this**.name = name;

**this**.age = age;

}

**public** Builder setSex(String sex) {

**this**.sex = sex;

**return** **this**;

}

**public** People build() {

**return** **new** People(**this**);

}

}

**private** People(Builder builder) {//私有构造函数

**this**.name = builder.name;

**this**.age = builder.age;

**this**.sex = builder.sex;

}

}

People people = new People.Builder("zjj", 26).setSex("man").build();

## 泛型

声明中具有一个或者多个类型参数的类或者接口，就是泛型。

1. 列表优先于数组

列表可以在编译时发现错误，数组在运行时才发现错误

## 枚举

1. 用enum代替int常量或者switch分支时

public static final int *ONE\_DAY* = 1;  
public static final int *TWO\_DAYS* = 2;  
public static final int *SEVENT\_DAYS* = 7;

public enum DAYS {  
*ONE\_DAY*, *TWO\_DAYS*, *SEVENT\_DAYS*;  
}

1. 用实例域代替序数

不要根据枚举的序数导出与它有关联的值，而是将它保存再一个实例域中。

public enum DAYS {  
*ONE\_DAY*(1), *TWO\_DAYS*(2), *SEVENT\_DAYS*(7);  
 private final int num;  
DAYS(int num){  
this.num=num;  
}  
public int getNum(){  
return num;  
}  
}  
  
public static void main(String[] args) {  
 System.*out*.println(DAYS.*ONE\_DAY*.getNum());  
}

## 

## 方法

1. 检查参数的有效性

每当编写方法或者构造器的时候，应该考虑它的参数有哪些限制。

1. 谨慎使用设计方法签名

a、谨慎选择方法的名称。应该遵循标准的命名习惯

b、不要过于追求提供便利的方法

c、避免过长的参数列表

1. 慎用重载

public static String classify(Set<?> s) {  
return "Set";  
}  
  
public static String classify(List<?> lst) {  
return "List";  
}  
  
public static String classify(Collection<?> c) {  
return "Collection";  
}  
  
public static void main(String[] args) {  
 Collection<?>[] collections = {  
new HashSet<String>(),   
 new ArrayList<BigInteger>(),   
 new HashMap<String, String>().values()  
 };  
 for (Collection<?> c : collections) {  
 System.*out*.println(*classify*(c));  
}  
}

期望输出：

Set

List

Collection

实际输出：



原因：因为classify方法被重载了，而要调用哪个重载方法，是在编译时做出的决定。

上例在编译的时候类型都是Collection<?> ，所以只会重载public static String classify(Collection<?> c) {  
return "Collection";  
}

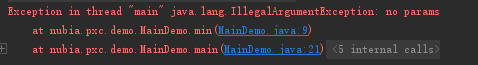
对于具有相同参数数目的方法来说，应该尽量避免重载方法。

1. 慎用可变参数

static int min(int ... args){  
if(args.length==0){  
throw new IllegalArgumentException("no params");  
}  
int min =args[0];  
 for(int a :args){  
if(a<min){  
 min=a;  
}  
 }  
return min;  
}  
public static void main(String[] args) {  
System.*out*.println(*min*(5,2,3,4));//打印 2  
}

如果上面的方法，客户端没有传参数，就会报错。客户端是不可控的。

int min = *min*();  
System.*out*.println(min);



1. 返回零长度的数组或者集合，而不是null

public static List<String>getList(List<String> list){  
if(list.size()==0){  
return null;  
}  
return list;  
}

返回null,每次用到该方法都需要做非空判断

## 通用程序设计

1. 将局部变量的作用域最小化

将局部变量的作用域最小化，可以增强代码的可读性和可维护性，降低出错的可能性。

要使局部变量的作用域最小化，最有力的方法就是在第一次使用它的地方声明。

如果变量是在使用它的块之外被声明的，当程序退出该块之后，该变量仍是可见的。如果变量在它的目标使用区域之前或者之后被意外使用的话，后果可能是灾难性的。

1. for-each循环优先于传统的for循环

用for的做法，索引变量很容易造成一些混乱。for-each循环，通过隐藏索引变量，避免了混乱和出错的可能。

List<String> list ;//定义一个list  
for(int i =0;i<list.size();i++){//for  
 System.*out*.println(list.get(i));  
}

for(String str:list){//for-each  
 System.*out*.println(str);  
}

1. 了解和使用类库

通过使用标准类库，可以充分利用这些编写标准类库的专家的知识，以及其他使用人的经验。

1. 如果要精确答案，避免使用float和double

float和double执行二进制浮点运算，是为了在广泛的数值范围上提供较为精确的快速近似计算而精心设计的。他们并没有提供完全精确的结果。

如果想让系统记录十进制小数点，并且不介意不使用基本类型而带来的不便，可以使用BigDecimal。

final BigDecimal TEN\_CENTS = new BigDecimal(".10");  
BigDecimal a= new BigDecimal("1.00");  
BigDecimal price = a.add(TEN\_CENTS);  
System.*out*.println(price);//打印结果1.10

如果数值没有超过9位十进制数字，可以用int；如果不超过18位数字，可以用long；超过18位，就必须用BigDecimal。

1. 基本类型优先于装箱基本类型

JAVA类型包括基本类型和引用类型。每个基本类型都有一个对应的引用类型，称作装箱基本类型。如int->Integer、double->Double、boolean->Boolean

static Integer *i*;  
public static void main(String[] args){  
if(*i*==2){  
 System.*out*.println("Unbelievable");  
}  
}

在计算i==2的时候抛出空指针异常。问题在于i是Integer,而不是int,Integer的初始值是null

而且，自动拆箱是有范围的，比如，int 是-127到127。如果超过127



而且，自动拆箱是有范围的，比如，int 是-127到127。如果超过127，直接写==会报错；

Integer a = 129;

Integer b = 129;

System.***out***.print(a==b);

System.***out***.println(a.intValue()==b.intValue());

输入：

false

true

把static Integer i;换成static int i;时，不会报这个错。

1. 如果其他类型更适合，则尽量避免使用字符串

字符串不适合替代其他的值类型。若使用不当，字符串会比其他的类型更加笨拙、速度更慢。

1. 当心字符串连接的性能。

字符串连接符（“+”）。

当两个字符串被连接在一起，它们的内容都要被拷贝。使用StringBuilder替代String.

1. 通过接口引用对象

应该使用接口而不是类作为参数的类型。应该优先使用接口而不是类来引用对象。

1. 接口优先于反射机制

反射机制提供了通过程序来访问已装载的类的信息的能力。但是通过反射机制：

a、丧失了编译时类型检查的好处：企图调用不存在的或不可访问的方法，运行时会失败。

b、执行反射访问所需要的代码非常笨拙和冗长

c、性能损失

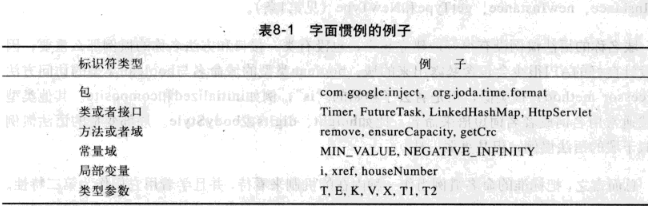
应该仅仅使用反射机制来实例化对象，访问对象时则使用编译时已知的某个接口或者超类。

1. 谨慎地使用本地方法

本地方法指用本地程序设计语言（C/C++）来编写的特殊方法.

本地代码中的一个bug就有可能破坏整个应用程序。

1. 遵守普遍接受的命名惯例



## 异常

1. 只针对异常的情况才使用异常

异常是为了异常情况下使用而设计的，不要将它们用于普通的控制流，也不要编写迫使它们这么做的API。

1. 可恢复的情况使用受检异常，对编程错误使用运行时异常

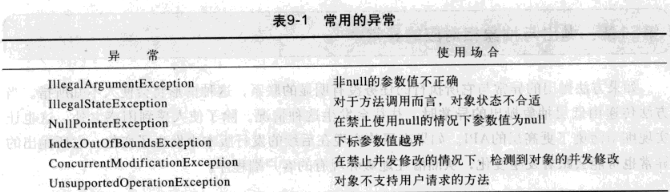
JAVA提供三种可抛出结构：受检异常、运行时异常和错误。

如果期望调用者能够适当地恢复，对于这种情况应该使用受检异常。通过抛出受检异常，强迫调用者在一个catch子句中处理该异常，或者将它传播出去。

如果程序抛出未受检的异常或错误，往往就属于不可恢复的情形，继续执行下去有害无益。

1. 避免不必要的使用受检异常
2. 优先使用标准的异常

重用现有的异常，增加可读性。



1. 抛出与抽象相对应的异常

如果方法抛出的异常与它所执行的任务没有明显的联系，将会使人不知所措。

1. 每个方法抛出的异常都要有文档

要为编写的每个方法所能抛出的每个异常建立文档。

1. 不要忽略异常

空的catch块会使异常达不到应有的目的。

## 并发

1. 同步访问共享的可变数据

当多个线程共享可变数据的时候，每个读写数据的线程必须执行同步。否则，就无法保证一个线程所做的修改可以被另一个线程获知。

1. 避免过度同步

在一个被同步的区域内部，不要调用设计成要被覆盖的方法，或者是由客户端以函数对象的形式提供的方法。因为这种方法是外来的，不可空，从同步方法中调用它会导致异常、死锁或者数据损坏。

1. executor和task优先于线程

ExecutorService 比Thread使用起来更方便，易于管理。

1. 并发工具优先于wait和notify

Executor Framework 、并发集合和同步器

ConcurrentHashMap除了提供卓越的并发性之外，速度也非常快。除非不得已，否则应该优先使用ConcurrentHashMap，而不是使用Collections.synchronizedMap或者Hashtable。只要用并发Map替换老式的同步Map，就可以极大地提升并发应用程序的性能。

直接使用wait和notify就像用“并发汇编语言”进行编程一样，而concurrent则提供了更高级的语言。

1. 线程安全性的文档化

每个类都应该使用说明或者线程安全注解，清楚地在文档中说明它的线程安全属性。

1. 慎用延迟初始化

延迟初始化降低了初始化类或者创建实例的开销，却增加了访问被延迟初始化的域的开销。

1. 避免使用线程组

线程组并没有提供太多有用的功能，而且它们提供的许多功能有缺陷的。

## 序列化

对象序列化，用来将对象编码成字节流，并从字节流编码中重新构建对象。

将一个对象编码成一个字节流，叫做将该对象序列化；相反的过程叫做反序列化。

1. 谨慎实现Serializable接口

实现Serializable接口付出的代价是：

a、一旦一个类被发布，就大大降低了“改变这个类的实现”的灵活性。改变这个类的内部，不导致序列化形式不兼容。

每个序列化的类都有一个唯一标识号，如果没有在一个名为serialVersionUID的私有静态final的long域中显示地指定该标识号，系统会根据这个类来调用一个复杂的运算过程。如果改变了类的信息，自动产生的序列化UID也会发生变化，因此，如果没有声明显示的序列版本UID，兼容性将会遭到破坏，再运行时导致InvalidClassException异常。

b、增加了Bug和安全漏洞的可能性

序列化机制是一种语言以外的对象创建机制。反序列化是一个隐藏的构造器。依靠默认的反序列化机制，很容易使对象的约束关系遭到破坏，以及遭受到非法访问。

c、随着类发行新的版本，相关的测试负担也增加了。

序列化的类被修订时，要检查是否可以在新版本中序列化一个实例，然后在旧版本中反序列化。

序列化,压缩存储，节约内存