1 [声明和初始化(DCL)](https://www.cnblogs.com/flydean/p/13620709.html)

**1.1防止类的循环初始化**

初始化顺序

根据JLS（Java Language Specification）中的定义，class在初始化过程中，需要同时初始化class中定义的静态初始化程序和在该类中声明的静态字段（类变量）的初始化程序。

而对于static变量来说，如果static变量被定义为final并且它值是编译时常量值，那么该static变量将会被优先初始化。

那么使用了final static变量，是不是就没有初始化问题了呢？

我们来看下面一个例子：

public class StaticFiledOrder {

private final int result;

private static final StaticFiledOrder instance = new StaticFiledOrder();

private static final int intValue=100;

public StaticFiledOrder(){

result= intValue - 10;

}

public static void main(String[] args) {

System.out.println(instance.result);

}

}

输出结果是什么呢？

答案是90。 根据我们提到的规则，intValue是final并且被编译时常量赋值，所以是最先被初始化的，instance调用了StaticFiledOrder类的构造函数，最终导致result的值是90。

接下来，我们换个写法，将intValue改为随机变量：

public class StaticFiledOrder {

private final int result;

private static final StaticFiledOrder instance = new StaticFiledOrder();

private static final int intValue=(int)Math.random()\* 1000;

public StaticFiledOrder(){

result= intValue - 10;

}

public static void main(String[] args) {

System.out.println(instance.result);

}

}

运行结果是什么呢？

答案是-10。为什么呢？

因为instance在调用StaticFiledOrder构造函数进行初始化的过程中，intValue还没有被初始化，所以它有一个默认的值0，从而导致result的最终值是-10。

怎么修改呢？

将顺序调换一下就行了：

public class StaticFiledOrder {

private final int result;

private static final int intValue=(int)Math.random()\* 1000;

private static final StaticFiledOrder instance = new StaticFiledOrder();

public StaticFiledOrder(){

result= intValue - 10;

}

public static void main(String[] args) {

System.out.println(instance.result);

}

}

循环初始化

既然static变量可以调用构造函数，那么可不可以调用其他类的方法呢？

看下这个例子：

public class CycleClassA {

public static final int a = CycleClassB.b+1;

}

public class CycleClassB {

public static final int b = CycleClassA.a+1;

}

上面就是一个循环初始化的例子，上面的例子中CycleClassA中的a引用了CycleClassB的b，而同样的CycleClassB中的b引用了CycleClassA的a。

这样循环引用虽然不会报错，但是根据class的初始化顺序不同，会导致a和b生成两种不同的结果。

所以在我们编写代码的过程中，一定要避免这种循环初始化的情况。

**1.2不要重用Java标准库的已经公开的标识**

不要使用java标准库中的类名作为自己的类名

java标准库中为我们定义了很多非常优秀的类，我们在搭建自己的java程序时候可以很方便的使用。

但是我们在写自定义类的情况下，一定要注意避免使用和java标准库中一样的名字。

这个应该很好理解，就是为了避免混淆。以免造成不必要的意外。

**1.3将所有增强for语句的循环变量声明为final类型**

不要在增强的for语句中修改变量值

我们在遍历集合和数组的过程中，除了最原始的for语句之外，java还为我们提供了下面的增强的for循环：

for (I #i = Expression.iterator(); #i.hasNext(); ) {

{VariableModifier} TargetType Identifier =

(TargetType) #i.next();

Statement

}

在遍历的过程中，#i其实相当于一个本地变量，对这个本地变量的修改是不会影响到集合本身的。

我们看一个例子：

public void noncompliantUsage(){

int[] intArray = new int[]{1,2,3,4,5,6};

for(int i: intArray){

i=0;

}

for(int i: intArray){

System.out.println(i);

}

}

我们在遍历过程中，尝试将i都设置为0，但是最后输出intArray的结果，发现没有任何变化。

所以，一般来说我们需要在增强的for语句中，将#i设置成为final，从而消除这种不必要的逻辑误会。

public void compliantUsage(){

int[] intArray = new int[]{1,2,3,4,5,6};

for(final int i: intArray){

}

for(int i: intArray){

System.out.println(i);

}

}

2 [数值类型与运算(NUM)](https://www.cnblogs.com/flydean/p/13644102.html)

**2.1检测和避免整数溢出**

**2.2不要对同一数据进行位运算和数学运算**

**2.3确保除法运算和模运算中的除数不为0**

**2.4使用可容纳无符号数据合法取值范围的整数类型**

**2.5不要使用浮点数进行精细计算**

**2.6不要使用非标准化数**

**2.7使用strict fp修饰符确保跨平台浮点运算的一致性**

**2.8不要尝试与NaN进行比较**

**2.9检查浮点输入特殊的数值**

**2.10不要使用浮点变量作为循环计数器**

**2.11不要从浮点字元构造BigDecimal对象**

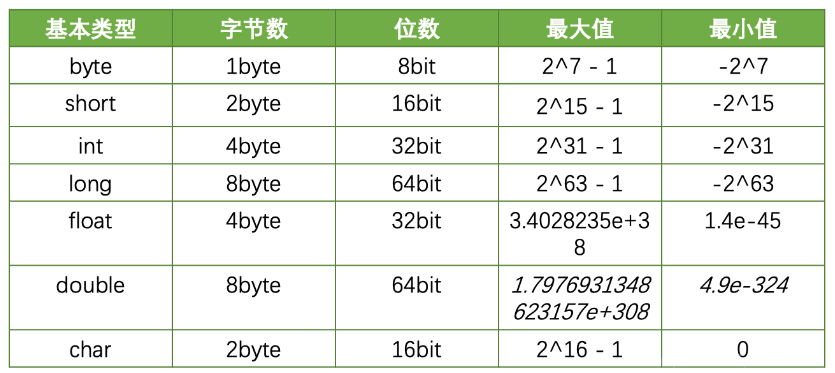
**2.12不要比较或者审查以字符串表达的浮点数值**

**2.13确保将数值转换成较小类型时不会产生数据丢失或曲解**

**2.14转换基本整数类型至浮点类型时应避免精度损失**

Number的范围

每种Number类型都有它的范围，我们看下java中Number类型的范围：



考虑到我们最常用的int操作，虽然int的范围够大，但是如果我们在做一些int操作的时候还是可能超出int的范围。

超出了int范围会发送什么事情呢？看下面的例子：

public void testIntegerOverflow(){

System.out.println(Integer.MAX\_VALUE+1000);

}

运行结果：-2147482649。

很明显Integer.MAX\_VALUE+1000将会超出Integer的最大值范围，但是我们没有得到异常提醒，反而得到了一个错误的结果。

正确的操作是如果我们遇到了Overflow的问题，需要抛出异常：ArithmeticException。

怎么防止这种IntegerOverflow的问题呢？一般来讲，我们有下面几种方式。

第一种方式：在做Integer操作之前，进行预判断是否超出范围：

举个例子：

static final int safeAdd(int left, int right) {

if (right > 0 ? left > Integer.MAX\_VALUE - right

: left < Integer.MIN\_VALUE - right) {

throw new ArithmeticException("Integer overflow");

}

return left + right;

}

上面的例子中，我们需要进行两个整数相加操作，在相加之前，我们需要进行范围的判断，从而保证计算的安全性。

第二种方式：使用Math的addExact和multiplyExact方法：

Math的addExact和multiplyExact方法已经提供了Overflow的判断，我们看下addExact的实现：

public static int addExact(int x, int y) {

int r = x + y;

// HD 2-12 Overflow iff both arguments have the opposite sign of the result

if (((x ^ r) & (y ^ r)) < 0) {

throw new ArithmeticException("integer overflow");

}

return r;

}

看下怎么使用：

public int addUseMath(int a, int b){

return Math.addExact(a,b);

}

第三种方式：向上转型

既然超出了Integer的范围，那么我们可以用范围更大的long来存储数据。

public static long intRangeCheck(long value) {

if ((value < Integer.MIN\_VALUE) || (value > Integer.MAX\_VALUE)) {

throw new ArithmeticException("Integer overflow");

}

return value;

}

public int addUseUpcasting(int a, int b){

return (int)intRangeCheck((long)a+(long)b);

}

上面的例子中，我们将a+b转换成了两个long相加，从而保证不溢出范围。

然后进行一次范围比较，从而判断相加之后的结果是否仍然在整数范围内。

第四种方式：使用BigInteger

我们可以使用BigInteger.valueOf(a)将int转换成为BigInteger，再进行后续操作：

public int useBigInteger(int a, int b){

return BigInteger.valueOf(a).add(BigInteger.valueOf(b)).intValue();

}

区分位运算和算数运算

我们通常会对Integer进行位运算或者算数运算。虽然可以进行两种运算，但是最好不要将两种运算同时进行，这样会造成混淆。

比如下面的例子：

x += (x << 1) + 1;

上面的例子是想做什么呢？其实它是想将3x+1的值赋给x。

但是这样写出来让人很难理解，所以我们需要避免这样实现。

再看下面的一个例子：

public void testBitwiseOperation(){

int i = -10;

System.out.println(i>>>2);

System.out.println(i>>2);

System.out.println(i/4);

}

本来我们想做的是将i除以4，结果发现只有最后一个才是我们要的结果。

我们来解释一下，第一个i>>>2是逻辑右移，将会把最左边的填充成0,所以得出的结果是一个正值1073741821。

第二个i>>2是算数右移，最左边的还是会填充成1，但是会向下取整，所以得出结果是-3.

直接使用i/4，我们是向上取整，所以得出结果是-2.

注意不要使用0作为除数

我们在使用变量作为除数的时候，一定要注意先判断是否为0.

兼容C++的无符号整数类型

在java中只有16位的char表示的是无符号整数，而int实际上表示的是带符号的整数。

而在C或者C++中是可以直接表示无符号的整数的，那么，如果我们有一个32位的无符号整数，该怎么用java来处理呢？

public int readIntWrong(DataInputStream is) throws IOException {

return is.readInt();

}

看上面的例子，我们从Stream中读取一个int值，如果是一个32位的无符号整数，那么读出来int就变成了有符号的负整数，这和我们的期望是相符的。

考虑一下，long是64位的，我们是不是可以使用long来表示32位的无符号整数呢？

public long readIntRight(DataInputStream is) throws IOException{

return is.readInt() & 0xFFFFFFFFL; // Mask with 32 one-bits

}

看上面的例子，我们返回的是long，如果将32位的int转换成为64位的long，会自动根据符号位进行补全。

所以这时候我们需要和0xFFFFFFFFL进行mask操作，将高32位重置为0.

NAN和INFINITY

在整型运算中，除数是不能为0的，否则直接运行异常。但是在浮点数运算中，引入了NAN和INFINITY的概念，我们来看一下Double和Float中的定义。

public static final double POSITIVE\_INFINITY = 1.0 / 0.0;

public static final double NEGATIVE\_INFINITY = -1.0 / 0.0;

public static final double NaN = 0.0d / 0.0;

public static final float POSITIVE\_INFINITY = 1.0f / 0.0f;

public static final float NEGATIVE\_INFINITY = -1.0f / 0.0f;

public static final float NaN = 0.0f / 0.0f;

1除以0就是INFINITY，而0除以0就是NaN。

接下来，我们看一下NAN和INFINITY的比较：

public void compareInfinity(){

System.out.println(Double.POSITIVE\_INFINITY == Double.POSITIVE\_INFINITY);

}

运行结果是true。

public void compareNaN(){

System.out.println(Double.NaN == Double.NaN);

}

运行结果是false。

可以看到NaN和NaN相比是false。

那么我们怎么比较NaN呢？

别急，Double提供了一个isNaN方法，我们可以这样使用：

System.out.println(Double.isNaN(Double.NaN));

接下来我们看一个在代码中经常会用到的一个Double解析：

public void incorrectParse(String userInput){

double val = 0;

try {

val = Double.valueOf(userInput);

} catch (NumberFormatException e) {

}

//do something for val

}

这段代码有没有问题？咋看下好像没有问题，但是，如果我们的userInput是NaN，Infinity，或者-Infinity，Double.valueOf是可以解析得到结果的。

public void testNaN(){

System.out.println(Double.valueOf("NaN"));

System.out.println(Double.valueOf("Infinity"));

System.out.println(Double.valueOf("-Infinity"));

}

运行输出：

NaN

Infinity

-Infinity

所以，我们还需要额外去判断NaN和Infinity：

public void correctParse(String userInput){

double val = 0;

try {

val = Double.valueOf(userInput);

} catch (NumberFormatException e) {

}

if (Double.isInfinite(val)){

// Handle infinity error

}

if (Double.isNaN(val)) {

// Handle NaN error

}

//do something for val

}

不要使用float或者double作为循环的计数器

考虑下面的代码：

for (float x = 0.1f; x <= 1.0f; x += 0.1f) {

System.out.println(x);

}

上面的代码有什么问题呢？

我们都知道java中浮点数是不准确的，但是不一定有人知道为什么不准确。

这里给大家解释一下，计算机中所有与的数都是以二进制存储的，我们以0.6为例。

0.6转成为二进制格式是乘2取整，0.6x2=1.2，取整剩余0.2，继续上面的步骤0.2x2=0.4，0.4x2=0.8,0.8x2=1.6,取整剩余0.6，产生了一个循环。

所以0.6的二进制格式是.1001 1001 1001 1001 1001 1001 1001 ... 无限循环下去。

所以，有些小数是无法用二进制精确的表示的，最终导致使用float或者double作为计数器是不准的。

BigDecimal的构建

为了解决float或者Double计算中精度缺失的问题，我们通常会使用BigDecimal。

那么在使用BigDecimal的时候，请注意一定不要从float构建BigDecimal，否则可能出现意想不到的问题。

public void getFromFloat(){

System.out.println(new BigDecimal(0.1));

}

上面的代码，我们得到的结果是：0.1000000000000000055511151231257827021181583404541015625。

这是因为二进制无法完美的展示所有的小数。

所以，我们需要从String来构建BigDecimal：

public void getFromString(){

System.out.println(new BigDecimal("0.1"));

}

类型转换问题

在java中各种类型的Number可以互相进行转换：

比如：

short to byte or char

char to byte or short

int to byte, short, or char

long to byte, short, char, or int

float to byte, short, char, int, or long

double to byte, short, char, int, long, or float

或者反向：

byte to short, int, long, float, or double

short to int, long, float, or double

char to int, long, float, or double

int to long, float, or double

long to float or double

float to double

从大范围的类型转向小范围的类型时，我们要考虑是否超出转换类型范围的情况：

public void intToByte(int i){

if ((i < Byte.MIN\_VALUE) || (i > Byte.MAX\_VALUE)) {

throw new ArithmeticException("Value is out of range");

}

byte b = (byte) i;

}

比如上面的例子中，我们将int转换成为byte，那么在转换之前，需要先判断int是否超出了byte的范围。

同时我们还需要考虑到精度的切换，看下面的例子：

public void intToFloat(){

System.out.println(subtraction(1111111111,1111111111));

}

public int subtraction(int i , float j){

return i - (int)j;

}

结果是多少呢？

答案不是0，而是-57。

为什么呢？

因为这里我们做了两次转换，第一次从1111111111转换到float，float虽然有32位，但是只有23位是存放真正的数值的，1位是符号位，剩下的8位是指数位。

所以从1111111111转换到float发送了精度丢失。

我们可以把subtraction方法修改一下，首先判断float的范围，如果超出了23bit的表示范围，则说明发送了精度丢失，我们需要抛出异常：

public int subtraction(int i , float j){

System.out.println(j);

if ((j > 0x007fffff) || (j < -0x800000)) {

throw new ArithmeticException("Insufficient precision");

}

return i - (int)j;

}

当然还有一种办法，我们可以用精度更高的double来做转换，double有52位来存放真正的数据，所以足够了。

public int subtractionWithDouble(int i , double j){

System.out.println(j);

return i - (int)j;

}

3 [表达式(EXP)](https://www.jb51.net/article/195582.htm)

http://www.flydean.com/java-security-code-line-expresion/

3.1不要忽略方法的返回值

3.2不要解引用空指针

3.3使用两个参数的Arrays.equals() 方法来比较两个数组的内容

3.4不要用相等操作符来比较两个基础数据类型的值

3.5确保使用正确的类型来自动封装数值

3.6不要在一个表达式中对同一变量进行多次写人

3.7不要在断言中使用有副作用的表达式

注意表达式的返回值

我们在使用JDK库的时候，一定要注意认真的读一下JDK中方法的含义和它的返回值。

有些返回值可能表示这个操作是否成功，有的返回值可能是方法操作的结果。我们看两个常见的例子：

public void deleteFileWrong(){

File file= new File("/tmp/www.flydean.com.txt");

file.delete();

System.out.println("File delete success!");

}

public void deleteFileRight(){

File file= new File("/tmp/www.flydean.com.txt");

if(file.delete()){

System.out.println("File delete success!");

}

}

先看一个文件删除的例子，delete方法是有返回值的，所以我们在调用delete方法之后，一定要判断一下返回值，看是否删除成功。

再看一个常见的String中字符替换的例子：

public void stringReplaceWrong(){

String url="www.flydean.com";

url.replace("www","WWW");

System.out.println("replaced url..."+url);

}

public void stringReplaceRight(){

String url="www.flydean.com";

url=url.replace("www","WWW");

System.out.println("replaced url..."+url);

}

我们要记住，String是不可变的，所以它的replace方法，会返回一个替换过后的String，但是原String是不变的，所以我们需要将返回值重新赋值。

注意避免NullPointerException

NullPointerException应该是最最常见的运行时异常了。怎么避免这个异常呢？

我们要做的就是在调用object的方法时候，一定要判断这个object是不是为空。

在JDK8之后，我们引入了Stream操作：

public void streamWrong(Collection<Object> collection){

collection.stream().filter(obj->obj.equals("www.flydean.com"));

}

Stream操作的过程中，我们需要注意stream中的元素是否为空。

有时候，我们可能觉得已经判断是为空了，但是条件判断不准确，导致未知的异常，看下面这个例子：

public int countWrong(Collection<Object> collection, Object object){

int count=0;

if(collection ==null){

return count;

}

for(Object element: collection){

if((element ==null && object== null)

|| element.equals(object)){

count++;

}

}

return count;

}

这个例子是用来查找collection中到底有多少元素和object相同，如果两者都为空，也记为相同。

但是上面的例子有一个漏洞，它没有考虑element ==null 而 object ！=null的情况，所以会导致NullPointerException的生成。

我们需要这样修改：

public int countRight(Collection<Object> collection, Object object){

int count=0;

if(collection ==null){

return count;

}

for(Object element: collection){

if((element ==null && object== null)

|| (element !=null && element.equals(object))){

count++;

}

}

return count;

}

数组相等的判断

如果我们需要比较两个数组是否相等，其实我们想比较的是两个数组中的元素是否相等。

我们知道数组是一个特殊的Object，那么数组对象也有一个equals方法，考虑下面的例子：

public boolean compareWrong(){

int[] array1 = new int[10];

int[] array2 = new int[10];

return array1.equals(array2);

}

返回的结果是false，因为数组直接使用了Object中定义的equals方法，我们看下该方法的定义：

public boolean equals(Object obj) {

return (this == obj);

}

可以看到，该方法比较的是两个地址是否相等。所以我们的到了false结果。

其实，我们可以使用Arrays.equals工具类中的方法来进行两个数组的比较：

public boolean compareRight(){

int[] array1 = new int[10];

int[] array2 = new int[10];

return Arrays.equals(array1, array2);

}

基础类型的封装类间的比较

在java中，我们知道有一些基础类型像boolean, byte，char, short, int他们会有相对应的封装类型：Boolean，Byte，Character，Short，Integer等。

我们可以直接将基础类型的值赋值给封装类型，封装类型会自行进行转换。

考虑下面的例子：

Boolean boolA=true;

Boolean boolB=true;

System.out.println(boolA==boolB);

结果是多少呢？

答案是true。为什么两个不同对象的比较会是true呢？

在回答这个问题之前，我们看一下字符串的比较：

String stringA="www.flydean.com";

String stringB="www.flydean.com";

System.out.println(stringA==stringB);

这个我们大家应该都知道，因为String有一个字符串常量池，直接从字符串常量构建的String对象，其实是同一个对象。

同样的对于Boolean和Byte来说，如果直接从基础类值构建的话，也是同一个对象。

而对于Character来说，如果值的范围在\u0000 to \u007f，则属于同一个对象，如果超出了这个范围，则是不同的对象。

对于Integer和Short来说，如果值的范围在-128 and 127，则属于同一个对象，如果超出了这个范围，则是不同的对象。

再考虑下面的例子：

Boolean boolA=true;

Boolean boolC=new Boolean(true);

System.out.println(boolA==boolC);

输出的结果是false，因为boolC使用了new关键字，构建了一个新的对象。

集合中类型不匹配

现在java集合可以通过指定类型，从而只存储特定类型的对象。考虑下面的一个例子：

public void typeMismatch(){

HashSet<Short> shortSet= new HashSet<>();

for(int i=0;i<10;i++){

shortSet.add((short)i);

shortSet.remove(i);

}

System.out.println(shortSet.size());

}

上面代码我们定义了一个Short的集合，然后将0-9添加进去，接着我们又调用了remove方法把i从集合删除。

但是最后输出结果是10，表明我们并没有删除成功。为什么呢？

看下HashSet的remove方法：

public boolean remove(Object o) {

return map.remove(o)==PRESENT;

}

remove方法的参数是Object，我们传入的i是int类型的，跟short不匹配，所以导致删除失败。

我们需要这样修改：

public void typeMatch(){

HashSet<Short> shortSet= new HashSet<>();

for(int i=0;i<10;i++){

shortSet.add((short)i);

shortSet.remove((short)i);

}

System.out.println(shortSet.size());

}

Asset的副作用

我们会使用Asset语句在代码中做调试使用，在使用的过程中需要注意Asset语句不要对系统的业务逻辑产生副作用，也就是说即使Asset语句不运行，也不会修改代码的业务逻辑。

看下面的例子：

public void assetWrong(ArrayList<Integer> list){

assert list.remove(0)>0;

}

上的代码我们从list中删除第一个元素，并判断删除的元素是否大于0.

上面的代码如果assert语句不执行的话，会导致业务逻辑也不执行，所以需要修改成下面这样：

public void assetRight(ArrayList<Integer> list){

int result=list.remove(0);

assert result>0;

}

4 控制语句

https://blog.csdn.net/goandozhf/article/details/109991258

https://blog.csdn.net/cangely/article/details/79838474

https://www.jianshu.com/p/b9886f0ac26d

4.1**在一个switch块内，每个case要么通过break/return等来终止，要么注释说明程序将继续执行到哪一个case为止；在一个switch块内，都必须包含一个default语句并且放在最后，即使空代码。**

4.2**在if/else/for/while/do语句中必须使用大括号。即使只有一行代码，避免采用单行的编码方式：if (condition) statements;**

4.3**在高并发场景中，避免使用 ”等于 ”判断作为中或退出的条件。**

说明： 如果并发控制没有处理好，“击穿”的情况，使用大于或小于区间判断条件来代替。

反例：判断剩余奖品数量等于0时，终止发放奖品，但因为并处理错误导致数量瞬间变成了负数， 这样的话活动无法终止。

4.4**表达异常的分支时，少用if-else方式。**

推荐方式:

if (condition) {

...

return obj;

}

// 接着写else的业务逻辑代码;

说明：如果非得使用if()…else if()…else…方式表达逻辑，[强制] 避免后续代码维护困难，请勿超过3层。

正例: 超过3层的 if-else 的逻辑判断代码可以使用卫语句、策略模式、状态模式等来实现，其中卫语句示例如下：

public void today() {

if (isBusy()) {

System.out.println(“change time.”);

return;

}

if (isFree()) {

System.out.println(“go to travel.”);

return;

}

System.out.println(“stay at home to learn Alibaba Java Coding Guidelines.”);

return;

}

4.5**除常用方法（如getXxx/isXxx）等外，不要在条件判断中执行其它复杂的语句，将复杂逻辑判断的结果赋值给一个有意义的布尔变量名，以提高可读性。**

5 [方法(MET)](https://www.cnblogs.com/flydean/p/13780288.html)

**5.1验证方法参数**

**5.2不要使用断言验证方法参数**

**5.3不要使用弃用的或过时的类和方法**

**5.4进行安全检测的方法必须声明为private或final**

**5.5不要增加被覆写方法和被隐藏方法的可访问性**

**5.6确保构造函数不会调用可覆写的方法**

**5.7不要在clone() 中调用可覆写的方法**

**5.8不要定义类方法来隐藏基类或基类接口中声明的方法**

**5.9确保比较等同的对象能得到相等的结果**

**5.10定义了equ las() 方法的类必须定义hashCode() 方法**

**5.11实现compare To() 方法时遵守常规合约**

**5.12确保比较中的关键码是不可变的**

**5.13不要使用析构函数**

不要在构造函数中调用可以被重写的方法

一般来说在构造函数中只能调用static，final或者private的方法。为什么呢？

如果父类在执行构造函数的时候调用了一个可以被重写的方法，那么在该方法中可能会使用到未初始化的数据，从而导致运行时异常或者意外结束。

另外，还可能到方法获取到未初始化完毕的实例，从而导致数据不一致性。

举个例子，我们定义了一个Person的父类：

public class Person {

public void printValue(){

System.out.println("this is person!");

}

public Person(){

printValue();

}

}

然后定义了一个Boy的子类，但是在Boy子类中，重新了父类的printValue方法。

public class Boy extends Person{

public void printValue(){

System.out.println("this is Boy!");

}

public Boy(){

super();

}

public static void main(String[] args) {

Person persion= new Person();

Boy boy= new Boy();

}

}

输出结果：

this is person!

this is Boy!

可以看到Boy调用了自己重写过的printValue方法。

注意，这里并不是说会产生语法错误，而是这样会导致业务逻辑看起来非常混乱。

怎么解决呢？简单办法就是将Person中的printValue置位final即可。

不要在clone()方法中调用可重写的方法

同样的，我们在定义clone方法的时候也不要调用可重写的方法，否则也会产生意想不到的变化。

还是上面的例子，这次我们添加了clone方法到Person类：

public Object clone() throws CloneNotSupportedException {

Person person= (Person)super.clone();

person.printValue();

return person;

}

接下来我们添加clone方法到Boy类：

public Object clone() throws CloneNotSupportedException {

Boy clone = (Boy) super.clone();

clone.printValue();

return clone;

}

因为在clone方法中调用了可重写的方法，从而让系统逻辑变得混乱。不推荐这样使用。

重写equals()方法

考虑一下父类和子类的情况，如果在父类中我们定义了一个equals方法，这个方法是根据父类中的字段来进行比较判断，最终决定两个对象是否相等。

如果子类添加了一些新的字段，如果不重写equals方法，而是使用父类的equals方法，那么就会遗漏子类中新添加的字段，最终导致equals返回意想不到的结果。

所以一般来说，子类需要重写equals方法。

如果重新equals方法，需要满足下面几个特性：

reflexive反射性

对于一个Object a来说，a.equals(a)必须成立。

symmetric对称性

对于一个Object a和Object b来说，如果a.equals(b)true，那么b.equals(a)true一定成立。

transitive传递性

对于Object a,b,c来说，如果a.equals(b)true,b.equals(c)true,那么a.equals(c)==true一定成立。

consistent一致性

对于Object a，b来说，如果a和b没有发生任何变化，那么a.equals(b)的结果也不能变。

对于非空的引用a，a.equals(null) 一定要等于false

具体代码的例子，这里就不写了，大家可以自行练习一下。

hashCode和equals

hashCode是Object中定义的一个native方法：

@HotSpotIntrinsicCandidate

public native int hashCode();

根据Oracle的建议，如果两个对象的equals方法返回的结果是true，那么这两个对象的hashCode一定要返回同样的int值。

为什么呢？

我们看下下面的一个例子：

public class Girl {

private final int age;

public Girl(int age) {

this.age = age;

}

@Override

public boolean equals(Object o) {

if (o == this) {

return true;

}

if (!(o instanceof Girl)) {

return false;

}

Girl cc = (Girl)o;

return cc.age == age;

}

public static void main(String[] args) {

HashMap<Girl,Integer> hashMap= new HashMap<>();

hashMap.put(new Girl(20), 20);

System.out.println(hashMap.get(new Girl(20)));

}

}

上面的Girl中，我们定义了equals方法，但是并没有重写hashCode，最后返回的结果是null。

因为我们new了两次Girl这个对象，最后导致native方法中两个不同对象的hashCode是不一样的。

我们可以给Girl类中添加一个hashCode方法：

public int hashCode() {

return age;

}

这样就可以返回正确的值。

compareTo方法的实现

我们在实现可比较类的时候，通常需要实现Comparable接口。Comparable接口定义了一个compareTo方法，用来进行两个对象的比较。

我们在实现compareTo方法的时候，要注意保证比较的通用规则，也就是说，如果x.compareTo(y) > 0 && y.compareTo(z) > 0 那么表示 x.compareTo(z) > 0.

所以，我们不能使用compareTo来实现特殊的逻辑。

最近看了一个日本的电影，叫做dubo默示录，里面有一集就是石头，剪刀，布来判断输赢。

当然，石头，剪刀，布不满足我们的通用compareTo方法，所以不能将逻辑定义在compareTo方法中。

http://wfscbw.cn/index.aspx?id=UDCJXKjTspc

声明 初始化 有

数据类型 有

表达式 有

控制语句

方法 有

http://www.flydean.com/security-coding-rule/