# 1.case中必须添加break

对于此类问题，还有一个最简单的解决办法：修改IDE的警告级别，例如在Eclipse中，可以依次点击Performaces→Java→Compiler→ErrorsWarnings→Potential Programming problems，然后修改‘switch’case fall-through为Errors级别，如果你胆敢不在case语句中加入break，那Eclipse直接就报个红叉给你看，这样就可以完全避免该问题的发生了。

# 2.用整数类型处理货币

在日常生活中，最容易接触到的小数就是货币，比如你付给售货员10元钱购买一个9.60元的零食，售货员应该找你0.4元也就是4毛钱才对，我们来看下面的程序：

public class Client{

public static void main（String[]args）{

System.out.println（10.00-9.60）；

}

}

我们期望的结果是0.4，也应该是这个数字，但是打印出来的却是0.40000000000000036，这是为什么呢？

这是因为在计算机中浮点数有可能（注意是可能）是不准确的，它只能无限接近准确值，而不能完全精确。为什么会如此呢？这是由浮点数的存储规则所决定的，我们先来看0.4这个十进制小数如何转换成二进制小数，使用“乘2取整，顺序排列”法（不懂？这就没招了，太基础了），我们发现0.4不能使用二进制准确的表示，在二进制数世界里它是一个无限循环的小数，也就是说，“展示”都不能“展示”，更别说是在内存中存储了（浮点数的存储包括三部分：符号位、指数位、尾数，具体不再介绍），可以这样理解，在十进制的世界里没有办法准确表示13，那在二进制世界里当然也无法准确表示15（如果二进制也有分数的话倒是可以表示），在二进制的世界里15是一个无限循环小数。

各位要说了，那我对结果取整不就对了吗？代码如下：

public class Client{

public static void main（String[]args）{

NumberFormat f=new DecimalFormat（#.##）；

System.out.println（f.format（10.00-9.60））；

}

}

打印出结果是0.4，看似解决了，但是隐藏了一个很深的问题。我们来思考一下金融行业的计算方法，会计系统一般记录小数点后的4位小数，但是在汇总、展现、报表中，则只记录小数点后的2位小数，如果使用浮点数来计算货币，想想看，在大批量的加减乘除后结果会有多大的差距（其中还涉及后面会讲到的四舍五入问题）！会计系统要的就是准确，但是却因为计算机的缘故不准确了，那真是罪过。要解决此问题有两种方法：

（1）使用BigDecimal

BigDecimal是专门为弥补浮点数无法精确计算的缺憾而设计的类，并且它本身也提供了加减乘除的常用数学算法。特别是与数据库Decimal类型的字段映射时，BigDecimal是最优的解决方案。

（2）使用整型

把参与运算的值扩大100倍，并转变为整型，然后在展现时再缩小100倍，这样处理的好处是计算简单、准确，一般在非金融行业（如零售行业）应用较多。此方法还会用于某些零售POS机，它们的输入和输出全部是整数，那运算就更简单。

# 3.建议23：不要让类型默默转换

我们出一个小学生的题目给大家做做看，光速是每秒30万公里，根据光线旅行的时间，计算月亮与地球、太阳与地球之间的距离。代码如下：

public class Client{

光速是30万公里秒，常量

public static final int LIGHT\_SPEED=30100001000；

public static void main（String[]args）{

System.out.println（题目1：月亮光照射到地球需要1秒，计算月亮和地球的距离。）；

long dis1=LIGHT\_SPEED1；

System.out.println（月亮与地球的距离是：+dis1+米）；

System.out.println（--------------------------------------------）；

System.out.println（题目2：太阳光照射到地球上需要8分钟，计算太阳到地球的距离。）；

可能要超出整数范围，使用long型

long dis2=LIGHT\_SPEED608；

System.out.println（太阳与地球的距离是：+dis2+米）；

}

}

估计你要鄙视了，这种小学生乘法计算有什么可做的。不错，确实就是一个乘法运算，我们运行一下看看结果：

题目1：月亮光照射到地球需要1秒，计算月亮和地球的距离。

月亮与地球的距离是：300000000米

--------------------------------------------

题目2：太阳光照射到地球上需要8分钟，计算太阳到地球的距离。

太阳与地球的距离是：-2028888064米

太阳和地球的距离竟然是负的，诡异。dis2不是已经考虑到int类型可能越界的问题，并使用了long型吗，为什么还会出现负值呢？

那是因为Java是先运算然后再进行类型转换的，具体地说就是因为disc2的三个运算参数都是int类型，三者相乘的结果虽然也是int类型，但是已经超过了int的最大值，所以其值就是负值了（为什么是负值？因为过界了就会从头开始），再转换成long型，结果还是负值。

问题知道了，解决起来也很简单，只要加个小小的“L”即可，代码如下：

long dis2=LIGHT\_SPEED60L8；

60L是一个长整型，乘出来的结果也是一个长整型（此乃Java的基本转换规则，向数据范围大的方向转换，也就是加宽类型），在还没有超过int类型的范围时就已经转换为long型了，彻底解决了越界问题。在实际开发中，更通用的做法是主动声明式类型转化（注意不是强制类型转换），代码如下：

long dis2=1LLIGHT\_SPEED608；

既然期望的结果是long型，那就让第一个参与运算的参数也是long型（1L）吧，也就是明说“嗨，我已经是长整型了，你们都跟着我一起转为长整型吧”。

注意　基本类型转换时，使用主动声明方式减少不必要的Bug。

# 4.反序列化时构造函数不会执行。

总结一下，反序列化时final变量在以下情况下不会被重新赋值：

通过构造函数为final变量赋值。

通过方法返回值为final变量赋值。

final修饰的属性不是基本类型。

# 5.instanceof 关键字

## 5.1'A'instanceof Character

这句话可能有读者会猜错，事实上它编译不通过，为什么呢？因为'A'是一个char类型，也就是一个基本类型，不是一个对象，instanceof只能用于对象的判断，不能用于基本类型的判断。

## 5.2null instanceof String

返回值是false，这是instanceof特有的规则：若左操作数是null，结果就直接返回false，不再运算右操作数是什么类。

## 5.3new Date（）instanceof String

编译通不过，因为Date类和String没有继承或实现关系，所以在编译时直接就报错了，instanceof操作符的左右操作数必须有继承或实现关系，否则编译会失败。

# 6.不要只替换一个类(class文件)。

因为替换的是常量类时，其他引用的类中的常量不会重新编译，还是旧的值。

# 7.货币的四舍五入

使用RoundingMode类中的HALF\_EVEN