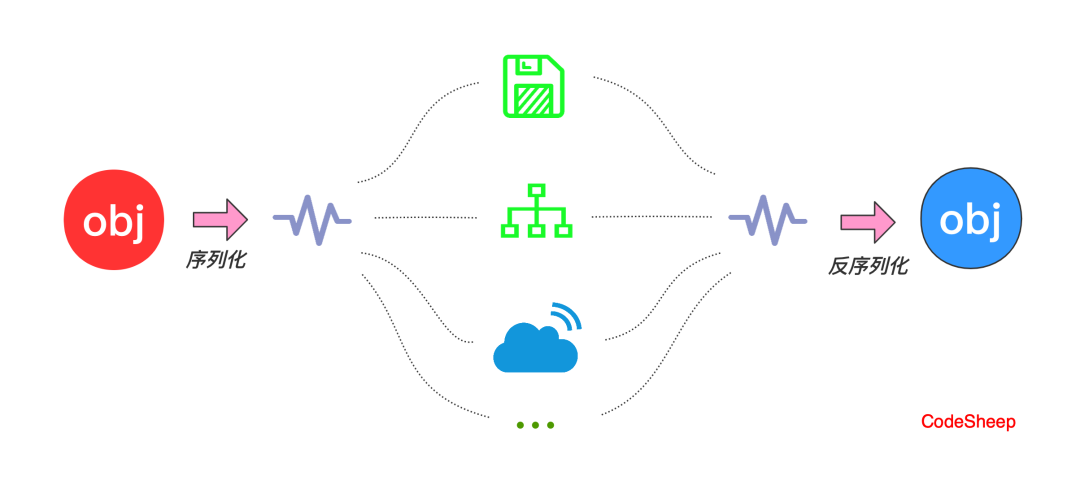
<https://mp.weixin.qq.com/s/0EfIUB9E-0Oh_Clwuxswuw>

# 序列化和反序列化

# 序列化是干啥用的？

序列化的原本意图是希望对一个Java对象作一下“变换”，变成字节序列，这样一来方便持久化存储到磁盘，避免程序运行结束后对象就从内存里消失，另外变换成字节序列也更便于网络运输和传播，所以概念上很好理解：

* **序列化**：把Java对象转换为字节序列。
* **反序列化**：把字节序列恢复为原先的Java对象。



而且序列化机制从某种意义上来说也弥补了平台化的一些差异，毕竟转换后的字节流可以在其他平台上进行反序列化来恢复对象。

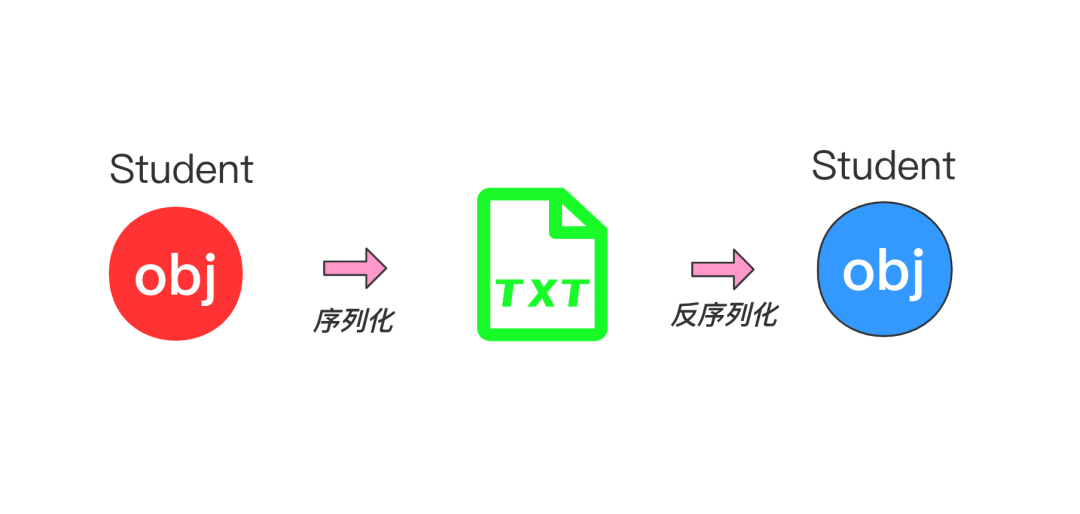
事情就是那么个事情，看起来很简单，不过后面的东西还不少，请往下看。

# 对象如何序列化？

然而Java目前并没有一个关键字可以直接去定义一个所谓的“可持久化”对象。

对象的持久化和反持久化需要靠程序员在代码里手动**显式地**进行序列化和反序列化还原的动作。

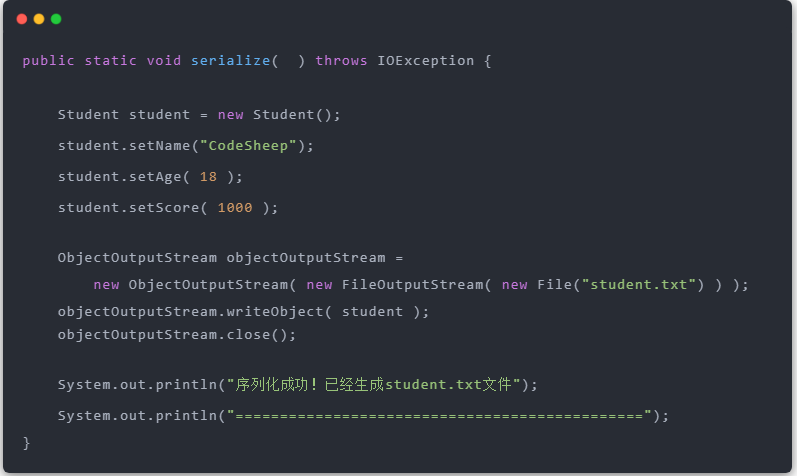
举个例子，假如我们要对Student类对象序列化到一个名为student.txt的文本文件中，然后再通过文本文件反序列化成Student类对象：



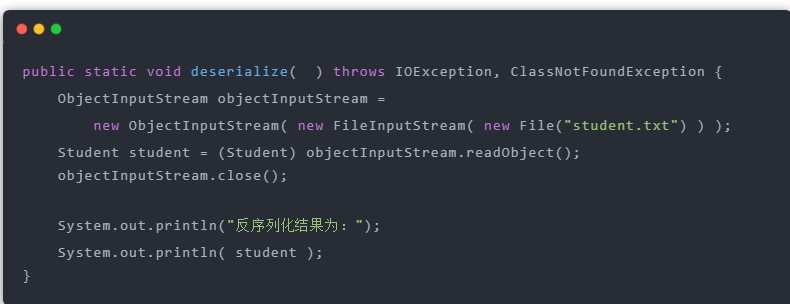
1. Student类定义



1. 序列化

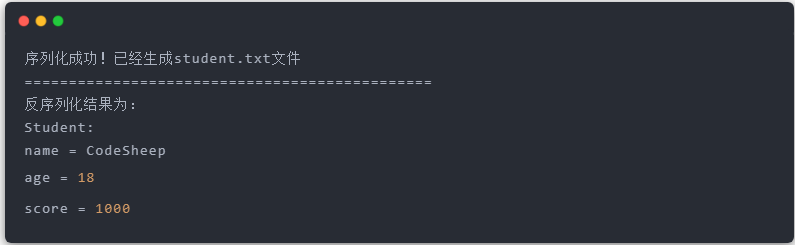


1. 反序列化



4、运行结果

控制台打印：



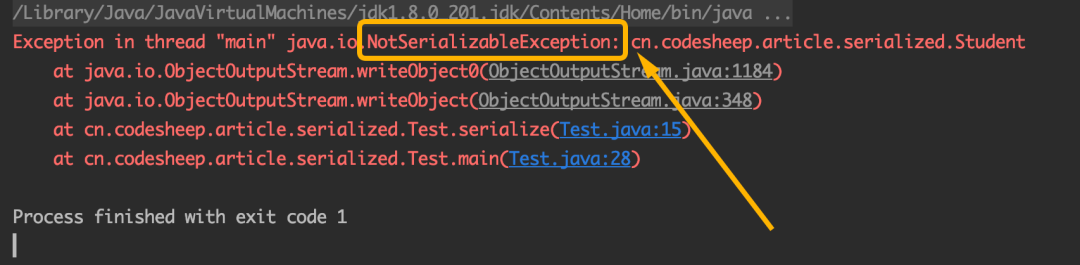
# Serializable接口有何用？

上面在定义Student类时，实现了一个Serializable接口，然而当我们点进Serializable接口内部查看，发现它**竟然是一个空接口**，并没有包含任何方法！

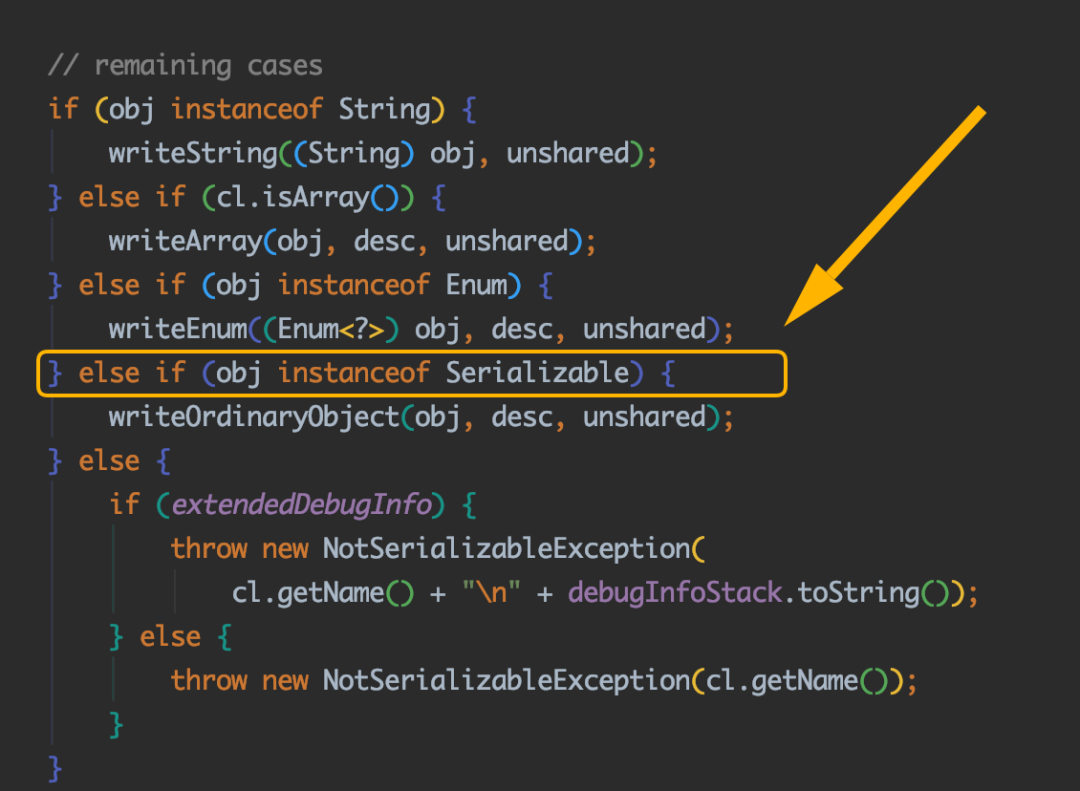


试想，如果上面在定义Student类时忘了加implements Serializable时会发生什么呢？

实验结果是：此时的程序运行**会报错**，并抛出NotSerializableException异常：



我们按照错误提示，由源码一直跟到ObjectOutputStream的writeObject0()方法底层一看，才恍然大悟：



如果一个对象既不是**字符串**、**数组**、**枚举**，而且也没有实现Serializable接口的话，在序列化时就会抛出NotSerializableException异常！

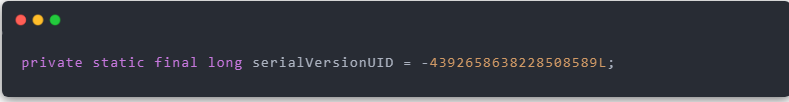
哦，我明白了！

原来Serializable接口也仅仅只是做一个标记用！！！

它告诉代码只要是实现了Serializable接口的类都是可以被序列化的！然而真正的序列化动作不需要靠它完成。

# serialVersionUID号有何用？

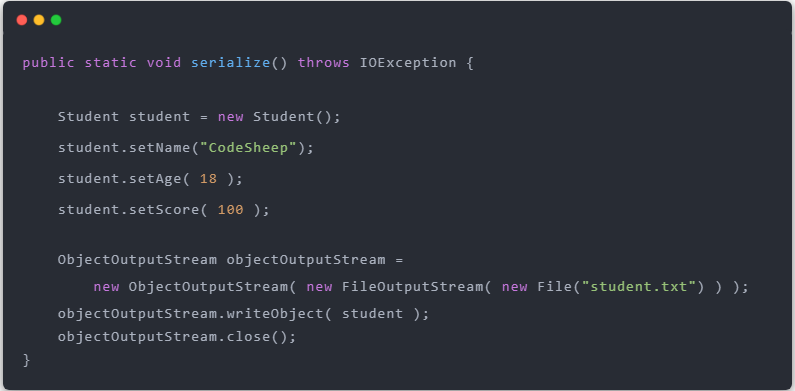
相信你一定经常看到有些类中定义了如下代码行，即定义了一个名为serialVersionUID的字段：



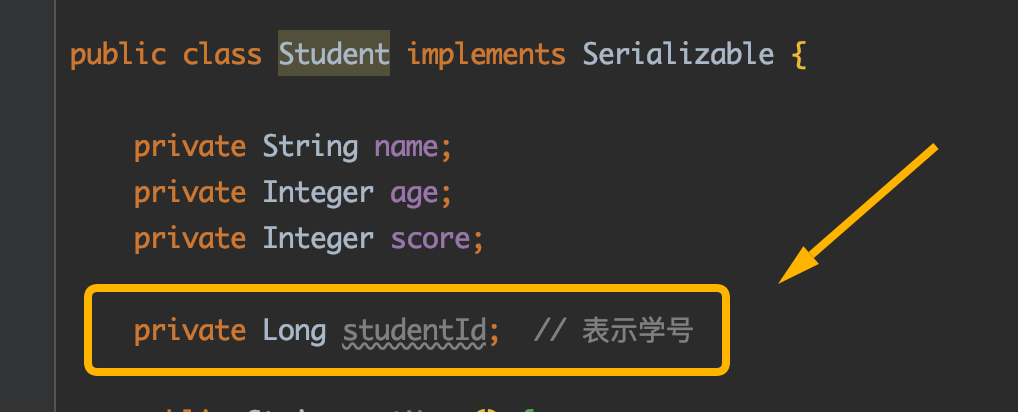
**你知道这句声明的含义吗？为什么要搞一个名为serialVersionUID的序列号？**

继续来做一个简单实验，还拿上面的Student类为例，我们并没有人为在里面显式地声明一个serialVersionUID字段。

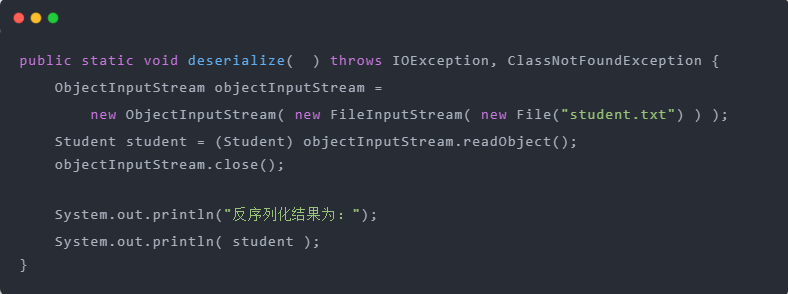
我们首先还是调用上面的serialize()方法，将一个Student对象序列化到本地磁盘上的student.txt文件：



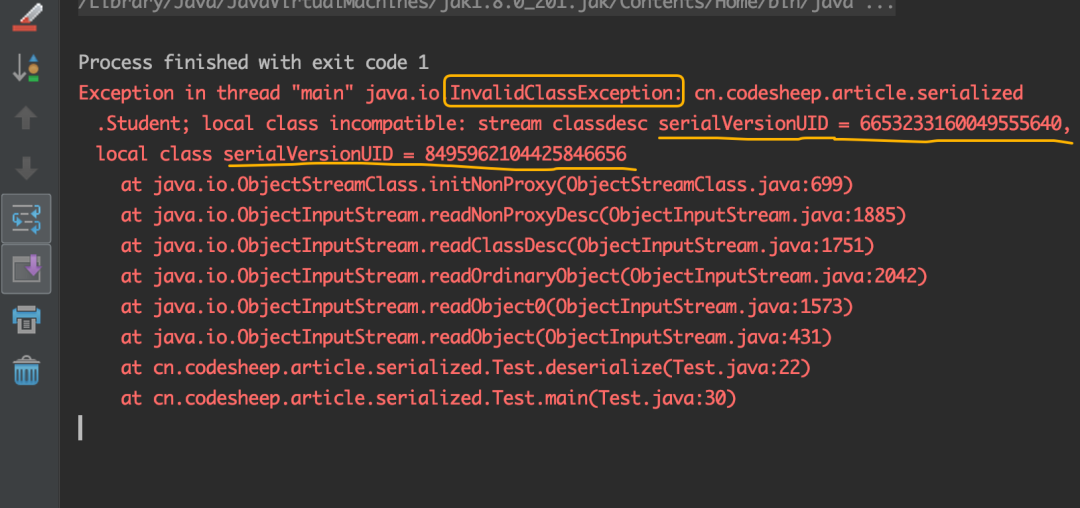
接下来我们在Student类里面动点手脚，比如在里面再增加一个名为studentID的字段，表示学生学号：



这时候，我们拿刚才已经序列化到本地的student.txt文件，还用如下代码进行反序列化，试图还原出刚才那个Student对象：



运行发现**报错了**，并且抛出了InvalidClassException异常：



这地方提示的信息非常明确了：序列化前后的serialVersionUID号码不兼容！

从这地方最起码可以得出**两个**重要信息：

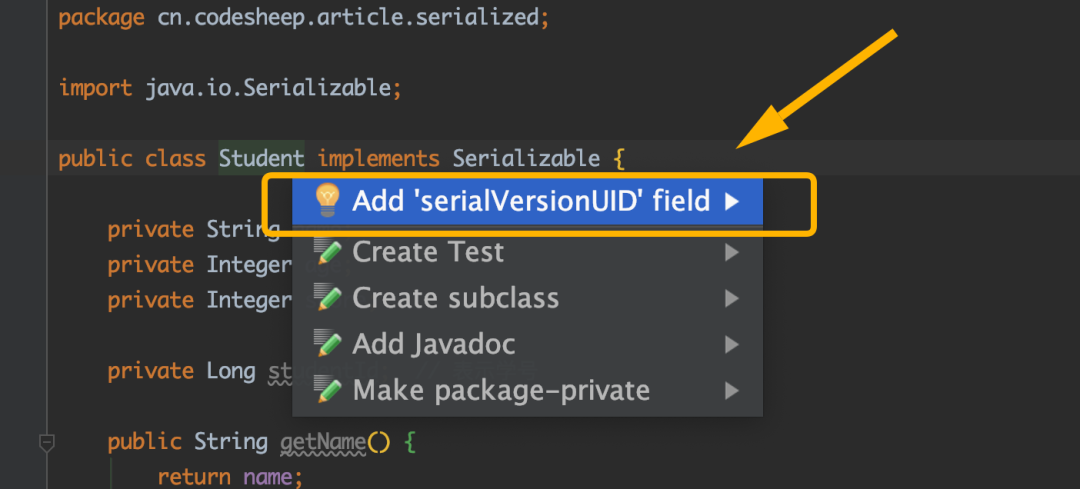
* **1、serialVersionUID是序列化前后的唯一标识符**
* **2、默认如果没有人为显式定义过serialVersionUID，那编译器会为它自动声明一个！**

**第1个问题：** serialVersionUID序列化ID，可以看成是序列化和反序列化过程中的“暗号”，在反序列化时，JVM会把字节流中的序列号ID和被序列化类中的序列号ID做比对，只有两者一致，才能重新反序列化，否则就会报异常来终止反序列化的过程。

**第2个问题：** 如果在定义一个可序列化的类时，没有人为显式地给它定义一个serialVersionUID的话，则Java运行时环境会根据该类的各方面信息自动地为它生成一个默认的serialVersionUID，一旦像上面一样更改了类的结构或者信息，则类的serialVersionUID也会跟着变化！

所以，为了serialVersionUID的确定性，写代码时还是建议，凡是implements Serializable的类，都最好人为显式地为它声明一个serialVersionUID明确值！

当然，如果不想手动赋值，你也可以借助IDE的自动添加功能，比如我使用的IntelliJ IDEA，按alt + enter就可以为类自动生成和添加serialVersionUID字段，十分方便：



# 两种特殊情况

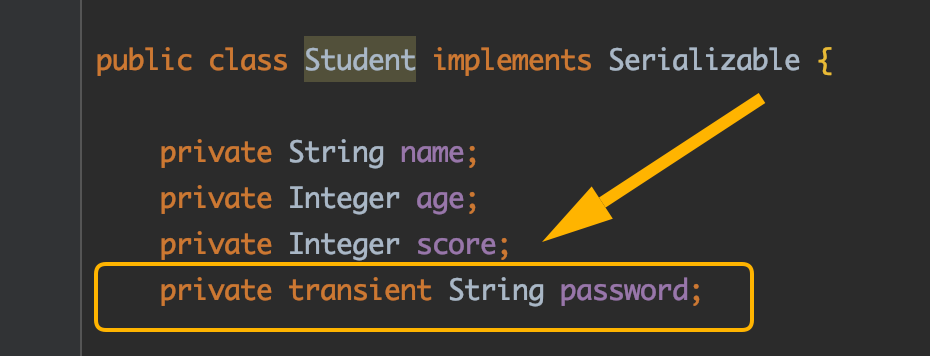
* 1、凡是被static修饰的字段是不会被序列化的
* 2、凡是被transient修饰符修饰的字段也是不会被序列化的

**对于第一点**，因为序列化保存的是**对象的状态**而非类的状态，所以会忽略static静态域也是理所应当的。

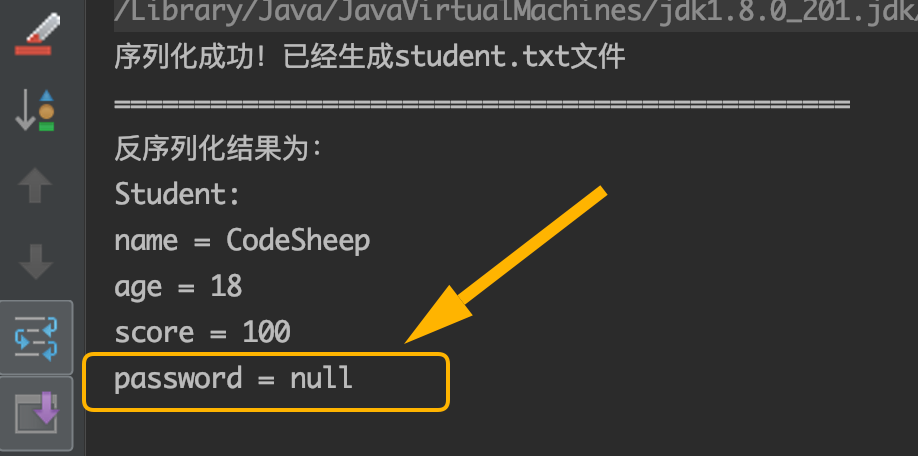
**对于第二点**，就需要了解一下transient修饰符的作用了。

如果在序列化某个类的对象时，就是不希望某个字段被序列化（比如这个字段存放的是隐私值，如：密码等），那这时就可以用transient修饰符来修饰该字段。

比如在之前定义的Student类中，加入一个**密码字段**，但是不希望序列化到txt文本，则可以：



这样在序列化Student类对象时，password字段会设置为默认值null，这一点可以从反序列化所得到的结果来看出：



# 序列化的受控和加强

## 约束性加持

从上面的过程可以看出，序列化和反序列化的过程其实是**有漏洞的**，因为从序列化到反序列化是有中间过程的，如果被别人拿到了中间字节流，然后加以伪造或者篡改，那反序列化出来的对象就会有一定风险了。

毕竟反序列化也相当于一种 **“隐式的”对象构造** ，因此我们希望在反序列化时，进行**受控的**对象反序列化动作。

那怎么个受控法呢？

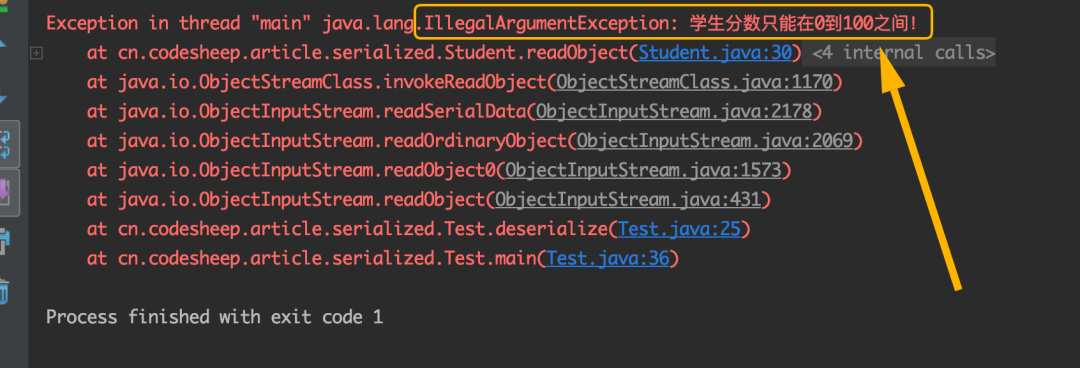
**答案就是：** 自行编写readObject()函数，用于对象的反序列化构造，从而提供约束性。

既然自行编写readObject()函数，那就可以做很多可控的事情：比如各种判断工作。

还以上面的Student类为例，一般来说学生的成绩应该在0 ~ 100之间，我们为了防止学生的考试成绩在反序列化时被别人篡改成一个奇葩值，我们可以自行编写readObject()函数用于反序列化的控制：



比如我故意将学生的分数改为101，此时反序列化立马终止并且报错：



对于上面的代码，有些小伙伴可能会好奇，为什么自定义的private的readObject()方法可以被自动调用，这就需要你跟一下底层源码来一探究竟了，我帮你跟到了ObjectStreamClass类的最底层，看到这里我相信你一定恍然大悟：



又是反射机制在起作用！是的，在Java里，果然万物皆可“反射”（滑稽），即使是类中定义的private私有方法，也能被抠出来执行了，简直引起舒适了。

## 单例模式增强

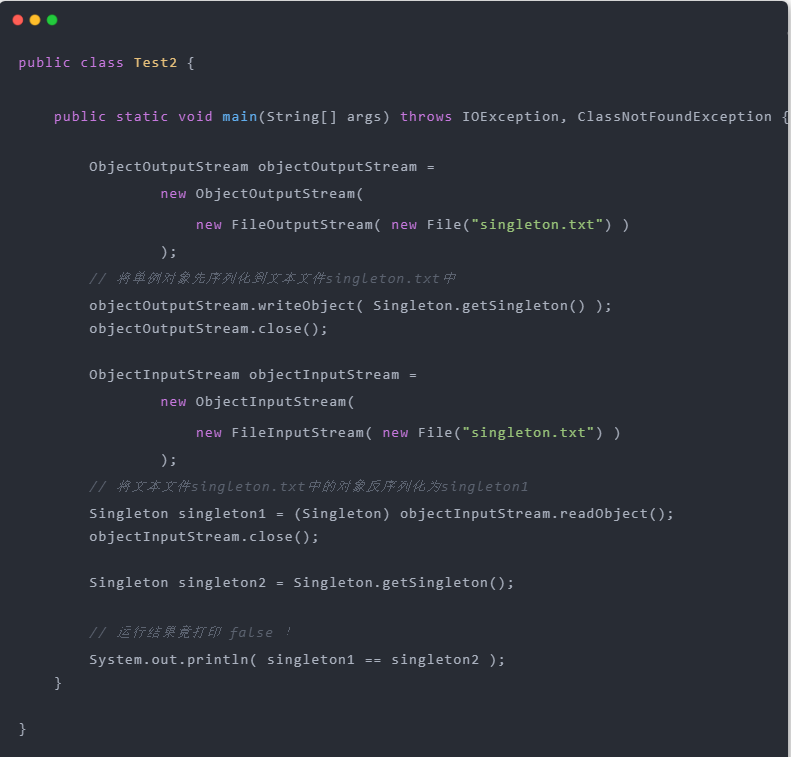
一个容易被忽略的问题是：**可序列化的单例类有可能并不单例**！

举个代码小例子就清楚了。

比如这里我们先用java写一个常见的「静态内部类」方式的单例模式实现：



然后写一个验证主函数：



运行后我们发现：**反序列化后的单例对象和原单例对象并不相等**了，这无疑没有达到我们的目标。

**解决办法是**：在单例类中手写readResolve()函数，直接返回单例对象，来规避之：

