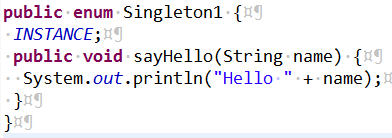
# Java单例模式的学习笔记

举个不严谨的通俗例子，我们用的电脑，只能配一个键盘，不论谁用，都只能通过这一个键盘进行输入，你说我非要用俩键盘，小明这边正敲代码，小张那边写文档，没必要还乱了套。

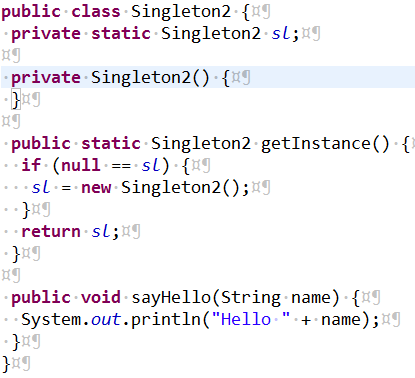
## 枚举

网上一搜单例模式的代码，上去肯定是什么饿汉懒汉，我先上个牛的。



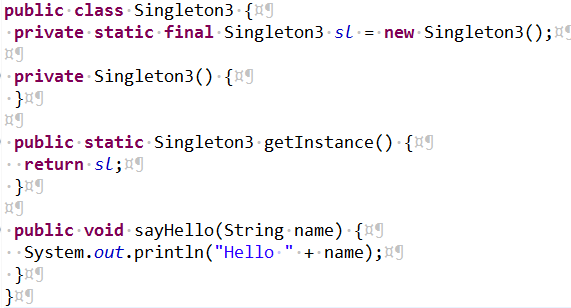
为什么牛呢？我只用一句话解释：这是Josh Bloch推荐的。不知道这位大神的出门左拐，面壁思过。

## 懒汉



这段代码应该是单例模式的Hello World了，一看就懂，想必很多人学习单例就是从这个开始的。代码、逻辑很清晰，很能说明问题，就是线程不安全，举个栗子打个样可以，实际工作中不要用。

## 饿汉

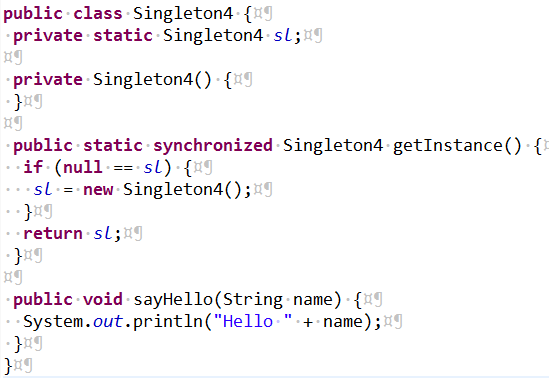


这个代码也很清晰，因为static final，在类加载的时候就把对象实例化了，又是线程安全。

这种写法还有个变种，就是new Singleton3()这一块用static块包起来，效果一样，不写了。

## 懒汉，线程安全

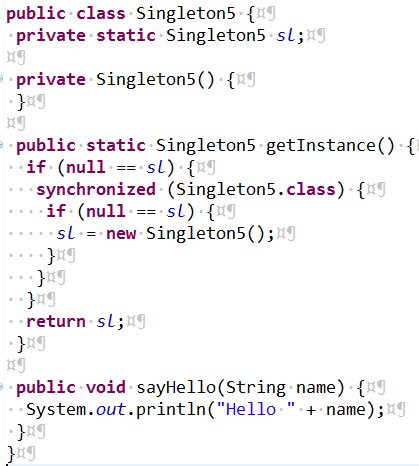
之前说的懒汉线程不安全，如果要求线程安全的话，大家第一反应肯定就是加锁。



就是在实例化的方法上加了同步锁，线程安全是做到了，但是大家都知道**synchronized**太耗资源了，没这个必要吧~~

## 双重校验锁

**synchronized**太耗资源了，它包含的代码越少越好，那么我们只需要把实例化这一块代码包住不就行了？



第一次校验的时候没有同步锁，如果已经实例化了，直接返回即可，省下了同步锁要消耗的资源。第二次校验是因为第一次校验没有同步锁，所以多个线程都可能进入，同步块内二重校验一下，保证不会生成多个实例。

但是这样其实还有隐患，就在于sl=new Singleton5()这一行，这不是一个原子操作！

类初始化步骤是

1. 给sl分配内存
2. 调用Singleton5的构造函数初始化
3. 将sl的对象指向分配好的内存空间

**在第3步，这个sl就不是null了**。

但是在多线程情况下，JVM很鸡贼的做了优化——指令重排序，简单说就是，JVM一旦觉得第2步花的时间长，就先把第3步给执行了。

对于上面的代码来说，先分配内存，然后不等初始化，直接指向分配好的内存，sl非null了！第二个线程过来一看是非null，那就直接返回，但是因为没有初始化，所以一用就会报错。

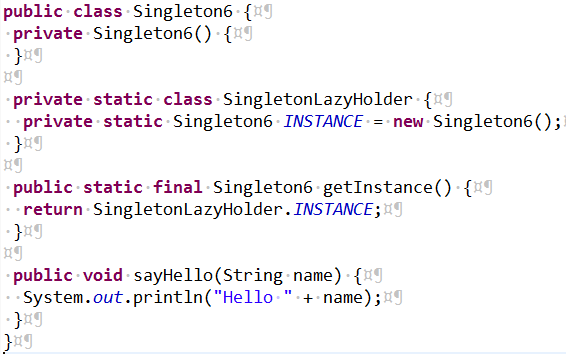
所以做如下改动即可：

**private** **volatile** **static** Singleton5 *sl*;

**volatile**可以禁止指令重排序优化。

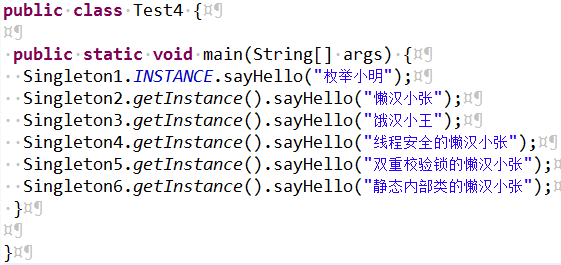
JDK5之前这么用还是有问题，JDK5已经修复了，可以放心大胆使用。

## 静态内部类



多解释两句，总体思想还是懒汉思维，采用一个静态内部类（SingletonLazyHolder），因为JVM的机制（静态内部类被引用才加载），使得SingletonLazyHolder只在调用getInstance()方法时，因为SingletonLazyHolder.INSTANCE这行代码，产生了对它的引用才加载进去，而且天然的线程安全，达到线程安全的懒汉效果。

## 测试类



## 结果：

