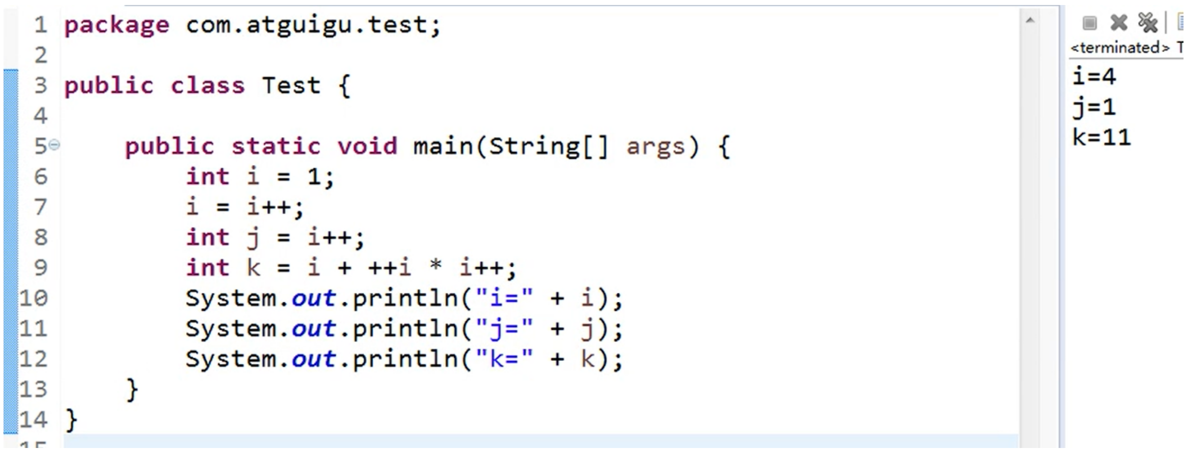
day1

时间：2022年4月6日13:05:17

一.自增运算符



```java
package com.atguigu.test;

public class Test {

    public static void main(String[] args) {
        int i = 1;
        i = i++;
        int j = i++;
        int k = i + ++i * i++;
        System.out.println("i=" + i);
        System.out.println("j=" + j);
        System.out.println("k=" + k);
    }
}
```

```
<terminated>
i=4
j=1
k=11
```

二.单例

要点

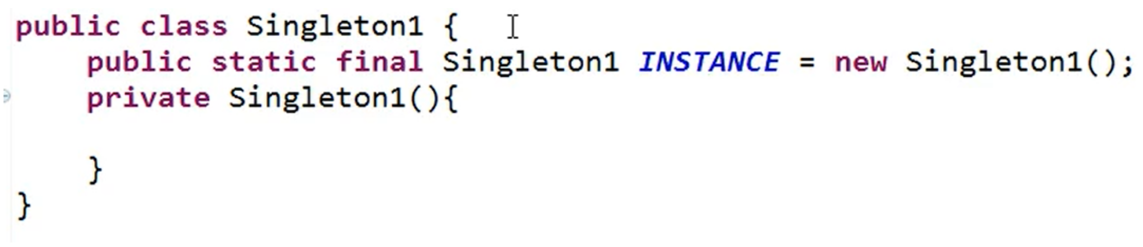
一是某个类只能有一个实例;(构造器私有化)

二是它必须自行创建这个实例;(类的静态变量保存这个实例)

三是它必须自行向整个系统提供这个实例;(public或者getter)

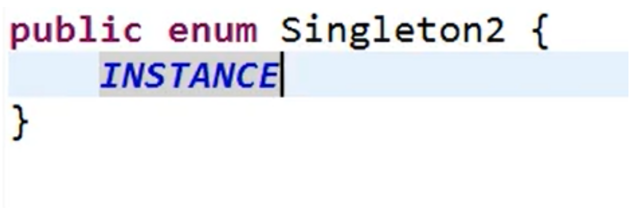
1.饿汉式：直接创建，不存在线程安全问题

a.静态初始化（简介直观）



```java
public class Singleton1 {
    public static final Singleton1 INSTANCE = new Singleton1();
    private Singleton1(){

    }
}
```

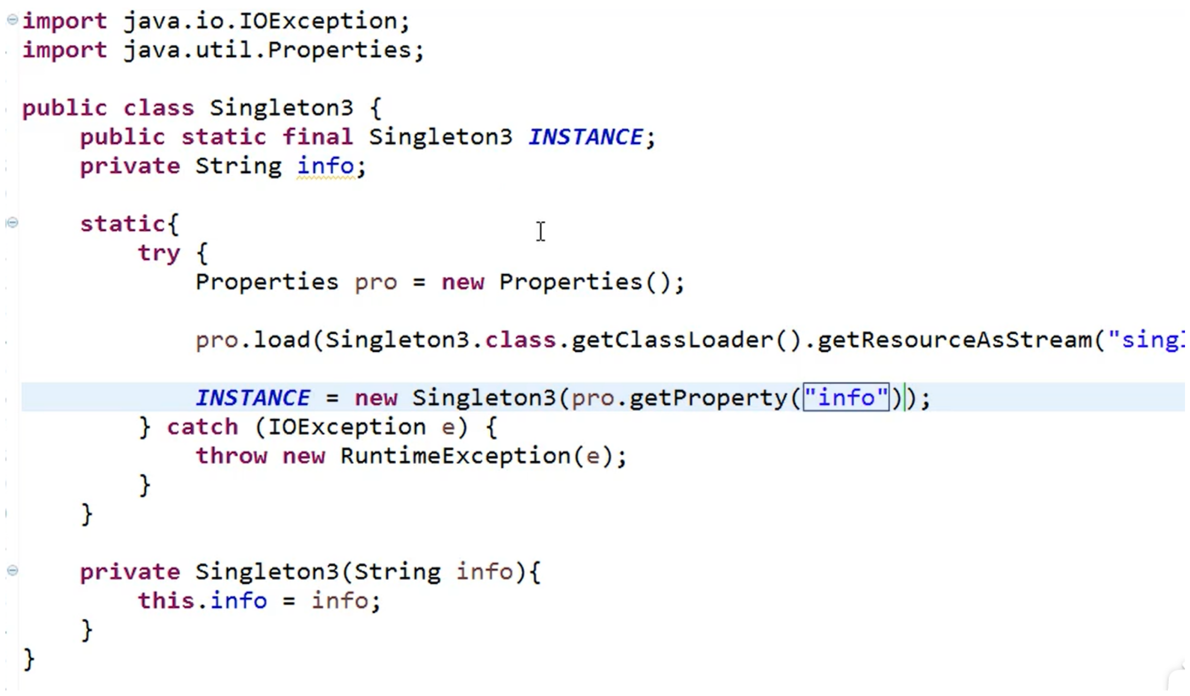
b.枚举(最简洁)



```java
public enum Singleton2 {
    INSTANCE
}
```

```
使用
    由于是public，直接点运算符即可
两者的toString不同
    枚举的toString就是INSTANCE
    前者的toString自定义
```

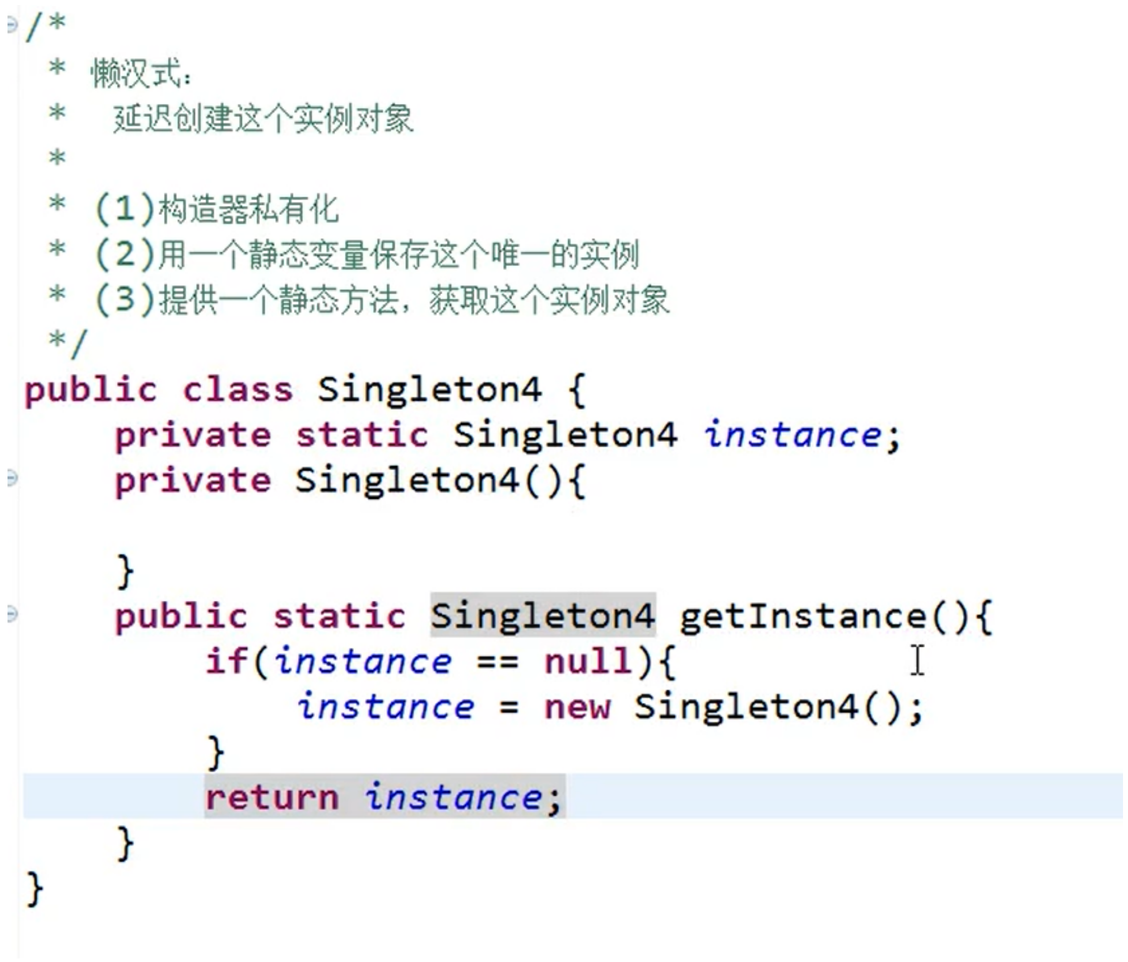
c.静态代码块（适合复杂化实例，例如需要配置文件）



```java
import java.io.IOException;
import java.util.Properties;

public class Singleton3 {
    public static final Singleton3 INSTANCE;
    private String info;

    static{
        try {
            Properties pro = new Properties();

            pro.load(Singleton3.class.getClassLoader().getResourceAsStream("sing

            INSTANCE = new Singleton3(pro.getProperty("info"));
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }

    private Singleton3(String info){
        this.info = info;
    }
}
```

2.懒汉式：延迟创建对象

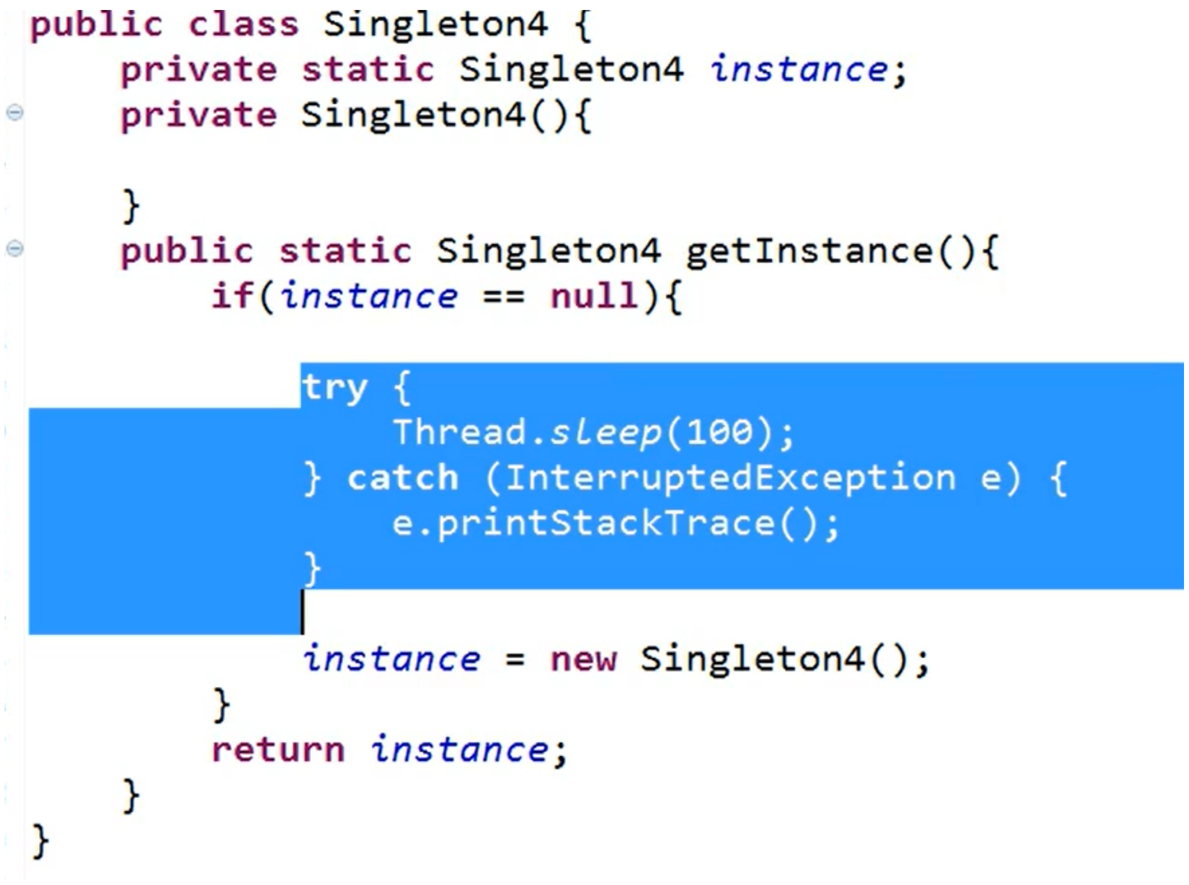
a.线程不安全（单线程）



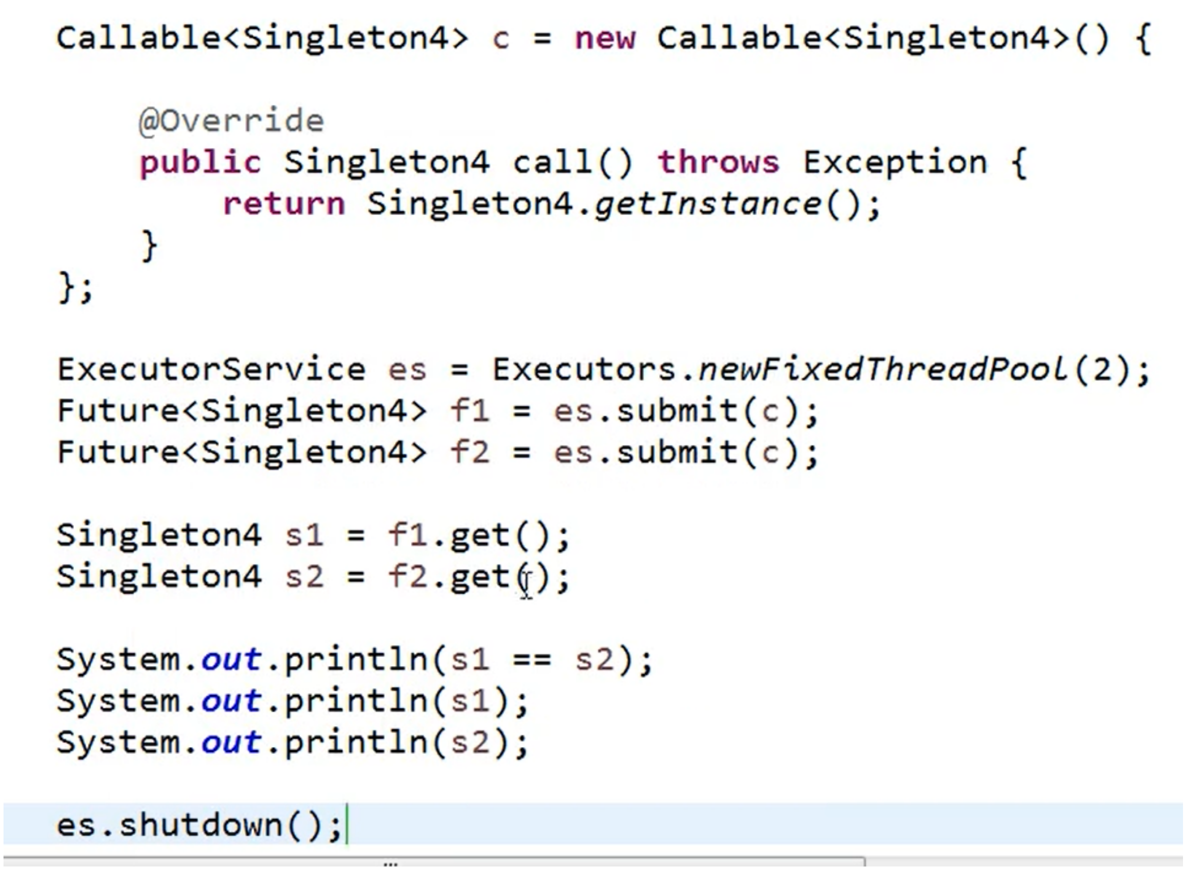
```java
/*
 * 懒汉式：
 *   延迟创建这个实例对象
 *
 * (1)构造器私有化
 * (2)用一个静态变量保存这个唯一的实例
 * (3)提供一个静态方法，获取这个实例对象
 */
public class Singleton4 {
    private static Singleton4 instance;
    private Singleton4(){

    }
    public static Singleton4 getInstance(){
        if(instance == null){
            instance = new Singleton4();
        }
        return instance;
    }
}
```

```
instance必须是private，防止获取null
```

会出现的问题



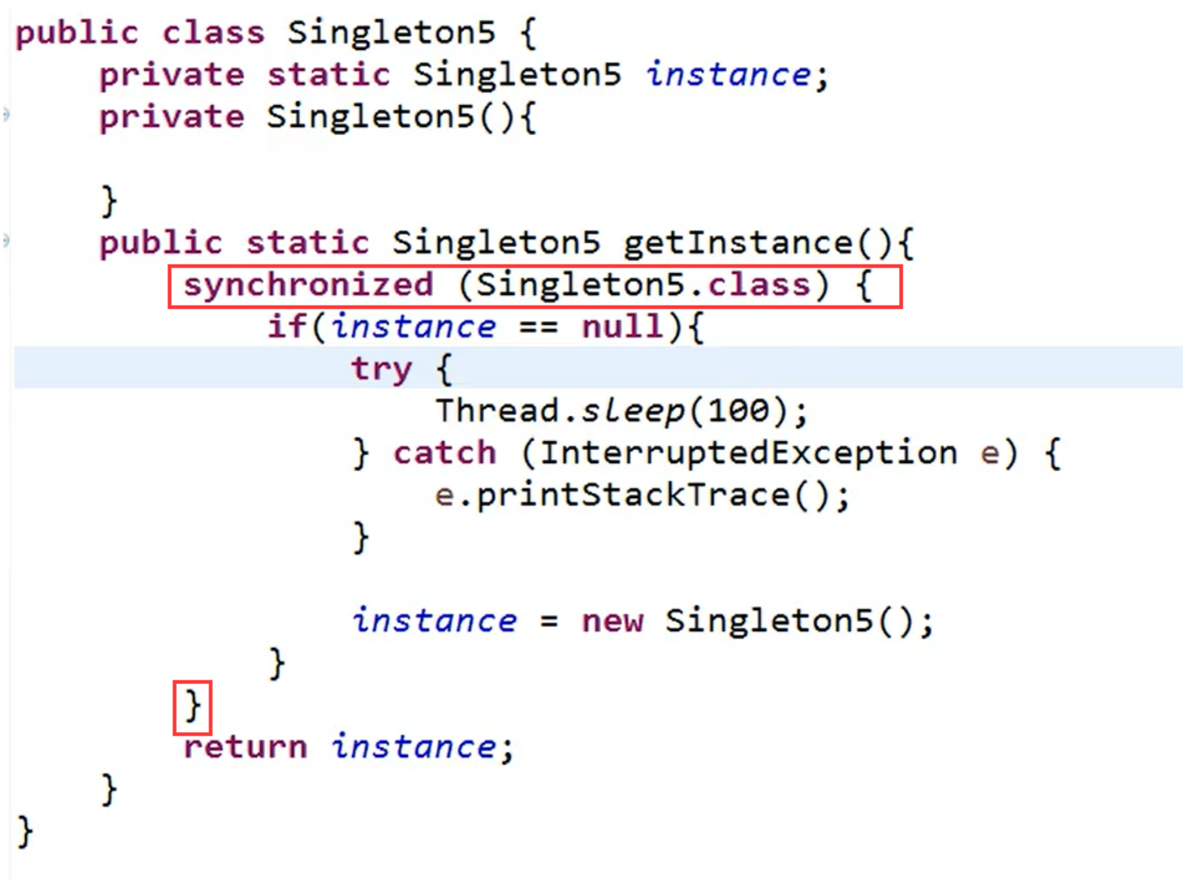
```java
public class Singleton4 {
    private static Singleton4 instance;
    private Singleton4(){

    }
    public static Singleton4 getInstance(){
        if(instance == null){

            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            instance = new Singleton4();
        }
        return instance;
    }
}
```


```java
    Callable<Singleton4> c = new Callable<Singleton4>() {

        @Override
        public Singleton4 call() throws Exception {
            return Singleton4.getInstance();
        }
    };

    ExecutorService es = Executors.newFixedThreadPool(2);
    Future<Singleton4> f1 = es.submit(c);
    Future<Singleton4> f2 = es.submit(c);

    Singleton4 s1 = f1.get();
    Singleton4 s2 = f2.get();

    System.out.println(s1 == s2);
    System.out.println(s1);
    System.out.println(s2);

    es.shutdown();
```

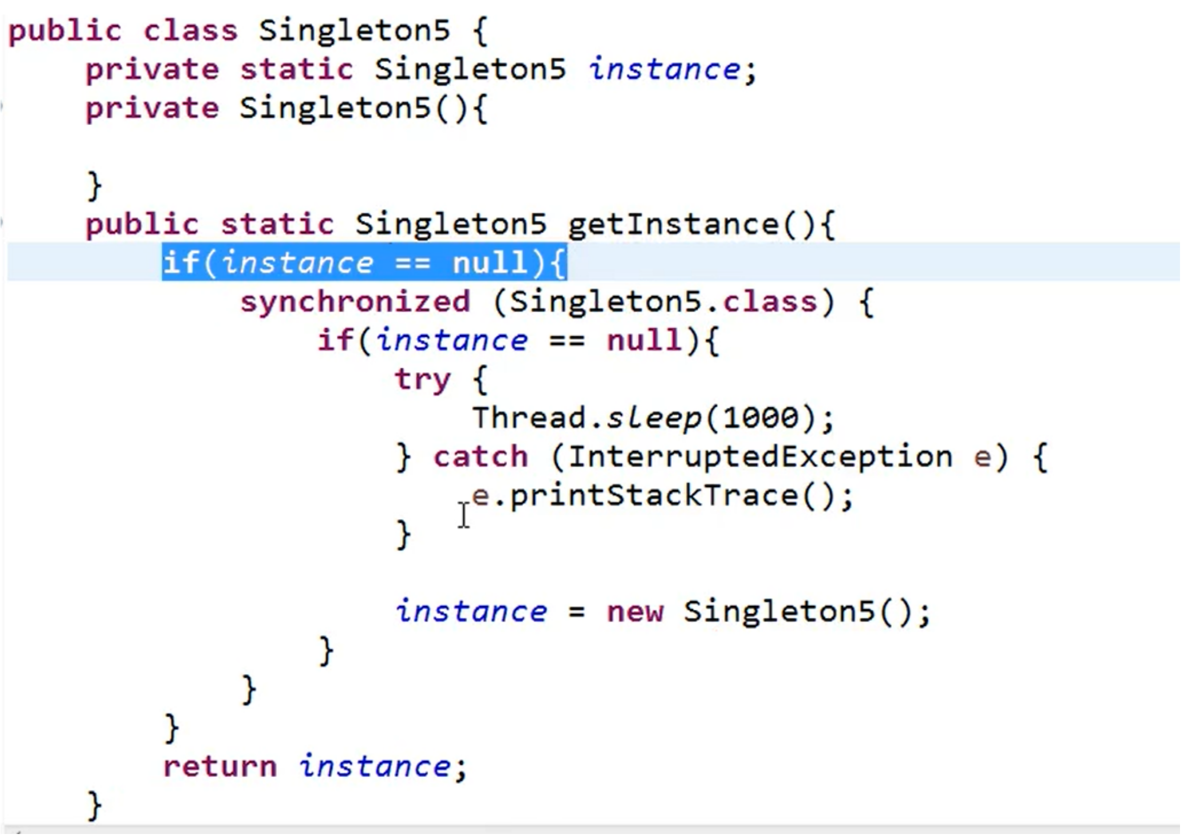
```
使用了sleep之后，直接让出cpu，导致其他线程进入
```

b.线程安全（多线程）

版本一（解决安全问题）

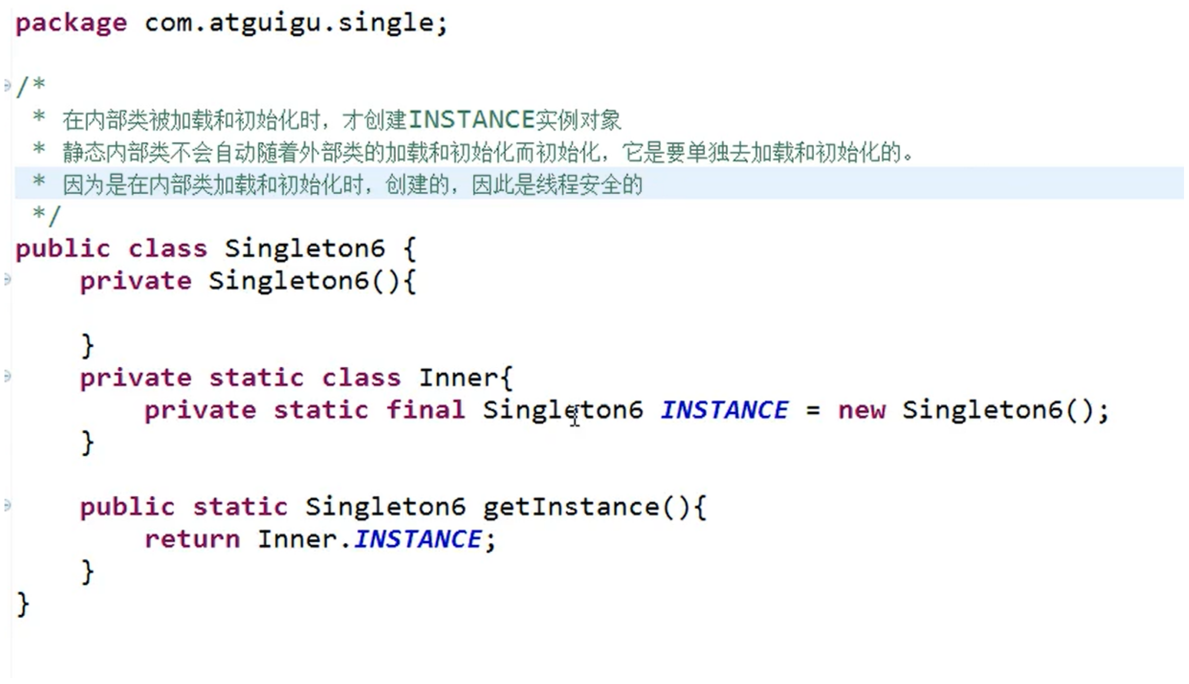

```java
public class Singleton5 {
    private static Singleton5 instance;
    private Singleton5(){

    }
    public static Singleton5 getInstance(){
        synchronized (Singleton5.class) {
            if(instance == null){
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                instance = new Singleton5();
            }
        }
        return instance;
    }
}
```

版本二（解决效率问题）



```java
public class Singleton5 {
    private static Singleton5 instance;
    private Singleton5(){

    }
    public static Singleton5 getInstance(){
        if(instance == null){
            synchronized (Singleton5.class) {
                if(instance == null){
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }

                    instance = new Singleton5();
                }
            }
        }
        return instance;
    }
```

c.静态内部类（多线程，简洁一些）



```java
package com.atguigu.single;

/*
 * 在内部类被加载和初始化时，才创建INSTANCE实例对象
 * 静态内部类不会自动随着外部类的加载和初始化而初始化，它是要单独去加载和初始化的。
 * 因为是在内部类加载和初始化时，创建的，因此是线程安全的
 */
public class Singleton6 {
    private Singleton6(){

    }
    private static class Inner{
        private static final Singleton6 INSTANCE = new Singleton6();
    }

    public static Singleton6 getInstance(){
        return Inner.INSTANCE;
    }
}
```