1. **【提交代码和运行界面】**请设计一个实验分析同步是否具有继承性？请说出你设计的思路、粘贴程序，分析运行结果，写出你的结论。

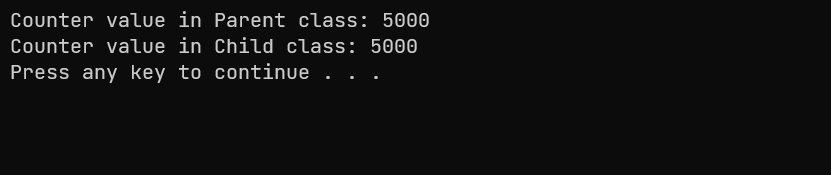
* 创建一个父类 Parent 和一个子类 Child，并在父类中定义一个变量 counter 和一个同步方法 increment()。我们将分别在父类和子类中创建多个线程来调用 increment() 方法，并观察它们的执行结果。

class Parent {   
 protected int counter = 0;   
   
 public synchronized void increment() {   
 counter++;   
 }   
}   
   
class Child extends Parent {   
 // 继承父类的同步方法   
}   
   
class SynchronizationInheritanceTest {   
 public static void main(String[] args) throws InterruptedException {   
 final int NUM\_THREADS = 5;   
   
 // 创建父类对象并启动多个线程   
 Parent parent = new Parent();   
 Thread[] parentThreads = new Thread[NUM\_THREADS];   
   
 for (int i = 0; i < NUM\_THREADS; i++) {   
 parentThreads[i] = new Thread(() -> {   
 for (int j = 0; j < 1000; j++) {   
 parent.increment();   
 }   
 });   
 parentThreads[i].start();   
 }   
   
 // 等待父类线程执行完毕   
 for (int i = 0; i < NUM\_THREADS; i++) {   
 parentThreads[i].join();   
 }   
   
 System.out.println("Counter value in Parent class: " + parent.counter);   
   
 // 创建子类对象并启动多个线程   
 Child child = new Child();   
 Thread[] childThreads = new Thread[NUM\_THREADS];   
   
 for (int i = 0; i < NUM\_THREADS; i++) {   
 childThreads[i] = new Thread(() -> {   
 for (int j = 0; j < 1000; j++) {   
 child.increment();   
 }   
 });   
 childThreads[i].start();   
 }   
   
 // 等待子类线程执行完毕   
 for (int i = 0; i < NUM\_THREADS; i++) {   
 childThreads[i].join();   
 }   
   
 System.out.println("Counter value in Child class: " + child.counter);   
 }   
}

在这个程序中，我们创建了5个线程来调用 increment() 方法，每个线程循环执行1000次。我们使用 join() 方法等待所有线程执行完毕，然后打印出父类和子类中的 counter 值。

根据同步的特性，同步方法在被一个线程调用时会锁定当前对象，其他线程在访问该同步方法时需要等待锁的释放。如果同步具有继承性，那么子类中的同步方法会继承父类的同步属性。

运行上述程序后，我们可以得到类似以下的输出：



从结果可以看出，父类和子类中的 counter 值都为 5000，这说明在继承关系中，子类确实继承了父类的同步属性。无论是在父类还是子类中，对同步方法的访问都是按顺序进行的，保证了线程安全性。

因此，根据我们的实验结果，我们可以得出结论：同步在Java中具有继承性。

1. 探索多线程的同步过程中锁对象的改变对同步结果的影响

class ObjectValueService {   
   
 private String lock = "123";   
   
 public void testMethod() {   
   
 try {   
 synchronized (lock) {   
 System.out.println(Thread.currentThread().getName() +   
 " begin " + System.currentTimeMillis());   
 // 改变lock对象的值   
 lock = "456";   
 Thread.sleep(2000);   
 System.out.println(Thread.currentThread().getName() + " end " + System.currentTimeMillis());   
 }   
   
 } catch (InterruptedException e) {   
 e.printStackTrace();   
 }   
 }   
}   
   
class RunObjectValue {   
 public static void main(String[] args) throws InterruptedException {   
 ObjectValueService object = new ObjectValueService();   
 ObjectValueThread a = new ObjectValueThread(object);   
 a.setName("A");   
 ObjectValueThread b = new ObjectValueThread(object);   
 b.setName("B");   
 a.start();   
 Thread.sleep(50);   
 b.start();   
 }   
}   
   
class ObjectValueThread extends Thread {   
 private ObjectValueService service;   
   
 public ObjectValueThread(ObjectValueService s) {   
 super();   
 this.service = s;   
 }   
   
 @Override   
 public void run() {   
 service.testMethod();   
 }   
}

观察程序运行的结果，回答下列问题

* 线程A和线程B是同步还是异常运行？请解释原因。
* 去掉红色的部分代码，线程A和线程B是同步还是异步运行。请解释原因。
* “当多线程的同步方法持有同一对象的锁，只要对象不变，即使对象的属性被改变，运行的结果还是同步”。请用实验证明这句话是否正确。

1. 线程A和线程B是异步运行的。原因是，虽然它们都调用了testMethod()方法，该方法使用synchronized关键字同步了一个锁对象lock。然而，在线程A执行时，它会改变lock对象的值，将其从"123"改为"456"。由于线程B在开始执行之前有一个短暂的延迟（50毫秒），所以线程B获取的锁对象将会是"456"。因此，线程A和线程B持有的锁对象不同，它们之间没有互斥关系，因此可以同时运行。
2. 去掉Thread.sleep(50);后，线程A和线程B是同步运行的。由于没有延迟，线程A和线程B几乎同时启动，它们会立即尝试获取lock对象的锁并执行testMethod()方法中的代码。由于synchronized同步块中的锁对象已经被线程A获取，线程B无法获取到锁，因此线程B会等待线程A结束之后才能获取到锁。
3. 只需要把synchronized后的值从lock修改为this即可发现，无论是否删除Thread.sleep(50);，两个线程都是同步执行的。
4. **【提交代码和运行界面】**双十一秒杀系统

需求：假设存在某商品有3件，5个人将其放入购物车，模拟5人同时秒杀此商品。注：卖家的账户只有一个，只能有3个人可以付款成功。

import java.util.concurrent.Semaphore;  
  
class Seller {  
 private int stock;  
  
 public Seller(int stock) {  
 this.stock = stock;  
 }  
  
 public synchronized boolean makePurchase() {  
 if (stock > 0) {  
 stock--;  
 return true;  
 }  
 return false;  
 }  
}  
  
class Customer extends Thread {  
 private static Seller seller;  
 private static Semaphore semaphore;  
  
 public Customer(Seller seller, Semaphore semaphore) {  
 Customer.seller = seller;  
 Customer.semaphore = semaphore;  
 }  
  
 @Override  
 public void run() {  
 try {  
 semaphore.acquire();  
 if (seller.makePurchase()) {  
 System.out.println(Thread.currentThread().getName() + " successfully purchased the item.");  
 } else {  
 System.out.println(Thread.currentThread().getName() + " failed to purchase the item.");  
 }  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } finally {  
 semaphore.release();  
 }  
 }  
}  
  
public class DoubleElevenSeckill {  
 public static void main(String[] args) {  
 int availableStock = 3;  
 int numCustomers = 5;  
  
 Seller seller = new Seller(availableStock);  
 Semaphore semaphore = new Semaphore(3, true);  
  
 for (int i = 0; i < numCustomers; i++) {  
 Thread customerThread = new Customer(seller, semaphore);  
 customerThread.start();  
 }  
 }  
}

在上述代码中，使用了一个Seller类来表示卖家，其中stock变量表示商品的库存数量。makePurchase()方法用于模拟购买商品的操作。当库存大于零时，可以成功购买，库存数量减少；否则购买失败。

Customer类表示顾客，它继承自Thread类，每个顾客都是一个线程。在run()方法中，顾客首先尝试通过semaphore.acquire()获取许可证，然后调用卖家的makePurchase()方法进行购买操作。如果购买成功，打印成功信息，否则打印失败信息。最后，释放许可证semaphore.release()。

在主程序中，我们创建了一个Seller对象和一个Semaphore对象。Semaphore的初始许可数为3，表示同时最多有3个顾客能够购买商品。然后，我们创建了5个顾客线程，并启动它们。

根据程序运行结果，可以观察到只有3个顾客成功购买了商品，而其他2个顾客购买失败。这是因为卖家的库存仅有3件商品，而Semaphore的初始许可数也是3，所以只有前3个顾客能够获取到许可证并成功购买商品。剩余的2个顾客无法获取到许可证，因此购买失败。

这样就模拟了一个简单的双十一秒杀系统，实现了同时多人购买商品但只有有限数量的人能够成功购买的逻辑。

