**实验4 多线程Java程序的设计和调试**

### 1.实验目的

（1）练习Java并发程序设计，理解线程安全设计模式。

（2）通过程序调试理解并掌握同步控制策略、锁的设计和使用、同步控制范围等。

### 2.主要知识要点

2.1 线程安全性

线程安全性强调程序运行结果的正确性不会因线程调度状态的变化而不满足。一个线程在调用一个对象方法时，如果其他线程也不受控制的访问该对象，则会使该对象的状态变化不受控，因而产生不可预料的问题。假设该对象方法本身的功能实现正确，则这样的问题可称为线程不安全问题。解决线程不安全问题的主要措施是进行访问控制。线程安全的类设计是一种推荐的解决措施，一个类如果是线程安全的类，则任何使用该类的线程无需额外使用同步控制措施来访问对该类对象。

2.2 寻找线程不安全的方法

（1）明确哪些类会被多个线程使用

（2）明确多个线程的共享数据（即共享对象）

（3）明确线程代码中哪些语句操作/访问共享数据

有可变成员变量的类用于多线程时存在不安全问题，其原因是线程可能对这个成员变量实施非原子性的读写操作。而定义在方法内的局部变量一般不会引起线程安全问题，其原因是这些数据存在栈中，不会产生共享。但是需要注意的是，如果局部变量是一个对象引用，则一样难以避免线程不安全隐患。

考虑下面程序中的变量读取、写入例子，通过方法锁来控制线程同步对SafeCounter对象方法的调用

/\* 线程安全的计数器SafeCounter，维护计数变量的自增\*/

public final class SafeCounter {

private long value = 0;

public synchronized long getValue() {

return value;

}

public synchronized long increment() {

return ++value;

}

}

再例如下面例子中通过方法锁来解决check-then-act类型的线程安全问题。

/\*\*懒汉式单例模式中对变量的判断操作\*/

public class Singleton {

private static Singleton singleton = null; //私有的静态成员变量

//构造方法被设为私有，防止外部使用new来创建对象，破坏单例

private Singleton(){

System.out.println("构造函数被调用");

}

//公有的静态方法，供外部调用来获取单例对象

public static synchronized Singleton getInstance(){

if(singleton == null){ //第一次调用该方法时，创建对象。

singleton = new Singleton();

}

return singleton;

}

2.3 确保线程安全的办法

把复合的操作原子化能够解决线程安全的问题。例如Read-Modify-Write，或者Check-Then-Act这样的操作，都是一种复合操作，如果我们将这些操作都原子化，那问题也就解决了。原子操作可以由多种方式实现，如对操作加锁，或者对整个方法加锁等。

另外还可以使用JDK自带的线程安全类Atomic\*\*\*。JDK自带的线程安全类只能保证对单一对象的访问安全，如果多个对象之间存在数据依赖关系，则这种决绝方案不能杜绝线程不安全问题。

线程安全问题来源于多个线程并发访问共享对象。如果共享对象不会发生变化，即是不可变对象，则并发读不会产生不安全问题。如JDK提供的String和Integer类是不可变对象类。此外，程序设计者也可通过final关键词来强制程序在运行时不能修改变量的取值，从而构造出相应的不可变对象。

### 3. 实验内容

**题目1：**以下是一段用Java模拟的两个银行账户之间的操作，完成将资金从一个账户转入另一个账户，该操作有一个基本的不变性条件，如账户的余额不能为负数，否则没法进行转帐。在开始转帐之前，程序试图获得这两个Account对象的锁，以确保同时更新两个账户中余额时的线程安全。请构造能够让所给程序出现死锁的场景，然后修复程序消除死锁。

提示：动态锁的顺序有可能引发死锁。在制定锁的顺序时，可以使用System.identityHashCode方法，该方法将返回由Object.hashCode返回的值。

**import** java.util.concurrent.atomic.\*;

**public** **class** DynamicOrderDeadlock {

**public** **static** **void** transferMoney(Account fromAccount,

Account toAccount,

DollarAmount amount)

**throws** InsufficientFundsException {

**synchronized** (fromAccount) {

**synchronized** (toAccount) {

**if** (fromAccount.getBalance().compareTo(amount) < 0)

**throw** **new** InsufficientFundsException();

**else** {

fromAccount.debit(amount);

toAccount.credit(amount);

}

}

}

}

**static** **class** DollarAmount **implements** Comparable<DollarAmount> {

// Needs implementation

**public** DollarAmount(**int** amount) {

}

**public** DollarAmount add(DollarAmount d) {

**return** **null**;

}

**public** DollarAmount subtract(DollarAmount d) {

**return** **null**;

}

**public** **int** compareTo(DollarAmount dollarAmount) {

**return** 0;

}

}

**static** **class** Account {

**private** DollarAmount balance;

**private** **final** **int** acctNo;

**private** **static** **final** AtomicInteger *sequence* = **new** AtomicInteger();

**public** Account() {

acctNo = *sequence*.incrementAndGet();

}

**void** debit(DollarAmount d) {

balance = balance.subtract(d);

}

**void** credit(DollarAmount d) {

balance = balance.add(d);

}

DollarAmount getBalance() {

**return** balance;

}

**int** getAcctNo() {

**return** acctNo;

}

}

**static** **class** InsufficientFundsException **extends** Exception {

}

###### 题目2：请编写代码模拟银行窗口服务的排号工作过程，即利用多个线程模拟银行柜台叫号的过程。

###### 提示：

###### 1. 银行大厅里有一台取号机，不断地有顾客进入银行的大厅，每位客户进门就需要从取号机中取出号码。号码顺序递增产生新号（新客户）。现假定取号机产生两个相邻号码的时间间隔不超过3秒。每次有客户取到号就提示“\*\*号顾客正在等待服务”。

###### 2. 柜台为每一名客户服务的时间模拟为不超过100秒，可以是1～100秒之间的任意一个时间。（可以用TimeUnit.MILLISECONDS.sleep(rand.nextInt(100000))实现）

当客户队列中没有更多需要服务的客户时，柜台处于等待状态。

###### 3. 用生产者消费者模型解决并发协作，你可以使用如下任意一种：

###### 1）wait和notify方法在生产者和消费者线程中合作，在队列满了或者队列是空的条件下阻塞（参考代码详见demo1.rar）

###### 2) 你也可以使用阻塞队列（BlockingQueue）来完成这个任务（参考代码详见demo2.rar）

###### 4. 程序模拟的叫号提示输出格式统一为“请\*\*号顾客到\*\*号窗口”，工作人员为顾客提供服务的过程中，窗口显示“\*\*号窗口正在为\*\*号顾客办理业务”。

###### 请根据给出的参考代码以及上述要求，任意选择一种思路补充和完善参考代码中缺少的部分，使之能通过CalNum.java中的测试。