public class TestWindow1 {  
  
 public static void main(String[] args) {  
 Window1 w1 = new Window1();  
  
 Thread t1 = new Thread(w1);  
 Thread t2 = new Thread(w1);  
 Thread t3 = new Thread(w1);  
  
 t1.setName("窗口1");  
 t1.setName("窗口2");  
 t1.setName("窗口3");  
 t1.start();  
 t2.start();  
 t3.start();  
 }  
}  
  
class Window1 implements Runnable {  
 int ticket = 100;  
  
 @Override  
 public void run() {  
 while (true) {  
 if (ticket > 0) {  
 try {  
 Thread.sleep(10);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.***out***.println(Thread.currentThread().getName() + "售票，票号为：" + ticket--);  
 } else {  
 break;  
 }  
 }  
 }  
}

* **此程序存在线程安全问题：打印车票时会出现重票、错票。**
* 线程安全问题出现的原因：
* 由于一个线程在操作共享数据过程中，未执行完毕的情况下，另外的线程参与进来，导致共享数据存在了安全问题。
* 如何解决线程的安全问题？
* 对多条操作共享数据的语句，只能让一个线程都执行完，在执行过程中，其他线程不可参与执行。
* **线程的同步**
* 多个线程执行的不确定性引起执行结果的不稳定。
* 多个线程对账本的共享，会造成操作的不完整性，会破坏数据。
* **实现线程安全问题——同步机制**
* **方式一：同步代码块**

synchronized(同步监视器）{

//需要被同步的代码；（即为操作共享数据的代码块）

}

* 同步监视器：由任何一个类的对象来充当。哪个线程获取此监视器，谁就执行大括号里被同步的。俗称：锁。

public class TestWindow2 {  
  
 public static void main(String[] args) {  
 Window2 w2 = new Window2();  
  
 Thread t1 = new Thread(w2);  
 Thread t2 = new Thread(w2);  
 Thread t3 = new Thread(w2);  
  
 t1.setName("窗口1");  
 t1.setName("窗口2");  
 t1.setName("窗口3");  
 t1.start();  
 t2.start();  
 t3.start();  
 }  
}  
  
class Window2 implements Runnable {  
 /\*\*  
 \* 共享数据  
 \*/  
 int ticket = 100;  
 //Object obj = new Object();  
  
 @Override  
 public void run() {  
 while (true) {  
 synchronized (this) {  
 if (ticket > 0) {  
 try {  
 Thread.sleep(10);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.***out***.println(Thread.currentThread().getName() + "售票，票号为：" + ticket--);  
 } else {  
 break;  
 }  
 }  
 }  
 }  
}

* 要求所有线程必须公用同一把锁。
* 注：在实现的方式中，考虑同步的话，可以使用this来充当锁。但是在继承的方式中，慎用this。

public class TestWindow3 {  
  
 public static void main(String[] args) {  
  
 Window3 w1 = new Window3();  
 Window3 w2 = new Window3();  
 Window3 w3 = new Window3();  
  
 w1.setName("窗口1");  
 w1.setName("窗口2");  
 w1.setName("窗口3");  
  
 w1.start();  
 w2.start();  
 w3.start();  
 }  
}  
  
class Window3 extends Thread {  
  
 static int ticket = 100;  
 static Object obj = new Object();  
  
 @Override  
 public void run() {  
 while (true) {  
 synchronized (obj) { //在本问题中，this表示w1，w2，w3。  
 if (ticket > 0) {  
 try {  
 Thread.sleep(10);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.***out***.println(Thread.currentThread().getName() + "售票，票号为：" + ticket--);  
 } else {  
 break;  
 }  
 }  
 }  
 }  
}

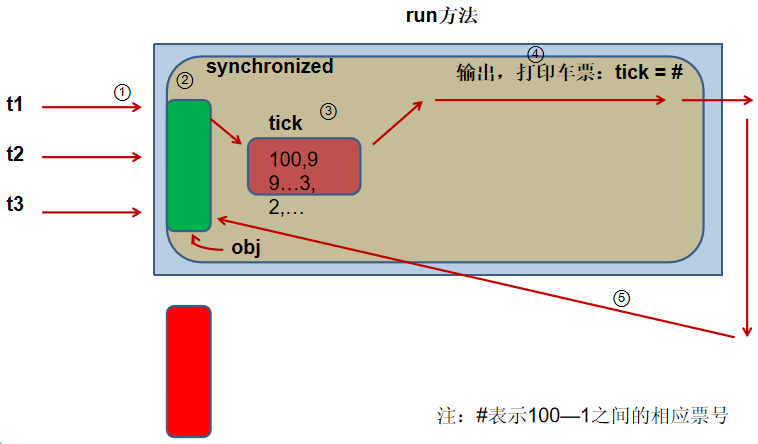
* **方式二：同步方法**

public synchronized void show(String name){

….

}

* 将操作共享数据的方法声明为synchronized。即此方法为同步方法，能够保证当其中一个线程执行此方法时，其他线程在外等待直至此线程执行完此方法。
* 同步方法的锁：this。
* **分析同步原理**



* **设计模式值懒汉式的线程安全问题**

public class TestSingleton {  
  
 public static void main(String[] args) {  
 Singleton s1 = Singleton.getInstance();  
 Singleton s2 = Singleton.getInstance();  
 System.***out***.println(s1 == s2);  
 }  
}  
  
/\*\*  
 \* 关于单例设计模式之懒汉式的线程安全问题。  
 \* 对于静态方法而言，使用当前类本身充当锁。  
 \* 线程的同步的弊端：由于同一个时间只能有一个线程访问共享数据，效率变低了。  
 \*/  
class Singleton {  
  
 /\*\*  
 \* 私有化构造器  
 \*/  
 private Singleton() {  
  
 }  
  
 /\*\*  
 \* 声明类的引用  
 \*/  
 private static Singleton instance = null;  
  
 /\*\*  
 \* 设置公共的方法来访问类的实例  
 \* @return  
 \*/  
 public static Singleton getInstance() {  
  
 if (instance == null) {  
   
 synchronized (Singleton.class) {  
 if (instance == null) {  
 instance = new Singleton();  
 }  
 }  
 }  
  
 return instance;  
 }  
}

* 互斥锁

在Java语言中，引入了对象互斥锁的概念，来保证共享数据操作的完整性。

* 每个对象都对应于一个可称为“互斥锁”的标记，这个标记用来保证在任一时刻，**只能有一个线程访问该对象**。
* 关键字synchronized来与对象的互斥锁联系。当某个对象用synchronized修饰时，表明该对象在任一时刻只能由一个线程访问。
* 同步的局限性：导致程序的执行效率要降低
* 同步方法（非静态的）的锁为this。
* 同步方法（静态的）的锁为当前类本身。