лабораторно упражнение 11.

Синхронизация на нишки. Реализиране на многонишков сървър със синхронизирани методи.

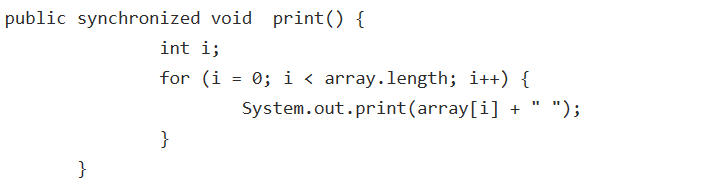
Нека имаме масив с 1500 случайни числа от целочислен тип. В дадения пример ще създадем две нишки. Едната сортира масива, а другата принтира масива.

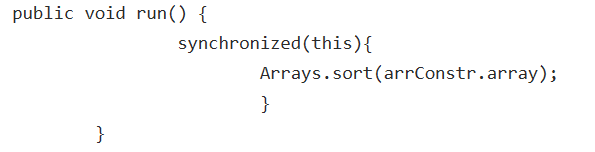
**public class** Test {  
  
 **public static void** main(String[] args) {  
 ArrayConstruction arrConstr = **new** ArrayConstruction();  
 PrintingThread printingThr = **new** PrintingThread(arrConstr);  
 SortingThread sortingThr = **new** SortingThread(arrConstr);  
 arrConstr.create();  
 arrConstr.print();  
 System.***out***.println();  
 System.***out***.println(**"After construction"**);  
 printingThr.start();  
 sortingThr.start();  
  
  
 }  
}  
**public class** ArrayConstruction {  
 **int**[] **array** = **new int**[1500];  
  
 **public** ArrayConstruction() {  
 **for** (**int** i = **this**.**array**.**length** - 1; i >= 0; i--) {  
 **this**.**array**[i] = i;  
  
 }  
 }  
  
 **public void** create() {  
 **int** i;  
 **for** (i = 0; i < **array**.**length**; i++) {  
 **array**[i] = (**int**) (Math.*random*() \* 100);  
 }  
 System.***out***.println(**"Creation finished"**);  
 }  
  
 **public void** print() {  
 **int** i;  
 **for** (i = 0; i < **array**.**length**; i++) {  
 System.***out***.print(**array**[i] + **" "**);  
 }  
 }  
  
}  
**public class** PrintingThread **extends** Thread {  
 ArrayConstruction **arrConstr**;  
  
 **public** PrintingThread(ArrayConstruction arrConstr) {  
 **this**.**arrConstr** = arrConstr;  
 }  
  
 @Override  
 **public void** run() {  
 **arrConstr**.print();  
 }  
}  
**public class** SortingThread **extends** Thread {  
 **private** ArrayConstruction **arrConstr**;  
  
 **public** SortingThread(ArrayConstruction arrConstr) {  
 **this**.**arrConstr** = arrConstr;  
 }  
  
 @Override  
 **public void** run() {  
 Arrays.sort(**arrConstr**.**array**);  
 }  
  
}

При многократното изпълнението на този код са възможни три сценария: числата са подредени в реда на първоначалното им подредба, числата са частично сортирани и числата са напълно сортирани. Изходът на дадения код зависи от това коя нишка ще достъпи първа дадения общ ресурс(масива от случайни числа) и дали тя ще е преключила своята работа до момента, в който другата нишка достъпи същия ресурс. Това е основен проблем при многонишковото програмиране.  
Нека дедем същия пример, реализиран по различен начин.

**public class** Test {  
  
 **public static void** main(String[] args) {  
  
 CreateArr ob = **new** CreateArr();  
 **for** (**int** i = 0; i < ob.**array**.**length**; i++) {  
 ob.**array**[i] = (**int**) (Math.*random*() \* 100);  
 }  
 **for** (**int** i = 0; i < ob.**array**.**length**; i++) { System.***out***.print(**" "** + ob.**array**[i]); } System.***out***.println(**" After construction"**); PrintingThread obb1 = **new** PrintingThread(ob); SortingThread obb2 = **new** SortingThread(ob); obb1.start(); obb2.start(); } } **class** CreateArr { **int**[] **array** = **new int**[1000]; **public** CreateArr() { **for** (**int** i = **this**.**array**.**length** - 1; i >= 0; i--) {  
 **this**.**array**[i] = i;  
  
}  
}  
  
}  
  
**class** PrintingThread **extends** Thread {  
 CreateArr **ob**;  
  
 **public** PrintingThread(CreateArr ob) {  
 **this**.**ob** = ob;  
 }  
 @Override  
 **public void** run() {  
 print();  
 }  
  
 **public void** print() {  
 **int** i;  
 **for** (i = 0; i < **ob**.**array**.**length**; i++) {  
  
 System.***out***.print(**ob**.**array**[i] + **" "**);  
 }  
 }  
}  
  
**class** SortingThread **extends** Thread {  
 CreateArr **ob**;  
  
 **public** SortingThread(CreateArr ob) {  
 **this**.**ob** = ob;  
 }  
 @Override  
 **public void** run() {  
 sortArr();  
 }  
  
 **private void** sortArr(){  
 Arrays.sort(**ob**.**array**);  
 }  
}

Каква е разликата между двата подхода? В първия случай методът print() е в класа ArrayConstruction. Създаваме два обекта от класове, наследници на класа Thread, и подаваме като параметър обект от класа ArrayConstruction. По този начин в run() метода извикваме метода print() на подадения обект. При втория вариант методът print() е в класа PrintingThread, наследник на Thread.  
Основният въпрос, който възниква и в двата случая, е как да се справим с проблема две нишки да не използват един и същи ресурс едновременно?  
Отговорът на този въпрос е- чрез използването на синхронизирани методи или синхронизирани блокове код.  
Ето как биха изглеждали синхронизиран метод и синхронизиран блок от първия пример.  За втория случай е аналогично.

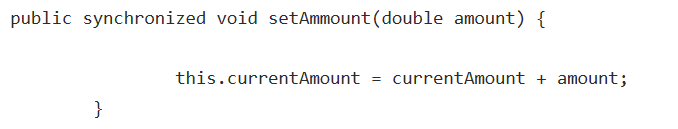




Нека да разгледаме класическия пример с банка, в която много клиенти внасят пари. Всеки един клиент представлява отделна нишка. При достъпване на общия ресурс (банкова сметка) едновременно на две или повече нишки, резултатът няма да бъде коректен. Нека покажем това с пример:

**public class** BANK {  
  
 **public static void** main(String[] args) **throws** InterruptedException {  
  
 **for** (**int** i = 1; i < 1000; i++) {  
 Account ob = **new** Account();  
 ClientThread ob1 = **new** ClientThread(ob, 100);  
 ClientThread ob2 = **new** ClientThread(ob, 200);  
 ClientThread ob3 = **new** ClientThread(ob, 300);  
 ClientThread ob4 = **new** ClientThread(ob, 400);  
 ClientThread ob5 = **new** ClientThread(ob, 500);  
 Thread T1 = **new** Thread(ob1);  
 Thread T2 = **new** Thread(ob2);  
 Thread T3 = **new** Thread(ob3);  
 Thread T4 = **new** Thread(ob4);  
 Thread T5 = **new** Thread(ob5);  
 Thread T6 = **new** Thread(ob5);  
 Thread T7 = **new** Thread(ob2);  
 Thread T8 = **new** Thread(ob3);  
 Thread T9 = **new** Thread(ob5);  
 T1.start();  
 T2.start();  
 T3.start();  
 T4.start();  
 T5.start();  
 T6.start();  
 T7.start();  
 T8.start();  
 T9.start();  
 Thread.*sleep*(100);  
 **if** (ob.getAmmount() != 3000)  
 System.***out***.println(ob.getAmmount());  
  
 }  
 System.***out***.println(**"finished"**);  
 }  
}  
  
**class** ClientThread **extends** Thread {  
 **private double amount**;  
 **private** Account **account**;  
  
 ClientThread(Account account, **double** amount) {  
 **this**.**account** = account;  
 **this**.**amount** = amount;  
  
 }  
  
 @Override  
 **public void** run() {  
 **account**.setAmmount(**amount**);  
 }  
  
}  
  
**class** Account {  
  
 **private double currentAmount** = 0;  
  
 **public void** setAmmount(**double** amount) {  
  
 **this**.**currentAmount** = **currentAmount** + amount;  
 }  
  
 **double** getAmmount() {  
 **return currentAmount**;  
  
 }  
}

В дадения код се стартират девет нишки(клиента), които внасят различни суми. Общата сума в дадения случай е 3000. Ако резултатът е различен от коректния, то той ще бъде показан на конзолата. За целта на експеримента са направени 1000 итерации.  
Отново възниква въпроса как да решим дадения проблем? Отговорът отново е чрез синхронизиране на метода setAmmount.



Както се вижда от показаните примери синхронизацията е важна част от многонишковото програмиране за коректната работа на дадена програма. Важно е да се знае, че от друга страна не трябва да се прекалява със синхронизирането на методи, защото това нарушава принципа на конкурентното програмиране, което е основа на многонишковото програмиране. Добра практика е да не се синхронизират методи, а отделни блокове с код.

**Задача за изпълнение**: да се напише програма клиент-сървър, в която отделни клиенти имат право да четат и пишат в определен текстов файл.