CAPITOLUL 1

1. Platformă online de găzduire web GIT.

Git este un sistem de control al versiunilor distribuit gratuit, open-source cod sursă, conceput pentru a lucra rapid și eficient cu orice proiecte - de la mici la foarte mari. Git este ușor de învățat și ocupă puțin spațiu și are performanțe fulgerătoare. Este superior unor astfel de SCM-uri (Supply Managementul lanțului – trad. Supply Chain Management - instrumente precum Subversion, CVS, Perforce și ClearCase, cu caracteristici precum local low-cost ramificații, zone de depozitare convenabile și fluxuri de lucru multiple.

1.1 Ramificare și contopire

Caracteristica Git care îl face cu adevărat să iasă în evidență de aproape toți ceilalți alte SCM-uri, este modelul de ramificare. Git permite și încurajează creația mai multe ramuri locale, care pot fi complet independente unele de altele prieten. Crearea, îmbinarea și ștergerea acestor ramuri durează doar câteva secunde.

Aceasta înseamnă că puteți face lucruri precum:

• Comutare de context fără contact. Creați o ramură pe care să o încercați orice idee, fă câteva comite, întoarce-te la locul de unde ramificație a fost făcută, aplicați remedierea, întoarceți-vă unde au fost efectuate experimente și combinate.

• Linii de cod pentru jocuri de rol. Ai o ramură care conține întotdeauna doar ce trimis la producție, altul, în care munca este fuzionată pentru testare și câteva ramuri mici pentru munca de zi cu zi.

• Flux de lucru bazat pe funcții. Creați filiale noi pentru fiecare funcție nouă la care lucrați, astfel încât să puteți face fără probleme comuta între ele și apoi șterge fiecare ramură când funcția vor fi îmbinate cu linia principală.

• Experimente unice. Creați un fir pentru experimentare, înțelegeți asta nu funcționează și șterge-l, părăsește-ți jobul și nimeni nu o va mai vedea ea (chiar dacă în acest timp ai promovat alte ramuri).

1.2 Distribuire

Una dintre cele mai utile caracteristici ale oricărui SCM distribuit este inclusiv Git, este distribuția sa. Aceasta înseamnă că în loc de efectuați o „verificare” a vârfului codului sursă curent, efectuați o „clonă” întregul depozit.

1.3 Copii de rezervă multiple

Aceasta înseamnă că, chiar dacă utilizați un flux de lucru centralizat, fiecare utilizator are în esență o copie de rezervă completă a serverului principal.

Fiecare dintre aceste copii poate fi ridicată pentru a înlocui serverul principal în caz de defecțiune sau deteriorare. În esență, nu există un singur punct de eșec în Git decât dacă există o singură copie a depozitului.

1.4 Flux de lucru

Datorită naturii distribuite a lui Git și sistemului excelent de ramificare este posibil să se implementeze cu relativă uşurință un număr practic infinit procesele de lucru.

1.5 Flux de lucru în stil subversiune (Subversion)

Fluxul de lucru centralizat este foarte comun, mai ales în rândul utilizatorilor, care au trecerea dintr-un sistem centralizat. Git nu te va lăsa să realizați push dacă cineva a făcut un push de la ultima extracție, deci centralizat funcționează modelul în care toți dezvoltatorii realizează push pe același server pur si simplu minunat.

1.6 Instalarea Git

Descărcați programul de instalare (https://git-scm.com/downloads), rulați programul de instalare,selectați opțiunile de care aveți nevoie (sau lăsați totul așa cum este, dacă nu prea mult înțelegeți), așteptați finalizarea procesului de instalare și ați terminat.

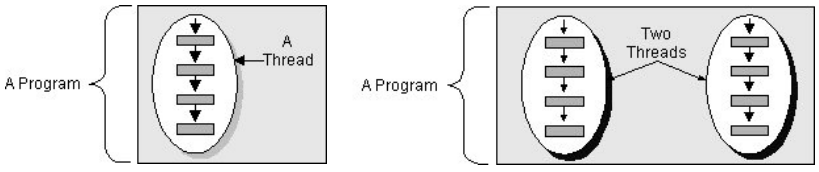
1.7 Utilizare Git

În folderul de lucru în care se află fişierele sursă ale aplicaţiei/proiectului, prin terminal sau linia de comandă, utilizaţi comanda git init (documentaţia oficială - https://git-scm.com/docs/git-init) pentru a iniţializa depozitul local, după ce Git va urmări modificările la fişierele sursă şi le va remedia modificările - utilizaţi comanda git commit (documentaţie oficială - https://git scm.com/docs/git-commit).
În viitor, dacă trebuie să sincronizaţi modificările în comun pentru toate depozitele - utilizaţi comanda git push

2. Fire de execuţie

2.1. Ce este un fir de execuţie?

Firele de execuţie realizează trecerea de la programarea secvenţială la programarea concurentă. Un program secvenţial reprezintă modelul clasic de program: are un început, o secvenţă de execuţie a instrucţiunilor sale şi un sfârşit. Cu alte cuvinte, la un moment dat programul are un singur punct de execuţie. Un program aflat în execuţie se numeste *proces*. Un sistem de operare monotasking (MS-DOS) execută numai un singur proces la un moment dat, în timp ce un sistem de operare multitasking (UNIX, Windows) poate rula o mulţime de procese în acelaşi timp (concurent), alocând periodic cuante din timpul de lucru al CPU fiecărui proces. Deoarece noţiunea de fir de execuţie nu are sens decât în cadrul unui sistem de operare multitasking. Un fir de execuţie este similar unui proces secvenţial –are un început, o secvenţă de execuţie şi un sfârşit. Diferenţa între un fir de execuţie şi un proces constă în faptul că un fir de execuţie nu poate rula independent, ci trebuie să ruleze în cadrul unui proces.



a) b)

Figura.1 Fire de execuţie în cadrul unui program
a) Program cu un fir de execuţie;
b) Program cu două fire de execuţie

În Figura. 1 este indicat locul firelor de execuţie în cadrul programului.

Definiţie : Un *fir de execuţie* este o succesiune secvenţială de instrucţiuni care se execută în cadrul unui proces.

Într-un program pot fi definite cîteva fire de execuţie, ceea ce înseamnă că în cadrul unui proces se pot executa simultan mai multe fire de execuţie, aceasta permite execuţia concurentă a sarcinilor independente ale acestui program. Un fir de execuţie poate fi asemănat cu o versiune redusă a unui proces, ambele rulând simultan şi independent pe o structură secvenţială de execuţie a instrucţiunilor sale. De asemenea execuţia simultană a firelor de execuţie în cadrul unui proces este similară cu execuţia concurentă a proceselor: sistemul de operare va aloca ciclic cuante din timpul procesorului fiecărui fir de execuţie până la terminarea lor. Din acest motiv firele de execuţie mai sunt numite şi *procese uşoare*. Deosebirea majoră dintre un fir de execuţie şi un proces constă în faptul că firele de execuţie pot rula în cadrul unui proces. O altă deosebire rezultă din faptul că fiecare proces are propria sa memorie (propriul său spaţiu de adrese), iar la crearea unui nou proces (fork) este realizată o copie exactă a procesului părinte : cod + date; la crearea unui fir de execuţie este copiat doar codul procesului părinte; toate firele de execuţie au acces la aceleaşi date, ale procesului original. Astfel, un fir de execuţie poate fi privit şi ca un *context de execuţie* în cadrul unui proces părinte. Firele de execuţie sunt utile în multe privinţe, însă uzual ele sunt folosite pentru executarea unor operaţii consumatoare de timp fără a bloca procesul principal: calcule matematice, aşteptarea eliberării unei resurse, acestea realizându-se de obicei pe fundal.

2.2. Concurenţa thread-urilor

Reieşind din particularităţile hardware, este bine de spus că procesoarele pot executa doar o instrucţiune la un moment dat. Pe o maşină multiprocesor, thread-urile se executa în procesoare diferite

în acelaşi timp fizic, chiar dacă procesoarele sunt pe calculatoare diferite conectate într-o rețea. Dar şi pe o maşină cu un singur procesor, thread-urile pot ”împărți” acelaşi procesor, rulând într-o manieră întrețesută, competiția pentru timpul CPU crează iluzia că ele se execută simultan. Această iluzie pare reală când 30 de imagini distincte pe secundă captate de ochiul uman sunt percepute într-un flux continuu de imagine. Comutarea între thread-uri are şi ea un ”preț”: consumă timp CPU pentru ca acesta să înghețe starea unui thread şi să dezgheț starea unui alt thread (schimbare de context). Dacă thread-urile concurente sunt executate pe acelaşi procesor şi toate execută calcule, atunci timpul total de execuție nu va lua mai mult decât timpul de execuție al unui program secvențial care realizează acelaşi lucru. Creşterea vitezei sistemului se realizează prin intersectarea diferitelor faze ale diferitelor thread-urilor. Multe task-uri pot fi segmentate logic în tipuri de faze: faza de calcul şi faza I/O.

Faza de calcul necesită atenția maxima din partea CPU prin utilizarea diferitelor metode de calcul. Faza de I/O (intrare/ieşire) necesită atenție maximă din partea perifericelor (imprimantei, hard discului, plăcii de rețea, etc) şi în aceste situații procesorul este în general liber, aşteptând ca perifericul să-şi termine sarcina. Creşterea vitezei este obținută prin intersectarea fazelor. În timp ce un thread se află într-o fază de I/O aşteptând ca o secvență de date să fie încărcată de pe hard disk, un thread cu o fază de calcul poate ocupa procesorul şi când ajunge la o fază I/O, celălalt thread (care tocmai a terminat faza I/O proprie) poate începe să utilizeze CPU-ul.

2.3. Crearea firelor de execuție

Ca orice alt obiect în Java, un fir de execuție este o instanță a unei clase. Firele de execuție definite de o clasă vor avea acelaşi cod, prin urmare, şi aceeaşi secvență de instrucțiuni. Crearea unei clase, care să definească fire de excuție poate fi făcută prin două modalități:

- **prin extinderea clasei Thread**
- **prin implementarea interfeței Runnable**

Orice clasă ale cărei instanțe vor fi executate într-un fir de execuție trebuie declarată ca fiind Runnable. Aceasta este o interfață care conține o singură metodă, şi anume metoda ”run”. Asadar, orice clasă ce descrie firele de execuție va conține o metodă **run()** în care este implementat codul, ce va fi executat de firul de execuție. Interfața Runnable este concepută ca fiind un protocol comun pentru obiecte, care doresc să execute un cod pe durata existenței lor (care reprezintă fire de execuție). Cea mai importantă clasă care implementează interfața Runnable este clasa Thread. Clasa Thread implementează un fir de execuție generic care, implicit, nu face nimic. Cu alte cuvinte, metoda run nu conține nici un cod. Orice fir de execuție este o instanță a clasei Thread sau a unei subclase a acesteia.

2.3.1 Extinderea clasei Thread

Cea mai simplă metodă de a crea un fir de execuție, care să realizeze ceva, este extinderea clasei Thread şi supraîncărcarea metodei run a acesteia. Formatul general al unei astfel de clase este:

```
public class SimpleThread extends Thread {
   public SimpleThread(String nume) {
     super(nume);
    //apelez constructorul superclasei Thread
   }

   public void run() {
   //codul executat de firul de execuție
   }
}
```

Prima metodă a clasei este constructorul, care primeşte ca argument un şir ce va reprezenta numele firului de execuție creat în momentul când constructorul este apelat.

```
SimpleThread t = new SimpleThread("Java");
//creeaza un fir de execuţie cu numele Java
```

În cazul în care nu dorim să dam nume firelor de execuție pe care le creăm, putem renunța la definirea acestui constructor şi să lasăm doar cu constructorul implicit, fără argumente, care creează un fir de execuție fără nici un nume. Ulterior acesta poate fi numit utilizînd metoda setName(String).

Se pot defini şi alți constructori, aceştia fiind utili când vrem să trimitem diverşi parametri firului de execuție.

A doua metodă este metoda run, "inima" oricărui fir de execuție în care scriem efectiv codul pe care trebuie să-l execute firul de execuție. Un fir de execuție creat nu este automat pornit, lansarea lui în execuție se realizează prin metoda start, definită de asemenea în clasa Thread.

```
SimpleThread t = new SimpleThread("Java");
t.start();
//creează şi lansează un fir de execuție
```

Să considerăm în continuare un exemplu în care definim un fir de execuție ce afişează numerele întregi dintr-un interval cu un anumit pas. Firul de execuție este implementat de clasa Counter.

```
class Counter extends Thread {
  //clasa care defineşte firul de execuție
  private int from, to, step;

  public Counter(int from, int to, int step) {
        this.from = from;
        this.to = to;
        this.step = step;
  }

  public void run() {
        for(int i = from; i <= to; i += step)
              System.out.print(i + " " );
  }
}

public class TestCounter { //clasa principală
  public static void main(String args[]) {
        Counter cnt1, cnt2;

        cnt1 = new Counter(0, 10, 2);
        //numară de la 0 la 100 cu pasul 5

        cnt2 = new Counter(100, 200, 10);
        //numară de la 100 la 200 cu pasul 10

        cnt1.start();
        cnt2.start();
//pornim firele de execuție
//ele vor fi distruse automat la terminarea lor
  }
}
```

Execuția acestui program in mod secvențial va afişa prima data numerele de la 0 la 100 cu pasul 5, apoi numerele de la 100 la 200 cu pasul 10, întrucât primul apel este către contorul cnt1, deci rezultatul afişat pe ecran ar trebui să fie: 0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100 100 110 120 130 140 150 160 170 180 190 200.

În realitate însă, rezultatul obținut va fi o intercalare de valori produse de cele două fire de execuție ce rulează simultan. La rulări diferite se pot obține rezultate diferite deoarece timpul alocat fiecărui fir de execuție poate să nu fie acelaşi, el fiind controlat de procesor într-o manieră "aparent" aleatoare: 0 100 5 110 10 120 15 130 20 140 25 150 160 170 180 190 200 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100…

2.3.2. Implementarea interfeței Runnable

Ce facem însă când dorim să creăm o clasă care instanțiază fire de execuție, iar aceasta are deja o superclasă, ştiind că în Java nu este permisă moştenirea multiplă?

class FirExecuţie extends Părinte, Thread //ilegal !

În acest caz nu mai putem extinde clasa Thread, ci trebuie să implementăm direct în clasa noastră interfața Runnable. Clasa Thread implementeaza interfața Runnable şi, din acest motiv, la extinderea ei obținem o implementare implicită a interfeței. Asadar, interfața Runnable permite unei clase să fie active, fără a extinde clasa Thread.

Interfața Runnable se găseşte în pachetul java.lang şi este definită astfel:

```
public interface Runnable {
        pulic abstract void run( );
}
```

Prin urmare, o clasă care instanțiază fire de execuție prin implementarea interfeței Runnable trebuie obligatoriu să implementeze metoda run. Formatul general al unei clase care implementează interfața Runnable este:

```
public class SimpleThread implements Runnable {

  private Thread simpleThread = null;

  public SimpleThread() {
        if (simpleThread == null) {
        simpleThread = new Thread(this);
              simpleThread.start();
        }
        public void run() {
        //codul executat de firul de execuţie
  }
}
```

Spre deosebire de abordarea anterioară, se pierde tot suportul oferit de clasa Thread, pentu crearea unui fir de execuție. Simpla instanțiere a unei clase care implementează interfața Runnable nu creează nici un fir de execuție. Din acest motiv crearea firelor de execuție prin instanțierea unei astfel de clase trebuie făcută explicit. Aceasta se face prin declararea unui obiect de tip Thread, ca variabilă membră a clasei respective. Acest obiect va reprezenta firul de execuție propriu-zis al cărui cod se găseşte în clasa:

private Thread simpleThread = null;

Următorul pas este instanțierea şi inițializarea firului de execuție. Acesta se realizează prin instrucțiunea new, urmată de un apel la constructorul, care primeaşte drept argument o instanță a clasei. După creare, firul de execuție poate fi lansat printr-un apel la metoda start. (Aceste operațiuni sunt scrise de obicei în constructorul clasei, pentru a fi executate la inițializarea unei instanțe, dar pot fi scrise oriunde în corpul clasei sau chiar în afara ei)

```
simpleThread = new Thread( this );
simpleThread.start();
```

Specificarea argumentului **this** în constructorul clasei Thread, determină crearea unui fir de execuție, care la lansarea sa va căuta în clasa noastră metoda run şi o va executa. Acest constructor acceptă ca argument orice instanță a unei clase "Runnable". Asadar, metoda run nu trebuie apelată explicit, cea ce se realizează automat la apelul metodei **start()**.

Apelul explicit al metodei run nu va furniza nici o eroare, însă aceasta va fi executată ca orice altă metodă, deci nu într-un fir de execuție. Sa rescriem acum exemplul anterior (afişarea numerelor întregi dintr-un interval cu un anumit pas), folosind interfața Runnable. Observaţi că, implementarea interfeței Runnable permite o flexibilitate sporită în lucrul cu fire de execuție.

Varianta 1 (standard)

Crearea firului de execuţie se realizează în constructorul clasei Counter:

```
class Counter implements Runnable {
  private Thread counterThread = null;
  private int from, to, step;
  public Counter(int from, int to, int step) {
      this.from = from;
      this.to = to;
      this.step = step;
      if (counterThread == null) {
         counterThread = new Thread(this);
         counterThread.start();
      }
  }
  public void run() {
     for(int i = from; i <= to; i += step)
        System.out.print(i + " " );
  }
}
public class TestThread2 {
  public static void main(String args[]) {
        Counter cnt1, cnt2;
//lansez primul fir de execuţie (prin constructor)
        cnt1 = new Counter(0, 100, 5);
//lansez al doilea fir de execuţie (prin constructor)
        cnt2 = new Counter(100, 200, 10);
  }
}
```

Varianta 2

Crearea firului de execuţie se realizează în afara clasei Counter:

```
class Counter implements Runnable {
  private int from, to, step;
  public Counter(int from, int to, int step) {
        this.from = from;
        this.to = to;
        this.step = step;
  }
  public void run() {
        for(int i = from; i <= to; i += step)
              System.out.print(i + " " );
  }
}
public class TestThread2 {
  public static void main(String args[]) {
        Counter cnt1, cnt2;
        cnt1 = new Counter(0, 100, 5);
        cnt2 = new Counter(100, 200, 10);
        new Thread( cnt1 ).start();
        //lansez primul fir de execuţie
        new Thread( cnt2 ).start();
        //lansez al doilea fir de execuţie
}}
```

2.4. Ciclu de viață al unui fir de execuție

Fiecare fir de execuție are propriul său ciclu de viață: este creat, devine activ prin lansarea sa în execuție şi, la un moment dat, se sfîrşeşte. În continuare vom descrie, mai detaliat, stările în care se poate găsi un fir de execuție. Diagrama ce ilustează generic aceste stări precum şi metodele care provoacă tranziția dintr-o stare în alta este ilustrată în figura 2:

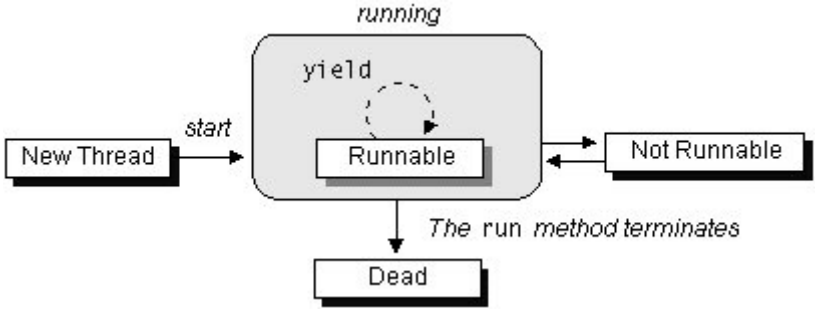


Figura.2 Stările firului de execuție

Aşadar, un fir de execuție se poate găsi în una din următoarele patru stări:

1. Starea "New Thread"

Un fir de execuție se găseşte în această stare imediat după crearea sa, cu alte cuvinte după instanțierea unui obiect din clasa Thread sau dintr-o subclasă a sa.

```
Thread counterThread = new Thread ( this );
//counterThread se găseşte în starea New Thread
```

2. Starea "Runnable"

După apelul metodei start un fir de execuție va trece în starea "Runnable", adică se găseşte în execuție.

```
counterThread.start();
//counterThread se găseşte în starea Runnable
```

Metoda start realizează urmatoarele operațiuni necesare rulării firului de execuție: alocă resursele sistem necesare, ordonează firul de execuție în fir de aşteptare la CPU pentru a fi lansat, apelează metoda run a obiectului reprezentat de firul de execuție.

3. Starea "Not Runnable"

Un fir de execuție ajunge în această stare în una din urmatoarele situații: este "adormit" prin apelul metodei **sleep;** a apelat metoda **wait**, aşteptând ca o anumită condiție să fie satisfăcută; este blocat într-o operație de intrare/ieşire.

"Adormirea" unui fir de execuție

Metoda **sleep** este o metodă statică a clasei Thread care provoacă o pauză în timpul rulării firului curent aflat în execuție, cu alte cuvinte îl "adoarme" pentru un timp specificat. Lungimea acestei pauze este specificată în milisecunde şi chiar nanosecunde.

```
public static void sleep( long millis )
throws InterruptedException
  public static void sleep( long millis, int nanos )
throws InterruptedException
```

Întrucât poate provoca excepții de tipul InterruptedException apelul acestei metode se face într-un bloc de tip try-catch:

```
try {
      Thread.sleep(1000);
      //face pauză de o secundă
    } catch (InterruptedException e) {
            . . .
    }.
```

4. Starea "Dead"

Este starea în care ajunge un fir de execuție la terminarea sa. Un fir de execuție nu poate fi oprit din program printr-o anumită metodă, ci trebuie să se termine în mod natural la terminarea metodei run pe care o execută. Terminarea unui fir de execuție folosind o variabilă de terminare:

```
import java.io.*;
public class TestThread {
   public static void main(String args[]) throws
        IOException {
      WaitKey thread = new WaitKey();
      thread.start();
      System.in.read();//astept apăsarea tastei Enter
      thread.running = false;
      System.out.println("S-au scurs " + thread.sec
                                        + " secunde");
  }
}
class WaitKey extends Thread {
   public int sec = 0;
   public boolean running = true;
   //variabila de terminare
public void run() {
  while ( running ) {
        try {
           Thread.sleep(1000);
           sec ++;
           } catch(InterruptedException e){
 }}}}
```

Lucrare de laborator 1

Tema lucrării: Crearea thread-urilor

Obiectivele lucrării:

- Înțelegerea conceptului de thread și diferența dintre procese și fire de execuție.
- Familiarizarea cu metodele de creare și gestionare a thread-urilor în limbajul de programare utilizat.
- Exersarea tehnicilor de comunicare între thread-uri.
- Evaluarea avantajelor și dezavantajelor programării multithread.
- Formarea deprinderilor practice pentru scrierea programelor paralele și concurrente.

Scopul lucrării:

- Înțelegerea principiilor programării concurrente.
- Exersarea modului de creare și gestionare a thread-urilor în Java.
- Formarea abilităților practice pentru dezvoltarea aplicațiilor multithread eficiente și sigure.

Exemplu de realizare:

```
import java.util.*;
class Counter1 extends Thread
{
private int from, to, step;
        private  int[] tablou;

public Counter1(int from, int to, int step, int[] tablou) {
      this.from = from;
      this.to = to;
      this.step = step;
```

```
            this.tablou = tablou;
      }

      public void run() {
                     int s1=0, s2=0, s=0;
                     int i=from;

                          while(i!=to){
                              if(tablou[i]<=50){
                               s1=i;
                               i+=step;
                               do{
                                if(tablou[i]<=50){
                                s2=i;
                                s=s1+s2;
       System.out.println(getName()+" " + s1+ " " + s2 +" "+s+ " "+ tablou[s1]+"
"+tablou[s2] );
                                break;
                                }
                                i+=step;
                               }while(true);

                              // i+=step;
                               }
                                i+=step;
                           }
                        }
                     }

            public class Main {
      public static void main(String args[])
      {

                     Counter1 cnt1, cnt2;

                     int[] tablou = new int[101];

                     for(int i=0; i<100; i++){
        tablou[i] = (int)(Math.random()*99);
                          System.out.print(tablou[i]+" ");
                     }
                     System.out.println(" ");

      cnt1 = new Counter1(0, 99, 1, tablou);

      cnt2 = new Counter1(99, 0, -1, tablou);

                cnt1.start();
                        cnt1.setName("Unu");
              cnt2.start();
                       cnt2.setName("Doi");

      }}
```

Rezultatul realuzării:

65 90 74 92 40 41 53 27 88 34 52 70 37 17 0 75 42 56 90 46 37 11 50 77 81 20 49 19 86 65 36 17 46 31 14 87 34 72 29 93 91 84 1 47 98 22 45 41 57 84 70 44 23 12 47 46 6 4 56 81 2 48 11 42 80 54 51 86 16 67 69 36 59 90 9 84 94 69 41 88 27 57 45 33 84 36 10 60 68 70 54 18 26 75 18 81 68 81 80 22

Doi 99 94 193 22 18
Unu 4 5 9 40 41
Doi 92 91 183 26 18
Unu 7 9 16 27 34
Doi 86 85 171 10 36
Unu 12 13 25 37 17
Doi 83 82 165 33 45
Unu 14 16 30 0 42
Doi 80 78 158 27 41
Unu 19 20 39 46 37
Doi 74 71 145 9 36
Unu 21 22 43 11 50
Doi 68 63 131 16 42
Unu 25 26 51 20 49
Unu 27 30 57 19 36
Unu 31 32 63 17 46
Doi 62 61 123 11 48
Unu 33 34 67 31 14
Doi 60 57 117 2 4
Unu 36 38 74 34 29
Doi 56 55 111 6 46
Unu 42 43 85 1 47
Doi 54 53 107 47 12
Unu 45 46 91 22 45
Doi 52 51 103 23 44
Unu 47 51 98 41 44
Doi 47 46 93 41 45
Unu 52 53 105 23 12
Unu 54 55 109 47 46
Doi 45 43 88 22 47
Unu 56 57 113 6 4
Doi 42 38 80 1 29
Unu 60 61 121 2 48
Doi 36 34 70 34 14
Unu 62 63 125 11 42
Doi 33 32 65 31 46
Unu 68 71 139 16 36
Doi 31 30 61 17 36
Unu 74 78 152 9 41
Doi 27 26 53 19 49
Unu 80 82 162 27 45
Doi 25 22 47 20 50
Doi 21 20 41 11 37
Unu 83 85 168 33 36
Doi 19 16 35 46 42
Unu 86 91 177 10 18
Doi 14 13 27 0 17
Unu 92 94 186 26 18
Doi 12 9 21 37 34
Doi 7 5 12 27 41
BUILD SUCCESSFUL (total time: 1 second)

Etapele de realizare a lucrării:

1. Studenții grupei se divizează în echipe a câte doi. Lucrarea practică va fi realizată în echiră.
2. Creați un cont pe GitHub (https://github.com/) (dacă nu aveți unul) și trimiteți numele de utilizator la linkul de profil.
3. Instalați mediul Git pe dispozitivul dvs., pregătiți un folder pentru aplicație și inițializați depozitul local din acest folder (comanda git init).
4. Implementați o aplicație simplă (din ***Sarcina lucrării***). În echipă decideți cine ce părți a aplicației va implementa.
5. După ce ați primit instrucțiuni de la profesor, introduceți modificările în ramura dvs. separată de pe GitHub. Efectuați commit pentru modificări în codul aplicației (comanda git commit).
6. La finalizarea lucrării, elaborați un raport care trebuie să conțină - numele, prenumele, grupa, sarcina și opțiunea dvs. pentru implementarea sarcinii, scurtă descriere, un link către codul sursă de pe GitHub. Salvați raportul în format PDF sau WORD și trimiteți pe ELSE.

Sarcina lucrării:

Problema:

Fiecare membru a echipei creează câte o clasă cu doua fire de execuție. Primul fir Th1 va afișa: Condiție 1 din tabelul 1. Al doilea fir Th2 va afișa: Condiție 2 din tabelul 1. Toate patru fire de execuție vor citi datele din același tablou de date mas[] de tipul întreg, generat aleatoriu cu dimensiunea 100 și cuprinde valori intre 1 și 100. Apoi după terminarea ambelor fire de execuție thread-ul principal va afișa informația despre studenții care au efectuat lucrarea dată de laborator, literele textului vor apărea pe ecran cu un interval de 100 milisecunde.

Tabelul 1 Condițiile pentru realizarea sarcinii problemei conform variantelor

Nr/var	Condiția 1	Condiția 2
1	Sumele numerelor pare două câte două începând căutarea şi sumarea de la primul element	Sumele numerelor pare două câte două începând căutarea şi sumarea de la la ultimul element
2	Suma pozițiilor numerelor pare două câte două începând căutarea şi sumarea de la primul element	Suma pozițiilor numerelor pare două câte două începând căutarea şi sumarea de la ultimul element
3	Sumele numerelor impare două cate două începând căutarea şi sumarea de la primul element	Sumele numerelor impare două cate două începând căutarea şi sumarea de la ultimul element
4	Sumele pozițiilor numerelor impare două câte două începând căutarea şi sumarea de la primul element	Sumele pozițiilor numerelor impare două câte două începând căutarea şi sumarea de la ultimul element
5	Sumele produselor numerelor de pepoziții pare două câte două începând căutarea și sumarea cu primul element	Sumele produselor numerelor de pepoziții pare două câte două începând căutarea și sumarea cu ultimul element
6	Sumele produselor numerelor de pe poziții impare două câte două începând căutarea și sumarea cu primul element	Sumele produselor numerelor de pe poziții impare două câte două începând căutarea și sumarea cu ultimul element
7	Sumele produselor numerelor pare două câte două începând căutarea și sumarea cu primul element	Sumele produselor numerelor pare două câte două începând căutarea și sumarea cu ultimul element
8	Sumele produselor numerelor impare două câte două începând căutarea și sumarea cu primul element	Sumele produselor numerelor impare două câte două începând căutarea și sumarea cu ultimul element
9	Diferența produselor numerelor de pepoziții pare două câte două începând căutarea și sumarea cu primul element	Diferența produselor numerelor de pepoziții pare două câte două începând căutarea și sumarea cu ultimul element
10	Diderența produselor numerelor de pe poziții impare două câte două începând căutarea și sumarea cu primul element	Diderența produselor numerelor de pe poziții impare două câte două începând căutarea și sumarea cu ultimul element
11	Diferența produselor numerelor pare două câte două începând căutarea și sumarea cu primul element	Sumele produselor numerelor pare două câte două începând căutarea și sumarea cu ultimul element
12	Sumele produselor numerelor impare două câte două începând căutarea și sumarea cu primul element	Sumele produselor numerelor impare două câte două începând căutarea și sumarea cu ultimul element

Criterii de evaluare:

1. Înregistrarea pe GitHub și crearea a inui GIT local pe calculatorul gazdă.
2. Crearea şi inițializarea thread-urilor.
3. Crearea thread-urilor prin interfața Runnable.
4. Crearea interfeței grafice a programului.
5. Corectitudinea codului - verificarea dacă codul este corect, fără erori de funcționare.
6. Respectarea instrucțiunilor și cerințelor - verificarea corectitudenii cerințelor sarcinii, cum ar fi numărul de fire de execuție.
7. Optimizarea codului - Evaluarea eficienței codului în utilizarea resurselor și evitarea codului.

8. Respectarea termenului de susținere - evaluarea punctajului în funcție de punctualitate, dacă lucrarea a fost predată în termenul stabilit.

9. Evaluarea cunoștințelor - explicațiile oferite despre procesul de realizare a lucrării, ceea ce poate include descrierea funcțiilor principale și a logicii utilizate.

10. Utilizarea de către student a IA.

Pentru obținerea notei 5 - 6 sunt obligatorii criteriile 1,2,5,8,9

Pentru obținerea notei 7 - 8 sunt obligatorii criteriile 1,2,3,5,6,8,9

Pentru obținerea notei 9 - 10 sunt obligatorii criteriile 1-9

Dacă a fost utilizat criteriul 10 nota este scăzută cu 2 baluri, numai dacă studentul sa lămurit în funcționarea codului. În caz contrar lucrare de laborator nu este susținută.

***Întrebări de verificare*:**

1. Definiți un fir de execuție?
2. Numiți modalitățile de creare a firelor de execuție?
3. Cum poate fi definită și implementate interfața Runnable?
4. Numiți stările unui fir de execuție?
5. Ce este un thread și cum diferă de un program?
6. Explicați ce este o secțiune critică.

Lista de literatură recomandată:

1. Schildt, H. – *Java: Ghid complet*. Editura Teora, București, 2012.
2. Deitel, P. J., Deitel, H. M. – *Java – Cum să programezi*. Editura Teora, București, 2008.
3. Pătrașcu, V. – *Programarea concurentă și paralelă*. Editura Universității din București, 2015.
4. Stănculescu, A. – *Structuri de date și algoritmi în Java*. Editura Polirom, Iași, 2016.
5. Tanenbaum, A. S. – *Sisteme de operare moderne*. Traducere în limba română, Editura Polirom, 2010.
6. Шилдт, Г. – *Java. Полное руководство*. Издательство Вильямс, Москва, 2021.
7. Гослинг, Дж., Арнольд, К., Холмс, Д. – *Язык программирования Java*. СПб: Питер, 2018.
8. Дейтел, П., Дейтел, Х. – *Java. Как программировать*. Москва: Вильямс, 2016.
9. Гамма, Э., Хелм, Р., Джонсон, Р., Влиссидес, Дж. – *Приёмы объектно-ориентированного проектирования. Паттерны проектирования*. СПб: Питер, 2019.