Математичка гимназија

МАТУРСКИ РАД

-из предмета Програмирање и програмски језици-

Умрежавање рачунара и нитно програмирање у програмском језику Јава

Ученик:
Богдана Колић 4А

Ментор:
Снежана Јелић

Београд, Јун 2021.

Садржај

1 Увод **5**
2 Умрежавање рачунара **7**
2.1 Основе мрежног програмирања 7
2.2 Повезивање клијента са сервером 9
2.3 Конфигурација сервера 10
2.4 Размена података између клијнета и сервера 13
3 Нитно програмирање **17**
3.1 Основни појмови, класа Thread и интерфејс Runnable 17
3.2 Блокирање нити 20
3.3 Синхронизација нити – синхронизоване методе 23
3.4 Синхронизоване наредбе – синхронизација по објекту 26
3.5 Мртва петља 27
3.6 Методе wait(), notify() и notifyAll() 27
4 Примене нитног програмирања при умрежавању рачунара **31**
5 Закључак **35**
Литература **37**

1

Увод

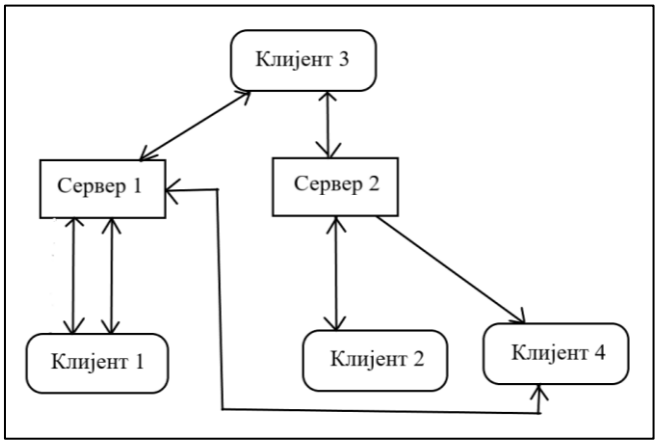
Друштвене мреже, мобилни телефони, разне технологије представљају неизоставан део свакодневног живота. Главни разлог за то је што нам омогућавају да будемо повезани са другим људима, да будемо „умрежени“. Мени су часови рачунарства увек били занимљиви и није ми било тешко да се определим за матурски рад баш из овог предмета. Имајући у виду да се у школи умрежавање рачунара обрађивало само на теоријском нивоу, одлучила сам да свој рад посветим управо примени тих концепата у пракси и да читаоцима приближим начин на који сам комбинујући знање стечено на часовима рачунрства успела да испрограмирам апликацију која служи за међусобну интеракцију корисника. Апликација је рађена у програмском језику Јава, тако да и овај рад представља водич за мрежно програмирање у Јави. Поред тога, обрађене су и основе нитног програмирања, чијом применом се отварају разне могућности за унапређивање основних програма за умрежавање.

2

Умрежавање рачунара

2.1 Основе умрежавања рачунара

Рачунари се у програмском језику Јава повезују помоћу **клијентских** и **серверских** програма. Свака засебна веза остварује се искључиво између једног сервера и једног клијента, али се на један сервер може повезати више клијената, као што и један клијентски програм може да се повеже на неколико серверских. Могуће је, чак, остварити и вишеструку конекцију између истог клијентског и серверског програма. На сваком крају везе налазе се **сокети (енгл. Socket)** који омогућавају њихову комуникацију. Како се на један сервер може повезати више клијената, могуће је међусобно повезати и више од два рачунара(слика1). Сви они биће директно повезани на исти сервер, али ће такође и сви бити међусобно умрежени његовим посредством. Битно је још истаћи да се клијент повезује са сервером, а никако обнуто, и зато је при комуникацији више рачунара управо сервер посредник.

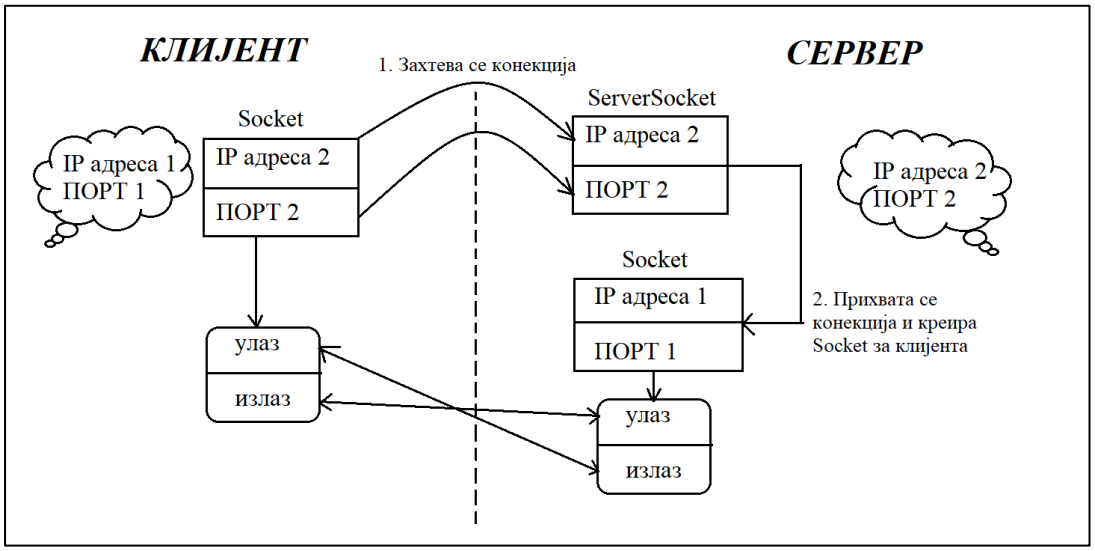


слика 1

За разумевање начина на који се остварује веза, потребно је познавање концепата IP адресе и портова. **IP адеса** је логичка адреса сваког рачунара који је повезан на интернет. Постоје две верзије, IPv4 и IPv6 адреса. За повезивање рачунара сокетима у Јави користи се IPv4 адреса. Она може бити глобална и локална, а постоји и такозвана loopback адреса (127.0.0.1). Сваки рачунар има и локалну и глобалну IP адресу. Глобална адреса је идентична за све рачунаре који су повезани на исту мрежу,

али је јединствена за сваку мрежу. Локална адреса служи за разликовање рачунара повезаних на исту мрежу. Дакле, два рачунара који се налазе на истој мрежи сигурно имају различите локалне, а исте глобалне IP адресе. С друге стране, два рачунара на различитим мрежама могу (али не морају) имати исте локалне IP адресе. Loopback адреса за рачунар само уопштено представља његову логичку адресу. Локалне IP адресе такође се могу поделити на статичке и динамичке, у зависности од тога да ли се мењају при поновном повезивању на мрежу. Статичке ће увек имати исту вредност, а динамичку адресу рачунар добија од DNS сервера када се повеже на мрежу. **Портови** имају сличну улогу као и локалне IP адресе. Веза између два рачунара остварује се помоћу глобалне IP адресе и порта. Како сви рачунари на истој мрежи имају исту глобалну адресу, рутер користи порт да одлучи ка којем рачунару треба да усмери конекцију. У рутеру постоји шема где су портови једнозначно повезани са локалним IP адресама повезаних уређаја. Могуће је ручно доделити локалну адресу рачунара одговарајућем порту и то се назива **„port forwarding".** Ова техника ће бити корисна при конфигурисању сервера.

Како се онда остварује веза у Јави? Са серверске стране налази се програм који чека да клијент затражи конекцију. То се постиже креирањем серверског сокета, при чему се тачно одређује на којем порту се очекују клијенти. Да би клијент иницирао повезивање потребно је да зна IPv4 адресу сервера и порт на којем се клијенти ослушкују. Ако су рачунари на којима се покрећу клијентски и серверски програм повезани на исту мрежу, у клијентском програму може да стоји и локална IP адреса сервера, али ако нису, адреса мора да буде глобална. Клијент тражи повезивање тако што на свом крају креира јединствени сокет за сервер са датом IP адресом и портом. Серверски сокет тада прихвата конекцију и креира јединствени сокет само за комуникацију са тим клијентом. На тај начин су рачунари повезани(слика 2). Напомена је да на овај начин још увек нису остварени сви услови за њихову комуникацију. За то је потребно користити токове података (енгл. DataStream), и то посебно за примање и слање података, и на серверској и на клијентској страни везе.



слика 2

2.2 Повезивање клијента са сервером

Прва ствар коју је потребно установити су IP адреса сервера (глобална!) и порт на којем се ишчекују клијенти. Даље, креира се објекат класе **Socket**, који ће представљати клијентски крај везе са сервером. За то је неопходно импортовати класу java.net.Socket, али је сасвим у реду и укључити цео пакет java.net.*. Класа Socket садржи девет конструктора од којих су најважнији наведени у табели 1. Неке корисне методе наведене су у табели 2.

1.	Socket()	Креира неповезани сокет.
2.	Socket(InetAddress address, int port)	Везује сокет за сервер са датом адресом на задатом порту.
3.	Socket(String host, int port)	Овде је IP адреса сервера типа string.

Табела 1

1.	void close()	Сокет се затвара.
2.	void connect(SocketAddress endpoint)	Повезује сокет са сервером.
3.	void connect(SocketAddress endpoint, int timeout)	Повезује сокет са сервером уз додатак временског ограничења чекања на одговор.
4.	InputStream getInputStream()	Враћа ток за примање података.
5.	OutputStream getOutputStream()	Враћа ток за слање података.
6.	int getPort()	Враћа порт преко којег је повезан са сервером.
7.	InetAddress getRemoteSocketAddess()	Враћа IP адресу сервера.
8.	boolean isClosed()	Тачно је ако је сокет затворен.
9.	boolean isConnected()	Тачно је ако је сокет повезан са сервером или ако је затворен, а пре затварања је била остварена веза са сервером.

Табела 2

Претпоставимо да већ постоји серверски програм који очекује клијенте на порту 9009 и који је покренут на рачунару повезаном на мрежу са глобалном адресом 174.234.198.100. Пример кода који повезује клијента са овим сервером је дат испод.

```
import java.net.*;

public class PrimerKlijent{

public static void main(String[] args){
        try{Socket server=new Socket("174.234.198.100",9009); }
```

```
catch(Exception e){}

}}
```

Уочимо try-catch блок у овом примеру. Постоји много разлога због којих повезивање са сервером може да буде неуспешно, стога нам је ово гаранција да програм неће пући при испаљивању изузетка, већ ће наставити са радом. Неке од могућих грешака су погрешна IP адреса сервера или порт.

2.3 Конфигурација сервера

Писање програма за серверски крај конекције је нешто комплексније од клијентског, из разлога што сервер користи две врсте сокета. Када прихвати конекцију клијента, за крај везе са њим креира се јединствени објекат класе **Socket**, аналогно као што клијент креира сокет за крај везе са сервером. Међутим, сервер захтева и посебан објекат класе **ServerSocket**, који служи за ишчекивање клијентских захтева за конекцију. Класа ServerSocket такође се налази у пакету java.net. Неки конструктори ове класе дати су у табели 3, а важније методе у табели 4.

1.	ServerSocket()	Креира неповезани серверски сокет – напомена: овде се под „неповезани“ мисли да није одређено на којем порту очекује клијенте.
2.	ServerSocket(int port)	Креира серверски сокет везан за одговарајући порт.
3.	ServerSocket(int port, int backlog)	Овде backlog представља максималан број захтева у реду за чекање на конекцију.

Табела 3

1.	Socket accept()	Чека захтев за конекцију и прихвата га.
2.	void close()	Сокет се затвара.
3.	InetAddress getInetAddress()	Враћа IP адресу мреже на коју је повезан.
4.	boolean isClosed()	Тачно је када је сокет затворен.
5.	boolean isBound()	Тачно је када је сокет „везан“ за порт и IP адресу.
6.	String toString()	Враћа String репрезентацију серверског сокета: IP адресу и порт за које је „везан“.
7.	void setSoTimeout(int timeout)	Ограничава се време чекања методе accept() на клијенте.

Табела 4

Дакле, када се креира објекат класе ServerSocket, позивањем његове методе accept() чека се захтев клијента за повезивањем. Приметимо да метода accept() враћа објекат класе сокет. То значи да се прихватањем клијентовог захтева уједно креира и сокет за крај везе са клијентом. Следећи код може да представља сервер за који се конектовао клијент из претходног примера.

```
import java.net.*;

public class PrimerServer{

public static void main(String[] args){

   try {

      ServerSocket SS=new ServerSocket(9009);

      Socket klijent1=SS.accept();}

   catch (Exception e) {}

}}
```

Програм ће се извршавати све док се не прими захтев за конекцију од клијента, било повезивање успешно или не. Време чекања може да се ограничи коришћењем методе setSoTimeout(). Време се уноси у милисекундама.

```
import java.net.*;

public class PrimerServer{

public static void main(String[] args){

   try {

      ServerSocket SS=new ServerSocket(9009);

      SS.setSoTimeout(10000);

      Socket klijent1=SS.accept();

   } catch (Exception e) {}

}}
```

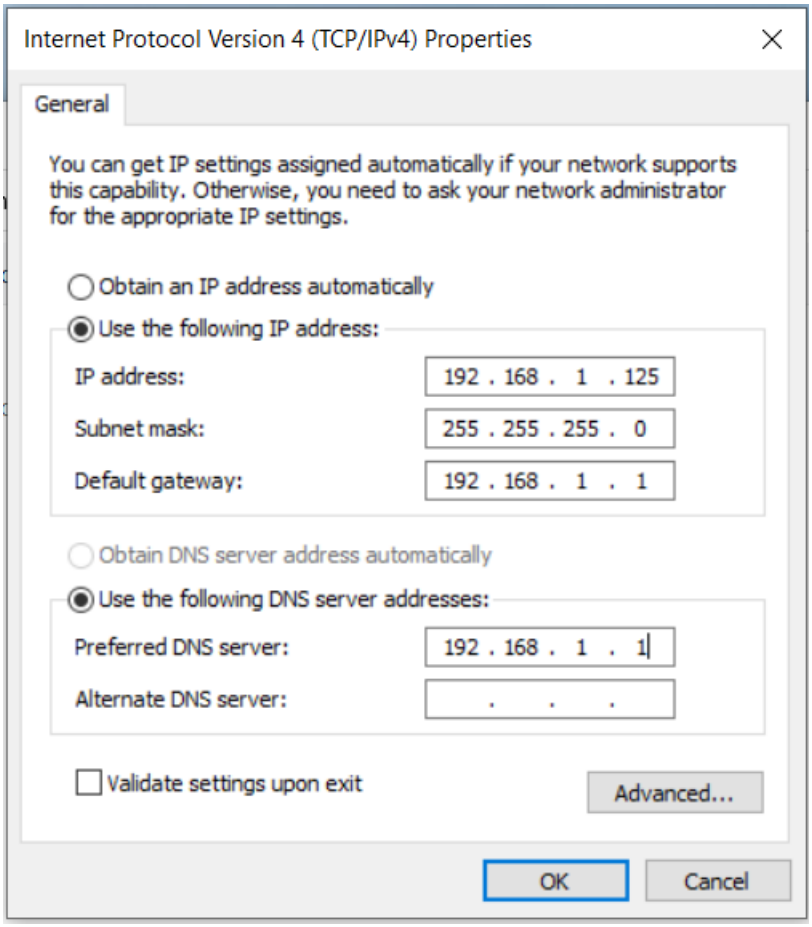
У овом примеру, сервер чека на захтев клијента 10 секунди, а затим се искључује.

Пре него што се започне са писањем кода потребно је извршити подешавање рачунара на коме ће се покренути серверски програм. Приметимо да серверски код није универзалан за све рачунаре. Да би био функционалан, потребно је да порт који стоји у конструктору серверског сокета буде „додељен“ рачунару на коме се покреће

серверски програм. Ту примењујемо технику „port forwarding”. Да бисмо уопше могли то да урадимо, за почетак је потребно да датом рачунару поставимо статичку IP адресу.

Постављање статичке IP адресе на примеру Windows рачунара:

- Control Panel -> Network and Internet -> Network and Sharing center -> Change adapter settings
- Потребан је десни клик на Wi-Fi и отворити properties
- Селектовањем „Internet Protocol Version 4 (TCP/IPv4)” и кликом на properties или само двоструким кликом отвориће се подешавања
- Потребно је означити поља „Use the following IP address “ и „ Use the following DNS server addresses” и идентично попунити сва поља као на слици 3, уз изузетак првог поља, где је само битно ставити IP адресу која тренутно није у употреби на локалној мрежи. Ту вредност ће дати рачунар узимати за своју статичку IP адресу при сваком повезивању на мрежу на коју је у том тренутку повезан.



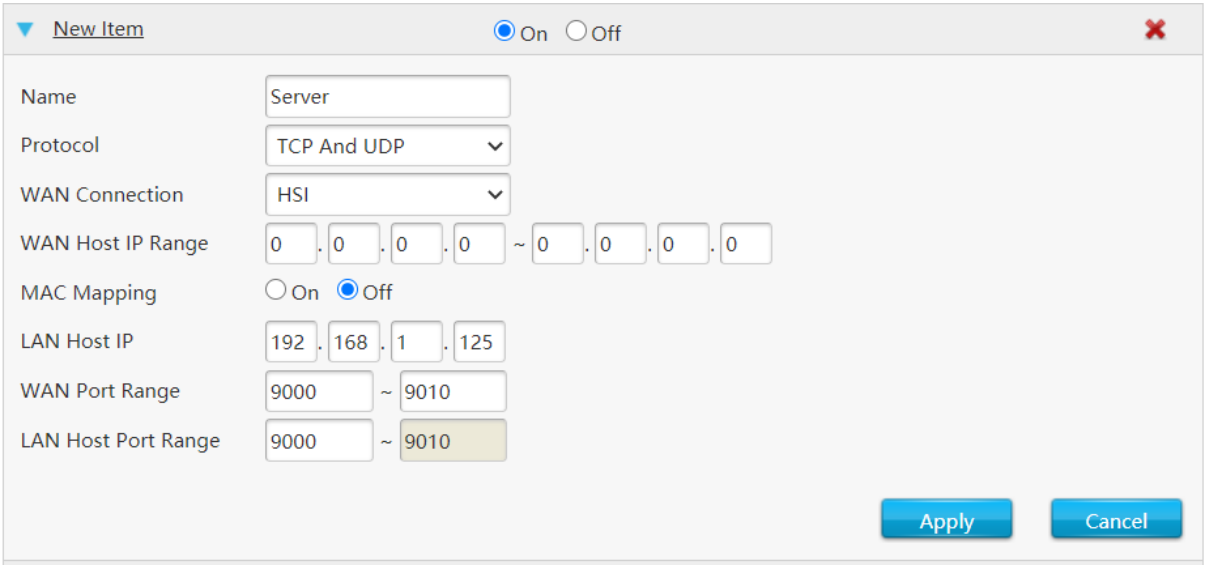
слика 3

Када је овај корак завршен, на реду је „port forwarding”. Потребно је отворити подешавања рутера(у произвољном претраживачу укуцати адресу рутера 192.168.1.1) и припојити изабрану статичку адресу порту на којем желимо да сервер очекује клијенте. Следећи пример је рађен на рутеру ZXHN H168N V3.1 у телеком мрежи.

Поступак:

- Након логовања на портал за конфигурацију отворена је картица „Internet”
- Изабрана је опција „Security” из левог менија
- Отворило се подешавање „Port forwarding” и изабрана је ставка „Create new”

На слици 4 је приказано одговарајуће попуњавање поља за изабрану статичку IP адресу 192.168.1.125 и жељени порт 9009.



слика 4

- Кликом на „Apply” потврђује се дато подешавање. Код специфицирања интервала којем припада порт треба имати у виду да је боље отворити што је могуће мање портова, јер је безбедније да некоришћени портови остану затворени. На овом примеру је одмах отворено више портова јер ће у наредним примерима бити коришћено још портова из датог опсега.
- Да проверимо да ли је порт отворен можемо да искористимо сајт https://ismyportopen.com/ или било који други са истом наменом. Напомена: серверски програм мора да буде покренут да би одговор био потврдан.

2.4 Размена података између клијента и сервера

Када знамо како да повежемо клијента са сервером, следећи корак је остваривање њихове комуникације. У ту сврху ћемо користити класе InputStream, OutputStream, DataInputStream и DataOutputStream. Оне се налазе у пакету java.io. Приметимо методе getInputStream() и getOutputStream() у класи Socket. Помоћу њих добићемо InputSream и OutputSream који ће служити као аргументи у конструкторима

DataInputSream() и DataOutputStream(). Заправо ћемо објекте ових класа користити за читање улаза и слање излазних података. Једини тип података који може да се прими/пошаље овим токовима јесте String. У наредном примеру налазе се допуњени кодови за сервер и клијента из претходног примера.

```
import java.net.*;
 import java.io.*;
public class PrimerServer{
public static void main(String[] args){
   try {
      ServerSocket SS=new ServerSocket(9009);
      Socket klijent1=SS.accept();
      InputStream is=klijent1.getInputStream();
         DataInputStream ulaz=new DataInputStream(is);
         OutputStream os=klijent1.getOutputStream();
         DataOutputStream izlaz=new DataOutputStream(os); }
    catch (Exception e) {}
}}
```

```
import java.net.*;
import java.io*;
public class PrimerKlijent{
public static void main(String[] args) {
      try{
         Socket server=new Socket("174.234.198.100",9009);
         DataInputStream ulaz=new DataInputStream(server.getInputStream());
         DataOutputStream izlaz=new DataOutputStream(server.getOutputStream());
}catch(Exception e){}
}}
```

Објекти класа DataInputStream користе методу readUTF() за пријем текста, а readInt(), readDouble(), ... за читање одговарајућих бројевних типова података. Аналогно, објекти класе DataOutputStream користе методе writeUTF(), writeInt(),… Улаз сервера директно је повезан са излазом клијента, и обрнуто. То значи да и кодови клијента и сервера морају бити савршено усклађени тако да када сервер шаље информације, клијент их очекује, а када их клијент шаље, сервер их очекује. У супротном ће се појавити грешке, највероватније ће доћи до бесконачног извршавања програма.

Додатак коду за сервер:

```
izlaz.writeUTF(“Povezani ste na server!”);
```

Додатак коду за клијента:

```
String porukaServera=ulaz.readUTF();

System.out.println(porukaServera);
```

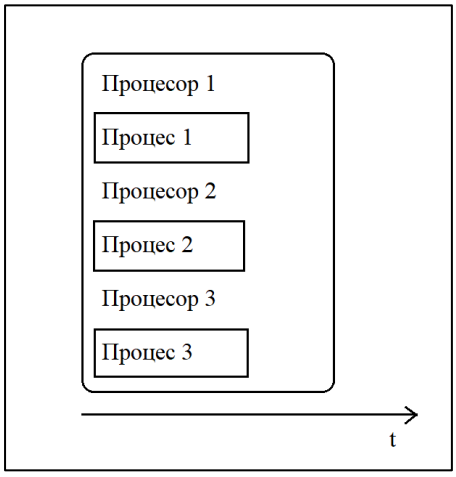
Извршавањем ових програма у конзоли клијента исписаће се порука „Povezani ste na server!”.

3

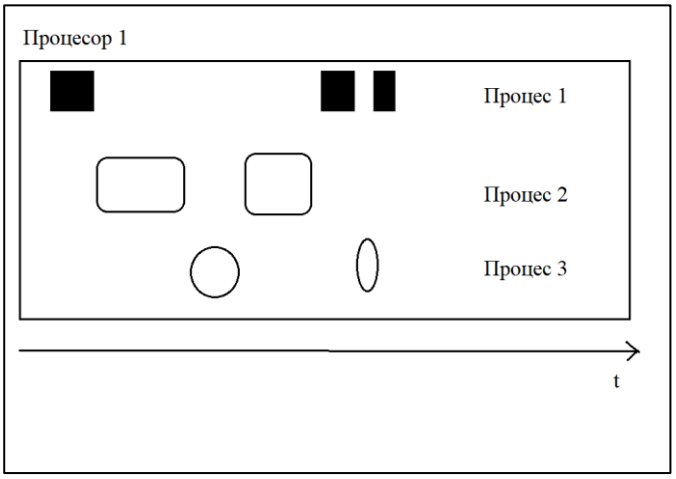
Нитно програмирање

3.1 Основни појмови, класа Thread и интерфејс Runnable

Примери рачунарских програма који су до сада били приказани представљају **задатке** које рачунар извршава **секвенцијално** – један за другим, редоследом којим су задати. Са друге стране, рачунари се одликују својством да могу извршавати више процеса истовремено – код мултипроцесорских рачунара то је у потпуности тачно, а код рачунара са једним процесором се паралелно извршавање симулира дељењем више процеса на мање целине и њиховим наизменичним извршавањем (слика 5).



слика 5а



слика 5б

Дакле, поставља се питање да ли је могуће написати програм који ће се састојати од неколико процеса који ће се истовремено извршавати. Могуће је, и овај концепт се назива **вишенитно програмирање**. Сваки процес, целина која представља један задатак – низ корака који се сукцесивно извршавају, назива се **нит (енгл. thread)**. Одатле и мултипроцесно програмирање добија назив вишенитно (енгл. multithreading). У Јави, сваки програм садржи бар једну основну нит у којој се налази main() метода, а ако је у питању графички програм, онда их садржи бар две. Ако желимо да се неки

задатак извршава паралелно са основном нити (и осталим нитима ако постоје), неопходно је да за њега креирамо нит која ће контролисати његово извршавање. У Јави, задатак ће бити објекат класе која имплементира интерфејс **Runnable**, док ће нит бити објекат класе **Thread**, или класе која је наслеђује. Интерфејс Runnable садржи само једну методу – **run()**, у којој се дефинише задатак. Ако би се у програму извршио позив те методе, не би се започело паралелно извршавање нове нити и датог задатка, већ би се задатак извршио у склопу нити у којој је позвана метода run(). Да би се заправо започело извршавање нове нити, односно методе run() те нити, неопходно је позвати методу **start(),** која је дефинисана у класи Thread. Метода run() може директно бити дефинисана у класи Thread, и тада није неопходно креирати и класу задатка. Са друге стране, ако је дата класа креирана, метода run() у класи Thread може бити редефинисана прослеђивањем инстанце дате класе конструктору класе Thread. У наредном примеру приказан је начин дефинисања класе задатка (која имплементира интерфејс Runnable). Нека је задатак испис времена.

```
import java.text.SimpleDateFormat;

public class Vreme implements Runnable{

public void run(){

   try{

      while(true){

         SimpleDateFormat formatiraj = new SimpleDateFormat("yyyy:MM:dd HH:mm:ss");

         System.out.println(formatiraj.format(System.currentTimeMillis()));

   }} catch(Exception e){}

}}
```

Класа Thread дозвољава и именовање нити, као и одређивање њиховог приоритета. Приоритет нити се дефинише целим бројевима од 1 до 10, где 1 означава нит најнижег, а 10 нит највишег приоритета. Приоритет 5 се зове још и нормирани приоритет. Метода main() има нормирани приоритет. Нити нижег приоритета могу да се извршавају паралелно са нитима вишег приоритета, али је једино сигурно да ће се извршавати када су нити вишег приоритета блокиране – успаване или чекају да нека друга нит заврши извршавање. Нит наслеђује приоритет нити која ју је креирала, а он се може мењати методом setPriority().

Нити могу бити корисничке(енгл. user) или демонске(енгл. daemon). Програм се извршава све док се не заврши извршавање свих корисничких нити. Без обзира на то да ли је завршено извршавање демонских нити, ако су корисничке нити „мртве“, демоске нити се заустављају и програм престаје са радом. Да ли ће дата нит бити демонска

зависи од тога да ли је нит која ју је креирала демонска. Демонство нити се може мењати методом setDaemon(boolean on), али искључиво пре почетка извршавања те нити.

У табели 5 налазе се важнији конструктори класе Thread, а у табели 6 неке чешће коришћене методе.

1.	Thread()	Креира нову нит.
2.	Thread(Runnable target)	Креира нову нит са дефинисаним задатком.
3.	Thread(String name)	Креира нову нит са датим именом.
4.	Thread(Runnable target, String name)	Креира нову нит са датим именом и дефинисаним задатком.

Табела 5

1.	static Thread currentThread()	Враћа референцу на нит која се тренутно извршава.
2.	int getPriority()	Враћа приоритет нити која је позива.
3.	static boolean holdsLock(Object obj)	Говори да ли дата нит држи браву објекта obj.
4.	void interrupt()	Прекида дату нит.
5.	boolean isInterrupted()	Проверава да ли је дата нит прекинута.
6.	boolean isAlive()	Проверава да ли је нит „жива“.
7.	boolean isDaemon()	Проверава да ли је нит демонска.
8.	void run()	Метода у којој се дефинише задатак.
9.	void join()	Чека се да дата нит „умре“.
10.	void join(int millis)	Чека се да дата нит „умре“ или millis милисекунди.
11.	void setDaemon(boolean on)	Одређује се да ли је нит корисничка или демонска.
12.	void setPriority(int newPriority)	Поставља нови приоритет нити.
13.	static void sleep(long millis)	Зауставља извршавање нити(успављује је) millis милисекунди.
14.	void start()	Започиње извршавање методе run().
15.	static void yield()	Шаље се обавештење да је нит спремна да препусти процесор другим нитима.

Табела 6

Наредни пример показује на које се начине могу креирати нити, а у конзоли се истовремено исписује и системско време и порука.

```
public class PrviNacin extends Thread{

    public void run(){
```

```
        try{while(true){
            System.out.println("Nit kreirana na prvi nacin");
        }}
        catch(Exception e){}
}}
public class Konstrukcija{
public static void main(String[] args){
    PrviNacin pn=new PrviNacin();
    Vreme ispisVremena=new Vreme(); // klasa Vreme iz prethodnog primera
    Thread drugiNacin=new Thread(ispisVremena);
    pn.start();
    drugiNacin.start();}}
```

3.2 Блокирање нити

Код приоритета нити речено је да је нит блокирана ако је успавана или чека. Нит се „успављује“ позивом методе sleep(long millis), што је приказано у наредном примеру.

```
public class NaizmenicnoIspisivanje implements Runnable{
        int brojKojiSeIspisuje, vremeCekanja;
        NaizmenicnoIspisivanje(int b, int millis){
                        brojKojiSeIspisuje=b;
                        vremeCekanja=millis;
       }
        public void run(){
                        try{while(true){
                                System.out.println(brojKojiSeIspisuje);
```

```
                    Thread.sleep(vremeCekanja);
                }} catch(Exception e){}
        }}
public class PrimerSleep{
public static void main(String[] args) {
    Thread prvaNit=new Thread(new NaizmenicnoIspisivanje(1,1000));
    Thread drugaNit=new Thread(new NaizmenicnoIspisivanje(2,500));
    prvaNit.start();
    drugaNit.start();
 }}
```

У овом примеру, прва нит исписује број 1, прави паузу од једне секунде и затим понавља ове две радње. Друга нит исписује број 2, прави паузу од пола секунде, и поново започиње циклус исписивања и чекања. Очекивани резултат је да ће се број 2 исписивати чешће него број 1.

Ако је нит у стању чекања, то може значити да су позване методе join(), yield(), или wait(). Код методе join(), нит у којој је извршен позив чека да објекат над којим се метода позива(нит која је позива) „умре“ да би наставила са извршавањем. Када се ова метода позове са аргументом који представља број милисекунди, чека се или да дата нит „умре“ или да протекне задато време. Постоји и облик методе join() са два аргумента који служи за прецизније одређивање времена чекања, где први аргумент представља број милисекунди, а други број наносекунди. У следећем примеру имамо две нити које креирају по један низ насумичних бројева из задатог интервала и одређене дужине. У методи main() се врши креирање нити, а затим и њихов почетак. Након стартовања прве нити, одмах се позива и метода за испис првог низа, а код другог низа се прво позива метода join(), а тек онда врши испис. Приликом позива методе за испис првог низа може доћи до грешке јер не можемо бити сигурни да је генерисан цео низ који се исписује. С обзиром на то да је за другу нит позвана метода join(), са сигурношћу да је цео низ креиран можемо позвати методу за испис другог низа.

```
public class NasumicniNiz extends Thread{
    int duzina, niz[];
    double skaliraj;
    public NasumicniNiz(double s, int d,String ime){
```

```
            super(ime);
            skaliraj=s;
            duzina=d;
            niz = new int[d];
        }
    public void run(){
        try{ for(int i=0;i<duzina;i++){
            niz[i]=(int)(Math.random()*skaliraj);
            System.out.println(super.getName() + "+" + niz[i]);
          }} catch(Exception e){}
        }
    public void ispis(){
        System.out.print(super.getName()+": ");
        for(int i=0;i<duzina;i++){
            System.out.print(niz[i]+" ");
        }System.out.println();
    }
}
public class PrimerJoin{
public static void main(String[] args) {
    try {
        NasumicniNiz nit1=new NasumicniNiz(3000,137,"PrviNiz");
        NasumicniNiz nit2=new NasumicniNiz(350,48,"DrugiNiz");
        nit1.start();
        nit1.ispis();
        /*nit2.start();
        nit2.join();
```

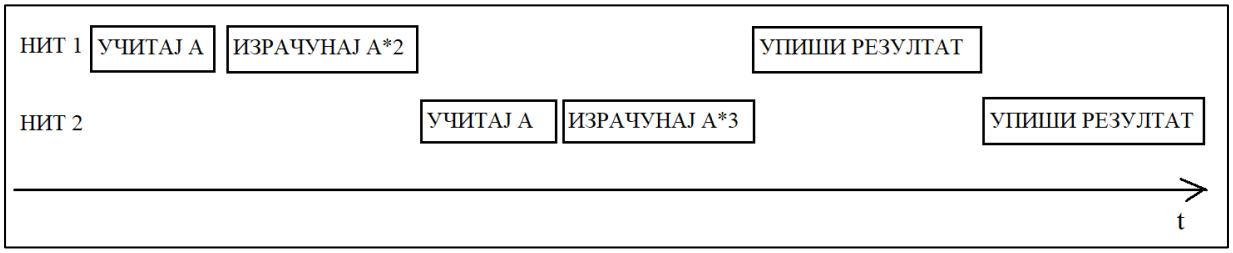
```
        nit2.ispis();*/
    } catch (Exception e) {}
}}
```

За разлику од методе join() која нити која је позива „помаже да се раније изврши“, метода yield() привремено зауставља објекат над којим је позвана – обавештава виртуелну машину да препусти процесор некој другој нити. Ипак, позивом методе yield() не гарантује се да ће се процесорско време доделити некој другој нити, јер јава виртуелна машина може да занемари послати сигнал.

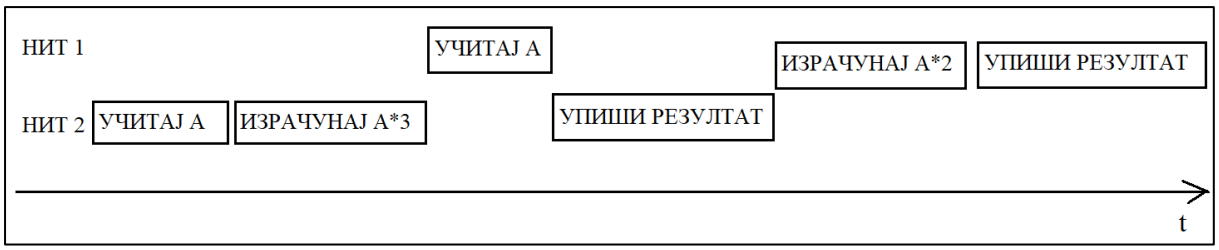
Нити могу изаћи из блокираног стања тако што им се пошаље сигнал прекида – позове се метода interrupt() над том нити.

3.3 Синхронизација нити – синхронизоване методе

Прости програми где нити раде независно једне од других су ретки. Много су чешћи програми где неке нити користе исте методе и раде над заједничким објектима. У таквим програмима важно је водити рачуна о синхронизацији нит, јер у супротном лако може доћи до грешака. Један уобичајени проблем познат је под називом **„Неизвесност трке“(енгл. „Race hazard”)**. Неизвесност трке се јавља ако у току извршавања једне нити нека друга нит може да промени објекте које користи прва нит, или услове који су неопходни за њено извршавање. Примера ради, нека наш програм садржи објекат класе KlasaU са пољем a=1 и методом uvecajXPuta(int x) која увећава поље a x пута, и две нити, тако да прва нит позива методу uvecajXPuta(2), а друга uvecajXPuta(3) класе KlasaU над тим објектом. Ако би се ове нити извршиле једна за другом, вредност поља a би на крају имала вредност 6. Међутим, при паралелном извршавању могу се добити и вредности 2 и 3. Ради се о томе да се увећање променљиве a одвија у три корака: учитавање тренутне вредности, израчунавање нове вредности и смештање нове вредности у променљиву, где након сваког од ових корака обе нити могу бити прекинуте, а да пре него што наставе са извршавањем она друга нит може да измени стање објекта. На слици 6 приказана су две могућности извршавања програма, тако да се у првом добије резултат a=3, а у другом a=2.



слика 6а



слика 6б

Решење овог, и сличних проблема, лежи у синхронизацији. Синхронизација нити заснива се на принципу **браве(енгл. lock)** и **кључа(енгл. key)** неког објекта, а до закључавања објекта долази при позиву неке од синхронизованих метода тог објекта. Синхронизоване методе имају модификатор synchronized. Када нека нит позове синхронизовану методу откључаног објекта, та нит преузима кључ тог објекта, а он се закључава све док дата нит не заврши извршавање те синхронизоване методе. Друге нити које за то време позивају синхронизоване методе тог истог, сада закључаног објекта, морају да чекају да прва нит заврши са коришћењем објекта и да се он откључа, како би започеле своје извршавање. Оне ће тада преузети кључ објекта, и он ће се поново закључати. Може се десити да нит која има кључ објекта позове неку другу синхронизовану методу над тим објектом – али тада неће доћи до грешке (нит неће блокирати саму себе). У Јави се ово својство назива reentrant synchronization. У наредном примеру је приказан код за испис тачног решења применом синхронизације методе uvecajXPuta(int x).

```
public class KlasaU{
        private int a=1;
        public synchronized void uvecajXPuta(int x){
                a=a*x;
        }
public void ispisi(){
        System.out.println(a);
    }
}
public class Uvecanje extends Thread{
        int uvecaj;
        KlasaU objekat;
        Uvecanje(int u, KlasaU obj){
```

```
            super();
            uvecaj=u;
            objekat=obj;
        }
        public void run(){
            try{objekat.uvecajXPuta(uvecaj);}
            catch(Exception e){}
        }}
public Class PrimerSinhronizacije{
        public static void main(String[] args) {
            try {
                KlasaU obj=new KlasaU();
                Thread t1=new Uvecanje(2,obj), t2=new Uvecanje(3,obj);
                 t1.start();
                t2.start();
                t1.join(); // join() utiče na main() metodu – sprečava ispis pre
                t2.join(); // svih uvećanja promenljive a
                 obj.ispisi();} catch (Exception e) {}
        }}
```

Овај код ће увек исписивати вредност шест, али треба имати на уму да синхронизација не мора увек да гарантује тачно решење. На пример, да поље a није било private, тј. да није било неопходно приступати му преко метода, могло је да се деси да му нека нит промени вредност директним приступом, при чему нема синхронизације која би је у томе спречила.

Дефинисање конструктора као синхронизоване методе сматра се синтаксном грешком, јер док објекат још није креиран приступа му само она нит која га креира. Код редефинисања метода из наткласе синхронизованост се не наслеђује. Ако метода наткласе није синхронизована, то не спречава редефинисану методу да буде синхронизована – њена синхронизација се остварује путем модификатора synchronized. Могуће је синхронизовати и статичке методе, с тим што тада нити добијају такозвани „кључ класе“, уместо кључа објекта. Кључ класе функционише независно од кључева

објеката, он утиче само на извршавање статичких метода. Ако нека нит има кључ класе, друге нити могу да извршавају само нестатичке методе те класе, а блокирају се и чекају на кључ ако позову синхронинзовану статичку методу те класе.

3.4 Синхронизоване наредбе – синхронизација по објекту

Други начин на који је претходни код могао да се модификује, а да исписује решење 6, подразумева коришћење синхронизације по објекту. Конкретно, до промене у коду долази код класе Uvecanje, али узимајући да метода uvecajXPuta(int x) није синхронизована.

```
public class Uvecanje extends Thread{
        int uvecaj;
        KlasaU objekat;
        Uvecanje(int u, KlasaU obj){
                super();
                uvecaj=u;
                objekat=obj;
        }
        public void run(){
                try{    synchronized(objekat){
                        objekat.uvecajXPuta(uvecaj);
                }}catch(Exception e){}
  }}
```

Из примера закључујемо да синхронизација блока наредби по неком објекту значи да када нит дође до извршавања датог блока наредби, у случају да је брава тог објекта откључана узима кључ тог објекта до завршетка извршења тог блока наредби, а у случају да је објекат закључан, привремено се блокира и чека да се кључ ослободи. Дакле, нит се понаша аналогно као да је блок наредби синхронизована метода класе објекта по којем је синхронизована, позвана над тим објектом. Синхронизација по објекту је јако корисна када је потребно без прекида извршавати више

несинхронизованих метода над истим објектом, а посебно када се извршава и више метода над различитим објектима.

3.5 Мртва петља

Синхронизација нити може довести до проблема **мртве петље(енгл. deadlock),** тј. узајамног блокирања нити. На пример, нека се паралелно извршавају две нити: Нит1 и Нит2. Нека Нит1 извршава синхронизован блок наредби по објекту А, при чему се у том блоку налази синхронизован блок наредби по објекту Б, и нека Нит2 извршава блок наредби синхронизован по објекту Б, у којем се налази блок наредби синхронизован по објекту А. На почетку Нит1 има кључ објекта А, а Нит2 кључ објекта Б. У току свог извршавања, Нит1 ће доћи до блока наредби који је синхронизован по објекту Б, блокираће се, и чекати да Нит2 заврши извршавање како би јој препустила кључ над бравом објекта Б. С друге стране, Нит2 не може да препусти кључ објекта Б све док не изврши и блок наредби који је синхронизован по А, а за то јој је потребан кључ објекта А, који има блокирана Нит1. Када на ред дође извршавање блока који је синхронизован по А, и Нит2 ће се блокирати. На тај начин, обе нити су блокиране, и ниједна се неће одблокирати. Решавање проблема мртве петље је задатак програмера, јер Јава нема начин за откривање ове грешке. Један начин на који може да се избегне узајамно блокирање јесте организација ресурса. Одређивањем јединственог редоследа за приступање синхронизујућим објектима који ће поштовати све нити не долази до мртве петље. Да је у горњем примеру утврђен редослед објеката А,Б, тада би и Нит2 морала да прво набави кључ објекта А, пре него што би заузела и кључ објекта Б, тј. пре него што би уопште започела извршавање морала би да сачека да се заврши Нит1, и мртва петља би била избегнута.

3.6 Методе wait(), notify() и notifyAll()

У случају када извршавање једне нити зависи од резултата извршавања друге нити, а узмимо и да нисмо у могућности да искористимо методу join(), њихов правилан рад може се установити методама wait() и notify(). Позиви ових метода морају се налазити унутар синхорнизованих метода или наредби по истом објекту. Позивом методе wait() над неким објектом нит прелази у блокирано стање и чека обавештење да може да настави са радом – да нека друга нит позове методу notify() над истим објектом. Том приликом дата нит такође препушта кључ објекта којим се синхронизује. Тиме се обезбеђује да друга нит може да започне извршавање блока наредби међу којима се налази и позив методе notify() – биће избегнута мртва петља.

```
import java.util.ArrayList;
public class Kockica{
   private ArrayList<Integer> bacanja=new ArrayList<Integer>();

   synchronized void baciKockicu(){
         bacanja.add((int)(Math.random()*6+1));
         notify();
   }
   synchronized int getSize(){
         return bacanja.size();
   }
   synchronized int getIshod(){
         return bacanja.remove(0);
   }
}
public class Bacanje extends Thread{
         Kockica k;
         Bacanje(Kockica k){
               this.k=k;
         }
         public void run(){
              try{
                  while(true){
                  k.baciKockicu();
              }} catch(Exception e){}
       }
}
```

```
public class Ishod extends Thread{
        Kockica k;
        Ishod(Kockica k){
                this.k=k;
        }
    public void run(){
    try{
    while(true){
        synchronized(k){
                while(k.getSize()==0){
                        k.wait();
                }
                System.out.println(k.getIshod());
                }}
                }catch(Exception e){}
        }
}
public class PrimerWaitNotify{
public static void main(String[] args) {
    Kockica kockica=new Kockica();
    Thread ishod=new Ishod(kockica);
    Thread bacanje = new Bacanje(kockica);
    ishod.start();
    bacanje.start();
}}
```

У примеру имамо нит за ,,бацање коцкице“ и нит за исписивање исхода бацања. Важно је да пре исписивања постоји исход који ће се исписати, тако да нит да исписивања мора да позива методу wait() сваки пут када је низ исхода празан. Са друге стране,

методом за „бацање“ позива се метода notify() којом се обавештава да је коцкица бачена. Приметимо да су обе нити синхронизоване по истом објекту, иначе би позиви метода wait() и notify() били бесмислени.

4

Примене вишетнитног програмирања при умрежавању рачунара

Уводни пример: Знамо како да креирамо сервер који очекује једног клијента и са њим размењује податке. Рецимо да сада желимо да креирамо сервер који ће прихватати 2 клијента. То бисмо постигли двоструким позивом методе accept() у серверском програму. Ако желимо да прихватимо n клијената, методу accept() бисмо сместили унутар циклуса. Међутим, шта ако желимо да примимо неодређени број клијената, а да притом омогућимо и размену података са њима? Ако је метода accept() смештена унутар бесконачног циклуса, серверски програм ће моћи да прими неограничени број клијената, али је повезивање са клијентима уједно и једини задатак који ће извршавати. Зато ће бити јако корисно да се прихватање нових клијената и комуникација са њима одвијају паралелно у различитим нитима.

```
import java.util.*;
import java.net.*;
import java.io.*;
public class Klijent extends Thread{
   Socket klijent;
   DataInputStream ulaz;
   DataOutputStream izlaz;
   String ime;
   Klijent(Socket s,int i){
      klijent=s;
      ime="Klijent "+i+" ";
   }
   synchronized void ispis(String s){
```

```
        System.out.println(ime+": "+s);
    }
    public void run(){
        try{
            ulaz=new DataInputStream(klijent.getInputStream());
            izlaz=new DataOutputStream(klijent.getOutputStream());
            izlaz.writeUTF("Uspešno povezivanje sa serverom!");
            while(true){
                String s;
                s=ulaz.readUTF();
                ispis(s);
            }}catch(Exception e){}
    }}
public class MultiprocesniServer {
    public static void main(String[] args) {
        try{
        ServerSocket ss=new ServerSocket(9009);
        Klijent k;
        ArrayList<Klijent> klijenti = new ArrayList<Klijent>();
        while(true){
            k=new Klijent(ss.accept(),klijenti.size());
            klijenti.add(k);
            k.start();
        }}catch(Exception e){}
    }}
```

Овај серверски програм садржи класу Klijent која паралелно са главном нити, у којој се врши повезивање сервера и клијената, управља разменом података између сервера и клијената. У овом примеру, када се изврши повезивање клијента са сервером, сервер

шаље обавештење да је повезивање било успешно, а затим прима поруке од клијената и исписује их у конзоли. Приметимо да се за испис текста позива синхронизована метода ispis(). Тиме је усклађено исписивање порука свих клијената који могу да их шаљу „истовремено“. Међутим, због евентуалног чекања на испис текста може доћи до не примања неке поруке због блокираности нити. Зато би још бољу било да се и примање и исписивање порука налаза у засебним нитима. У том случају би чак било омогућено и паралелно примање и слање порука између сервера и клијената(уз одговарајуће клијентске програме код којих се примање и слање података такође врши у различитим нитима). Следи приказ одговорајућег програма клијента за серверски програм изнад. Напомена: При повезивању са сервером користи се loopback адреса уз претпоставку да се и клијентски и серверски програм покрећу на истом рачунару.

```
import java.util.*;

import java.net.*;

import java.io.*;

public class Klijent {

   public static void main(String[] args) {

        try{

        Scanner sc=new Scanner(System.in);

        String s;

        Socket server=new Socket("127.0.0.1", 9009);

        DataInputStream ulaz=new DataInputStream(server.getInputStream());

        DataOutputStream izlaz=new DataOutputStream(server.getOutputStream());

        String porukaServera=ulaz.readUTF();

        System.out.println(porukaServera);

        while(true){

           System.out.print("Pošalji poruku serveru: ");

           s=sc.nextLine();

           izlaz.writeUTF(s);

        }

}catch(Exception e){}}}
```

5

Закључак

У првом делу рада дефинисани су појмови непоходни за разумевање начина на који се остварује веза између рачунара, а затим је објашњена основа мрежног клијент-сервер програмирања у програмском језику Јава (такозваног Java Networking-а). У другом делу рада обрађен је концепт вишенитног програмирања, као и његова имплементација у Јави. Обједињујући материјал из претходних поглавља, трећи део приказује важност познавања нити за програмирање разних клијент-сервер програма. Циљ апликације приложене уз рад је да додатно прикаже да се чак и програми који делују компликовано могу испрограмирати успомоћ основних рачунарских концепата и програмерских вештина, и да уједно тиме мотивише читаоце да не одустају од својих идеја за програме услед наизглед нерешивих проблема.

На крају желим да се захвалим и свом ментору, професорки Снежани Јелић, на свему што ме је научила током мог школовања, а посебно на инспирацији да наставим да се бавим рачунарством, истражујем и унапређујем своје знање.

Литература

[1] Дејан Живковић, *Јава програмирање,* Универзитет Сингидунум, 2011

[2] https://rti.etf.bg.ac.rs/rti/ir2oo2/predavanja/Java/09%20Niti%20-%20Java.pdf

[3] https://docs.oracle.com/javase/tutorial/

[4] https://www.tutorialspoint.com/java/java_networking.htm

[5] https://docs.oracle.com/javase/7/docs/api/index.html

[6] https://stackabuse.com/how-to-get-current-date-and-time-in-java/

[7] https://www.tutorialspoint.com/java/java_thread_synchronization.htm

[8] https://www.geeksforgeeks.org/synchronized-in-java/

[9] https://www.javatpoint.com/daemon-thread

[10] https://www.geeksforgeeks.org/daemon-thread-java/

[11] https://winterbe.com/posts/2015/04/30/java8-concurrency-tutorial-synchronized-locks-examples/

[12] https://www.logicbig.com/tutorials/core-java-tutorial/java-multi-threading/happens-before.html