

Šolski center Novo mesto
Srednja elektro šola in tehniška gimnazija
Šegova ulica 112
8000 Novo mesto

POTI PO MREŽI
(Maturitetna seminarska naloga)

Predmet: Računalništvo

Avtor: Nik Jenič, T4b

Mentor: Gregor Mede, univ. dipl. inž. rač. in inf.

Novo mesto, april 2022

POVZETEK IN KLJUČNE BESEDE

V seminarski nalogi predstavim celoten postopek kako sem ustvaril svojo seminarsko nalogo. Na začetku opišem orodja, brez katerih ne bi mogel napredovati, nato pa opišem vse sestavine svojega programa.

Da bi si bralci lažje predstavljali opis programa sem dodal tudi slike in vire, kjer se lahko uporabniki poglobijo v posamezne teme.

Na koncu sem pod prilogo dodal tudi hiperpovezavo, kjer si lahko bralci ta program naložijo in pogledajo celotno kodo.

Ključne besede:

- Java
- Knjižnica
- Metoda
- Razred
- Spremenljivka
- Program
- Mreža
- Poti
- Črte

KAZALA

KAZALO VSEBINE

1	UVOD	1
2	POTI PO MREŽI	2
2.1	TEORETIČEN DEL	2
2.1.1	PROGRAMSKI JEZIK JAVA	2
2.1.2	KNJIŽNICE	3
2.1.3	PROJECT EULER	5
2.2	PRAKTIČEN DEL	6
2.2.1	PROGRAM	6
2.2.2	PROBLEM	9
2.2.3	OKNO	10
2.2.4	MREŽA	13
2.2.5	ALGORITEM	15
2.2.6	GUMB FAST	22
2.2.7	PRILAGAJANJE PROGRAMA	28
3	ZAKLJUČEK	30
4	ZAHVALA	31
5	VIRI IN LITERATURA	32
6	STVARNO KAZALO	35
7	PRILOGE	36

KAZALO SLIK

Slika 1: Project Euler problem 15, Lattice paths (19).....	5
Slika 2: Končen izdelek	6
Slika 3: Mreža z velikostjo 4	8
Slika 4: Rekurzivna metoda algo	9
Slika 5: Main razred in main metoda.	10
Slika 6: Ukazi za prikazovanje okna	10
Slika 7: Ukaz za ustvarjanje gumba "Start".	11
Slika 8: Ukazi za dodajanje elementov na okno.	11
Slika 9: Določanje velikosti vpisnemu polju.	12
Slika 10: Dodajanje funkcionalnosti gumbu za zamik.....	12
Slika 11: Nastavljanje velikosti prostora za mrežo.....	13
Slika 12: Izris začetne mreže.....	14
Slika 13: Dodajanje nove črte v HashMap.....	17
Slika 14: Razredne spremenljivke razreda Line	18
Slika 15: Ustvarjanje niti za algoritem.....	20
Slika 16: Ukaz za poganjanje niti.....	20
Slika 17: CheckOverflow metoda	22
Slika 18: Delovanje metode multiply.....	23
Slika 19: Metoda factorial	24
Slika 20: Metoda za odštevanje.....	25
Slika 21: Metoda checkNegativeOverflow	25
Slika 22: Metoda za deljenje.....	26
Slika 23: Enačba za hiter rezultat	26
Slika 24: Vsebina konstruktorja razreda CustomizeFrame	28
Slika 25: Nastavitve gumba Customize	28
Slika 26: Nastavljanje funkcionalnosti gumba gridButton	29

1 UVOD

Mnogo ljudi, ki se želi izboljšati v programerskem načinu razmišljanja se obrne na spletno stran imenovano ProjectEuler. Ko sem na tej odkril 15. nalogo, sem opazil, da si je težko predstavljati točno kako deluje, zato sem se odločil, da jo bom spremenil v grafičen program, ki lahko vsem predstavi način reševanje le te naloge.

Takrat, ko sem izbral to nalogo, sem se ravno učil grafičnega programiranja v programskem jeziku Java, zato sem tudi ta program se odločil narediti v tem jeziku. Da bi pa ta program lahko učinkovito naredil, sem se rabil naučiti tudi upravljanja z več nitmi.

Da bi lahko program ustvaril sem se zanašal na spletne pisne vire za programiranje ter tudi na objavljene videoposnetke in na snov, ki smo jo obdelali pri pouku. Za lažjo predstavitev delovanja nameravam v seminarski nalogi uporabljati tudi slike in opise kode.

V opis kode se bom precej poglobil, vendar ne bom sproti vključeval celotnega programa. To je narejeno z namenom, da lahko izkušeni programerji dobijo idejo programiranja, brez da takoj dobijo rešitev.

Vključil bom tudi celoten program, da si ga lahko bralec naloži in testira njegovo delovanje.

2 POTI PO MREŽI

Za seminarsko nalogo sem naredil program, ki najde vse poti iz zgornjega levega kota mreže v spodnji desni kot, kjer se lahko premikamo samo desno in dol.

(1)

2.1 TEORETIČEN DEL

Da sem lahko ta program naredil sem se rabil naučiti programirati. Hkrati sem rabil dobiti idejo za svoj program. V tem delu bom opisal kater programski jezik sem uporabil, katere knjižnice in kje sem dobil inspiracijo.

2.1.1 PROGRAMSKI JEZIK JAVA

Java je visokonivojski programski jezik, ki je bil prvič izdan leta 1995 s strani Sun Microsystems. Od mladih nog se je ta jezik razvil v en najbolj uporabljenih jezikov današnjega časa.

(2) (3) (4)

Dandanes obstaja mnogo lažjih jezikov, ki se jih bi lahko hitreje naučili in bi z njimi opravili večinoma enake naloge, vendar Java ima svoje lastnosti, zaradi katerih je dolgo ostala na vrhu podjetniških aplikacij.

(5)

Zanesljivost

Ena izmed teh lastnosti je zanesljivost. Zaradi dolgih let delovanja je Java imela dovolj časa, da so bili preizkušene vse njene zmogljivosti in odpravljene mnoge napake. Zaradi teh lastnosti je bila Java uporabljena v aplikacija kot so Apache Kafka, eden najbolj popularnih programov za povezovanje podatkov na več sistemih.

(6)

Piši enkrat, poženi kjerkoli

Še ena privlačna lastnost programskega jezika Java je to, da se napisana koda (tako imenovana izvorna koda) ne prevede takoj v to, kar razume računalnik. Pri drugih jezikih se naša koda takoj pretvori v jezik računalnikov. Pri Javi se koda pretvori v vmesno kodo. Ko želimo to kodo pognati, jo rabimo še enkrat pretvoriti preko Javanskega prevajalnika. Zaradi tega ne rabimo kot programerji kode pretvarjati na

vsakem operacijskem sistemu posebej. To naredimo enkrat, in za ostalo poskrbi Javanski prevajalnik, ki nato zna program pognati na Windows, Linux in tudi MacOS sistemih, brez dodatnega dela s strani programerja.

Slaba stran tega sistema je ta, da rabi uporabnik imeti na sistemu naložen ta prevajalnik, ali pa rabimo celotnega vključiti skupaj z našo aplikacijo.

(7) (8)

Objektno usmerjeno programiranje

Odvisno koga vprašamo, je to dobra ali slaba lastnost Jave. Java je bila ustvarjena izključno kot objektno usmerjen programski jezik. V nekaterih primerih se ta način programiranja počuti celo vsiljen v tem jeziku, kjer ne bi rabil biti, vendar hkrati nam omogoča bolj logično organizacijo podatkov.

Objektno usmerjeno programiranje razdeli program na objekte. Te si lahko zamislimo kot predmete, vsak s svojimi podatki. Imamo lahko npr. objekt »Računalnik«, ki nam pove, ali ta ima vgrajeno tipkovnico, monitor, koliko pomnilnika ima itd. Več si lahko o objektih preberete na priloženih virih.

(9)

2.1.2 KNJIŽNICE

V programiranju knjižnica pomeni zbirka programov ali paketov kod, ki jih lahko uporabljamo v naših programih, brez da bi jih rabili sprogramirati sami.

(10) (11)

Da v programskem jeziku Java dodamo knjižnice uporabimo ukaz »import lokacija/ime knjižnice«;

(12)

V programskem jeziku Java se knjižnice imenujejo paketi.

(4)

Vse knjižnice, ki sem jih uporabil, pridejo skupaj z Java Development Kit in naj bi do njih imeli vsi programerji jezika Java. (13) V svojem programu sem uporabil naslednje knjižnice:

java.util.ArrayList in java.util.HashMap

Pri programiranju pogosto uporabljamo sezname. Te nam omogočajo shranjevanje več različnih vrednosti pod eno ime spremenljivke, vendar jim moramo vnaprej določiti število vrednosti, ki jih bomo uporabljali. ArrayList nam omogoča sprotno dodajanje in odstranjevanje vrednosti s preprostimi ukazi, kot so `.add(vrednost);` in `.remove(indeks);`

Podatke lahko shranjujemo tudi v objekt razreda HashMap. Ta nam omogoča dostop do podatkov preko imen, vendar so te neurejeni, zato jih ne moremo dobiti preko indeksov in ne moremo imeti podatkov z podvojenimi imeni. (12)

java.awt.Color in java.awt.Dimension

Java AWT (Abstract Window Toolkit) paket nam omogoča uporabo grafičnih elementov, kot so barve. V svoj program sem dodal možnost spreminjanja vseh barv znotraj programa. Temu mi je pomagala ta paket, ki omogoča preprosto delovanje z barvami, kjer rabimo samo poznati RGB format barv npr. 255,0,0 , ali pa jih kličemo po imenih, npr. `Color.black;`

Poleg Color sem uporabil tudi Dimension, ki nam samo omogoča preprosto shranjevanje dimenzij okna, gumba in podobno, kjer samo opišemo širino in višino elementa, npr. 100,20.

(14) (15) (16)

javax.swing.*

Ta paket nam omogoča uporabo nezahtevnih grafičnih komponent, kot so okno, gumbi, polja in podobno. Večina moja seminarske ne bi mogla delovati brez tega paketa. Ta je bil ustvarjen, da lahko nadomesti splošen AWT paket, ki je tudi dodal grafične elemente, ampak na veliko bolj primitiven način, kjer je uporabnik rabil sam dodelati zahtevnejše uporabe.

(17) (18)

2.1.3 PROJECT EULER

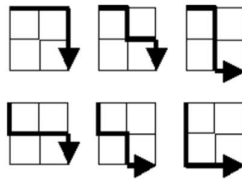
Project Euler, poimenovan po slavnem matematiku Eulerju, je spletna stran polna zahtevnih matematičnih in programerskih problemov. Mnogo ljudi, ki se želi izboljšati v programiranju se zanese na to spletno stran za učinkovite probleme.

Na tej spletni strani obstaja 15. problem »Lattice paths«. Ko sem sam reševal nalogo sem želel ta problem prikazati grafično, zato sem po njem naredil program. Sam problem zgleda tako, kot je prikazano na sliki 1:

Lattice paths

Problem 15

Starting in the top left corner of a 2×2 grid, and only being able to move to the right and down, there are exactly 6 routes to the bottom right corner.



How many such routes are there through a 20×20 grid?

Slika 1: Project Euler problem 15, Lattice paths (19)

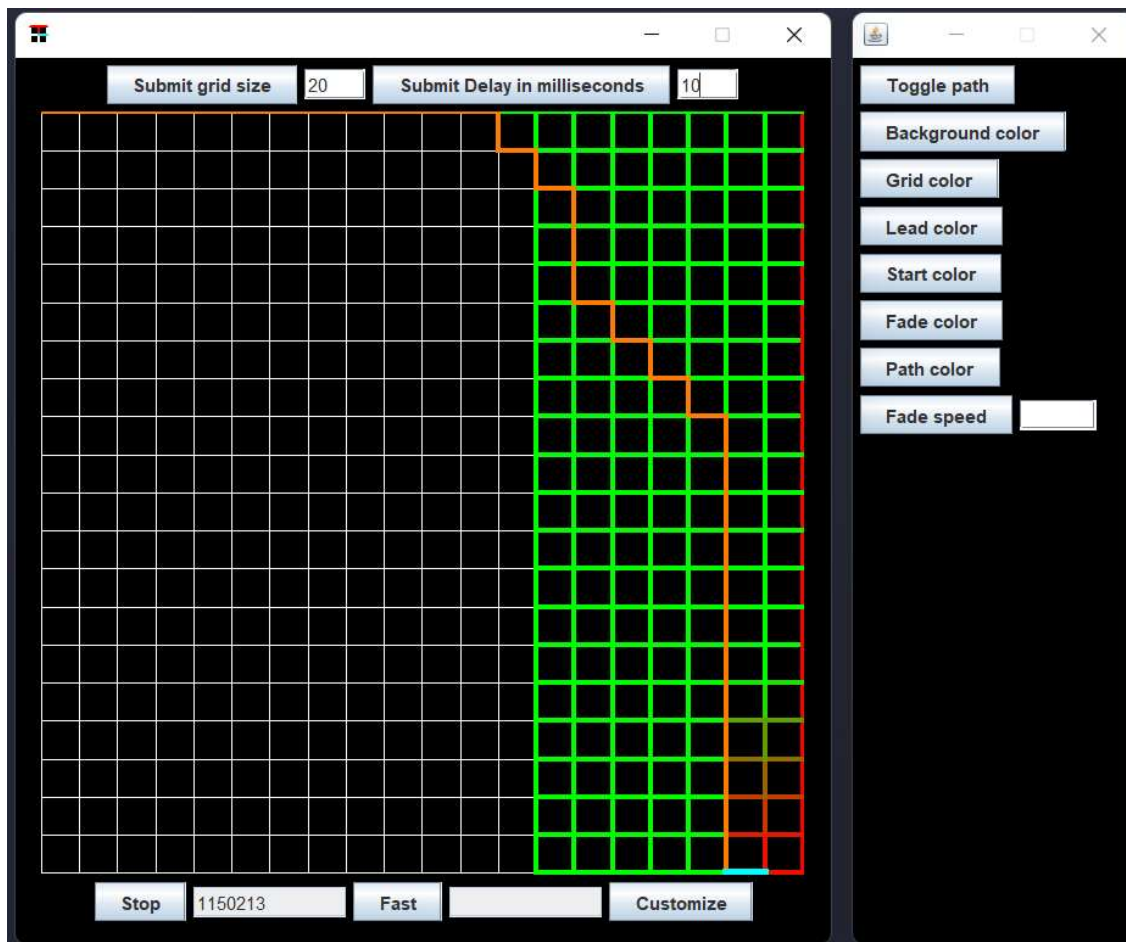
(19)

2.2 PRAKTIČEN DEL

Ko sem se odločil kateri jezik bom uporabil ter katere knjižnice bi mi najbolj služile, sem rabil začeti s programiranjem.

2.2.1 PROGRAM

Da si lahko lažje predstavimo čemu gradimo program, bi rad za začetek prikazal končni izdelek.



Slika 2: Končen izdelek

Kot vidimo na sliki 2, ima program glavno okno, na katerem imamo gumbe, vnosna polja ter glaven kvadrat, kjer se nam prikaže mreža.

Poleg tega ima program tudi svojo ikono.

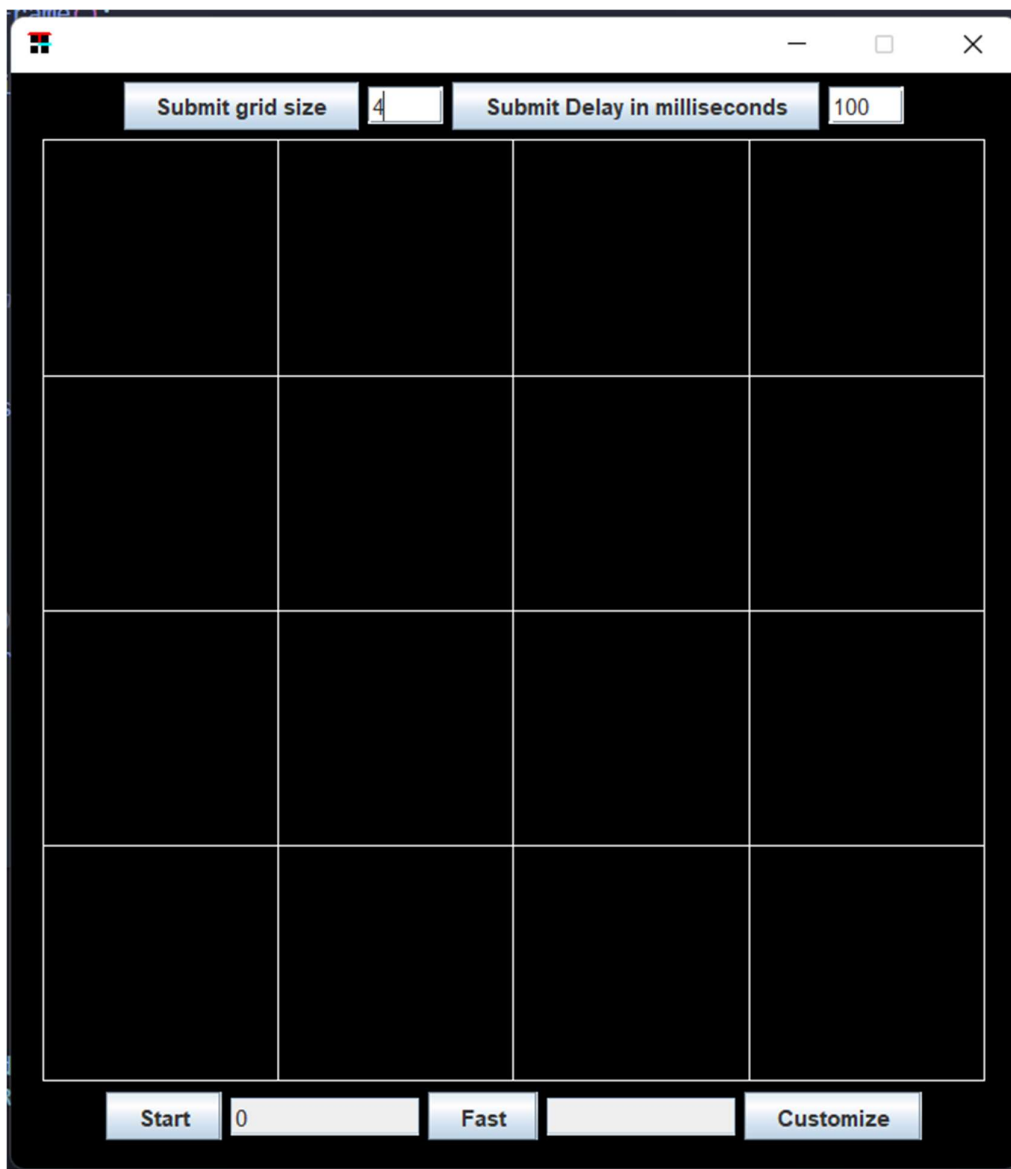
Funkcionalnost gumbov in vnosnih polj

Program ima precej gumbov. Tu bom opisal kaj vsak naredi in njihovo povezavo z vnosnimi polji:

- Submit grid size (Slovensko: Vnesi velikost mreže) -> Ko v polje zraven polja vnesemo neko število in nato kliknemo na gumb, se nam prikaže mreža take velikosti, katero smo vpisali. Poleg tega preneha delovanje iskanja poti, če je ta slučajno v teku.
- Submit delay in milliseconds (Slovensko: Vnesi zamik v milisekundah) -> Če želi uporabnik spremljati, kako program potuje čez mrežo, lahko v polje zraven gumba vnese vrednost, koliko milisekund želi da mine med posameznim premikom črte.
- Start/Stop -> Ta gumb požene iskanje poti. Ko program deluje, želimo da piše stop in ko program prekinemo ali ta najde vse poti, želimo da se spremeni v start. Poleg gumba je izpisno polje, ki nam pove koliko poti smo že našli.
- Fast (Slovensko: Hitro) -> Iskanje vseh poti v velikih mrežah, kot so mreže z velikostjo 15 in več, je lahko zelo zamudno. Če uporabnik želi samo rezultat brez animacije, lahko klikne na ta gumb in mu v polje poleg gumba vpiše pravilen rezultat mnogokrat hitreje kot pa bi trajalo, če bi računalnik »ročno« iskal vse poti.
- Customize (Slovensko: Prilagodi) -> Ta program nam odpre novo okno, kjer lahko uporabnik spremeni večino barv znotraj programa, vključno z ozadjem, začetno barvo črtic, končno barvo črtic, barvo »glave« iskanja, trenutno pot in več. Uporabnik lahko tudi vnese željeno trajanje animacije premikanja iz začetne barve črtic v končno barvo.

Glavna mreža

Glavna atrakcija v celotnem programu pa seveda niso vsi gumbi, ampak mreža, ki išče vse poti. Mreža vedno ostane enake fizične velikosti. Spreminja se le število polj znotraj te, kot vidimo na sliki 3.



Slika 3: Mreža z velikostjo 4

2.2.2 PROBLEM

Za samo temo naloge sem izbral 15. problem na spletni strani ProjectEuler. Ta zahteva, da najdemo vse poti iz zgornjega levega kota mreže v spodnji desni kot. Najbolj preprost način doseči le to, in hkrati najbolj grafično zanimiv način, je premikanje desno, dokler ne dosežemo desno stranico. Takrat stopimo eno nazaj in spet nadaljujemo desno. Ko dosežemo spodnjo stran mreže se spet premaknemo nazaj levo. Tako preiščemo vse poti.

Da bi dosegel tako premikanje sem uporabil rekurzivne metode. Tu sem metodi rekel, naj prišteje novo pot pod pogojem, da je dosegla čisto desno pot, takrat se naj tudi kliče samo sebe, ampak za eno stranico nižje. Če nismo dosegli desnega roba, spet kličemo isto metodo, vendar za eno stranico bolj desno. Tako sem ustvaril naslednjo metodo, imenovano algo (skrajšava za algoritem), ki jo vidimo na sliki 4.

```
public void algo(int vertical, int horizontal)
{
    if(horizontal<gridSize)
    {
        algo(vertical, horizontal+1);
    }
    else howManyPaths++;
    if(vertical<gridSize && horizontal<gridSize)
    {
        algo(vertical+1, horizontal);
    }
}
```

Slika 4: Rekurzivna metoda algo

V tej metodi nastopajo štiri spremenljivke, in sicer vertical, horizontal, gridSize in howManyPaths.

Spremenljivka gridSize je celo število in nam pove kako velika je naša kvadratna mreža in sicer koliko kvadratov in ne koliko stranic. Pri tem ima mreža 20x20 na razpolago 21 poti premikanja, saj ima vsaka celica dve stranici v obe smeri, kjer se stranici na robih mreže ne prekrivajo. Na srečo nam to programa ne zakomplicira, saj zanemarimo desno stranico in hkrati nočemo povečati vertikalnega premika, če smo na spodnji stranici.

Spremenljivka vertical je celo število in nam pove kako visoko se nahajamo po mreži, kjer je 0 čisto zgornja stranica, gridSize pa čisto spodnja stranica.

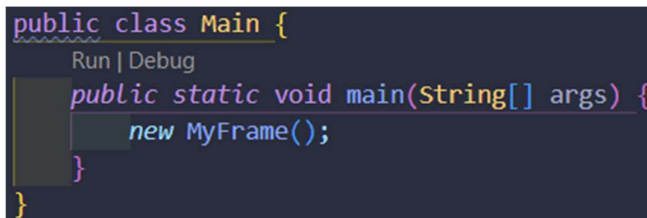
Spremenljivka `horizontal` je celo število in nam pove kolikokrat smo se pomaknili v desno, kjer levo stranico predstavlja vrednost 0 in desno stranico predstavlja spremenljivka `gridSize`.

Spremenljivka `gridSize` nam pove kako veliko mrežo smo si zastavili.

(20) (12) (1)

2.2.3 OKNO

Ko sem ugotovil kako bi problem rešil, sem začel z grafičnim delom programa. Da začnemo javanski program potrebujemo »main« metodo, ki jo vidimo na sliki 5. Znotraj te ustvarimo nov objekt »MyFrame«, ki podeduje lastnosti `JFrame` razreda iz paketa `javax.swing.JFrame`.



```
public class Main {  
    Run | Debug  
    public static void main(String[] args) {  
        new MyFrame();  
    }  
}
```

Slika 5: Main razred in main metoda.

Znotraj `MyFrame` razreda sem za začetek vstavil osnovne ukaze za prikazovanje okna, kot vidimo na sliki 6.



```
this.setIconImage(icon.getImage());  
this.getContentPane().setBackground(background-color);  
this.setResizable(false);  
this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
this.setPreferredSize(windowDimension);  
this.setLayout(new FlowLayout());  
this.pack();  
this.setLocationRelativeTo(null);  
this.setVisible(true);
```

Slika 6: Ukazi za prikazovanje okna

Oknu sem nastavil ikono, ki sem jo naredil v Photoshopu.

Barvo ozadja sem mu spremenil na črno barvo, ki sem jo shranil v razredno spremenljivko. (21)

Določil sem, da oknu ne moremo spreminjati velikosti.

Ko zapremo okno v Javi, se program ne konča, saj grafične komponente ne predstavljajo celotnega programa. Ukaz »setDefaultCloseOperation« določi, kaj se mora zgoditi, ko zapremo okno. »Jframe.EXIT_ON_CLOSE« je konstanta, ki določa, da se program ob zaprtem oknu terminira, saj v mojem primeru je celoten program uporaben samo v grafični obliki.

Določil sem tudi velikost okna in kako naj so elementi na njem razdeljeni, poravnavo na sredino zaslona in na koncu določil, naj se okno prikaže.

Zdaj ko imamo naše okno, rabimo nanj začetni dodajati elemente. Gumb sem dodal s preprostim ukazom za ustvarjanje objekta, kot vidimo na sliki 7. Da sem jih nato prikazal na oknu sem uporabil na okno .add() ukaz, kakor vidimo na sliki 8. Za razpored gumbov poskrbi zaporedje, v katerem jih dodamo ter FlowLayout, ki smo ga določili pri ustvarjanju okna. FlowLayout razvrsti elemente od leve proti desni, od zgoraj dol.

(22) (14)

```
static JButton startButton = new JButton("Start");
```

Slika 7: Ukaz za ustvarjanje gumba "Start".

```
this.add(submitGridButton);  
this.add(submitField);  
  
this.add(delayButton);  
this.add(delayTextField);  
  
this.add(algoRunnable);  
  
this.add(startButton);  
this.add(howManyPathsField);  
  
this.add(instantButton);  
this.add(instantTextField);  
  
this.add(customizeButton);
```

Slika 8: Ukazi za dodajanje elementov na okno.

Da se ne bi okno slučajno prikazalo brez teh elementov, sem te ukaze tudi napisal v konstruktorju in sicer pred ukazi za ustvarjanje okna.

Na enak način sem poleg gumbov dodal tudi vpisna polja in mrežo, ki jo opišem kasneje. Za vpisna polja sem rabil določiti še velikost, kar sem naredil z ukazom, ki ga vidimo na sliki 9.

```
submitField.setPreferredSize(new Dimension(40,20));
```

Slika 9: Določanje velikosti vpisnemu polju.

(15)

Ko dodamo gumbe, te ne naredijo ničesar, dokler jim ne povemo kaj storiti. Da ne bi delal mnogo posebej metod sem v pomoč uporabil lambda izraze. Te omogočajo ustvarjanje f in metod znotraj teh vse znotraj enega stavka. Več si lahko o njih preberete na virih. Te izrazi zgledajo tako, kot vidimo na sliki 10.

```
delayButton.addActionListener(  
    (e) -> {  
        try{  
            //set delay for thread.sleep() method in MyRunnable  
            MyRunnable.DELAY = Integer.parseInt(delayTextField.getText());  
        } catch(Exception e2){  
            System.out.println(e2);  
        }  
    }  
);
```

Slika 10: Dodajanje funkcionalnosti gumbu za zamik.

(23)

V tem primeru pritisk gumba proba prebrati število, in sicer željen časovni premor med potezami. Ker pa lahko uporabnik v polje vnese tudi črke poleg števil, rabimo preveriti, da lahko vrednost res zapišemo v spremenljivko namenjeno za celo število. Temu v prit uporabimo ukaz try-catch (probaj-ujemi). Ta ukaz proba izvršiti kodo znotraj »try« bloka. Če mu ta uspe, namesto da prekine delovanje programa, se samo premakne na »catch« blok in izvrši kodo znotraj tega. S tem lahko uporabnik vnese tudi besede, brez da bi se program prekinil.

(12)

2.2.4 MREŽA

Zdaj ko imamo vse gumbе rabimo dodati še mrežo. Za začetek naredimo nov razred imenovan GridLabel, ki podeduje lastnosti JLabel in hkrati implementira tudi Runnable. JLabel deluje kot prostor na katerega lahko rišemo, medtem ko nam Runnable omogoča delovanje z več nitmi. To potrebujemo, saj nočemo da izrisovanje mreže prepreči delovanje gumbov ipd.

Znotraj tega razreda tudi posodabljam o polje za najdeno število poti. To bi lahko delali tudi znotraj razreda za algoritem, ampak v splošnem ni pomembno kje se nahaja. Polje za število poti posodobimo z vsakim novim izrisom mreže.

Z naslednjim ukazom nastavimo velikost našega prostora za risanje. Tega sem nastavil na 500x500 pixlov + 1, da ta ni na desni in spodnji strani odrezan. To vidimo na sliki 11.

```
this.setPreferredSize(new Dimension(501,501));
```

Slika 11: Nastavljanje velikosti prostora za mrežo.

Ko imamo naš prostor, rabimo nanj začeti risati. Implementacij Runnable vmesnika nam doda metodo run. Ta se pokliče, ko program prvič poženemo. Znotraj te kličemo metodo repaint(), ki po dolgi poti pokliče metodo paint(Graphics g). repaint() kličemo samo enkrat znotraj GridLabel. Po tem, se kliče samo še znotraj razreda MyRunnable, ki ga opišem kasneje.

(22)

Znotraj paint metode imamo objekt `g` razreda `Graphics`. Tega pretvorimo v objekt razreda `Graphics2D`. Ta razred nam omogoča risanje preprostih oblik na naš prostor. S pomočjo tega objekta, ki sem ga poimenoval `g2D`, narišemo `gridSize + 1` vertikalnih in horizontalnih črt. Razmik med njimi tudi določimo preko spremenljivke `gridSize`. `GridSize` spremenljivka se nastavi vsakič, ko uporabnik pritisne gumb za vnos velikosti mreže. Privzeta vrednost te spremenljivke je 1. Mreži tudi nastavimo barvo. Te ukaze lahko vidimo na sliki 12.

```
public void drawStartGrid(int gridSize, Graphics2D g2D){
    g2D.setPaint(lineColor);
    for(int i=0; i<=gridSize; i++){
        int x1 = 0;
        int x2 = 500;
        int y1 = (int) Math.round(i*LINE_SIZE);
        int y2 = y1;
        g2D.drawLine(x1, y1, x2, y2);
    }
    for(int i=0; i<=gridSize; i++){
        int y1 = 0;
        int y2 = 500;
        int x1 = (int) Math.round(i*LINE_SIZE);
        int x2 = x1;
        g2D.drawLine(x1, y1, x2, y2);
    }
}
```

Slika 12: Izris začetne mreže.

(22)

2.2.5 ALGORITEM

Sedaj imamo naše okno, gumbe in mrežo. Čas je da sprogramiramo še algoritem, ki bo celoten program usposobil. Sam algoritem smo naredili pred vsem drugim, vendar rabimo preprosto rekurzivno metodo sedaj pretvoriti v nekaj grafičnega. Da sem to naredil sem samo nadgradil osnovno metodo, ki sem jo naredil na začetku.

Za začetek sem naredil nov razred imenovan MyRunnable. Ta razred podeduje lastnosti razreda GridLabel, saj bo ta risal na isto mesto kot GridLabel.

(23) (24)

Algoritem

Znotraj tega sem ustvaril metodo algo(int vertical, int horizontal) v kateri so vsi enaki ukazi kot na začetku. Metodi sem dodal le pregled, če želimo, da se ta neha izvajati, kar pa nadziramo s spremenljivko shouldReturn. Če je ta nastavljena na »true«, se metoda prekine takoj, ko je klicana in hkrati tudi takoj, ko iz nje izstopimo. Metoda je rekurzivna, zato je velika verjetnost, da ko jo prekinemo, se vrne v samo sebe. Tam ji moramo tudi ukazati izhod.

V metodo sem dodal tudi uspavanje niti, ki traja tako dolgo, kot je nastavil uporabnik. Ta vrednost je privzeto 100 milisekund. Vsakič ko vstopimo v metodo, če program ni ustavljen, počakamo toliko časa.

(25)

V metodo, pred vsakim vhodom in po vsakem izhodu iz metode shranimo vrednost spremenljivk vertical in horizontal v spremenljivko coordinates (Slovensko: koordinati). To spremenljivko nato pošljemo v metodo, ki koordinate zabeleži, da jih lahko kasneje izrišemo. Če uporabnik ni izključil črte, ki kaže trenutno sled, zapišemo iste koordinate v še v en ločen seznam črt, vendar te tudi izbrišemo ob izhodu. Poleg tega se koordinate pošljejo tudi v metodo, ki riše »glavo« te črte. »Glava« je privzeto modra črtica, ki kaže kje se trenutno nahaja program na mreži.

Shranjevanje črt

Omenil sem, da pošiljamo koordinate v metodo, ki te podatke shrani. Ta metoda se imenuje `addLineToHash(String coordinates)`. Te metode prejete koordinate razdeli na 4 komponente, in sicer `x1`, `x2`, `y1`, `y2`. Vsaka črta ima 4 koordinate, ki določajo njeno pozicijo. Začetek črte ima dve koordinati in konec črte tudi dve. Te koordinate shranimo v objekt razreda `HashMap`. Ta deluje kot neurejena tabela, ki nima vnaprej določene velikosti. V tem primeru je `HashMap` uporaben, saj lahko v njem shranjujemo podatke skupaj z imenom in do njim dostopamo brez časovne zahtevnosti. Ime nam prav pride, saj lahko za ime nastavimo kar nit koordinatov, ki smo jih poslali v metodo in preden dodamo koordinate v `HashMap`, preverimo če se te že nahajajo znotraj tega. Ko izrisujemo črte, izrišemo čisto vse črte v tem `HashMapu`, zato nočemo podvojenih rezultatov, ker bi nam to program močno upočasnilo.

V isti metodi dodamo tudi vrednosti črte poti v `ArrayList`, če je ta črta vključena. Tu uporabimo `ArrayList`, saj rabimo vedeti katera vrednost je bila dodana nazadnje ter želimo to vrednost tudi odstraniti, ko izstopimo iz metode `algo()`.

V `HashMap` in `ArrayList` vrednosti dodajamo preko razred `Line`. Ta razred sem tudi naredil sam. V njega lahko shranjujemo vse štiri koordinate črte ter tudi njihovo barvo.

Metoda koordinate prejme v spremenljivki tipa `nit`, kjer so posamezna števila ločena s simbolom, ki sem ga določil vnaprej. Ta simbol je v mojem primeru »@«. Te koordinate po tem simbolu razdelimo v novo spremenljivko, ki je tabela celih števil. Ta števila nato pomnožimo z vrednostjo spremenljivke `LINE_SIZE`. Tej spremenljivki vrednost določimo s preprosto enačbo:

$$LINE_SIZE = \frac{500}{gridSize}$$

S tem razdelimo naš prostor za risanje na toliko enakih delov, kot jih potrebujemo. 500 izberemo, saj je to velikost našega polja, ki smo jo določili že v razredu `GridLabel`. Spremenljivko `LINE_SIZE` v resnici izračunamo kar znotraj razreda `GridLabel`, ampak to nastavimo kot statično, zato jo lahko beremo tudi znotraj razreda `MyRunnable`.

(26)

Ko dobimo pravilne velikosti črt, jih dodamo, skupaj z začetno bravo, v objekt razreda Line in to nato zapišemo v HashMap ali ArrayList, kot sem omenil na začetku. Ta ukaz zglada tako, kakor je prikazano na sliki 13.

```
if(!(lines.containsKey(coordinates))){  
    lines.put(coordinates, new Line(tempx1, tempy1, tempx2, tempy2, initColor));  
}
```

Slika 13: Dodajanje nove črte v HashMap

Na sliki se tudi vidi, da preverimo, da naš HashMap teh koordinatov slučajno ne vključuje.

Program deluje na principu več niti. To pomeni, da deluje več delov programa naenkrat. Risanje in računanje deluje na različni niti, zato je možno, da medtem ko poskušamo dodati črto v HashMap, želi ta isti HashMap brati nit za risanje. V tem primeru bi prišlo do napake in program bi prenehal delovanje. Omenil sem že, da uporabimo try-catch, da se izognem napakam, vendar v tem primeru, če bi prišlo do napake, bi try-catch program nadaljeval in izgubili bi vrednost trenutne črte. Da se temu izognem, sem celoten try-catch postavil v while zanko. Ta ponavlja dodajanje koordinato v HashMap dokler nam to ne uspe in se spremenljivka »succeeded« ne spremenim v »true«. Od sedaj naprej bom ta postopek na kratko opisoval kot »preverjanje, ali nam je uspelo«.

Posebna črta je čisto desna črta. Znotraj metode algo imamo števec, ki preveri, kolikokrat se je metoda že ponovila. Če se je ta ponovila gridSize krat, potem dodamo v hashMap še čisto desno črto, saj to pomeni, da je program tisto stran dosegel. Od takrat dalje je tista črta vedno narisana.

(27) (12)

Razred Line

Pri vnosu koordinat sem omenil razred Line. Ta razred je bil narejen, da lahko za vsako črto posebej shranjujemo pozicijo in barvo. Če ne bi naredil tega razreda, bi rabil imeti HashMap za vsako koordinato in barvo črte posebej, sedaj pa rabimo samo en HashMap, ki pa shranjuje objekte razreda Line.

Razred je precej osnoven, saj je namenjen samo shranjevanju podatkov. Te podatki so taki, kot vidimo na sliki 14.

```
class Line {  
    int x1;  
    int x2;  
    int y1;  
    int y2;  
    Color color;  
    double[] doubleColor = new double[3];  
}
```

Slika 14: Razredne spremenljivke razreda Line

Vidimo, da poleg omenjenih koordinat in barve, shranjujemo še eno tabelo realnih števil. To uporabljamo, saj bomo kasneje opisovali tudi animacijo spreminjanja barv črt iz ene barve v drugo. Če to želimo doseči, rabimo vsako vrednost RGB modela posebej spreminjati in to tudi natančno, da lahko dobimo gladke animacije.

Poleg razrednih spremenljivk imamo v razredu tudi konstruktor za shranjevanje teh ter tudi metode za klic in spreminjanje podatkov.

(16)

Risanje

Zdaj ko imamo vse črte shranjene v različnih tabelah jih rabimo še izrisati. Ker smo vse uredili že prej ta postopek ni tako zahteven. Podobno kot v razredu GridLabel uporabimo metodo `paint(Graphics g)`. V metodi algo kličemo `repaint()` vsakič ko vstopimo v program. Ta `repaint()` velja tudi za GridLabel, zato sem v opisu GridLabel razreda omenil, da `MyRunnable` poskrbi za ostalo.

V metodi `paint` imamo samo dve zanki, ena riše vse črte, ki smo jih že obiskali, druga pa riše trenutno pot. Obe zanki damo v prej omenjen sistem za preverjanje, če nam je uspela izvršitev.

Ker se trenutna pot stalno posodablja, ima ta večjo verjetnost, da bo povzročila napako. Da se izognemo nepotrebnim upočasnitvam pri izvršitvi programa, damo števec, katero črto smo že izrisali ven iz tega sistema za preverjanje napak. Vedno ko se zgodi napaka, nadaljujemo z risanjem črt iz tam, kjer smo se nazadnje ustavili. Tega pri risanju vseh ostalih črt ne rabimo storiti, saj je tam verjetnost napake veliko manjša.

Poleg teh dveh zank pa znotraj te metode tudi kličemo metodi `drawLeadLine` in `fadeAnimation`.

Metodo `drawLeadLine` kličemo samo takrat, ko je program v teku. Ta preprosto nariše našo »glavo« na trenutnem mestu programa, na enak način kot so shranjene in narisane ostale črte.

Metoda `fadeAnimation` pa doda animacijo, ki sem jo omenjal pri opisu razreda `Line`. Ta metoda vsako ponovitev izrisa barvo črt počasi pretvori iz ene v drugo. Uporabnik lahko določi katere barve postanejo črte, ko gre program skozi njih. Ko smo črte shranjevali v `hashMap` smo jim tudi vedno dodelili neko začetno barvo, ali pa jim barvo spremenili na to isto začetno barvo. Zdaj to barvo počasi spremenimo v drugo, ki se imenuje »`fadeColor`«. Vse barve v RGB sistemu barv so sestavljene iz rdeče, zelene in modre. Vsaka izmed teh ima lahko vrednost od 0 do 255. Uporabnik določi koliko vrednosti želi da program skoči med temi. Ta metoda nato vzame barvo vsake črte in ji spremeni vrednosti rdeče, zelene in modre za toliko kot je določeno. Hkrati tudi preverimo, da te vrednosti ne presežejo 255 in da niso manjše od 0. Hkrati rabimo preveriti ali želimo vrednost rdeče, modre in zelene povečati ali pomanjšati, glede na RGB vrednost druge barve. Edina črta, ki ji ne spreminjamo barve, je čisto desna črta, saj vemo, da se bo ta stalno posodabljala. Vse znotraj metode damo v naš pregled za pravilno izvršitev.

(16)

Začetek algoritma

Zdaj ko vemo, kako vse deluje, lahko opišem kako se algoritem sploh požene. Ker ta razred, `MyRunnable`, podeduje lastnosti razreda `GridLabel`, podeduje tudi vmesnik `Runnable` in s tem tudi metodo `run`.

V metodo `run` za začetek dodamo par osnovnih ukazov za ponastavljanje programa. Pobrišemo vse črte, ponastavimo števec in podobno. Potem poženemo metodo `algo`, katere delovanje sem opisal. Ko pa se ta konča, se spet vrnemo v metodo `run`. Tu

pobrišemo črto, ki kaže trenutno pot, saj je možno, da je uporabnik sam prekinil delovanje in program ne bi imel priložnosti te pobrisati. Za tem imamo pa zanko, ki se ponavlja, dokler se vse črte ne pretvorijo v željeno barvo. Ta zanka samo kliče metodo repaint, hkrati pa ima par pregledov, da ne pride do napake. Ta zanka ima tudi implementiran zamik, da se ne premika hitreje kot je določil uporabnik. Ko se izvršuje ta del animacije izključimo blokado barvanja desne črte, saj se ta več ne ponavlja.

Ko se konča animacija, kličemo metodo za spremenitev stop gumba v start in za vsak slučaj vse črte pretvorimo v končno barvo, kar moramo tudi dati v sistem za preverjanje pravilne izvršitve, ter ponastavimo trenutne koordinate.

(25)

Poganjanje algoritma v drugih razredih

Da lahko začnemo celoten postopek imamo tudi nekaj logike v MyFrame razredu. Glavna sta ukaza za ustvarjanje niti, na kateri deluje ta razred, ter tudi začenjanje te niti. Za to poskrbijo trije ukazi, prikazani na slikah 15 in 16.

```
static MyRunnable algoRunnable = new MyRunnable();  
static Thread algoThread = new Thread(algoRunnable);
```

Slika 15: Ustvarjanje niti za algoritem

```
algoThread.start();
```

Slika 16: Ukaz za poganjanje niti

Na podoben način začnemo tudi nit za risanje začetne mreže.

(22)

Start/Stop gumb

Uporabnik program požene preko gumba start in s tem istim gumbom program tudi ustavi. Ta gumb mora zaradi tega imeti več različnih funkcionalnosti.

Prva stvar, ki jo naredi gumb, je da spremeni svoje besedilo, glede na to kaj želimo narediti. Za spreminjanje gumba iz stop v start, torej ustavljanje programa, sem dodal svojo metodo. To metodo kličemo tudi, ko se konča izvršitev animacije v `MyRunnable` razredu. Ta metoda se imenuje `startButtonStop`. Znotraj te metode spremenimo besedilo na gumbu, določimo, da smo gumb pravilno spremenili preko spremenljivke `pressedStart`, ki ima vrednost 0, če piše Start in vrednost 1, če piše stop, ter tudi metodi `algo`, ki se nahaja znotraj `MyRunnable` razreda, rečemo da se mora končati preko `shouldReturn` spremenljivke, ki sem jo opisal na začetku.

Če gumb pritisnemo, ko na tem piše »Start«, se požene druga koda. Da preverimo, če res piše »Start«, preverimo, da je vrednost `pressedStart` spremenljivke 0. Ko se tega prepričamo, spremenimo `pressedStart` spremenljivko na 1, ponastavimo število najdenih poti na 0, prekinemo nit za algoritem, spremenimo besedilo na gumbu, ustvarimo novo nit za algoritem ter to poženemo.

(22)

Prekinitev animacije znotraj `MyRunnable`

Omenil sem, da se po končanem izvrševanju algoritma animacija še nadaljuje, dokler se ne pretvorijo vse črte. Če uporabnik želi spremeniti velikost mreže, ta ne želi počakati, da se animacija konča, zato imamo spremenljivko `breakAnimation`, ki to animacijo prekine, ko pritisnemo gumb za spreminjanje velikosti mreže in je njena vrednost spremenjena na »true«.

2.2.6 GUMB FAST

Obstaja verjetnost, da uporabnik ne želi videti celotnega postopka iskanja poti po mreži, ampak želi samo končen rezultat. Temu v namen sem dodal gumb »Fast«, ali »hitro« po slovensko. Ko uporabnik pritisne ta gumb, izvršimo enačbo za končen rezultat. Ta enačba se glasi:

$$\text{štPoti} = \frac{(2 * \text{gridSize})!}{\text{gridSize!} * \text{gridSize!}}$$

Na žalost pa ne moremo programskega jezika prositi da to preprosto izvrši z navadnimi števili. Spremenljivke za cela števila imajo omejitev, kako velika števila lahko shranjujemo, zato sem razvil svoj sistem za računanje z velikimi števili. Števila sem shranjeval kot tabele, kjer ima vsaka številka svoje polje v tabeli. Ta števila sem nato obdeloval preko preprostih osnovnošolskih načinov računanja.

(28) (12)

Prekoračitev

Ko obdelujemo števila je velika verjetnost, da celica preseže število 9, kar je maksimalna vrednost za posamezno številko v številu. Zato sem naredil metodo, ki gre skozi vsa števila v tabeli in preveri, če so ta večja od 9. Če so, jim odreže vse številke razen enice, in te nato prišteje naslednji celici v tabeli. Ta postopek lahko vidimo sprogramiran na sliki 17.

```
static ArrayList<Integer> checkOverflow(ArrayList<Integer> al){
    //Check for each num in the array if it's bigger than 10. If it is, add the tens digit to the next num in the array
    for(int i=0; i<al.size(); i++){
        int currentNum = al.get(i);
        if(currentNum>=10){
            int overflow = currentNum/10;
            currentNum %= 10;
            al.set(i, currentNum);
            //If there is no next num in the array, add a 0 and add it to that
            if(i==al.size()-1){
                al.add(0);
            }
            al.set(i+1,al.get(i+1)+overflow);
            //i--;
        }
    }
    return al;
}
```

Slika 17: CheckOverflow metoda

Množenje

Vsako celico števila pomnožimo z vsako celico drugega števila. Ko to naredimo tudi vsakič kličemo metodo `checkOverflow`, ki poskrbi, da ne računamo napačno. Sprogramirano metodo lahko vidimo na sliki 18.

```
int[][] results = new int[arr2.length][arr1.length*2+1];
for(int i=0; i<arr2.length; i++){
    for(int j=0; j<arr1.length;j++){
        //Multiply each result by the amount it needs to be offset in the basic multiplication
        results[i][j+i] = arr1[j]*arr2[i];
    }
    results[i] = checkOverflow(results[i]);
}

//Add the lines together
int[] result = new int[results[0].length];
for(int i=0; i<results[0].length; i++){
    for(int j=0; j<results.length; j++){
        result[i] += results[j][i];
    }
    result = checkOverflow(result);
}
```

Slika 18: Delovanje metode `multiply`

(29)

Fakulteta

Glaven del naše enačbe je fakulteta. Ker je to tudi prvi del enačbe, lahko začnemo delo s preprostimi celimi števili. Znotraj metode množimo število 1 z vsemi števili med 1 in vnesenim številom. Sproti rezultat razbijemo v celice ArrayLista in tudi na koncu preverimo prekoračitev meja posamezne celice. Celotno metodo lahko vidimo na sliki 19.

```
int[] factorial(int n){
    if(n<0){
        int[] x = {0};
        return x;
    }

    ArrayList<Integer> al = new ArrayList<>();
    al.add(1);
    while(n>0){
        for(int i=0; i<al.size(); i++){
            int currentNum = al.get(i) * n;
            al.set(i, currentNum);
        }
        al = checkOverflow(al);
        n--;
    }

    int[] result = new int[al.size()];
    for(int i=0; i<al.size(); i++){
        result[i] = al.get(i);
    }
    return result;
}
```

Slika 19: Metoda factorial

(30)

Odštevanje

Števila odštevamo zelo preprosto. Vsaki celici prvega števila odštejemo celico na enakem mestu v drugem številu. Ker je enačba v naprej določena in vemo, da ne bomo nikoli dobili negativnega rezultata, za ta del ni implementirane rešitve. Metodo odštevanja lahko vidimo na sliki 20. Ko številom odštejemo pare, pošljemo rezultat v metodo, ki preveri negativno prekoračitev. To metodo lahko vidimo na sliki 21. Če je katerokoli število negativno ga odštejemo 10. Rezultat zapišemo v isto celico, medtem ko od naslednje celice odštejemo 1.

```
if(al1.size()<al2.size()) return convertList(new int[] {-1});

int[] result = new int[al1.size()];
for(int i=0; i<al1.size(); i++){
    int subtraction = 0;
    if(i<al2.size()){
        subtraction = al2.get(i);
    }

    result[i]+=al1.get(i) - subtraction;
}
result = checkNegativeOverflow(result);
```

Slika 20: Metoda za odštevanje

```
int[] checkNegativeOverflow(int[] arr){
    for(int i=0; i<arr.length; i++){
        if(arr[i]<0){
            if(arr.length<=i+1){
                return new int[] {-1};
            }
            arr[i+1]--;
            arr[i]+=10;
        }
    }
    return arr;
}
```

Slika 21: Metoda checkNegativeOverflow

(31)

Deljenje

Ko dobimo fakultete števil, rabimo ta števila še deliti. Deljenje deluje po postopku, ki smo se ga naučili v osnovni šoli. Od prvega števila odvezamo več in več števok, dokler dobljeno število ni večje od drugega števila. Temu novemu številu odštevamo drugo število, dokler rezultat ni negativen. Takrat število zapišemo v rezultat kot novo števko na koncu števila. Ta metoda je prikazana na sliki 22.

```
int[] result = new int[arr1.length];
ArrayList<Integer> remainder = new ArrayList<>();

for(int i=arr1.length-1; i>=0; i--){
    remainder.add(0, arr1[i]);

    result = multiply(result, 10);

    int[] temp = subtract(convertList(remainder), arr2);
    while(temp[0]>=0){
        remainder = subtract(remainder, convertList(arr2));
        result[0]++;
        temp = subtract(temp, arr2);
    }
}
```

Slika 22: Metoda za deljenje

(32)

Končna enačba

Te metode nam skupaj omogočijo izvršitev tega ukaza, ko uporabnik pritisne gumb »Fast«. Celoten ukaz lahko vidimo na sliki 23.

```
instantButton.addActionListener(
    (e) -> {
        String text = arrayToBigNum(divide(factorial(2*gridSize) , multiply(factorial(gridSize) , factorial(gridSize)) ) );
        instantTextField.setText(text);
    }
);
```

Slika 23: Enačba za hiter rezultat

(28)

BigInteger

Kasneje sem ugotovil, da ima programski jezik Java vgrajen razred `BigInteger`, ki deluje po enakem principu kot moj sistem, vendar po veliki verjetnosti hitreje in bolj učinkovito, zato bi lahko svojo implementacijo najverjetneje nadomestil s tem razredom.

(33)

2.2.7 PRILAGAJANJE PROGRAMA

V programu sem želel, da ima uporabnik nadzor nad izgledom in animacijo.

Razred CustomizeFrame

Ta razred tudi podeduje JFrame, enako kot MyFrame. V konstruktorju tega razreda tudi nastavim vse za prikaz okna, vendar pri tem ni nastavljen, da se prikaže na sredini zaslona, ampak malo na desno od glavnega okna. Poleg tega ne nastavimo tega, da se program terminira, ko to okno zapremo, saj tega uporabnik najverjetneje ne želi. Če želimo končati program, rabimo zapreti glavno okno. Na naše okno tudi dodamo vse gumbe in polja.

To je vse tudi prikazano na sliki 24.

```

        this.add(pathToggleButton);
        this.add(backgroundButton);
        this.add(gridButton);
        this.add(leadColorButton);
        this.add(startColorButton);
        this.add(fadeColorButton);
        this.add(pathColorButton);
        this.add(fadeSpeedButton);
        this.add(fadeSpeedField);

        this.getContentPane().setBackground(Color.black);
        this.setResizable(false);
        this.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        this.setPreferredSize(new Dimension(200, 620));
        this.setLayout(new FlowLayout(FlowLayout.LEADING));
        this.pack();
    }

```

Slika 24: Vsebina konstruktorja razreda CustomizeFrame

Prikaz okna ter njegovo pozicijo nastavimo ob kliku na gumb »Customize« (Slovensko: Prilagodi). To je bilo implementirano, kakor je prikazano na sliki 25.

```

customizeButton.addActionListener(
    (e) -> {
        customizeFrame.setVisible(true);
        customizeFrame.setLocation(this.getX()+(int)this.windowDimension.getWidth(), this.getY());
    }
);

```

Slika 25: Nastavitve gumba Customize

Gumbi

Vse gumbe znotraj tega okna nastavimo na enak način kot smo nastavili ostale gumbe na glavnem oknu. Funkcionalnost teh gumbov nastavimo znotraj razreda `MyFrame`, saj nam to omogoči lažje spreminjanje določenih nastavitev.

Pri nastavljanju funkcionalnosti vseh gumbov uporabi tehniko za preverjanje, ali je ukaz uspel, saj se lahko te nastavitve spreminja tudi takrat, ko je program v teku in kliče te vrednosti, ki jih spreminjamo.

Kako sem funkcionalnost nastavil lahko vidimo na sliki 26.

```
customizeFrame.gridButton.addActionListener(  
    (e) -> {  
        boolean succeeded = false;  
        while(!succeeded){  
            try{  
                Color color = JColorChooser.showDialog(null, "Grid color", GridLabel.lineColor);  
                if(color!=null) GridLabel.lineColor = color;  
                repaint();  
                succeeded=true;  
            }catch(Exception e2){}  
        }  
    }  
);
```

Slika 26: Nastavljanje funkcionalnosti gumba `gridButton`

(23)

3 ZAKLJUČEK

Tekom ustvarjanja seminarske naloge sem se naučil ogromno o bolj naprednih straneh Java programiranja, kot so uporaba več niti, uporabnost različnih razredov ter reševanje logičnih problemov. Hkrati mi je uspelo narediti tak program, kot sem si ga zamislil, brez da bi mi kdo pomagal na vsakem koraku.

Končna koda je daleč od perfektna. Dalje bi lahko nadgradil organiziranost le te, bolje bi lahko uporabil principe objektno usmerjenega programiranja ter lahko bi uporabil nekatere že vgrajene in hitrejške metode znotraj jezika Java. Čeprav pa nisem dosegel popolne optimizacije s programom, je končen produkt perfektno delujoč.

S seminarsko nalogo sem te koncepte tudi razložil bralcem. Za razumevanje seminarske naloge je potrebno nekaj predhodnega znanja v programiranju in interpretacije napisane kode, saj nisem prikazal vseh postopkov od začetka do konca, ampak sem te le opisal, da bi jih lahko vsak programer naredil na svoj način.

Program, ki sem ga napisal tudi učinkovito deluje za reševanje 15. problema spletne strani ProjectEuler. Ta zahteva le mrežo velikosti 20x20, kar pa ta program zmore rešiti v manj kot sekundi in je zmožen izračunati končen rezultat za mreže tudi nad velikostjo 5000x5000, zato na ta način tudi preseže originalno zastavljene cilje.

Da sem se naučil vseh potrebnih znanj v Javi sem potreboval precej dedikacije in trdega dela ter potrpežljivosti. Pri programiranju se redko zgodi, da koda deluje točno tako, kot smo si jo zamislili, na prvi poskus. Ampak tudi ko sem rabil kodo analizirati mnogo ur nisem obupal in na koncu mi je uspelo narediti funkcionalni program.

4 ZAHVALA

Rad bi se zahvalil profesorjema in mentorjema Gregorju Medetu in Albertu Zorku, ki sta me čez leta naučila vztrajnosti, trdega dela in seveda programiranja. Ko smo dijaki imeli težave, ali pa smo se počutili nedelavne, sta nam profesorja pokazala, da vedno obstaja pot do cilja.

Rad bi se zahvalil tudi svojemu bratrancu, ki je tudi programer in je vedno bil pripravljen pomagati, ko sem imel težave pri svojih izdelkih.

Koncem bi se rad zahvalil tudi svoji družini, ki me je čez vsa 4 leta spodbujala k osvojitvi svojih ciljev.

5 VIRI IN LITERATURA

1. **Edpresso Team.** What are lattice paths? *educative*. [Online] julij 24, 2020. <https://www.educative.io/edpresso/what-are-lattice-paths>.
2. **Vailshery, Lionel Sujay.** Most used programming languages among developers worldwide, as of 2021. *statista*. [Online] februar 23, 2022. <https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/>.
3. **Oracle.** What is Java technology and why do I need it? *Java*. [Online] https://www.java.com/en/download/help/whatis_java.html.
4. **Mahnič, Viljan.** Programiranje v Javi. *Študentski net*. [Online] december 21, 2018. https://studentski.net/gradivo/ulj_fri_ri3_pr1_sno_programiranje_v_javi_01.
5. **blazingSaddles.** Is Java losing popularity? *stackchief*. [Online] februar 24, 2021. <https://www.stackchief.com/questions/Is%20Java%20losing%20popularity%3F>.
6. **APACHE KAFKA.** APACHE KAFKA. *Kafka*. [Online] 2017. <https://kafka.apache.org/>.
7. **avi122186.** Why is Java 'write once and run anywhere'? *GeekforGeeks*. [Online] Maj 26, 2019. <https://www.geeksforgeeks.org/why-is-java-write-once-and-run-anywhere/>.
8. **Kavčič, Alenka.** KAKO NAREDIMO V JAVI. *študentski net*. [Online] april 2006. https://studentski.net/gradivo/ulj_fri_ri3_pr2_sno_kako_naredimo_v_javi_01.
9. **Gillis, Alexander S. and Lewis, Sarah.** object-oriented programming (OOP). *TechTarget*. [Online] julij 2021. <https://www.techtarget.com/searchapparchitecture/definition/object-oriented-programming-OOP>.
10. **Ryan.** What are libraries in programming? *iDTech*. [Online] september 11, 2020. <https://www.idtech.com/blog/what-are-libraries-in-coding>.
11. **Oxford.** Dictionary. *Oxford Languages*. [Online] <https://www.google.com/search?q=library+meaning&oq=library+&aqs=edge.0.69i59j0i67i457j69i57j0i402j0i67j0i512l3j69i65.1194j0j4&sourceid=chrome&ie=UTF-8>.

12. **W3schools.** Java Tutorial. *W3schools.* [Online] februar 2, 2022. <https://www.w3schools.com/java/>.
13. **Oracle.** Java Downloads. *Oracle.* [Online] avgust 18, 2021. <https://www.oracle.com/java/technologies/downloads/>.
14. **javaTpoint.** Java Tutorial. *javaTpoint.* [Online] 2021. <https://www.javatpoint.com/java-tutorial>.
15. **Oracle.** Java™ Platform, Standard Edition 7. *Java™ Platform, Standard Edition 7.* [Online] junij 24, 2020. <https://docs.oracle.com/javase/7/docs/api/overview-summary.html>.
16. **RapidTables.** RGB Color Codes Chart. *RapidTables.* [Online] https://www.rapidtables.com/web/color/RGB_Color.html.
17. **Bharadwaj, Surendra.** What is the difference between Swing and AWT? *stack overflow.* [Online] maj 29, 2013. <https://stackoverflow.com/questions/408820/what-is-the-difference-between-swing-and-awt#:~:text=AWT%20is%20a%20thin%20layer,work%20with%20compared%20to%20Swing..>
18. **Oracle.** Package javax.swing. *Java™ Platform, Standard Edition 7.* [Online] junij 24, 2020. <https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html>.
19. **Project Euler.** Project Euler. *Project Euler.* [Online] <https://projecteuler.net/about#234>.
20. **GeeksforGeeks.** Recursion. *GeeksforGeeks.* [Online] oktober 11, 2021. <https://www.geeksforgeeks.org/recursion/>.
21. **Taylor, Brandom E.** Setting background color for a JFrame. *stack overflow.* [Online] julij 4, 2009. <https://stackoverflow.com/questions/1081486/setting-background-color-for-a-jframe>.
22. **Code, Bro.** Java tutorial for beginners. [Online] oktober 26, 2020. https://www.youtube.com/watch?v=NBIUbTddde4&list=PLZPZq0r_RZOMhCAyywfnYLLrjiVOkdAI1.

23. **Oracle**. The Java™ Tutorials. *Java™ Documentation*. [Online] marec 4, 2022. <https://docs.oracle.com/javase/tutorial/index.html>.
24. **Programiz**. What is an Algorithm? *Programiz*. [Online] <https://www.programiz.com/dsa/algorithm>.
25. **Aravind, Anju**. How do I make a delay in Java? *stack overflow*. [Online] junij 8, 2014. <https://stackoverflow.com/questions/24104313/how-do-i-make-a-delay-in-java>.
26. **mkyong**. How to split a string in Java. *Mkyong*. [Online] februar 9, 2022. <https://mkyong.com/java/java-how-to-split-a-string/>.
27. **Java™ Platform, Standard Edition 8**. [Online] januar 5, 2022. <https://docs.oracle.com/javase/8/docs/api/overview-summary.html>.
28. **RudyPenteado**. ProjectEuler thread 15. *ProjectEuler*. [Online] avgust 11, 2004. <https://projecteuler.net/thread=15#234>.
29. **Furey, Edward**. Long Multiplication Calculator. *CalculatorSoup*. [Online] februar 26, 2022. <https://www.calculatorsoup.com/calculators/math/longmultiplication.php>.
30. **The Editors of Encyclopaedia Britannica**. factorial. *Britannica*. [Online] oktober 18, 2013. <https://www.britannica.com/science/factorial>.
31. **Explains, Professor Dave**. Addition and Subtraction of Large Numbers. *YouTube*. [Online] avgust 13, 2017. <https://www.youtube.com/watch?v=YFyOsvnr9ig&t=156s>.
32. **wikiHow**. How to Do Long Division. *wikiHow*. [Online] februar 23, 2022. <https://www.wikihow.com/Do-Long-Division>.
33. **Bright, Sean**. Java equivalent of unsigned long long? *stack overflow*. [Online] februar 3, 2009. <https://stackoverflow.com/questions/508630/java-equivalent-of-unsigned-long-long>.
34. **MadProgrammer**. Get location of a swing component. *stack overflow*. [Online] september 25, 2012. <https://stackoverflow.com/questions/12589824/get-location-of-a-swing-component>.

6 STVARNO KAZALO

barva, 4, 7, 10, 14, 16, 17, 18, 19, 20
črta, 7, 14, 15, 16, 17, 18, 19, 20, 21
gumb, 4, 6, 7, 11, 12, 13, 14, 15, 20, 21,
22, 26, 28, 29
Java, 4, 1, 2, 3, 4, 11, 27, 30
JFrame, 10, 28
knjižnica, 2, 3, 6
koordinati, 15, 16, 17, 20
metoda, 4, 9, 10, 13, 14, 15, 16, 17, 18,
19, 20, 21, 22, 23, 24, 25, 26, 30
mreža, 4, 2, 6, 7, 8, 9, 10, 12, 13, 14,
15, 20, 21, 22, 30
objekt, 3, 4, 10, 11, 14, 16, 17
objektno usmerjeno, 3
Objektno usmerjeno, 3
paket, 3, 4, 10
pot, 2, 7, 8, 9, 13, 16, 18, 19, 21, 22, 31
program, 4, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
11, 12, 13, 15, 16, 17, 18, 19, 21, 28,
29, 30
programer, 30, 31
programiranje, 1, 2, 3, 4, 5, 30, 31
programski jezik, 1, 2, 3, 22, 27
razred, 4, 10, 13, 14, 15, 16, 17, 18, 19,
20, 21, 27, 28, 29, 30
rekurzivno, 9, 15
RGB, 4, 18, 19
spremenljivka, 4, 9, 10, 12, 14, 15, 16,
17, 18, 21
ukaz, 3, 4, 10, 11, 12, 13, 15, 17, 20, 29
uporabnik, 3, 4, 7, 12, 14, 15, 19, 21,
22, 26, 28

7 PRILOGE

<https://github.com/nik1178/LatticePaths>

Priloga 1 – Povezava do programske kode