VISOKA ŠKOLA ELEKTROTEHNIKE I RAČUNARSTVA STRUKOVNIH STUDIJA

Dr Perica S. Štrbac, dipl. ing
Master Vukman Korać, dipl. ing
Spec. struk. ing. Dušan Marković

OBJEKTNO PROGRAMIRANJE 2
PRIRUČNIK ZA LABORATORIJSKE VEŽBE

Prvo izdanje

Beograd, 2019.

OBJEKTNO PROGRAMIRANJE 2 - PRIRUČNIK ZA LABORATORIJSKE VEŽBE

Autor: dr Perica S. Štrbac, dipl. ing,
master Vukman Korać, dipl. ing.
spec. str. ing. Dušan Marković

Recenzenti: dr Miroslav Lutovac, dipl. ing.
mr Miloš Pejanović, dipl. ing.

Izdavač: Visoka škola elektrotehnike i računarstva strukovnih studija, Beograd, Vojvode Stepe 283, www.viser.edu.rs

Za izdavača: dr Vera Petrović, dipl. ing.

Tehnička obrada: dr Perica Štrbac, dipl. ing.

Dizajn korica: dr Perica Štrbac, dipl. ing.

Godina izdanja: 2019.

Tiraž: 80 primeraka

Štampa: Razvojno-istraživački centar grafičkog inženjerstva TMF, Beograd.

ISBN 978-86-7982-310-6

Odlukom Nastavno-stručnog veća Visoke škole elektrotehnike i računarstva strukovnih studija iz Beograda, na sednici održanoj dana 21.02.2019. godine, odobreno je izdavanje i primena ovog priručnika u nastavi.

CIP - Каталогизација у публикацији
Народна библиотека Србије, Београд

004.42.045(075.8)(076)

ШТРБАЦ, Перица С., 1968-
Objektno programiranje 2 : priručnik za laboratorijske vežbe / Perica S. Štrbac, Vukman Korać, Dušan Marković. - 1. izd. - Beograd : Visoka škola elektrotehnike i računarstva strukovnih studija, 2019 (Beograd : Razvojno-istraživački centar grafičkog inženjerstva TMF). - 129 str. : ilustr. ; 30 cm

Tiraž 80. - Bibliografija: str. 129.

ISBN 978-86-7982-310-6

1. Кораћ, Вукман, 1963- [autor] 2. Марковић, Душан, 1994- [autor]

а) Објектно оријентисано програмирање -- Вежбе

COBISS.SR-ID 277463820

Predgovor

Priručnik za laboratorijske vežbe iz predmeta Objektno programiranje 2 namenjen je prvenstveno studentima Visoke škole elektrotehnike i računarstva strukovnih studija u Beogradu.

Ovaj materijal je nastao iz materijala za vežbe i predavanja iz predmeta Objektno programiranje 2 koji se sluša u petom semestru osnovnih strukovnih studija na smeru Računarska tehnika. Gradivo je podeljeno u 12 laboratorijskih vežbi i prati gradivo predavanja.

Vežbe pokrivaju gradivo kao što sledi: laboratorijska vežba 1 – Upoznavanje sa razvojnim okruženjem, laboratorijska vežba 2- Klase,konstruktori, instanciranje objekta, statički članovi, konstante, nizovi, laboratorijska vežba 3 – Nasleđivanje, apstraktne klase, interfejsi, polimorfizam, izuzeci, laboratorijska vežba 4: Rad sa tekstualnim datotekama, serijalizacija, moduli, laboratorijska vežba 5: Niti, laboratorijska vežba 6: Sinhronizaciji niti, laboratorijska vežba 7: Korisnički grafički interfejs, AWT biblioteka, laboratorijska vežba 8: AWT osluškivači, laboratorijska vežba 9: Swing biblioteka, laboratorijska vežba 10: Klijent-server arhitektura, mrežno programiranje, laboratorijska vežba 11: JavaFX, laboratorijska vežba 12: JavaFX fxml, upravljanje događajima.

Zahvaljujemo se svima koji su na bilo koji način doprineli da ovaj priručnik bude pred Vama, a posebnu zahvalnost dugujemo kolegama prof. dr Miroslavu Lutovcu, dipl. ing. i mr. Milošu Pejanoviću, dipl. ing, za korisne sugestije koje su uticale na pisanje ovog priručnika.

U ovom izdanju mogući su propusti, tako da su dobrodošle sve dobronamerne sugestije u cilju poboljšanja sledećeg izdanja.

Priručnik za laboratorijske vežbe iz predmeta Objektno programiranje 2 je podržan projektom Ministarstva prosvete, nauke i tehnološkog razvoja Republike Srbije MGKOP – Modernizacija kurikuluma grupe predmeta Objektno programiranje 1 i Objektno programiranje 2 na studijskom programu Računarska tehnika.

U Beogradu, januara 2019.

Autori

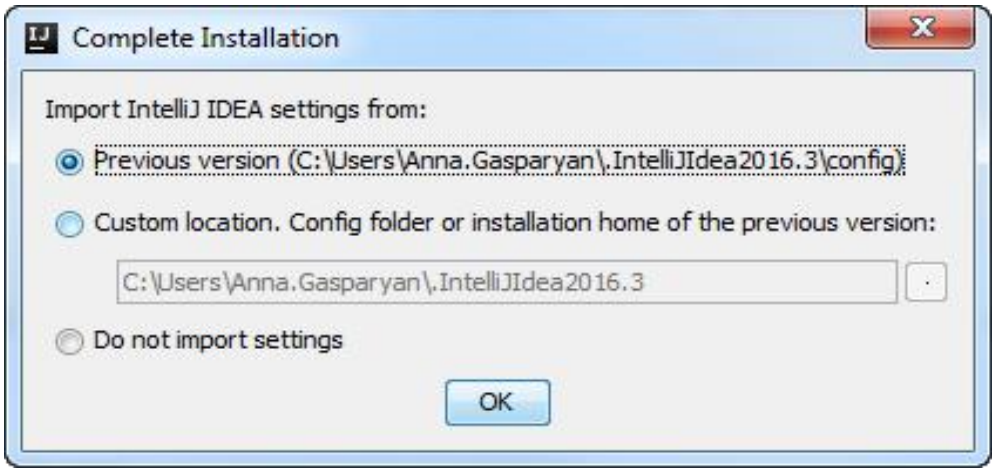
Sadržaj

Laboratorijska vežba 1 – Upoznavanje sa razvojnim okruženjem

Radno ili razvojno okruženje *(eng. IDE- Integrated development environment)* ima za ulogu da programeru pruži pomoć pri pisanju programa u različitim programskim jezicima. Okruženje pruža uvid čitave strukture programa ili čak nekim eksternim sadržajima čime programeru olakšava posao pisanja i održavanja koda. Napisani izvorni kod (eng. source code) prevodi se pomoću kompilatora (eng Compiler, fonetski kompajler) koji se nalazi u samom okruženju i osim prevođenja može da pruži dodatne korisne informacije tokom samog izvršavanja programa. Osim ovih pogodnosti, okruženje nudi pomoć kod samog pisanja koda (IntelliSense). Pomoć može da se ogleda u tome da okruženje samo dopuni ostatak linije ili izgeneriše već poznatu strukturu čime se skraćuje vreme pisanja, prepozna sintaksne greške, preporuči neko od mogućih rešenja itd.

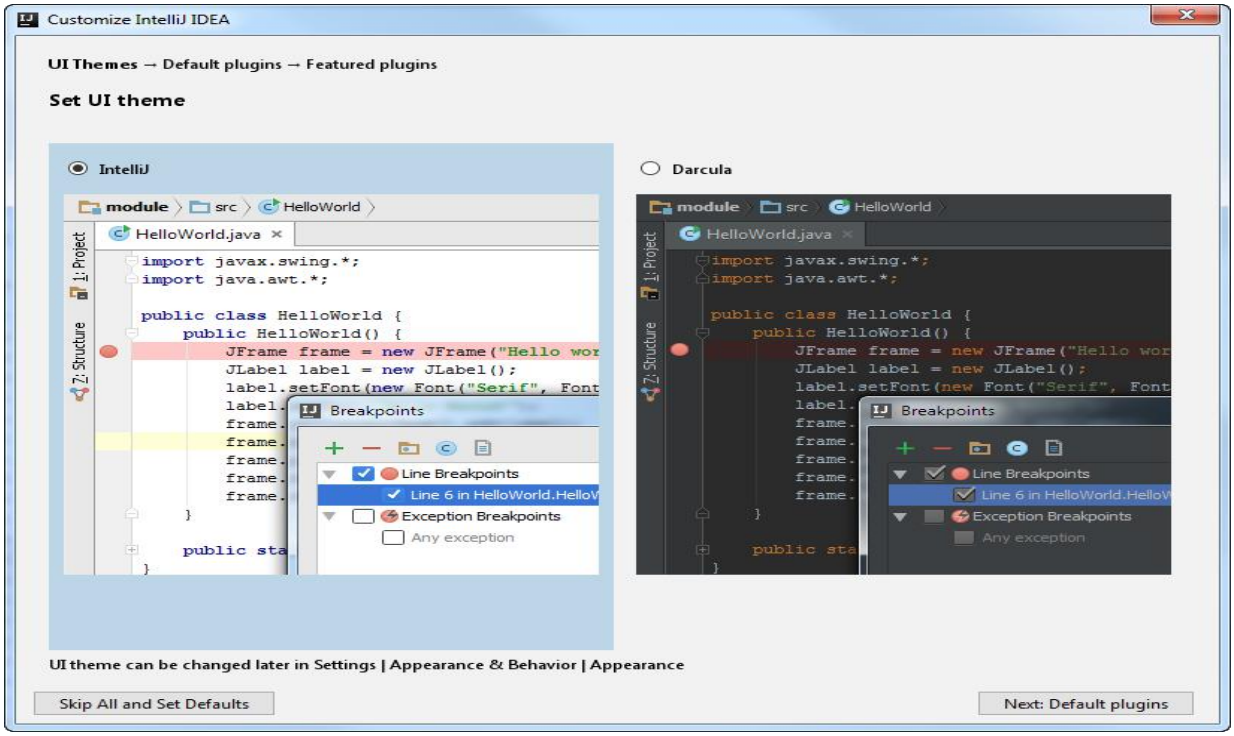
IntelliJ IDEA je jedno od specijalizovanih razvojnih okruženja posebne namene za pisanje Java aplikacija u proizvodnji JetBrains- a. Samo okruženje napisano je u Javi i dostupno je za sledeće operativne sisteme: Linux, Windows i macOS. Hardverski zahtevi su sledeći: Minimum 2 GB RAM memorije (preporučeno 4GB), minimum 2.5 GB memorije na hard disku i minimum 1024x768 rezolucija ekrana. Preuzimanje se vrši sa zvaničnog sajta Jetbrains-a (www.jetbrains.com/idea/download) i moguće je besplatno korišćenje u probnom periodu. Nakon preuzimanja potrebno je instalirati okruženje. Okruženje dolazi u kompletu sa JRE 1.8 tako da ga je moguće pokrenuti bez instalirane Jave na računaru, ali zbog potrebe za JDK-om (Java Devolepment Kit) potrebno je i nju instalirati ukoliko već nije. Preuzimanje Jave vrši se sa zvaničnog sajta na adresi www.java.com/en/download .

Pri prvom pokretanju pojaviće se prozor u kome se od korisnika traži da odabere neku od ponuđenih opcija. Postoji mogućnost ubacivanja podešavanja iz stare verzije (ako je već instalirana na računaru). Sledeća opcija odnosi se na uvoz podešavanja sa određene lokacije, gde je potrebno navesti tačnu putanju do željene datoteke. Ukoliko se korisnik prvi put susreće sa okruženjem preporuka je da se odabere poslednja opcija kojom se ne unose prethodna podešavanja.



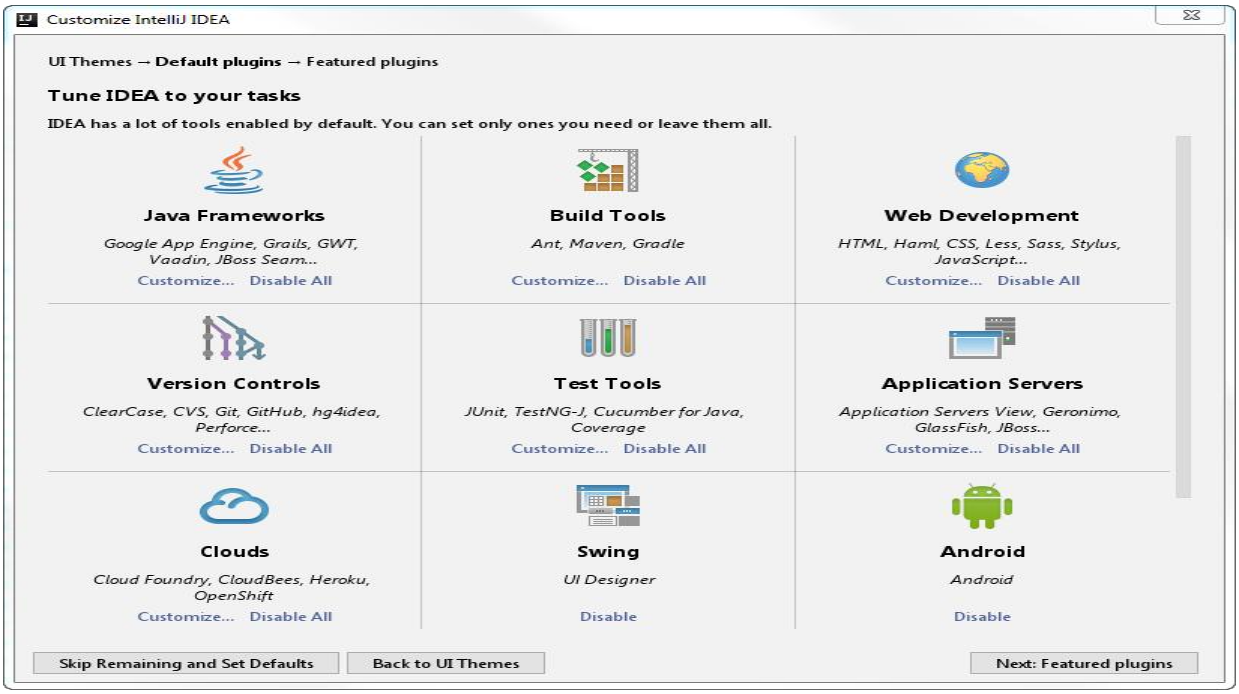
Slika 1.1- Ubacivanje podešavanja okruženja

Na prozoru prakazanom na slici 1.2 korisnik bira jednu od dve ponuđene teme korisničkog interfejsa.



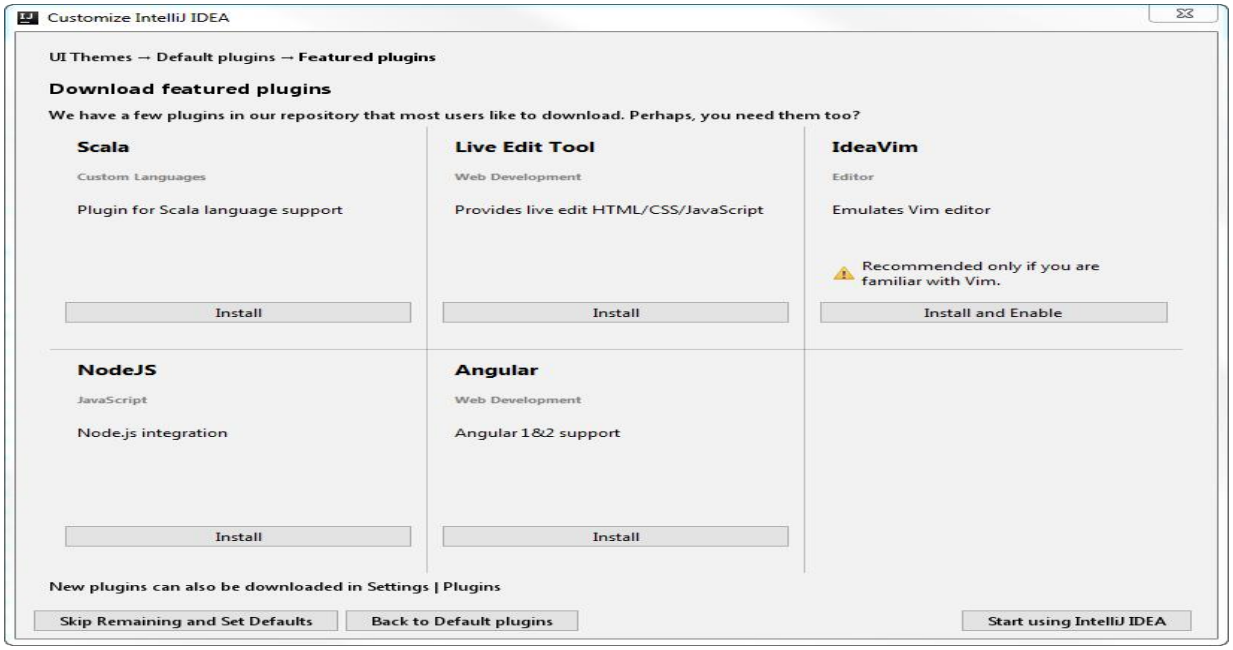
Slika 1.2- Odabir teme okruženja

U sledećem koraku korisnik moze da onemogući neke od programskih dodataka ukoliko ne želi da ih koristi.



Slika 1.3- Onemogućavanje programskih alata

Na sledećem koraku korisnik može dodatno da instalira neke od ponuđenih programskih alata.

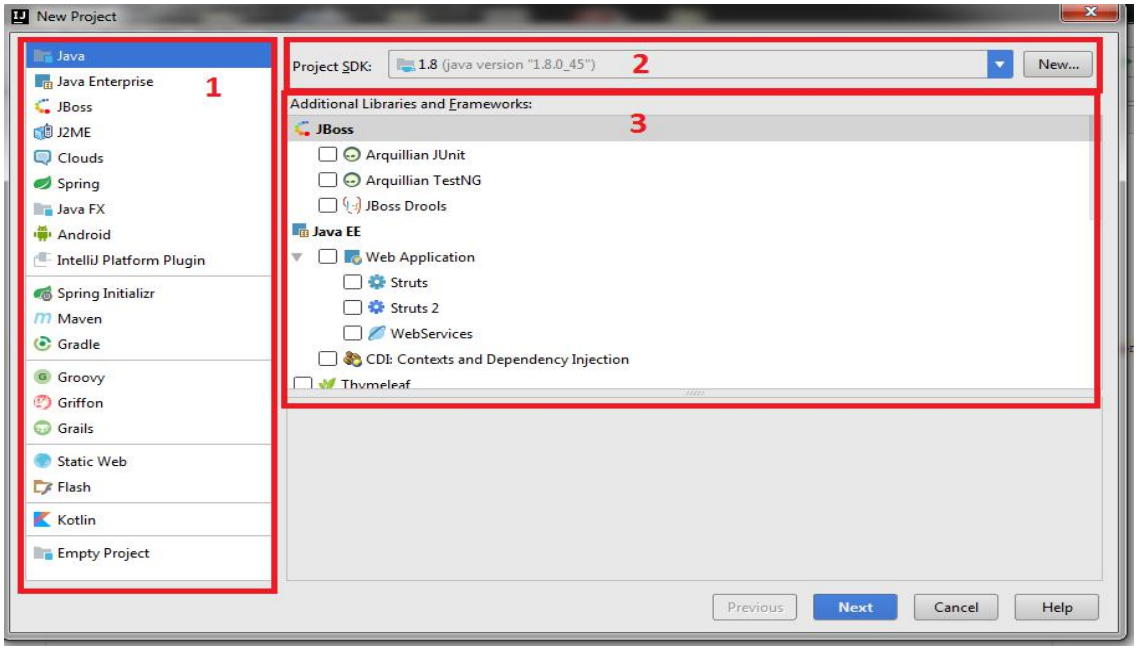


Slika 1.4- Ubacivanje dodatnih programskih alata

Na poslednjem koraku korisnik bira neku od ponuđenih opcija. Odabirom prve opcije počinje proces kreiranja novog projekta, dok su druga i treća namenjene za otvaranje već postojećih.

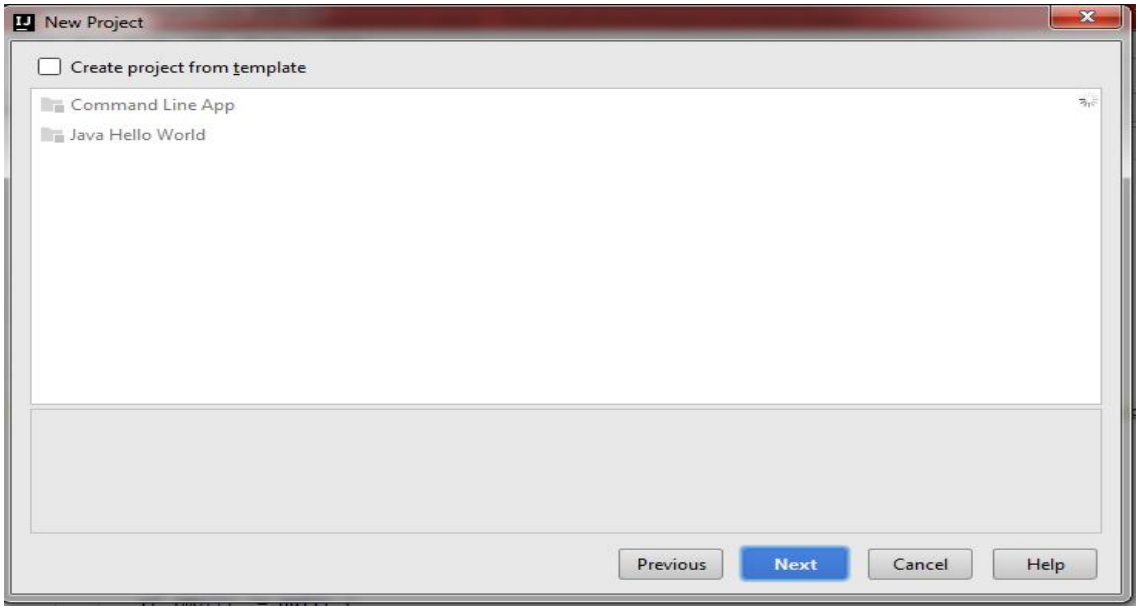
Slika 1.5- Kreiranje/otvaranje projekta

Po odabiru opcije za kreiranje novog projekta pojavljuje se prozor prikazan na sledećoj slici. Naznačeni su segmenti u kojima korisnik treba da odabere neku od opcija.

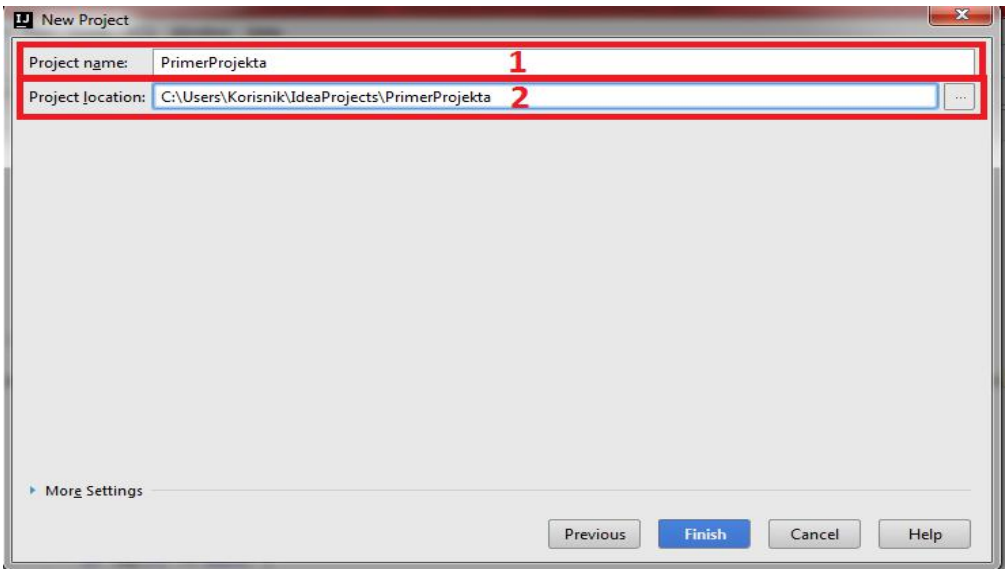


Slika1.6- Odabir tipa/SDK projekta

U prvom segmentu korisnik bira tip projekta, u ovom slučaju Java. Drugi je namenjen za odabir SDK-a (Software Development Kit). Ukoliko nije odabran bira se opcija *New->JDK* nakon čega se navodi putanja gde je instaliran JDK. U trećem segmentu biraju se dodatni programski alati ukoliko su potrebni. Po završetku ovog koraka prelazi se na sledeći pritiskom na dugme Next. U ovom koraku moguće je odabrati neku od opcija za kreiranje projekta po nekom od predefinisanih šablona. Za prazan projekat ne bira se nijedna od opcija.

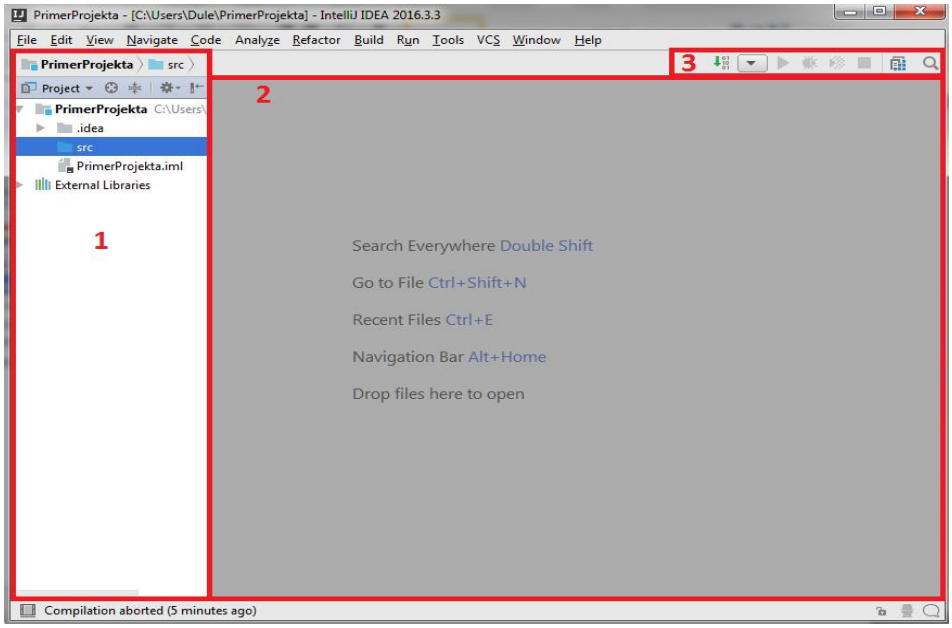


Slika1.7- Odabir šablona projekta



Slika1.8- Dodeljivanje imena/lokacije projekta

U prozoru prikazanom na slici 1.8 naznačena su dva polja u kojima korisnik treba da unese tražene informacije. U prvom unosi naziv projekta dok u drugom upisuje ili bira lokaciju gde će se projekat skladištiti. Kada su polja popunjena pritisnuti dugme Finish kako bi se dala potvrda za kreiranje projekta. Ukoliko je projekat pravilno kreiran prikazuje se prozor sa sadržajem prikazanim na slici 1.9.

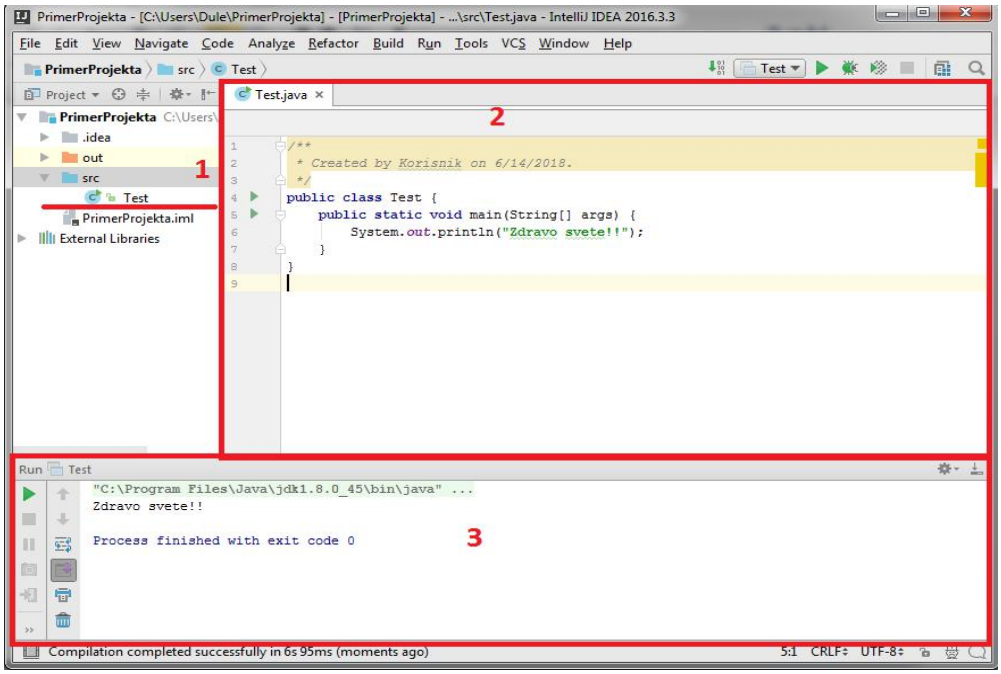


Slika 1.9- Grafički interfejs okruženja

Okvirima su naznačeni glavni segmenti:

1. Navigator koji prikazuje strukturu čitavog projekta.
2. Editor u kome se prikazuje i unosi sadržaj datoteka .
3. Alatke kompajlera pomoću kojih se vrši konfiguracija, pokretanje itd.

Na slici 1.10 prikazan je primer korisćenja tj. testiranja razvojnog okruženja.



Slika 1.10- Testiranje okruženja

U prvom koraku dodali smo novu klasu u okviru ***src*** direktorijuma. Dodavanje se vrši desnim tasterom miša na **src** nakog čega se prikazuje padajuća lista. Iz liste se bira opcija New->Java Class nakon čega se otvara prozor za dijalog u kome se unosi ime klase, u ovom slučaju **Test**.

Nakon toga sadržaj **Test.java** datoteke se prikazuje u editoru gde se unosi sadržaj. Potrebno je napisati **main**() metodu u okviru klase, a zatim pozvati naredbu za ispis u konzoli. Sadržaj **Test.java** datoteke dat je u sledećem delu:

```
public class Test {
    public static void main(String[] args) {
        System.out.println("Zdravo svete!!");
    }
}
```

Nakon toga može se izvršiti konfiguracija u kojoj se navodi klasa u kojoj se nalazi **main()** metoda ili jednostavno da se desnim tasterom miša klikne na samu klasu i iz padajuće liste odabere opcija run. Ovaj postupak će automatski proslediti kompajleru klasu sa **main**() metodom i pozvati je.

Ukoliko nema grešaka u konzoli bi trebalo da se ispiše poruka prosleđena **System.out.println()** metodi. Konzola se nalazi u donjem delu prozora, na slici 1.10 je je označena brojem 3.

Konzolni ispis:

```
Zdravo svete!!
```

Laboratorijska vežba 2- Klase, konstruktori, instanciranje objekta, statički članovi, konstante, nizovi

Klasa predstavlja model objekta koji je realizovan pomoću atributa i metoda. Pre svega moraju se uočiti zajedničke karakteristike koje dele objekti . Na osnovu tih uvida kreira se klasa. Primerak klase naziva se instanca ili objekat klase. Atributi klase opisuju stanja u kojim objekti mogu da se nalaze dok se metodama opisuje njihovo ponašanje. Atributi mogu biti prostog ili složenog tipa. Java razlikuje osam prostih tipova : celobrojne vrednosti (**byte** - 8 bita, **short**- 16 bita, **int**- 32 bita, **long**- 32bita), realne vrednosti (**float-**32 bita, **double**- 64 bita), znakovne vrednosti (**char-** 16 bita), logičke vrednosti (**boolean-** 1 bit) . Dostupnost atributa i metoda definišu se pomoću tri modifikatora : **public, private** i **protected**. Ovo su rezervisane reči koje se pišu ispred atributa ili metode i definišu da li su dostupni na nivou klase (**private**) , javni (**public**) ili su dostupni u okviru klase i u izvedenim klasama te klase (**protected**). Ukoliko se ne navede ni jedan modifikator, član klase ima vidljivost na nivou paketa (**friendly**). Sama klasa može imati dve vidljivosti: na nivou paketa (bez modifikatora) ili na nivou projekta (**public**). Klasa može posedovati i privatnu (**private**) vidljivost ako se radi o unutrašnjoj klasi. Imena atributa moraju biti jedinstvena na nivou klase pri čemu imena mogu sadržati velika i mala slova, numeričke vrednosti i karakter donja crta, s tim da prvi karakter mora biti slovo ili donja crta (npr. ime_Atributa1). Nije dozvoljeno da se ime sastoji samo od karaktera donja crta ("_"). Pravila imenovanja ista su i za klase i metode. Konvencijsko pravilo je da imena klasa počinju velikim slovom, a imena atributa i metoda malim.

Sadržaj **Tacka.java** datoteke:

```
public class Tacka {
    private int x;
    private int y;

    //metoda koja vraca rastojanje izmedju tacke i koordinantnog pocetka
    public double rastojanjeOdCentra(){
        double rastojanje=Math.sqrt(x*x+y*y);
        return rastojanje;
    }

}
```

Osim prostih, tu su i složeni tipovi koji se dobijaju kreiranjem klasa. **String** je tip koji ima složenu strukturu, ali zbog česte upotrebe u Javi se koristi kao prost (instanciranje se može vršiti i bez operatora **new).**

Metode klase opisuju njeno ponašanje tj. definišu operacije koje će moći da se izvršavaju nad objektima. Pri deklaraciji metoda osim modifikatora vidljivosti (dostupnosti) mora se napisati i njen povratni tip. Vraćanje vrednosti vrši se pomoću rezervisane reči **return**. Ukoliko ne vraća nijedan podatak, povratni tip je **void.** Metoda se izvršava do poziva za vraćanje vrednosti ili do kraja bloka ukoliko je bez povratne vrednosti. Parametri koje koristi metoda navode se u okviru zagrada nakon imena same metode. Osim imena parametra potrebno je navesti i kog su tipa (npr. public int imeMetode(String ime, int godiste)).

Konstruktori predstavljaju posebnu vrstu metoda koji se pozivaju pri instaciranju klase. Služe za početnu inicijalizaciju vrednosti atributa objekta. Za razliku od standardnih metoda konstruktori nemaju povratnu vrednost tako da se ne navodi povratni tip, čak ni **void**. Osim toga konstruktori imaju isto ime kao i sama klasa. Ukoliko nije definisan konstruktor, prilikom instraciranja poziva se podrazumevani koji ne prima parametre. Podrazumevani konstruktor postavlja podrazumevane vrednosti za atribute prostog tipa i **null** za složene. Moguće je definisati više konstruktora u jednoj klasi, s tim da im se razlikuju potpisi tj. moraju im se razlikovati parametri (po broju, tipu ili rasporedu). Poziv konstrukora vrši se u kombinaciji sa operatorom **new** koji se piše ispred konstruktora. Ovakav poziv konstruktora kreira objekat u memoriji i vraća njegovu lokaciju. Adresa se može skladištiti u referencu preko koje će se dalje pristupati objektu. Referenca **this** referiše na objekat u kome se nalazi (referiše na samu sebe).

Sadržaj **Osoba.java** datoteke

```
//Primer klase sa definisanim atributima i konstruktorima
package paketPrvi;
public class Osoba{
    private String ime;
    private int godiste;
    //konstruktor 1
    public Osoba(String ime, int godiste){
        this.ime=ime;
        this.godiste=godiste;+
    }
    //konstruktor 2
    public Osoba(String ime){
        this.ime=ime;
        this.godiste=-1;
    }
    //metoda koja  vraca vrednost atributa ime
    public String getIme(){
        return this.ime;
    }
}
```

Sadrzaj **Program.java** datoteke

```
//Primer klase koja sadrzi main metodu u kojoj su prikazani primeri instanciranja
objekata
package glavni;
import paketPrvi.Osoba;
public class Program {
    public static void main(String[] args) {
        Osoba pera=new Osoba("Pera",1994);//instaciranje objekta
        Osoba milan=new Osoba("Milan");//instaciaranje objekta
        System.out.println("Ime prve osobe je"+pera.getIme());

        System.out.println("Ime druge osobe je"+milan.getIme());
    }
}
```

U situacijama gde objekti neke klase dele zajedničko polje, to polje treba definisati kao statički član klase. Statičko polje definišemo rezervisanom rečju **static** u deklaraciji promenljive. Statičko polje kreira se u memoriji pri prevođenju programa, a ne prilikom instanciranja objekta te klase. Svi objekti klase mogu prisupiti statičkom polju, a moguć je pristup i preko imena same klase. Takođe i metode mogu biti statičke i imaju slično ponašanje

kao i statička polja. Statička metoda vezuje se za klasu, a ne za objekte. Unutar tela statičkih metoda nije moguće koristiti nestatička polja klase kao ni referencu **this**. Pozivanje statičkih metoda vrši se preko imena klase(npr. ImeKlase.statickaMetoda()).

Sadržaj **Osoba.java** datoteke

```
public class Osoba {
    private String ime;
    private int godiste;
    private static int brojOsoba=0; //staticko polje koje cuva broj osoba
    public Osoba(String ime,int godiste){
        this.ime=ime;
        this.godiste=godiste;
        /*
        Svaki put kada se instancira objekat(pozove konstruktor)
        brojac se poveca za jedan
         */
        brojOsoba++;
    }
    public static int getBrojOsoba(){
        /*
         return this.brojOsoba; //greska, yabranjeno koriscenje reference this
         */
        return brojOsoba; //ispravno
    }
}
```

Pristup statičkim članovima:

```
Osoba.brojOsoba;// greska, promenljiva brojOsoba nema javnu vidljivost

Osoba.getBrojOsoba(); //ispravno

System.out.println(); //klasa System poseduje javno staticko polje out koje poziva
metodu println()
```

Konstante se u Javi deklarišu pomoću rezervisane reči **final**. Prilikom deklaracije potrebno je izvršiti i inicijalizaciju. Konstante nakon inicijalizacije nije moguće menjati. Metode koje u svojoj deklaraciji imaju **final** ne mogu se preklopiti (redefinisati) u klasama koje nasleđuju(proširuju) tu klasu.

```
private  final int MAX; //Greska, potrebna inicijalizacija

private  final int MAX=5;

private static final double PI=3.14; //staticka konstanta
```

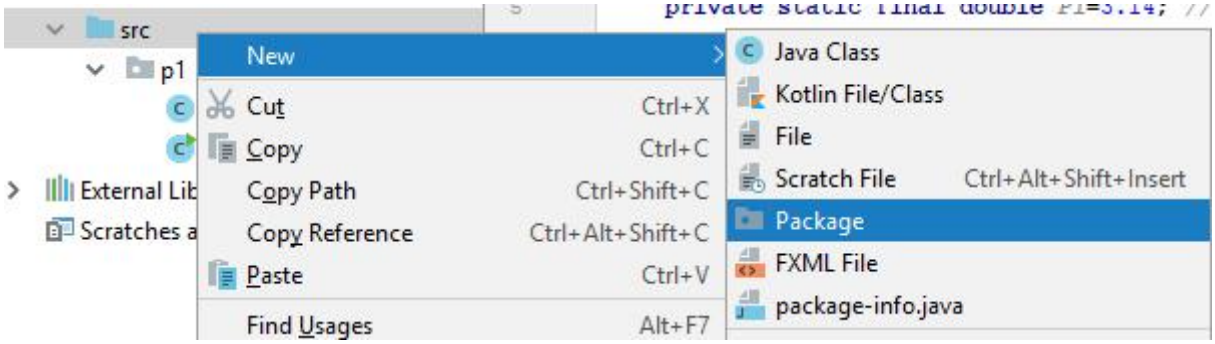
Deklaracija niza u Javi vrši se pomoću uglastih zagrada koje se postavljaju posle imena ili posle tipa samog niza. Kreiranje se vrši pomoću operatora **new** na sličan način kao i kod inicijalizacije objekata. Pri kreiranju potrebno je navesti veličinu niza. Ukoliko se radi o nizu primitivnog (prostog) tipa, alokacija se vrši automatski što nije slučaj kod niza složenih tipova gde se kreiraju samo reference koje imaju vrednost **null**. Dodela vrednosti elementima niza može se izvršiti pomoću vitičastih zagrada između kojih se navode vrednosti elemenata odvojene zarezom. Na ovaj način vrši se definicija, alokacija i inicijalizacija niza u jednom koraku. Prvi član niza ima indeks 0. Višedimenzionalni nizovi predstavljaju se kao nizovi nizova.

```
int niz[]=new int[5]; //deklaracija i definicija niza koji ima 5 clana tipa int
System.out.println(niz[0]); //prikaz prvog clana niza niz
Osoba[] osobe=new Osoba[3];//deklaracija i definicija niza koji ima 3 clana tipa Osoba
System.out.println(osobe[0]); //prikaz prvog clana niza osobe
int[] niz2={3,4,5};
System.out.println(niz2[1]); //prikaz drugog clana niza niz2
Osoba osobe2[][]={{new Osoba("Pera",1990)},{new Osoba("Miki",1998),new
Osoba("Fica",1997)}}; // deklaraija dvodimenzionalnog niza 1X1 sa definicijom i
inicijalizacijom
System.out.println(osobe2[1][0].vratiIme()); //prikaz prvog elementa drugog podniza
niza osobe2
```

Konzolni izlaz:

```
0
null
4
Fica

Process finished with exit code 0
```

Paketi imaju ulogu da grupišu klase po celinama kako bi bila preglednija struktura programa. Pre deklaracije klase potrebno je navesti kom paketu pripada u formatu ***package*** *putanja.do.klase;*. Paket u suštini predstavlja direktorijum u koji je klasa smeštena. Svaka klasa mora pripradati nekom paketu. Značaj grupisanja klasa po paketima dolazi do izražaja kod pisanja složenijih aplikacija zbog velikog broja klasa koje se koriste. Dve ili više klasa mogu deliti isto ime, ali je uslov da se nalaze u različitim paketima. Preporuka je da se uvek vrši grupisanje po paketima čak i kod manje složenih programa.



Slika 2.1- Dodavanje novog paketa

Ukoliko u okviru klase koristimo one koje se ne nalaze u istom paketu potrebno ih je ubaciti (importovati) u tu klasu. Ubacivanje se vrši pozivom ***import*** *putanja.do.klase.NazivEksterneKlase;* pre same deklaracije klase. Dodavanje paketa vrši se tako sto se klikne desnim tasterom miša na ***src*** ili na neki drugi paket i iz padujeće liste bira se opcija **Package**. Nakon toga otvoriće se prozor za dijalog u kom je potrebno da se unese ime paketa, a zatim da se potvdi klikom na dugme OK.

Komentari su sadržaj koda koji kompajler ne prevodi, a služe za opisivanje linije ili dela koda kako bi programer imao pojašnjenje o samom kodu . Razlog korišćenja komentara je možda na prvi pogled neshvatljiv, ali zapravo su dosta korisni. Ukoliko korisnik čita kod koji nije on napisao biće mu potrebno više vremena da razume šta je napisano ukoliko nisu pisani komentari. Razumevanje koda može predstavljati poteškoću i samom autoru posle nekog vremena. Smernice : Pisati koncizne komentare (kratko i jasno), pisati šta neki deo radi ne kako radi, ne komentarisati očigledne delove. Postoje dva tipa komentara: linijski i blokovski.

Primer:

```
// ovo je linijski komentar//

/*      ovo je
        viselinijski (blokovski)
        komentar
                */
```

Zadatak:

U paketu "**vozila**" napraviti klasu **Automobil**. Klasa opisuje vozilo automobil pomoću atributa: registracioni_broj, model, godiste, zapremina_motora. Registracioni broj generiše se tako sto se na string "SR" doda redni broj automobila. Unutar klase postoji brojač koji računa koliko se automobila instanciralo. Klasa poseduje dva konstruktora od kojih prvi prima sve potrebne parametre za postavljanje vrednosti svih atributa, a drugi prima vrednosti za model i godište dok zapreminu postavlja na vrednost -1.0. Klasa poseduje metodu ***ispisi()*** koja ispisuje automobil u formatu - *Automobil {registracioni_broj, model , zapremina)* i metodu ***brojAutomobila()*** koja ispisuje ukupan broj automobila.

U paketu "**saobracaj**" kreirati klasu **Parking**. Klasa poseduje sledeće atribute: adresa, broj_redova, broj_kolona, automobili . Atribut "automobili" je niz objekata klase **Automobil**, a veličina se određuje formulom (broj_redova * broj_kolona). Konstruktor prima parametre uz pomoć kojih se postavljaju vrednosti svih atributa. Klasa poseduje metode : ***parkiraj*()** koja osim automobila prima i red i kolonu parking mesta (ukoliko je mesto zauzeto ispisati poruku u konzoli), metodu ***isparkiraj()*** koja izbacuje automobil na osnovu prosleđenog mesta (red, kolona), metodu ***ispisi()*** koja ispisuje sve automobile koji su na parkingu.

U paketu "**glavni**" kreirati klasu **Program** u kojoj će se testirati sve metode klasa **Automobil** i **Parking**.

Rešenje:

Slika 2.2- Struktura projekta

Sadržaj **Automobil.java** datoteke:

```
package vozila;
public class Automobil {
    private String registacioni_broj;
    private String model;
    private int godiste;
    private double zapremina_motora;
    private static int brojac=0;
    public Automobil(String model,int godiste,double zapremina_motora){
        brojac++;
        this.registacioni_broj="SR"+brojac;
        this.model=model;
        this.godiste=godiste;
        this.zapremina_motora=zapremina_motora;
    }
    public Automobil(String model,int godiste){
        brojac++;
        this.registacioni_broj="SR"+brojac;
        this.model=model;
        this.godiste=godiste;
        this.zapremina_motora=-1;
    }
    public void ispisi(){
System.out.println("A{"+this.registacioni_broj+","+this.model+","+this.godiste+","+this.
zapremina_motora+"}");
    }
}
```

Sadržaj **Parking.java** datoteke:

```
package saobracaj;
import vozila.Automobil;
public class Parking {
    private String adresa;
    private int broj_redova;
    private int broj_kolona;
    private Automobil automobili[][];
    public Parking(String adresa,int broj_redova,int broj_kolona){
        this.adresa=adresa;
        this.broj_redova=broj_redova;

        this.broj_kolona=broj_kolona;
        automobili=new Automobil[broj_redova][broj_kolona];

    }
    public void parkiraj(int red,int kolona,Automobil a){
        if(red>this.broj_redova || kolona>this.broj_kolona){
            System.out.println("Nepostojece parking mesto!!");
```

```
        }
        else {
            if(automobili[red-1][kolona-1]!=null){
                System.out.println("Mesto zauzeto!!");
            }
            else{
                System.out.println("Automobil uparkiran!!");
                automobili[red-1][kolona-1]=a;
            }
        }
    }
    public void isparkiraj(int red,int kolona){
        if (automobili[red-1][kolona-1]==null){
            System.out.println("Nema vozila na trazenom mestu");
        }
        else {
            automobili[red-1][kolona-1]=null;
            System.out.println("Automobil isparkiran!!");
        }
    }
    public void ispisi(){
        for (int i=0;i<this.broj_redova;i++) {
            for (int j=0;j<broj_kolona;j++){
                if (automobili[i][j]!=null){
                    automobili[i][j].ispisi();
                }
            }
        }
    }
}
```

Sadržaj **Program.java** datoteke:

```
package glavni;
import saobracaj.Parking;
import vozila.Automobil;
public class Program {
    public static void main(String[] args) {
        Automobil a1=new Automobil("Audi a1",2005,1993.2);
        Automobil a2=new Automobil("Golf 2",1976,1392.5);
        Automobil a3=new Automobil("Yugo koral",1997,1223.7);
        Parking p=new Parking("Vojvode Stepe 283,Beograd",4,4);
        p.parkiraj(2,2,a1);
        p.parkiraj(3,1,a2);
        p.parkiraj(1,2,a3);
        System.out.println("Trenutno stanje na parkingu!!");
        p.ispisi();
        p.isparkiraj(2,2);
        System.out.println("Azurirano stanje na parkingu!!");
        p.ispisi();
    }
}
```

Konzolni ispis:

```
Automobil parkiran!!
Automobil parkiran!!
Automobil parkiran!!
Trenutno stanje na parkingu:
A{SR3,Yugo koral,1997,1223.7}
A{SR1,Audi a1,2005,1993.2}
```

```
A{SR2,Golf 2,1976,1392.5}
Automobil isparkiran!!
Azurirano stanje na parkingu:
A{SR3,Yugo koral,1997,1223.7}
A{SR2,Golf 2,1976,1392.5}
```

Zadatak za samostalan rad:

U paketu "geometrija" napisati klase **Tacka**, **Trougao** i **Krug**.

Klasa **Tacka** opisana je pomoću tri atributa: ime, x , y. Ime se odnosi na oznaku koju će tačka imati dok se atributi x i y odnose na vrednosti u koordinatnom sistemu po x i y osi respektivno. Kontstruktor ove klase prima sve potrebne parametre za postavljanje vrednosti svih atributa. Klasa ima metodu ***ispisi()*** koja ispisuje tačku u formatu ime(x,y).

Klasa **Trougao** poseduje tri tačke. Konstruktor klase prima sve potrebne parametre za postavljanje vrednosti svih atributa. Klasa poseduje metode: za računanje površine, za računanje obima i metodu koja za prosleđenu tačku ispituje koja tačka (teme) trougla joj je najbliža.

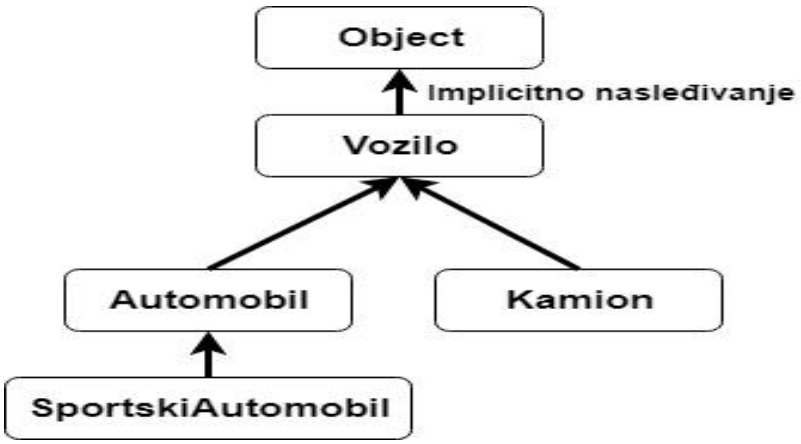
Klasa **Krug** opisana je pomoću tačke i poluprečnika. Kontstruktor klase prima sve potrebne parametre za postavljanje vrednosti svih atributa kao i konstruktor koji prima tačku i vrednost površine kruga. Klasa poseduje metode za računanje površine, obima kao i metodu koja za prosleđeni krug ispituje da li im se kružnice seku u nekoj tački.

U paketu "**glavni**" napisati klasu **Program** u kojoj će biti instancirani objekti prethodnih klasa i testirane njihove metode.

Laboratorijska vežba 3 – Nasleđivanje, apstraktne klase, interfejsi, polimorfizam, izuzeci

Nasleđivanje ili proširivanje neke klase je postupak u kome klasa uzima neke od osobina svoje roditeljske klase (natklase). Postupkom generalizacije utvrđuju se zajedničke osobine koje dele klase. Na osnovu tih uvida se vrši definicija osnovne (roditeljske) klase čime se omogućava bolje ponovno korišćenje koda. Izvedena klasa ima sve atribute i metode roditeljske klase koje nemaju privatnu vidljivost (**private**). Java dozvoljava samo jednostruko nasleđivanje tako da klasa može biti izvedena iz maksimalno jedne klase. Izvođenje se vrši pomoću rezervisane reči **extends** u deklaraciji izvedene klase. Java definiše implicitno nasleđivanje iz klase **Object** sto znači da je ona natklasa svih klasa.

```
public class Vozilo {
/*
definicija klase Vozilo koja prosiruje klasu Object
 */
}
public class Automobil extends Vozilo {
/*
definicija klase Automobil koja prosiruje klasu Vozilo
 */
}
public class Kamion extends Vozilo {
    /*
definicija klase Kamion koja prosiruje klasu Vozilo
 */
}

public class SportskiAutomobil extends Automobil {
        /*
definicija klase SportskiAutomobil koja prosiruje klasu Automobil
 */

}
```



Slika 2.1 – Dijagram zavisnosti klasa

U izvedenim klasama moguće je izvršiti preklapanje (eng. override) metoda iz roditeljske klase koje nisu deklarisane kao konstantne (**final**) ili privatne (**private)** metode. Preklapanje je postupak u kome se redefiniše telo metode u izvedenim klasama. Primer preklapanja može se uočiti kod **toString()** metode koja se nalazi u **Object** klasi. Ova metoda

definiše format u kome će se neki objekat predstaviti pri ispisivanju. U osnovnom obliku ima sledeći format: *putanja.do.paketa.ImeKlase***@***memorijskaLokacijaObjekta.*

Primer :

```
Automobil a1=new Automobil("Audi a1",2005,1993.2);
System.out.println(a1);
```

Konzolni ispis:

```
vozila.Automobil@4554617c
```

Metoda za ispis objekta poziva se implicitno u situacijama kada se zahteva tip **String,** a prosledi se objekat drugog tipa. Ovo ne važi u situcijama kada se vrši konkatenacija (spajanje) stringova. Okruženje će prepoznati preklopljene metode i u prikazu pored broja linije koda pojaviće se simbol koji će na to i ukazivati. Ukoliko se kursor miša pozicionira na njega, prikazaće se i klasa iz koje se vrši redefinicija. Konvencijsko pravilo je da se iznad preklopljene metode piše anotacija **@Override.**

```
26      @Override
27      public String toString(){
28          return "Automobil{"+this.registracioni_broj+","+this.model+
29                  ","+this.godiste+","+this.zapremina_motora+"}";
30
31      }
```

Slika 3.2- Preklapanje toString metode

Nakon preklapanja **toString()** metode sa slike 2.4, ispis objekta klase **Automobil** imaće sledeći oblik: `Automobil{SR1,Audi a1,2005,1993.2}`. Metoda **toString()** nije jedina koju možemo preklopiti, a da se nalazi u **Object** klasi, tu su i:

```
clone(),equals(Object obj),finalize(),getClass(),hashCode(),notifyAll(),wait(),wait(lon
g timeout), wait(long timeout, int nanos).
```

Postupak preklapanja vrši se tako što se prepiše deklaracija metode iz roditeljske klase, a zatim se izvrši nova definicija metode. U preklopljenoj metodi moguće je promeniti i modifikator vidljivosti ili postaviti da je metoda **final** kako bi onemogućili dalju redefiniciju u klasama koje nasleđuju tu klasu.

Prilikom instanciranja objekta prvo se poziva konstruktor natklase, a zatim konstruktor klase. Pristup roditeljskim članovima može se izvršiti pomoću reference **super**. Poziv konstruktora natklase poziva se pomoću ove reference kojoj se prosleđuju parametri koji odgovaraju nekom od potpisa konstruktora roditeljske klase. Nakon toga vrši se inicijalizacija atributa koji su svojstveni za izvedenu klasu.

Sadržaj **Vozilo.java** datoteke:

```
package vozila;
public class Vozilo {
    protected String registracioni_broj;
    protected String model;
    protected int godiste;
    protected double zapremina_motora;
```

```
    public Vozilo(String registracioni_broj, String model, int godiste, double
zapremina_motora) {
        this.registracioni_broj = registracioni_broj;
        this.model = model;
        this.godiste = godiste;
        this.zapremina_motora = zapremina_motora;
    }

    protected void pokreni(){
        System.out.println("Vozilo pokrenuto!!");
    }
    /*preklapanje toString meotode iz klase Object*/
    public String toString() {
        return "registracioni_broj=" + registracioni_broj +
                ", model=" + model +
                ", godiste=" + godiste +
                ", zapremina_motora=" + zapremina_motora ;
    }
}
```

Sadržaj **Automobil.java** datoteke:

```
package vozila;

public class Automobil extends Vozilo{
    private int broj_vrata;
    private int broj_sedista;
    public Automobil(String registracioni_broj, String model, int godiste, double
zapremina_motora, int broj_vrata, int broj_sedista) {
        super(registracioni_broj, model, godiste, zapremina_motora);
        this.broj_vrata = broj_vrata;
        this.broj_sedista = broj_sedista;
    }
    /*preklapanje pokreni meotode iz klase Vozilo*/
    @Override
    public void pokreni(){
        System.out.println("Automobil pokrenut;");
    }
    /*preklapanje toString meotode iz klase Vozilo*/
    @Override
    public String toString(){
        return "Automobil{"+super.toString()+",broj vrata="+this.broj_vrata+
                ", broj sedista="+this.broj_sedista+"}";

}
}
```

Sadržaj **Kamion.java** datoteke:

```
package vozila;

public class Kamion extends Vozilo {
    private Prikolica p;
    private double nosivost;
    public Kamion(String registracioni_broj, String model, int godiste, double
zapremina_motora, Prikolica p, double nosivost) {
        super(registracioni_broj, model, godiste, zapremina_motora);
        this.p = p;
        this.nosivost = nosivost;
    }
   public boolean zakaciPrikolicu(Prikolica p){
        if (p==null) {
            this.p = p;
            return true;
        }
        else{
            return false;
        }
   }
   public void otkaciPrikolicu(){
        p=null;
   }
    /*preklapanje pokreni meotode iz klase Vozilo*/
    @Override
    public void pokreni(){
        System.out.println("Kamion pokrenut!!");
    }

    /*preklapanje toString meotode iz klase Vozilo*/
    @Override
    public String toString() {
        return "Kamion{"+super.toString()+"
prikolica="+p.toString()+",nosivost="+this.nosivost+"}";
    }
}
```

Sadržaj **Program.java** datoteke:

```
package glavni;
import vozila.Automobil;
import vozila.Kamion;
import vozila.Prikolica;

public class Program {
    public static void main(String[] args) {
        Automobil a1=new Automobil("KS123","Golf 5",2001,1598.3,5,5);
        System.out.println(a1);
        a1.pokreni();
        Kamion k1=new Kamion("BG124","Volvo FH460",2010,5605.3,new Prikolica(),20000);
        System.out.println(k1);
        k1.pokreni();
    }
}
```

Konzolni ispis:

```
Automobil{registracioni_broj=KS123, model=Golf 5, godiste=2001,
zapremina_motora=1598.3,broj vrata=5, broj sedista=5}
```

```
Automobil pokrenut!!
Kamion{registracioni_broj=BG124, model=Volvo FH460, godiste=2010,
zapremina_motora=5605.3
prikolica=vozila.Prikolica@4554617c,nosivost=20000.0}
Kamion pokrenut!!
```

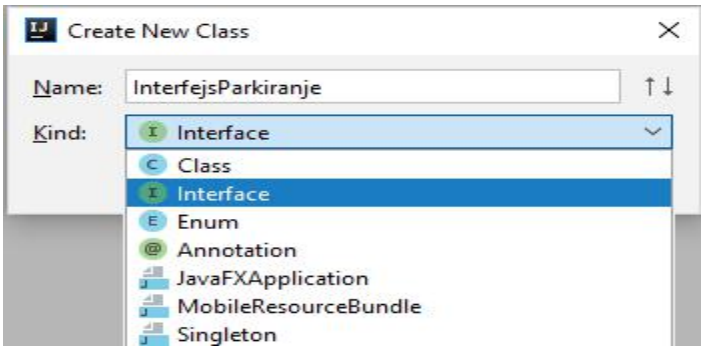
Apstraktne metode su one koje nemaju definisano telo već su samo deklarisane. Definicija tela mora se izvršiti u izvedenoj klasi. Ukoliko klasa sadrži bar jednu apstraktnu metodu onda i ona mora biti tako deklarisana. Deklaracija apstraktnih klasa i metoda vrši se pomoću rezervisane reči **abstact**. Apstraktna klasa služi isključivo za izvođenje novih jer nije podržano instanciranje objekata ovakve klase. U prethodnom primeru može se definisati klasa **Vozilo** kao apstraktna sa apstraktnom metodom **pokreni**().

Nov sadržaj **Vozilo.java** datoteke:

```
package vozila;

public abstract class Vozilo {
    protected String registracioni_broj;
    protected String model;
    protected int godiste;
    protected double zapremina_motora;

    public Vozilo(String registracioni_broj, String model, int godiste, double
zapremina_motora) {
        this.registracioni_broj = registracioni_broj;
        this.model = model;
        this.godiste = godiste;
        this.zapremina_motora = zapremina_motora;
    }

    protected abstract void pokreni();


    /*preklapanje toString meotode iz klase Object*/
    public String toString() {
        return "registracioni_broj=" + registracioni_broj +
                ", model=" + model +
                ", godiste=" + godiste +
                ", zapremina_motora=" + zapremina_motora ;
    }
}
```

Interfejsi predstavljaju poseban koncept koji podseća na apstraktnu klasu. Unutar interfejsa dozvoljena je upotreba samo statičkih konstanti koje imaju javnu vidljivost. Zbog ovih pravila moguće je izostaviti **static**, **final** i **public** . Unutar interfejsa nije dozvoljeno pisanje tela metoda osim ako metoda nije deklarisana kao statička, privatna ili podrazumevana. Podazumevane metode interfejsa kreiraju se pomoću rezervisane reči **default** koja se navodi u deklaraciji metode. Ukoliko se izostavi modifikator, metoda će imati javnu vidljivost. Klase ne proširuju interfejse već ih implementiraju. Implementiranje se vrši pomoću rezervisane reči **implements**. Moguće je implementirati više interfejsa u okviru jedne klase. Klasa može proširivati drugu i implementirati interfejse u isto vreme. Takođe i interfejsi mogu implementirati druge interfejse. U okviru klase moraju se implementirati sve nestatičke metode iz interfejsa i definisati njihova tela, osim ako se redeklarišu u apstraktne metode u

okviru apstraktne klase. Interfejsima se definiše zajedničko "ponašanje" klasa. Dodavanje novog interfejsa vrši se na sličan način kao i dodavanje nove klase. Klikom desnim tasterom miša na paket iz padajuće liste bira se opcija *New>Java Class*. Pojaviće se prozor za dijalog u kome je osim imena interfejsa potrebno naglasiti da se radi o interfejsu biranjem opcije **Interface** u sekciji *"**Kind**"* .



Slika 2.5- Dodavanje interfejsa

Sadržaj **InterfejsParkiranje.java** datoteke:

```
package interfejsi;

public interface InterfejsParkiranje {
    int brojRaspolozivihMesta=5;//staticko polje unutar iterfejsa
    //staticka metoda interfejsa
    static void ispisiBrojMesta(){
        System.out.println("Broj mesta:"+brojRaspolozivihMesta);
    }
    void parkiraj();
    void isparkiraj();
}
```

Sadržaj **InterfejsTovar.java** datoteke:

```
package interfejsi;
import tovar.Paket;
public interface InterfejsTovar {
    //privatna metoda interfejsa
    private void upakuj(Paket p){
        System.out.println("Paket upakovan!");
    }
    //podrazumevana metoda interfejsa
    default Paket pripremaPaketaZaUtovar(Paket p){
        upakuj(p);

        System.out.println("Paket spreman za utovar");
        return p;

    }
    boolean utovar(Paket p);
    void istovar();
}
```

Sadržaj **Paket.java** datoteke:

```
package tovar;
public class Paket {
    /*
    definicija klase Paket
```

```
     */
}
```

Sadržaj **Vozilo.java** datoteke:

```
package vozila;
public class Vozilo {
    /*
    definicija klase Vozilo
     */
}
```

Sadržaj **Automobil.java** datoteke:

```
package vozila;
import interfejsi.InterfejsParkiranje;
public class Automobil extends Vozilo implements InterfejsParkiranje {
    /*
    definicija  klase Automobil
     */
    @Override
    public void parkiraj() {
        System.out.println("Automobil parkiran!");
    }

    @Override
    public void isparkiraj() {
        System.out.println("Automobil isparkiran!");
    }
}
```

Sadržaj **Kamion.java** datoteke :

```
package vozila;

import interfejsi.InterfejsParkiranje;
import interfejsi.InterfejsTovar;
import tovar.Paket;

public class Kamion extends Vozilo implements InterfejsParkiranje, InterfejsTovar {
    /*
   definicija klase Kamion
    */
    @Override
    public void parkiraj() {
        System.out.println("Kamion parkiran");
    }

    @Override
    public void isparkiraj() {
        System.out.println("Kamion isparkiran!");
    }

    @Override
    public boolean utovar(Paket p) {
        System.out.println("Kamion izvrsio utovar!");
        return true;
    }

    @Override
    public void istovar() {
```

```
        System.out.println("Kamion izvrsio istovar!");
    }
}
```

Sadržaj **Program.java** datoteke:

```
package glavni;

import tovar.Paket;
import vozila.Automobil;
import vozila.Kamion;
public class Program {
    public static void main(String[] args) {
        Automobil a=new Automobil();
        Kamion k=new Kamion();
        Paket p=new Paket();
        a.parkiraj();
        k.isparkiraj();
        k.pripremaPaketaZaUtovar(p);
        k.utovar(p);
    }
}
```

Konzolni ispis:

```
Automobil parkiran!
Kamion isparkiran!
Paket upakovan!
Paket spreman za utovar
Kamion izvrsio utovar!
```

Polimorfizam se u slobodnom prevodu može prevesti kao "više oblika". U objektnom programiranju označava situacije gde objekti mogu imati više oblika u zavisnosti od stanja u kojem se nalaze. Na primer, referenca roditeljske klase može u sebi sadržati objekat dete klase. U generalizovanom slučaju referenca **Object** tipa može sadržati bilo koji objekat jer, kao što je ranije rečeno, to je natklasa svih klasa. Apstraktne klase i interfejsi mogu takođe imati svoje reference. Kako je nemoguće instacirati objekat interfejsa ili apstraktne klase tako u ovim referencama mogu biti lokalizovani objekti klasa koje implementiraju ili nasleđuju apstraktne klase odnosno interfejse. Poziv metoda vrši se na osnovu tipa reference, a ne na osnovu objekta koji se na toj lokaciji nalazi. U tom slučaju pristup svim dostupnim članovima objekta vršio bi se nakon eksplicitne konverzije reference u svoj tip. Konverzija se vrši u sledećem obliku *ImeKlase ime_reference = (ImeKlase)referenca_drugog_tipa.* Konverzija je moguća samo ukoliko se na datoj lokaciji stvarno nalazi objekat tog tipa. Ispitivanje kog je tipa objekat sa datom refencom vrši se pomoću operatora **instanceof**. Sa desne strane operatora nalazi se ime klase, dok je sa leve refenca. Ukoliko je objekat instanca date klase , izraz ce imati vrednost **true,** u suprotnom **false.**

Sadržaj **Program.java** datoteke:

```
package glavni;

import interfejsi.InterfejsTovar;
```

```
import vozila.Automobil;
import vozila.Kamion;
import vozila.Prikolica;
import vozila.Vozilo;

public class Program {
    public static void main(String[] args) {
        Vozilo v1=new Automobil("XX777","Golf 6",2007,1998.3,5,5);
        v1.parkiraj();//greska, metoda parkiraj() se ne nalazi u klasi parkiraj
        if(v1 instanceof Automobil){
            System.out.println("Objekat 'v1' je instanca klase Automobil!!");
            Automobil a1=(Automobil)v1;
        }

        InterfejsTovar it=new Kamion("KK578","Volvo FM3",2003,4508.9,new
Prikolica(),1500);
        it.istovar();
        it.isparkiraj(); //greska, metoda nije iz InterfejsaTovar
        if(it instanceof Kamion){
            System.out.println("Objekat 'it' je instanca klase Kamion!!");
            Kamion k1=(Kamion)it;
            k1.isparkiraj();
        }

        Vozilo vozniPark[]=new Vozilo[5];
        vozniPark[0]=new Automobil("XX888","Golf 7",2007,1578.8,5,4);
        vozniPark[1]=new Kamion("Kv578","FAP 1118",2008,3408.2,new Prikolica(),1500);
    }
}
```

Izuzeci (eng. Exception) predstavljaju mehanizam za obradu grešaka nastalih prilikom izvršavanja programa. Greške mogu biti: deljenje sa nulom, pristup nepostojećem članu niza, korišćenje **null** reference za poziv metode, ne pronalaženje datoteke za upis sa traženim imenom itd. Osnovni oblik obrade sastoji se od **try** i bar jednog **catch** bloka. U okviru **try** bloka nalazi se kod u kome može doći do greške, dok je u **catch** delu definisan scenario koji će se izvršiti u tom slučaju. **Catch** klauzula prima argument koji mora biti **Exception** ili nekog drugog tipa koji proširuje ovu klasu. Iz tog razloga možemo imati više **catch** klauzula, s tim da svaka "hvata" različit tip greške. U svakom od blokova definišu se različiti scenariji koji se izvršavaju u zavisnosti od tipa greške koja se dogodila u **try** bloku. Obično se klauzula sa generičkim tipom **Exception** postavlja poslednja u nizu. Razlog tome je što će se ona pozvati u svakom slučaju bez obzira na tip izuzetka (greške) koji se dogodio. U situaciji gde se ispisuje samo poruka o grešci, koristi se samo ova **catch** klauzula. Za definisanje scenarija koji se izvršava bez obzira da li se dogodila greška ili ne, koristi se **finally** klauzula. Ova klauzula nije obavezna, ne prima argumente i sadrži deo koda koji će se svakako izvršiti. Ukoliko je nađen **finally** blok, **catch** klauzula se može izostaviti. Moguće je ugnježdavanje **try/catch** obrade.

Definisanje korisničkih izuzetaka vrši se nasleđivanjem **Exception** ili neke druge klase koja je nasleđuje. Unutar novodobijene klase opciono se kreiraju konstruktori i metode za izvršenje. Podizanje korisničkog izuzetka moguće je samo programskim pozivom **throw** *korisnickiIzuzetak;* u okviru **try** bloka. Moguće je i definisati metode u kojima je moguća greška. Poziv ovih metoda mora biti u okviru **try** bloka ili druge metode koja je deklarisana kao metoda sa mogućim izuzetkom. Deklaracija ovih metoda vrši se navođenjem rezervisane reči **throws**, između argumenata metode i početka tela, nakon koje se navodi tip izuzetka.

Sadržaj **Primer.java** datoteke:

```
package glavni.Primer;

import izuzeci.MojIzuzetak;
import vozila.Automobil;

public class Primer {
    public static void main(String[] args) {
        try {
            if (provera()) { //pre drugog pokretanja promeniti povratnu vrednost metode
provera
                throw new MojIzuzetak("Greska pri proveri!!");
            }
            double a = 5 / 0; // pre treceg pokretanja zakomentarisati ovu liniju
            Automobil a1 = null;
            a1.isparkiraj();
        } catch (MojIzuzetak mi) {
            System.out.println("Dogodila se korisnicka greska:" + mi.metodaIzuzetka());

        } catch (ArithmeticException e) {
            System.out.println("Dogodila se aritmeticka greska:" + e.getMessage());
        } catch (Exception e) {
            System.out.println("Dogodila se greska!!");
            e.printStackTrace();
        } finally {
            System.out.println("Poruka koja se svakako prikazuje!!");
        }

        try {
            primerMetode();

        } catch (MojIzuzetak mojIzuzetak) {
            System.out.println( mojIzuzetak.getMessage());

        }

    }
    private static boolean provera() {
        /*
        telo metode provera
         */
        return true;
    }
    private static void primerMetode() throws MojIzuzetak{
        /*
        telo metode primerMetode
         */


        throw new MojIzuzetak("Greska u metodi 'primerMetode'");
    }
}
```

Try-with-resource je poseban oblik **try** izraza koji koristi resurse unutar **try** bloka koje je nakon upotrebe potrebno zatvoriti. Ovaj izraz olakšava proces iterativne obrade sa privremenim objektima. Objekti se mogu koristiti u ovim izrazima samo ako njihove klase implementiraju interfejs **java.lang.AutoCloseable** ili **java.io.Closeable.** Nakon upotrebe resurs će automatski biti zatvoren čim se završi blok sa ovim izrazom.

Primer try-with-resource izraza:

```
public void procitajSadržajDatoteke(String putanja) throws IOException {
    String linijaTeksta = "";
    try (BufferedReader br = new BufferedReader(new FileReader(putanja))) {
        linijaTeksta = br.readLine();
        while (linijaTeksta != null) {
            System.out.println(linijaTeksta);
        }

    }//moguce je izostaviti cath i finali blok pod uslovom da se u potpisu metode
navede da metoda podize taj tip izuzetka

    //nakon zavrsetka obrade, objekat br se automatski zatvara
}
```

Čitanje podataka sa podrzumevanog ulaza može se izvršiti pomoću **Scanner** klase. Za početak potrebno je kreirati objekat ove klase sa prosleđenim objektom podrazumevanog ulaznog strima (**System.in**). Nakon toga moguće je pozivati metode za čitanje podataka unetih preko tastature. U zavisnosti od tipa ulaznih podataka pozivaju se neke od sledećih metoda za čitanje unosa: **nextInt(), nextLong(), nextDouble(), nextFloat(), nextBoolean(), nextBigDecimal(), nextLine()** itd. Poziv ovih metoda stopira dalje izvršenje programa sve dok se ne završi unos. Čitanje se podrazumevano izvršava nakon unosa karaktera za novi red. Nakon završetka unosa podataka potrebno je zatvoriti objekat ove klase pozivom metode **close().**

Sadržaj **Program.java** datoteke:

```
package glavni;

import java.util.Scanner;

public class Program {

    public static void main(String[] args) {
        Scanner ulaz=new Scanner(System.in);
        System.out.print("Unesite tekstualni sadržaj:");
        String ulazString=ulaz.nextLine();
        System.out.println("Unos:"+ulazString);
        System.out.print("Unesite celobrojnu vrednost:");
        int ulazCelobrojni=ulaz.nextInt();
        System.out.println("Unos:"+ulazCelobrojni);
        System.out.print("Unesite decimalnu vrednostu:");
        double ulazDecimalni=ulaz.nextDouble();
        System.out.println("Unos:"+ulazDecimalni);

        ulaz.close();

    }
}
```

Ukoliko u toku izvršavanja programa dođe do potrebe za korišćenjem dinamičkih struktura ne bi bilo loše razmotriti mogućnost korišćenja **ArrayList** klase. Ova klasa je deo **collection** "radnog okvira" (eng. framework, u daljem tekstu frejmvork) koji se nalazi u **java.base** modulu u paketu **java.util**. Klasa omogućava kreiranje dinamičkih nizova. Ovakva struktura je sporija od niza, ali efikasnija kada je potrebno izvršiti manipulaciju sa nepoznatim

brojem objekata. Pri kreiranju reference potrebno je navesti tip elemenata koji će se smestiti u listu. Tip se navodi između znakova <> (operator romboid) i ne može biti prost. Kod poziva konstruktora klase **ArrayList** može se izovstaviti tip.

Sadržaj **PrimerLista.java** datoteke**:**

```
package primer;

import java.util.ArrayList;

public class PrimerLista {
    public static void main(String[] args) {
        ArrayList<String> listaStringova=new ArrayList<>(); //lista stringova
        listaStringova.add("Prvi element liste");
        listaStringova.add("Drugi element liste");
        listaStringova.add("Treci element liste");
        System.out.println(listaStringova.get(0));
        listaStringova.remove(2);
        System.out.println("Prikaz svih elemenata liste:");
        //prikaz se vrsi primenom foreach petlje
        for (String s:listaStringova) {
            System.out.println(s);
        }
    }
}
```

Konzolni ispis:

```
Prvi element liste
Prikaz svih elemenata liste:
Prvi element liste
Drugi element liste
```

Zadatak za samostalan rad:

Kreirati konzolnu aplikaciju koja sadrži klasu **Telefon** u paketu **komunikacija/telefon**. Klasa je opisana atributima: proizvodjac, model i broj. Nije moguće instancirati objekat ove klase. Klasa poseduje konstruktor koji prihvata potrebne parametre za inicijalizaciju vrednosti svih atributa. Klasa poseduje statičku metodu koja vraća ukupan broj telefona.

Unutar istog paketa nalaze se i klase **MobilniTelefon** i **FiksniTelefon** koje proširuju klasu **Telefon**. Klasa **MobilniTelefon** osim osnovnih poseduje i atribute mreža i prečnikEkrana. Ispis objekta ove klase vrši se u formatu *M(proizvodjac, model, broj, Operater:mreza, precnik ekrana:precnikEkrana).* Klasa **FiksniTelefon** osim osnovnih poseduje i atribut dužinaKabla. Ispis objekta ove klase vrši se u formatu *F(proizvodjac, model, broj, duzina kabla:precnikEkrana)*. Konstruktori ovih klasa prihvataju potrebne atribute za inicijalizaciju vrednosti svih atributa.

Interfejs **KomunikacijaTelefoni** poseduje metode kojim se opisuje "ponašanje" mobilnih i fiksnih telefona. Interfejs poseduje metode **pozovi()**, koja prima broj koji treba pozvati kao argument, i metodu **javiSe()** koja ne prima argumente. Metode nemaju složeno telo već ispisuju poruke da je fiksni ili mobilni telefon pozvao odredjeni broj ili da je primio poziv. Klasa **MobilniTelefon** poseduje i metode **posaljiSMS**() i **posaljiMMS**() koje ispisuju poruku da je dati telefon poslao SMS/MMS.

Kreirati paket glavni i u njemu klasu **Program** koja sadrži **main()** metodu u kojoj treba testirati funkcionalnost prethodnih klasa. Testiranje se vrši tako što se kreira korisnički interfejs za unos telefona. Pre unosa korisniku se nude 3 opcije: za unos mobilnog telefona, za unos fiksnog telefona i kraj unosa.

Laboratorijska vežba 4: Rad sa tekstualnim datotekama, serijalizacija, moduli

Rad sa datotekama je gotovo neizbežan posao prilikom razvijanja aplikacije. Iako se danas za skladištenje podataka koriste baze podataka (MySQL, SQLite, Mongo itd.) datoteke se i dalje koriste za čuvanje lokalnih informacija. Te informacije mogu se odnositi na konfiguraciona stanja, čuvanje informacija o pristupu, čuvanje serijalizovanih objekata itd. U ovoj vežbi biće objašnjen postupak upisa i čitanja podataka iz tekstualnih datoteka.

Pre realizacije samog postupka upisa ili čitanja potrebno je kreirati datoteku. U okviru standardnog Javinog programskog alata (**JDK**) nalazi se klasa **File** koja nudi mogućnost upravljanja procesom manipulacije nad datotekama i direktorijumima. Pre korišćenja potrebno je importovati klasu iz paketa **java.io.** U osnovi konstruktor ove klase prima **String** koji se odnosi na putanju datoteke ili direktorijuma. Ukoliko se navede samo ime, lokacija će biti u okviru samog projekta ili direktorijuma iz kog se program pokrenuo, ukoliko se radi o izvršnoj datoteci.

Često korišćene metode klase **File**:

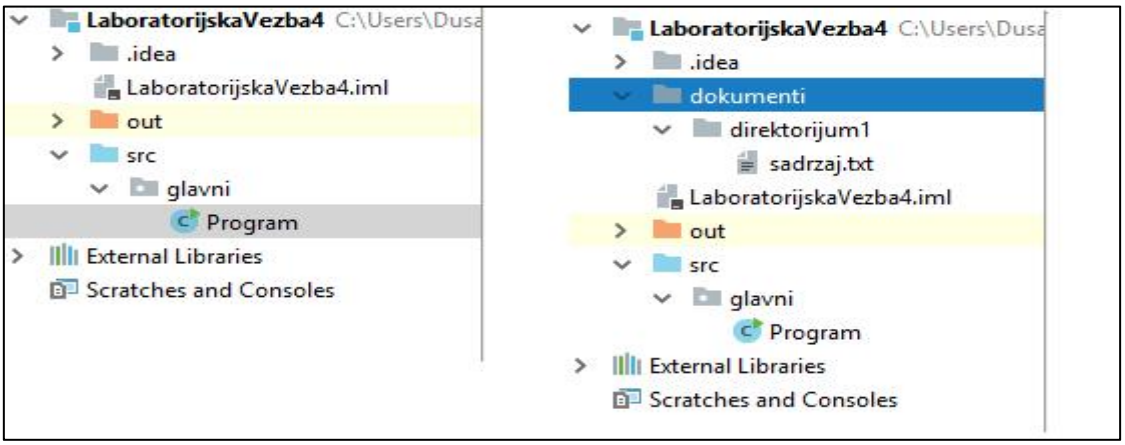
- `exists()` – vraća vrednost **true** ili **false** u zavisnosti da li tražena datoteka ili direktorijum postoji
- `isDirectory()` – vraća vrednost **true** ili **false** u zavisnosti da li se uneta putanja odnosi na direktorijum.
- `isFile()` - vraća vrednost **true** ili **false** u zavisnosti da li se uneta putanja odnosi na datoteku.
- `createNewFile()`- vraća vrednost **true** ili **false** u zavisnosti da li je datoteka kreirana ili nije. Kreiranje će se izvršiti samo ako ne postoji datoteka sa zadatim imenom na toj lokaciji.
- `mkdir()`– vraća vrednost **true** ili **false** u zavisnosti da li je direktorijum kreiran ili nije.
- `mkdirs()`–vraća vrednost **true** ili **false** u zavisnosti da li je direktorijum kreiran ili nije, s tim da će kreirati i roditeljske direktorijume iz putanje ukoliko ne postoje.
- `delete()`– vraća vrednost **true** ili **false** u zavisnosti da li je obrisan direktorijum ili datoteka sa unetom putanjom.
- `list()`– vraća listu direktorijuma i datoteka koje se nalaze na traženoj lokaciji pod pretpostavkom da je pravilno uneta putanja do postojećeg direktorijuma. Rezultat je predstavljen kao niz stringova (**String[]**).
- `listFiles()`– vraća listu datoteka koje se nalaze na traženoj lokaciji pod pretpostavkom da je pravilno uneta putanja do postojećeg direktorijuma. Povratni tip je niz **File** objekata (**File[]**).

Sadržaj **Program.java** datoteke

```
package glavni;

import java.io.File;
import java.io.IOException;

public class Program {
    public static void main(String[] args) {
        //kreiranje stabla direktorijuma
        String putanjaDoDirektorijuma = "dokumenti\\direktorijum1";
        File fDirektorijum = new File(putanjaDoDirektorijuma);
        //provera da li postoji trazena struktura i kreiranje ukoliko ne postoji
        if (!fDirektorijum.exists()) {
            fDirektorijum.mkdirs();
        }
        //kreiranje datoteke unutar direktorijuma "direkrtorijum1"
        File fDatoteka = new File(putanjaDoDirektorijuma + "\\sadrzaj.txt");
        //provera da li postoji datoteka i kreiranje ukoliko ne postoji
        if (!fDatoteka.exists()) {
            try {
                fDatoteka.createNewFile();
            } catch (IOException e) {
                System.out.println("Desila se greska prilikom kreiranja datoteke!");
                e.printStackTrace();
            }
        }
        //prikaz datoteka iz dokumenti\direktorijum1 direktorijuma
        for (File f : fDirektorijum.listFiles()){
            System.out.println(f);
        }
    }
}
```

Konzolni ispis:

```
dokumenti\direktorijum1\sadrzaj.txt
```



Slika 4.1 – Struktura projekta pre i posle pokretanja

Upis tekstualnog sadržaja u datoteke se može izvršiti uz pomoć **FileWriter** klase. Prvi argument konstruktora ove klase može biti putanja do datoteke(**String)** ili objekat **File** klase. Drugi argument je opcioni i odnosi se na način upisa tj. da li se postojeći sadržaj briše ili ne (**append** opcija). Ukoliko se dodaje nov sadržaj na postojeći prosleđuje se vrednost **true.** Konstruktor podiže (baca) **IOException** tako da se poziv mora odviti u okviru **try/catch**

sekcije ili da se u potpisu metode, koja ga poziva, doda izuzetak. Upis se zatim vrši pozivom metode **write()** kojoj se prosleđuje sadržaj za upis. Dodavanje novog reda može se izvršiti pomoću kodne sekvence **'\n'.** Ovo nije najbolje rešenje ukoliko aplikacija treba da se izvršava na različitim operativnim sistemima. U ovakvim situacijama preporuka je da se koristi **System.getProperty("line.separator")** metoda koja od samog operativnog sistema zahteva karakter za novi red. Naknadni upis na postojeći sadržaj može se izvršiti pozivom metode **append()** koja vrši dopisivanje nezavisno od opcije koja je prosleđena konstruktoru. Dopisivanje će se izvršiti samo pod uslovom da je prethodno izvršena **write()** metoda, u suprotnom vrši običan upis. Prethodne metode takođe podižu **IOException**. Zatvaranje strima (eng. stream) vrši se pozivom **close()** metode.

Klasa **FileReader** služi nam za čitanje podataka iz tekstualne datoteke. Konstruktor klase prima jedan argument koji može biti putanja datoteke ili **File** objekat. Konstruktor podiže **IOException** tako da je obrada poziva ista kao i kod **FileWriter** klase. Metoda **read()** u osnovnom obliku bez argumenata uzima **ASCII** vrednost karaktera i pozionira se na sledeći. Ukoliko nije moguće čitanje ili nema više sadržaja (kraj datoteke) vraća vrednost **-1**. U kombinaciji sa odgovarajućim parametrima metoda može da pročita sekvencu karaktera. Ukoliko se prosledi niz karaktera (**char[n]**) izvršiće se čitanje **n** karaktera u odnosu na trenutnu poziciju u dokumentu. Eksplicitno pozicioniranje vrši se pozivom **skip()** metode kojoj se prosleđuje celobrojna vrednost pomeraja. Prethodne metode takođe podižu **IOException**. Zatvaranje strima vrši se pozivom **close()** metode.

Sadržaj **ProgramZaUpisICitanje.java** datoteke:

```
package glavni;

import java.io.*;

public class ProgramZaUpisICitanje {
    public static void main(String[] args) {
        String putanjaDoDatoteke = "dokumenti\\direktorijum1\\sadrzaj.txt";
        //otvaranje izlaznog strima i upis tekstualnog sadrzaja
        try {
            FileWriter fw = new FileWriter(putanjaDoDatoteke, false);//otvaranje strima
za upis sa brisanjem postojeceg sadrzaja. Isprobati i drugi scenario postavljanjem
append opcije na true
            fw.write("Prva linija teksta" + System.getProperty("line.separator") +
"Druga linija teksta");
            fw.append(System.getProperty("line.separator") + "Dodatni sadrzaj");
            fw.close(); //zatvaranje izlaznog strima
        } catch (IOException e) {
            System.out.println("Dogodila se greska prilikom otvaranja izlaznog strima
ka datoteci \"sadrzaj.txt\"");
            e.printStackTrace();
        }
        //otvaranje ulaznog strima i citanje tekstualnog sadrzaja

        try {
            FileReader fr = new FileReader(putanjaDoDatoteke);
            String sadrzajDatoteke = "";
            int c = fr.read();//uzimanje ASCII vrednost karaktera i pozicioniranje na
sledeci
            while (c != -1) {
                sadrzajDatoteke += (char) c;
```

```
                c = fr.read();
            }
            System.out.println(sadrzajDatoteke);
            fr.close();//zatvaranje ulaznog strima
            fr = new FileReader(putanjaDoDatoteke);
            char baferKaraktera[] = new char[10];
            fr.skip(5);//pomeranje za 5 mesta
            fr.read(baferKaraktera);//popunjavanje bafera
            System.out.println("Sadrzaj bafera:");
            System.out.println(baferKaraktera);
            fr.close();
        } catch (FileNotFoundException e) {
            System.out.println("Dogodila se greska prilikom otvaranja ulaznog strima ka
datoteci \"sadrzaj.txt\"");
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Konzolni ispis:

```
Prva linija teksta
Druga linija teksta
Dodatni sadrzaj
Sadrzaj bafera:
linija tek
```

Sadržaj **sadrzaj.txt** datoteke nakon izvršenja programa:

```
Prva linija teksta
Druga linija teksta
Dodatni sadrzaj
```

U praksi tekstualni sadržaj ne mora biti uvek zapisan u ASCII kodnom sistemu. Na primer za pisanje ćiriličnih ili latiničnih slova iz neengleskog alfabeta (Ć, Č, Š, Ž itd.) koristi se **UTF-8**. U takvim situcijama preporuka je da se koriste sledeće klase koje olakšavaju rukovanje sa ulazno/izlaznim strimovima.

Čitanje

BufferedReader klasa čita teksualni sadržaj iz karakter - ulaznog strima. Baferovanjem (eng. Buffering) karaktera postiže se efikasnije čitanje karaktera, nizova i linija teksta. Čitanje linije teksta može se izvršiti pomoću metode **readLine().** Nakon uspešnog čitanja vrši se pozicioniranje na sledeću. Kada nema više sadržaja (linija) za čitanje metoda vraća vrednost **null.** Metoda podiže **IOException.** Nakon korišćenja potrebno je zatvoriti strim pozivom metode **close(). InputStreamReader** je klasa koja premošćava prenos iz ulaznog strima bajtova u strim karaktera. U konstruktoru ove klase može se eksplicitno navesti koji će se kodni sistem koristiti prilikom kodovanja (eng. Encoding, podrazumevano **ASCII**). Konstruktor ove klase podiže **UnsupportedEncodingException** izuzetak. **FileInputStream** otvara ulazni (bajtovski) strim podataka. Konstruktor ove klase podiže **FileNotFoundException** izuzetak.

Upis

BufferedWriter klasa rukuje sa izlaznim strimom baferovanjem karaktera čime se postiže efikasniji upis pojedinačnih karaktera, nizova ili linije teksta. Upis se vrši pomoću metode **write()** (podiže **IOException**). Poseduje i metodu **newLine()** koja zahteva od operativnog sistema karakter za novi red i upisuje ga u datoteku. Nakon korišćenja potrebno je zatvoriti strim - poziv metode **close(). OutputStreamWriter** premošćava prenos iz izlaznog strima bajtova u strim karaktera. U konstruktoru ove klase može se eksplicitno navesti koji će se kodni sistem koristiti prilikom kodovanja (eng. Encoding - podrazumevano ASCII). Konstruktor ove klase podiže **UnsupportedEncodingException** izuzetak. **FileOutputStream** kreira izlazni (bajtovski) strim podataka. Prvi argument konstruktora ove klase može biti putanja datoteke (**String)** ili objekat **File** klase. Drugi argument je opcioni i odnosi se na način upisa tj. da li se postojeći sadržaj briše ili ne (**append** opcija). Konstruktor ove klase podiže **FileNotFoundException** izuzetak.

Sadržaj **ProgramZaUpisICitanje.java** datoteke:

```
package glavni;

import java.io.*;
import java.util.ArrayList;

public class ProgramZaUpisICitanje {
    public static void main(String[] args) {
        String putanjaDoDatoteke = "dokumenti\\direktorijum1\\sadrzaj.txt";

        //otvaranje izlaznog strima i upis tekstualnog sadrzaja
        try {
            BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(new
FileOutputStream(putanjaDoDatoteke, false), "UTF-8"));
            bw.write("Прва линија текста");
            bw.newLine();
            bw.write("Друга линија текста");
            bw.close();
        } catch (UnsupportedEncodingException e) {
            System.out.println("Dogodila se greska prilikom enkodovanja izlaznog
strima!");
            e.printStackTrace();
        } catch (FileNotFoundException e) {
            System.out.println("Dogodila se greska prilikom otvaranja izlaznog
strima!");
            e.printStackTrace();
        } catch (IOException e) {
            System.out.println("Dogodila se greska priliko upisa!");
            e.printStackTrace();
        }


        //otvaranje ulaznog strima i citanje tekstualnog sadrzaja
        try {
            BufferedReader br = new BufferedReader(new InputStreamReader(new
FileInputStream(putanjaDoDatoteke), "UTF-8"));
            ArrayList<String> sadrzaj = new ArrayList<String>();//kreiranje liste za
skladistenje linija teksta
            String linija = br.readLine();//citanje linije i pozicioniranje na sledecu
            while (linija != null) {
                sadrzaj.add(linija);
                linija = br.readLine();
```

```
            }
            System.out.println("Sadrzaj datoteke:");
            for (String l : sadrzaj) {
                System.out.println(l);
            }
        } catch (FileNotFoundException e) {
            System.out.println("Dogodila se greska prilikom otvaranja ulaznog strima!");
            e.printStackTrace();
        } catch (UnsupportedEncodingException e) {
            System.out.println("Dogodila se greska prilikom enkodovanja ulaznog
strima!");
            e.printStackTrace();
        } catch (IOException e) {
            System.out.println("Dogodila se greska prilikom citanja sadrzaja!");
            e.printStackTrace();
        }
    }
}
```

Konzolni ispis:

```
Sadrzaj datoteke:
Прва линија текста
Друга линија текста
```

Sadržaj **sadrzaj.txt** datoteke nakon izvršenja programa:

```
Прва линија текста
Друга линија текста
```

Serijalizacija je proces prevođenja objekta u niz bajtova. Nakon toga niz bajtova može se preneti putem mreže ili sačuvati u datoteci. Deserijalizacija je obrnuti postupak iz kog se dobija originalno stanje objekta. Da bi se objekat serijalizovao njegova klasa mora da implementira interfejs **Serializable.** Strimovi, soketi (eng. socket) i niti (eng. thread) ne mogu se serijalizovati. Ukoliko u okviru klase postoje ovakva polja potrebno je navesti rezervisanu reč **transient** u njihovoj deklaraciji. Isti je postupak i za sva ostala polja koja se ne serijalizuju. Postupkom deserijalizice ova polja uzimaju podrazumevane ili **null** vrednost za proste odnosno složene tipove.

Postupkom serijalizacije i deserijalizacije upravljaju **ObjectOutputStream** i **ObjectInputStream** klase**.** Klasama se prosleđuju izlazni odnosno ulazni (bajtovski) strimovi (**FileOutputStream, FileInputStream).** Konstruktori ovih klasa podižu **IOException** izuzetak. Pozivom metode **writeObject()** (preporuka: koristite **writeUnshared()**) objekta klase **ObjectOutputStream** upisuje se serijalizovani objekat koji se prosleđuje kao argument. Metoda **readObject()** objekta klase **ObjectInputStream** vraća deserijalizovani objekat koji je upisan u datoteci. Metoda podiže **ClassNotFoundException** izuzetak. Povratni tip metode je **Object** pa je potrebno izvršiti ekscplicitnu konverziju u specijalizovani tip objekta.

Sadržaj **Osoba.java** datoteke:

```
package serijalizacija;

import java.io.Serializable;

public class Osoba implements Serializable {
```

```
    private String ime;
    private int godiste;
    transient private double plata;

    public Osoba(String ime, int godiste, double plata) {
        this.ime = ime;
        this.godiste = godiste;
        this.plata = plata;
    }

    @Override
    public String toString() {
        return "Osoba{" +
                "ime='" + ime + '\'' +
                ", godiste=" + godiste +
                ", plata=" + plata +
                '}';
    }
}
```

Sadržaj **PrimerSerijalazicije.java** datoteke:

```
package serijalizacija;

import java.io.*;

public class PrimerSerijalizacije {
    public static void main(String[] args) {
        String putanjaDoDatoteke = "dokumenti\\direktorijum1\\dat.bin";
        Osoba Pera=new Osoba("Pera",1990,5000.0);
        System.out.println(Pera);
        //otvaranje izlaznog strima i upis serijalizovanog objekta
        try {
            ObjectOutputStream oos=new ObjectOutputStream(new
FileOutputStream(putanjaDoDatoteke));
            oos.writeObject(Pera);
        } catch (IOException e) {
            e.printStackTrace();
        }
        //otvaranje ulaznog strima, citanje i deserijalizacija objekta
        try {
            ObjectInputStream ois=new ObjectInputStream(new
FileInputStream(putanjaDoDatoteke));
            Osoba PeraP=(Osoba)ois.readObject();
            System.out.println("Ispis objekta nakon deserijalizacije:");
            System.out.println(PeraP);
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Konzolni ispis:

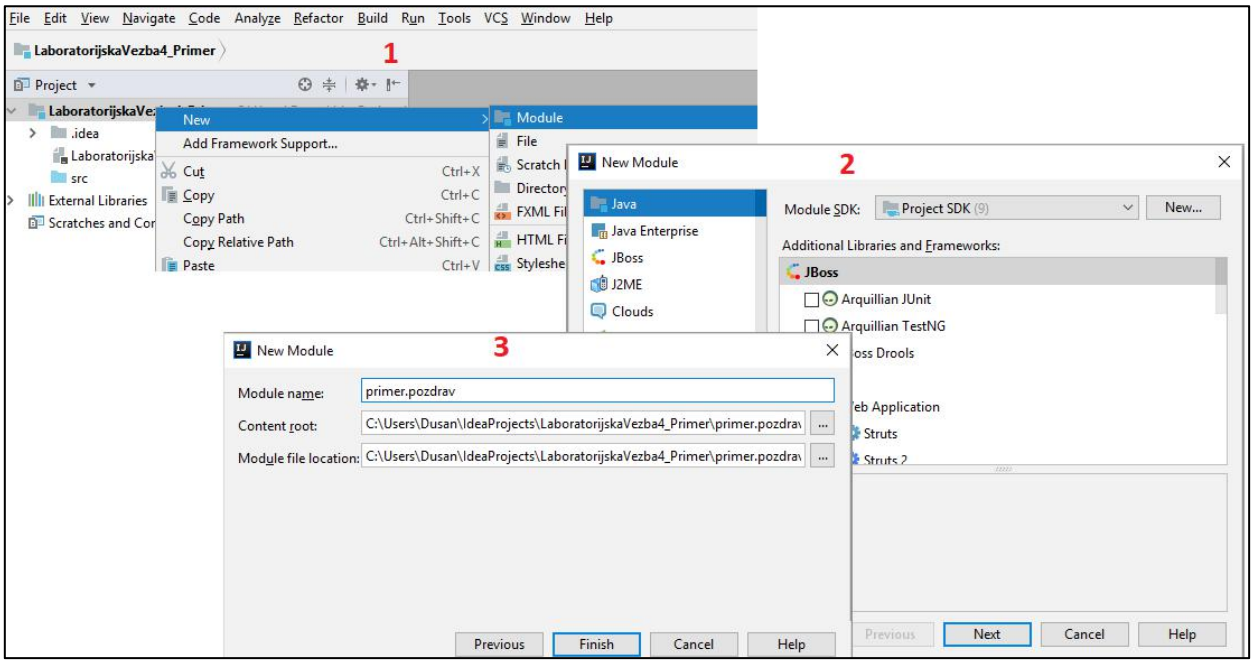
```
Osoba{ime='Pera', godiste=1990, plata=5000.0}
Ispis objekta nakon deserijalizacije:
Osoba{ime='Pera', godiste=1990, plata=0.0}
```

Modularni pristup razvijanja aplikacija omogućen je u Javi od verzije 9 u okviru **Jigsaw** projekta koji je zatim preimenovan u **Java Platform Module System (JPMS).** Modularni pristup može se posmatrati kao posebna vrsta dizajna i konstrukcija računarskog softvera. Ovakav dizajn sastoji se od skupa modula koji zajedno čine celinu. Modul je posebna funkcionalna celina koja može da deluje nezavisno od ostatka sistema. On je kolekcija podataka i kodova tj. paketa, klasa, interfejsa, kodova, podataka i resursa. Ovakav pristup značajno se odrazio na perfomanse samih aplikacija. Aplikacije su postale manje sa mogućnošću skaliranja za manje uređaje, smanjena je ukupna složenost sistema, kod je postao lakši za održavanje, dobijene su nove mogućnosti za enkapsulaciju itd. Modul se može posmatrati i kao kontejner paketa.

Ime modula mora biti jedinstveno na nivou celog projekta. Pravila koja važe za imenovanje paketa, važe i kod imenovanja modula. Preporuka je da se u imenu ne koristi donja crta ('_') zbog pretpostavki da će ovaj karakter imati posebnu namenu u budućim verzijama. U korenskom direktorijumu svakog modula mora postojati deklaraciona datoteka. U okviru ove datoteke specificirana je zavisnost modula od drugih (**requires)** kao i kojim sve paketima iz modula mogu drugi da pristupe (**exports)**. U okviru **java.base** modula nalaze se osnovni alati. Sam modul ne zahteva druge, već služi za izvoz velikog broja paketa kao što su **java.io, java.lang, java.math, java.net, java.time, java.util** i mnogi drugi. Ukoliko se u okviru deklaracije modula ne navede zahtevanje ovog modula, on će automatski biti dodat prilikom kompajliranja. U deklaraciji nije dozvoljeno navesti cikličnu zavisnost. Na primer ako modul **A** zahteva modul **B,** onda je zabranjeno da modul **B** zahteva modul **A.**

U sledećem delu biće objašnjen postupak kreiranja i povezivanja modula unutar projekta korišćenjem **IntellJ** razvojnog okruženja. U primeru će biti korišćen "Zdravo Svete" (Hello World) primer. Primer se sastoji od dva modula: **primer.pozdrav** i **test.pozdrav**.

Kreriranje projekta vrši se na isti način kao i primeru na prvoj vežbi, s tim da se posebna pažnja treba obratiti na projektni **SDK**. Uzevši u obzir da se kreira modularna aplikacija, verzija **JDK**-a mora biti minimun 9. Po kreiranju projekta sa zadatim imenom, u ovom slučaju "LaboratorijskaVezba4_Primer", dobija se uobičajena struktura projekta gde treba dodati module. Dodavanje modula vrši se klikom desnim tasterom miša i odabira opcija **New>Module.** Nakon toga pokreće se proces sličan onom kod kreiranja novog projekta. Treba ispratiti date korake i dati ime modulu. Na sledećoj slici prikazan je postupak dodavanja modula "primer.pozdrav" (crvenom bojom numerisani su koraci).



Slika 4.2- Dodavanje novog modula

Isti postupak je potrebno ponoviti i za "**test.pozdrav**" modul. Po kreiranju, modul dobija sličnu strukturu kao i prazan projekat. Unutar prvog modula treba dodati klasu **PozdravSvete** koja se nalazi u **primer/pozdrav** paketu. Klasa sadrži samo metodu **pozdrav()** koja vraća string " Zdravo svete". U drugom modulu treba dodati klasu **TestPozdrav** unutar paketa **testiranje/pozdrav.** Klasa sadrži **main()** metodu u kojoj se instancira objekat klase **PozdravSvete** i poziva njena metoda. Povratna vrednost metode se prikazuje u konzoli. Ovde se može uočiti da drugi modul zahteva korišćenje prvog tj. jednog njegovog dela. Zavisnost mora biti navedena u deklaracionim datotekama (**module-info.java)** oba modula. Dodavanje deklaracionih datoteka može se izvršiti klikom desnim tasterom miša na **src** dirtektorijum unutar modula i odabrati opcija **New>module-info.java**.

Sadržaj deklaracione datoteke **primer.pozdrav** modula:

```
module primer.pozdrav {
    exports primer.pozdrav;
}
```

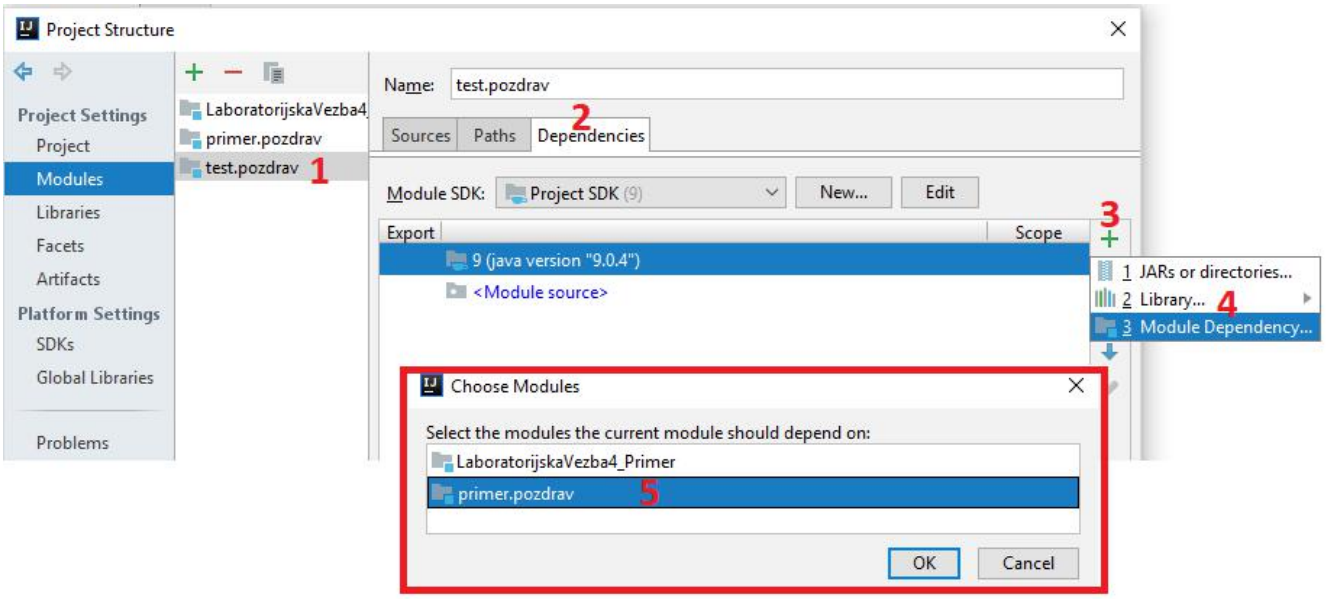
Unutar ove deklaracione datoteke napisana je dozvola za korišćenje klasa iz paketa **primer/pozdrav** u "spoljnom svetu" tj. u svim modulima koji zahtevaju ovaj modul.

Sadržaj deklaracione datoteke **test.pozdrav** modula:

```
module test.pozdrav {
    requires primer.pozdrav;
}
```

Kao što se može uočiti napisana je zavisnost **test.pozdrav** modula od modula **primer.pozdrav**. Opciono se može dodati zavisnost oba modula od **java.base,** ali kao što je već rečeno ovo je podrazumevana zavisnost svakog novokreiranog modula, tako da je nije potrebno navesti. Navođenje zavisnosti unutar deklaracione datoteke nije dovoljno da bi se moduli povezali na nivou projekta, tako da je potrebno otvoriti podešavanja modula projekta klikom desnim tasterom miša na ime projekta, a zatim odabrati opciju **Open Module Settings.** Odabirom ove opcije otvara se nov prozor gde je potrebno odabrati modul kojem se dodaje zavisnost, a zatim odabrati karticu **Dependencies** iz desnog segmenta. U novootvorenom

segmentu prikazuje se zavisnost odabranog modula gde je moguće i dodati novu klikom na ikonicu sa simbolom "+" . Nakon toga treba specificirati da se radi o zavisnosti od modula, a zatim odabrati modul u ovom slučaju **primer.pozdrav.**



Slika 4.3- Dodavanje zavisnosti modula

Nakon izvršenog ovog koraka, modul **test.pozdrav** može da koristi sve što je dozvoljeno za "izvoz" (**exports**) iz modula **primer.pozdrav**.

Sadržaj **PozdravSvete.java** datoteke:

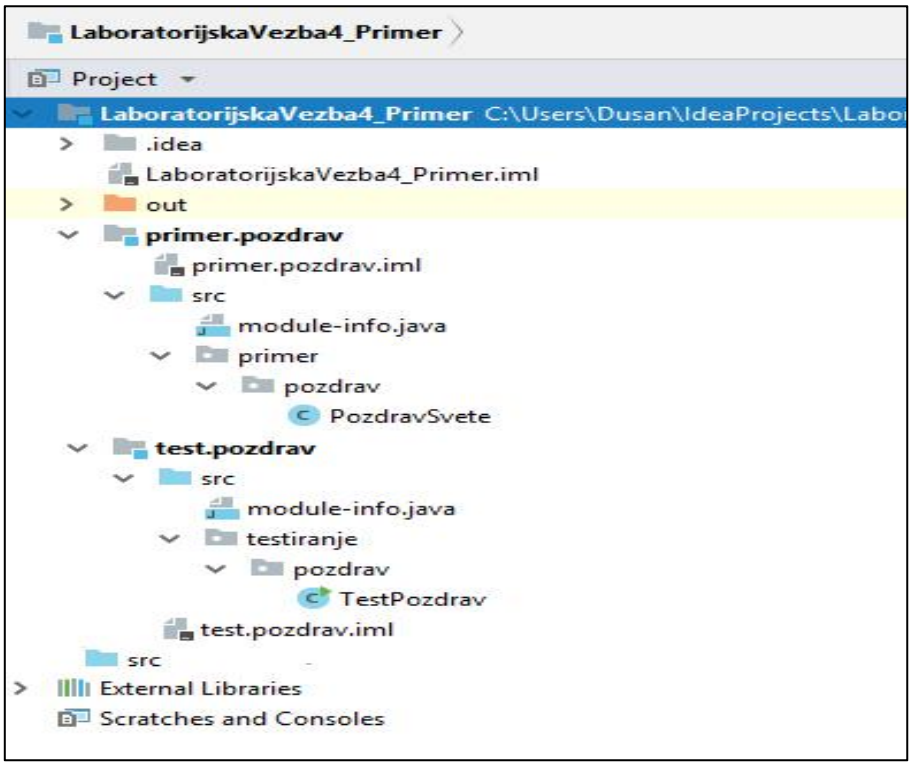
```
package primer.pozdrav;
public class PozdravSvete {
    public String pozdrav(){

        return "Zdravo svete";
    }
}
```

Sadržaj **TestPozdrav.java** datoteke:

```
package testiranje.pozdrav;
import primer.pozdrav.PozdravSvete;

public class TestPozdrav {
    public static void main(String[] args) {
        PozdravSvete ps=new PozdravSvete();
        System.out.println(ps.pozdrav());
    }
}
```

Konzolni ispis:

```
Zdravo svete
```



Slika 4.4 – Struktura projekta

Zadatak za samostalni rad:

Kreirati konzolnu aplikaciju za uređivanje zapisnika. Po pokretanju korisniku se prikazuje sledeći meni u kome može da odabere jednu od opcija unosom odgovarajućeg broja preko tastature. Meni sadrži sledeće opcije:

1. Unos nove arhive
2. Unos novog zapisnika
3. Unos sadrzaja u zapisnik
4. Dodaj sadrzaj u zapisnik
5. Izlistaj sve zapisnike
6. Prikazi zapisnik
7. Obrisi zapisnik
8. Izlaz

Odabirom **prve** opcije prikazuje se nov dijalog u kome se od korisnika traži da unese ime nove arhive. Ukoliko je pravilno uneto ime kreira se direktorijum sa datim imenom u direktorijum "**arhive"** u okviru projekta. Korisnik po unosu imena dobija poruku da li je arhiva kreirana ili nije nakon čega mu se pojavljuje početni meni.

Odabirom **druge** opcije prikazuje se nov dijalog u kome se od korisnika traži da unese ime arhive, a zatim ime zapisnika. Ukoliko je unos pravilan kreiraće se datoteka sa datim imenom u arhivi koja je navedena. Korisnik po unosu imena dobija poruku da li je arhiva kreirana ili nije nakon čega mu se pojavljuje početni meni.

Odabirom **treće** opcije od korisnika se traži da unese ime zapisnika. Ukoliko zapisnik postoji pokreće se proces za unos sadržaja. U suprotnom prikazuje se poruka da zapisnik nije

pronađen, a zatim se prikazuje i početni meni. Sadržaj se može unositi latiničnim i ćiriličnim slovima. Po završetku unosa, korisnik se obaveštava da je sadržaj uspešno sačuvan, a zatim mu se prikazuje početni meni.

Odabirom **četvrte** opcije od korisnika se traži da unese ime zapisnika. Ukoliko zapisnik postoji pokreće se proces za unos dodatnog sadržaja. U suprotnom se prikazuje poruka da zapisnik nije pronađen, a zatim se prikazuje i početni meni. Po završetku unosa dodatnog sadržaja, korisnik se obaveštava da je uspešno sačuvan, a zatim mu se prikazuje početni meni.

Odabirom **pete** opcije korisniku se prikazuje spisak zapisnika i u kojim arhivama se nalaze. Nakon listanja prikazuje se ponovo početni meni.

Odabirom **šeste** opcije od korisnika se traži da unese ime arhive i ime zapisnika koji želi da mu se prikaže. Ukoliko je unos pravilan i ukoliko datoteka postoji, prikazuje mu se njen sadržaj, u suprotnom se prikazuje poruka o grešci, a zatim i početni meni.

Odabirom **sedme** opcije od korisnika se traži da unese ime arhive i ime zapisnika koji želi da obriše. Nakon brisanja prikazuje se poruka da je zapisnik sa datim imenom obrisan, a zatim i početni meni.

Odabirom **osme** opcije korisnik napušta aplikaciju.

Aplikaciju treba realizovati u okviru dva modula: **arhiviranje.glavni** i **arhiviranje.radSaDatotekama.** U okviru drugog modula kreirane su klase koje su potrebne za obavljanje poslova sa datotekama i direktorijumima, dok se u prvom obavljaju interakcije sa korisnikom.

Laboratorijska vežba 5: Niti

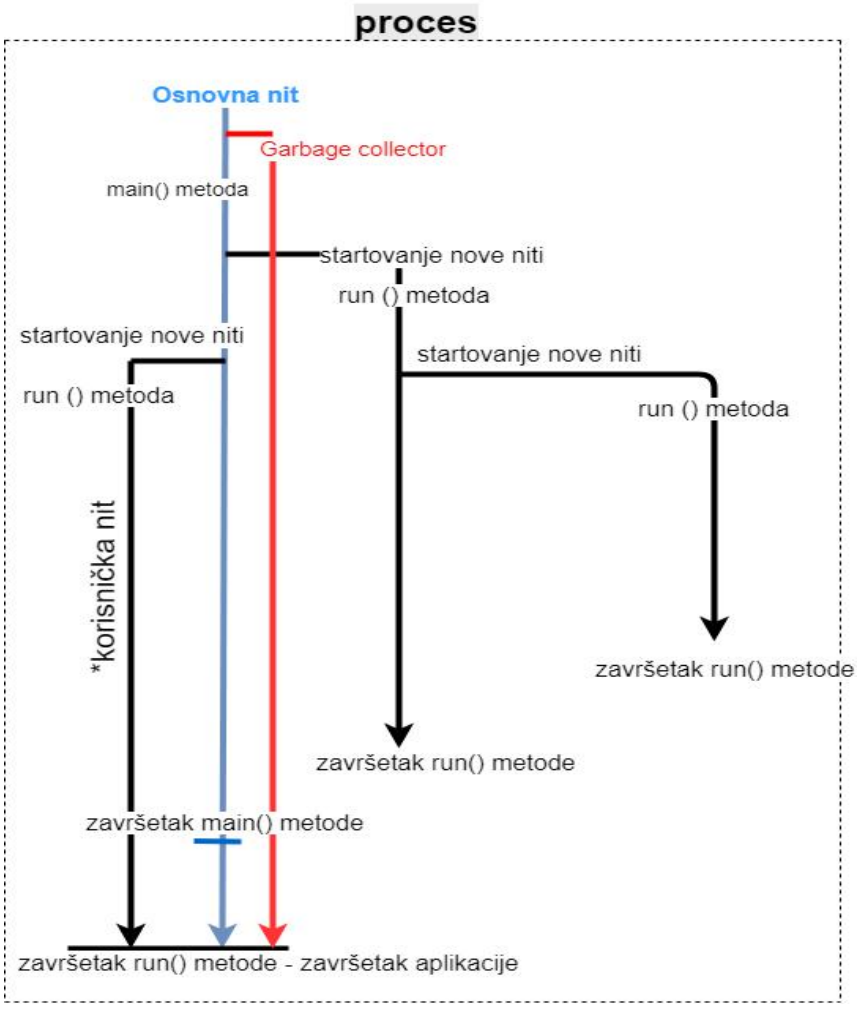
Skoro svaka moderna aplikacija zahteva konkurentno izvršavanje određenih zadataka. Konkurentnost se ogleda u tome da se jedan zadatak razloži na više manjih celina koje će se zatim paralelno izvršavati. Treba precizirati da se radi o uslovno paralelnom izvođenju jer se pojedinačne celine ne startuju i ne završavaju u isto vreme. Konkurentna obrada može se realizovati korisćenjem niti (eng. **thread**). Niti predstavljaju zadatke koji se paralelno izvršavaju unutar jednog procesa. Kreiranjem niti ne kreira se nov proces tj. kreira se takozvani laki proces (eng. lightweight process). Niti se nalaze unutar klasičnog, teškog (eng. heavyweight) procesa i dele memoriju koja mu je dodeljena što olakšava njihovu međusobnu komunikaciju. Na dosadašnjim primerima prikazani su zadaci koji su se izvršavali unutar jedne niti.

Kreiranje niti u Javi realizuje se korišćenjem klase **Thread** iz modula **java.base** , paket **java.lang**. Klasa poseduje 43 metode od kojih su 6 zastarele (eng. deprecated). Često korišćene metode su:

- `activeCount()`– statička metoda koja vraća broj aktivnih metoda unutar trenutne nitske grupe (klasa **ThreadGroup**) i njenih podgrupa. Povratni tip je **int**.
- `currentThread()`– statička metoda koja vraća nit iz koje je pozvana metoda. Povratni tip je **Thread.**
- `sleep(long millis)`– statička metoda koja pauzira dalje izvršavanje date niti za vreme koje se prosledi metodi. Prosleđeno vreme izraženo je u milisekundama**.** Metoda podiže **InterruptedException.**
- `isAlive()`- proverava da li je nit aktivna ili nije. Povratni tip je **boolean.**
- `isDaemon()`– ispituje da li je nit demonska ili nije. Povratni tip je **boolean**.
- `setDaemon(boolean on)`– postavlja da je nit demonska ili nedemonska u zavisnosti od prosleđene vrednosti. Povratni tip je **void.**
- `start()`– startuje nit. Povratni tip je **void.**
- `run()`– definiše ponašanje niti ili ne radi ništa u zavisnosti od načina na koji se kreira nit. Povratni tip je **void.**
- `getId()`– vraća jedinstveni identifikator niti. Povratni tip je **long.**
- `getName()`– vraća ime niti. Povratni tip je **String.**
- `setName(String name)`– postavlja ime niti. Povratni tip je **void.**
- `getPriority()`– vraća prioritet niti. Povratni tip je **int.**
- `setPriority(int newPriority)`- postavlja prioritet niti. Povratni tip je **void.**
- `join()`– stopira nit u kojoj se nalazi do trenutka kad data niti postane neaktivna.
- `yield()`– predaje procesor sledećoj niti.

Kontrolu dobijanja procesorskog vremena kontroliše operativni sistem, dok kontrolu izvršavanja niti kontroliše Java. Odlučivanje o tome koja će se nit prva izvršiti obično se vrši na osnovu toga koliko je nit čekala i kog je prioriteta. Prioriteti se mogu eksplicitno dodeliti, ali smatra se da to nije dobra praksa ukoliko se time želi kontrolisati redosled izvršavanja niti.

U Javi se mogu razlikovati dva različita tipa niti: demonske i nedemonske (eng. daemon i non-daemon). Nedemonska nit naziva se još i korisnička. Razlika između njih je da glavna nit čeka da sve njene nedemonske niti postanu neaktivne, dok se demonske niti prekidaju kada glavna nit postane neaktivna. Roditeljska nit je ona iz koje je nit kreirana. Inicijalno program startuje sa jednom nedemonskom niti u kojoj se poziva metoda **main().** Ova nit se naziva osnovna, glavna ili korenska. Definisanje da li je nit demonska ili ne mora se izvršiti pre njenog startovanja. Primer demonske niti je "sakupljač smeća" (eng. **garbage collector**). Niti su podrazumevano korisničke (nedemonske).



Slika 5.1- Dijagram toka niti

Postoje dva načina na koji je moguće realizovati niti. Prvi način podrazumeva kreiranje nove klase koja proširuje klasu **Thread.** Unutar novokreirane klase potrebno je između ostalog preklopiti **run()** metodu. U okviru ove metode definiše se zadatak koji treba da se obavi u niti. Instanciranjem objekta i pozivom metode **start()** kreira se, priprema i startuje nova nit koja vrši implicitni poziv metode **run().** Kada se završi ova metoda nit postaje neaktivna.

Drugi pristup obuhvata kreiranje klase koja implementira interfejs **Runnable,** a zatim prosleđivanje objekta ove klase konstruktoru klase **Thread**. Interfejs poseduje jednu metodu koju treba implementirati, a to je metoda **run().** Ovim se definiše da se data klasa "ponaša"

kao nit i da kao takva može biti prosleđena konstruktoru klase **Thread.** Pozivom metode **start()** nad dobijenim objektom kreira se i startuje nit u kojoj se implicitno poziva metoda **run()** prosleđenog objekta.

U praksi je češće korišćen drugi način jer se time omogućava da klasa, koja implementira interfejs **Runnable**, nasledi drugu klasu što nije moguće kod prvog načina jer se već vrši nasleđivanje.

Primer realizacije niti korišćenjem nasleđivanja klase **Thread.**

Sadržaj **MojaNit.java** datoteke:

```
package primeri;

public class MojaNit extends Thread {
    private int vreme;
    public MojaNit(String imeN,int vreme) {
        this.vreme = vreme;
        setName(imeN); //metoda iz klase Thread
    }

    @Override
    public void run(){
        System.out.println(this.getName()+" startovana...");
        //stopira se dalje izvrsenje niti za vreme koje je prosledjeno u konstruktoru
        try {
            Thread.sleep(vreme*1000);//vreme se mnozi sa 1000 da bi se trajanje pauze
merilo u sekundama
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(this.getName()+" zavrsena...");
    }
}
```

Sadržaj **Program.java** datoteke:

```
package primeri;

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

        MojaNit mn1=new MojaNit("Prva nit",5);
        System.out.println("Broj aktivnih niti:"+Thread.activeCount());

        System.out.println("Id trenutne niti:"+Thread.currentThread().getId());
        System.out.println("Da li je "+mn1.getName()+"aktivna:"+mn1.isAlive());

        System.out.println("Da li je "+mn1.getName()+"demonska:"+mn1.isDaemon());
        System.out.println("Id niti "+mn1.getName()+":"+mn1.getId());

        // m1.setDaemon(true); //odkomentarisati ovu liniju i ponovo pokrenuti program
        mn1.start();

        /*
        //odkomentarisati ovaj blok i testirati
        try {
            m1.join();
        } catch (InterruptedException e) {
```

```
            e.printStackTrace();
        }
        */

        System.out.println("Da li je "+mn1.getName()+" aktivna:"+mn1.isAlive());
        System.out.println("Broj aktivnih niti:"+Thread.activeCount());

    }
}
```

Primer realizacije niti korišćenjem implementacije interfejsa **Runnable.**

Sadržaj **MojaNit2.java** datoteke:

```
package primeri;

public class MojaNit2 implements Runnable {
    private int vreme;

    public MojaNit2(int vreme) {
        this.vreme = vreme;

    }

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName()+" startovana...");
        //stopira se dalje izvrsenje niti za vreme prosledjeno u konstruktoru
        try {
            Thread.sleep(vreme*1000);//vreme se mnozi sa 1000 da bi se trajanje pauze
merilo u sekundama
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName()+" zavrsena...");
    }
}
```

Sadržaj **Program.java** datoteke:

```
package primeri;

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

        MojaNit2 mn2 = new MojaNit2(5);

        Thread nitMN2 = new Thread(mn2);
        nitMN2.setName("Prva nit");

        System.out.println("Broj aktivnih niti:" + Thread.activeCount());
        System.out.println("Id trenutne niti:" + Thread.currentThread().getId());
        System.out.println("Da li je "+ nitMN2.getName()+" aktivna:" +
nitMN2.isAlive());
        System.out.println("Da li je "+ nitMN2.getName()+" demonska:" +
nitMN2.isDaemon());
        System.out.println("Id niti "+ nitMN2.getName()+":" + nitMN2.getId());
        // nitMN2.setDaemon(true); //odkomentarisati ovu liniju i ponovo pokrenuti
program
        nitMN2.start();

        /*
```

```
        //odkomentarisati ovaj blok i testirati
        try {
            nitMN2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
*/

        System.out.println("Da li je "+ nitMN2.getName()+" aktivna:" +
nitMN2.isAlive());
        System.out.println("Broj aktivnih niti:" + Thread.activeCount());

    }
}
```

Konzolni ispis identičan je za oba slučaja:

```
Broj aktivnih niti:2
Id trenutne niti:1
Da li je Prva nit aktivna:false
Da li je Prva nit demonska:false
Id niti Prva nit:14
Da li je Prva nit aktivna:true
Broj aktivnih niti:3
Prva nit startovana...
Prva nit zavrsena...
```

Zadatak za samostalan rad:

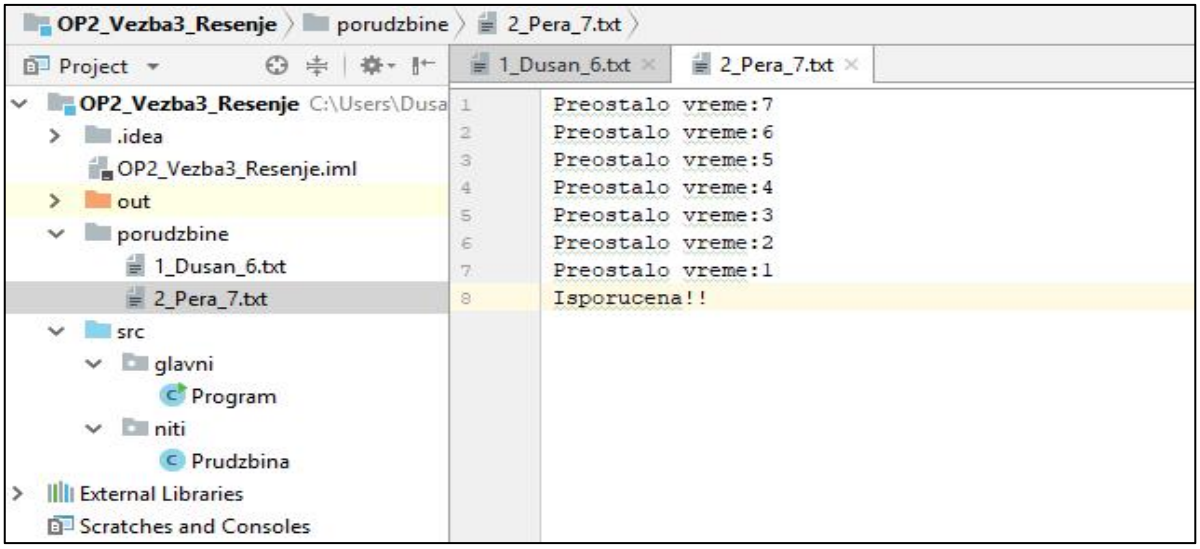
Kreirati Java aplikaciju koja treba da simulira rad kioska brze hrane u kojoj korisnik (radnik) preko konzole treba da unosi porudžbine. Svaka porudžbina sastoji se od: imena poručioca, imena hrane, broj porcija i cene. Korisnik preko konzole treba da unese sve potrebne informacije za porudžbinu. Kada unese poslednji parametar porudžbina se kreira. Proces unosa se ponavlja sve dok korisnik ne unese "kraj" umesto imena poručioca. Porudžbina osim ovih osnovnih informacija ima i vreme potrebno da se ona isporuči (uzima se vrednost uz pomoć generatora slučajnih vrednosti u opsegu od 5-10s) i redni broj. Prilikom startovanja porudžbine kreira se tekstualna datoteka u kojoj se svake sekunde upisuje preostalo vreme ili "isporučena" ukoliko je vreme 0. Imenovati datoteku u formatu *Rednibroj_imePorucioca_vremeIsporuke.txt.* Omogućiti da korisnik može unositi nove porudžbine bez obzira na broj aktivnih porudžbina. Datoteke se nalaze unutar **porudzbine** direktorijuma koji se kreira pri prvom startovanju aplikacije unutar projekta. Aplikacija se završava kada korisnik završi unos i kada se isporuče sve porudžbine.

Primer unosa i ispisa u konzoli:

```
Aplikacija pokrenuta...
Narucilac:Dusan
Jelo:pilece belo
```

```
Kolicina:2
Cena:450
Porudzbina uspesno unesena!
Narucilac:Pera
Jelo:pljeskavica
Kolicina:5
Cena:1250
Porudzbina uspesno unesena!
Narucilac:Kraj
Radno vreme je zavrseno, ne primaju se vise porudzbine!!!
```

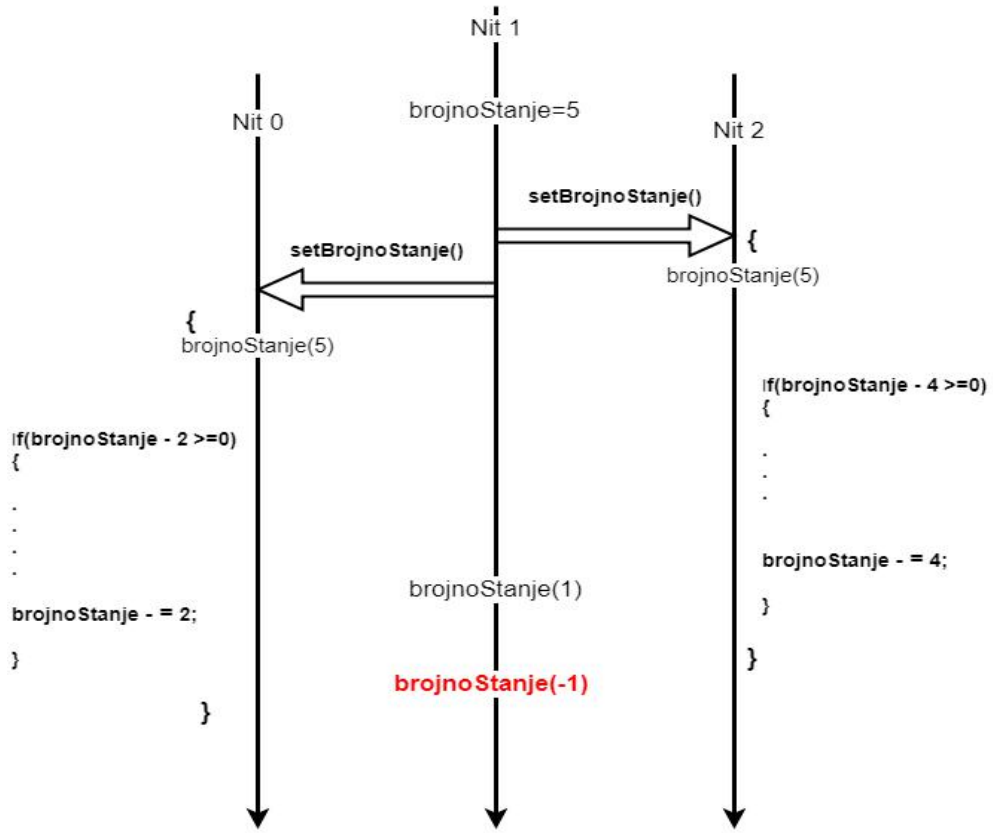
Rezultat izvršenja:



Slika 5.2- Rezultat izvršenja

Laboratorijska vežba 6: Sinhronizaciji niti

U prethodnom poglavlju rečeno je da niti dele zajedničku memoriju unutar procesa što olakšava komunikaciju između njih. Ovo može da predstavlja veliki problem u situacijama kada niti koriste zajednički objekat. U tom slučaju neizbežna je situacija da će doći do korišćenja neažurnog stanja objekta što će se odraziti na dalju validnost podataka - obrade. Na primer ako dve niti koriste zajednički objekat **brojnoStanje** koji treba da izmene u nekom trenutku, problem nastaje kada nit uzima vrednost u periodu dok druga nit vrši promene. U tom trenutku nit će imati neažurirano stanje, izvršiće izmene i vratiti nazad objekat. Na primer objekat **brojnoStanje** ne sme imati negativnu vrednost, ali kako niti nemaju informacije o tome da možda druga nit menja njegovo stanje može se desiti da dođe do toga da u jednom trenutku poprimi negativnu vrednost. Na slici 6.1 prikazan je grafik koji ilustruje ovakvu situaciju.



Slika 6.1- Ilustracija pristupa zajedničkom objektu bez sinhronizacije

Postupkom sinhronizacije može se izbeći ovakav scenario. Sinhronizacijom se vrši zaključavanje objekta korišćenjem brava (eng. lock). Kada nit prihvati deljeni objekat, postavlja bravu tako da ostale niti ne mogu obavljati operacije nad njim. Nakon izvršenih promena, objekat se otključava kako bi se drugim nitima omogućio pristup. Ovakvim rešenjem obezbeđuje se korišćenje objekta sa ažurnim stanjem.

Sinhronizacija se može realizovati direktno nad objektom ili nad metodom koja mu menja stanje. Da bi metoda bila sinhronizovana u potpisu mora da sadrži rezervisanu reč **synchronized.** Sinhronizacija objekta vrši se tako što se deo u kome se menja stanje objekta

navodi u **synchronized** bloku. U izvedenim klasama metode ne nasleđuju ovo svojstvo iz natklasa.

Primer metode sa blokom za deljenje objekta:

```
public boolean prodajArtikal(int id) {
    synchronized (artikli) {
        for (Artikal a : artikli) {
            if (a.getId() == id) {

                a.setKolicina(a.getKolicina() - 1);
                return true;
            }
        }
    }
}
```

Primer sinhrone metode:

```
public synchronized boolean prodajArtikal(int id) {

        for (Artikal a : artikli) {
            if (a.getId() == id) {
                a.setKolicina(a.getKolicina() - 1);
                return true;
            }
        }
}
```

U sledećem primeru simuliran je proces konkurentne kupovine. Objekat klase **Prodavnica** je zajednički za kupce. U simulaciji imamo tri kupca koji istovremeno žele da izvrše kupovinu. Prodavnica ima sinhronizovanu metodu **prodaj()** kojoj se prosleđuje celobrojna vrednost koja predstavlja količinu artikala.

Sadržaj **Prodavnica.java** datoteke:

```
package primer;

public class Prodavnica {
    private int brojnoStanje;

    public Prodavnica(int brojnoStanje) {
        this.brojnoStanje = brojnoStanje;
    }


    //izostaviti sinchronized iz potpisa i videti rezultat izvrsenja.Pokrenuti program
nekoliko puta.
    public synchronized void prodaj(int n) {
        System.out.println(Thread.currentThread().getName() + " zahteva kupovinu " + n
+ " proizvoda...");
        if (brojnoStanje - n >= 0) {

            brojnoStanje -= n;
            System.out.println(Thread.currentThread().getName() + " izvrsio
kupovinu,novo brojno stanje:" + this.brojnoStanje);
        }
    }
```

```
    @Override
    public String toString() {
        return "Prodavnica{" +
                "brojnoStanje=" + brojnoStanje +
                '}';
    }
}
```

Sadržaj **Kupac.java** datoteke:

```
package primer;

public class Kupac implements Runnable {
    private int kolicina;
    private Prodavnica p;
    private int vreme;

    public Kupac(Prodavnica p,int kolicina,int vreme) {
        this.p=p;
        this.kolicina = kolicina;
        this.vreme=vreme;
    }

    @Override
    public void run() {
        //opciono se moze napraviti pauza tokom izvrsavanja u zavisnosti od vrednosti
koja je dodeljena atributu vreme
        try {
            Thread.sleep(vreme*1000);

        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        this.p.prodaj(kolicina);
    }
}
```

Sadržaj **Program.java** datoteke:

```
package primer;

package primer;

public class Program {
    public static void main(String[] args) {
        Prodavnica p = new Prodavnica(15);
        System.out.println(p);

        Kupac prvi = new Kupac(p, 1, 0);

        Thread prviNit = new Thread(prvi);
        prviNit.setName("Prvi kupac");

        Kupac drugi = new Kupac(p, 2, 0);
        Thread drugiNit = new Thread(drugi);
        drugiNit.setName("Drugi kupac");

        Kupac treci = new Kupac(p, 2, 0);
        Thread treciNit = new Thread(treci);
        treciNit.setName("Treci kupac");
```

```
        Kupac cetvrti = new Kupac(p, 2, 0);
        Thread cetvrtiNit = new Thread(cetvrti);
        cetvrtiNit.setName("Cetvrti kupac");

        prviNit.start();
        drugiNit.start();
        treciNit.start();
        cetvrtiNit.start();
    }
}
```

Konzolni ispis:

```
Prodavnica{brojnoStanje=15}
Prvi kupac zahteva kupovinu 1 proizvoda...
Prvi kupac izvrsio kupovinu,novo brojno stanje:14
Treci kupac zahteva kupovinu 2 proizvoda...
Treci kupac izvrsio kupovinu,novo brojno stanje:12
Drugi kupac zahteva kupovinu 2 proizvoda...
Drugi kupac izvrsio kupovinu,novo brojno stanje:10
Cetvrti kupac zahteva kupovinu 2 proizvoda...
Cetvrti kupac izvrsio kupovinu,novo brojno stanje:8
```

Prethodni primer može se proširiti scenariom u kojem se brojno stanje artikala u prodavnici periodično dopunjuje novom količinom. To će dalje zahtevati modifikaciju same sinhronizacije kako bi se povećala efikasnost izvršenja. Novo stanje u kome nit može da se nađe je stanje čekanja. Ukoliko stanje deljenog resursa ne odgovara niti, ući će u stanje čekanja. U tom periodu nit privremeno predaje objekat na korišćenje drugim nitima sve dok se ne ažurira stanje objekta. Tom prilikom potrebno je obavestiti nit da je promenjeno stanje kako bi ponovo nit, koja čeka promenu, uzela objekat na korišćenje. Pozivom metode **wait()** nit ulazi u stanje čekanja i oslobađa objekat. Metoda ima tri potpisa: **wait(), wait(long milisec)** i **wait(long milisec, int nanosec).** Prosleđene vrednosti odnose se na vreme (milisekunde, nanosekunde) za koje nit treba da bude u stanju čekanja. Ukoliko se ne prosledi parametar, čekanje se vrši do promene stanja ili prinudnog prekidanja (eng. interrupt, u daljem tekstu interapt). Metoda podiže **InterruptedException** izuzetak**.** Metode **notify()** i **notifyAll()** obaveštavaju da je stanje objekta ažurirano. Razlika između njih je u načinu na koji vrše obaveštavanje. Pošto nije isključena mogućnost da više niti istovremeno budu u režimu čekanja mogu se obavestiti sve niti da je ažurirano stanje (**notifyAll()**) ili samo nit koja je najduže čekala (**notify()**). Prethodne metode nalaze se u klasi **Object** tako da ih je moguće pozvati u bilo kojoj klasi, s tim da se poziv mora izvršiti u okviru sinhronizovane metode ili nad deljenim objektom.

Prethodni primer će prema tome imati još jednu nit (**Magacin)** u kojoj će se količina brojnog stanja u prodavnici periodično povećavati. Kupac će, sa druge strane, da sačeka koliko je potrebno kako bi izvršio kupovinu. Prilikom promene stanja, kupac koji je najduže čekao imaće prednost u odnosu na ostale.

Nov sadržaj **Prodavnica.java** datoteke:

```
package primer;
public class Prodavnica {
    private int brojnoStanje;
    public Prodavnica(int brojnoStanje) {
        this.brojnoStanje = brojnoStanje;
    }
    public synchronized void prodaj(int n) {
        System.out.println(Thread.currentThread().getName() + " zahteva kupovinu " + n
+ " proizvoda...");
        //sistem koji osigurava da ce se prodaja izvrsiti tek kada na stanju bude
dodvoljne kolicine
        while (brojnoStanje - n < 0) {
            System.out.println(Thread.currentThread().getName() + " nije izvrsio
kupovinu,ceka na novu kolicinu!!");
            try {
                wait(); //zaustavlja dalje izvrsenje dok ce ne pozove notify ili
notfyAll metoda
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        brojnoStanje -= n;
        System.out.println(Thread.currentThread().getName() + " izvrsio kupovinu,novo
brojno stanje:" + this.brojnoStanje);
        notify();// obavestava da mozda na stanju ima dovoljne kolicine da sledeci
kupac izvrsi kupovinu, da ne ceka azurranje iz magacina

    }

    public synchronized void azurirajStanje(int kolicina) {
        brojnoStanje += kolicina;
        System.out.println("Azurirano brojno stanje!!");
        notify();
    }

    @Override
    public String toString() {
        return "Prodavnica{" +
                "brojnoStanje=" + brojnoStanje +
                '}';
    }
}
```

Nov sadržaj **Program.java** datoteke:

```
package primer;


public class Program {


    public static void main(String[] args) {
        Prodavnica p = new Prodavnica(7);
        System.out.println(p);
        Kupac prvi = new Kupac(p, 5, 0);
        Thread prviNit = new Thread(prvi);
        prviNit.setName("Prvi kupac");
        Kupac drugi = new Kupac(p, 6, 0);
```

```
        Thread drugiNit = new Thread(drugi);
        drugiNit.setName("Drugi kupac");
        Kupac treci = new Kupac(p, 4, 0);
        Thread treciNit = new Thread(treci);
        treciNit.setName("Treci kupac");
        Magacin m=new Magacin(p);
        Thread mNit=new Thread(m);
        mNit.setName("Magacin");
        mNit.start();
        prviNit.start();
        drugiNit.start();
        treciNit.start();
    }
}
```

Sadržaj **Magacin.java** datoteke:

```
package primer;

public class Magacin implements Runnable {
    private Prodavnica p;

    public Magacin(Prodavnica p) {
        this.p = p;
    }

    @Override
    public void run() {
        System.out.println("Magacin startovan," + p.toString());
        for (int i = 0; i < 5; i++) {
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            p.azurirajStanje(5);

        }
    }
}
```

Sadržaj **Kupac.java** datoteke ostaje nepromenjen.

Konzolni ispis:

```
Prodavnica{brojnoStanje=7}
Magacin startovan,Prodavnica{brojnoStanje=7}
Prvi kupac zahteva kupovinu 5 proizvoda...
Prvi kupac izvrsio kupovinu,novo brojno stanje:2
Treci kupac zahteva kupovinu 4 proizvoda...
Treci kupac nije izvrsio kupovinu,ceka na novu kolicinu!!
Drugi kupac zahteva kupovinu 6 proizvoda...
Drugi kupac nije izvrsio kupovinu,ceka na novu kolicinu!!
Azurirano brojno stanje!!
Treci kupac izvrsio kupovinu,novo brojno stanje:3
```

```
Drugi kupac nije izvrsio kupovinu,ceka na novu kolicinu!!
Azurirano brojno stanje!!
Drugi kupac izvrsio kupovinu,novo brojno stanje:2
Azurirano brojno stanje!!
Azurirano brojno stanje!!
Azurirano brojno stanje!!
```

Prekidi (eng. interrupt) šalju posebne signale niti kako bi promenila stanje u kome se nalazi i izvršila nesto drugo. Programer je zadužen za definisanje scenarija koji će se izvršiti i uglavnom se koriste kako bi se nit "nasilno" prekinula. Nit mora biti suspendovana od strane **sleep()**, **wait()** ili **join()** metoda da bi prihvatila prekid. Prekidom se podiže **InterruptedException** izuzetak u okviru bloka gde je neka od prethodnih metoda pozvana. Ukoliko se u **catch** klauzoli navede **return**, trenutno će se završiti **run()** metoda i nit će postati neaktivna.

Sadržaj **PrimerNit.java** datoteke:

```
package primer;

public class PrimerNit implements Runnable {
    @Override
    public void run() {
        System.out.println("Startovana nit...");
        //nit pauzirana 5 sekundi
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            System.out.println("Dogodio se interapt...");
            return;//zakomentarisati ovu liniju i ponovo pokrenuti pogram
        }
        System.out.println("Kraj run metode...");
    }
}
```

Sadržaj **Program.java** datoteke:

```
package primer;

public class Program {
    public static void main(String[] args) {
        PrimerNit pn=new PrimerNit();
        Thread t=new Thread(pn);
        t.start();
        //pauza od 2 sekundi
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        t.interrupt();//slanje siignala za prekid
    }
}
```

Konzolni ispis:

```
Startovana nit...
```

```
Dogodio se interapt...
```

Zadatak za samostalan rad:

Kreirati aplikaciju koja će simulirati rad benzinske pumpe. Pretpostavka je da pumpa ima jednu stanicu na kojoj mušterije- automobili toče gorivo. Pumpa ima kapacitet od 80 l dok automobili toče količinu goriva između 10 l – 25 l. Količinu goriva odrediti pomoću generatora slučajnih vrednosti. Pumpa se periodično dopunjuje novom količinom iz cisterne. Cisterna vrši dopunu na svakih 7 s. Ukoliko na pumpi nema dovoljno goriva, automobil treba da sačeka cisternu da je dopuni. Za to vreme prepušta stanicu na korišćenje drugim automobilima. U simulaciji treba napraviti 15 automobila koji pristaju na pumpu u vremenskim intervalima od 1 – 2 s. Ispisivati poruke o trenutnim stanjima u kojima se ovi objekti nalaze.

Primer:

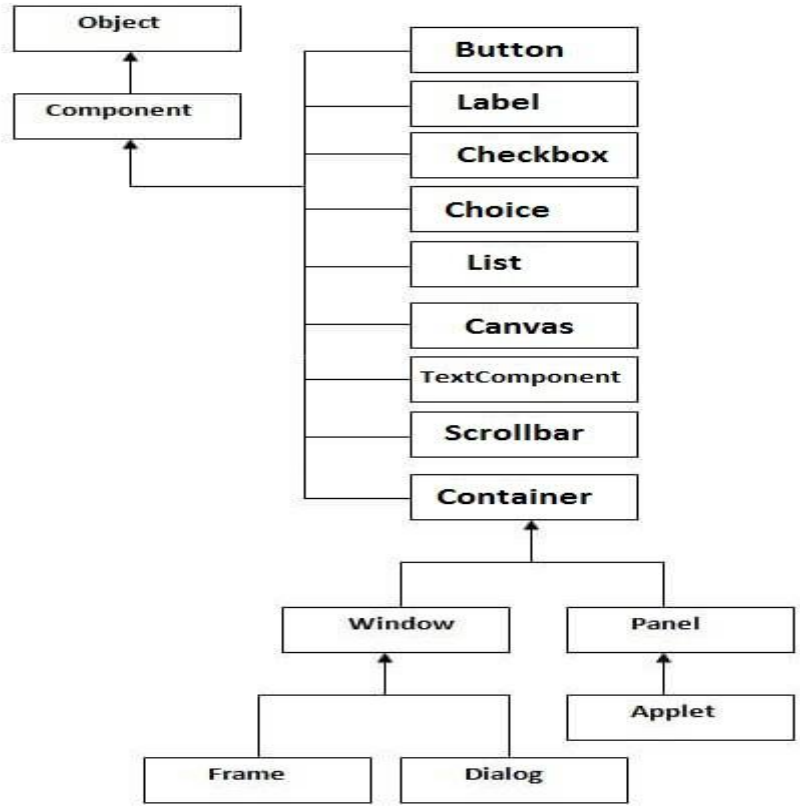
```
Automobil (regBR:RB001) pristaje na pumpu. Trazena kolicina 21.11 L.
Pumpa raspolaze sledecom kolicinom: 50.00

Automobil (regBR:RB001) natocio gorivo. Trenutna kolicina na pumpi:28.89

Automobil (regBR:RB002) pristaje na pumpu. Trazena kolicina 19.53 L.
Pumpa raspolaze sledecom kolicinom: 28.89

Automobil (regBR:RB002) natocio gorivo. Trenutna kolicina na pumpi:9.36

Automobil (regBR:RB003) pristaje na pumpu. Trazena kolicina 10.06 L.
Pumpa raspolaze sledecom kolicinom: 9.36

Automobil (regBR:RB003) ne moze da natoci gorivo, ceka dopunu...

Automobil (regBR:RB004) pristaje na pumpu. Trazena kolicina 19.25 L.
Pumpa raspolaze sledecom kolicinom: 9.36

Automobil (regBR:RB004) ne moze da natoci gorivo, ceka dopunu...

Automobil (regBR:RB005) pristaje na pumpu. Trazena kolicina 20.42 L.
Pumpa raspolaze sledecom kolicinom: 9.36

Automobil (regBR:RB005) ne moze da natoci gorivo, ceka dopunu...

Automobil (regBR:RB006) pristaje na pumpu. Trazena kolicina 13.92 L.
Pumpa raspolaze sledecom kolicinom: 9.36

Automobil (regBR:RB006) ne moze da natoci gorivo, ceka dopunu...

Automobil (regBR:RB007) pristaje na pumpu. Trazena kolicina 16.58 L.
Pumpa raspolaze sledecom kolicinom: 9.36

Automobil (regBR:RB007) ne moze da natoci gorivo, ceka dopunu...

Cisterna izvrsila dopunu.

Automobil (regBR:RB003) natocio gorivo. Trenutna kolicina na pumpi:69.94

Automobil (regBR:RB007) natocio gorivo. Trenutna kolicina na pumpi:53.36

Automobil (regBR:RB006) natocio gorivo. Trenutna kolicina na pumpi:39.44

Automobil (regBR:RB005) natocio gorivo. Trenutna kolicina na pumpi:19.02

Automobil (regBR:RB004) ne moze da natoci gorivo, ceka dopunu...
```

```
Automobil (regBR:RB008) pristaje na pumpu. Trazena kolicina 15.42 L.
Pumpa raspolaze sledecom kolicinom: 19.02

Automobil (regBR:RB008) natocio gorivo. Trenutna kolicina na pumpi:3.60

Automobil (regBR:RB009) pristaje na pumpu. Trazena kolicina 18.02 L.
Pumpa raspolaze sledecom kolicinom: 3.60

Automobil (regBR:RB009) ne moze da natoci gorivo, ceka dopunu...

Automobil (regBR:RB0010) pristaje na pumpu. Trazena kolicina 13.27 L.
Pumpa raspolaze sledecom kolicinom: 3.60

Automobil (regBR:RB0010) ne moze da natoci gorivo, ceka dopunu...

Automobil (regBR:RB0011) pristaje na pumpu. Trazena kolicina 21.41 L.
Pumpa raspolaze sledecom kolicinom: 3.60

Automobil (regBR:RB0011) ne moze da natoci gorivo, ceka dopunu...

Automobil (regBR:RB0012) pristaje na pumpu. Trazena kolicina 20.72 L.
Pumpa raspolaze sledecom kolicinom: 3.60

Automobil (regBR:RB0012) ne moze da natoci gorivo, ceka dopunu...

Automobil (regBR:RB0013) pristaje na pumpu. Trazena kolicina 12.96 L.
Pumpa raspolaze sledecom kolicinom: 3.60

Automobil (regBR:RB0013) ne moze da natoci gorivo, ceka dopunu...

Automobil (regBR:RB0014) pristaje na pumpu. Trazena kolicina 13.17 L.
Pumpa raspolaze sledecom kolicinom: 3.60

Automobil (regBR:RB0014) ne moze da natoci gorivo, ceka dopunu...

Cisterna izvrsila dopunu.

Automobil (regBR:RB004) natocio gorivo. Trenutna kolicina na pumpi:60.75

Automobil (regBR:RB0014) natocio gorivo. Trenutna kolicina na pumpi:47.58

Automobil (regBR:RB0013) natocio gorivo. Trenutna kolicina na pumpi:34.62

Automobil (regBR:RB0012) natocio gorivo. Trenutna kolicina na pumpi:13.91

Automobil (regBR:RB0011) ne moze da natoci gorivo, ceka dopunu...

Automobil (regBR:RB0010) natocio gorivo. Trenutna kolicina na pumpi:0.64

Automobil (regBR:RB009) ne moze da natoci gorivo, ceka dopunu...

Automobil (regBR:RB0015) pristaje na pumpu. Trazena kolicina 22.82 L.
Pumpa raspolaze sledecom kolicinom: 0.64

Automobil (regBR:RB0015) ne moze da natoci gorivo, ceka dopunu...

Cisterna izvrsila dopunu.

Automobil (regBR:RB0011) natocio gorivo. Trenutna kolicina na pumpi:58.59

Automobil (regBR:RB0015) natocio gorivo. Trenutna kolicina na pumpi:35.76

Automobil (regBR:RB009) natocio gorivo. Trenutna kolicina na pumpi:17.74
```

Laboratorijska vežba 7: Korisnički grafički interfejs, AWT biblioteka

Zbog složenosti same interakcije između aplikacije i korisnika nije moguće uvek koristiti konzolni pristup te je tako upotreba GUI neizbežna. Korisnički grafički interfejs (GUI-Graphical user interface) služi korisniku da vrši interakciju sa aplikacijom putem grafičkih elemenata. Grafički elementi namenjeni su da korisnik pomoću njih pokrene neke operacije, vidi rezultat izvršenja ili unese potrebne podatke. Aplikacija sa GUI upravlja samo delom ekrana (prozorima) unutar kojeg se iscrtavaju grafičke komponente.

AWT (Abstract Windowing Toolkit) biblioteka nalazi se u okviru modula **java.desktop**, paket **java.awt.** Biblioteka nudi skup klasa pomoću kojih se može kreirati korisnički interfejs, kao i grafički prikaz crteža i slika. Objekti koji služe za kreiranje korisničkog interfejsa kao što su: dugme (eng. Button), tekst polje (eng. TextField), labela (eng. Label), klizači (eng. Scroll bar) itd. jednim imenom u ovoj biblioteci se nazivaju komponente (eng. Component). Klasa **Component** je zajednička za sve komponente unutar biblioteke. Unutar nje definisane su neke od osnovnih osobina kao što je pozicija, veličina, sposobnost da se iscrta na ekranu i da prihvati određene događaje. Klase koje direktno nasleđuju ovu klasu su: **Button, Canvas, Checkbox, Choice, Container, Label, List, Scrollbar, TextComponent.** Za kreiranje prozora treba obratiti pažnju na klasu **Container**. Klase koje nasleđuju ovu klasu sposobne su, između ostalog, da sadrže druge komponente. Na slici 7.1 prikazana je hijerarhija nekih od bitnijih klasa.



Slika 7.1 – Hijerarhija bitnijih klasa AWT biblioteke

Klasa **Window** je kontejnerska i njeni objekti su prozori najvišeg nivoa. Klasa **Frame** nasleđuje ovu klasu i služi za kreiranje glavnog prozora aplikacije. Objekti ovog tipa sposobni su da prikažu naslov i traku menija (eng. menu bar). Često korišćene metode ove klase:

`setTitle(String title)`– postavlja naslov prozora na prosleđenu vrednost

`getTitle()`- vraća naslov prozora. Povratni tip je **String.**

`setSize(int width,int height)`- nasleđena metoda iz **Window** klase. Postavlja veličinu prozora na osnovu prosleđenih parametara (širina, visina). Prosleđene vrednosti predstavljaju broj piksela po x i y osi.

`setVisible(boolean b)`– nasleđena metoda iz **Window** klase. Menja vidljivost prozora na prosleđenu vrednost. Podrazumevana vidljivost je **false.**

`setLocation(int x, int y)`- nasleđena metoda iz **Window** klase. Menja lokaciju prozora. Prosleđene vrednosti predstavljaju broj piksela od gornjeg levog ugla ekrana- nulte tačke (0,0) i gornje leve ivice prozora. Prosleđene vrednosti odnose se na rastojanje po x i y osi.

`paint(Graphics g)`- nasleđena metoda iz **Window** klase. Pomoću prosleđenog objekta tipa **Graphics** iscrtava grafičke elemente na ekranu.

`setBackground(Color c)`- nasleđena metoda iz **Window** klase. Postavlja pozadinsku boju na vrednost koja se prosleđuje kao parametar.

`getBackground()`- nasleđena metoda iz **Window** klase. Vraća pozadinsku boju. Povratni tip je **Color.**

`setBounds(int x, int y, int weight, int height)`– nasleđena metoda iz **Window** klase. Postavlja novu poziciju i novu veličinu prozora. Prva dva parametra odnose se na rastojanje po x i y osi od nulte tačke. Druga dva parametra odnose se na širinu i visinu prozora.

`add(Component c)`– nasleđena metoda iz **Container** klase. Dodaje komponentu unutar kontejnerskog objekta na sledeću slobodnu poziciju.

`add(Component c, int index)`– nasleđena metoda iz **Container** klase. Dodaje komponentu unutar kontejnerskog objekta na polje sa datim indeksom.

`add(String name,Component c)`– nasleđena metoda iz **Container** klase. Dodaje komponentu unutar kontejnerskog objekta na polje koje je specificirano imenom koje se prosleđuje kao prvi parametar.

`setLayout(LayoutManager mgr)`– nasleđena metoda iz **Container** klase. Postavlja raspoređivač (eng. layout manager) na vrednost koja se prosleđuje kao parametar.

`setFont(Font f)`– nasleđena metoda iz **Container** klase. Postavlja font na vrednost koja se prosleđuje kao parametar.

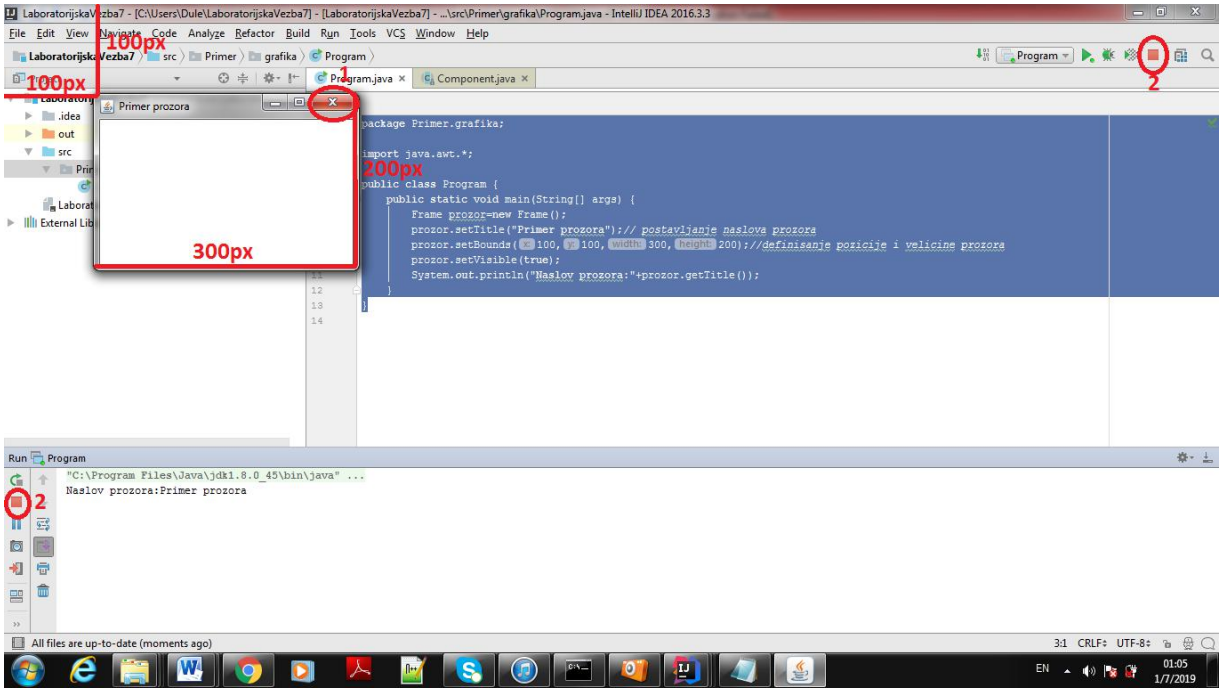
U sledećem delu prikazan je primer kreiranja osnovnog prozora bez dodatnih komponenti.

Sadržaj **Program.java** datoteke**:**

```
package Primer.grafika;

import java.awt.*;

public class Program {
    public static void main(String[] args) {
        Frame prozor=new Frame();
        prozor.setTitle("Primer prozora");// postavljanje naslova prozora
        prozor.setBounds(100,100,300,200);//definisanje pozicije i velicine prozora
        prozor.setVisible(true);
        System.out.println("Naslov prozora:"+prozor.getTitle());
    }
}
```

Konzolni ispis:

Naslov prozora: Primer prozora



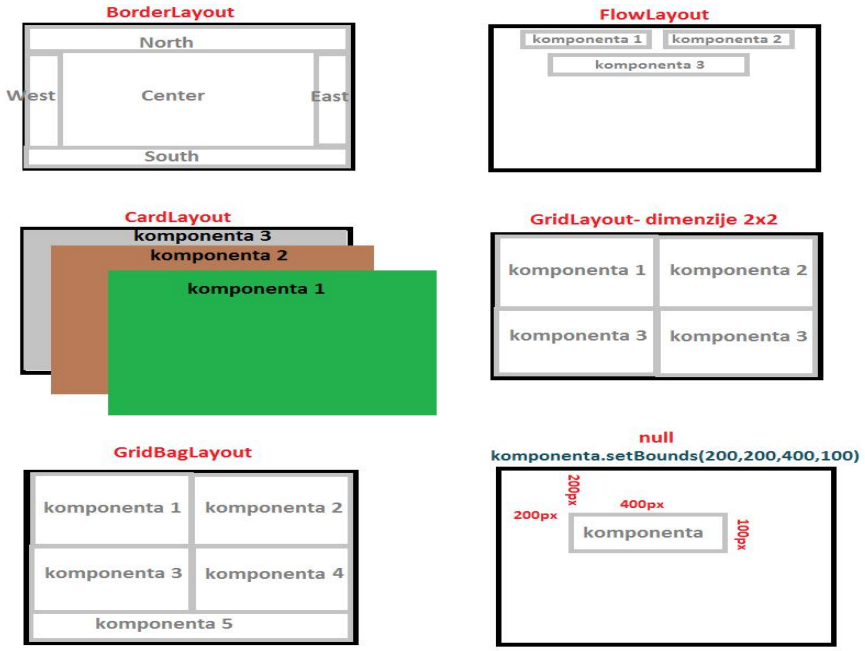
Slika 7.2- Glavni prozor

Kreirani prozor za sada nema definisanu ni jednu funkcionalnost osim operacija za maksimiziranje i minimiziranje prozora. Zatvaranje prozora izvršava se pomoću razvojnog okruženja pritiskom na tastere koji su na slici obeleženi brojem 2. Zatvaranje na podrazumevani način, taster obeležen brojem 1, nije moguće dok se ne postavi osluškivač (eng. listener) na sam prozor.

U ovom primeru nema nijednog komandnog objekta tako da prozor nema neku posebnu funkcionalnost. Dodavanje komandi i drugih kontejnerskih objekata vrši se pomoću metode **add().** Komponente se dalje raspoređuju po nekom od dostupnih uređivača rasporeda (eng. layout manager). Uređivač definiše poziciju i veličinu komponenti po nekom od definisanih

šablona i poseduje ih svaki kontejnerski objekat. Svaka klasa uređivača rasporeda implementira interfejs **LayoutManager.** Podrazumevani raspored za glavni prozor je **BorderLayout.** Dostupni uređivači rasporeda:

1. **FlowLayout** koji komponente u kontejneru raspoređuje po redovima u nizovima sleva-udesno.
2. **BorderLayout** koja komponente u kontejneru raspoređuje po ivicama i u sredini kontejnera.
3. **CardLayout** koja komponente u kontejneru raspoređuje kao jednu iza druge (kao špil karata).
4. **GridLayout** koja komponente u kontejneru raspoređuje matrično.
5. **GridBagLayout** koja komponente u kontejneru raspoređuje prema skupu objekata **GridBagConstraints**.
6. **null-layout** koja omogućuje da programer eksplicitno odredi poziciju i veličinu komponente.



Slika 7.2- Raspoređivači

Panel je najjednostavnija kontejnerska klasa koja služi za grupisanje komponenti. Paneli se mogu nalaziti direktno unutar glavnog prozora ili unutar drugih kontejnera – drugih panela. U sledećem delu dat je primer korišćenja raspoređivača, panela i češće korišćenih komandi. U primeru je korišćena klasa **Prozor** koja proširuje klasu **Frame**.

Sadržaj **PrimerRasporeda.java** datoteke:

```
package grafika;
import java.awt.*;
public class PrimerRasporeda extends Frame {
    String nazivkarte[] = {"prva karta", "druga karta",
            "treca karta", "cetvrta karta", "peta karta"};
    public PrimerRasporeda() {
        super("Raspored"); //poziv konstruktora natklase (Frame). Prosledjeni parametar
se odnosi na naslov prozora.
```

```
        setLayout(new GridLayout(3, 2));
        setSize(500, 700);
        dodajBorderPanel();
        dodajFlowPanel();
        dodajCardPanel();
        dodajGridPanel();
        dodajGridBagPanel();
        dodajNullPanel();
        setVisible(true);
    }
    private void dodajBorderPanel() {
        Panel borderP = new Panel();
        borderP.setLayout(new BorderLayout());
        borderP.add("North", new Label("Pozicija North"));//dodavanje komponente (label)
na poziciju 'Noth'
        borderP.add("East", new Label("Pozicija East"));
        borderP.add("West", new Label("Pozicija West"));
        borderP.add("South", new Label("Pozicija South"));
        borderP.add("Center", new Label("Pozicija Center"));
        borderP.setBackground(new Color(66, 134, 244));//postavljanje pozadinske boje
        add(borderP);
    }
    private void dodajCardPanel() {
        Panel cardPanel = new Panel();
        CardLayout cardlayout = new CardLayout();
        cardPanel.setLayout(cardlayout);
        for (int i = 0; i < nazivkarte.length; i++) {
            Button b = new Button(nazivkarte[i]);
            b.setBackground(new Color(244, 229, 65));
            cardPanel.add(nazivkarte[i], b);
        }
        cardlayout.show(cardPanel, nazivkarte[1]);//menjati indeks u nizu i videti
rezultat izvrsenja
        add(cardPanel);
    }
    private void dodajGridBagPanel() {
        Panel gridBagPanel = new Panel();
        gridBagPanel.setLayout(new GridBagLayout());
        GridBagConstraints pozicija = new GridBagConstraints();
        pozicija.weightx = 0.5;

        pozicija.fill = GridBagConstraints.HORIZONTAL;
        pozicija.ipadx = 0;//postavljanje vrednosti za indeks kolone
        pozicija.gridy = 0;//postavljanje vrednosti za indeks reda
        gridBagPanel.add(new Label("Ime:"), pozicija);
        pozicija.ipadx = 1;
        pozicija.ipady = 0;
        gridBagPanel.add(new TextField(), pozicija);

        pozicija.ipadx = 0;
        pozicija.gridy = 1;
        gridBagPanel.add(new Label("Prezime:"), pozicija);
        pozicija.ipadx = 1;
        pozicija.ipady = 1;
        gridBagPanel.add(new TextField(), pozicija);
        pozicija.gridy = 2;
        pozicija.gridx = 0;
        pozicija.gridwidth = 2;
        gridBagPanel.add(new Button("Posalji"), pozicija);
        gridBagPanel.setBackground(new Color(153, 145, 64));
        add(gridBagPanel);
    }
    private void dodajFlowPanel() {
```

```
        Panel flowP = new Panel();
        flowP.setLayout(new FlowLayout());
        flowP.add(new Button("Dugme 1"));
        flowP.add(new Button("Dugme 2"));
        flowP.add(new Button("Dugme 3"));
        flowP.setBackground(new Color(63, 145, 153));
        add(flowP);
    }
    private void dodajGridPanel() {
        Panel gridP = new Panel();
        gridP.setLayout(new GridLayout(2, 3));
        Label lblPol = new Label("Pol:");
        Font f = new Font("Times New Roman", Font.BOLD, 15);//kreiranje fonta
        lblPol.setFont(f);
        Checkbox cbMuski = new Checkbox("Muski");
        cbMuski.setFont(f);
        Checkbox cbZenski = new Checkbox("Zeski");
        cbZenski.setFont(f);
        //dodavanjem CheckBox objekata u CheckBoxGroup- u definise se ponasanje kao
radio dugmad


        CheckboxGroup cbgPol = new CheckboxGroup();
        cbMuski.setCheckboxGroup(cbgPol);
        cbZenski.setCheckboxGroup(cbgPol);
        gridP.add(lblPol);
        gridP.add(cbMuski);
        gridP.add(cbZenski);
        Checkbox cbIzbor1 = new Checkbox("Izbor 1");
        Checkbox cbIzbor2 = new Checkbox("Izbor 2");
        gridP.add(cbIzbor1);
        gridP.add(cbIzbor2);
        gridP.setBackground(Color.cyan);
        add(gridP);
    }


    private void dodajNullPanel() {
        Panel nullP = new Panel();
        nullP.setLayout(null);
        TextArea taTekst = new TextArea();
        taTekst.setBounds(30, 50, 100, 50);//definisanje pozicije (30,50) i velicine
komponente (100X50)
        nullP.add(taTekst);
        nullP.setBackground(Color.GRAY);
        add(nullP);
    }
}
```

Sadržaj **Program.java** datoteke:

```
package glavni;


import grafika.PrimerRasporeda;
public class Program {
    public static void main(String[] args) {
        PrimerRasporeda pr=new PrimerRasporeda();
    }

}
```

Izgled rezultujućeg prozora prikazan je na slici 7.4. Primetiti kako se veličina i raspored komponenti menja prilikom promena dimenzija prozora.

Slika 7.4- Primer raspoređivača

Metoda **paint()** zadužena je za grafički prikaz svih komponenti. Poziv metode može biti izazvan od samog sistema prilikom inicijalnog prikaza, menjanja veličine komponenti ili ukoliko dođe do greške pa je potrebno obnoviti prikaz. Takođe postoje i pozivi ove metode koji su uzrokovani od samog korisnika. Na primer kada se tasterom miša klikne na dugme, ono treba da promeni pozadinsku boju što zapravo radi ova metoda. Bez obzira ko je uzročnik, poziv se vrši implicitno pomoću svog povratnog mehanizma. Ukoliko dođe do takve situacije programer može da izvrši eksplicitni poziv pozivom metode **repaint().** Metoda **paint()** prima jedan argument klase **Graphics.** Pomoću ovog objekta moguće je iscrtati i druge grafičke elemente kao što su linija, krug, elipsa, pravougaonik, 3D objekat, slika itd. U sledećem delu dat je primer preklapanja ove metode unutar novokreirane klase koja proširuje klasu **Frame.**

Sadržaj **Prozor.java** datoteke:

```
package grafika;
import java.awt.*;
public class Prozor extends Frame {
    public Prozor(){
        setTitle("Primer paint meotode");
        setSize(500,300);
        setVisible(true);
    }
```

```
    @Override
    public void paint(Graphics g) {
        /*metoda koja iscrtava tekstualni sadrzaj.Prvi argument je tekst koji
        se ispisuje dok se druga dva odnose na poziciju*/
        g.drawString("Pozdrav",50,50);
        /*metoda koja isrctava pravougaonik dimenzija 100X100px
        na poziciji 100,50*/
        g.drawRect(100,50,100,100);
        /*metoda koja isrctava popunjeni pravougaonik dimenzija 80X120px
        na poziciji 100,250*/
        g.fillRect(100,160,80,120);
        //promena boje 'olovke', podrazumevana boja je crna,RGB(0,0,0)
        g.setColor(Color.CYAN);
        /*metoda koja isrctava popunjeni pravougaonik dimenzija 80X120px
        na poziciji 200,80*/
        g.fillRect(200,80,80,120);
        //postavljanje fonta
        g.setFont(new Font("Verdana",Font.ITALIC,20));
        g.drawString("Ovo je Verdana font",180,220);
        g.setColor(new Color(50,130,10));
        /*metoda koja iscrtava ovalni oblik dimenzija 70X90px
        na poziciji 250,100
         */
        g.fillOval(250,100,70,90);
    }
}
```

Sadržaj **Program.java** datoteke:

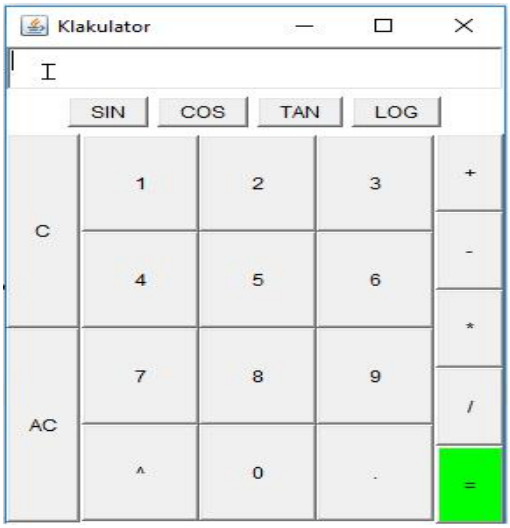
```
package glavni;
import grafika.Prozor;
public class Program {
    public static void main(String[] args) {
        Prozor p=new Prozor();

    }
}
```

Izgled rezultujućeg prozora prikazan je na slici 7.4.

*Slika 7.4- Primer korišćenja **paint()** metode*

Zadatak za samostalan rad:

Kreirati grafički korisnički interfejs sa sadržajem koji je prikazan na slici 7.5:



Slika 7.5- GUI Kalkulator

Laboratorijska vežba 8: AWT osluškivači

U prethodnoj vežbi prikazan je postupak kreiranja grafičkog interfejsa koji nije posedovao sposobnost da reaguje na akcije od strane korisnika. Da bi grafički interfejs bio funkcionalan, na komponentama se moraju postaviti osluškivači. Osluškivači (eng. **listener)** zaduženi su da detektuju određene događaje koji su se dogodili na grafičkom interfejsu. Osluškivači rade u pozadini tako da ne utiču na izvršavanje ostatka programa. U zavisnosti od tipa događaja koji treba detektovati razlikuju se i osluškivači. Svaki od osluškivača implementira neki od interfejsa koji reaguju na različite tipove događaja (miš, tastatura, događaj prozora itd.). Svi ovi interfejsi proširuju **EventListener** interfejs. Unutar ovih interfejsa nalaze se metode koje klasa osluškivač mora da implementira. Implementacijom ovih metoda definišu se scenariji koji će se izvršiti nakon detekcije akcija koje su se dogodile. Osluškivači se mogu realizovati na pet načina:

1. Upotrebom zasebne klase
2. Upotrebom unutrašnje klase
3. Upotrebom anonimne klase
4. Definicijom klase glavnog prozora kao osluškivač
5. Upotrebom adapter klase

U sledećem delu biće objašnjen postupak kreiranja osluškivača za glavni prozor na svaki od gore navedenih načina. Pre toga treba napomenuti da klasa glavnog prozora (**Frame)** generiše događaje klase **WindowEvent.** Ovo je događaj niskog nivoa koje izazivaju **Window** objekti kada se: otvara, zatvara, menja fokus itd. Kada se dogode, događaje detektuje **WindowListener** osluškivač te ga je tako potrebno realizovati. U ovom primeru biće prikazan postupak definicije metoda koje reaguju kada se prozor otvori i kada se detektuje komanda za zatvaranje. Nakon ove realizacije glavni prozor moći će da se zatvara na podrazumevani način.

Prvi način - Realizacija upotrebom zasebne klase.

Sadržaj **OsluskivacProzora.java** datoteke:

```
package grafika;

import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

public class OsluskivacProzora implements WindowListener  {
    @Override
    public void windowOpened(WindowEvent e) {
        System.out.println("Prozor otvoren!!");
    }

    @Override
    public void windowClosing(WindowEvent e) {
        System.out.println("Detektovan zahtev za zatvaranje!");

        System.exit(1);// naredba za prekidanje izvrsavanja programa. Prosledjeni broj
se odnosi na stasut zatvaranja

    }
    @Override
```

```
    public void windowClosed(WindowEvent e) {
    }

    @Override
    public void windowIconified(WindowEvent e) {
    }

    @Override
    public void windowDeiconified(WindowEvent e) {
    }

    @Override
    public void windowActivated(WindowEvent e) {
    }

    @Override
    public void windowDeactivated(WindowEvent e) {
    }
}
```

Sadržaj **Prozor.java** datoteke:

```
package grafika;

import java.awt.*;

public class Prozor extends Frame {
    public Prozor (){
        setTitle("Primer osluskivaca");
        setSize(500,500);
        OsluskivacProzora op=new OsluskivacProzora();//kreiranje objekta osluskivaca
        addWindowListener(op);//dodavanje osluskivaca na prozor
        setVisible(true);
    }
}
```

Sadržaj **Program.java** datoteke:

```
package glavni;
import grafika.Prozor;

public class Program {
    public static void main(String[] args) {
        Prozor p=new Prozor();
    }
}
```

Konzolni ispis nakon pokretanja aplikacije i pritiska tastera za zatvaranje.

```
Prozor otvoren!!
Detektovan zahtev za zatvaranje!
```

Drugi način - Realizacija upotrebom unutrašnje klase

Sadržaj **Prozor.java** datotke:

```
package grafika;

import java.awt.*;
```

```
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

public class Prozor extends Frame {
    public Prozor (){
        setTitle("Primer osluskivaca");
        setSize(500,500);
        OsluskivacProzora op=new OsluskivacProzora();//kreiranje objekta osluskivaca
        OsluskivacKaoUnutrasnjaKlasa okuk=new OsluskivacKaoUnutrasnjaKlasa();
        addWindowListener(okuk);//dodavanje osluskivaca na prozor
        setVisible(true);
    }

    private class OsluskivacKaoUnutrasnjaKlasa implements WindowListener{
        @Override
        public void windowOpened(WindowEvent e) {
            System.out.println("Prozor otvoren!!");
        }

        @Override
        public void windowClosing(WindowEvent e) {
            System.out.println("Detektovan zahtev za zatvaranje!");
            System.exit(1);// naredba za prekidanje izvrsavanja programa. Prosledjeni
broj se odnosi na stasut zatvaranja
        }

        @Override
        public void windowClosed(WindowEvent e) {
        }

        @Override
        public void windowIconified(WindowEvent e) {
        }

        @Override
        public void windowDeiconified(WindowEvent e) {
        }

        @Override
        public void windowActivated(WindowEvent e) {
        }

        @Override
        public void windowDeactivated(WindowEvent e) {
        }
    }
}
```

Sadržaj **Program.java** datoteke i konzolni ispis isti je kao u prvom primeru.

Treći način - Realizacija upotrebom anonimne klase

Anonimne klase su nalik lokalnim klasama s tim da nemaju ime. Omogućuju deklaraciju i inicijalizaciju u istom trenutku. Upotrebom ovih klasa kod postaje koncizan i preporuka je da se koriste kada je potrebna lokalna klasa koju treba instancirati samo jednom.

Osnovni oblik anonimne klase:

```
// Test moze biti interfejs ili apstraktna klasa
  Test t = new Test()
```

```
  {
      // polja i metode klase
      public void testMetoda()
      {
        // definicija tela metode
      }
  };
```

Sadržaj **Prozor.java** daoteke:

```
package grafika;

import java.awt.*;
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

public class Prozor extends Frame {
    public Prozor (){
        setTitle("Primer osluskivaca");
        setSize(500,500);

        addWindowListener(new WindowListener() {
            @Override
            public void windowOpened(WindowEvent e) {
                System.out.println("Prozor otvoren!!");
            }
            @Override
            public void windowClosing(WindowEvent e) {
                System.out.println("Detektovan zahtev za zatvaranje!");
                System.exit(1);// naredba za prekidanje izvrsavanja programa.
Prosledjeni broj se odnosi na stasut zatvaranja
            }

            @Override
            public void windowClosed(WindowEvent e) {
            }
            @Override
            public void windowIconified(WindowEvent e) {
            }
            @Override
            public void windowDeiconified(WindowEvent e) {
            }
            @Override
            public void windowActivated(WindowEvent e) {
            }
            @Override

            public void windowDeactivated(WindowEvent e) {

            }
        });//dodavanje osluskivaca na prozor
        setVisible(true);
    }

}
```

Sadržaj **Program.java** datoteke i konzolni ispis isti je kao u prvom primeru.

Četvrti način - Realizacija definicijom klase glavnog prozora kao osluškivač

Sadržaj **Prozor.java** datoteke:

```
package grafika;
import java.awt.*;
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

public class Prozor extends Frame implements WindowListener {
    public Prozor (){
        setTitle("Primer osluskivaca");
        setSize(500,500);

        addWindowListener(this);//dodavanje osluskivaca na prozor
        setVisible(true);
    }
    @Override
    public void windowOpened(WindowEvent e) {
        System.out.println("Prozor otvoren!!");
    }
    @Override
    public void windowClosing(WindowEvent e) {
        System.out.println("Detektovan zahtev za zatvaranje!");
        System.exit(1);// naredba za prekidanje izvrsavanja programa. Prosledjeni broj
se odnosi na stasut zatvaranja
    }
    @Override
    public void windowClosed(WindowEvent e) {
    }
    @Override
    public void windowIconified(WindowEvent e) {
    }
    @Override
    public void windowDeiconified(WindowEvent e) {
    }
    @Override
    public void windowActivated(WindowEvent e) {
    }
    @Override
    public void windowDeactivated(WindowEvent e) {
    }
}
```

Sadržaj **Program.java** datoteke i konzolni ispis isti je kao u prvom primeru.

Peti način - Realizacija upotrebom adapter klase

Kao što se može primetiti, u prethodnim primerima ima dosta nepotrebnog koda jer su korišćene samo dve od sedam metoda koje se nalaze u interfejsu. Da bi se izbegla ovakva situacija koriste se klase adapteri. Adapter klase su zapravo apstraktne klase koje služe da se pomoću njih implementiraju samo određene metode nekog interfejsa. Metode interfejsa definisane su kao apstraktne unutar adapter klase tako da je moguće preklopiti samo određene u klasi koja je proširuje. Unutar **AWT** biblioteke postoje već definisane klase adapteri između ostalog i **WindowAdapter** klasa. Kako se radi o apstraktnoj moguće je kreirati anonimnu klasu pomoću nje.

Sadržaj **Prozor.java** datoteke**:**

```
package grafika;
import java.awt.*;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
public class Prozor extends Frame {
```

```
    public Prozor (){
        setTitle("Primer osluskivaca");
        setSize(500,500);
        addWindowListener(new WindowAdapter() {
            @Override
            public void windowOpened(WindowEvent e) {
                System.out.println("Prozor otvoren!!");
            }
            @Override
            public void windowClosing(WindowEvent e) {
                System.out.println("Detektovan zahtev za zatvaranje!");
                System.exit(1);// naredba za prekidanje izvrsavanja programa.
Prosledjeni broj se odnosi na stasut zatvaranja
            }
        });//dodavanje osluskivaca na prozor

        setVisible(true);
    }
}
```

Sadržaj **Program.java** datoteke i konzolni ispis isti je kao u prvom primeru.

U situacijama kada se od korisnika traži potvrda za neku operaciju ili treba da mu se prikaže neko obaveštenje koriste se prozori za dijalog. Dijalozi su posebni podprozori koji deluju nezavisno od glavnog prozora. Pomoću njih korisniku se može prikazati bitno obaveštenje, tražiti potvrda, tražiti unos određene vrednosti itd. U sledećem primeru biće prikazan postupak pravljenja i korišćenja dijaloga kao i upotreba **MouseListener** osluškivača**.**

Sadržaj **Prozor.java** datoteke:

```
package grafika;
import java.awt.*;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;

import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

public class Prozor extends Frame {
    int x;
    int y;

    public Prozor() {
        x = 0;
        y = 0;
        setTitle("Laboratorijska vezba 8");
        setSize(300, 500);
        //kreiranje dialoga, prvi parametar se odnosi na roditeljski prozor
        ProzorZaDijalog dijalog = new ProzorZaDijalog(this, "Da li zelite da izadjete
iz aplikacije?");
        addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent e) {
                dijalog.prikazi();//prikazivanje dijaloga
            }
        });
        //postavljanje osluskivac dogadjaja prouzrokovanim misem
        addMouseListener(new MouseAdapter() {
            //definisanje scenarija za akciju "klik misa"
            @Override
            public void mouseClicked(MouseEvent e) {
                x = e.getX();//uzimanje pozicije kursora po x osi
                y = e.getY();//uzimanje pozicije kursora po y osi
```

```
                repaint();//eksplicitni poziv paint metode kako bi se osvezio prikaz
            }
        });
        setVisible(true);
    }
    @Override
    public void paint(Graphics g) {
        g.drawString("Pozicija(X:" + x + ",Y:" + y + ")", x, y);//crtanje teksta na
pozicji x, y
        g.drawLine(10, 30, x, y); //crtanje linije od pozicije 30,30 do tacke x,y
    }
}
```

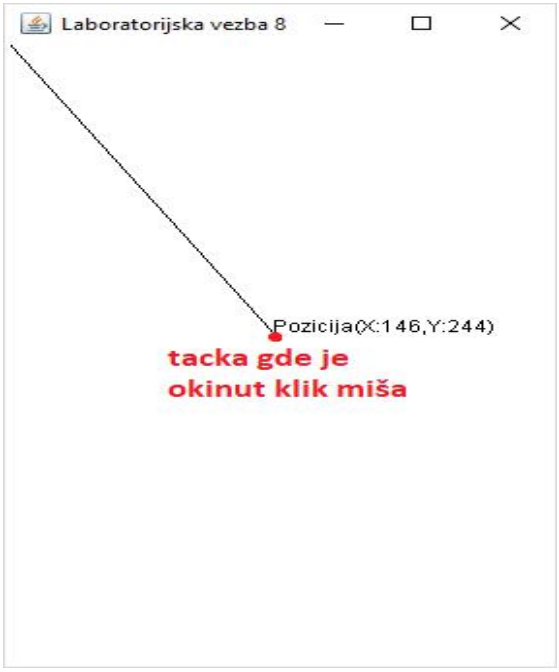
Sadržaj **ProzorZaDijalog.java** datoteke:

```
package grafika;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
public class ProzorZaDijalog extends Dialog {
    public ProzorZaDijalog(Frame owner, String naslov) {
        super(owner);//postavljanje roditeljskog prozora
        setModal(true);//postavljanje modalnosti na true. Uociti razlike kada je
vrednost false
        setSize(300, 100);
        setTitle(naslov);
        addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent e) {

                setVisible(false);
            }

        });
        setLayout(new FlowLayout());

        Button btnDa = new Button("DA");
        btnDa.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                System.exit(1);
            }
        });
        Button btnNe = new Button("NE");
        btnNe.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                setVisible(false);
            }
        });
        add(btnDa);
        add(btnNe);
    }
    public void prikazi() {
        setVisible(true);
    }
}
```

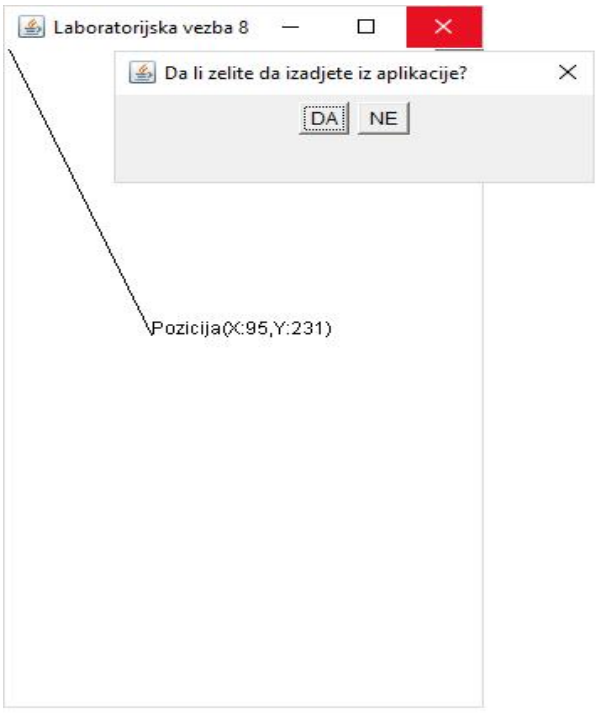
Sadržaj **Program.java** datoteke:

```
package glavni;
import grafika.Prozor;
public class Program {
    public static void main(String[] args) {
        Prozor p=new Prozor();
    }
}
```

Na slikama 8.1 i 8.2 prikazani su različiti scenariji izvršenja programa:



Slika 8.1- Obrada događaja

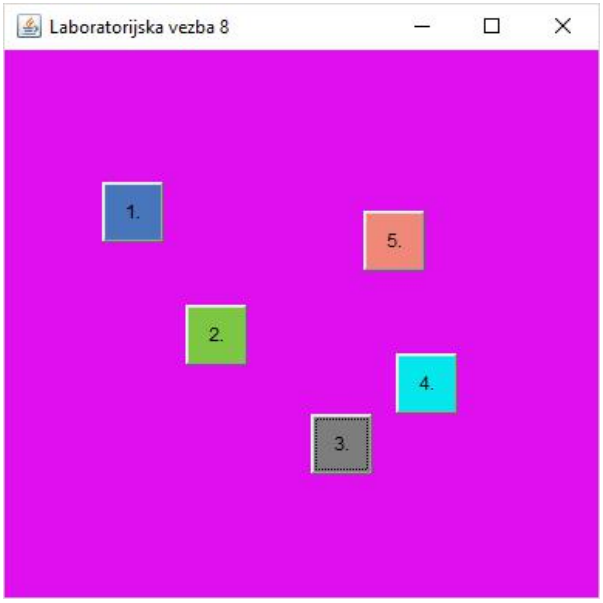


Slika 8.2- Dijalog prilikom zatvaranja aplikacije

Zadatak za samostalan rad.

Kreirati aplikaciju čija će se pozadinska boja prozora menjati u zavisnosti od pozicije kursora miša na prozoru. Prilikom klika miša kreira se dugme na toj poziciji čija pozadinska boja odgovara trenutnoj boji pozadine. Ukoliko se izvrši akcija na dugme, njegova pozadinska boja postaje siva **rgb(125,125,125).** Na dugmicima se ispisuje i njihov redni broj. Primer korisćenja prikazani su na slikama 8.3 i 8.4.

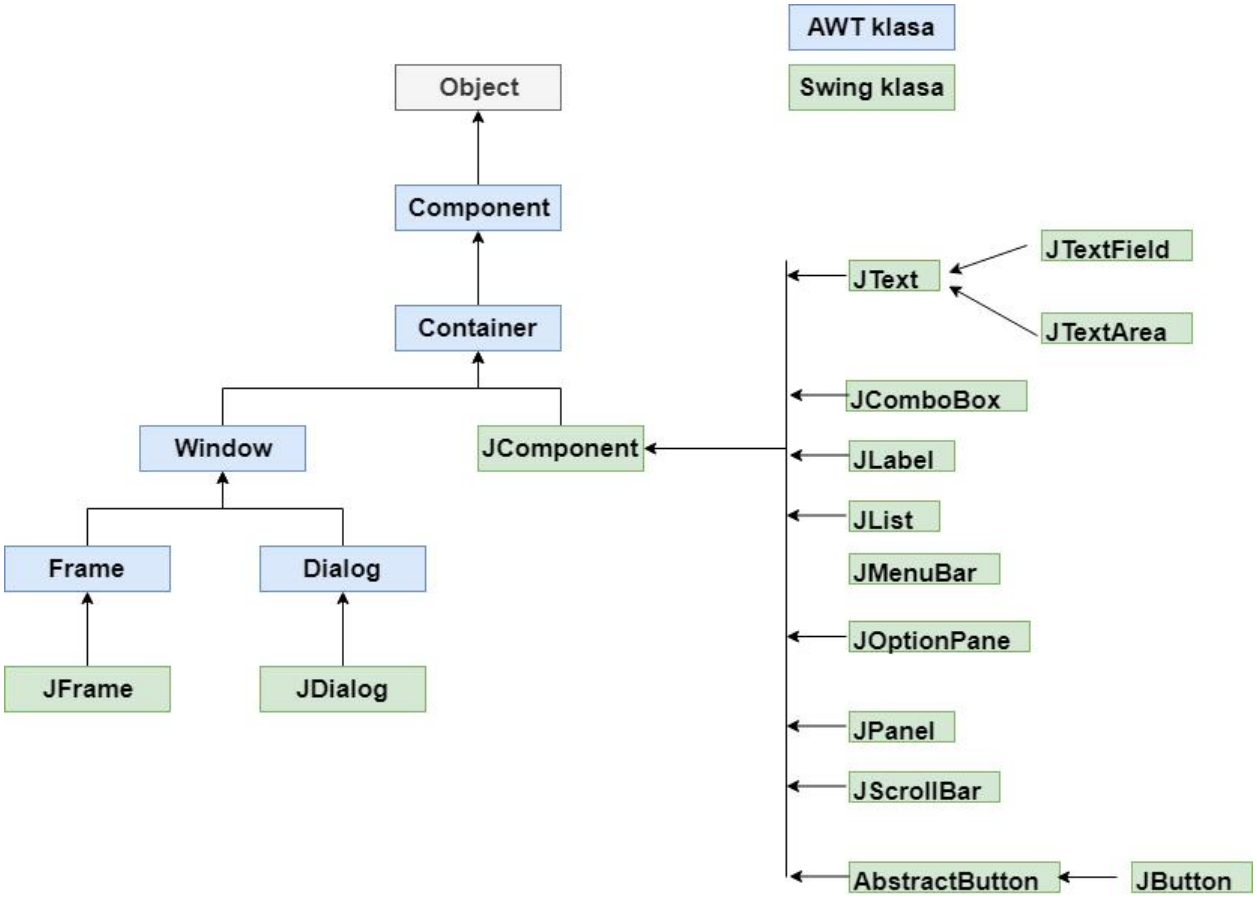
Slika 8.3- Primer korišćenja aplikacije



Slika 8.4 – Izgled prozora nakon izvršene akcije na dugme "3.".

Laboratorijska vežba 9: SWING biblioteka

Swing biblioteka nastala je nadogradnjom na **AWT** biblioteku. Novodobijena biblioteka omogućava kreiranje komponenti boljih performansi čiji je izgled nezavisan od operativnog sistema na kojem se aplikacija izvršava. Swing je zadržao od prethodnika obradu događaja kao i upotrebu raspoređivača. Obično komponente u ovoj biblioteci imaju prefiks '**J**'. Osim toga, dodate su i nove komponete koje nisu postojale u **AWT** biblioteci (tabele, liste, stabla itd). Biblioteka se nalazi okviru **java.desktop** modula. Na slici 9.1 prikazana je hijerarhija bitnijih klasa u **Swing** biblioteci.



Slika 9.1- Hijerarhija bitnijih klasa

Kao što se može primetiti na dijagramu, **JFrame** proširuje klasu **Frame** što zapravo znači da se glavni prozor kreira pomoću ove klase. Postupak kreiranja i upravljanja komponentama je sličan kao u **AWT**. U sledećem delu prikazan je primer korišćenja raspoređivača koji je prikazan u sedmoj vežbi s tim da će sada biti korišćena **Swing** biblioteka.

Sadržaj **Prozor.java** datoteke:

```
package grafika;
import javax.swing.*;
import java.awt.*;
public class Prozor extends JFrame {
    String nazivkarte[] = {"prva karta", "druga karta",
            "treca karta", "cetvrta karta", "peta karta"};
    public Prozor() {
```

```
        super("Raspored"); //poziv konstruktora natklase (JFrame). Prosledjeni
parametar se odnosi na naslov prozora.
        setLayout(new GridLayout(3, 2));
        setSize(500, 700);
         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);//postavljanje podrazumevane
operacije za zatvaranje.
        dodajBorderPanel();
        dodajFlowPanel();
        dodajCardPanel();
        dodajGridPanel();
        dodajGridBagPanel();
        dodajNullPanel();
        setVisible(true);
    }
    private void dodajBorderPanel() {
        JPanel borderP = new JPanel();
        borderP.setLayout(new BorderLayout());
        borderP.add("North", new JLabel("Pozicija North"));//dodavanje komponente
(JLabel) na poziciju 'Noth'
        borderP.add("East", new JLabel("Pozicija East"));
        borderP.add("West", new JLabel("Pozicija West"));
        borderP.add("South", new JLabel("Pozicija South"));
        borderP.add("Center", new JLabel("Pozicija Center"));
        borderP.setBackground(new Color(66, 134, 244));//postavljanje pozadinske boje
        add(borderP);
    }
    private void dodajCardPanel() {
        JPanel cardPanel = new JPanel();
        CardLayout cardlayout = new CardLayout();
        cardPanel.setLayout(cardlayout);
        for (int i = 0; i < nazivkarte.length; i++) {
            JButton b = new JButton(nazivkarte[i]);
            b.setBackground(new Color(244, 229, 65));
            cardPanel.add(nazivkarte[i], b);
        }
        cardlayout.show(cardPanel, nazivkarte[1]);//menjati indeks u nizu i videti
rezultat izvrsenja
        add(cardPanel);
    }
    private void dodajGridBagPanel() {
        JPanel gridBagPanel = new JPanel();
        gridBagPanel.setLayout(new GridBagLayout());
        GridBagConstraints pozicija = new GridBagConstraints();
        pozicija.weightx = 0.5;
        pozicija.fill = GridBagConstraints.HORIZONTAL;
        pozicija.ipadx = 0;//postavljanje vrednosti za indeks kolone
        pozicija.gridy = 0;//postavljanje vrednosti za indeks reda
        gridBagPanel.add(new JLabel("Ime:"), pozicija);
        pozicija.ipadx = 1;
        pozicija.ipady = 0;
        gridBagPanel.add(new JTextField(), pozicija);
        pozicija.ipadx = 0;
        pozicija.gridy = 1;
        gridBagPanel.add(new JLabel("Prezime:"), pozicija);
        pozicija.ipadx = 1;

        pozicija.ipady = 1;
        gridBagPanel.add(new JTextField(), pozicija);
        pozicija.gridy = 2;
        pozicija.gridx = 0;
        pozicija.gridwidth = 2;
        gridBagPanel.add(new JButton("Posalji"), pozicija);
```

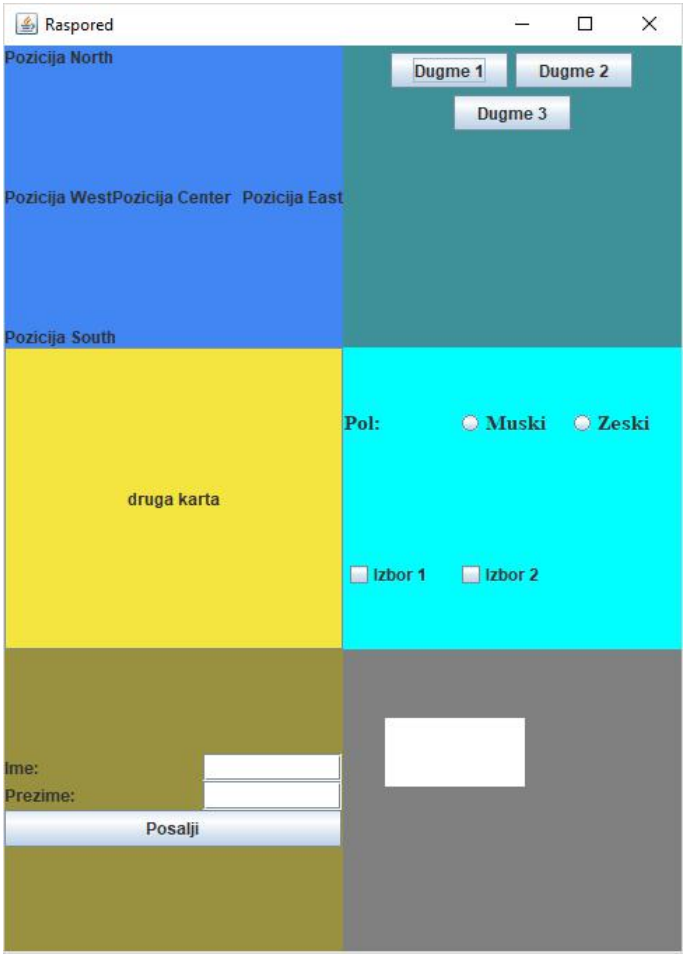
```
        gridBagPanel.setBackground(new Color(153, 145, 64));
        add(gridBagPanel);
    }
    private void dodajFlowPanel() {
        JPanel flowP = new JPanel();
        flowP.setLayout(new FlowLayout());
        flowP.add(new JButton("Dugme 1"));
        flowP.add(new JButton("Dugme 2"));
        flowP.add(new JButton("Dugme 3"));
        flowP.setBackground(new Color(63, 145, 153));
        add(flowP);
    }
    private void dodajGridPanel() {
        JPanel gridP = new JPanel();
        gridP.setLayout(new GridLayout(2, 3));
        JLabel lblPol = new JLabel("Pol:");
        Font f = new Font("Times New Roman", Font.BOLD, 15);//kreiranje fonta
        lblPol.setFont(f);
        JRadioButton rbMuski = new JRadioButton("Muski");
        rbMuski.setFont(f);
        rbMuski.setOpaque(false);//postavljanje neprozirnosti na false. Izostaviti ovu
liniju i videti rezultat izvrsenja.
        JRadioButton rbZenski = new JRadioButton("Zeski");
        rbZenski.setOpaque(false);
        rbZenski.setFont(f);
        ButtonGroup btnGPol = new ButtonGroup();
        btnGPol.add(rbMuski);
        btnGPol.add(rbZenski);
        gridP.add(lblPol);
        gridP.add(rbMuski);
        gridP.add(rbZenski);
        JCheckBox cbIzbor1 = new JCheckBox("Izbor 1");
        cbIzbor1.setOpaque(false);

        JCheckBox cbIzbor2 = new JCheckBox("Izbor 2");
        cbIzbor2.setOpaque(false);
        gridP.add(cbIzbor1);
        gridP.add(cbIzbor2);
        gridP.setBackground(Color.cyan);
        add(gridP);
    }
    private void dodajNullPanel() {
        JPanel nullP = new JPanel();
        nullP.setLayout(null);
        JTextArea taTekst = new JTextArea();
        taTekst.setBounds(30, 50, 100, 50);//definisanje pozicije (30,50) i velicine
komponente (100X50)
        nullP.add(taTekst);
        nullP.setBackground(Color.GRAY);
        add(nullP);
    }
}
```

Sadržaj **Program.java** datoteke:

```
package glavni;

import grafika.Prozor;

public class Program {
    public static void main(String[] args) {
        Prozor p=new Prozor();
```

```
    }
}
```

Izgled rezultujućeg prozora prikazan je na slici 9.2. Primetiti kako se veličina i raspored komponenti menja prilikom promena dimenzija prozora.



Slika 9.2- Primer raspoređivača

U sledećem primeru prikazan je postupak kreiranja menija sa pratećim stavkama, kao i postupak obrade događaja na njima. U primeru su korišćeni i prozori za dijalog čije je kreiranje i rukovanje jednostavnije nego u **AWT**. Za to je zaslužna **JOptionPane** klasa i njene statičke metode **showInputDialog()** (dijalog za unos vrednosti), **showMessageDialog()** (dijalog za prikaz poruke), **showConfirmDialog()** (dijalog za potvrdu). U okviru glavnog kontejnera postavljen je **JTabbedPane** objekat koji nudi mogućnost višestraničnog prikaza. Sadržaj koji se postavlja (obično paneli sa pratećim komponentama) ređa se jedan preko drugog, slično kao i kod **CardLayout**, s tim da postoji deo koji služi za navigaciju.

Sadržaj **PrimerProzor.java** datoteke:

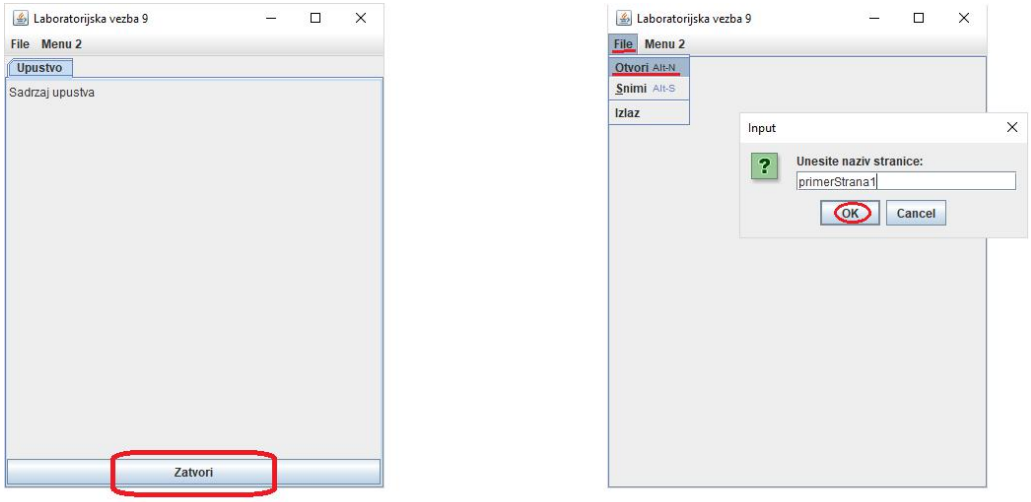
```
package grafika;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class PrimerProzor extends JFrame {
    JTabbedPane tPane;
```

```
    public PrimerProzor() {
        setTitle("Laboratorijska vezba 9");
        setSize(400, 500);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        dodajMenu();
        dodajTabPane();
        setVisible(true);
    }

    private void dodajMenu() {
        JMenuBar menuB = new JMenuBar();//kreiranje meni table na kojoj se postavljaju
meniji. Tabla se postavlja u gornjem delu ispod naslovne linije nezavisno od
rasporedjivaca koji je postavljen
        JMenu menuFile = new JMenu("File");//kreiranje menija sa nazivom "File"
        JMenuItem miOtvori = new JMenuItem("Otvori");//kreiranje stavke menija sa
nazivom "Otvori"
        miOtvori.setAccelerator(KeyStroke.getKeyStroke(
                KeyEvent.VK_N, ActionEvent.ALT_MASK));//postavljanje precice. Precica
je realizovana kombinacijom ALT+N

        JMenuItem miSnimi = new JMenuItem("Snimi", KeyEvent.VK_S);
        miSnimi.setAccelerator(KeyStroke.getKeyStroke(
                KeyEvent.VK_S, ActionEvent.ALT_MASK));

        JMenuItem miIzlaz = new JMenuItem("Izlaz");

        //kreiranje osluskivaca za "File" meni stavke
        ActionListener osluskivacFileMenu = new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                switch (e.getActionCommand()) {
                    case "Otvori":
                        String naziv = JOptionPane.showInputDialog("Unesite naziv
stranice:");//kreiranje djaloga za unos vrednosti
                        if (naziv != null) {
                            JPanel novPanel = new JPanel();
                            novPanel.add(new JLabel("Sadrzaj panela"));
                            tPane.add(naziv, novPanel);
                        }
                        break;
                    case "Snimi":

                        JOptionPane.showMessageDialog(null, "Uspesno ste snimili
podesavanja");//kreiranje dijaloga za prikaz obavestenja

                        break;
                    case "Izlaz":
                        //kreirnaje dijaloga za potvrdu
                        int rez = JOptionPane.showConfirmDialog(null, "Da li sigurno
zelite da napustite aplikaciju?", "Izlaz", JOptionPane.YES_NO_OPTION);
                        if (rez == 0)
                            System.exit(1);
                        break;
                }
            }
        };
        miOtvori.addActionListener(osluskivacFileMenu);//postavljanje osluskivaca na
stavku menija
        miSnimi.addActionListener(osluskivacFileMenu);
        miIzlaz.addActionListener(osluskivacFileMenu);

        menuFile.add(miOtvori);//dodavanje stavke u meniju "File"
```

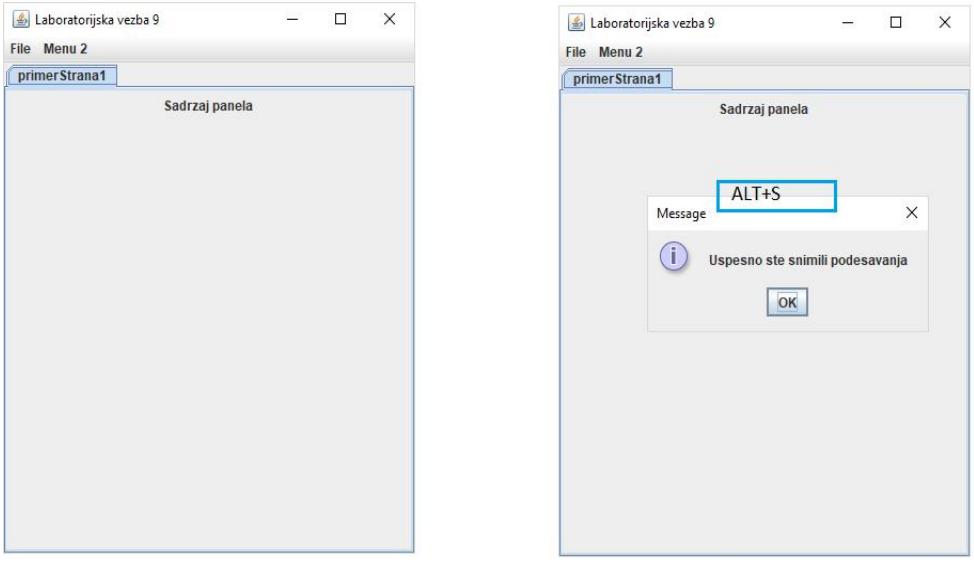
```
        menuFile.add(miSnimi);
        menuFile.addSeparator();//dodavanja horizontalne linije
        menuFile.add(miIzlaz);

        //kreirnaje drugog menija
        JMenu meni2 = new JMenu("Menu 2");
        JRadioButtonMenuItem opcija1 = new JRadioButtonMenuItem("opcija1");//dodavanje
stavke
        opcija1.addItemListener(new ItemListener() {
            @Override
            public void itemStateChanged(ItemEvent e) {
                JOptionPane.showMessageDialog(null, "Opcija1:" + opcija1.isSelected());
            }
        });
        meni2.add(opcija1);
        menuB.add(menuFile);//dodavanje menija na meni baru
        menuB.add(meni2);
        setJMenuBar(menuB);//postavljanje meni vara na prozor
    }

    private void dodajTabPane() {
        tPane = new JTabbedPane();//instanciranje tabedpane-a
        JPanel panelPrvi = new JPanel();//kreiranje panela
        panelPrvi.setLayout(new BorderLayout());
        panelPrvi.add("North", new Label("Sadrzaj upustva"));//dodavanje sadrzaja na
panel
        JButton btnZatvori = new JButton("Zatvori");
        btnZatvori.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                tPane.remove(tPane.getSelectedIndex());//brisanje trenutno izabrane
strane na tabbedpan-u
            }
        });
        panelPrvi.add("South", btnZatvori);
        tPane.add("Upustvo", panelPrvi);//dodavanje prve strane
        add("Center", tPane);//postavljanje tabedspane na prozor
    }
}
```

Sadržaj **Program.java** datoteke:

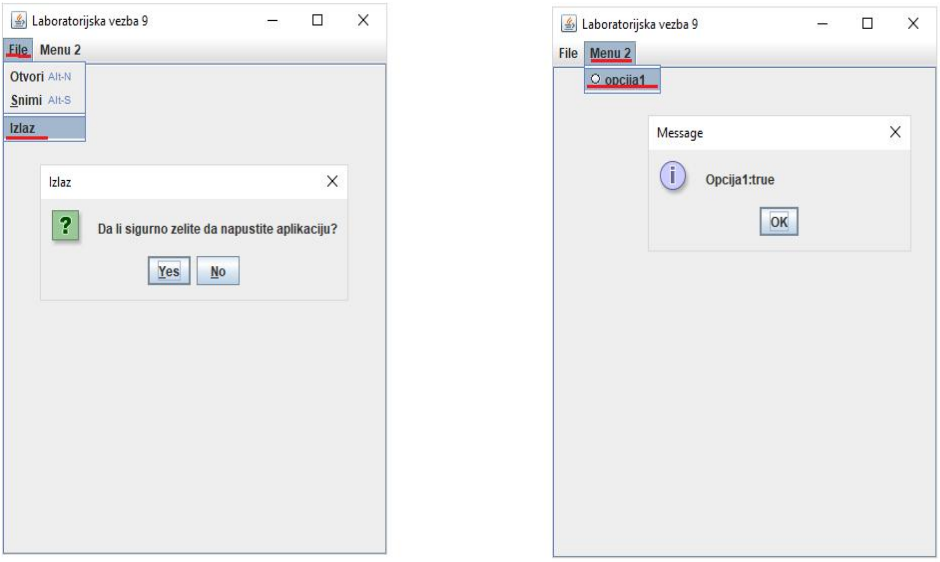
```
package glavni;

import grafika.PrimerProzor;

public class Program {
    public static void main(String[] args) {
        PrimerProzor pp=new PrimerProzor();
    }
}
```



Slika 9.3- Zatvaranje stranice pane, dijalog za unos



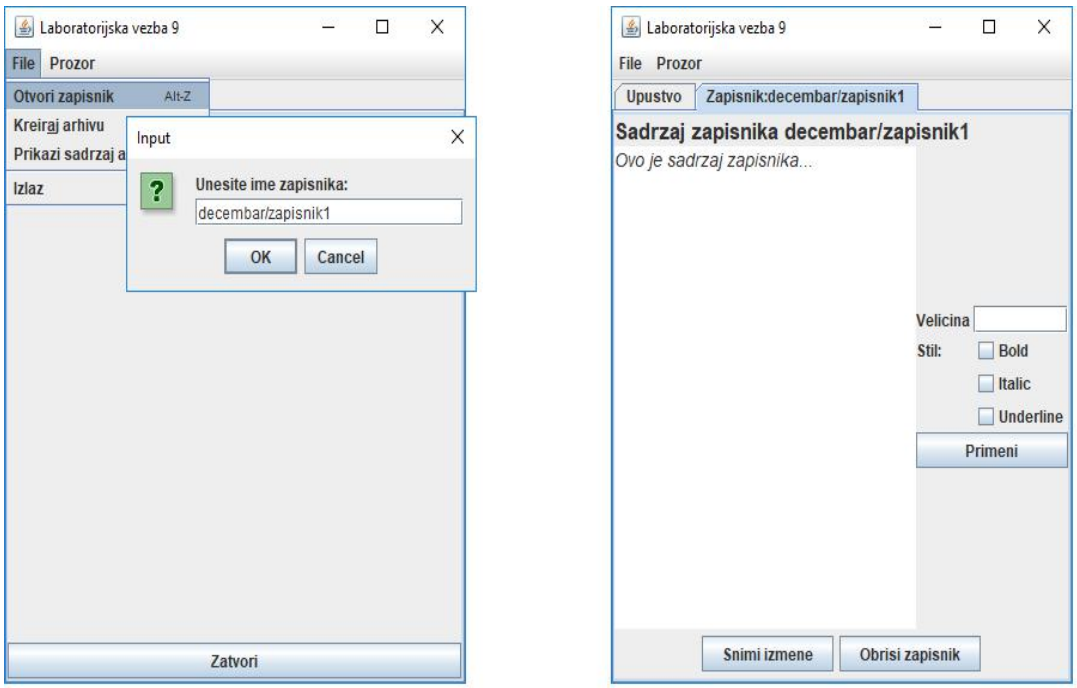
Slika 9.4- Kreiranje nove stranice, poziv stavke menija pomoću prečice



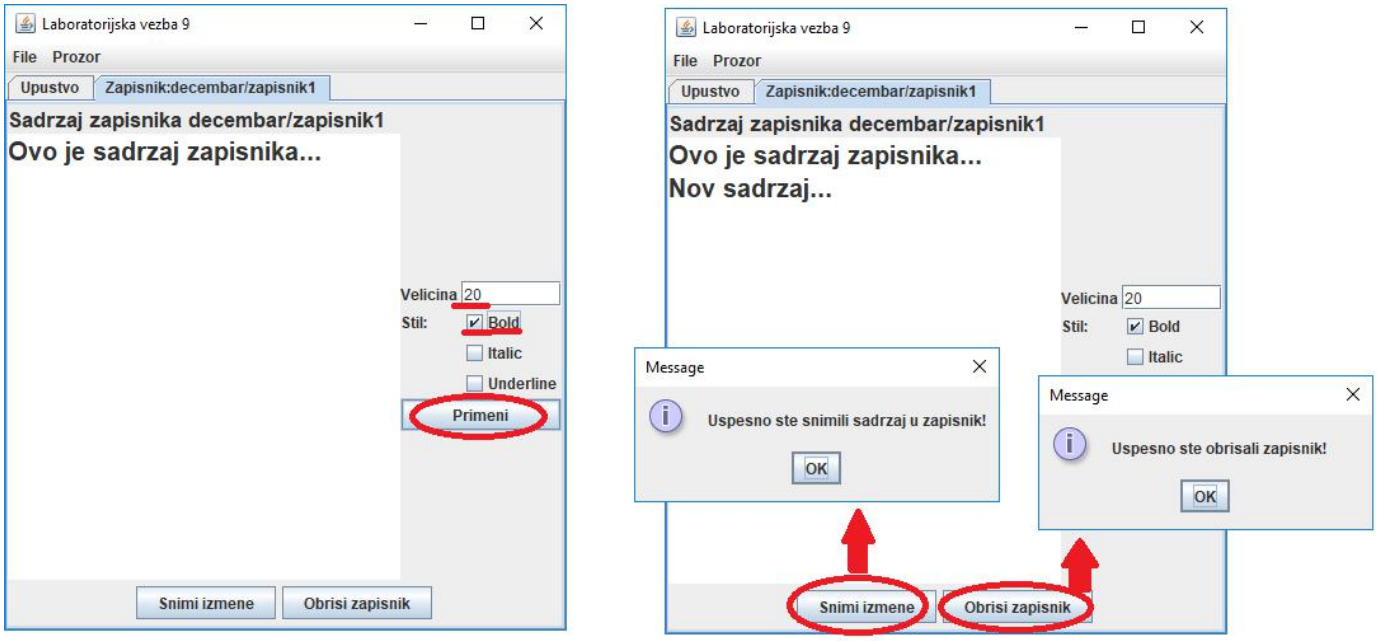
Slika 9.5- Dijalog za potvrdu, odabir stavke iz menija „Menu 2“

Zadatak za samostalan rad:

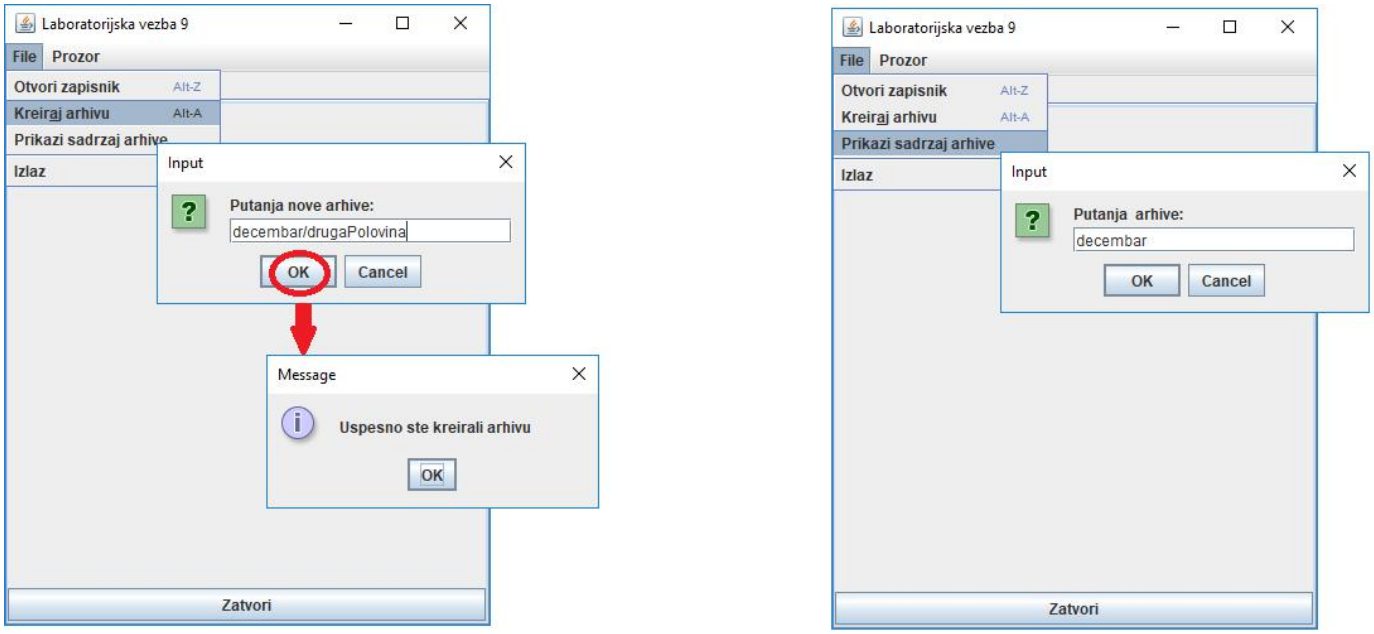
Modifikovati prethodni primer i kreirati grafički korisnički interfejs za aplikaciju koja je rađena na trećoj vežbi (aplikacija za uređivanje zapisnika). Interfejs treba da sadrži funkcionalnost kako si se izvršile komande i prikazali rezultati izvršenja. Na slikama 9.6 – 9.10 prikazan je izgled interfejsa kao i različiti scenariji upotrebe.



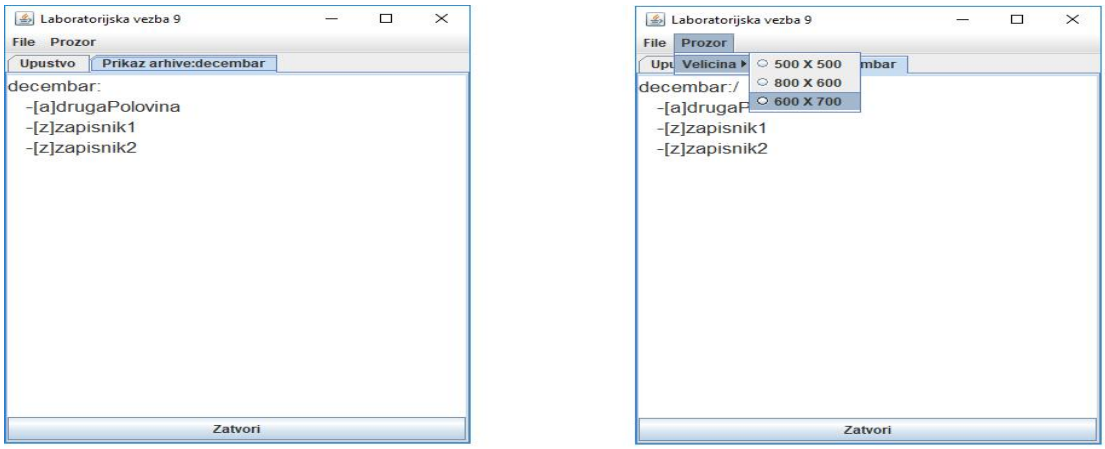
Slika 9.6- Otvaranje/kreiranje zapisnika



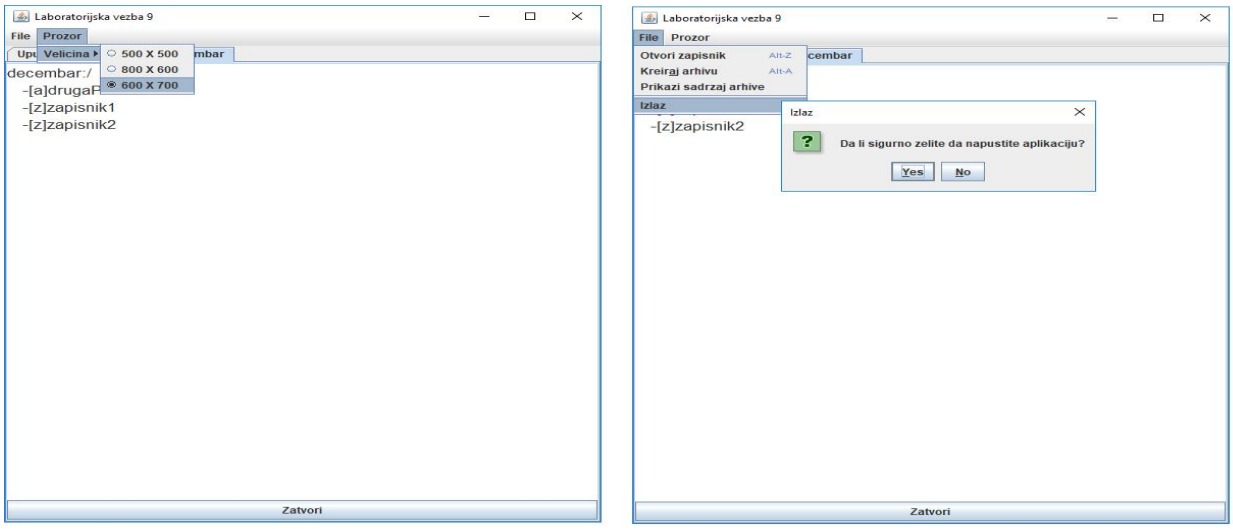
Slika 9.7 – Snimanje/zapisnika, uređivanje prikaza



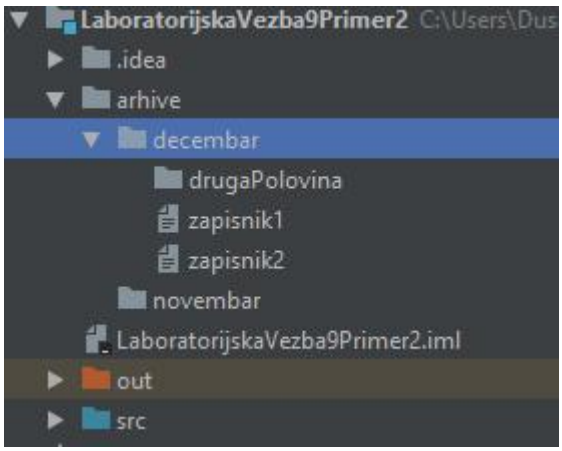
Slika 9.8- Kreiranje arhive, prikaz arhive



Slika 9.9- Prikaz arhive, menjanje veličine prozora



Slika 9.10- Promena veličine prozora, zatvaranje aplikacije

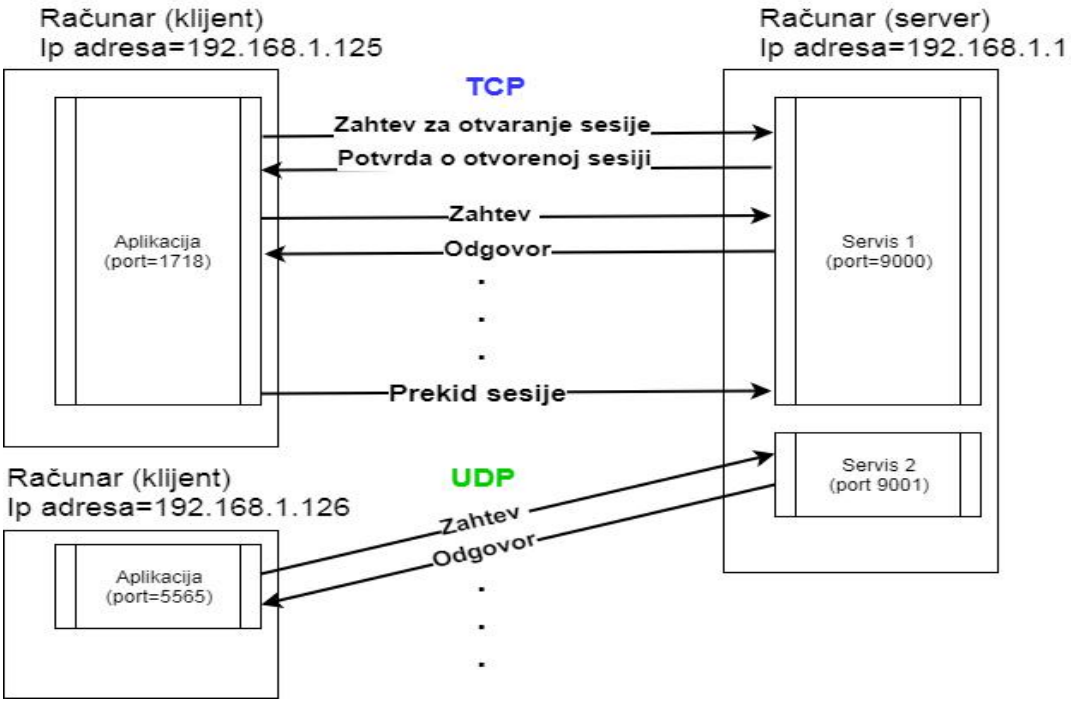


Slika 9.11- Struktura projekta

Laboratorijska vežba 10: Klijent-server arhitektura, mrežno programiranje

Klijent-server arhitektura definiše komunikaciju između dva računara (aplikacije) u kojoj jedna strana nudi mogućnost korišćenja servisa (usluga) koji su potrebni drugoj strani. U osnovi, ovakva arhitektura se sastoji od dva činioca: klijenta i servera. Server deo zadužen je da prihvati zahteve od klijenta i da ih obradi gde se neretko rezultat izvršenja vraća klijentu. Obično je klijent taj koji inicira komunikaciju mada nije isključena mogućnost obrnutog scenarija. Korišćenjem mrežnih adaptera može se realizovati ovakva komunikacija. Komunikacija se realizuje korišćenjem mrežnih protokola iz četvrtog nivoa **OSI** referentnog sistema. U zavisnosti da li se obavlja komunikacija sa ili bez uspostave sesije koristiće se **TCP** odnosno **UDP** protokol. Komunikacija sa uspostavom sesije osigurava da je poslati mrežni paket stigao do odredišta, proverava da li je došlo do greške prilikom prenosa itd. Sve ovo se odražava na brzinu prenosa tako da je preporuka da se u situaciji gde je potrebno realizovati brzu komunikaciju, kao što je prenos sadržaja u realnom vremenu (slike, zvuka itd.) koristi **UDP** protokol. **TCP** komunikaciju treba koristiti u svim ostalim situacijama pošto nudi gore navedene sigurnosne provere. Komunikacija se obično svodi na niz parova zahtev/odgovor poruka. Zahteve šalje klijent, a odgovore server.

Da bi se komunikacija uspostavila obe strane moraju znati kako je druga identifikovana na mreži. Identifikacija se vrši pomoću **IP** (Internet protocol) adrese koja mora biti jedinstvena u okviru lokalne/globalne mreže. Osim IP adrese potrebno je identifikovati i proces na računaru u kojem se izvršava aplikacija.



Slika 10.1- Tok komunikacije korišćenjem TCP i UDP protokla

Razlog tome je to što se na računaru izvršava veći broj aplikacija te ih je potrebno identifikovati kako bi se znalo za koju aplikaciju je namenjen pristigli paket. Uređeni par IP adrese i porta naziva se soket (eng. socket). Servis podignut na serveru mora imati fiksni port

kako bi klijent znao kome treba da se obrati. Raspon vrednosti portova kreće se od 0 – 65535 s tim da su portovi od 0 – 1023 rezervisani pa ih ne treba koristiti. Port na klijentskim aplikacijama dodeljuje operativni sistem po pokretanju i nije uvek isti. Server na osnovu zaglavlja iz pristiglog paketa zaključuje kojoj adresi i na kom portu treba da pošalje odgovor. Na slici 10.1 prikazan je tok komunikacije između servera i dva klijenta. Klijenti se obraćaju istom serveru, ali različitim servisima koristeći različite protokole (TCP/UDP).

Kreiranje klijentskog dela

Klasa **Socket** iz paketa **java.net** (modul **java.base**) omogućava otvaranje **TCP** sesije između klijenta i servera. Konstruktoru ove klase potrebno je proslediti IP adresu (objekat klase **InetAddress**) i celobrojnu vrednost porta serverske strane. Kreiranjem ovog objekta, klijent zahteva otvaranje sesije sa serverom. Ukoliko na mreži ne postoji server i servis na njemu sa datom IP adresom odnosno portom, podiže se **IOException** izuzetak pri instaciranju. Nakon otvaranja sesije klijent može krenuti sa slanjem zahteva. Slanje zahteva i prijem odgovora može se realizovati pomoću **PrintWriter** i **BufferedReader** klasa. Postupak slanja i prijema sličan je postupku upisa i čitanja sadržaja iz tekstualne datoteke s tim da je ovde potrebno koristiti strim dobijenog od otvorenog soket objekta. Uzimanje ulaznog i izlaznog strima vrši se pomoću metoda **getOutputStream()** odnosno **getInputStream().** Po završetku razmene poruka, klijent zatvara komunikaciju (sesiju) zatvaranjem ulazno izlaznih strimova i soket objekta (poziv metode **close()**).

Kreiranje serverskog dela

Da bi klijent mogao da otvori sesiju, na serverskoj strani treba podignuti servis koji će čekati zahtev na tom portu. Klasa **ServerSocket** (paket **java.net,** modul **java.base**) omogućava kreiranja "osluškivača" koji će vršiti obradu zahteva za otvaranje **TCP** sesija. Konstruktoru ove klase potrebno je proslediti port na kome će se čekati zahtevi. Ukoliko je traženi port zauzet, podići će se **IOException** izuzetak. Nakon uspešnog instanciranja, potrebno je pozvati metodu **accept()** koja prihvata zahteve za otvaranje sesije. Metoda blokira dalje izvršenje programa sve do prijema zahteva. Po prijemu, vraća objekat klase **Socket** koji predstavlja vezu (sesiju) sa klijentom. Ovaj objekat osim IP adrese poseduje i port klijenta. Čitanje zahteva i slanje odgovora pomoću **PrintWriter** i **BufferedReader** klasa na isti način kao i kod klijenta.

U sledećem delu prikazan je primer razmena poruka između klijenta i servera korišćenjem **TCP** protokola. Server klijentu treba da vrati sadržaj koji je dobio u obrnutom redosledu. U okviru jednog projekta biće kreirane obe strane i izvršavaće se na istom računaru. To zapravo znači da će se za IP adresu koristiti **loopback** adresa (127.0.0.1). Voditi računa da je server pokrenut pre klijenta. Komunikacija se obavlja sve dok klijent ne unese poruku "kraj".

Sadržaj **ServerProgram.java** datoteke:

```
package server;

import java.io.*;
import java.net.ServerSocket;
```

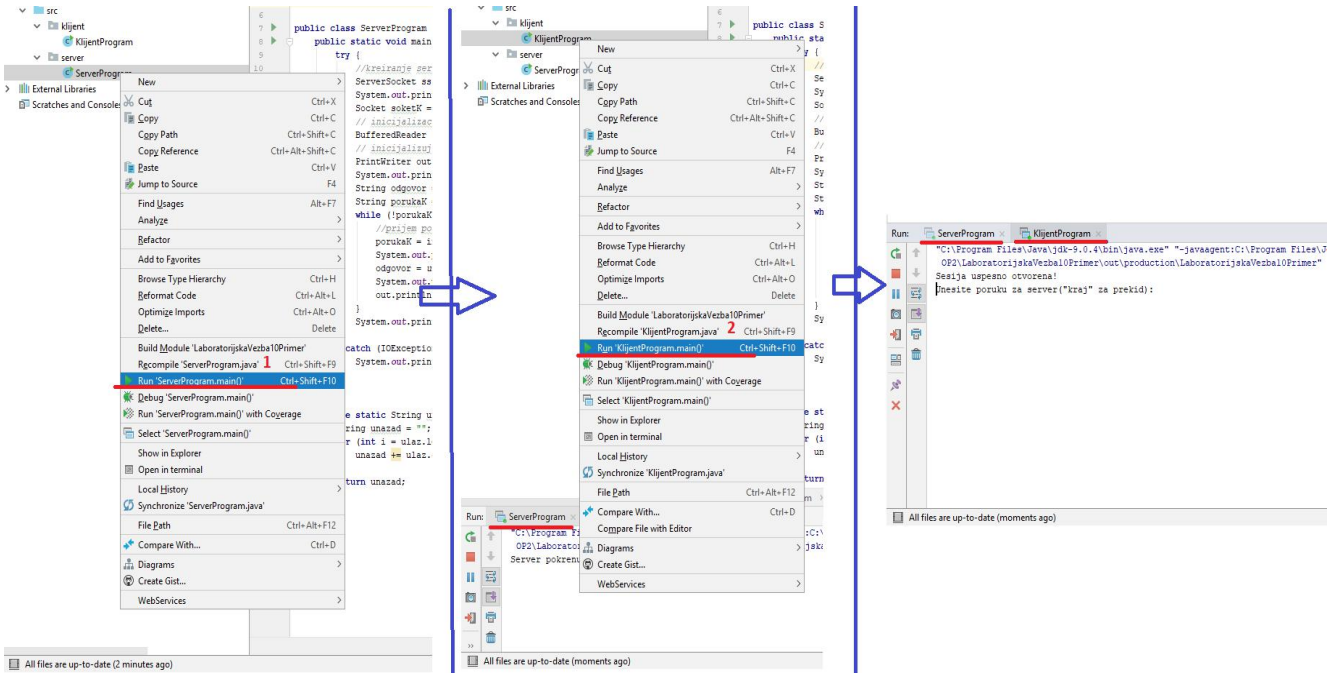
```
import java.net.Socket;

public class ServerProgram {
    public static void main(String[] args) {
        try {
            //kreiranje servisa koji ceka zahtev na portu 9000
            ServerSocket ss = new ServerSocket(9000);
            System.out.println("Server pokrenut, ceka na zahtev za otvaranje
sesije...");
            Socket soketK = ss.accept();
            // inicijalizacija ulaznog streama
            BufferedReader in = new BufferedReader(new
InputStreamReader(soketK.getInputStream()));
            // inicijalizuj izlazni stream
            PrintWriter out = new PrintWriter(new BufferedWriter(new
OutputStreamWriter(soketK.getOutputStream())), true);
            System.out.println("Server otvorio sesiju (port
klijenta:"+soketK.getPort()+"), ceka na zahtev klijenta...");
            String odgovor = "";
            String porukaK = "";
            while (!porukaK.equals("kraj")) {
                //prijem poruke od klijenta
                porukaK = in.readLine();
                System.out.println("Server primio poruku od klijenta:" + porukaK);
                odgovor = unazadRec(porukaK);
                System.out.println("Server salje odgovor:" + odgovor);
                out.println(odgovor);
            }
            System.out.println("Server zavrsio sa radom...");

        } catch (IOException e) {
            System.out.println("Greska prilikom podizanja servisa.");
        }
    }

    private static String unazadRec(String ulaz) {
        String unazad = "";
        for (int i = ulaz.length() - 1; i > -1; i--) {
            unazad += ulaz.charAt(i);
        }
        return unazad;
    }
}
```

Sadržaj **KlijentProgram.java** datoteke:

```
package klijent;


import java.io.*;
import java.net.InetAddress;
import java.net.Socket;


import java.net.UnknownHostException;
import java.util.Scanner;

public class KlijentProgram {
    public static void main(String[] args) {
        try {
            InetAddress ipAdresaServera =
InetAddress.getByName("127.0.0.1");//prosledjeni string treba da odgovara vrednosti ip
```

```
adrese
            //InetAddress ipAdresaServera2=InetAddress.getLocalHost();//drugi nacin za
uzimanje lokalne ip adrese
            //InetAddress ipAdresaServer =
InetAddress.getByName("www.viser.edu.rs");//moze i na ovaj nacin,navodjenjem URL adrese

            int portServisa = 9000;
            Socket soket = new Socket(ipAdresaServera, portServisa);//otvaranje sesije
sa serverom
            System.out.println("Sesija uspesno otvorena!");


            // inicijalizacija ulaznog streama


            BufferedReader in = new BufferedReader(new
InputStreamReader(soket.getInputStream()));
            // inicijalizuj izlazni stream


            PrintWriter out = new PrintWriter(new BufferedWriter(new
OutputStreamWriter(soket.getOutputStream())), true);
            Scanner unos = new Scanner(System.in);
            String poruka = "";
            //slane poruka i primanje odgovora od servera sve dok korisnik ne unese
sadrzaj poruke "kraj"
            while (!poruka.equals("kraj")) {


                System.out.print("Unesite poruku za server(\"kraj\" za prekid):");
                poruka = unos.nextLine();
                System.out.println("Klijent salje poruku:" + poruka);
                out.println(poruka);
                System.out.println("Ceka se odgovor servera...");
                String odgovor = in.readLine();
                System.out.println("Odgovor servera:" + odgovor);
            }
            in.close();
            out.close();


            soket.close();
            System.out.println("Sesija i strimovi zatvoreni");

        } catch (UnknownHostException e) {
            System.out.println("Desila se greska prilikom kreiranja ip adrese.");
        } catch (IOException e) {
            System.out.println("Doslo je do greske prilikom otvaranja sesije");
        }
    }

}
```

Na slici 10.2 prikaz je postupak pokretanja.



Slika 10.2 – Postupak pokretanja server/klijent komunikacije

Konzolni ispis u konzoli servera:

```
Server pokrenut, ceka na zahtev za otvaranje sesije...
Server otvorio sesiju (port klijenta:58559), ceka na zahtev klijenta...
Server primio poruku od klijenta: Pozdrav
Server salje odgovor: vardzoP
Server primio poruku od klijenta: kraj
Server salje odgovor: jark
Server zavrsio sa radom...
```

Konzolni ispis u konzoli klijenta:

```
Sesija uspesno otvorena!
Unesite poruku za server (“kraj” za prekid):Pozdrav
Klijent salje poruku:Pozdrav
Ceka se odgovor servera...
Odgovor servera:vardzoP
Unesite poruku za server (“kraj” za prekid):kraj
Klijent salje poruku:kraj
Ceka se odgovor servera...
Odgovor servera:jark
Sesija i strimovi zatvoreni
```

Za realizaciju beskonekcione (**UDP**) veze potrebno je koristiti klase **DatagramPacket** i **DatagramSocket** iz paketa **java.net** (modul **java.base**). Strana koja šalje poruku (klijent/server) treba da kreira **DatagramPacket** objekat kome treba navesti sledeće vrednosti: niz bajtova poruke, dužina poruke(broj bajtova), IP adresu odredišta i odredišni port. Ove vrednosti moguće je proslediti i konstruktoru poštujući dati redosled. Slanje i prijem

sadržaja vrši se pomoću objekta klase **DatagramSocket**. Pri pozivu konstruktora ove klase potrebno je navesti port samo ukoliko je objekat namenjen za izvršavanje na server strani, u suprotnom se ne navodi. Slanje se vrši pozivom **send()** metode (podiže **IOException** izuzetak) kojoj se prosleđuje **DatagramPacket** objekat. Prijem se vrši pozivom metode **recieve()** metode (podiže **IOException** izuzetak) kojoj se prosleđuje **DatagramPacket** objekat kome nije potrebno navesti IP adresu i port, već samo niz (u kome će se smestiti sadržaj) i veličinu tog niza. U tom slučaju, izostavljena polja će imati vrednosti koja identifikuju pošiljaoca tj. njegovu IP adresu i port. Metoda za prijem zadržaja stopira izvršenje programa sve do prijema sadržaja.

U sledećem delu prikazan je prethodni primer, s tim da je ovde realizacija izvršena korišćenjem **UDP** protokola.

Sadržaj **ServerProgram.java** datoteke:

```
package server;

import java.io.*;
import java.net.DatagramPacket;
import java.net.DatagramSocket;

public class ServerProgram {
    public static void main(String[] args) {
        try {
            DatagramSocket datagramSoketServer = new DatagramSocket(9001);
            byte[] bafer = new byte[1000];
            DatagramPacket paketZahtev = new DatagramPacket(bafer, bafer.length);
            DatagramPacket paketOdgovor;
            System.out.println("Server pokrenut, ceka  zahtev klijenta...");
            String odgovor = "";
            String porukaK = "";
            while (!porukaK.equals("kraj")) {
                //prijem poruke od klijenta
                datagramSoketServer.receive(paketZahtev);
                //pretvaranje niza bajtova u string. Metoda getData() vraca sadrzaj
poruke iz paketa.
                porukaK = new String(paketZahtev.getData());
                System.out.println("Server primio poruku od klijenta (port=" +
paketZahtev.getPort() + ")" + porukaK);
                odgovor = unazadRec(porukaK);
                byte[] odgovorB = odgovor.getBytes();
                //kreiranje odgovora. Ip adresa i port se uzimaju iz pristiglog paketa.
                paketOdgovor = new DatagramPacket(odgovorB, odgovor.length(),
paketZahtev.getAddress(), paketZahtev.getPort());
                System.out.println("Server salje odgovor :" + odgovor);
                datagramSoketServer.send(paketOdgovor);
                //ciscenje bafera
                paketZahtev.setData(new byte[1000]);

                //brisanje praznina
                porukaK = porukaK.trim();
            }

            System.out.println("Server zavrsio sa radom...");

        } catch (IOException e) {
            System.out.println("Greska prilikom slanja ili primanja paketa.");
        }
    }
```

```
    private static String unazadRec(String ulaz) {
        String unazad = "";
        for (int i = ulaz.length() - 1; i > -1; i--) {
            unazad += ulaz.charAt(i);
        }
        return unazad;
    }
}
```

Sadržaj **KlijentProgram.java** datoteke:

```
package klijent;

import java.io.*;
import java.net.*;
import java.util.Scanner;

public class KlijentProgram {
    public static void main(String[] args) {
        try {
            InetAddress ipAdresaServera =
InetAddress.getByName("127.0.0.1");//prosledjeni string treba da odgovara vrednosti ip
adrese
            int portServera = 9001;
            DatagramSocket datagramSocketKlijent = new DatagramSocket();
            DatagramPacket paketZahtev;
            //kreiranje bafera za skladistenje sadrzaja paketa.
            byte[] bafer = new byte[1000];
            DatagramPacket odgovor = new DatagramPacket(bafer, bafer.length);
            Scanner unos = new Scanner(System.in);
            String poruka = "";
            String odgovorServera = "";
            //slanje poruka i primanje odgovora od servera sve dok korisnik ne unese
sadrzaj poruke "kraj"
            while (!poruka.equals("kraj")) {
                System.out.print("Unesite poruku za server(\"kraj\" za prekid):");
                poruka = unos.nextLine();
                System.out.println("Klijent salje poruku:" + poruka);
                //pretvaranje poruke u niz bajtova
                byte[] porukaB = poruka.getBytes();
                paketZahtev = new DatagramPacket(porukaB, porukaB.length,
ipAdresaServera, portServera);
                //slanje paketa
                datagramSocketKlijent.send(paketZahtev);
                System.out.println("Ceka se odgovor servera...");
                //primanje paketa od servera
                datagramSocketKlijent.receive(odgovor);
                //pretvaranje niza bajtova u string

                odgovorServera = new String(odgovor.getData());
                System.out.println("Odgovor servera:" + odgovorServera);
            }

            System.out.println("Klijent prestaje sa radom.");

        } catch (UnknownHostException e) {
            System.out.println("Desila se greska prilikom kreiranja ip adrese.");
        } catch (IOException e) {
            System.out.println("Doslo je do greske prilikom slanja ili primanja
paketa.");
        }
```

```
    }

}
```

Konzolni ispis na serveru i klijentu isti je kao u prethodnom primeru. Ovde nije potrebno pokretanje servera pre klijenta.

Glavni nedostatak prethodnih primera bio je u tome što je server u jednom trenutku mogao da opsluži samo jednog klijenta, a nakon toga bi prestao sa radom što nije slučaj u praksi. Server treba da omogući konkurentnu obradu i da pruži usluge klijentima u svakom trenutku. U sledećem delu prikazan je primer realizacije klijent/server sistema za razmenu poruka. U okviru ovog primera server je realizovan tako da može istovremeno da opsluži veći broj klijenata. Rešenje se bazira na korišćenju niti. Po otvaranju sesije, kreirara se nova nit kojoj se prosleđuju potrebni objekti između ostalog i dobijeni soket. Svaka novokreirana nit zadužena je opsluživanje jednog klijenta. Za kreiranje korisničkog grafičkog interfejsa korišćena je **Swing** biblioteka. U primeru je realizovan i postupak serijalizacije i deserijalizacije kako bi se preneli objekti između klijenta i servera.

Sadržaj **Poruka.java** datoteke:

```
package poruke;

import java.io.Serializable;
import java.text.SimpleDateFormat;
import java.util.Date;

public class Poruka implements Serializable {
    private String primalac;
    private String posiljalac;
    private String sadrzaj;
    private Date vreme;

    public Poruka(String primalac, String posiljalac, String sadrzaj, Date vreme) {
        this.primalac = primalac;
        this.posiljalac = posiljalac;
        this.sadrzaj = sadrzaj;
        this.vreme = vreme;
    }

    public String getPrimalac() {
        return primalac;
    }

    public String getPosiljalac() {
        return posiljalac;
    }

    public String getSadrzaj() {
        return sadrzaj;
    }

    public String getVreme() {
        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/YYYY HH:mm:ss");
        return sdf.format(this.vreme);
    }
}
```

Sadržaj **Sanduce.java** datoteke:

```
package poruke;

import java.util.ArrayList;

public class Sanduce {
    private ArrayList<Poruka> poruke;
    public Sanduce() {
        this.poruke = new ArrayList<>();
    }
    public synchronized void dodajPoruku(Poruka p) {
        poruke.add(p);
    }
    public synchronized ArrayList<Poruka> getPoruke(String korime) {
        ArrayList<Poruka> tmp = new ArrayList<>();
        for (Poruka p : poruke) {
            if (korime.equals(p.getPrimalac()))
                tmp.add(p);
        }
        return tmp;
    }
}
```

Sadržaj **ServerNit.java** datoteke:

```
package server;

import poruke.Poruka;
import poruke.Sanduce;
import java.io.*;
import java.net.Socket;
import java.util.ArrayList;

public class ServerNit implements Runnable {
    private Socket soket;
    private PrintWriter out;
    private BufferedReader in;
    private ObjectOutputStream outO;

    private ObjectInputStream inO;
    private Sanduce inbox;

    public ServerNit(Socket soket, Sanduce inbox) {
        this.soket = soket;
        this.inbox = inbox;
        //inicijalizacija ulazno izlaznih strimova
        try {
            in = new BufferedReader(new InputStreamReader(soket.getInputStream()));
            out = new PrintWriter(new BufferedWriter(new
OutputStreamWriter(soket.getOutputStream())), true);
            outO = new ObjectOutputStream(soket.getOutputStream());
            inO = new ObjectInputStream(soket.getInputStream());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void run() {
        System.out.println("Server nit(ID=" + Thread.currentThread().getId() + ")
pokrenuta.");
```

```
        String zahtev = "";
        String[] parametri;
        Poruka porPrimljena;
        //petlja se izvrsava sve dok od klijenta ne dobije poruku "kraj"
        while (!zahtev.equals("kraj")) {
            try {
                zahtev = in.readLine();
                parametri = zahtev.split("##");
                switch (parametri[0]) {
                    case "slanje":
                        out.println("ok");//potvrda da je server spreman da procita
objekat
                        porPrimljena = (Poruka) inO.readObject();//prijem objekta klase
Poruka
                        //dodavanje poruke u inbox
                        inbox.dodajPoruku(porPrimljena);
                        break;
                    case "mojePoruke":
                        ArrayList<Poruka> mojePor = inbox.getPoruke(parametri[1]);
                        //slanje liste Poruka klijentu
                        outO.writeObject(mojePor);
                        break;
                    case "kraj":
                        zahtev = "kraj";
                        break;
                    default:
                        out.println("Server nije u mogucnosti da odgovori na trazeni
zahtev.");
                        break;
                }

            } catch (IOException e) {
                e.printStackTrace();
            } catch (ClassNotFoundException e) {
                e.printStackTrace();

            }
        }
        System.out.println("Server nit(ID=" + Thread.currentThread().getId() + ")
ukinuta.");
    }
}
```

Sadržaj **ServerProgram.java** datoteke:

```
package server;

import poruke.Sanduce;
import java.io.IOException;
import java.net.ServerSocket;

public class ServerProgram {

    public static void main(String[] args) {
        Sanduce inbox = new Sanduce();
        try {
            ServerSocket ss = new ServerSocket(9000);
            System.out.println("Server pokrenut...");
            //beskonacna petlja koja otvara sesiju i kreira nit koja ce vrsiti
komunikaciju sa datim soketom
            while (true) {
                //kreiranje niti i startovanje nove niti po otvaranju nove sesije
```

```
                new Thread(new ServerNit(ss.accept(), inbox)).start();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Sadržaj **Klijent.java** datoteke:

```
package klijent;

import poruke.Poruka;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.InetAddress;
import java.net.Socket;
import java.util.ArrayList;
import java.util.Date;

public class Klijent extends JFrame {
    private Socket soket;
    private PrintWriter out;
    private BufferedReader in;
    private ObjectOutputStream outO;
    private ObjectInputStream inO;
    private ArrayList<Poruka> poruke;


    public Klijent() {
        setTitle("Aplikacija za razmenu poruka");

        setSize(450, 300);
        setLayout(new BorderLayout());
        JLabel lblPrikazGore = new JLabel();
        add("North", lblPrikazGore);

        JPanel pCentar = new JPanel();
        pCentar.setLayout(new GridLayout(4, 2));
        pCentar.add(new JLabel("Korisnicko ime:"));
        JTextField txtKorIme = new JTextField();
        pCentar.add(txtKorIme);
        pCentar.add(new JLabel("Poruka za:"));
        JTextField txtPorZa = new JTextField();
        pCentar.add(txtPorZa);

        JRadioButton rbSlanje = new JRadioButton("Posalji poruku");

        JRadioButton rbMojeP = new JRadioButton("Moje poruke");
        ButtonGroup btnG = new ButtonGroup();
        rbSlanje.setSelected(true);
        btnG.add(rbSlanje);
        btnG.add(rbMojeP);
        pCentar.add(rbSlanje);
        pCentar.add(rbMojeP);

        pCentar.add(new JLabel("Sadrzaj:"));
        JTextArea taSadrzaj = new JTextArea();
        pCentar.add(taSadrzaj);
        add("Center", pCentar);
```

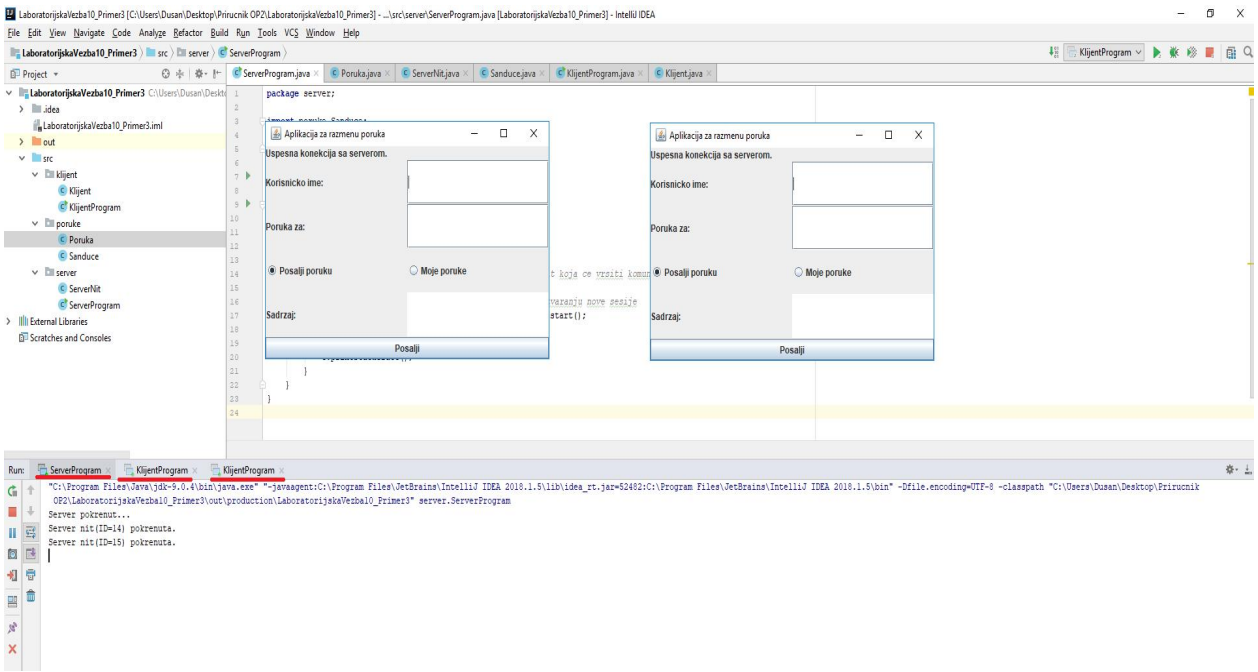
```
        JButton btnPosalji = new JButton("Posalji");
        add("South", btnPosalji);
        //otvaranje sesije i  inicijalizacija ulazno izlaznih strimova
        try {
            InetAddress ipAdresaServera = InetAddress.getByName("127.0.0.1");
            soket = new Socket(ipAdresaServera, 9000);
            in = new BufferedReader(new InputStreamReader(soket.getInputStream()));
            out = new PrintWriter(new BufferedWriter(new
OutputStreamWriter(soket.getOutputStream())), true);
            //inicijalizacija izlaznog strima za slanje serijalizovanih objekata
            outO = new ObjectOutputStream(soket.getOutputStream());
            //inicijalizacija ulaznog strima za prijem serijalizovanih objekata
            inO = new ObjectInputStream(soket.getInputStream());
            lblPrikazGore.setText("Uspesna konekcija sa serverom.");
        } catch (IOException e) {
            lblPrikazGore.setText("Doslo je do greske prilikom povezivanja sa
serverom.");
            btnPosalji.setEnabled(false);
        }
        btnPosalji.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                if (rbMojeP.isSelected()) {
                    out.println("mojePoruke##" + txtKorIme.getText());
                    try {
                        //prijem liste poruka od servera i konverzija u
ArrayList<Poruka> tip
                        poruke = (ArrayList<Poruka>) inO.readObject();
                    } catch (IOException e1) {
                        lblPrikazGore.setText("Doslo je do greske prilikom preuzimanja
poruka!");
                    } catch (ClassNotFoundException e1) {
                        //greska prilikom konverzije
                    }
                    taSadrzaj.setText("");
                    for (Poruka p : poruke) {
                        taSadrzaj.append(p.getPosiljalac() + ":  " + p.getSadrzaj() +
"\n" + p.getVreme() + "\n");
                    }
                } else {
                    out.println("slanje##");
                    try {
                        in.readLine();//potvrda da je server spreman da procita objekat
                        //slanje objekta klase Poruka serveru
                        outO.writeObject(new Poruka(txtPorZa.getText(),
txtKorIme.getText(), taSadrzaj.getText(), new Date()));
                    } catch (IOException e1) {
                        lblPrikazGore.setText("Doslo je do greske priliko slanja
poruke!");
                    }
                    taSadrzaj.setText("");
                    lblPrikazGore.setText("Poruka uspesno poslata!");
                }
            }
        });
        rbSlanje.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                taSadrzaj.setText("");
            }
```

```
        });
        addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent e) {
                //slanje serveru poruke za je prekinuto izvrsavanje klijenta i
zatvaranje strimova i sesije
                try {
                    out.println("kraj");
                    in.close();
                    out.close();
                    outO.close();
                    inO.close();
                    soket.close();
                } catch (IOException e1) {
                } catch (Exception ex) {
                }
                System.exit(1);
            }
        });
        setVisible(true);

    }
}
```

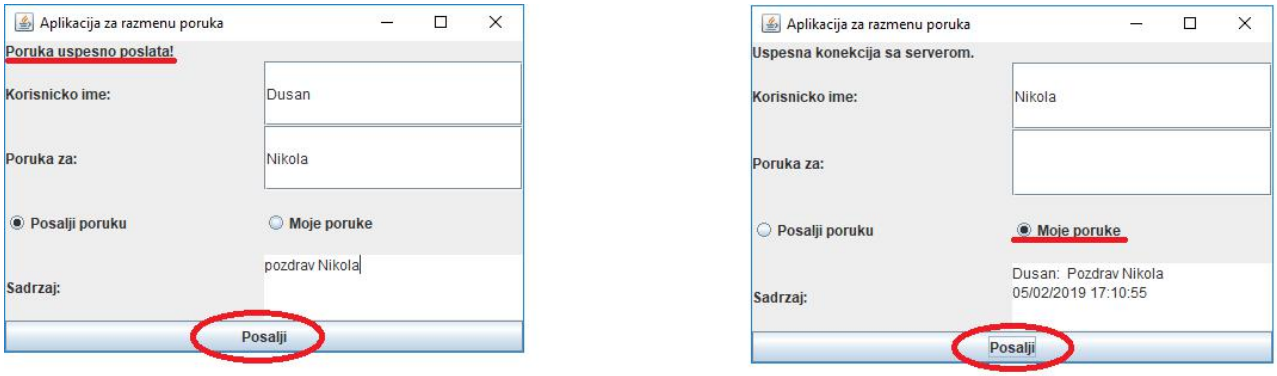
Sadržaj **KlijentProgram.java** datoteke:

```
package klijent;

public class KlijentProgram {
    public static void main(String[] args) {
        Klijent k = new Klijent();
    }
}
```

Struktura programa i različiti scenariji upotrebe prikazani su na slikama 10.3 i 10.4.



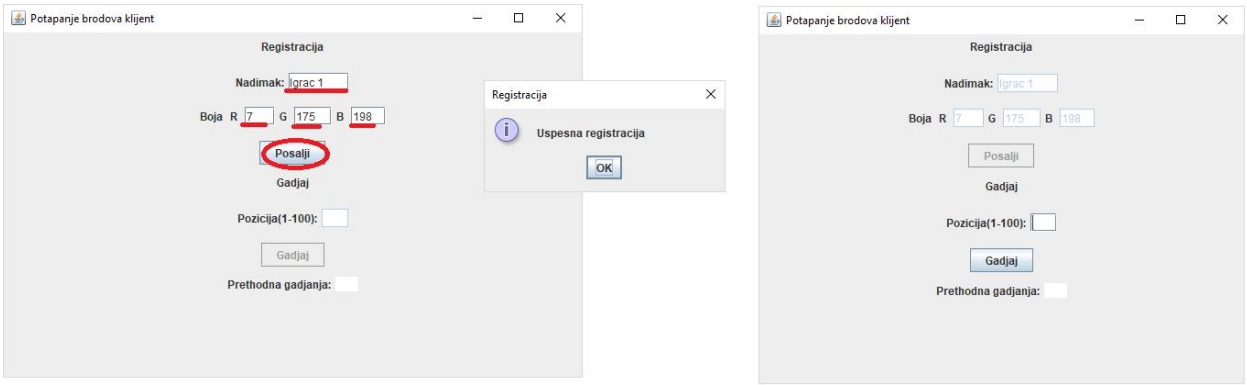
Slika 10.3- Struktura projekta, ispis serverske konzole, pokrenuta dva klijenta



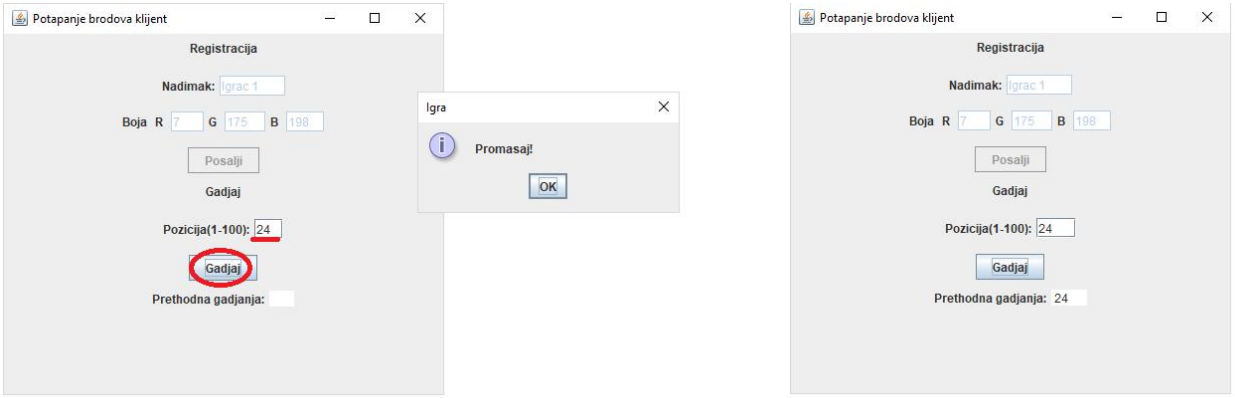
Slika 10.4 – Slanje/čitanje poruka

Zadatak za samostalni rad

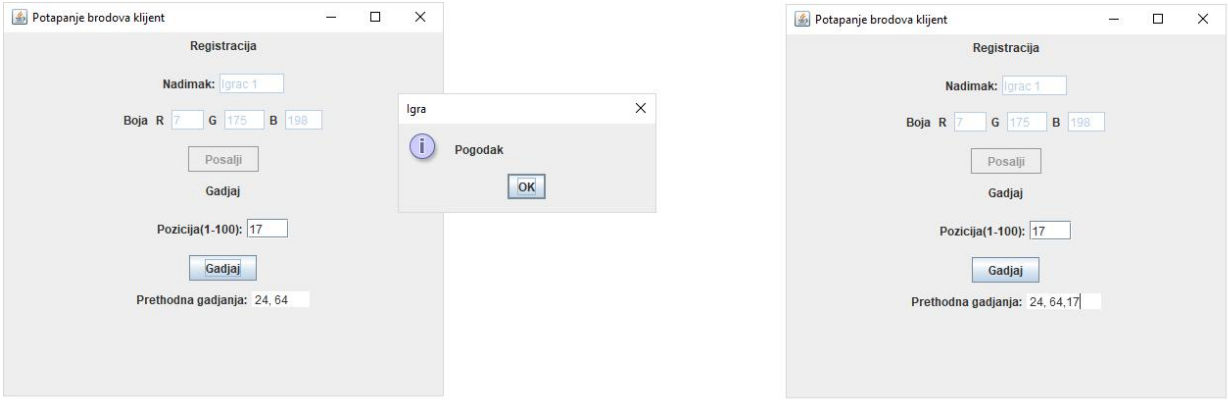
Kreirati klijent-server aplikaciju koja će simulirati igru potapanja brodova. Na serveru se treba prikazati tabla sa poljima na kojima su skriveni brodovi. Ukupno ima ima cetiri broda. Prvi brod se sastoji od jednog polja, drugi od dva, treći od tri i četvrti od četiri. Polja mogu biti raspoređena vertikalno, horizontalno i dijagonalno. Koristiti predefinisanu listu u kojoj su upisane ove pozicije. Svaka pozicija se imenuje svojim rednim brojem. Osim tabele sa poljima, server poseduje segment u kojem se ispisuje lista uspešno registrovanih igrača (klijenata) koji su sortirani po njihovim broju pogodaka. Klijent takođe poseduje grafički interfejs preko kojeg prvo vrši registraciju a zatim i "gađanje". Prilikom registracije treba da unese boju (RGB format) kojom će se označavati polja koja je gađao kao i tekst u tabeli rezultata. Registracija je uspešna ukoliko ne postoji korisnik sa unetim nadimkom i ako je format boje pravilno naveden. Pogođena polja označena su nadimkom igrača koji je pogodio i boja pozadine odgovara boji koja opisuje tog igrača. Promašena polja imaju belu pozadinu. Na tim poljima ispisuje se sadržaj u sledećem formatu "X nadimak igrača X". Na slikama 10.5 - 10.8 prikazani su različiti scenariji upotrebe.



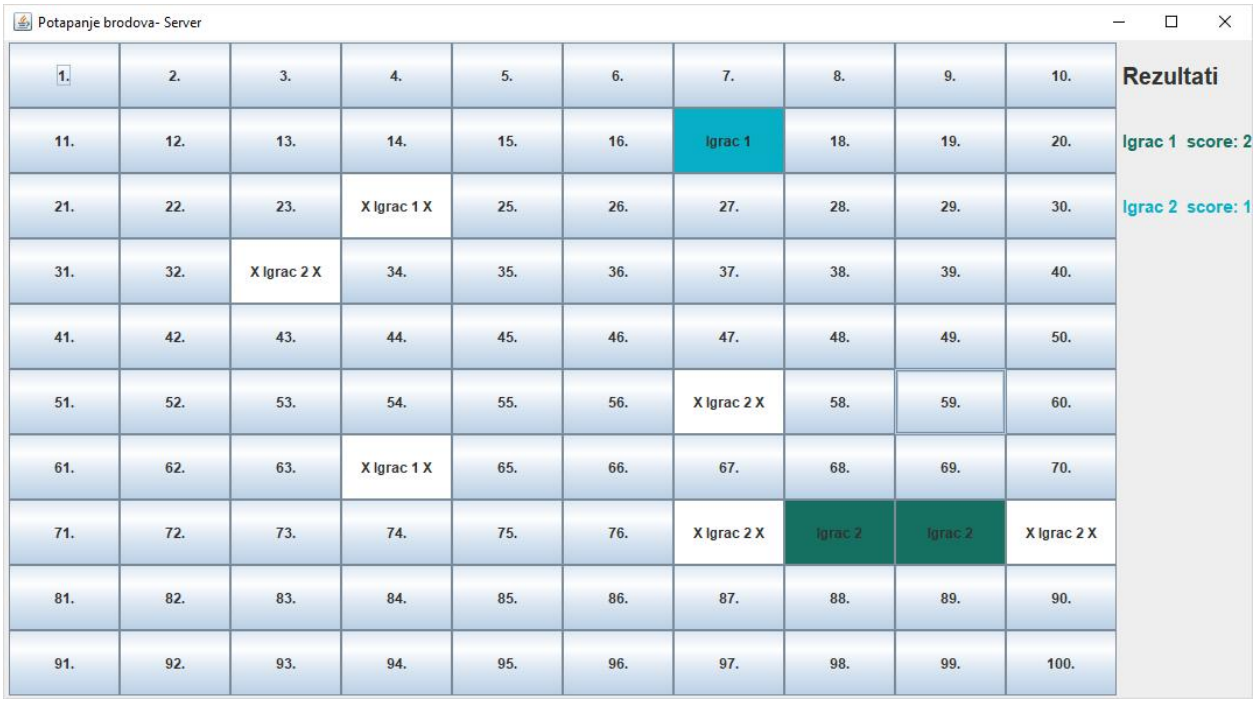
Slika 10.5- Registracija korisnika



Slika 10.6- Neuspešno gađanje



Slika 10.7 – Uspešno gađanje



Slika 10.8 – Prikaz servera

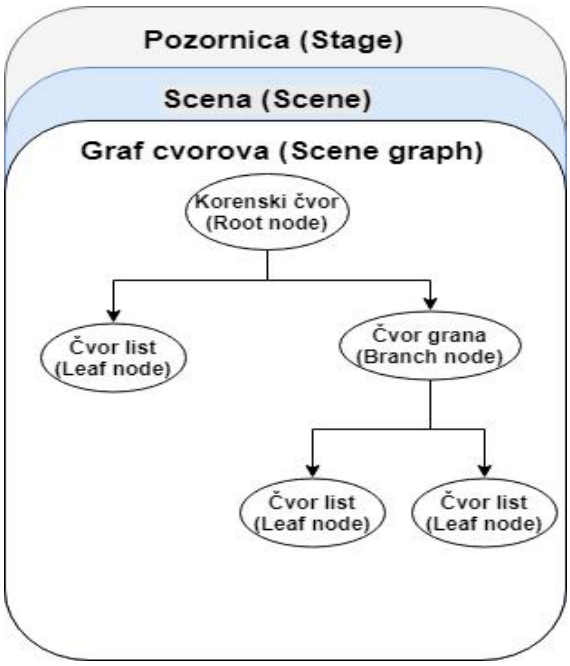
Laboratorijska vežba 11: JavaFX

Na vežbama 7, 8 i 9 prikazani su postupci kreiranja korisničkog grafičkog interfejsa korišćenjem **AWT/Swing** biblioteka. Sa pojavom **JavaFX API** (eng. application programming interface) ove biblioteke skoro da su potpuno izašle iz upotrebe. **JavaFX** je kolekcija paketa koji nudi programeru mogućnost da dizajnira, kreira, testira i razvija aplikacije visokih perfomansi. Razvijanje ovakvih aplikacija može se prilagoditi izvršavanju na različitim tipovima uređaja (mobilnim telefonima, pametnim telefonima, računarima, tabletima). Sam **API** nalazi se u **javafx.graphics** modulu koji sadrži potrebne alate za obradu sadržaja (audio, video, animacija, grafika) kao i da upravlja samim procesom izvršavanja aplikacije. Na sledećem spisku navedeni su značajniji paketi koje ovaj modul izvozi:

1. **javafx.application** – Poseduje kolekciju klasa koje upravljaju životnim ciklusom aplikacije.
2. **javafx.stage –** Poseduje kolekciju klasa za kreiranje osnovnog prikaza i funkcionalnosti prozora.
3. **javafx.scene –** Poseduje kolekciju korenskih klasa za definisanje hijerarhije u stablu scenskog grafa.
4. **javafx.animation** – Poseduje kolekciju klasa koje omogućavaju kreiranje animacija baziranih na tranziciji komponenti.
5. **javafx.concurrent –** Poseduje kolekciju klasa koje omogućavaju konkurentnu obradu.
6. **javafx.css –** Poseduje klase koje omogućavaju uređivanje vizuelnog prikaza korišćenjem **CSS- a** (eng. Cascading Style Sheets).
7. **javafx.geometry –** Poseduje kolekciju klasa koje definišu i izvršavaju operacije nad objektima u dvodimenzionalnom prostoru.
8. **javafx.scene.effect –** Poseduje kolekciju klasa koje omogućavaju kreiranje vizuelnih efekata nad čvorovima unutar scene.
9. **javafx.scene.input –** Poseduje kolekciju klasa koje upravljaju ulaznim događajima nastalim korišćenjem miša ili tastature.
10. **javafx.scene.layout –** Poseduje kolekciju klasa koje upravljaju raspoređivačima na sceni.
11. **javafx.scene.text –** Poseduje kolekciju klasa koje omogućavaju rad sa fontovima i tekstualnim prikazom generalno.
12. **javafx.scene.transform –** Poseduje kolekciju klasa koje omogućavaju izvršavanje pokreta kao što su rotacija, skaliranje, pomeranje itd.

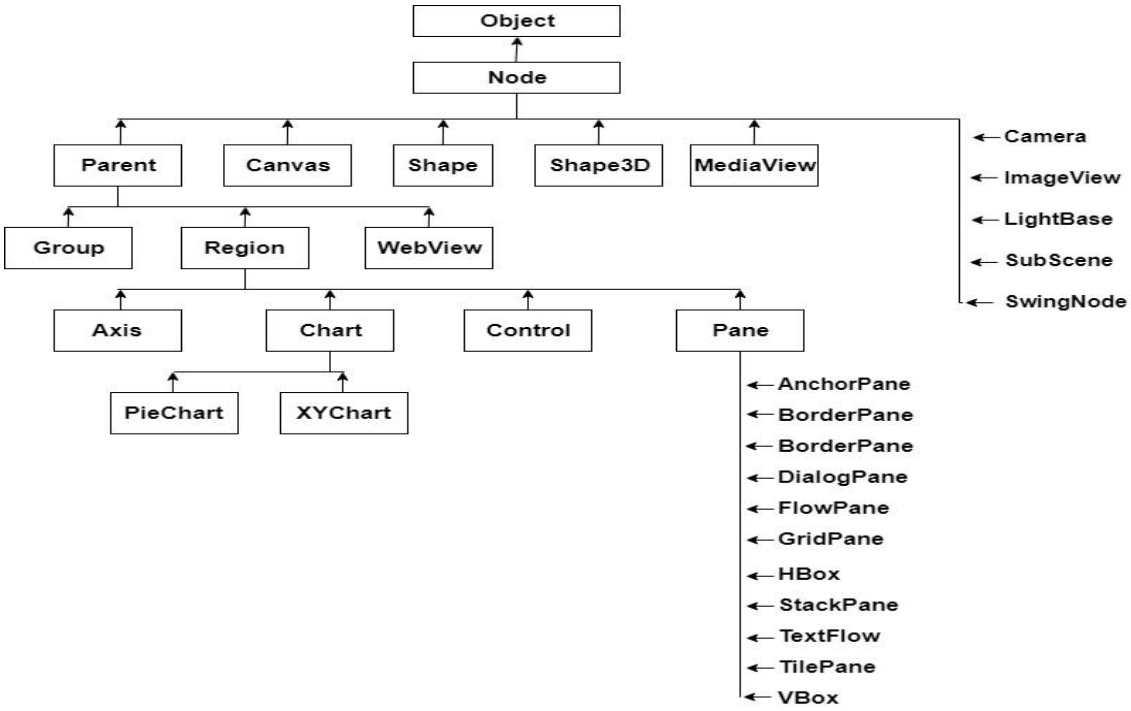
Osnovna arhitektura **JavaFX** aplikacija bazira se na korišćenju tri osnovne komponente: pozornica (eng. Stage), scena (eng. Scene) i elemenata- čvorova (eng. Nodes). U analogiji možemo posmatrati kao pozorišnu postavu koja se sastoji od pozornice na kojoj se postavlja scena sa njenim pratećim elementima. Ovakva arhitektura ilustrovana je i na slici 11.1.

Pozornica u stvari predstavlja prozor u koji spada i naslovna linija, ivice i osnovne komande za zatvaranje, minimiziranje i maksimiziranje. Definicija izgleda i funkcionalnosti prozora realizovani su klasom **Stage.** Objektom ove klase moguće je definisati osnovni izgled glavnog prozora kao i postaviti scenu koja će biti prikazana na njemu. Scena u sebi sadrži grafičke komponente koje treba prikazati u okviru korisničkog interfejsa.



Slika 11.1- Dijagram arhitekture JavaFX aplikacije

Komponente se pojedinačno nazivaju čvorovi i organizovani su kao graf sa jednim korenskim elementom (graf stablo). Natklasa svih čvorovskih elemenata zove se **Node.** Na slici 11.2 prikazan je dijagram gde je data hijerarhija bitnijih klasa čija je osnovna klasa **Node.**



*Slika 11.2- Hijerarhija bitnijih klasa izvedenih iz **Node** klase*

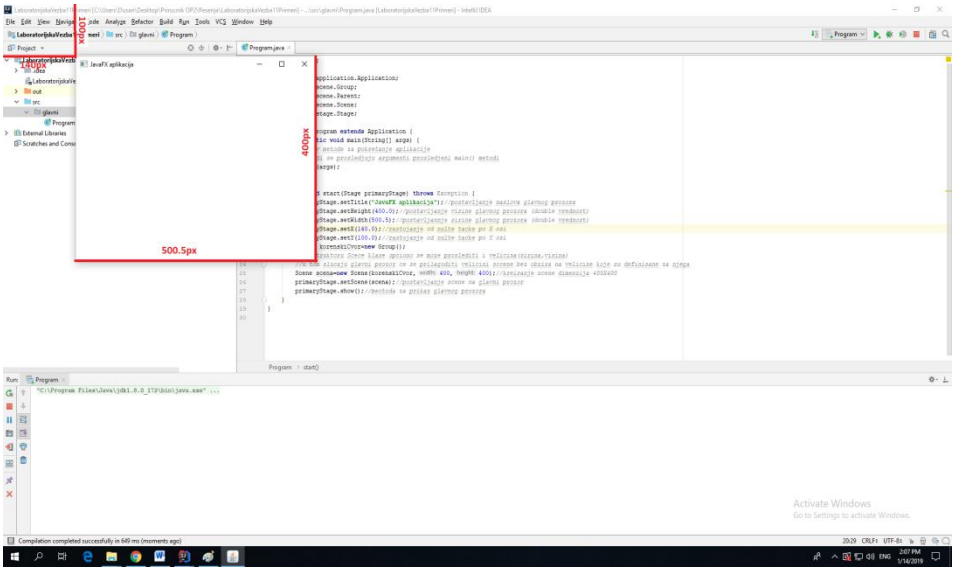
Objekti čvorovskih osobina dodaju se na scenu i na taj način postaju vidljivi u okviru glavnog prozora. Scena se kreira pomoću klase **Scene.** Prilikom instanciranja objekta ove klase potrebno je proslediti korenski čvor. Korenski čvor mora posedovati osobine **Parent** klase. Ova klasa poseduje osobine koja omogućava da čvor u sebi sadrži druge, tj. da ima deca čvorove.

Kreiranje **JavaFX** aplikacije vrši se nasleđivanjem klase **Application.** Unutar izvedene klase treba preklopiti metodu **start()**, a zatim kreirati **main()** metodu unutar koje se poziva statička metoda **launch()**. Poziv ove metode zapravo pokreće aplikaciju. Prilikom pokretanja vrši se implicitni poziv metode **start()** u kojoj se vrši kreiranje izgleda i funkcionalnosti aplikacije. Ova metoda prima objekat klase **Stage** koji je dobijen od same platforme prilikom startovanja. Prosleđenom objektu mogu se postaviti vrednosti za širinu, visinu, naslov itd. (**setWidth(), setHeight(), setTitle()**). Nakon toga treba kreirati scenu i postaviti je na pozornicu. Postavljanje scene vrši se pozivom metode **setScene()** objekta glavnog prozora. Da bi prozor bio vidljiv potrebno je pozvati i metodu **show()**. U sledećem delu prikazan je primer kreiranja **JavaFX** aplikacije.

Sadržaj **Program.java** datoteke:

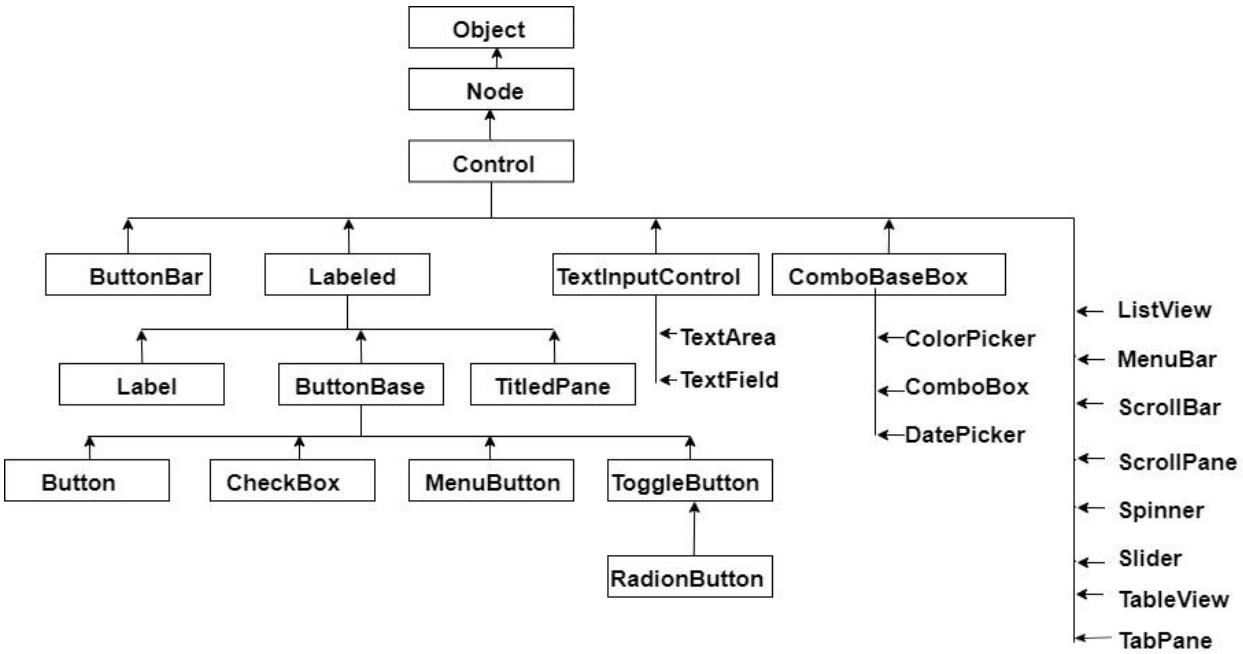
```
package glavni;

import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class Program extends Application {
    public static void main(String[] args) {
        //poziv metode za pokretanje aplikacije
        //metodi se prosledjuju argumenti prosledjeni main() metodi
        launch(args);
    }


    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("JavaFX aplikacija");//postavljanje naslova glavnog
prozora
        primaryStage.setHeight(400.0);//postavljanje visine glavnog prozora (double
vrednost)
        primaryStage.setWidth(500.5);//postavljannje sirine glavnog prozora (double
vrednost)
        primaryStage.setX(140.0);//rastojanje od nulte tacke po X osi
        primaryStage.setY(100.0);//rastojanje od nulte tacke po Y osi
        Parent korenskiCvor=new Group();
        //konstruktoru Scece klase opciono mogu se proslediti parametri za
velicinu(sirina,visina)

        Scene scena=new Scene(korenskiCvor,400,400);//kreiranje scene dimenzija 400X400
        primaryStage.setScene(scena);//postavljanje scene na glavni prozor
        primaryStage.show();//meotoda za prikaz glavnog prozora
    }
}
```

Izgled rezultujućeg prozora prikazan je na slici 11.3.



Slika 11.3- Izgled rezultujućeg prozora

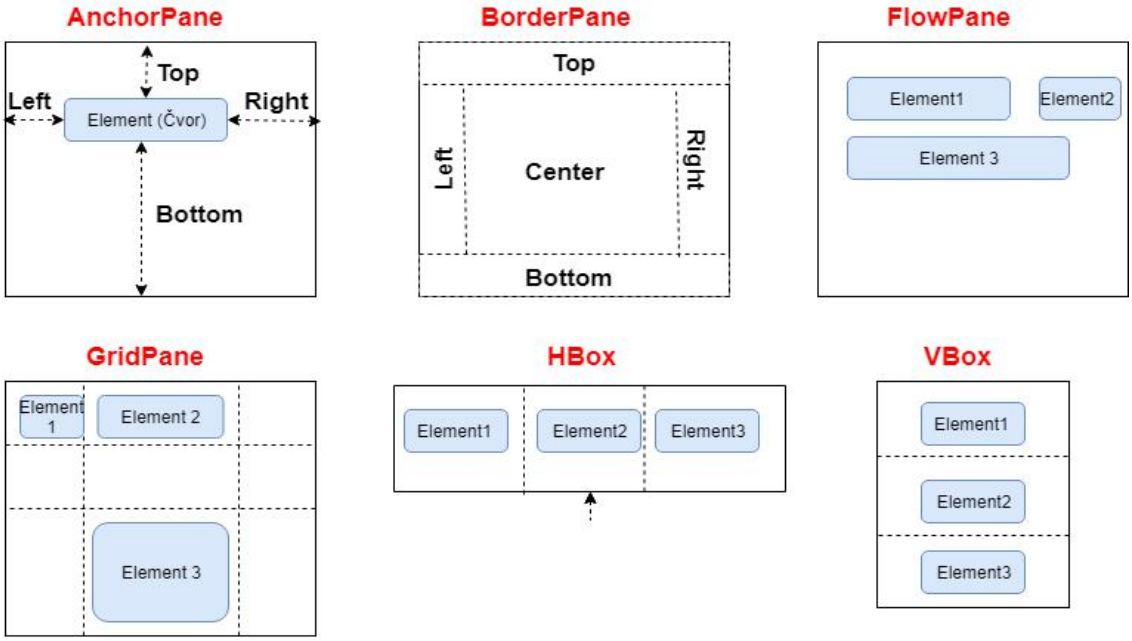
Postupak dodavanja kontrola (dugme, tekst polje, labela itd.) na scenu realizuje se korišćenjem **javafx.control** modula koji zahteva korišćenje **javafx.graphics** i izvozi sledeće pakete: **javafx.scene.chart, javafx.scene.control, javafx.scene.control.cell** i **javafx.scene.control.skin.** Unutar ovih paketa nalaze se klase koje omogućavaju rad sa kontrolama, dijagramima kao i klase zadužene za opis izgleda samih kontrola. Na slici 11.4 prikazana je hijerarhija bitnijih klasa koje su izvedene iz klase **Control.**



*Slika 11.4- Hijerarhija bitnijih klasa izvedenih iz klase **Control***

Raspored elemenata (čvorova) definiše se pomoću prozorskih okvira (eng. pane). Okviri određuju koju će veličinu i poziciju imati elementi. Menjanjem veličine prozora menja se i veličina elemenata. Osnovne osobine okvira definisani su u klasi **Pane.** Bitnije klase koje poseduju osobine okvira su:

1. **AnchorPane –** Definiše okvir koji raspoređuje elemente po zadatim vrednostima za rastojanja od ivica (gornje, donje, leve, desne).
2. **BorderPane –** Definiše okvir koji raspoređuje elemente po već definisanim pozicijma. Pozicije koje se mogu dodeliti elementu su : **Top, Bottom, Left, Right, Center.**
3. **FlowPane –** Definiše okvir koji raspoređuje elemente u horizontalnim redovima. Ukoliko nema dovoljno mesta komponente se prebacuju u red ispod.
4. **GridPane –** Definiše okvir za tabelarno raspoređivanje elemenata. Elementi se raspoređuju po ćelijama unutar fleksibilne pravougaone mreže.
5. **Hbox –** Definiše okvir koji raspoređuje elemente u horizontalnom poretku.
6. **Vbox –** Definiše okvir koji raspoređuje elemente u vertikalnom poretku.



Slika 11.5 – Okviri rasporeda

Unutar klase **Parent** definisana je metoda **getChildren()** koja vraća listu deca elemenata koji se nalaze unutar čvora. Povratni tip metode je **ObservableList<Node>.** Dodavanje novih elemenata vrši se tako što se element ubacuje u ovu listu pozivom metoda **add()** ili **addAll().**

Izgled samih komponenti može se menjati pozivom metoda kojim bi se prosledila nova vrednost za određeni parametar. U **JavaFX** biblioteci omogućena je i upotreba **CSS** (eng. Cascading Stylesheets) za opis izgleda komponenti. Kreiranjem stila definisaće se kako će neka komponenta izgledati. Svaki stil sastoji se od niza pravila koja opisuju određeni segment izgleda. Definicija pravila vrši se tako što se prvo navodi ime svojstva (font-size, background-color, border-style itd.), a zatim se iza dvotačke (:) navodi i vrednost (20, green, dashed) koju će to svojstvo poprimiti. Ispred imena svojstva navodi se prefiks "**-fx-".** Razdvajanje pravila vrši se pomoću karaktera tačka-zarez (;). Stilovi se primenjuju na sličan način kao što se primenjuju i kod **HTML** elemenata u web stranicama. Prvo se primenjuju na roditeljske elemente, a zatim na njihove podelemente (deca elementi). Stilovi se mogu pisati kao linijski (eng. inline) ili kao eksterni. Linijski se prosleđuju metodi **setStyle()** kao **String** vrednost. Eksterni stilovi su zapisani u okviru datoteka koje imaju ekstenziju **.css**. Ovde su stilovi

grupisani u blokovima koji se vezuju za selektor. Selektorom se određuje na koje će elemente stil biti primenjen. Postoje tri različita tipa selektora:

1. Klasni selektor - Dati stil se primenjuje na svaki element koji koristi ovu stilsku klasu. Primer:

```
.imeKlase {
-fx-min-width: 200px;
-fx-font-size: 20;
-fx-background-color: rgb(225, 228, 203);
}
```

2. Tipski selektor - Stil se primenjuje na elemente koji odgovaraju tipu koji je dat u selektoru. Primer:

```
Button {
-fx-min-width: 200px;
-fx-font-size: 20;
-fx-background-color: rgb(225, 228, 203);
}
```

3. Selektor po identifikatoru - Stil će se primeniti na element čiji jedinstveni identifikator odgovara selektoru. Moguća je primena samo na jedan element. Primer:

```
#dugmePosalji {
    -fx-border-color: rgb(55,200,128);
}
```

Da bi se eksterni stilovi primenili na komponente potrebno ih je uvesti u okviru scene. Uvođenje se vrši pomoću sledeće naredbe:

```
scena.getStylesheets().add("putanjaDoEksterneCSSDatoteke");
```

U sledećem delu dat je primer u kome su komponente dodate unutar gore navedenih okvira. Primer sadrži i postupak korišćenja **CSS** stilova.

Sadržaj **Program.java** datoteke:

```
package glavni;

import javafx.application.Application;

import javafx.collections.ObservableList;
import javafx.scene.Node;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.*;
import javafx.stage.Stage;

public class Program extends Application {
    public static void main(String[] args) {
        launch(args);
    }
```

```
    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("Primer rasporedjivaca");
        Parent korenskiCvor = new GridPane();
        dodajAnchorPane(korenskiCvor);
        dodajBorderPane(korenskiCvor);
        dodajFlowPane(korenskiCvor);
        dodajGridPane(korenskiCvor);
        dodajHBox(korenskiCvor);
        dodajVBox(korenskiCvor);

        Scene scena = new Scene(korenskiCvor);
        scena.getStylesheets().add("stilovi/primerStil.css");//ubacivanje eksternog CSS
stila
        primaryStage.setScene(scena);
        primaryStage.show();
    }

    private void dodajAnchorPane(Parent korenskiCvor) {
        AnchorPane aP = new AnchorPane();//kreiranje okvira
        aP.setStyle("-fx-background-color: ff0f0f0f;-fx-border-style:
solid");//postavljanje CSS stila za aP okvir
        aP.setPrefSize(200, 200);//postavljanje podrazumevane velicine 200,200
        Button dugme1 = new Button("Dugme 1");//kreiranje dugmeta
        AnchorPane.setTopAnchor(dugme1, 30.0);//staticka metoda koja postavlja
rastojanje od gornje ivice i dugmeta
        AnchorPane.setLeftAnchor(dugme1, 50.0);//staticka metoda koja postavlja
rastojanje od leve ivice i dugmeta

        ObservableList<Node> elementiOkvira = aP.getChildren();
        elementiOkvira.add(dugme1);//dodavanje elementa dugme1 na okvir aP

        ((GridPane) korenskiCvor).add(aP, 0, 0);//dodavanje okvira aP unutar korenskog
cvora unutar prve kolone prvog reda. Metoda add(Node n, int ri,int ci) nalazi se unutar
klase GridPane
    }

    private void dodajBorderPane(Parent korenskiCvor) {
        BorderPane bP = new BorderPane();
        bP.setStyle("-fx-border-style: solid");//postavljanje CSS stila za bP okvir

        Label lblGore = new Label("Pozicija Top");
        lblGore.getStyleClass().add("labelaBorderPane");//dodela stilske klase
'labelaBorderPane'
        Label lblDole = new Label("Pozicija Bottom");
        lblDole.getStyleClass().add("labelaBorderPane");//dodela stilske klase
'labelaBorderPane'
        Label lblLevo = new Label("Pozicija Left");
        lblLevo.getStyleClass().add("labelaBorderPane");//dodela stilske klase
'labelaBorderPane'
        Label lblDesno = new Label("Pozicija Right");
        lblDesno.getStyleClass().add("labelaBorderPane");//dodela stilske klase
'labelaBorderPane'
        Label lblCentar = new Label("Pozicija Center");
        lblCentar.getStyleClass().add("labelaBorderPane");//dodela stilske klase
'labelaBorderPane'
        bP.setTop(lblGore);//dodavanje elementa na poziciji Top
        bP.setBottom(lblDole);//dodavanje elementa na poziciji Bottom
        bP.setLeft(lblLevo);//dodavanje elementa na poziciji Left
        bP.setRight(lblDesno);//dodavanje elementa na poziciji Right
```

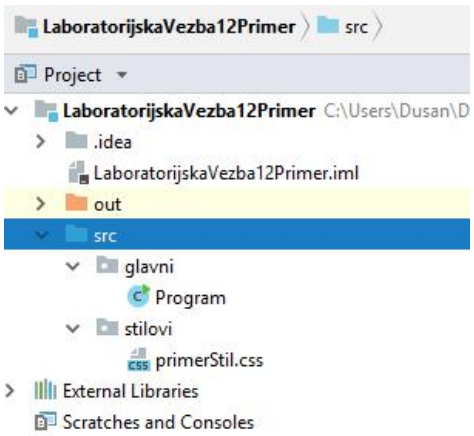
```
        bP.setCenter(lblCentar);//dodavanje elementa na poziciji Centar
        ((GridPane) korenskiCvor).add(bP, 1, 0);//dodavanje okvira aP unutar korenskog
cvora unutar druge kolone prvog reda.
    }
    private void dodajFlowPane(Parent korenskiCvor) {
        FlowPane fP = new FlowPane();
        fP.setStyle("-fx-border-style: solid");
        String stil = "-fx-font-size: 15";
        Label lblPol = new Label("Pol:  ");
        lblPol.setStyle(stil);
        RadioButton rb1 = new RadioButton("Muski");
        rb1.setStyle(stil);
        RadioButton rb2 = new RadioButton("Zenski");
        rb2.setStyle(stil);
        ToggleGroup grupaPol = new ToggleGroup();//kreiranje grupe za radio dugmad
        rb1.setToggleGroup(grupaPol);// dodeljivanje grupe radio dugmetu
        rb2.setToggleGroup(grupaPol);
        fP.getChildren().addAll(lblPol, rb1, rb2);
        ((GridPane) korenskiCvor).add(fP, 2, 0);//dodavanje okvira aP unutar korenskog
cvora unutar trece kolone prvog reda.
    }
    private void dodajGridPane(Parent korenskiCvor) {
        GridPane gP = new GridPane();
        gP.setStyle("-fx-border-style: solid");
        Label lblIme = new Label("Ime:");
        TextField txtIme = new TextField();//kreiranje polja za tekstualni unos
        txtIme.setPromptText("Unesite svoje ime");//postavljanje privremenog teksta
        Label lblPrezime = new Label("Prezime:");
        TextField txtPrezime = new TextField();
        txtPrezime.setPromptText("Unesite svoje prezime");
        GridPane.setConstraints(lblIme, 0, 0);//staticka metoda koja postavlja redni
broj kolone i reda pozicije gde se postavlja element
        gP.getChildren().add(lblIme);
        //kraci nacin
        //gP.add(lblIme,0,0);

        gP.add(txtIme, 1, 0);
        gP.add(lblPrezime, 0, 1);
        gP.add(txtPrezime, 1, 1);
        Button btnPotvrda = new Button("Potvrda");
        gP.add(btnPotvrda, 0, 2, 2, 1);//postavljanje elementa unutar prve kolone
treceg reda koji se prostire na dve celije kolone i jednom redu

        btnPotvrda.setId("dugmePotvrda");//dodela jedinstvenog identifikatora
        ((GridPane) korenskiCvor).add(gP, 0, 1);//dodavanje okvira gP unutar korenskog
cvora unutar prve kolone drugog reda.

    }

    private void dodajHBox(Parent korenskiCvor) {
        HBox hB = new HBox(20);//kreiranje hB okvira. Razmak izmedju elemenata
postavljen je na 20
        hB.setStyle("-fx-border-style: solid");
        CheckBox cb1 = new CheckBox("Izbor 1");
        CheckBox cb2 = new CheckBox("Izbor 2");
        CheckBox cb3 = new CheckBox("Izbor 3");
        hB.getChildren().addAll(cb1, cb2, cb3);

        ((GridPane) korenskiCvor).add(hB, 1, 1);//dodavanje okvira gP unutar korenskog
cvora unutar druge kolone drugog reda.
    }

    private void dodajVBox(Parent korenskiCvor) {
```

```
        VBox vB = new VBox(10);//kreiranje VB okvira. Razmak izmedju elemenata
postavljen je na 10
        vB.setStyle("-fx-border-style: solid");
        CheckBox cb1 = new CheckBox("Izbor 1");
        CheckBox cb2 = new CheckBox("Izbor 2");
        CheckBox cb3 = new CheckBox("Izbor 3");
        vB.getChildren().addAll(cb1, cb2, cb3);

        ((GridPane) korenskiCvor).add(vB, 2, 1);//dodavanje okvira gP unutar korenskog
cvora unutar trece kolone drugog reda.
    }

}
```

Sadržaj **primerStil.css** datoteke:

```
/*definicija stila koji ce se primeniti na sve RadioButton objekte*/
CheckBox{


    -fx-background-color: yellow;
    -fx-font-size: 17;
    -fx-font-family: "Times New Roman";


}
/*definicija stila koji ce se primeniti na elemente koji koriste ovu stilsku klasu*/
.labelaBorderPane{
    -fx-font-size: 20;
    -fx-background-color:#4286f4;
    -fx-border-style: solid;
}
/*definicija stila za element sa id-em 'dugmePotvrda'*/
#dugmePotvrda{
    -fx-min-width: 200px;
    -fx-background-color: rgb(55,200,128);
    -fx-font-size: 18;
}
```

Struktura projekta data je na slici 11.5:



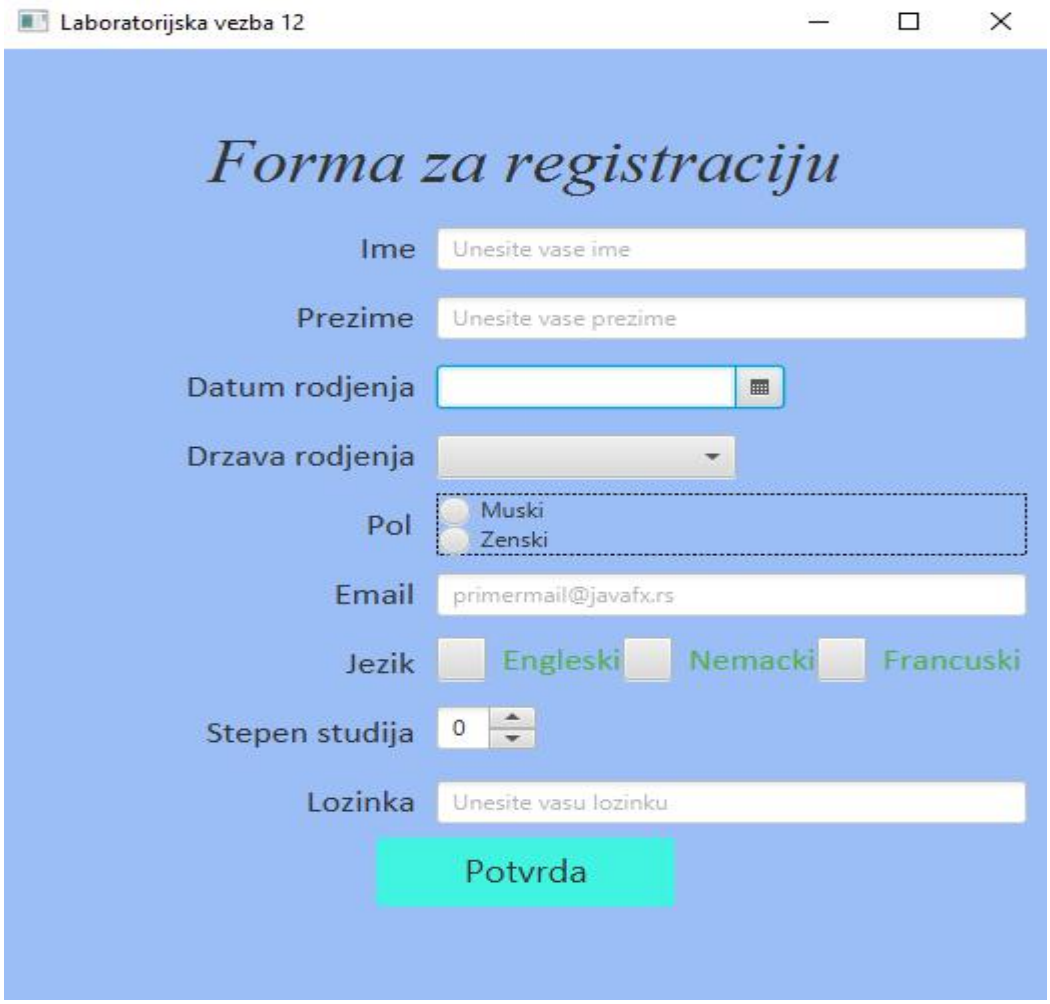
Slika 11.5 - Struktura projekta

Rezultujući prozor prikazan je na slici 11.6. Primetiti kako se veličina i pozicija elemenata menja pri promeni veličine glavnog prozora.

Slika 11.6 - Raspoređivači

Zadatak za samostalan rad:

Kreirati korisnički grafički interfejs sa sadržajem koji je prikazan na slici 11.7. Pri kreiranju polja za unos podataka koristiti sledeće komponente: ime (**TextField**), prezime (**TextField**), datum rođenja (**DatePicker**), država rođenja (**ChoiseBox** sa dve moguće vrednosti "Srbija" i "Strana drzava"), pol (**RadioButton**), email (**TextField**), jezik (**CheckBox**), stepen studija (**Spinner**, 0- 3), lozinka (**PasswordField**). Naslov forme napisan je "Times New Roman" fontom veličine 36, veličina teksta u labelama je 18 , font "Calibri". Pozadinska boja forme je **rgb(155, 190, 247)**, boja teksta u poljima za odabir jezika je **rgb(81, 173, 69)**. Pozadinska boja tastera za potvrdu je **rgb(65, 244, 226)**, veličina slova 18.

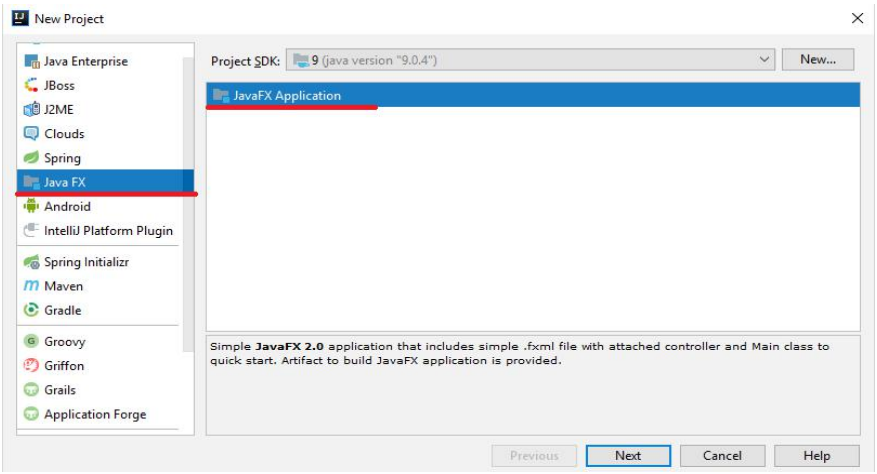


Slika 11.7- Forma za registraciju korisnika

Laboratorijska vežba 12: JavaFX FXML, upravljanje događajima

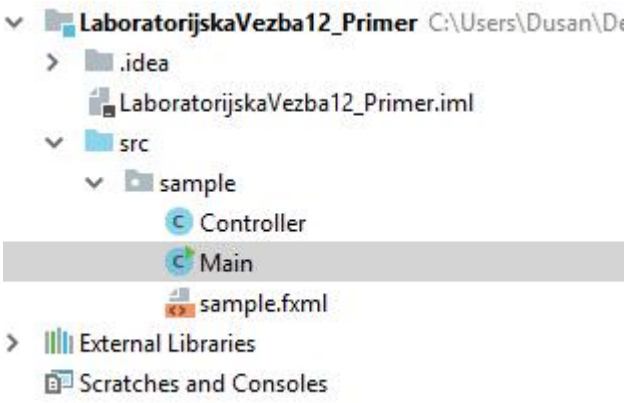
FXML je skripni jezik baziran na **XML** stukturi pomoću kojeg je moguće kreirati korisnički grafički interfejs nezavisno od ostatka aplikacije. Ovaj pristup odgovara **MVC** (Model View Controller) konceptu koji zalaže podelu aplikacije na tri celine (model, kontroler, izgled). Na taj način moguće su promene izgleda bez izmena u logičkoj strukturi aplikacije. **FXML** sadržajem opisuje se izgled komponenti (scene) dok se kontorolerom upravlja izvršavanjem komandi koje su zadate putem grafičkog interfejsa. Kontroler je ništa drugo do klase koja je deklarisana kao kontroler za **FXML** datoteku. Kontroler klasa opciono implementira **Initializable** interfejs. **FXML** ne poseduje raspored (**XML schema**) po kojem se elementi (tagovi) pišu, ali ima predefinisanu strukturu koja se mora poštovati. Obično imena **JavaFX** klasa odgovaraju imenima elemenata u **FXML**. Učitavanje **FXML** sadržaja vrši se pomoću **FXMLLoader** klase. Klasa poseduje statičku metodu **load()** kojoj se prosleđuje putanja do **FXML** datoteke i koja vraća hijerarhiju objekata koja je opisana u datoteci.

Ukoliko se prilikom kreiranja projekta odabere tip **JavaFX Application,** projekat će biti kreiran sa podrazumevanim sadržajem koji odgovara **MVC** konceptu. Projekat će posedovati paket (**sample)** u kome će se nalaziti kontroler klasa (**Controller), FXML** datoteka (**sample.fxml)** i **Main** klasa u kojoj se nalazi glavna logika aplikacije. Na slici 12.1 prikazan je prozor za odabir tipa projekta.



Slika 12.1 – Odabir tip projekta

Korenski čvor je podrazumevano postavljen kao **GridPane** bez podelemenata što se može videti iz sadržaja **sample.fxml** datoteke. U korenskom elementu izvršeno je povezivanje sa kontroler klasom pomoću atributa **fx:controller,** kao i: postavljanje oblasti važenja (**xmlns:fx="http://javafx.com/fxml"),** centralno pozicioniranje (**alignment="center"**), postavljanje horizontalnog rastojanja između elemenata (**hgap="10")** i postavljanje vertikalnog rastojanja između elemenata (**vgap="10").**



Slika 12.2- Struktura JavaFX projekta

Sadržaj **sample.fxml** datoteke:

```
<!-- Ubacivanje potrebnih elemenata-->
<?import javafx.geometry.Insets?>
<?import javafx.scene.layout.GridPane?>

<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<!-- Definicija korenskog cvora-->
<GridPane fx:controller="sample.Controller"
          xmlns:fx="http://javafx.com/fxml" alignment="center" hgap="10" vgap="10">
</GridPane>
```

Sadržaj **Main.java** datoteke:

```
package sample;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class Main extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception{
        Parent root =
FXMLLoader.load(getClass().getResource("sample.fxml"));//ucitavanje korenskog cvora iz
FXML datoteke
        primaryStage.setTitle("Hello World");
        primaryStage.setScene(new Scene(root, 300, 275));
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Elementi u **FXML** mogu predstavljati sledeće vrednosti: instancu klase, osobinu objekta, statičko svojstvo, blok za definiciju i skriptni blok. Pre korišćenja potrebno je importovati potrebne klase. Uvođenje klasa vrši na početku dokumenta, na sledeći način **<? import**

putanja.do.klase **?>** . Atributi elemenata mogu predstavljati sledeće vrednosti: svojstvo objekta, statičko svojstvo ili rukovaoca događaja (eng. event handler). U sledećem delu dat je primer sa okvirima za raspored koji je rađen na prethodnoj vežbi, s tim da je ovde izgled opisan u **sample.fxml** datoteci. U primeru su korišćeni različiti načini za kreiranje i postavljanje vrednosti elementa i njegovih atributa.

Sadržaj **sample.fxml** datoteke:

```
<!-- Ubacivanje potrebnih elemenata-->
<?import javafx.scene.layout.GridPane?>

<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.layout.RowConstraints?>
<?import javafx.scene.layout.AnchorPane?>
<?import javafx.scene.layout.BorderPane?>
<?import javafx.scene.layout.FlowPane?>
<?import javafx.scene.control.RadioButton?>
<?import javafx.scene.control.ToggleGroup?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.layout.HBox?>
<?import javafx.scene.control.CheckBox?>
<?import javafx.scene.layout.VBox?>
<!-- kreiranje korenskog elementa i importovanje eksternog css stila "primerStil.css"--
>
<GridPane xmlns:fx="http://javafx.com/fxml" fx:controller="sample.Controller"
stylesheets="@../stilovi/primerStil.css">
    <!-- definicija redova GridPane-a -->
    <rowConstraints><!-- svojstvo GridPane-a -->
        <RowConstraints percentHeight="70"/><!-- prvi red zauzima 70% visine prozora-->
        <RowConstraints percentHeight="30"/><!-- drugi red zauzima 30% visine prozora--
>
    </rowConstraints>
    <!--dodavanje elemenata unutar glavnog cvora -->
    <children>
        <!--kreiranje okvira AnchorPane i postavljanje na poziciju prva kolona prvog
reda, sa vidljivom ivicom-->
        <AnchorPane GridPane.columnIndex="0" GridPane.rowIndex="0"
                    style="-fx-background-color: ff0f0f0f;-fx-border-style: solid"
prefWidth="200">
            <children>
                <Button text="Dugme 1" AnchorPane.topAnchor="30.0"
                        AnchorPane.leftAnchor="50.0"></Button> <!--kreiranje dugmeta i
postavljanje rastojanja od gornje i leve ivice (30,50) -->
                <!-- moze i na sledeci nacin
                <Button>
                    <AnchorPane.topAnchor>30.0</AnchorPane.topAnchor>
                    <AnchorPane.leftAnchor>50.0</AnchorPane.leftAnchor>
                    <text>Dugme 1</text>
                </Button>
                -->
            </children>
        </AnchorPane>

        <!--kreiranje okvira BorderPane i postavljanje na poziciju druga kolona prvog
reda, sa vidljivom ivicom-->
        <BorderPane style="-fx-border-style: solid" GridPane.rowIndex="0"
GridPane.columnIndex="1">
            <!--dodavanje labele na poziciji Top-->
            <top>
```

```
            <!--kreiranje labele i postavljanje stilske klase "labelaBorderPane"-->
            <Label text="Pozicija top" styleClass="labelaBorderPane"/>
        </top>
        <!--dodavanje labele na poziciji Right-->
        <right>
            <Label text="Pozicija right" styleClass="labelaBorderPane"/>
        </right>
        <!--dodavanje labele na poziciji Center-->
        <center>
            <Label text="Pozicija center" styleClass="labelaBorderPane"/>
        </center>
        <!--dodavanje labele na poziciji Left-->
        <left>
            <Label text="Pozicija left" styleClass="labelaBorderPane"/>
        </left>
        <!--dodavanje labele na poziciji Bottom-->
        <bottom>
            <Label text="Pozicija bottom" styleClass="labelaBorderPane"/>
        </bottom>
    </BorderPane>
    <!--kreiranje okvira FlowPane i postavljanje na poziciju treca kolona prvog
reda, sa vidljivom ivicom-->
    <FlowPane GridPane.columnIndex="2" GridPane.rowIndex="0" style="-fx-border-
style: solid">
        <children>
            <!--kreiranje grupe za RadioButton, primer bloka za definiciju-->
            <fx:define>
                <ToggleGroup fx:id="tgPol"/>
            </fx:define>
            <Label text="Pol:" style="-fx-font-size: 15"/>
            <!-- kreiranje radio polja, postaljanje grupe i podesavanje velicine
slova na 15-->
            <RadioButton text="Muski" toggleGroup="$tgPol" style="-fx-font-size:
15"/>
            <RadioButton text="Zenski" toggleGroup="$tgPol" style="-fx-font-size:
15"/>
        </children>
    </FlowPane>
    <!--kreiranje okvira GridPane i postavljanje na poziciju prva kolona drugog
reda, sa vidljivom ivicom-->
    <GridPane GridPane.rowIndex="1" GridPane.columnIndex="0" style="-fx-border-
style: solid">
        <children>
            <Label>
                <text>Ime:</text><!-- svojstvo objekta-->
                <GridPane.rowIndex>0</GridPane.rowIndex><!-- staticko svojstvo
GridPane-a-->
                <GridPane.columnIndex>0</GridPane.columnIndex><!-- staticko
svojstvo GridPane-a-->
            </Label>
            <TextField GridPane.rowIndex="0" GridPane.columnIndex="1"/>
            <Label text="Prezime:" GridPane.columnIndex="0" GridPane.rowIndex="1"/>
            <TextField GridPane.rowIndex="1" GridPane.columnIndex="1"/>
            <!-- kreiranje dugmeta sa id-em "dugmePotvrda" koje se prostire na dve
kolone-->
            <Button text="Potvrda" fx:id="dugmePotvrda" GridPane.columnIndex="0"
GridPane.rowIndex="2"
                    GridPane.columnSpan="2"/>
        </children>
    </GridPane>
    <!--kreiranje okvira HBox i postavljanje na poziciju druga kolona drugog reda,
```

```
sa vidljivom ivicom-->
        <HBox GridPane.columnIndex="1" GridPane.rowIndex="1" style="-fx-border-style:
solid" spacing="20">
            <children>
                <!-- kreiranje i dodavanje CheckBox elemenata-->
                <CheckBox text="Izbor 1"/>
                <CheckBox text="Izbor 2"/>
                <CheckBox text="Izbor 3"/>
            </children>
            <!--kreiranje okvira VBox i postavljanje na poziciju treca kolona drugog
reda, sa vidljivom ivicom-->
        </HBox>
        <VBox GridPane.columnIndex="2" GridPane.rowIndex="1" style="-fx-border-style:
solid" spacing="10">
            <children>
                <CheckBox text="Izbor 1"/>
                <CheckBox text="Izbor 2"/>
                <CheckBox text="Izbor 3"/>
            </children>
        </VBox>
    </children>
</GridPane>
```

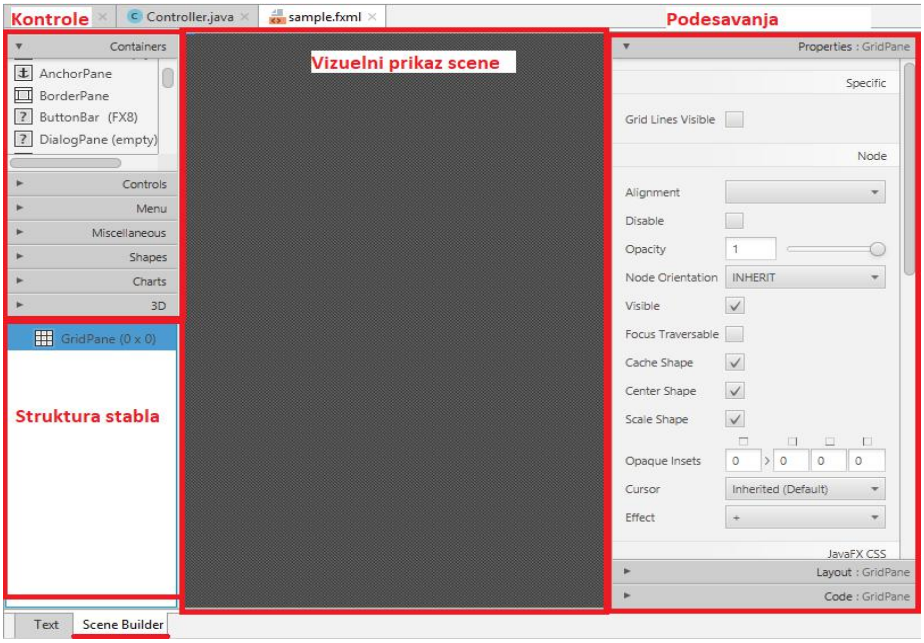
Sadržaj **Main.java** datoteke:

```
package sample;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class Main extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception{
        Parent root =
FXMLLoader.load(getClass().getResource("sample.fxml"));//ucitavanje korenskog cvora iz
FXML datoteke
        primaryStage.setTitle("Primer rasporedjivaca");
        primaryStage.setScene(new Scene(root));
        primaryStage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

Izgled rezultujućeg prozora prikazan je na slici 12.3.

Slika 12.3- Raspoređivači

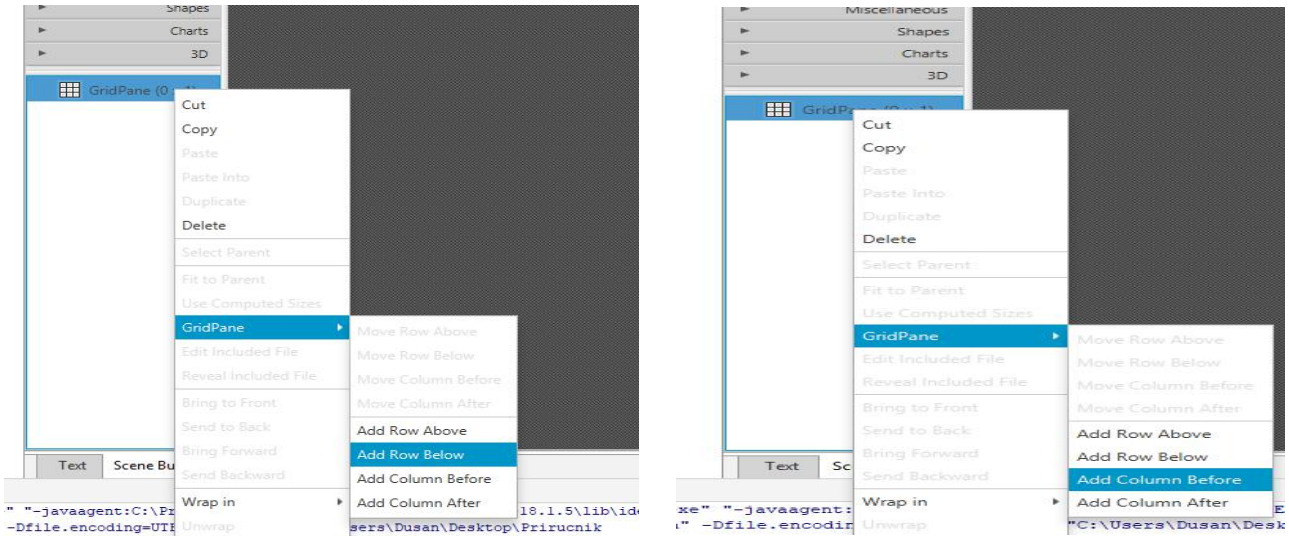
Kako bi se dodatno olakšao posao kreiranja korisničkog interfejsa razvijen je alat **Scene Builder.** Ovaj alat omogućava kreiranje grafičkog interfejsa bez pisanja koda. Kreiranje se vrši jednostavnim prevlačenjem komponenti na deo koji predstavlja prozor - scenu. Osim dodavanja, moguće je izvršiti osnovna i dodatna podešavanja komponenti odabirom kartice iz panela za podešavanja.



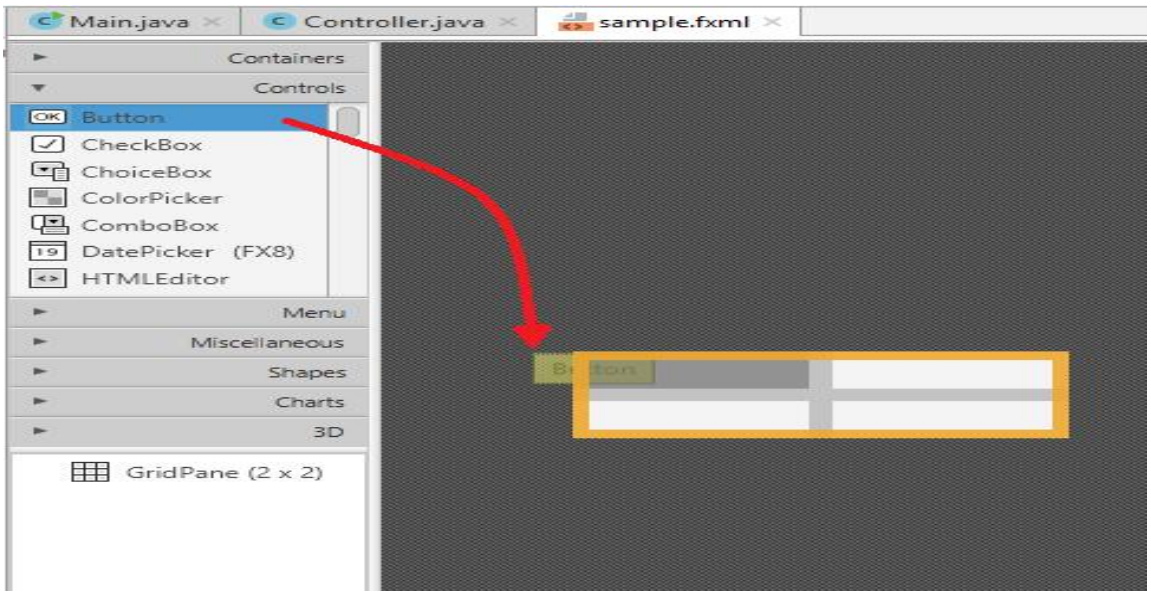
Slika 12.4- Izgled Scene Builder alata

Samo grafičko okruženje se sastoji od tri veće celine. U levom segmentu nalaze se padajući meniji za odabir komponenti (kontejneri, kontrole, meni, ostale regije, geometrijski oblici, dijagrami i 3D objekti). Po odabiru, traženi element treba prevući na centralni deo koji predstavlja vizuelni prikaz trenutnog izgleda scene. Sa desne strane nalaze se padajući meniji za odabir tipa podešavanja za označenu komponentu (osnovna podešavanja, podešavanja rasporeda/pozicije, podešavanja kontrolera). Prilikom prevlačenja i izmena podešavanja generiše se sadržaj u pripadajućoj **FXML** datoteci koji odgovara datom prikazu. Alat se pokreće tako što se po otvaranju **FXML** datoteke odabere kartica "Scene Builder" u donjem levom uglu. Za vraćanje na tekstualni režim bira se kartica "Text". U sledećem primeru

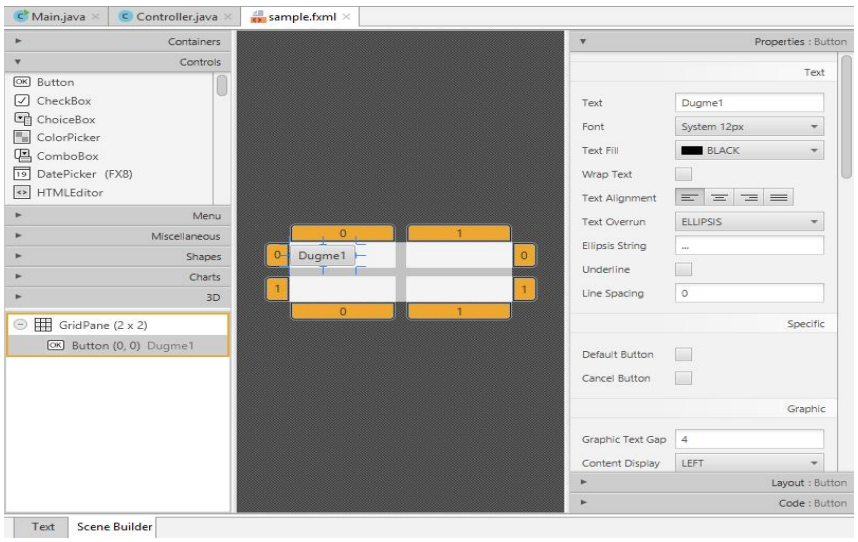
prikazan je izgled **Scene builder,** postupak dodavanja reda i kolone u okviru korenskog **GridPane** kontejnera, a zatim i dodavanje komponente u njemu. Nakon toga izvršiće se modifikacija podrazumevanih podešavanja za datu komponentu.



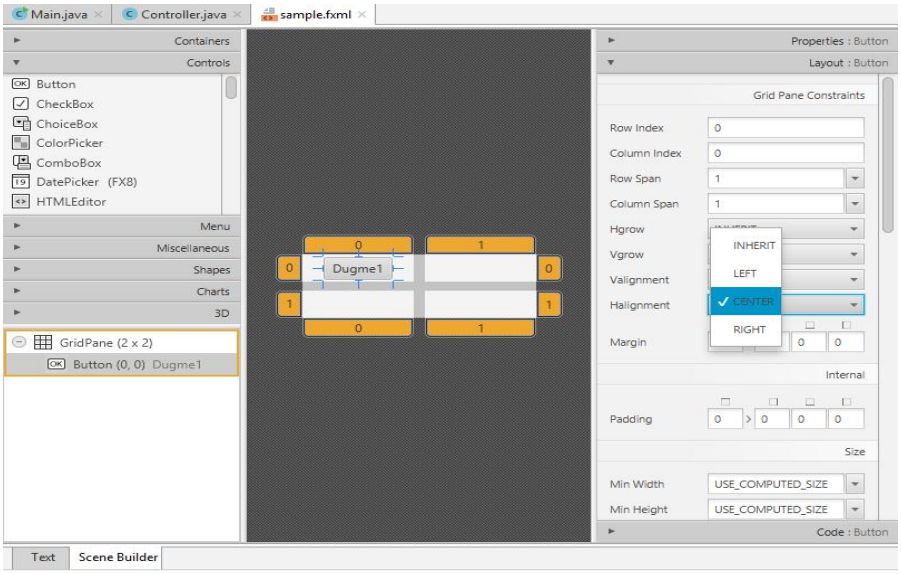
Slika 12.5- Dodavanje reda i kolone u okviru korenskog čvora



Slika 12.6- Dodavanje komponente (Button) prevlačenjem



Slika 12.7- Modifikacija osnovnih podešavanja



Slika 12.8- Podešavanje pozicije komponente

Pripadajući sadržaj u **sample.fxml** datoteci:

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.layout.ColumnConstraints?>
<?import javafx.scene.layout.GridPane?>
<?import javafx.scene.layout.RowConstraints?>
<GridPane alignment="center" hgap="10" vgap="10" xmlns:fx="http://javafx.com/fxml/1"
xmlns="http://javafx.com/javafx/8.0.121" fx:controller="sample.Controller">
   <rowConstraints>
      <RowConstraints minHeight="10.0" prefHeight="30.0" />
      <RowConstraints minHeight="10.0" prefHeight="30.0" />
   </rowConstraints>

   <columnConstraints>
      <ColumnConstraints minWidth="10.0" prefWidth="100.0" />

      <ColumnConstraints minWidth="10.0" prefWidth="100.0" />
   </columnConstraints>
   <children>
      <Button mnemonicParsing="false" text="Dugme1" GridPane.halignment="CENTER" />
```

```
    </children>
</GridPane>
```

Da bi korisnički interfejs dobio funkcionalnost tj. da bi korisnik mogao da vrši interakciju sa aplikacijom, na komponentama moraju biti postavljeni rukovaoci događaja (eng. event handler). Rukovaoci se postavljaju kako bi detektovali i obradili događaji (akcije). Događaji mogu biti uzrokovani od strane korisnika ili same platforme. Na osnovu toga postoje dva osnovna tipa događaja: događaji u prvom planu(eng. foreground) i pozadinski (eng. background). U prvu kategoriju spadaju događaji koji su izazvani od strane korisnika (klik na dugme, pomeraj miša, prevlačenje itd.), dok u drugu spadaju događaji izazvani od platforme (prekid - interapt od strane operativnog sistema, greška u softveru ili hardveru, završetak određene operacije itd.). Osnovna klasa koja opisuje događaj je klasa **Event.** Klasa se nalazi u paketu **javafx.event** modula **javafx.base**. U sledećoj listi nabrojane su češće korišćene klase događaja:

1. **ActionEvent –** Događaj koji može da predstavlja različite vrste akcija. Događaj može da ima više oblika i koristi se za široku predstavu akcija kao što je pritisak na dugme (**Button** objekat), klik na komande za odabir (**CheckBox, RadioButton)** itd.
2. **MouseEvent-** Ulazni događaj koji opisuje akcije kao što su: klik miša, prelazak miša, ulazak kursora na određenu regiju itd.
3. **KeyEvent-** Ulazni događaj koji opisuje akcije kao što su unos preko tastature, pritisak tastera na tastaturi, otpuštanje tastera itd.
4. **WindowEvent-** Ovaj događaj opisuje akcije kao što su prikazivanje/ skrivanje prozora.

Rukovalac događaja kontroliše događaj i odlučuje šta će se izvršiti kada se detektuje. Rukovalac može biti bilo koji objekat klase koja implementira interfejs **EventHandler.** Ovo je generički interfejs kojem je prilikom implementiranja potrebno proslediti i tip koji se odnosi na vrstu događaja sa kojim treba da rukuje (**ActionEvent, MouseEvent, KeyEvent** itd.). Interfejs poseduje jednu metodu koju je potrebno implementirati, metodu **handle().** Unutar ove metode definiše se scenario koji će se izvršiti kada se detektuje akcija događaja. Kreiranje rukovaoca može se realizovati korišćenjem: unutrašnje klase, zasebne klase, anonimne klase, ali i korišćenjem lambda izraza zato što poseduje samo jednu metodu koju treba implementirati. Postavljanje rukovaoca na komponente vrši se metodama koje imaju sledeću sintaksu **setOn***NazivAkcije(objekatRukovalac).* U sledećem primeru prikazani su različiti načini kreiranja rukovaoca, a zatim i postupak kojim se postavlja na date komponente.

Sadržaj **Main.java** datoteke:

```
package sample;

import javafx.application.Application;
import javafx.concurrent.Task;

import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.HPos;
import javafx.geometry.VPos;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.input.KeyEvent;
```

```
import javafx.scene.layout.ColumnConstraints;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.RowConstraints;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

import java.text.SimpleDateFormat;
import java.util.Date;

public class Main extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        // Parent root = FXMLLoader.load(getClass().getResource("sample.fxml"));
        GridPane root = new GridPane();
        RowConstraints prviRed = new RowConstraints();
        prviRed.setPercentHeight(40);
        RowConstraints drugiRed = new RowConstraints();
        drugiRed.setPercentHeight(15);
        RowConstraints treciRed = new RowConstraints();
        treciRed.setPercentHeight(15);
        RowConstraints cetvrtiRed = new RowConstraints();
        cetvrtiRed.setPercentHeight(15);
        RowConstraints petiRed = new RowConstraints();
        petiRed.setPercentHeight(15);
        root.getRowConstraints().addAll(prviRed, drugiRed, treciRed, cetvrtiRed,
petiRed);

        ColumnConstraints prvaKolona = new ColumnConstraints();
        prvaKolona.setPercentWidth(30);
        ColumnConstraints drugaKolona = new ColumnConstraints();
        drugaKolona.setPercentWidth(70);
        root.getColumnConstraints().addAll(prvaKolona, drugaKolona);

        Label lblVreme = new Label("Vreme??");
        lblVreme.setStyle("-fx-font-size: 20;");

        root.add(lblVreme, 0, 0);

        Label lblBrojac = new Label("Broj unetih reci:0");
        lblBrojac.setStyle("-fx-font-size: 20;");

        root.add(lblBrojac, 1, 0);

        TextField txtUnos = new TextField();
        txtUnos.setPromptText("Unesite teksualni sadrzaj");
        txtUnos.setFocusTraversable(false);
        root.add(txtUnos, 1, 1);

        Label lblStilFonta = new Label("Font:");

        lblStilFonta.setStyle("-fx-font-size: 15");
        GridPane.setValignment(lblStilFonta, VPos.TOP);
        GridPane.setHalignment(lblStilFonta, HPos.CENTER);
        root.add(lblStilFonta, 0, 2);

        HBox hbStil = new HBox(20);
        CheckBox cb1 = new CheckBox("Bold");
        CheckBox cb2 = new CheckBox("Italic");
        CheckBox cb3 = new CheckBox("Underline");
        hbStil.getChildren().addAll(cb1, cb2, cb3);
```

```
        root.add(hbStil, 1, 2);

        Label lblBoja = new Label("Boja:");
        lblBoja.setStyle("-fx-font-size: 15");
        GridPane.setHalignment(lblBoja, HPos.CENTER);
        root.add(lblBoja, 0, 3);

        ComboBox<String> cobBoja = new ComboBox<>();
        cobBoja.getItems().addAll("Plava", "Crvena", "Zelena");
        root.add(cobBoja, 1, 3);

        Button btnStart = new Button("Start vreme");
        btnStart.setStyle("-fx-font-size: 15");
        GridPane.setHalignment(btnStart, HPos.CENTER);
        root.add(btnStart, 0, 4, 2, 1);

        //postavljanje provere prilikom otpustanja dugmeta, kada polje txtUnos ima
fokus
        txtUnos.setOnKeyReleased(new EventHandler<KeyEvent>() {
            @Override
            public void handle(KeyEvent event) {
                if (txtUnos.getText().length() == 0)
                    lblBrojac.setText("Broj unetih reci:0");
                else
                    lblBrojac.setText("Broj unetih reci:" +
txtUnos.getText().trim().split("\\b* \\b").length);
            }
        });
        //skraceni oblik koriscenjem lambda izraza. e se odnosi na parametar handle()
metode
        /*
         txtUnos.setOnKeyReleased(e->{
              if (txtUnos.getText().length()==0)
                lblBrojac.setText("Broj unetih reci:0");
                else
                  lblBrojac.setText("Broj unetih
reci:"+txtUnos.getText().trim().split("\\b* \\b").length);
        });
         */

        //kreiranje rukovaoca, koriscenjem anonimne klasa, koji ce biti dodeljen
poljima za izbor stila fonta(CheckBox)
        EventHandler<ActionEvent> rukovalacZaCheckBox = new EventHandler<ActionEvent>()
{
            @Override
            public void handle(ActionEvent event) {
                String boldItalic = "";
                String underline = "false";
                if (cb1.isSelected())
                    boldItalic += " bold ";
                if (cb2.isSelected())
                    boldItalic += " italic ";
                if (cb3.isSelected())
                    underline = "true";
                lblBrojac.setStyle("-fx-font:" + boldItalic + " 20 'System';-fx-
underline: " + underline + ";");
                lblVreme.setStyle("-fx-font:" + boldItalic + " 20 'System';-fx-
underline: " + underline + ";");
            }
        };
        cb1.setOnAction(rukovalacZaCheckBox);
```

```
        cb2.setOnAction(rukovalacZaCheckBox);
        cb3.setOnAction(rukovalacZaCheckBox);

        //postavljanje rukovaoca na ComboBox
        cobBoja.setOnAction(e -> {
            String boja = "-fx-text-fill: ";
            switch (cobBoja.getValue()) {
                case "Plava":
                    lblVreme.setTextFill(Color.BLUE);
                    lblBrojac.setTextFill(Color.BLUE);
                    break;
                case "Crvena":
                    lblVreme.setTextFill(Color.RED);
                    lblBrojac.setTextFill(Color.RED);
                    break;
                case "Zelena":
                    lblVreme.setTextFill(Color.GREEN);
                    lblBrojac.setTextFill(Color.GREEN);
                    break;
            }

        });
        btnStart.setOnAction(e -> {
            /*kreiranje objekta pomocu anononimne klase koja prosiruje apstraktnu klasu
Task<T>.
            Pomocu ove klase moguce je definisati poslove za konkurentno izvrsavanje u
JavaFx.
            Prikaz u labeli lblPrikaz sadrzace trenutno vreme koje ce se osvezavati na
svakih 1 sek.
            */
            Task t1 = new Task<Void>() {
                @Override
                protected Void call() throws Exception {
                    SimpleDateFormat sdf = new SimpleDateFormat("HH:mm:ss");
                    while (true) {
                        updateMessage(sdf.format(new Date()));
                        try {
                            Thread.sleep(100);
                        } catch (InterruptedException ex) {
                            break;
                        }
                    }
                    return null;
                }
            };
            lblVreme.textProperty().bind(t1.messageProperty());
            Thread t2 = new Thread(t1);
            t2.setDaemon(true);
            t2.start();
            ((Button) e.getSource()).setDisable(true);
        });

        //dodavanje rukovaoca dogadjaja na korenski cvor. Boja pozadine generisace se
na osnovu trenutne pozicije misa.
        root.setOnMouseMoved(e -> {
            double x = e.getX();//uzimanje pozicije misa(x osa).
            double y = e.getY();//uzimanje pozicije misa(y osa).
            String rgb = "rgb(" + x % 255 + "," + y % 255 + "," + (x + y) % 255 + ")";
            root.setStyle("-fx-background-color: " + rgb);
        });
        primaryStage.setTitle("Laboratorijska vezba 12");
```

```
        primaryStage.setScene(new Scene(root));
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Na slici 12.10 prikazan je rezultujući izgled prozora.

Slika 12.10- Izgled prozora

Prethodni primer može se realizovati i korišćenjem **FXML** tehnologije. Kako je izgled odvojen od funkcionalnosti potrebno je učitati neophodne objekte u kontroler klasi. Učitavanje se vrši tako što se kreiraju reference čija imena odgovaraju vrednosti **fx:id** atributa elementa u **FXML** datoteci. Tip reference u kontroler klasi i tip elementa u **FXML** dokumentu moraju se poklapati. Pre deklaracije referenci potrebno je dodati anotaciju **@FXML** kako bi se naglasilo da se kreiranje objekta vrši na osnovu sadržaja iz datoteke**.** Dodela rukovaoca događaja vrši se pomoću atributa koji imaju prefiks "on". Nakon prefiksa sledi ime akcije na koju komponenta treba da reaguje. Vrednost ovog atributa treba da odgovara nekom od imena metoda koja je definisana u kontroler klasi. Prilikom navođenja imena potrebno je navesti i prefiks znak '#'. Povezivanje sa rukovaocima može se izvršiti i pomoću Scene Builder alata u okviru **Code** podešavanja.

Sadržaj **sample.fxml** datoteke:

```
<?xml version="1.0" encoding="UTF-8"?>

<?import java.lang.String?>
<?import javafx.collections.FXCollections?>
<?import javafx.scene.control.CheckBox?>
<?import javafx.scene.control.ComboBox?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.layout.ColumnConstraints?>
<?import javafx.scene.layout.GridPane?>
<?import javafx.scene.layout.HBox?>
<?import javafx.scene.layout.RowConstraints?>

<?import javafx.scene.control.Button?>
```

```
<GridPane fx:id="korenskiCvor" onMouseMoved="#rukovalacBojaPozadine" alignment="center"
hgap="10" vgap="10"
          xmlns:fx="http://javafx.com/fxml/1" xmlns="http://javafx.com/javafx/8.0.121"
          fx:controller="sample.Controller">
    <rowConstraints>
        <RowConstraints percentHeight="40"/>
        <RowConstraints percentHeight="15"/>
        <RowConstraints percentHeight="15"/>
        <RowConstraints percentHeight="15"/>
        <RowConstraints percentHeight="15"/>
    </rowConstraints>
    <columnConstraints>
        <ColumnConstraints percentWidth="30"/>
        <ColumnConstraints percentWidth="70"/>
    </columnConstraints>

    <children>
        <Label fx:id="lblVreme" style="-fx-font-size: 20" text="Vreme??"
GridPane.columnIndex="0"
               GridPane.rowIndex="0"/>
        <Label fx:id="lblBrojac" style="-fx-font-size: 20" text="Broj unetih reci:0"
GridPane.columnIndex="1"
               GridPane.rowIndex="0"/>
        <TextField fx:id="txtUnos" onKeyReleased="#rukovalacBrojac" promptText="Unesite
tekstualni sadrzaj"
                   GridPane.columnIndex="1" GridPane.rowIndex="1"/>
        <Label style="-fx-font-size: 15;" text="Font" GridPane.columnIndex="0"
GridPane.halignment="CENTER"
               GridPane.rowIndex="2" GridPane.valignment="TOP"/>
        <HBox GridPane.columnIndex="1" GridPane.rowIndex="2" spacing="20">
            <children>
                <CheckBox onAction="#rukovalacStil" fx:id="cb1" text="Bold"/>
                <CheckBox onAction="#rukovalacStil" fx:id="cb2" text="Italic"/>
                <CheckBox onAction="#rukovalacStil" fx:id="cb3" text="Underline"/>
            </children>
        </HBox>

        <Label text="Boja" GridPane.columnIndex="0" GridPane.rowIndex="3"
GridPane.halignment="CENTER"

               style="-fx-font-size: 15"/>
        <ComboBox onAction="#rukovalacBoja" fx:id="cobBoja" GridPane.columnIndex="1"
GridPane.rowIndex="3">
            <items>
                <FXCollections fx:factory="observableArrayList">
                    <String fx:value="Crvena"/>
                    <String fx:value="Plava"/>
                    <String fx:value="Zelena"/>
                </FXCollections>
            </items>
        </ComboBox>
        <Button text="Start vreme" onAction="#rukovalacVrmeStart" style="-fx-font-size:
15" GridPane.halignment="CENTER"
                GridPane.columnIndex="0" GridPane.rowIndex="4"
GridPane.columnSpan="2"/>
    </children>
</GridPane>
```

Sadržaj **Controller.java** datoteke:

```
package sample;

import javafx.concurrent.Task;
```

```
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.*;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.GridPane;
import javafx.scene.paint.Color;

import java.text.SimpleDateFormat;
import java.util.Date;

public class Controller {
    @FXML
    GridPane korenskiCvor;
    @FXML
    Label lblVreme;
    @FXML
    Label lblBrojac;
    @FXML
    TextField txtUnos;
    @FXML
    CheckBox cb1;
    @FXML
    CheckBox cb2;
    @FXML
    CheckBox cb3;
    @FXML
    ComboBox<String> cobBoja;

    public void rukovalacBrojac() {
        if (txtUnos.getText().length() == 0)
            lblBrojac.setText("Broj unetih reci:0");
        else
            lblBrojac.setText("Broj unetih reci:" +
txtUnos.getText().trim().split("\\b* \\b").length);
    }

    public void rukovalacStil() {
        String boldItalic = "";
        String underline = "false";
        if (cb1.isSelected())
            boldItalic += " bold ";
        if (cb2.isSelected())
            boldItalic += " italic ";
        if (cb3.isSelected())
            underline = "true";
        lblBrojac.setStyle("-fx-font:" + boldItalic + " 20 'System';-fx-underline: " +
underline + ";");
        lblVreme.setStyle("-fx-font:" + boldItalic + " 20 'System';-fx-underline: " +
underline + ";");
    }

    public void rukovalacBoja() {
        String boja = "-fx-text-fill: ";
        switch (cobBoja.getValue()) {
            case "Plava":
                lblVreme.setTextFill(Color.BLUE);
                lblBrojac.setTextFill(Color.BLUE);
                break;
            case "Crvena":
                lblVreme.setTextFill(Color.RED);
                lblBrojac.setTextFill(Color.RED);
```

```
                break;
            case "Zelena":
                lblVreme.setTextFill(Color.GREEN);
                lblBrojac.setTextFill(Color.GREEN);
                break;
        }
    }

    //ukoliko je potrebno metodi se moze proslediti Event objekat
    public void rukovalacVrmeStart(ActionEvent e) {
        Task t1 = new Task<Void>() {
            @Override
            protected Void call() throws Exception {
                SimpleDateFormat sdf = new SimpleDateFormat("HH:mm:ss");
                while (true) {
                    updateMessage(sdf.format(new Date()));
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException ex) {
                        break;
                    }
                }
                return null;
            }
        };
        lblVreme.textProperty().bind(t1.messageProperty());
        Thread t2 = new Thread(t1);
        t2.setDaemon(true);
        t2.start();
        ((Button) e.getSource()).setDisable(true);
    }

    public void rukovalacBojaPozadine(MouseEvent e) {
        double x = e.getX();//uzimanje pozicije misa(x osa).
        double y = e.getY();//uzimanje pozicije misa(y osa).
        String rgb = "rgb(" + x % 255 + "," + y % 255 + "," + (x + y) % 255 + ")";
        korenskiCvor.setStyle("-fx-background-color: " + rgb);
    }

}
```

Sadržaj **Main.java** datoteke:

```
package sample;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class Main extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        Parent root = FXMLLoader.load(getClass().getResource("sample.fxml"));
        primaryStage.setTitle("Laboratorijska vezba 12");
        primaryStage.setScene(new Scene(root));
        primaryStage.show();
    }
```

```
    public static void main(String[] args) {
        launch(args);
    }
```

Zadatak za samostalan rad.

Kreirati formu koja je rađena na prethodnoj vežbi korišćenjem **FXML** tehnologije. Dodati funkcionalnosti tako da:

- Se vrši provera pri unosu imena i prezimena. Ukoliko korisnik unese karakter koji nije dozvoljen za ime ili prezime (!, # , % ,$, ^ itd.) okvir tog polja treba da poprimi crvenu boju.
- Proverava da li uneta vrednost za email adresu odgovara tom formatu.
- Kreirati još jedno polje za unos lozinke u kojoj treba potvrditi lozinku. Sve dok se lozinke ne poklope ispisivati poruku da se uneta druga vrednost ne podudara sa prvom.
- Prilikom potvrde (dugme “Potvrda”) proveriti da li su sva polja uneta i ukoliko jesu sve podatke treba ispisati u okviru nove scene. Ukoliko neko od polja nije popunjeno ili je nevalidno, obeležiti ga crvenom bojom.

Literatura

[1] Herbert Schildt : "*Java kompletan priručnik"*, prevod desetog izdanja, Mikroknjiga, Beograd, 2018.

[2] Dr. Edward Lavieri, Peter Verhas : "*Java 9"*, prevod prvog izdanja, Kompjuter biblioteka, Mikroknjiga, Beograd, 2018.

[3] Yakov Fain : "*Java 8 programiranje*", Kompjuter biblioteka, Beograd, 2015.

[4] Laslo Kraus : "*Rešeni zadaci iz programskog jezika JAVA JSE 8*", Akademska misao, Beograd, 2015.

[5] Laslo Kraus : "*Programski jezik Java sa rešenim zadacima JSE 8*", Akademska misao, Beograd, 2015.

[6] Bruce Eckel : "*Misliti na Javi"*, prevod 4. izdanja, Mikroknjiga, Beograd, 2007.

[7] Ivor Horton,"*Ivor Horton's Beginning Java™*", 2 JDK™ 5 Edition, Wiley Publishing, Inc, 2005.

[8] Branko Milosavljević, Vidaković M. : "*Java i Internet programiranje*", GInT, Novi Sad, 2002.

[9] Elliotte Rusty Harold : "*Java Network Programming, 3rd Edition*", O'Reilly Media, 2004.

[10] Herbert Schildt : "*Java The Complete Reference, 8th Edition*", Oracle Press, 2011.

[11] https://docs.oracle.com/javase/tutorial/ : "*The Java™ Tutorials*", Oracle, 2016.

[12] http://www.w3schools.com/, World Wide Web Consortium.

[13] https://docs.oracle.com/javafx/2/ , JavaFX, Oracle, 2018.

[14] Habraken, J. : "*Osnove umrežavanja*", CET, Beograd, 2002.

[15] Joshua Bloch : "*Effective Java™, Second Edition*", Prentice Hall, 2008.