

1.	Implementiranje klasa koje utjelovljuju entitete korištene u projektnom zadatku. Svaka klasa mora biti smještena u paket s klasama koje imaju zajednička svojstva (npr. entiteti moraju biti u jednom paketu, a glavna klasa za pokretanje aplikacije u drugom paketu).
----	---

Svi entiteti se nalaze u paketu „entiteti“.

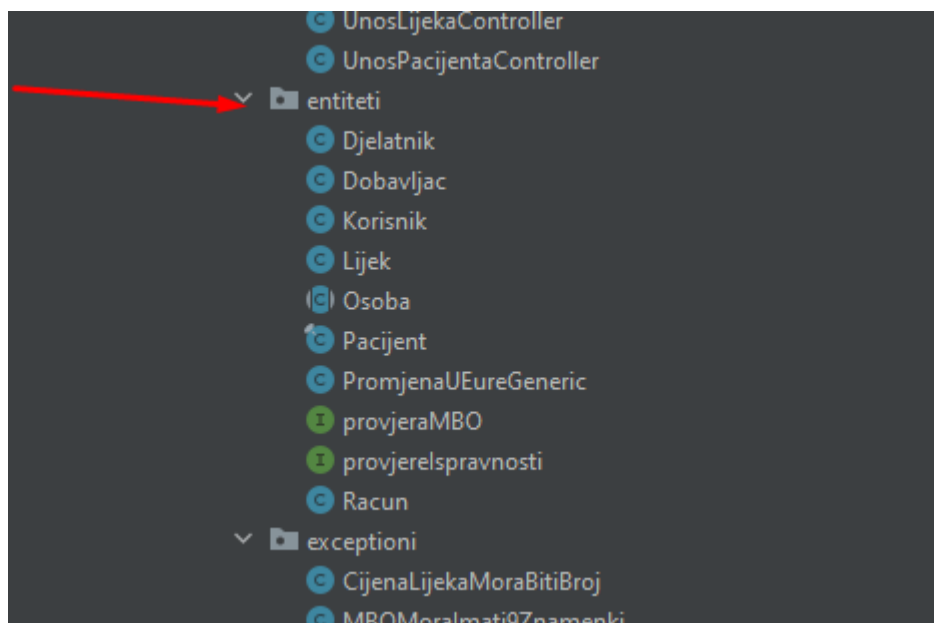
Klase: Djelatnik, Dobavljac, Korisnik, Lijek, Pacijent, Racun

Apstraktna klasa: Osoba

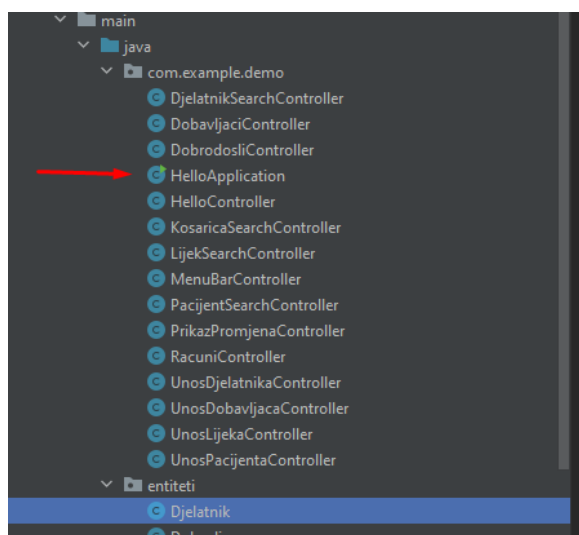
Generička klasa: PromjenaUEureGeneric

Interface: provjerelspravnosti

Sealed interface: provjeraMBO



Za pokretanje koristimo klasu HelloApplication. Desni klik i run()

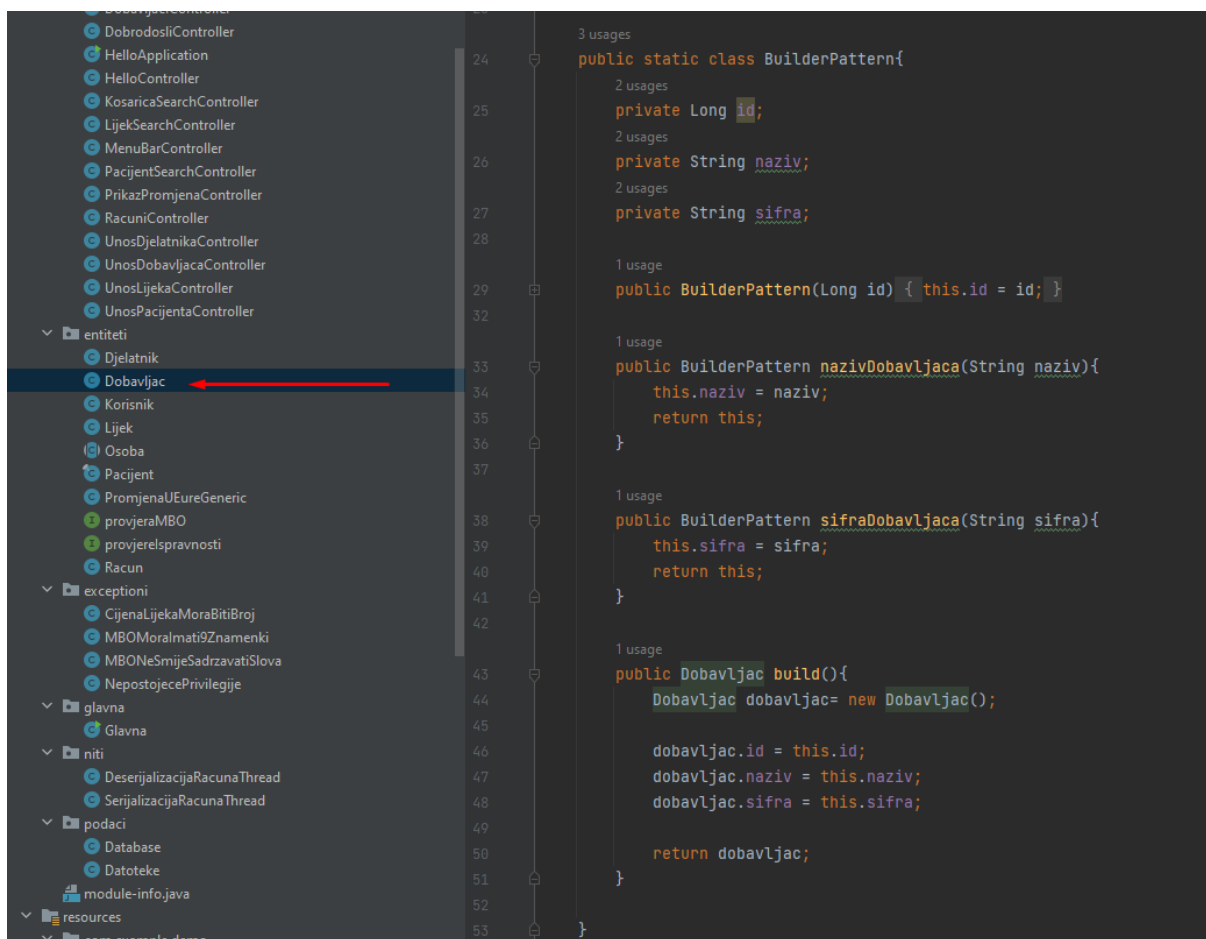


Također možemo kod pokrenuti sa klasom „Glavna“. Kada mu prezentiramo program onda ga nećemo tako pokretati (nego preko HelloApplication), ali je dobro poslužiti o da u toj klasi stavimo neke stvari koje su nam falile u kodu (npr. Mapa koji nismo imali gdje za staviti)

2.	Korištenje apstraktnih klasa, sučelja, zapisa, zapečaćenih sučelja te „builder pattern“ oblikovnog obrasca kako bi se iskoristile sve objektno orijentirane paradigme programskog jezika Java.
----	--

U entitetu se nalazi apstraktna klasa, sučelje (interface)

Builder Pattern smo stavili u klasu Dobavljac (jer ta klasa ima najmanje podataka pa je bilo najbrže za napraviti ga tamo)



Ako pita čemu služi Builder Pattern?

Kada radimo objekt uvijek mu unosimo sve podatke preko konstruktora

Npr: Dobavljac nekolme = new Dobavljac(1, "nazivDobavljava", "sif02"). Onda nam objekt dobavljava ima ta 3 podatka. Ali što ako ne želimo sva 3 podatka staviti odmah na objekt? Pomoću Builder Patterna možemo tom objektu dodati podatke malo po malo (samo one koji nam trebaju).

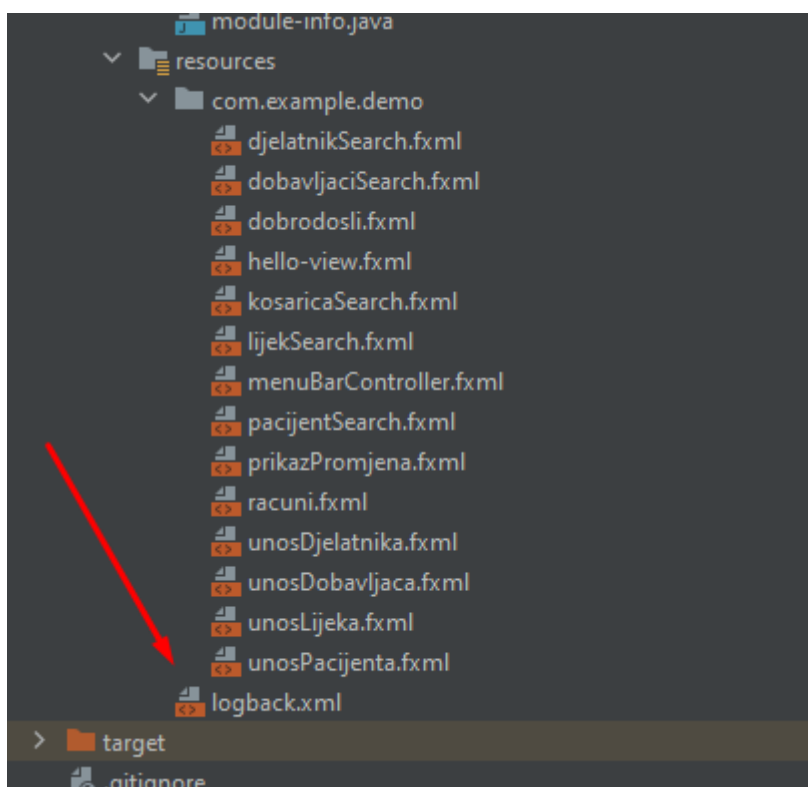
Npr: Dobavljac nekolme = new

Dobavljac.BuilderPattern(1).nazivDobavljava("nazivDobavljava").build();

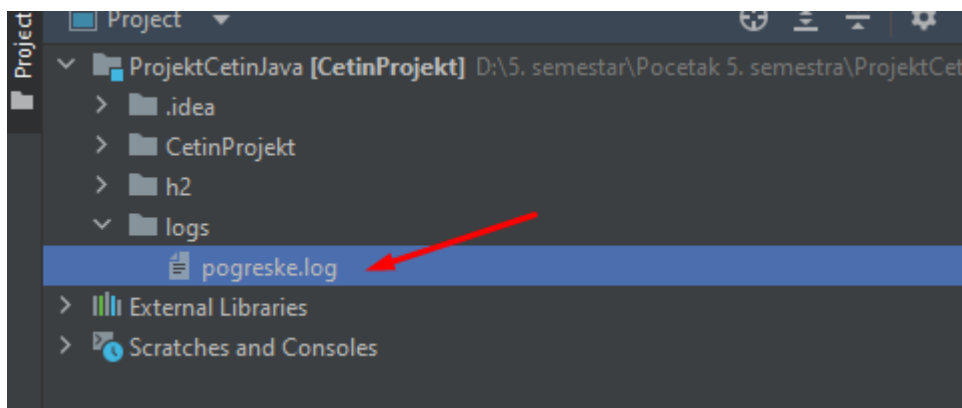
Tako nam sada na ovom primjeru objekt dobavljača ima samo podatke id i naziv. Kasnije mozemo dodati sifru kada zelimo.

3.	Hvatanje i bacanje iznimaka na svim mjestima u programu gdje se mogu dogoditi. Svaka iznimka se mora logirati korištenjem Logback biblioteke. Osim toga je potrebno kreirati barem dvije označene i dvije neoznačene iznimke te ih bacati i hvatati u programskom kodu aplikacije te logirati korištenjem Logback biblioteke. Klase iznimaka moraju biti smještene u zaseban paket.
----	---

Logback biblioteka je napravljena ovdje:



Ona osigurava da se sve što se zapise sa loggerom spremi u neku datoteku. (U nasem slucaju sprema se u datoteku pogreske.log)

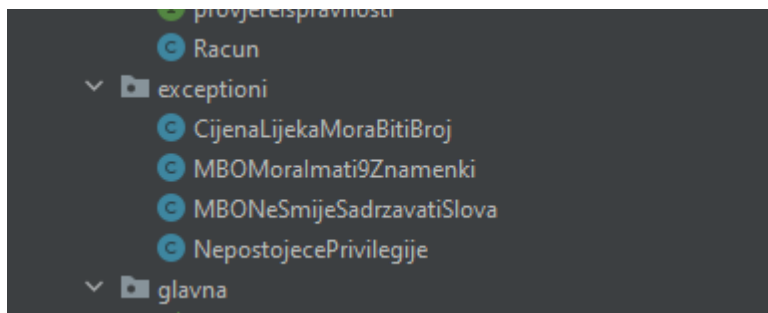


Logger nam služi da sve greške koje se dogode zapisujemo u neku našu datoteku umjesto da se ispisuju korisnicima u aplikaciji. Npr. ako korisnik napravi neku grešku i dogodi se Exception, nas program mora i dalje nastaviti raditi (ali moramo negdje zapisati da se ta greška dogodila)

```
}  
else{  
    HelloController.logger.error("Korisnik je napravio sljedece greske: "+korisnikovePogreske);  
  
    Alert alert = new Alert(Alert.AlertType.ERROR);  
    alert.setTitle("Greska kod spremanja dobavljacka");  
    alert.setHeaderText("Napravili ste gresku!");  
    alert.setContentText(korisnikovePogreske.toString());  
  
    alert.showAndWait();  
}
```

Morali smo loggeru dodati „HelloController“ ispred jer se logger nalazi u HelloController-u. Pomoću `logger.error(poruka)` zapisujemo to god zelimo u logger (odnosno sve greske koje su se dogodile)

Piše u zadatku da smo morali napraviti 4 exceptiona (2 označene i 2 neoznačene iznimke). Exceptioni nam se nalaze u paketu „exceptioni“



Označene iznimke: MBONeSmijeSadrzavatiSlova i MBOMoraimati9Znamenki

Neoznačene iznimke: CijenaLijekaMoraBitiBroj i NepostojecePrivilegije

Razlika je u tome što se označene iznimke moraju odmah riješavati.

Razlika je u tome što označene u definiciji imaju „extends Exception“, a neoznačene imaju „RuntimeException“

U našem kodu smo napravili da se prilikom unošenja novog Pacijenta provjerava MBO koji smo unijeli. Ako MBO ima više ili manje od 9 znakova onda se baca exception „MBOMoraimati9Znamenki“.

Ako MBO sadrži slovo onda se baca iznimka „MBONeSmijeSadrzavatiSlova“.

U kodu imamo interface „provjeraMBO“ koji ima metodu „provjeraIspravnostiMBO“ i taj interface je sealed i permita samo „Pacijent“ klasu (odnosno samo Pacijent klasa ga može koristiti)

```
1 usage 1 implementation
public sealed interface provjeraMBO permits Pacijent{

    1 usage 1 implementation
    void provjeraIspravnostuMBO(String mbo) throws MBOMoraImati9Znamenki, MBONeSmijeSadrzavatiSlova;
}
```

Pošto smo implementirali interface „provjeraMBO“ u Pacijentu, morali smo definirati što će ta metoda raditi.

```
69
70 @Override
71 public void provjeraIspravnostuMBO(String mbo) throws MBOMoraImati9Znamenki, MBONeSmijeSadrzavatiSlova {
72     if(mbo.length() != 9){
73         throw new MBOMoraImati9Znamenki("Mbo mora imati 9 znamenki!");
74     }
75
76     char[] znamenkeMBO = mbo.toCharArray();
77
78     for(int i=0; i<znamenkeMBO.length; i++) {
79         if(!Character.isDigit(znamenkeMBO[i])){
80             throw new MBONeSmijeSadrzavatiSlova("Mbo ne smije sadrzavati slova!");
81         }
82     }
83 }
84
85
86
87 }
```

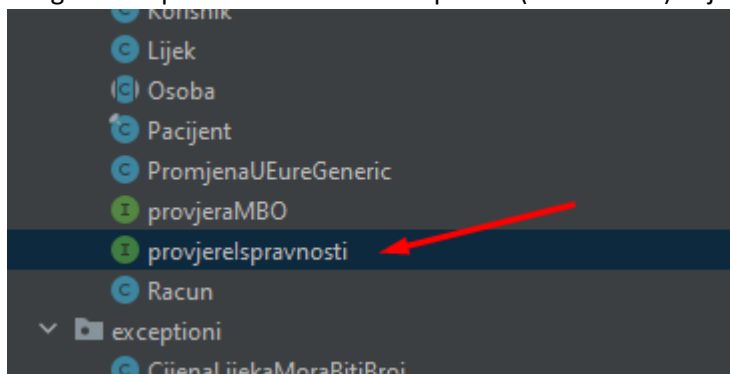
Ta metoda će pregledavati uneseni MBO i baciti Exceptione koje smo mi napravili u slučaju da ne zadovoljavaju određene uvijete (9 znakova i samo brojevi)

```
46
47 String mboPacijentaText=mboPacijenta.getText();
48 if(mboPacijentaText.isEmpty()){
49     korisnikovePogreske.append("Mbo pacijenta ne smije biti prazno!\n");
50 }else{
51     try {
52         Pacijent objekt = new Pacijent();
53         objekt.provjeraIspravnostuMBO(mboPacijentaText);
54     } catch (MBOMoraImati9Znamenki | MBONeSmijeSadrzavatiSlova e) {
55         HelloController.logger.error("Korisniku se desio exception: "+e.getLocalizedMessage());
56         korisnikovePogreske.append(e.getMessage()+"\n");
57     }
58 }
59 }
```

Ako je unesen MBO, on se šalje u tu metodu koja ga provjerava i sprema se hvatati ta 2 Exceptiona u „catch“ bloku. Ako se dogodi iti jedan, ne smijemo dozvoliti korisniku da unese Pacijenta sve dok ne ispravi tu pogresku.

Pogreška se također zapisuje u logger (jer se svi exceptioni moraju zapisivati u logger)

Druga 2 exceptiona su RuntimeException-i (neoznačeni) i njih smo implementirali na sljedeći način:



Napravili smo interface u kojem smo napravili metodu „provjerelspravnosti“. Njega može implementirati bilo koja klasa, u našem slučaju implementira ga klasa „Lijek“.

Nakon implementacije, Lijek toj metodi „provjerelspravnosti“ pridodaje sljedeći kod:

```
1 usage
117      @Override
118      public void provjeraIspravnosti(String cijena) {
119          char[] znamenkeCijene = cijena.toCharArray();
120
121          for(int i=0; i<znamenkeCijene.length; i++){
122              if(!Character.isDigit(znamenkeCijene[i])){
123                  throw new CijenaLijekaMoraBitiBroj("Cijena mora biti broj!");
124              }
125          }
126      }
127  }
128
```

Nakon što unesemo cijenu lijeka, ako smo unijeli neko slovo umjesto broja, baca se exception „CijenaLijekaMoraBitiBroj“. Radi identično kao i provjera MBO-a.

Zadnji exception smo napravili u „MenuBarController“ klasi. U toj klasi se mijenjaju sve Scene na Stage-u. Napravili smo exception „NepostojecaPrivilegije“. I implementirali smo taj exception na sljedeći način:

Ako se ulogirao pacijent (a ne admin) onda on nema pravo dodavati nove Lijekove, Djelatnike, Dobavljače i Pacijente u ljekarnu. U tom slučaju se baca taj exception i zabranjuje nam otići na taj zaslon.

```

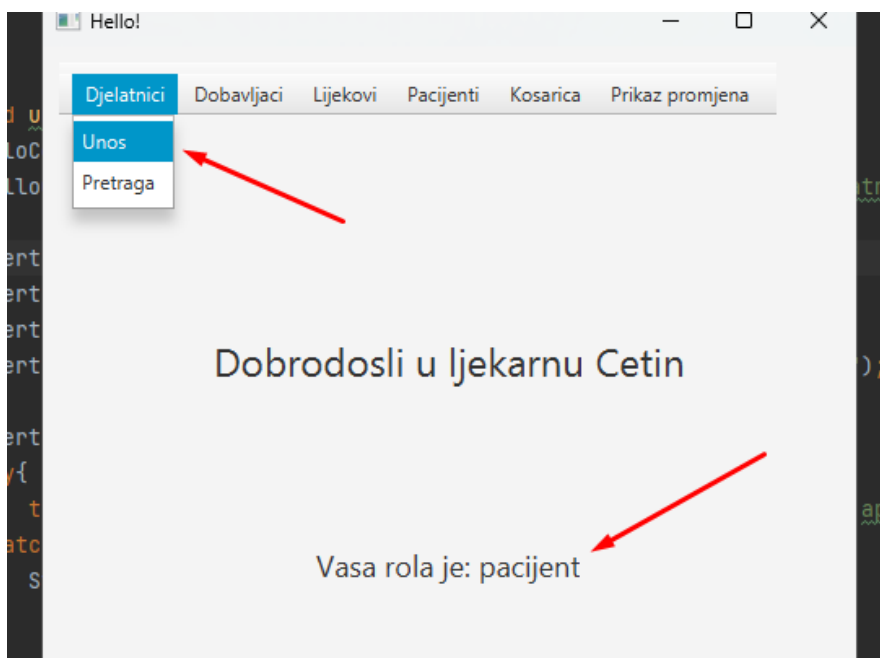
1 usage
@FXML
public void unosDjelatnika() throws IOException {
    if(HelloController.userRole.equals("pacijent")){
        HelloController.logger.error("Pacijent je pokusao dodati novog djelatnika!");

        Alert alert = new Alert(Alert.AlertType.ERROR);
        alert.setTitle("Greska");
        alert.setHeaderText("Pogreska!");
        alert.setContentText("Nemate privilegiju dodavanja novog djelatnika");

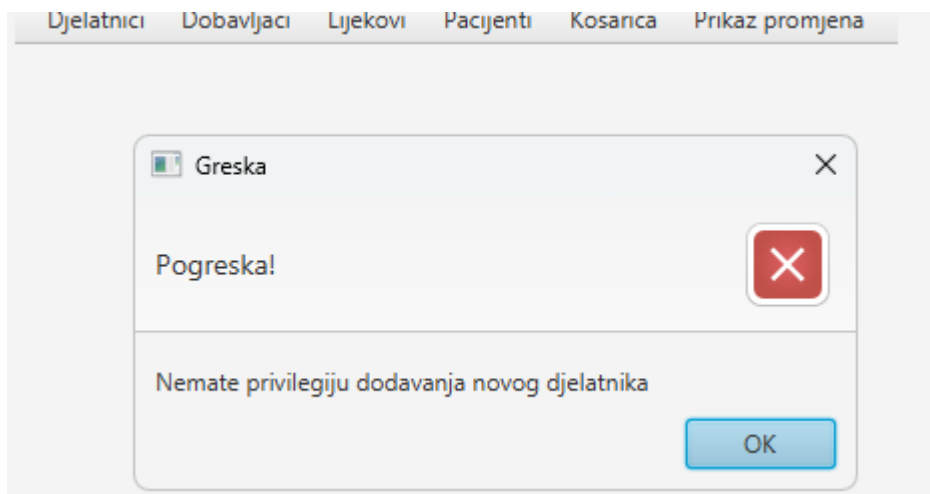
        alert.showAndWait();
        try{
            throw new NepostojecaPrivilegije("Nemate privilegije mijenjanja aplikacije!");
        }catch (NepostojecaPrivilegije e){
            System.out.println("Nemate privilegije");
        }
    }else{

```

Ovdje je mala greška jer hvatamo exception odma cim smo ga bacili (U pravilu se ne radi tako, ali je normalna amaterska pogreska)



Ako pokusamo unijeti neki novi podatak u aplikaciju a rola nam je „pacijent“



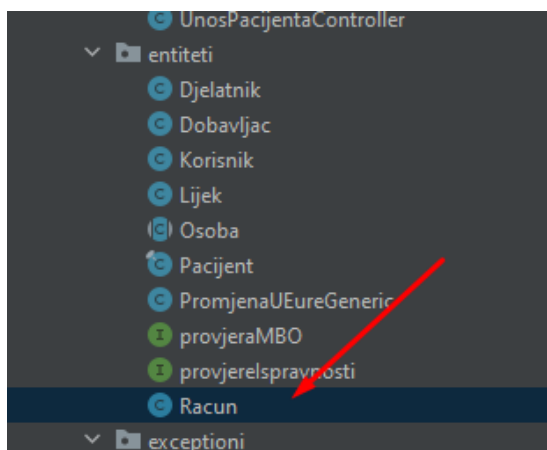
4. Korištenje zbirki iz tipa lista, setova i mapa, uz korištenje lambda izraza za filtriranje i sortiranje svih entiteta u aplikaciji.

Liste koristimo za sve entitet

```
1 public class HelloController {
2
3     public static List<Djelatnik> djelatnici;
4     public static List<Dobavljac> dobavljac;
5     public static List<Lijek> lijekovi;
6     public static List<Pacijent> pacijenti;
7     public static List<Racun> racuni = new ArrayList<>();
8
9     public static List<Korisnik> korisnici;
10
11     public static final Logger logger = LoggerFactory.getLogger(HelloController.class);
12 }
```

U njih spremamo sve podatke koji se citaju iz baze i prikazuju na aplikaciji.

U klasi Racun smo napravili Set<Lijek> (set lijekova) tek tolko da ga negdje iskoristimo. Radi isto kao i Lista samo što ne podržava duplikate. (Ovdje smo namjerno napravili malu grešku, jer bi zapravo smjeli moći imati duplikate u računu npr. za više istih lijekova, ali takve greške su normalne i dešavaju se svima pa nije očito.)



```
15 usages
public class Racun implements Serializable {
    @Serial
    private static final long serialVersionUID = 8510517524520303549L;

    public Set<Lijek> lijekovi;
    public String pacijent;
    public LocalDateTime datumIVrijemeKupovine;
}
```


Pošto ne znam di bi stavili mapu, Mapa je stavljena u klasu „Glavna“. U mapi nam se nalaze svi Dobavljac i lista njihovih lijekova. Znači svaki dobavljač ima listu lijekova koje oni dostavljaju.

Npr: Dobavljac1 – Andol, Brufen

Dobavljac2 – Lekadol....

```
12 public class Glavna {
13     public static void main(String[] args) {
14         Scanner input = new Scanner(System.in);
15
16         Djelatnik[] nizDjelatnika=new Djelatnik[3];
17         Dobavljac[] nizDobavljacka=new Dobavljac[3];
18         Lijek[] nizLijekova=new Lijek[5];
19         Pacijent[] nizPacijenata=new Pacijent[3];
20
21         Map<Dobavljac, List<Lijek>> mapaDobavljacka= new HashMap<>();
22
23     }
```

```
43
44
45     for(Lijek lijek:nizLijekova){
46         if(mapaDobavljacka.containsKey(lijek.getDobavljac())){
47             List<Lijek> lijekovi = mapaDobavljacka.get(lijek.getDobavljac());
48             lijekovi.add(lijek);
49             mapaDobavljacka.put(lijek.getDobavljac(),lijekovi);
50         }else{
51             List<Lijek> listaNovihLijekova = new ArrayList<>();
52             listaNovihLijekova.add(lijek);
53             mapaDobavljacka.put(lijek.getDobavljac(),listaNovihLijekova);
54         }
55     }
56 }
```

Píše da se koriste lambda izrazi za filtriranje i sortiranje u aplikaciji.

```
com.example.demo
├── DjelatnikSearchController
├── DobavljackiController
├── DobrodosliController
├── HelloApplication
├── HelloController
├── KosaricaSearchController
├── LijekSearchController
├── MenuBarController
├── PacijentSearchController
├── PrikazPromjenaController
├── RacuniController
├── UnosDjelatnikaController
├── UnosDobavljackaController
└── UnosLijekaController
```

Lambde se nalaze u svim klasama koje imaju „SearchController“ u imenu osim KosaricaSearchController.

Izgledaju ovako:

```
98 @FXML
99 public void gumbPretrazi(){
100     String naziv= nazivLijekaTextField.getText();
101     String sifra = sifraLijekaTextField.getText();
102     String djelatnaIvar= djelatnaIvarLijekaTextField.getText();
103     String dobavljac = dobavljacLijekaTextField.getText();
104     String cijena = cijenaLijekaTextField.getText();
105
106
107     List<Lijek> lijekovi = HelloController.getLijekovi();
108
109     if(Optional.of(naziv).isEmpty() == false){
110         lijekovi = lijekovi.stream().filter(s->s.getNaziv().toLowerCase().contains(naziv.toLowerCase())).collect(Collectors.toList());
111     }
112
113     if(Optional.of(sifra).isEmpty() == false){
114         lijekovi = lijekovi.stream().filter(s->s.getSifra().toLowerCase().contains(sifra.toLowerCase())).collect(Collectors.toList());
115     }
116
117     if(Optional.of(djelatnaIvar).isEmpty() == false){
118         lijekovi = lijekovi.stream().filter(s->s.getDjelatnaIvar().toLowerCase().contains(djelatnaIvar.toLowerCase())).collect(Collectors.toList());
119     }
120
121     if(Optional.of(dobavljac).isEmpty() == false){
122         lijekovi = lijekovi.stream().filter(s->s.getDobavljac().getNaziv().toLowerCase().contains(dobavljac.toLowerCase())).collect(Collectors.toList());
123     }
124 }
```

Kada unesemo neki podatak po kojem želimo filtrirati podatke iz tablice, ti podaci se spremaju u varijable i prema njima se filtriraju podaci u tablici.

Npr:

Djelatnici Dobavljac Lijekovi Pacijenti Kosarica Prikaz promjena

Djelatnici

Ime: He

Prezime:

Sifra:

Funkcija:

Pretrazi

Id	Ime	Prezime	Sifra	Funkcija
1	Helena	Cetin	D01	Magistar farmacije

JavaFX je dodatno su objašnjena u PDF-u kojeg smo napravili za 6. ili 7. labos. Radili smo identičnu stvar kao i u tim priprema.

5.	Korištenje barem dvije generičke klase u aplikaciji koje su smještene u paket zajedno s entitetima. Jedna klasa mora imati samo jedan parametar, a druga klasa mora imati dva parametra generičkog tipa.
----	--

Pošto nikome nije ni gledao generičke klase, u našoj aplikaciji se nalazi samo jedna, a ne dvije. Pošto nam aplikacija stvarno ne treba generičke klase i nemamo ih razloga raditi.

Naša generična klasa se zove „PromjenaUEureGeneric“. To je klasa koja prima bilo koji tip podatka (npr. Integer, Float, Double, String)... I pretvara ih sve u float. Zatim ih zbraja i dijeli sa 7.5345 kako bi dobili cijenu u eurima.

```

3 usages
public class PromjenaUEureGeneric<T> {

    3 usages
    private List<T> cijene = new ArrayList<>();

    1 usage
    public void dodajCijenu(T cijena) { cijene.add(cijena); }

    1 usage
    public Float izracunajUEurima(){
        Float cijena=0F;

        for(int i=0;i<cijene.size();i++){
            cijena = cijena+ (float)cijene.get(i);
        }

        return cijena / 7.5345F;
    }
}

```

Implementira se u klasi „KosaricaSearchController“

```

79
80
81
82
83
84
85
86
87
88
89
90
    Float cijenaSvihLijekova = 0F;
    for(int i=0;i<odabraniLijekovi.size();i++){
        cijenaSvihLijekova = cijenaSvihLijekova+odabraniLijekovi.get(i).getCijena();
    }

    PromjenaUEureGeneric objekt = new PromjenaUEureGeneric();
    objekt.dodajCijenu(cijenaSvihLijekova);

    ukupnaCijenaLabel.setText("Ukupna cijena: "+cijenaSvihLijekova.toString()+" kn"+" / "+objekt.izracunajUEurima()+" EUR");

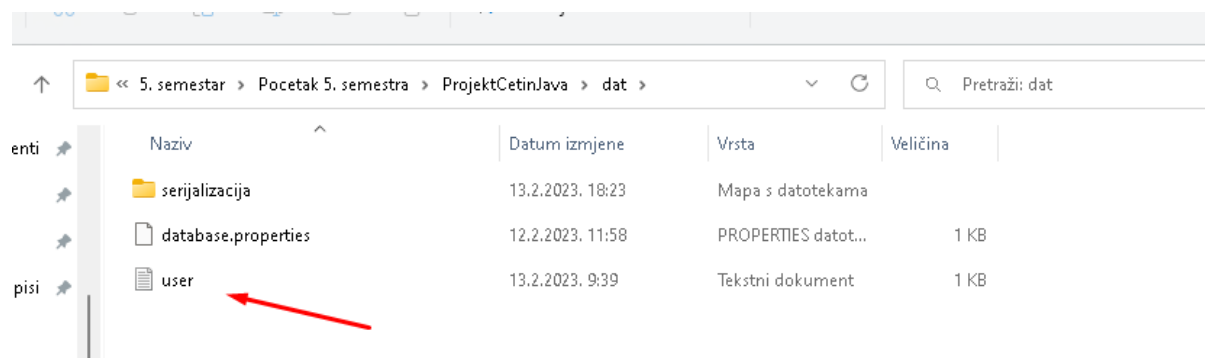
```

Šaljemo sve kune koje smo zbrojili u košarici (npr: cijena svih lijekova je 50 kuna). Zatim se ta cijena šalje u genericku klasu i prebacije u eure. Nakon toga se u „Label“ ispisuje cijena u kunama i u eurima nakon što se izračuna.

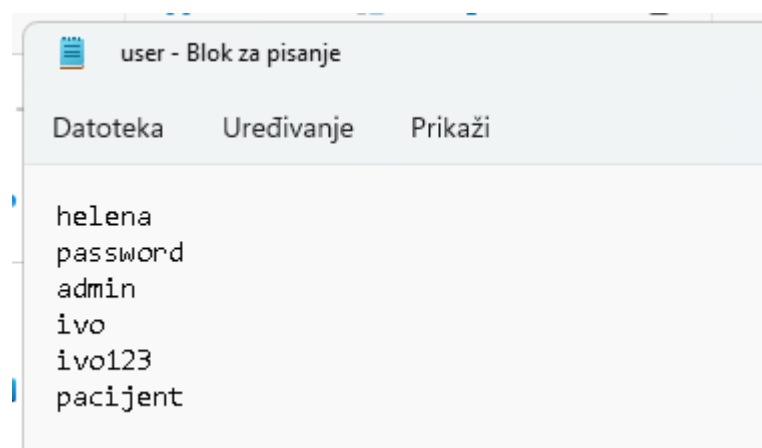
6.	Korištenje tekstualnih datoteka koje učitavaju podatke o korisničkim imenima i lozinkama prilikom prijave korisnika u aplikaciju. Potrebno je koristiti i binarne datoteke kojima se serijaliziraju i deserijaliziraju podaci o obavljenim promjenama podataka u projektnom zadatku (na primjer, nakon unošenja novih podataka te promjene postojećih).
----	---

Tekstualna datoteka za korisnicka imena i lozinke se nalazi u folderu „dat“.

Datoteka se zove „user“

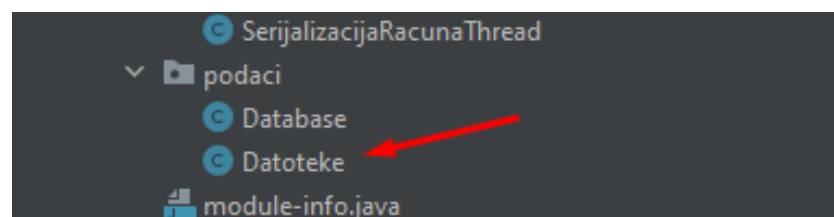


U tom file-u se nalaze podaci za login



Ovi useri se citaju u klasi „Datoteke“ (liniju po liniju, kao u 6. pripremi)

Od korisnika spremamo username, password i rolu. Ako se ulogiramo sa „helena password“ onda imamo sve privilegije, a ako se ulogiramo sa „ivo ivo123“ onda se ulogiravamo kao pacijent i mozemo samo pretrazivati stvari i kupovati lijekova (ne moze unositi nove)



7.	Implementirati JavaFX ekran za prijavu korisnika u aplikaciju koja čita podatke iz tekstualne datoteke o korisničkim imenima i „hashiranim“ lozinkama iz tekstualne datoteke kreirane u šestom koraku. Svaka aplikacija mora imati barem dvije korisničke role.
----	---

Provjera da li se korisnik dobro ulogirao se radi u „HelloController“-u.

Provjerava se da li su uneseni username, password i rola identicni onima koji su u datoteci.

```

1 usage
2 @FXML
3 private void onLoginClick() throws IOException {
4
5     String username = korisnikovUsernameTextField.getText();
6     String password = korisnikovPasswordTextField.getText();
7     String rola = korisnikoveRole.getValue();
8
9     for(int i=0; i<korisnici.size(); i++) {
10         if(username.equals(korisnici.get(i).getUsername())){
11             if(password.equals(korisnici.get(i).getPassword())){
12                 if(rola.equals(korisnici.get(i).getRola())){
13
14                     userRole = rola;
15
16                     Stage stage = HelloApplication.mainStage;
17
18                     FXMLLoader fxmlLoader = new FXMLLoader(HelloApplication.class.getResource("dobrodošli.fxml"));
19                     Scene scene = new Scene(fxmlLoader.load(), 500, 500);
20                     stage.setTitle("Hello!");
21                     stage.setScene(scene);
22                     stage.show();
23                 }
24             }
25         }
26     }
27 }

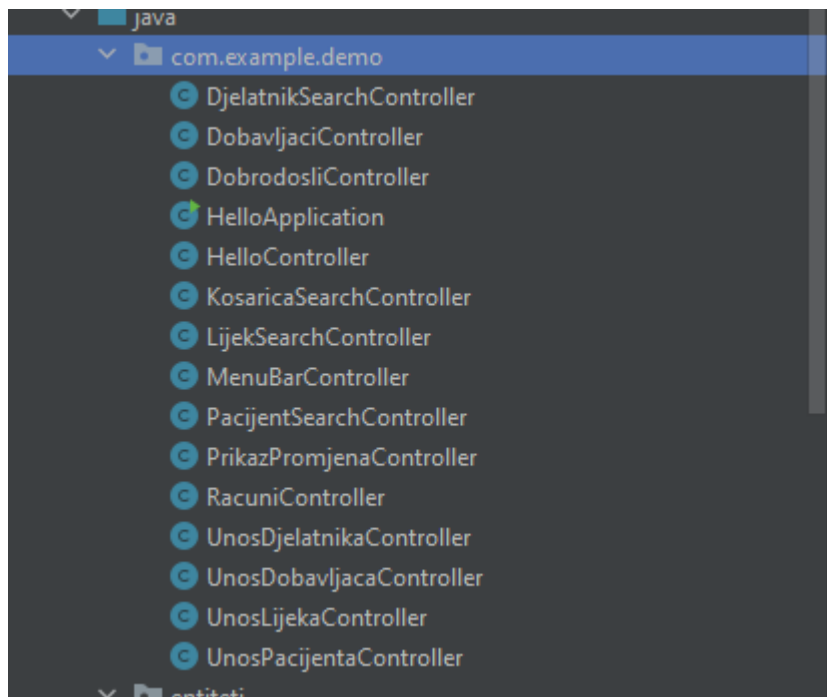
```

Ako jesu, šalju se na dobrodošli.fxml, i mogu koristiti aplikaciju.

Serijalizaciju i deserijalizaciju radimo na racunima. Tako da racuni uvijek ostanu u aplikaciju cak i nakon sto se zatvori. Koristili smo serijalizaciju i deserijalizaciju u kombinaciji s threadovima tako da cu to objasniti kasnije zajedno s thradovima.

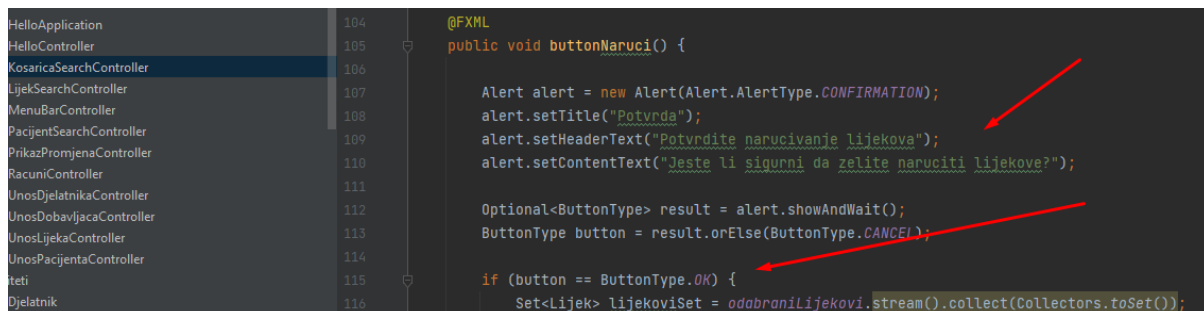
8.	Implementirati JavaFX ekran koji će za svaki entitet omogućavati korištenje funkcionalnosti pretrage i filtriranje podataka (korištenjem tablice TableView), dodavanje novog entiteta, promjene postojećih entiteta te brisanje entiteta. Svaka akcija promjene i brisanja entiteta mora uključivati dodatnu potvrdu korisnika da je suglasan s promjenom ili brisanjem korištenjem JavaFX dijaloga.
----	--

Nalaze se u svim scenama koje imaju „SeachController“ ili „Unos“ u imenu



„SearchController“ rade identicno kao i na 6. ili 7. labosu. Imamo TableView u kojem se nalaze svi podaci (koji se vuku iz baze podataka), korisnik unese neki podatak, npr: neko ime i nakon toga se filtriraju svi podaci u tablici koji imaju to ime. Svi SearchController-i rade na identičan način.

Unos Controller-i se koriste kako bi mogli unositi nove podatke u bazu podataka (tako da kada se pročitaju podaci iz baze podataka sljedeci put ce zauvijek ostati u aplikaciji). Prije kupnje lijekova, korisnika se pita da li je siguran da zeli kupiti te lijekova sa Alert-om)

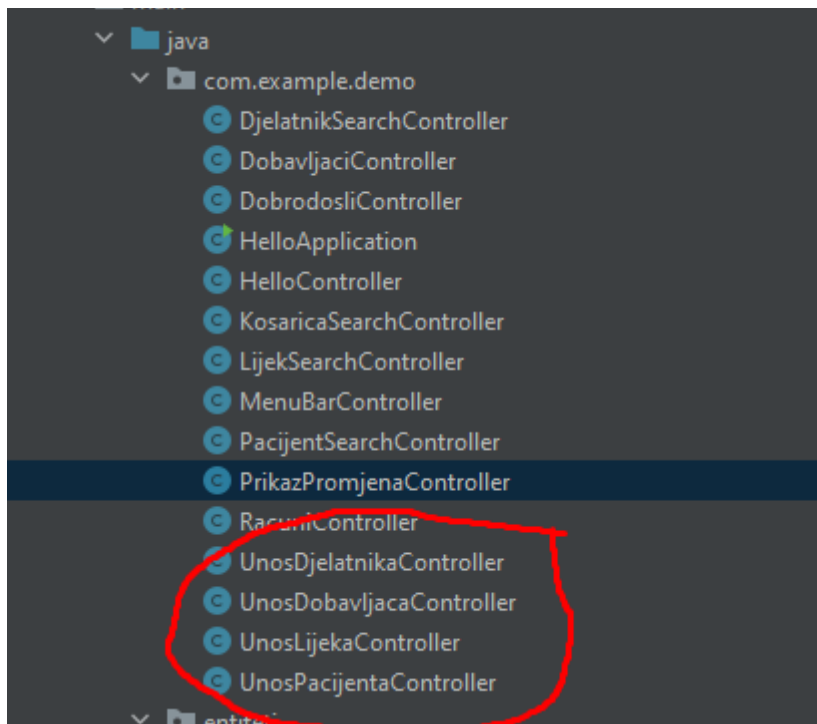


Samo ako je kliknuo „OK“ ce mu se ti lijekovi naruciti i spremiti u racune.

9.	Implementirati JavaFX ekran koji će omogućavati prikaz svih promjena koje su obavljene u aplikaciji projektnog zadatka korištenjem serijaliziranih podataka iz šestog koraka. Svaka promjena mora sadržavati podatak koji je promijenjen, staru i novu vrijednost, rolu koja ga je promijenila te datum i vrijeme kad se ta promjena dogodila.
----	--

U HelloController smo napravili listu stringova (listu poruka) u koju ćemo spremati sve poruke kao npr: „korisnik je unio novi lijek u aplikaciju“...

Taj zapis se sprema u kontrolerima koji imaju „Unos“ u imenu.



```

60      clearAll();
61
62      HelloController.promjeneUAplikaciji.add("Admin je dodao novog djelatnika u aplikaciju");
63

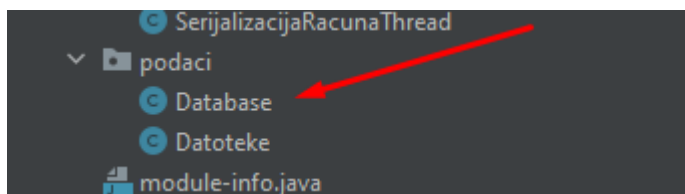
```

A kada odemo u „Prikaz promjena“ tamo se iz te liste ispisuju sve promjene koje su se desile od kad se aplikacija otvorila. Ne piše koji korisnik je promijenio jer samo admin može dodavati nove entitete. Ne ispisuje se datum i vrijeme jer bi bilo pre sumnjivo ako cijela aplikacija radi savršeno (sumnjam da će itko imati sve napravljeno).

10.	Kreirati bazu podataka koja će sadržavati podatke o svim entitetima koji se koriste u aplikaciji te implementirati klasu koja će implementirati funkcionalnosti kreiranje konekcije s bazom podataka, izvršavanje upita nad bazom podataka, dohvaćanje podataka iz baze podataka te zatvaranje konekcije s bazom podataka.
-----	--

Bazu smo kreirali na način kako je napisano u onom prošlom PDF-u od strane 5 do 25 ili tako nešto. Tamo je lijepo opisano korak po korak pa radije pogledaj tamo kako je napravljen dio sa bazom.

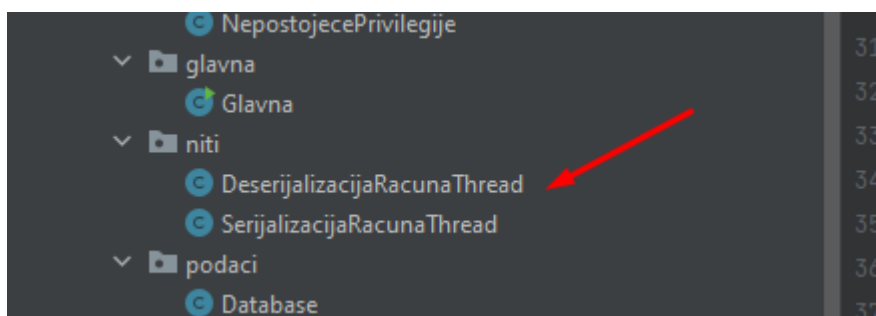
U našem slučaju imamo metodu koja se spaja na bazu, metode koje dohvaćaju podatke iz baze i metode koja stavljaju nove podatke na bazu. Sve što je vezano za bazu se nalazi u klasi „Database“



Sve što je zapisano u toj klasi je 90% radovanov kod. Samo smo izmjenili da se sve vuče iz naših tablica (naši podaci)

11.	Korištenjem niti implementirati funkcionalnosti osvježavanja podataka na ekranu aplikacije te konkurentno pristupanje dijeljenom resursu kojem pristupa više niti kroz sinkronizaciju niti (npr. jedna nit ispisuje detalje o posljednje promijenjenom podatku koji dohvaća iz serijalizirane datoteke, a za drugu nit koja sprema nove promjene u serijaliziranu datoteku osigurana je sinkronizaciju s tom prvom niti).
-----	---

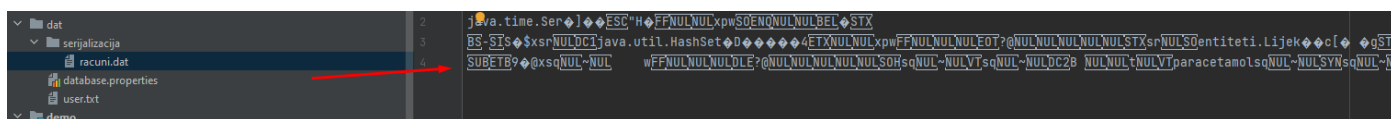
Niti se nalaze u paketu „niti“



Jedna nit služi za serijalizaciju podataka o racunima, a druga služi za deserijalizaciju podataka o racunima.

Serijalizacija je jako slična kao i zapisivanje u tekstualne datoteke, samo što ćemo umjesto pravog teksta dobiti neke random znakove

Npr:



Prednost toga je to što ne moramo čitati liniju po liniju nego možemo spremiti cijeli objekt i pročitati ga u par linija koda. Brže je i jednostavnija.

Niti smo implementirali u „KosaricaSearchController“. Kada korisnik klikne „naruci“ onda se oba threada puštaju (stavljaju se u stanje spremno za izvođenje). Threadovima ne možemo reci kad da se izvrše nego ih samo pustimo da se pokrenu i ne možemo odrediti koji thread će se izvršiti prije. Zato ih moramo sinkronizirati.

```
nettocontroller.setracuni(kosaricaSearchController);

SerijalizacijaRacunaThread serijalizacijaRacuna = new SerijalizacijaRacunaThread();
DeserijalizacijaRacunaThread deserijalizacijaRacuna = new DeserijalizacijaRacunaThread();

ExecutorService executorService = Executors.newCachedThreadPool();
executorService.execute(serijalizacijaRacuna);
executorService.execute(deserijalizacijaRacuna);
executorService.close();
```

Napravili smo nasa 2 threada (jedan će nam spremiti sve racune koje imamo do sada), a drugi će ih pročitati i prikazati na zaslону.

Serijalizacija -> sprema u datoteku

Deserijalizacija -> čita iz datoteke

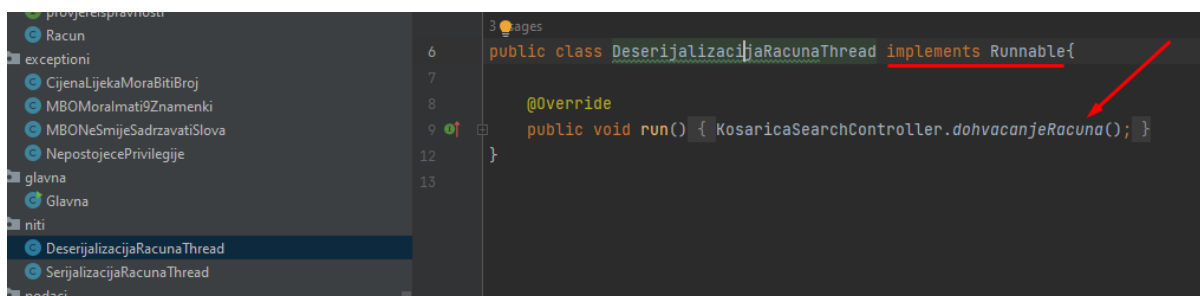
Kada se pozovu te dvije niti obje će izvršiti svoj dio koda koji se nalazi u metodi „run“.

BITNO! Je da klasa implementira sučelje „Runnable“ (to označava da će ta klasa raditi kao nit)



The screenshot shows the IDE with the project structure on the left. The 'niti' package contains 'DeserijalizacijaRacunaThread' and 'SerijalizacijaRacunaThread'. The 'SerijalizacijaRacunaThread' class is selected, and its code is visible in the editor. The class implements the 'Runnable' interface. The 'run()' method calls 'KosaricaSearchController.spremanjeRacuna()'. A red arrow points to the 'implements Runnable' part of the class declaration.

```
3 usages
6 public class SerijalizacijaRacunaThread implements Runnable{
7
8
9     @Override
10    public void run() { KosaricaSearchController.spremanjeRacuna(); }
11
12
13
14
```



The screenshot shows the IDE with the project structure on the left. The 'niti' package contains 'DeserijalizacijaRacunaThread' and 'SerijalizacijaRacunaThread'. The 'DeserijalizacijaRacunaThread' class is selected, and its code is visible in the editor. The class implements the 'Runnable' interface. The 'run()' method calls 'KosaricaSearchController.dohvacanjeRacuna()'. A red arrow points to the 'implements Runnable' part of the class declaration.

```
3 usages
6 public class DeserijalizacijaRacunaThread implements Runnable{
7
8
9     @Override
10    public void run() { KosaricaSearchController.dohvacanjeRacuna(); }
11
12
13
14
```

Svaki thread pokreće svoju metodu koje smo napisali u KosaricaSearchController“

Bitno je da smo tim metodama dodali `synchronized` (da budu sinkronizirane).

```
138 | usage
139 | public static synchronized void spremanjeRacuna(){
140 |     try {
141 |         ObjectOutputStream file = new ObjectOutputStream(new FileOutputStream( name: "dat/serijalizacija/racuni.dat"));
142 |
143 |         file.writeObject(HelloController.racuni);
144 |
145 |         file.close();
146 |     } catch (IOException e) {
147 |         throw new RuntimeException(e);
148 |     }
149 | }
150 |
151 | usage
152 | public static synchronized void dohvatanjeRacuna(){
153 |     try {
154 |         ObjectInputStream file = new ObjectInputStream(new FileInputStream( name: "dat/serijalizacija/racuni.dat"));
155 |         HelloController.racuni = (List<Racun>) file.readObject();
156 |         file.close();
157 |     } catch (IOException | ClassNotFoundException e) {
158 |         throw new RuntimeException(e);
159 |     }
160 | }
161 | }
```

Metoda „spremanjeRacuna“ će serijalizirati sve racune koje za sad imamo u file „racuni.dat“, a „dohvacanjeRacuna“ će ih pročitati i spremiti u listu (da bi nam se prikazivali na ekranu pošto se svi podaci zapisani u našoj aplikaciji čitaju iz listi).

Tako smo dobili konstantno spremanje / čitanje (odnosno osvježavanje) podataka za našu scenu koja prikazuje racune. U racunima su zapisani svi lijekovi koje je kupac narucio, pacijent koji je narucio lijekove i datum i vrijeme naručivanja.