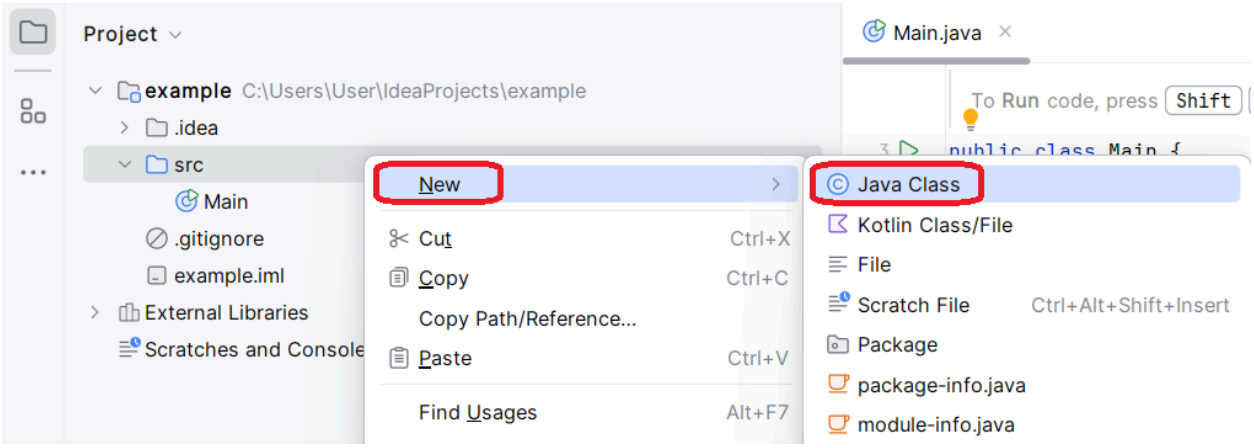
Laboratorijska vježba 2.

Klase u Java programskom jeziku

2.1. Kreiranje nove klase

U prethodnoj laboratorijskoj vježbi kreirana su prva programska rješenja u Java programskom jeziku, koristeći IntelliJ okruženje. Tako definisana rješenja sadržavala su *main()* funkciju koja vrši obradu korisničkog ulaza, a prikazan je i način kreiranja dodatnih funkcija u okviru *Main* klase. Međutim, na ovaj način moguće je kreirati samo jednostavne konzolne aplikacije. Za naprednije implementacije, neophodno je koristiti **klase** i kreirati klasne sisteme. Kada se želi kreirati nova klasa, to je moguće izvršiti u *Main.java* datoteci predefinisanoj u okviru novog projekta, međutim to umanjuje čitljivost programskog koda, dovodi do gomilanja i nemogućnosti snalaženja i u praksi se nikada ne koristi. Iz tog razloga, potrebno je za svaku klasu koja se želi definisati u sistemu dodati **novu datoteku koja ima isto ime kao i klasa koja se želi dodati**. Nova datoteka za klasu dodaje se u **src** folder (desni klik na folder → *New* → *Java Class*), na način prikazan na Slici 1.



Slika 1. Kreiranje nove klase u programskom rješenju

Na prethodno opisani način kreirana je nova klasa u folderu koji sadrži sve klase programskog rješenja. Sada je potrebno kreirati programski kod koji opisuje datu klasu. Neka se želi napraviti klasa koja će opisivati studente koristeći standardne podatke (ime, prezime, datum rođenja, broj indeksa, godina studija, odsjek, lista upisanih ocjena u toku studija). U tu svrhu

potrebno je definisati atribute klase *Student* koja je prethodno kreirana u novoj datoteci. To je moguće izvršiti na način prikazan u Listingu 1, pri čemu je važno napomenuti da se umjesto primitivnih tipova (npr. *int, double*) koriste izvedeni tipovi Java programskog jezika (npr. *Integer, Double*) zbog postojanja predefinisanih funkcija koje olakšavaju rad s navedenim tipovima podataka. Za korištenje klasa **Date** i **List** potrebno je dodati odgovarajuće biblioteke, pri čemu se preporučuje da se isto izvrši koristeći IntelliJ okruženje (*Show context actions* opcija opisana u prethodnoj laboratorijskoj vježbi), a ne ručnim unošenjem naziva biblioteka.

```
import java.util.Date;
import java.util.List;

public class Student
{
   String ime, prezime, brojIndeksa, odsjek;
   Date datumRodjenja;
   Integer godinaStudija;
   List<Integer> ocjene;
}
```

Listing 1. Dodavanje atributa klase

U narednom koraku, potrebno je dodati konstruktor koji inicijalizira date atribute, kao i neke metode potrebne za rad sa studentima, kao naprimjer izračunavanje ukupnog prosjeka u toku studija i formiranje izlaznog stringa sa osnovnim informacijama o studentu, kako bi se informacije mogle ispisati u konzoli. Kreiranje konstruktora i prethodno opisanih metoda prikazano je u Listingu 2. Kao što se može primijetiti, u programskom kodu konstruktora parametrima su dodijeljeni isti nazivi kao nazivi atributa klase *Student*, zbog čega je neophodno specificirati **this.atribut** kada se želi koristiti atribut klase, dok se bez korištenja **this.** prefiksa koristi parametar konstruktora. U okviru funkcije *Prosjek()* demonstrirano je korištenje **for each** petlje, u okviru koje se iterirajući kroz kolekciju kreiraju privremene varijable **ocjena** koje se koriste za izračunavanje prosjeka. Za omogućavanje lakog i brzog ispisa informacija o studentima u konzoli, kreirana je funkcija **toString()** koja omogućava tretiranje klase *Student* kao string, pritom implicitno pozivajući programski kod ove funkcije.

```
public Student(String ime, String prezime, Date datumRodjenja, String
brojIndeksa, String odsjek, Integer godinaStudija)
{
   this.ime = ime;
   this.prezime = prezime;
   this.datumRodjenja = datumRodjenja;
   this.brojIndeksa = brojIndeksa;
   this.odsjek = odsjek;
   this.godinaStudija = godinaStudija;
   ocjene = new ArrayList<Integer>();
}

public Double Prosjek()
{
```

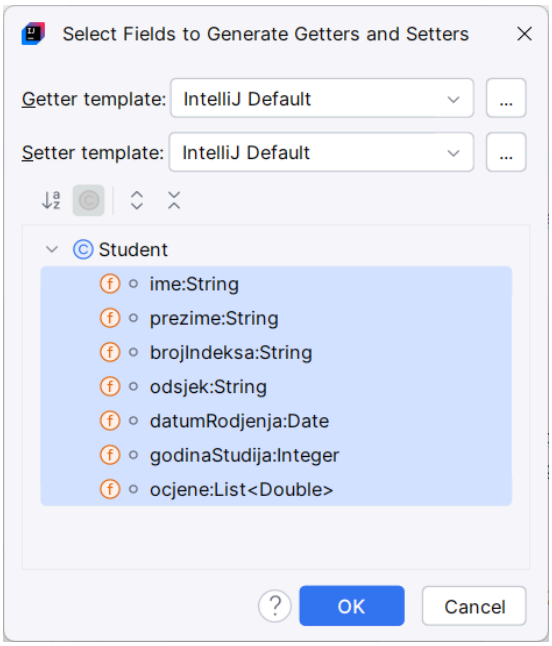
```
    Double prosjek = 0.0;
    for (Integer ocjena : ocjene)
        prosjek += ocjena;
    return prosjek / ocjene.size();
}

public String toString()
{
    return "Student: " + ime + " " + prezime + ", broj indeksa: " + brojIndeksa
+ ", prosjek: " + Prosjek();
}
```

Listing 2. Dodavanje konstruktora i metoda klase

2.2. get() i set() metode

Jedna od često neophodih funkcionalnosti pri korištenju klasa je pristup njenim atributima, kao i izmjena istih u toku rada programa, a ne samo tokom inicijalizacije u okviru konstruktora. Kako bi se ovo omogućilo, potrebno je kreirati **get()** i **set()** metode koje služe za enkapsulaciju atributa klase, koji uvijek moraju biti privatni i ne smije im se moći pristupiti izvan klase koristeći *public* stereotip. Ove metode moguće je kreirati ručno, međutim one se najčešće automatski generišu. Kako bi se izvršilo automatsko generisanje u IntelliJ okruženju, potrebno je izvršiti desni klik u okviru klase *Student* i odabrati opciju *Generate…* (ili kroz glavni menu, opcija *Code → Generate…*). Nakon toga moguće je odabrati da li se želi automatski generisati, konstruktor, *toString()* metoda, samo *get()* metode, samo *set()* metode ili i *get()* i *set()* metode. Nakon toga potrebno je selektovati sve definisane atribute klase *Student* (ili samo atribute za koje se želi omogućiti ova funkcionalnost) i odabrati opciju **OK**, kao što je prikazano na Slici 2.



Slika 2. Automatsko dodavanje get() i set() metoda

Kao rezultat prethodno opisanog postupka automatskog generisanja, dobiva se ukupno 14 novih metoda u klasi *Student* (*get()* i *set()* metode za svih 7 selektovanih atributa). Primjer nekih od njih prikazan je u Listingu 3, odakle je vidljivo da bez obzira na tip atributa, sve automatski generisane metode imaju isti oblik, u kojem *get()* metode samo vraćaju vrijednost atributa, a *set()* metode postavljaju vrijednost atributa na proslijeđeni parametar metode.

```
public String getIme() {
    return ime;
}

public void setIme(String ime) {
    this.ime = ime;
}

public Date getDatumRodjenja() {
    return datumRodjenja;
}

public void setDatumRodjenja(Date datumRodjenja) {
    this.datumRodjenja = datumRodjenja;
}

public Integer getGodinaStudija() {
    return godinaStudija;
}

public void setGodinaStudija(Integer godinaStudija) {
    this.godinaStudija = godinaStudija;
}

public List<Integer> getOcjene() {
    return ocjene;
}

public void setOcjene(List<Integer> ocjene) {
    this.ocjene = ocjene;
}
```

Listing 3. Prikaz primjera automatski generisanih get() i set() funkcija

Iako je u nekim slučajevima dovoljno imati ovako definisane *get()* i *set()* metode, najčešće je potrebno u njihovu strukturu dodati programski kod koji će zamijeniti različite provjere validnosti datih atributa. Naprimjer, neka se želi zabraniti postavljanje datuma rođenja studenta na datum koji se nalazi u budućnosti, kao i postavljanje godine studija na vrijednost izvan opsega [1, 5]. U Listingu 4 prikazani su različiti načini izmjena datih *set()* metoda kako bi se obavile tražene validacije, pri čemu se u metodi za datum rođenja baca izuzetak ukoliko je vrijednost parametra neispravna, dok se u metodi za godinu studija vrijednost parametra samo ignoriše ukoliko je pogrešna. Za poređenje dva datuma koristi se funkcija **compareTo()**, koja kao parametar prima datum s kojim se data varijabla poredi (današnji datum može se dobiti jednostavnom inicijalizacijom nove **Date()** varijable bez parametara konstruktora). Funkcija za poređenje vraća

vrijednost manju od nule ukoliko je datum u prošlosti u odnosu na parametar, nulu ukoliko su datumi jednaki, ili vrijednost veću od nule ukoliko je datum u budućnosti u odnosu na parametar.

```
public void setDatumRodjenja(Date datumRodjenja) throws Exception {
    if (datumRodjenja.compareTo(new Date()) < 0)
        this.datumRodjenja = datumRodjenja;
    else throw new Exception("Datum rođenja mora biti u prošlosti!");
}

public void setGodinaStudija(Integer godinaStudija) {
    if (godinaStudija > 0 && godinaStudija < 6)
        this.godinaStudija = godinaStudija;
}
```

Listing 4. Validacija u okviru set() metoda

S obzirom da su sada kreirane *get()* i *set()* metode za sve atribute, iste je neophodno iskoristiti u konstruktoru klase *Student*, a s obzirom da su metode *public* tipa, korisnik im može pristupiti iz *Main* klase po potrebi. Izmjene konstruktora prikazane su u Listingu 5, odakle je vidljivo da je ručno postavljanje atributa na vrijednosti parametara zamijenjeno korištenjem *set()* metoda. Za one *set()* metode koje ne posjeduju validacijsku logiku, korištenje ovih metoda ekvivalentno je prethodnim ručnim postavljanjima. Osim toga, s obzirom da je moguće da pri postavljanju datuma rođenja dođe do pojave izuzetka, potrebno je obraditi izuzetak u okviru konstruktora. U ovom slučaju, odlučeno je da se izuzetak samo proslijedi na iduću instancu (tj. mjesto poziva konstruktora) gdje ga je neophodno obraditi, tako što je u deklaraciji konstruktora samo dodano **throws Exception**, kao i u metodi *setDatumRodjenja()*.

```
public Student(String ime, String prezime, Date datumRodjenja, String
brojIndeksa,
               String odsjek, Integer godinaStudija) throws Exception
{
    setIme(ime);
    setPrezime(prezime);
    setDatumRodjenja(datumRodjenja);
    setBrojIndeksa(brojIndeksa);
    setOdsjek(odsjek);
    setGodinaStudija(godinaStudija);
    setOcjene(new ArrayList<Integer>());
}
```

Listing 5. Korištenje set() metoda u konstruktoru klase

2.3. Korisnički definisani tipovi izuzetaka

U prethodno opisanom postupku, korišten je opći tip izuzetaka **Exception**. Postoji veliki broj različitih predefinisanih tipova izuzetaka (pogledati sljedeći link) koji se koriste kako bi se pri izvršavanju programa ukazalo na razlog zbog kojeg je došlo do greške (naprimjer, *ArithmeticException* može ukazati da je došlo do dijeljenja s nulom). Kako bi se pri radu

programa mogle razlikovati različite situacije do kojih dolazi zbog različitih uzroka, mogu se definisati korisnički tipovi izuzetaka. U Listingu 6 prikazan je primjer dva korisnički definisana tipa izuzetaka, čija imena sama po sebi daju indiciju na koju grešku ukazuje njihova pojava - studenta čiji je datum rođenja postavljen na datum koji se nalazi u budućnosti, ili studenta čiji je datum rođenja postavljen na datum u prošlosti, ali nedovoljno daleko u prošlosti da bi se mogao proći validaciju. Važno je napomenuti da su ove klase korisnički definisanih izuzetaka smještene u posebne datoteke, kao i da deklaracija **extends Exception** označava da je u pitanju tip izuzetka, a funkcija **super()** poziva konstruktor osnovne klase izuzetka. O ovim stvarima biti će više riječi u idućoj laboratorijskoj vježbi.

```
public class StudentBuducnostException extends Exception
{
   public StudentBuducnostException (String message)
   {
       super(message);
   }
}

public class PremladStudentException extends Exception
{
   public PremladStudentException (String message)
   {
       super(message);
   }
}
```

Listing 6. Kreiranje korisnički definisanih tipova izuzetaka

Nakon definisanja korisnički definisanih izuzetaka, sada ih je moguće upotrijebiti u programskom kodu. Listing 7 prikazuje način na koji je sada moguće razraditi validaciju u okviru *setDatumRodjenja()* metode, kako bi se mogli razlikovati studenti koji su mlađi od 16 godina od studenata čiji su datumi rođenja postavljeni na današnji datum ili neki datum u budućnosti. Kako bi se provjerila razlika u godinama između dva datuma, upotrijebljena je predefinisana funkcija **getYear()**. Sada je putem **try-catch** blokova moguće obraditi pojavu oba izuzetka, naprimjer na način da se omogućava ponovni unos ukoliko je datum rođenja u budućnosti, a da se ispisuje poruka o grešci i brišu svi podaci za studenta mlađeg od 16 godina. Potrebno je napomenuti da, bez obzira što oba korisnička tipa izuzetaka predstavljaju vrstu nadtipa *Exception*, bilo je neophodno praviti izmjene u stereotipu funkcije tako da se deklariše da se dešava bacanje oba tipa izuzetaka (**throws StudentBuducnostException, PremladStudentException**). Istu promjenu potrebno je napraviti u stereotipu konstruktora klase *Student* koji poziva ovu metodu.

```
public void setDatumRodjenja(Date datumRodjenja) throws
StudentBuducnostException, PremladStudentException {
   Date today = new Date();
   if (datumRodjenja.compareTo(today) >= 0)
       throw new StudentBuducnostException("Datum rođenja ne može biti u
budućnosti!");
```

```
    else if (today.getYear() - datumRodjenja.getYear() < 16)
        throw new PremladStudentException("Student ne može biti mlađi od 16
godina!");

    this.datumRodjenja = datumRodjenja;
}
```

Listing 7. Korištenje korisnički definisanih izuzetaka u okviru validacije

2.4. Enumeracijski tipovi

Veoma često postoji potreba za korištenjem atributa koji mogu poprimiti samo par tačno određenih vrijednosti. Naprimjer, definisani atribut *odsjek* može poprimiti samo jednu od pet mogućih vrijednosti: “AIE”, “EE”, “RI”, “TK” i “RS”, ali s obzirom da je njegov tip *String*, potrebno je izvršiti različite validacije kako bi se provjerilo da li je unesena vrijednost jedna od ispravnih, ili neka od mnogobrojnih neispravnih vrijednosti (npr. prazan string, string koji se sastoji od brojeva, itd.). Kako bi se izbjegla potreba za korištenjem *if-else* logike za provjeru validnosti unesenih podataka, kao i kako bi se olakšao rad sa ovakvim tipovima podataka, koriste se enumeracijski tipovi. Enumeracijski tip se definiše na skoro isti način kao u C++ programskom jeziku, kao što je prikazano u Listingu 8, pri čemu je važno još jednom napomenuti da je enumeracijski tip definisan u okviru zasebne datoteke.

```
public enum Odsjek
{
    AIE, EE, RI, TK, RS
}
```

Listing 8. Dodavanje novog enumeracijskog tipa

Sada je potrebno izmijeniti definiciju klase *Student*, tako da se umjesto *String* atributa koristi atribut tipa *Odsjek*, uključujući i konstruktor i *get()* i *set()* metode. Ovako definisan atribut sada se može koristiti u različitim validacijama. Naprimjer, s obzirom da stručni studij Razvoj softvera traje dvije godine i nema master studij, atribut *godinaStudija* za ovaj studijski program može biti samo u opsegu [1, 2]. Zbog toga je potrebno dodati novu validaciju u okviru *set()* metode za ovaj atribut. U Listingu 9 prikazana je izmijenjena metoda *setGodinaStudija()*, koja sada uključuje provjeru odsjeka kojem student pripada. Enumeracijski tipovi se mogu porediti i koristeći funkciju **equals()**, i koristeći operator **==**, a za dobivanje neke od vrijednosti enumeracijskog tipa koristi se sintaksa **Enumeracija.Vrijednost** (*Odsjek.RS*).

```
public void setGodinaStudija(Integer godinaStudija) {
    if (this.odsjek != Odsjek.RS && godinaStudija > 0 && godinaStudija < 6)
        this.godinaStudija = godinaStudija;
    else if (this.odsjek == Odsjek.RS && godinaStudija > 0 && godinaStudija <
2)
        this.godinaStudija = godinaStudija;
}
```

Listing 9. Korištenje enumeracijskih tipova u programskom kodu

2.5. Korištenje u Main klasi

Nakon svih prethodno opisanih koraka, kao rezultat se dobiva klasa *Student* koja posjeduje veći broj atributa, konstruktor, *get()* i *set()* metode, kao i dodatne funkcije koje olakšavaju rad sa ovom klasom. Sve što je kreirano sada je moguće koristiti u okviru *Main* klase, kojoj je klasa *Student* vidljiva jer je označena kao **public** i nalazi se u istom *src* folderu. U Listingu 10 prikazan je primjer unosa informacije o datumu rođenja putem konzole, pretvaranje unesenih informacija u datum koristeći konstruktor **Date(godina, mjesec, dan)**. Važno je napomenuti da ovaj konstruktor očekuje da vrijednost parametra *godina* bude u odnosu na 1900. godinu, zbog čega je izvršeno oduzimanje vrijednosti 1900 pri formiranju ovog parametra. Parametar *mjesec* započinje s nulom, zbog čega je pri formiranju ovog parametra oduzeta vrijednost 1, a parametar *dan* započinje s jedinicom, zbog čega nije bilo potrebe za izmjenama. Za ekstrakciju unesenih vrijednosti s konzole korištena je funkcija **substring(početniIndeks, krajnjiIndeks)**. U okviru konstruktora klase *Student* korištene su predefinisane vrijednosti za sve ostale parametre osim datuma. Korišteni su **try-catch** blokovi za provjeru da li je došlo do pojave izuzetka, i omogućeno ponavljanje unosa u slučaju da je unesen datum u budućnosti, dok je u slučaju da je student premlad definisan završetak izvršavanja programa. Na kraju se vrši prikaz informacija o studentu u konzoli, pri čemu je zbog definisane *toString()* funkcije moguće tretirati varijablu kao da je u pitanju string pri ispisu.

```
import java.util.Date;
import java.util.Scanner;

public class Main {
   public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        boolean uspjesanUnos = false;
        Student student = null;
        while (!uspjesanUnos) {
            try {
                System.out.printf("Datum rođenja (dd/mm/yyyy):");
                String datumString = scanner.nextLine();
                int godina = Integer.valueOf(datumString.substring(6)) - 1900;
                int mjesec = Integer.valueOf(datumString.substring(3, 5)) - 1;
                int dan = Integer.valueOf(datumString.substring(0, 2));
                Date datumDate = new Date(godina, mjesec, dan);
                student = new Student("Ime", "Prezime", datumDate, "12345",
Odsjek.RI, 2);
                uspjesanUnos = true;
            } catch (PremladStudentException e) {
                System.out.println(e.getMessage());
                return;
            } catch (StudentBuducnostException e) {
                System.out.println(e.getMessage());
                System.out.println("Molimo ponovite unos datuma rođenja!");
            }
        }
        System.out.println("Unos studenta uspješan! " + student);
```

```
    }
}
```

Listing 10. Kreiranje programske logike u Main klasi

Na Slici 3 prikazan je izlaz ovako definisanog programa, odakle je vidljivo da se ispravno detektuje ukoliko je datum rođenja studenta u budućnosti, što znači da je *setDatumRodjenja()* metoda ispravno definisana i dolazi do pojave izuzetka. Nakon ponovnog unosa, prikazuju se informacije o studentu na način definisan *toString()* funkcijom, pri čemu se za prosjek studenta prikazuje **NaN** (*not a number*) vrijednost. Ovo je posljedica načina definisanja *Prosjek()* funkcije, koja vrši dijeljenje sume ocjena sa brojem ocjena. Međutim, s obzirom da je broj ocjena nula, kao rezultat se dobiva NaN vrijednost.

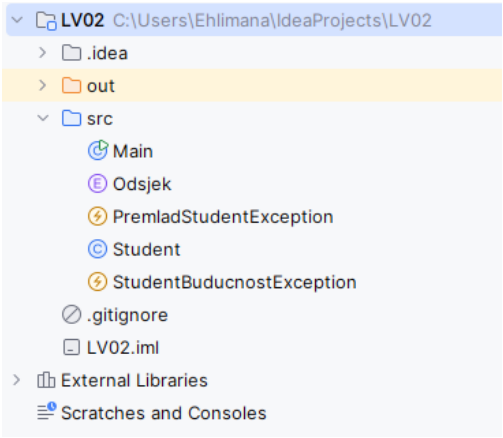
```
Datum rođenja (dd/mm/yyyy):10/10/2035
Datum rođenja ne može biti u budućnosti!
Molimo ponovite unos datuma rođenja!
Datum rođenja (dd/mm/yyyy):29/01/1996
Unos studenta uspješan! Student: Ime Prezime, broj indeksa: 12345, prosjek: NaN

Process finished with exit code 0
```

Slika 3. Izlaz u konzoli pri unosu datuma rođenja studenta

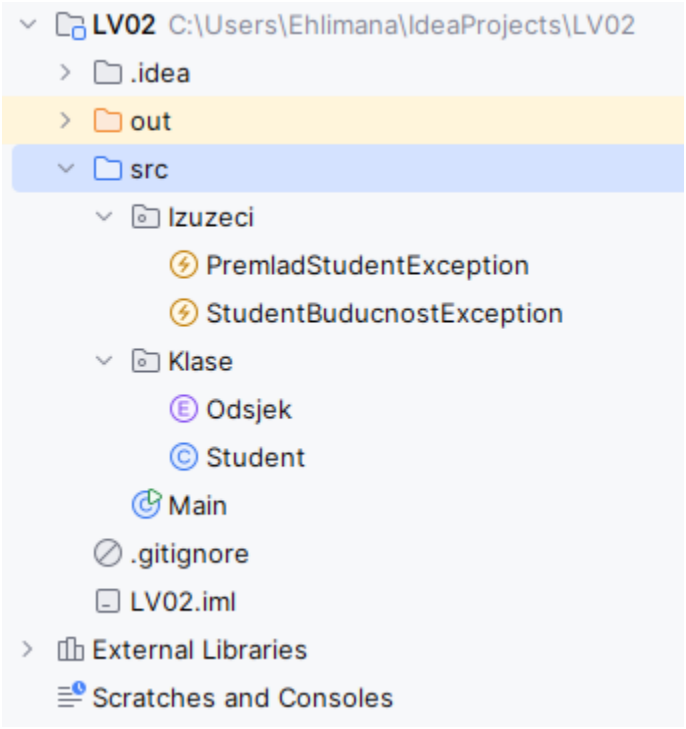
2.6. Kreiranje paketa

Prethodno definisane klase su sve definisane u *src* direktoriju, koji sada ima izgled kao na Slici 4. Međutim, nije preporučeno da se sve klase čuvaju u istom direktoriju, posebno ukoliko su neke od klasa tipovi izuzetaka, neke od klasa se koriste za pokretanje programa, a neke od klasa definišu podatke koji se žele koristiti.



Slika 4. Izgled src direktorija koji sadrži sve klase projekta

Može se primijetiti da je *src* direktorij označen plavom bojom, što znači da se u njemu nalazi izvorni programski kod. Folder *out* označen je narandžastom bojom, što znači da sadrži izvršne datoteke neophodne za pokretanje konzolne aplikacije. U slučaju nemogućnosti pokretanja programskog koda, potrebno je proglasiti folder sa klasama izvornim folderom, što se može izvršiti desnim klikom na folder i odabirom opcije *Mark Directory as → Sources Root*. U folderu izvršnog programskog koda sada se mogu definisati različiti **paketi** koji razdvajaju dijelove programskog koda. Ukoliko se izvrši desni klik na folder koji je proglašen izvornim folderom, neće biti moguće dodati novi direktorij (tj. folder), nego samo novi paket. Potrebno je dodati dva nova paketa u folderu *src*: **Izuzeci** i **Klase**, a zatim u njih pozicionirati odgovarajuće java datoteke (koristeći *drag-and-drop* operaciju) tako da se dobije izgled projektnog stabla kao na Slici 5. Pri vršenju *drag-and-drop* operacije, okruženje će postaviti upit da li se želi izvršiti refaktoring programskog koda, što je potrebno potvrditi.



Slika 5. Struktura projekta nakon dodavanja novih paketa

Ukoliko se sada izvrši pregled klasa, može se primijetiti da je u zaglavlju klasa koje su premještene u pakete dodana linija programskog koda koja određuje u kom paketu se data klasa nalazi. Slika 6 prikazuje izgled programskog koda klase *PremladStudentException* koja je premještena u paket *Izuzeci*, odakle je vidljivo da je ovoj klasi dodana linija koda `package Izuzeci;`, koja označava pripadnost datom paketu.

PremladStudentException.java

```
1  package Izuzeci;
2
3  public class PremladStudentException extends Exception  6 usages
4  {
5      public PremladStudentException (String message)  1 usage
6      {
7          super(message);
8      }
9  }
```

Slika 6. Prikaz klase koja pripada paketu Izuzeci

Automatski refaktoring programskog koda koji je izvršen pri premještanju klasa u pripadajuće pakete ne odnosi se samo na dodavanje zaglavlja o pripadnosti paketu. S obzirom da klase više nisu u istom (izvornom - *src*) paketu, one više ne mogu pristupiti jedne drugima bez navođenja da žele koristiti dati paket u svom zaglavlju. Naprimjer, klasa *Student*, koja sada pripada paketu *Klase*, koristi korisnički definisane tipove izuzetaka, koji se sada nalaze u paketu *Izuzeci*. Ukoliko se sada otvori datoteka klase *Student*, dobiti će se prikaz kao na Slici 7, odakle je vidljivo da je dodano zaglavlje koje određuje da klasa *Student* pripada paketu *Klase*. Međutim, dodane su i dvije linije koda koje određuju da ova klasa koristi klase iz paketa *Izuzeci* (za svaku klasu pojedinačno, a u slučaju neophodnosti za smanjenjem koda može se koristiti skraćena deklaracija `import Izuzeci.*;`). Na ovaj način omogućava se da ostatak programskog koda klase *Student* ostane neizmijenjen.

```
1   package Klase;
2
3   import Izuzeci.PremladStudentException;
4   import Izuzeci.StudentBuducnostException;
5
6   import java.util.ArrayList;
7   import java.util.Date;
8   import java.util.List;
9
10
11  public class Student  3 usages
12  {
```

Slika 7. Prikaz klase Student koja koristi klase iz drugih paketa

Ukoliko se otvori datoteka kojom je definisana *Main* klasa, primijetiti će se da su u njoj automatski izvršene iste izmjene kao u prethodno opisanim klasama. Međutim, postoji još jedan način za omogućavanje korištenja klasa iz vanjskih paketa, bez korištenja *import* direktive. Na Slici 8 prikazano je zaglavlje *Main* klase u kome su zakomentarisane automatski generisane direktive za import klasa koje su definisane u paketu *Klase*. Međutim, u liniji koda 13 uspješno se definiše atribut klase *Student* kao i u prethodnim dijelovima laboratorijske vježbe, s tim da je

sada neophodno eksplicitno navesti lokaciju ove klase među izvornim datotekama programskog koda (tj. relativno pozicioniranje u odnosu na *src* folder). U ovom slučaju jednostavno je moguće referirati se na klasu **Klase.Student** i na taj način će okruženje uspješno moći dobaviti programski kod klase *Student*. Na ovaj način može se kreirati hijerarhija paketa u programskom rješenju koja osigurava da su date klase vidljive samo unutar istog paketa i unutar onih paketa koji odluče da žele koristiti te klase, čime se ostvaruje veća čitljivost i razumljivost programskog koda.

```
1   import Izuzeci.PremladStudentException;
2   import Izuzeci.StudentBuducnostException;
3   //import Klase.Odsjek;
4   //import Klase.Student;
5
6   import java.util.Date;
7   import java.util.Scanner;
8
9   public class Main {
10      public static void main(String[] args) {
11          Scanner scanner = new Scanner(System.in);
12          boolean uspjesanUnos = false;
13          Klase.Student student = null;
```

Slika 8. Eksplicitno navođenje paketa u kome se nalazi klasa Student

2.7. Zadaci za samostalni rad

Za sticanje bodova na prisustvo laboratorijskoj vježbi, potrebno je uraditi sljedeće zadatke tokom laboratorijske vježbe i postaviti ih u repozitorij studenta na odgovarajući način:

Zadatak 1.

Prepraviti programski kod klase *Student* tako da nije moguće da pri ispisu informacija o studentu dođe do pojave NaN vrijednosti. Umjesto toga, definisati tip izuzetka *DijeljenjeSNulomException* i dovesti do njegove pojave u *Prosjek()* funkciji. S obzirom da nije moguće propagirati pojavu ovog izuzetka u *toString()* funkciji (jer se preklapa s postojećom funkcijom za ispis stringova), u ovoj funkciji (a ne u *main()* funkciji) obraditi izuzetak i ispisati informaciju o tome da student nema nijednu unesenu ocjenu, na način prikazan na Slici 4. Važno je napomenuti da ukoliko student nema ocjena, funkcija *toString()* treba vratiti prazan string kao rezultat.

```
Datum rođenja (dd/mm/yyyy):01/01/2000
Student nema nijednu unesenu ocjenu! Nije moguće ispisati podatke.
Unos studenta uspješan!

Process finished with exit code 0
```

Slika 4. Očekivani izlaz u zadatku 1

Zadatak 2.

Omogućiti unos ocjena studenta putem konzole na način prikazan na Slici 5, a zatim provjeriti da li se ispravno ispisuje prosjek studenta. Unos ocjena izvršiti nakon uspješnog unosa studenta, a prije ispisa informacija u konzoli, koristeći *set()* metodu za atribut ocjene. Unos ocjena izvršiti koristeći zarez kao delimiter, a zatim putem funkcije **split()** (vidjeti sljedeći link) razdvojiti uneseni string u pojedinačne ocjene. Ne vršiti direktni poziv *Prosjek()* funkcije, već iskoristiti postojeći programski kod koji vrši ispis svih informacija o studentu.

```
Datum rođenja (dd/mm/yyyy):01/01/2000
Unesite ocjene studenta: (x,y,...):6,6,8,8,10,10,9
Unos studenta uspješan! Student: Ime Prezime, broj indeksa: 12345, prosjek: 8.142857142857142

Process finished with exit code 0
```

Slika 5. Očekivani izlaz u zadatku 2