8 Objektno orijentisano programiranje

Sada kada smo prošli sve neophodne stvari koje se tiču običnog programiranja, možemo prebaciti našu pažnju na **objektno orijentisano programiranje**. Ovakav vid programiranja možemo primeniti samo sa programskim jezicima koji podržavaju kreiranje objekata, odnosno klasa. Već više od 3 decenije, ovakav vid programiranja predstavlja 'zlatni standard' kada je u pitanju produkcijski kod za većinu aplikacija i omogućava nam da na struktuarniji način organizujemo naše promenjive i funkcije, da lakše apstraktizujemo naš kod i da ga organizujemo u logičke celine.[1] Da stvarno uvidimo potrebu za OOP konceptima, pogledajmo jedan zadatak, koji od nas sada zahteva da napišemo čitav (ali relativno mali) program:

Zadatak 8.1 (Elektronski dnevnik)**.**

Napisati program za elektronski dnevnik. Takav program treba da omogući unos novih djaka. Svaki djak ima svoje ime i prezime, godište, niz ocena, brojčani i opisni prosek kao i dodatne napomene u vidu teksta uz pretpostavku da nemamo dva djaka sa istim imenom i prezimenom. Omogućiti sve neophodne operacije, poput izmene ocena, brisanje ocena, dopisivanje ocena, izmene napomena itd.

Jasno je da ovakav program bi imao više različitih tipova podataka asociranih po jednom djaku, npr:

```
String ime1, prezime1;
int godiste;
int[] ocene1;
double prosek1;
String opisniProsek1;
String napomena1;
```

i ovih 7 promenjivih bi činili jednu semantičku celinu našeg programa koju bi smo mi okarakterisali kao "Djak".
Kao što možemo da pretpostavimo prateći šablon lekcija do sada, ove promenjive možemo da grupišemo zajedno i čak da definišemo neke relacije medju njima putem funkcija, a za tako nešto nam treba pojam **klase**!

Klase

Definicija 8.1 (Klasa)**.**

Klasa, odnosno 'šablon', je definisana svojim **atributima** odnosno **poljima** i **metodama** i pretstavlja *abstraktizaciju* svojih **objekata**.

Definicija 8.2 (Atributi klase)**.**

Pod atributima, odnosno poljima klase podrazumevamo promenjive asocirane sa telom klase. Kao takvi, oni su dostupni u svim **metodama** klase.

Kako su pojmovi `atribut` i `polje` sinonimi, mi ćemo ih koristiti nasumično u ovoj skripti i svako pojavljivanje jednog od tih pojmova može se zameniti drugim.

Definicija 8.3 (Metode klase)**.**

Pod metodama klase podrazumevamo sve funkcije definisane u telu klase.

Opšti šablon pisanja klasa je sledeći:

```
[<modifikatori_klase>] class <imeKlase> {

    <tip_polja1> <identifikator_polja1>;
    <tip_polja2> <identifikator_polja2>;
    ...
    <tip_poljaN> <identifikator_poljaN>

    <modifikator_vidljivosti_konstruktora1> <imeKlase>
([<parametri_konstruktora1>]) {
        //Telo konstruktora 1
    }

    <modifikator_vidljivosti_konstruktora2> <imeKlase>
([<parametri_konstruktora2>]) {
        //Telo konstruktora 2
    }

    ...

    <modifikator_vidljivosti_konstruktoraM> <imeKlase>
([<parametri_konstruktoraM>]) {
        //Telo konstruktora M
    }

    [<modifikatori_metode1>] <identifikator_metode1>
```

```
([<parametri_metode1>]) {
        //Telo metode 1
    }

    [<modifikatori_metode2>] <identifikator_metode2>
([<parametri_metode2>]) {
        //Telo metode 2
    }

    ...

    [<modifikatori_metodeK>] <identifikator_metodeK>
([<parametri_metodeK>]) {
        //Telo metode K
    }
}
```

gde je:

- `[<modifikatori_klase>]` -> modifikatori klase.
- `<imeKlase>` -> naziv naše klase napisan po PascalCase konvenciji.
- `<tip_poljai> <identifikator_poljai>` -> neko `i` polje klase.
- `<modifikator_vidljivosti_konstruktorai>` -> neki od modifikatora vidljivosti za konstruktor klase.
- `[<parametri_konstruktorai>]` -> neki parametri konstruktora `i` klase.
- `[<modifikatori_metodei>]` -> modifikatori metode `i` klase.
- `<identifikator_metodei>` -> ime neke metode `i` klase.
- `[<parametri_metodei>]` -> neki parametri metode `i` klase.

ili šablon u čitljivijem zapisu:

```
[<modifikatori_klase>] class <imeKlase> {
    //Polja klase

    //Konstruktori klase

    //Metode klase
}
```

gde su:

- `//Polja klase` -> neka polja klase. Može ih biti 0.
- `//Konstruktori klase` -> neki konstruktori klase. Može ih biti 0.
- `//Metode klase` -> neke metode klase. Može ih biti 0.

Naravno, redosled pisanja ovih 3 grupa, kao i redosled unutar pisanja promenjivih/metoda po tim grupacija kao i njihovo mešanje nije važno i ne utiče na izvršavanje samog koda.
Ostalo je još da definišemo šta su ovi `konstruktori` klase.

Definicija 8.4 (Konstruktori klase)**.**

Konstruktori klase su specijalne metode klase, čiji je naziv uvek identičan imenu klase i nemaju povratnu vrednost (pišu se čak i bez `void` ključne reči), koje se pozivaju prilikom kreiranja objekata date klase i opisuju njihov način kreacije.

Klase možemo posmatrati i kao nove tipove podataka koje mi definišemo; takve da sa sobom vuku odredjen broj promenjivih i

funkcija, no možda je bolje posmatrati ih kao šablon po kome definišemo relacije izmedju nekih promenjivih i nekih funkcija.

Pogledajmo sve šta smo do sada napisali u teoriji na jednom praktičnom primeru.

Zadatak 8.2 (Čovek)**.**

Kreirati klasu *Covek* koja opisuje čoveka. Svaki čovek ima svoje ime, prezime i broj godina i svaki čovek može da kaže "Ja sam <ime> <prezime> i imam <godine> godina".

```
package covek;

public class Covek {

    //Atributi/Polja klase
    String ime;
    String prezime;
    int godine;

    //Konstruktor
    public Covek(String i, String p, int g) {
        ime = i;
        prezime = p;
        godine = g;
    }

    //Metoda klase
    public void pricaj() {
        System.out.pritnln("Ja sam " + ime + " " + prezime
+ " i imam " + godine + " godina.");
```

```
    }
}
```

Dakle ovde smo izneli jedan *šablon* po kome se ponaša jedan *Čovek*:

- ima svoje `ime`, `prezime` i `godine`
- kreira se tako što mu mi zadamo `ime`, `preizme` i `godine`
- može da priča pomoću metode `pricaj`

Prodjimo detaljno kroz priložen kod.
Kao i do sada, ceo napisan kod se nalazi unutar kreirane klase, `Covek`. Atributi klase su definisani, tačnije deklarisani. Nakon toga, kreirali smo jedan `konstruktor` klase `Covek`. Kao što smo rekli, to je jedna `metoda` bez ikakve povratne vrednosti (čak ne koristimo ni ključnu reč `void`) koja nam govori na koji način kreiramo `objekte` ove klase. Kao i svaka druga metoda, `konstruktor` može (a i ne mora) da primi proizvoljan broj proizvoljnih parametara. Pošto mi želimo da inicijalizujemo naša polja klase na neke vrednosti prilikom kreacije jednog čoveka, prosledjujemo mu dva `String` i jedan `int` parametara, koji će da inicijalizuju `ime`, `prezime` i `godine` čoveka koga kreiramo. Da neki od atributa nismo eksplicitno inicijalizovali, on bi se inicijalizovao na podrazumevanu vrednost prilikom pozivanja `konstruktora` (`ime` i `prezime` na `null`, a `godine` na `0`). Nakon toga, kreirali smo jednu `void` metodu (funkciju) koja ispisuje odredjenu poruku čiji tačan sadržaj zavisi od vrednosti atributa. Kako su ti atributi deklarisani u telu same klase, to znači da su vidljivi u svim pod-blokovima, odnosno svim ugnježdenim blokovima koda, a to su upravi **svi** mogući blokovi koda klase, pa su vidljivi i u `konstruktorima` i u `metodama` klase. Primetimo da kao modifikator za našu klasu, konstruktor i

metodu smo koristili ključnu reč `public` i to da ovaj put **nismo** koristili ključnu reč `static` prilikom pisanja zaglavlja naše metode. Nažalost, idalje moramo ove ključne reči da prihvatimo kao takve, ali ubrzo ćemo biti naoružani sa dovoljnim brojem informacija da i te modifikatore možemo da razumemo.

Pre nego što predjemo kreiramo objekte ove klase, ponovimo ovaj postupak da kreiramo klasu koja opisuje jedan pravougaonik, sada bez eksplicitnog zadatka, već zaključivanjem šta svaki pravougaonik treba da ima od atributa i kako treba da se ponaša, ondosno kakve metode da sadrži.

Jasno je da je svaki pravougaonik sačinjen od dva para naspramni stranica koje se ukrštaju pod pravim uglom i da svakom pravougaoniku možemo da izračunamo obim i površinu, dakle, naša klasa koja opisuje pravouganik će imati:

- Dva double atributa za stranice
- Metode `povrsina` i `obim`

```
package pravougaonik;

public class Pravougaonik {

    double a, b;

    public Pravougaonik(double stranica1, double stranica2)
{
        a = stranica1;
        b = stranica2;
    }

    public double obim() {
```

```
        double o = 2 * a + 2 * b;
        return o;
    }

    public double povrsina() {
        return a * b;
    }
}
```

Nalik rešenju prethodnog zadatka, analizom ovog koda vidimo da naša klasa ima dva atributa tipa `double` i jedan `konstruktor` koji prima dva `double` literala kao parametre i smešta ih atribute klase. Pored toga definisali smo i dve `metode`. U metodi `obim` kreiramo promenjivu `o` (koja je sada lokalna za tu metodu - taj blok u kome je kreirana - kao i inače) koju prosledjujemo kao povratnu vrednost. Metodu `povrsina` smo realizovali bez dodatne promenjive, vraćajući samo rešenje izraza `a * b`.

Sada kada smo videli kako možemo da kreiramo *klase*, predjimo na rad sa **objektima**, koji su neophodni za potpunu definiciju pojma *klase*.

Objekti

Definicija 8.4 (Objekat)**.**

Objekat neke klase predstavlja instancu te klase sa konkretnim vrednostima poljima date klase i konkretnim ponašanjima metoda. Kažemo da objekat *realizuje* svoju klasu.

Kako klasa predstavlja jedan *šablon*, nema nikakvog smisla pričati o vrednostima atributa klase[2] i konkretnim ponašanjima metoda

klase[2-1]. Jedino kada mi kreiramo jednu instancu te klase, kada je *realizujemo*, što radimo tako što joj dodelimo neke konkretne vrednosti za atribute, možemo da pričamo o njihovim vrednostima i načinu na koji se ponaša data kreirana stvar - objekat.
Po tom principu, ja mogu da tvrdim da će čitalac ove skripte sigurno imati svoje ime i prezime i da će umeti da čita srpski jezik, ali pre nego što upoznam jednog čitaoca, tj. pre nego što se on *realizuje*, ja ne mogu da znam kako će tačno glasiti to ime i prezime.
Da bi smo kreirali jedan **novi** objekat naše klase (za prvi primer uzećemo gore kreiranu klasu `Covek`) koristimo ključnu reč `new` koja očekuje jedan konstruktor date klase, i u memorijskom prostoru rezerviše neophodnu memoriju kako bi se ta operacija uspešno izvršila. Šablon kreiranja obejkata neke klase je:

```
<ime_klase> <identifikator_objekta> = new
<konstruktor_klase>([<neophodni_argumenti>]);
```

U ovom slučaju za `<identifikator>` ne kažemo da je `<promenjiva>` (mada možemo i tako gledati na stvari) već kažemo da je **objekat**, odnosno **instanca** klase `<ime_klase>`.
Ovaj zapis ne bi trebalo da nam bude stran. Naime, do sada smo ovakvu konstrukciju koristili prilikom kreiranja skenera i listi, pošto oni i jesu klase po sebi, pa smo i kreirali njihove objekte.
Na ovo možemo da gledamo i kao da smo kreirali jednu promenjivu tipa `<ime_klase>`.

Uz sva pravila koja prate primitivne promenjive (jedinstvenost identifikatora, njihova vidljivost itd.) poljima i metodama koje pripadaju objektima (koje smo definisali unutar klase) pristupamo sa separatorom `.`, tj. po šablonu:

```
<identifikator_objekta>.<atribut>
ili
<identifikator_objekta>.<metoda>
```

Naravno, i nešto poput ovoga ne bi trebalo da nam pravi previše pometnje jer smo upravo na ovaj način pristupali metodama poput `nextInt()`, `equalsIgnoreCase()`, `add()` ...

Važna napomena

> U odeljku koji se tiče Objektno Orijentisanog Programiranja većina klasa će pripadati jednom zajedničkom paketu i pored toga definisaćemo 'test' klasu koja će pored `main` metode biti prazna i koja će nam služiti da kreiramo objekte drugih klasa. Zato, umesto `nekiPaket` kao što smo pisali do sada, pisaćemo konkretan paket kome data klasa pripada, te je potrebno da vodimo računa kada posmatramo primere šta pripada kom paketu!

Kreirajmo konačno objekte naše klase `Covek` u praksi

```
package covek;

public class TestCovek {

    public static void main(String[] args) {
        String ime = "Alan";
        String prezime = "Turing";
        int godine = 41;

        Covek alanTuring = new Covek(ime, prezime, godine);

        System.out.println("Ime oca racunarstva je: " +
```

```
alanTuring.ime + " " + alanTuring.prezime);
        alanTuring.pricaj();

        Covek alonzoChurch = new Covek("Alonzo", "Church",
92);
        alonzoChurch.pricaj();
    }
}
```

Linije koda 6-8 pripremaju neophodne podatke za kasnije kreiranje objekta klase `Covek`. U liniji koda 10, kreiramo jedan **objekat** klase `Covek` sa identifikatorom `alanTuring` pozivajući `konstruktor` klase `Covek`, koji prima tri argumenta i vrednosti tih argumenata smešta redom u `ime`, `prezime` i `godinu` atribute koji sada pripadaju **objektu** `alanTuring`. Dakle, ovim putem smo *realizovali* klasu `Covek` i kreirali jedan *konkretan* primerak te klase, tj. jednog konkretnog čoveka. Slično tome, u liniji koda 15 kreiramo **novi** objekat klase `Covek` pod indetifikatorom `alonzoChurch` [3] s time što smo ovaj put direktno prosledili literale `kontrsuktoru`. Sa ovim načinima pozivanja funkcija smo već uveliko upoznati.
U liniji koda 12 štampamo poruku `"Ime oca racunarstva je: Alan Turing"`, tako što pristupamo poljima objekta `alanTuring` preko `.` separatora, potpuno nalik pristupanju polja `.length` kod nizova. Linija koda 13 poziva `void` metodu **objekta** `alanTuring` koja se zove `pricaj()` po istom principu kao što smo pozivali metode skener objekta, metode niski i metode listi. U tom momentu, metoda `pricaj()` se izvršava i štampa se poruka `"Ja sam Alan Turing i imam 41 godina."`. Ista metoda se poziva i nad **objektom** `alonzoChurch` u liniji 16 koja ovaj put štampa poruku sa vrednostima atributa objekta `alonzoChurch`, što će biti `"Ja sam Alonzo Church i imam 92 godina."`.

Primetimo važnu činjenicu da smo sve ovo postigli u `main` metodi koja se nalazi u fajlu (a svaki fajl je klasa za sebe u Javi!) **izvan** klase `Covek`. Ovo je bilo moguće jer smo klasu `Covek` definisali sa `public` modifikatorom (više o modifikatorima kasnije), što znači da možemo da joj pristpumi iz celog projekta! Upravo ovakvo ponašanje klasa nam omogućava da ih tretiramo i koristimo kao korisnički-definisane tipove podataka!
Ukoliko se nalazimo u istom paketu kao i fajl klase, ili u nekom podpaketu tog paketa, nema potrebe za ekpslicitnim importovanjem tog fajla, ali ako želimo da kreiramo objekat neke klase koja je definisana u fajlu koji se nalazi u drugačijem paketu, moramo ga uvesti pomoću ključne reči `import`, kao što smo to radili i za klase `Scanner`, `ArrayList` itd.

Kreirajmo sada objekte klase `pravougaonik`

```
package pravougaonik;

public class TestPravougaonik {

    public static void main(String[] args) {
        Pravougaonik p = new Pravougaonik(3, 4);
//Pravougaonik stranica 3 i 4
        Pravougaonik k = new Pravougaonik(5, 5);
//Pravougaonik stranica 5 i 5 - kvadrat

        double obimOdP = p.obim();
        double povrsinaOdP = p.povrsina();

        System.out.println("Obim od p: " + obimOdP);
        System.out.println("Povrsina od p: " +
povrsinaOdP);
```

```
        q.a = 6;
        q.b = 6;
        double stranicaAOdQ = q.a;
        stranicaAOdQ = 7;

        System.out.println("Obim od q: " + q.obim());
        System.out.println("Povrsina od q: " +
q.povrsina());
    }
}
```

U liniji koda 6 kreirali smo jedan `Pravougaonik` `p` sa stranicama 3 i 4, dok smo u liniji koda 7 kreirali jedan `Pravougaonik` `k` sa istim stranicama dužine 5, esencijalno, jedan kvadrat.
U linijama koda 9 i 10 izračunali smo obim i površinu pravougaonika `p` pozivajući odgorvarajuće metode sa `double` povratnom vrednošću i smestili smo je u promenjivu. Zatim smo ispisali poruke:

- `"Obim od p: 14"`
- `"Povrsian od p: 12"`
  Linije 15 i 16 pristupaju poljima objekta `q` i menjaju vrednost obe stranice na 6. Linija koda 17 kreira novu `double` promenjivu čija vrednost je vrednost stranice `a` pravougaonika `q` , dakle 6. U narednoj liniji koda menjamo promenjivu `stranicaAOdQ` na 7, ali **ovo se ne odražava na polje a objekta q**, jer smo u liniji 17 kopirali samo **literal** tipa `double` koji je smešten u `q.a` ! Zatim se ispisuju poruke:
- `"Obim od q: 28"`
- `"Povrsina od q: 49"`

Naravno, ukoliko to vidljivost dozvoljava, a `public` vidljivost je najmanje restriktivna, mi u jednom fajlu možemo pristputi i više različitm klasama:

```
package nekiPaket;

import covek.Covek;
import pravougaonik.Pravougaonik;

public class TestCovekPravougaonik {


    public static void main(String[] args) {
        Covek c = new Covek("Stephen", "Kleene", 85);
        Pravougaonik p = new Pravougaonik(2, 3);
    }
}
```

Pored toga, dodavanjem `[]` nakon imena klase prilikom kreiranja objekta te klase, kreiramo jedan *niz* objekata te klase:

```
package nekiPaket;

import covek.Covek;

public class TestNizObjekata {


    public static void main(String[] args) {
        Covek covek1 = new Covek("Ada", "Lovelace", 36);
        Covek covek2 = new Covek("Haskell", "Curry", 81);
        Covek covek3 = new Covek("Donald", "Knut", 87);

        Covek[] ljudi = new Covek[]{covek1, covek2,
```

```
covek3};

        Covek[] drugiLjudi = new Covek[]{
            new Covek("Edsger", "Dijkstra", 72),
            new Covek("Tony", "Hoare", 91),
            new Covek("Stephen", "Cook", 85),
            new Covek("John", "Backus", 82)
        };

        Covek adaLovelace = ljudi[0];
        System.out.println("Otac analitike algoritama je "
+ ljudi[2].ime + " " + ljudi[2].prezime);
    }
}
```

Svi kreirano objekti se nalaze u delu memorije koji nazivamo `heap`[4] koju sve funkcije medjusobno dele, te ukoliko prosledjujemo objekte kao argumente funkcije, mi prosledjujemo pokazivač na memorijsku lokaciju u `heap` memoriji. Drugim rečima, promenjive "objektnog" tipa su pokazivači, odnosno reference!
S toga, naredni kod:

```
package covek;

public class IspisObjekata {

    public static void main(String[] args) {
        Covek c = new Covek("A", "B", 20);
        System.out.println(c);
    }
}

Output: Covek@<heksadecimalan_broj>
```

ispisuje samo memorijsku lokaciju našeg objekta.

Ključna reč null

Do sada smo se susretali sa posebnom ključnom rečju `null` za koju smo rekli da označava `"ništa"` ali nismo mogli da objasnimo tačno kako ona funkcioniše jer njeno ponašanje je usko vezano za objektno orijentisano programiranje.

Definicija 8.5 (Ključna reč null)**.**

Ključna reč `null` je pokazivač na specijalnu memorijsku lokaciju na kojoj nije ništa smešteno i na koju se ne može ništa smestiti.

Ova ključna reč označava da je vrednost jednog objekta "ništa", tj. da taj objektni pokazivač pokazuje ni na šta. `null` vrednost je jedinstvena i svi objekti je dele. Tako na primer:

```
<Klasa1> <objekat1> = null;
<Klasa2> <objekat2> = null;
```

objekti `objekat1` i `objekat2` pokazuju na **istu** memorijsku lokaciju u memoriji iako su potencijalno objekti različitih klasa!

Napomenimo da postoji bitna razlika izmedju pojmova "nepostojanja" i "imanja ničega" u programiranjau! Iako je ovo kontraintuitivno od onoga što nam matematika nalaže (ako u matematici kažemo da neka čestica ima naelektrisanje 0, to je isto kao da kažemo da ta čestica nema naleketrisanja), u programiranju je vrlo važno da možemo da razlikujemo kada nam neke informacije uopšte nisu dostupne od slučaja kada su te informacije dostupne, ali one su ništa. Tako na primer, ako neko domaćinstvo

nema ni jedan čaj u svom kabinetu, rekli bi da to domaćinstvo "nema" čaja, dok za domaćinstvo koje ima kutiju čaja u svom kabinetu koja je prazna, kažemo da "imaju ništa" čaja.[5]

```
package covek;

public class NullPokazivac {

    public static void main(String[] args) {
        Covek c1 = null;
        Covek c2;

        System.out.println(c1);
        System.out.println(c2);
    }
}
```

U gornjem kodu smo kreirali objekat klase `Covek` po imenu `c1` koji je ništa. Priliko izvršavanja komande na liniji 9 ispisala bi se poruka `"null"`. Ovo je potpuno drugačije od obične deklarizacije objekta klase `Covek` `c2`! Naime, objekat `c1` ima vrednost, i to je upravo pokazivač na `null`, dok objekat `c2` je samo deklarisan i nije inicijalizovan, te s toga, linija koda 10 izbacuje kompajlersku grešku!

Ključna reč this

Prisetimo se u poglavlju koje se ticalo funkcija kako promenjiva koju prosledjujemo kao argument nema nikakve veze sa identifikatorom koji smo koristili kao parametar funkcije.

```
package nekiPaket;

public class NekaKlasa {

    public static void f(int x) {
        return;
    }

    public static void h() {
        int x = 5;
        f(x);
    }
}
```

Promenjiva `x` definisana unutar `h` funkcije nema nikakve veze sa promenjivom `x` koja je parametar funkcije `f`.

Slično tome, ukoliko parametre `konstruktora` klase nazovemo tako da im se identifikatori podudaraju sa identifikatorima atributa te klase, opet će se raditi o potpuno nepovezanim promenjivima. Posmatrajmo naredni kod:

```
package nekiPaket;

public class NekaKlasa {

    int x;
    double d;

    public NekaKlasa(int x, double d) {
        //hmm?
        x = x;
        d = d;
```

```
    }
}
```

Dakle, želimo da kreiramo konstruktor koji će primiti jedan `int x` i jedan `double d` i smestiti ih u vrednosti polja klase `x` i `d`. Nama kao bićima sa intuiciom jasno je da se `x` i `d` sa leve strane operatora dodele u `konstruktoru` odnose na polja klase, dok se `x` i `d` sa desne strane operatora dodele odnose na parametre `konstruktora`. Kako je `konstruktor` funkcija kao i bilo koja druga (sa nekim specijalnim ponašanje doduše) to u njoj može da se nadju komande koje mogu da se nadju i bilo kojoj drugoj funkciji (grananje, petlje, pozivi drugih funkcija itd.). Tada, može doći do zabune o kojoj promenjivoj je reč: onoj koja je atribut klase, ili onoj koja je parametar funkcije. Štavise, gornji kod i ne postavlja nikakve vrednosti za polja klase jer pristupa vrednostima parametara i dodeljuje ih istim promenjivma (efektivno ne menja ništa)! Da bi se ova distinkcija napravila, koristimo ključnu reč **this**.

Definicija 8.6 (Ključna reč this)**.**

Ključna reč `this` odnosi se na 'ovaj', konkretan primerak objekta, tj. na njegovu referencu.

Pre nego što dalje objasnimo ponašanje ove ključne reči, prepravimo prethodni primer tako zapravo dodeljuje prosledjene vrednosti atributima klase.

```
package nekiPaket;

publi class NekaKlasa {
```

```
    int x;
    double d;

    public NekaKlasa(int x, double d) {
        this.x = x;
        this.d = d;
    }

    public void promeniX(int a) {
        this.x = a;
    }

    public void promeniD(double b) {
        d = b;
    }
}
```

Sada, sa ključnom rečju `this` u linijama 9 i 10, sa leve strane operatora dodele pristupamo promenjivima `x` i `d` vezanih za **ovaj** objekat koji poziva `konstruktor`, dok sa desne strane operatora dodele pristupamo parametrima koji su prosledjeni kao argumenti prilikom poziva funkcije!

Kako je `this` referenca na objekat, sa deliminatorom `.` pristupamo atributima, odnosno metodama objekta na koji se odnosi `this`.

Ukoliko Java može jednoznačo da napravi distinkciju da li je reč o atributu klase ili o parametru metode, nije neophodno pisati ključnu reč `this`. Tako u liniji 14 možemo ali nije neophodno da pišemo ključnu reč `this` jer Java jednoznačno može da zaključi da se `x` odnosi na polje klase. Slično, u liniji 18, Java jednoznačno može da

odredi da se `d` odnosi na polje klase, te nema potrebe za sintagmom `this.d`.

Pogledajmo kako se ovo realizuje prilikom krearianja objekata ove `NekaKlasa` klase:

```
package nekiPaket;

public class TestNekaKlasa {

    public static void main(String[] args) {
        NekaKlasa nk = new NekaKlasa(5, 3.14);
        nk.promeniX(10);
    }
}
```

Prilikom poziva `NekaKlasa(5, 3.14)` konstruktora, ključna reč `this` će se odnositi baš na objekat `nk`, te linije 9 i 10 iz fajla sa klasom možemo posmatrati kao:

```
nk.x = x; //tj. nk.x = 5
nk.d = d; //tj. nk.d = 3.14
```

a prilikom poziva metode `promeniX(10)` kao:

```
nk.x = a; //tj. nk.x = 10
```

Preopterećivanje metoda

U Javi, svaki metod je moguće **preotperetiti** i **najdačati**. O **nadjačavanju** metoda pričaćemo u poglavlju koje se tiče

nasledjivanja, a sada ćemo se fokusirati na **preopterećivanje** metoda.

Definicija 8.7 (Preopterećivanje metoda)**.**

Preopterećivanje metoda (eng. method overloading) pretstavlja proces definisanja više različitih metoda sa istim identifikatorom koje se razlikuju po tipu ili broju ili redosledu parametara i eventualno po povratnoj vrednosti.

Radi lakšeg razumevanja ovog jako bitnog koncepta za objektno orijentisano programiranje, pogledajmo jednostavan zadatak nevezan za OOP koncepte.

Zadatak 8.3 (Negiranje)**.**

Napisati funkciju:

- koja prima jedan ceo broj i vraća njegovu negiranu vrednost.
- koja prima jedan realan broj i vraća njegovu negiranu vrednost.
- koja prima jedanu logičku konstantu i vraća njenu negiranu vrednost.

Ovo se rešava vrlo lako, definisanjem triju funkcija:

```
package nekiPaket;

public class Negacije {

    public static int negirajInt(int x) {
        return -x;
    }
```

```
    public static double negirajDouble(double d) {
        return -d;
    }

    public static boolean negirajBoolean(boolean b) {
        return !b;
    }

}
```

Jasno je da ove tri funkcije rade po istom principu, od kojih dve imaju identično telo do na naziv parametra a treća ima sintaksno drukačije telo, dok semantički obavlja isti postupak. Idealno bi bilo da možem oda imamo *jednu* metodu koja u zavisnosti od tipa promenjive koju primi, obavlja odredjeni postupak i vraća odgovarajući tip. Ovo je upravo moguće sa preopterećivanjem metoda. Kako nam preotprerećivanje omogućava da kreiramo metode sa istim imenom, dokle god se one razlikuju po broju argumenata, njihovim redosledom ili njihovim tipom, to možemo da definišemo jednu metodu `negiraj` na tri različita načina:

```
package nekiPaket;

public class NegirajOpterecivanje {

    public static int negiraj(int x) {
        return -x;
    }

    public static double negiraj(double d) {
        return -d;
    }
```

```
    public static boolean negiraj(boolean x) {
        return !x;
    }

    public static void main(String[] args) {
        int x = 5;
        double d = -3.14;
        boolean b = true;

        System.out.println(negiraj(x));
        System.out.println(negiraj(d));
        System.out.println(negiraj(b));
    }
}
Output:
-5
3.14
false
```

Na ovaj način, kreirali smo tri različite implementacije funkcije `negiraj`. Jedna koja radi sa celim brojevima, druga sa realnim a treća koja radi sa `boolean` vrednostima. Možemo čak primetiti i da identifikatori parametara mogu da se zovu isto, dokle god su različitog tipa! Odgovor na to kako Java zna koju implementaciju treba da obradi prilikom poziva funkcije je vrlo prost - gledanjem argumenata[6].
U liniji 22, Java vidi da pokušavamo da prosledimo jedan `int` kao argument našoj funkciji `negiraj`, te ona zna da treba da obradi funkciju definisanu linijama 5-7. U liniji 23 vidi da je prosledjen jedan `double` broj, pa ulazi u funkciju definisanu linijama 9-11. Slično i za `boolean`.
Baš zbog ovakvog ponašanja zaključujemo da **ne možemo**

nadjačati funkciju sa istim brojem parametara koji su istog tipa i istog redosleda (nezavisno od njihovog imenovanja i povratne vrednosti). Dakle:

- Dozvoljeno:

```
<povratnaVrednost1> f(<tip1> <parametar1>)
<povratnaVrednost1> f(<tip2> <parametar2>)

i

<povratnaVrednost1> f(<tip1> <parametar1>)
<povratnaVrednost1> f(<tip2> <parametar1>)

i

<povratnaVrednost1> f(<tip1> <parametar1>)
<povratnaVrednost2> f(<tip3> <parametar3>)

i

<povratnaVrednost1> f(<tip1> <parametar1>)
<povratnaVrednost1> f(<tip1> <parametar1>, <tip2>
<parametar2>)

i

<povratnaVrednost1> f(<tip1> <parametar1>)
<povratnaVrednost1> f(<tip1> <parametar1>, <tip1>
<parametar1>)
```

```
itd.
```

- Zabranjeno:

```
<povratnaVrednost1> f(<tip1> <parametar1>)
<povratnaVrednost2> f(<tip1> <parametar1>)

i

<povratnaVrednost1> f(<tip1> <parametar1>)
<povratnaVrednost2> f(<tip1> <parametar2>)
```

Ovaj proces preopterećivanja metoda je izuzetno koristan kada želimo da omogućimo korisnicima naših klasa da kreiraju objekte tih klasa na različite načine, dakle, najčešće ćemo koristi ovaj proces da preopteretimo konstruktor, tj. da imamo više konstruktora u našoj klasi.

Vratimo se na primer pravougaonika. Setimo se da prilikom kreiranja objekata te klase, kreirali smo jedan pravougaonik sa jednakim stranicama (jedan kvadrat) tako što smo prosledili istu vrednost za obe stranice. To smo morali da uradimo zato što jedini konstruktor koji postoji je konstruktor koji prima dva realna broja. Omogućimo sada korisnicima naše klase da proslede samo jedan realan broj konstruktoru i u tom slučaju kreirajmo jedan pravougaonik istih stranica (kvadrat).

```
package pravougaonik;

public class Pravougaonik {
```

```
    double a, b;

    public Pravougaonik(double stranica1, double stranica2)
{
        a = stranica1;
        b = stranica2;
    }

    public Pravougaonik(double stranica) {
        a = stranica;
        b = stranica;
    }

    public double obim() {
        double o = 2 * a + 2 * b;
        return o;
    }

    public double povrsina() {
        return a * b;
    }
}
```

Sada, ukoliko želimo da kreiramo pravougaonik sa jednakim stranicama, to možemo da uradimo na dva načina:

```
package pravougaonik;

public class TestPRavougaonik {

    public static void main(String[] args) {
        Pravougaonik kvadrat1 = new Pravougaonik(5, 5);
        Pravougaonik kvadrat2 = new Pravougaonik(5);
```

```
        Pravougaonik kvadrat3 = new Pravougaonik();
    }
}
```

Prilikom kreiranja objekta `kvadrat1` poziva se konstruktor definisan linijama 7-10, dok prilikom kreiranja objekta `kvadrat2` poziva se konstruktor definisan linijama 12-15.
Linija koda 8 poziva takozvani **podrazumevani** (eng. **default**) konstruktor koji je dostupan svakoj klasi čak i bez njegovog eksplictinog navodjenja i postavlja sva polja na njihove podrazumevane vrednosti (potpuno analogno nizovima). No u praksi se podrazumevani konstruktor eksplicitno navodi, te ponovo menjamo našu klasu `Pravougaonik` da sadrži i taj konstruktor:

```
package pravougaonik;

public class Pravougaonik {

    double a, b;

    public Pravougaonik(double stranica1, double stranica2)
{
        a = stranica1;
        b = stranica2;
    }

    public Pravougaonik(double stranica) {
        a = stranica;
        b = stranica;
    }

    public Pravougaonik() {}
```

```
    public double obim() {
        double o = 2 * a + 2 * b;
        return o;
    }

    public double povrsina() {
        return a * b;
    }
}
```

Preopterećivanje metoda smo uveliko koristili kada smo pozival `println()` metod. Toj metodi smo prosledjivali cele brojeve, realne brojeve, karaktere, niske itd; a svaki put dobijali ispis na konzolnom prozoru.

Iako najčešće koritimo preopterećivanje metoda kada pišemo konstruktore, to nije proces ekskluzivan njima, i po potrebi ćemo to raditi i sa drugim metodama klase.

toString metod - Serijaliizacija

Često kada kreiramo neku aplikaciju želimo da se uverimo da prikupljamo i ažuriramo podatke na prikladan način. Najjednostavniji vid provere je da ispišemo vrednosti željenih promenjivih i to možemo uraditi na ogroman broj načina: ispisati uh neku datoteku, prikazati ih na nekom grafičkom korisničkom interfejsu, ili pak običan konzolni ispis kao što smo to radili u navedenim primerima. U svakom slučaju, potrebno je formarati neki literal tipa `String`. Proces kojim prikupljamo neke podatke i od njih formiramo neku nisku naziva se **proces serijaliizacije** podataka.

Definicija 8.8 (Serijaliizacija)**.**

Pod serijaliziacijom podataka podrazumevamo proces formiranja niske od prikupljenih podataka.

U Javi postoji "ugradjena" funkcija koja se tiče serijaliizacije objekata klase i to je **toString()** metod koga treba nadjačati. Kreaciom metode `String toString()` u našoj klasi opisujemo na koji način serliizujemo naše podatke i ona se uopšteno piše na sladeći način:

```
@Override
public String toString() {
    //Kod metode
    return <neki_String_literal>;
}
```

`@Override` je *anotacija* koja nalaže da je upitanju *nadjačavanje* metode. O ovim pojmovi ćemo detljanije pričati u poglavlju koje se tiče *nasledjivanja* jer se tada ti koncepti mogu na prirodan način objasniti.
Vidimo da je metoda `toString()` jedna metoda tipa `String`, te da treba da vrati neki `String` literal. Naravno, izgled tog literala i način njegove formacije je ostavljen na nama.

Radi lakšeg kreiranja niski, optimalnijeg nadovezivanja i formatiranja, možemo koristiti `StringBuilder` klasu, koja za lako prikuplja i obradjuje podatke, a sa svojom metodom `toString()` vraća nisku od prikupljenih podataka. `StringBuilder` je deo `java.lang` paketa te ga ne treba eksplicitno uvoditi.

Serijaliizujmo klase `Covek` i `Pravougaonik`. Kako je proces serijalizacije proizvoljan, tj. nije nešto što je strogo definisano, na

nama je da se dogovorimo kako želimo da serijalizujemo naše podatke. Jedan od mogućih je:

- `Covek` serijalizujemo tako da dobijemo poruku `"Ja sam <ime> <prezime> i imama <godine> godina."` (baš sta ispisuje `pricaj()` metoda)
- `Pravougaonik` serijalizujemo tako da dobijemo poruku:

```
Pravougaonik stranica <a> i <b>
Obim: <obim>
Povrsina: <povrsina>
```

```
package covek;

public class Covek {

    //Atributi/Polja klase
    String ime;
    String prezime;
    int godine;

    //Konstruktor
    public Covek(String ime, String prezime, int godine) {
        this.ime = ime;
        this.prezime = prezime;
        this.godine = godine;
    }

    //Metoda klase
    public void pricaj() {
        System.out.pritnln("Ja sam " + ime + " " + prezime
+ " i imam " + godine + " godina.");
    }
```

```
    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();

        sb.append("Ja sam ").append(ime).append(" ");
        sb.append(prezime).append(" i imam ");
        sb.append(godine).append(" godina.");

        return sb.toString();
    }
}
```

```
package pravougaonik;

public class Pravougaonik {

    double a, b;

    public Pravougaonik(double stranica1, double stranica2)
{
        a = stranica1;
        b = stranica2;
    }

    public Pravougaonik(double stranica) {
        a = stranica;
        b = stranica;
    }

    public Pravougaonik() {}

    public double obim() {
        double o = 2 * a + 2 * b;
```

```
        return o;
    }

    public double povrsina() {
        return a * b;
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();

        sb.append("Pravougaonik stranica
").append(a).append(" i ").append(b).append("\n");
        sb.append("Obim: ").append(obim());
        sb.append("Porvsina: ").append(povrsina());

        return sb.toString();
    }
}
```

`StringBuilder` u sebi sadrži polje tipa `String` koje popunjava metodama `append()` koje dodaju na tu nisku prosledjen parametar. U linijama 33 i 34 u klasi `Pravougaonik` vidim oda pozivamo metode `obim` i `povrsina` tim redom, što znači da se te metode izvršavaju i dodajemo njihove `povratne vrednosti` na nisku `StringBuilder`-a.

Definisanje metode `toString()` dodatno menja ponašanje `println()` metode kada joj prosledimo objekat naše klase. Sada, `println()` metoda prvo poziva `toString()` metodu prosledjenog objekta, pa štampa povratnu nisku koju `toString()` metod vrati. Takodje, nije neophodno naglašavati je prilikom poziva `println()` metode.

```
Pravougaonik p = new Pravougaonik(3, 4);
System.out.println(p);
```

```
Output:
"Pravougaonik strainca 3 i 4
Obim: 14
Povrsina: 12"
```

Modifikatori vidljivosti

Došli smo konačno do dela gde možemo da objasnimo ključne reči koje su se ticale modifikatora vidljivosti, odnosno modifikatora pristupa. Ovi modifikatori nam omogućavaju da upotrebu nekih atributa ili metoda jedne klase uklonimo za neku drugu klasu koja pokušava da pristupi tim atributima ili metodama. Ovi modifikatori mogu da se dodele atributima klase, metodama klase kao i samim klasama i razlikujemo modifikatore:

- `default (package-private)`
- `public`
- `protected`
- `private`
  s time da za klase, možemo koristiti samo `default` i `public` modifikator.

Za početak, fokusirajmo se samo na atribute i metode klase. Izložimo koje sve druge klase mogu da pristupaju ovim podacima u zavisnosti od modifikatora koji stoji uz njih:

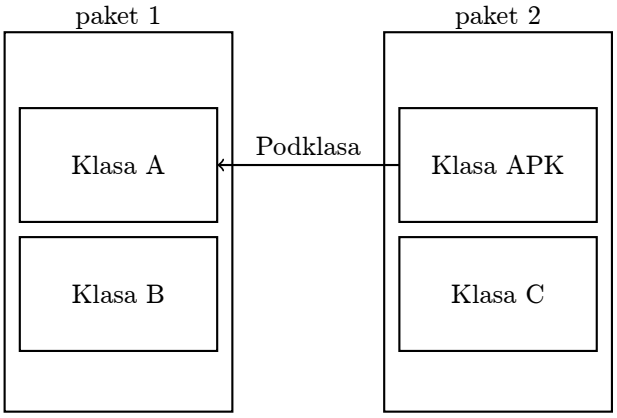
	`public`	`protected`	`default`	`private`
Ista klasa	✓	✓	✓	✓
Isti paket - podklasa	✓	✓	✓	✗
Isti paket - druga klasa	✓	✓	✓	✗
Drugi paket - podklasa	✓	✓	✗	✗
Drugi paket - druga klasa	✓	✗	✗	✗

Dakle, vidimo da klasa može da pristupa svim svojim elementima nezavisno od njihovog modifikatora - što je i intuitivno ponašanje. Modifikator `protected` štiti elemente klasa od drugih klasa koje nisu podklase van paketa, pa kažemo da su ovi elementi `package-protected`.
Podrazumevani modifikator, koji se u Javi eksplicitno ne naglašava[7], koga još nazivamo i `package-private` upravo radi na način na koji i njegovo drugo ime nalaže. Naime, on nalaže da su elementi privatni za klasu i sve njene podklase, dokle god se one nalaze u istom paketu (i svim podpaketima).
Privatni modifikator je najviše restriktujući od svih, i on nam omogućava da elemente klase zaštitimo od svih drugih klasa!

Do sada smo vidjali da su naše klase sve sa modifikatorom `public`, ali smo naglasili da one mogu biti i `default`, odnosno `package-private`. Razlika je u tome što objekte `public` klase možemo kreirati bilo gde u našem projektu, dok za `default` klase, možemo kreirati samo u istom paketu (i svim podpaketima).

Razmotrimo naredni slučaj. U našem projektu imamo dva paralelna paketa, `paket1` i `paket2`, oba sa dve klase ponaosob, `KlasaA`, `KlasaB`, `Klasa APK`, `Klasa C` redom. Klasa `Klasa APK` je podklasa

(tj. proširuje klasu) klase `Klasa A`, dok ostale klase nemaju veze sa klasom `Klasa A`, što je grafički prikazano slikom ispod:



Prikažimo koji sve elementi klase `Klasa A` su vidljivi ostalim klasama u zavisnosti od njihovog modifikatora tabelom:

Modifikator	Klasa A	Klasa B	Klasa APK	Klasa C
`public`	✓	✓	✓	✓
`protected`	✓	✓	✓	✗
`default`	✓	✓	✗	✗
`private`	✓	✗	✗	✗

U ovom delu poglavlja pominjali smo podsta pojam `podklasa`, kojeg ćemo bolje objasniti u narednom poglavlju.

Enkapsulacija

Modifikatori pristupa, tj. vidljivosti nam omogućavaju da **enkapsuliramo**, tj. učaurimo neke od atributa i polja klase tako da zabranimo njihov pristup korisnicima naše klase. Da bi jasno razumeli poentu ovog potpuno semantičkog postupka pogledajmo jedan fiktivan scenario:
Recimo da neka lokalna bolnica želi da digitalizuje svoje podatke i da omogući lakši i bolji rad svojim zaposlenima. U tu svrhu, unajmi

privatnu firmu koja će dizajnirati software u Javi po nalogu firminog poslodavca, te bolnice. Kako takav software zahteva ozbiljan pristup implementaciji, kako zbog svoje važnosti tako i zbog svog obima, u toj firmi se zadaci delegišu po timovima; pa tako jedan tim je zadužen za front-end GUI, drugi za bazu podataka, i više timova za back-end software. Jedan od tih timova je zadužen za kreiranje dela koda koji se tiče digitalizovanja pacijenata, i za potrebe ovog primera, neka je to jedna klasa `Pacijent` čija su sva polja i metode sa `public` modifikatorom. Dakle nešto poput:

```
package pacijent

public class Pacijent {

    public String ime, prezime;
    public int godine;
    public String dijagnoza;

    public Pacijent(String ime, String prezime, int godine,
String dijagnoza) {
        this.ime = ime;
        this.prezime = prezime;
        this.godine = godine;
        this.dijagnoza = dijagnoza;
    }
}
```

S druge strane, postoji tim koji se bavi kreiranjem dela programa za obradu pacijenata u kojem se nalazi neobazrivi Neša. Neobazrivom Neši je zadat zadatak da kreira neku funkciju `f` za rad sa pacijentima koja nije komplikovana, ali radi dosta stvari i veoma je dugačka i on ju je implementirao na sledeći način:

```java
package obrada;

import pacijent.Pacijent;

public void f(Pacijent p) {
    //Veoma dugacak kod
    p.godine = 15;
    //ostatak veoma dugackog koda
}
```

U svojoj neobazrivosti, Neša je nepotrebno ubacio liniju koda prikazanu kao linija 7 u primeru iznad, koja prolsedjenom pacijentu stavlja broj godina na 15. Kako je promenjiva `p` tipa `Pacijent`, to se ona prosledjuje kao referenca pa bilo koja izmena tog objekta `p` odraziće se i u funkciji koja poziva funkciju `f`, a to recimo može biti funkcija koja rasporedjuje pacijente po odeljenju.
Problem može nastati ako prosledjen pacijent nije maloletan. Tada nakon izvršavanja funkcije `f`, pacijentu se postavljaju godine tako kao da je maloletan pa ta pozivajuća funkcija ga smešta u pogrešan odsek, tj. šalje ga na kliniku za decu!

Ovako monumentalan propust možeda se zaobidje **enkapsulacijom** podataka, koja nas spašava od ovako neželjenih efekata.

Definicija 8.9 (Enkapsulacija)**.**

Pod enkapsulacijom podataka, odnosno njihovom učarivanju, podrazumevamo proces modifikovanja tih podataka sa `private` modifikatorom, čiji dalji pristup iz drugih klasa regulišemo (`get` i `set`) metodama.

Definicija 8.10 (Get metode)**.**

Pod `get` metodama, kolokvijalno `getterima`, podrazumevamo metode čija je povratna vrednost vrednost enkapsuliranog podatka.

Definicija 8.11 (Set metode)**.**

Pod `set` metodama, kolokvijalno `setterima`, podrazumevamo `void` metode koje primaju novu željenu vrednost enkapsuliranog podatka i čija svrha je adekvatno ažuranje tog podatka.

Po pravilu, kada pišemo `get` i `set` metode za neko polje `x`, nazivamo ih:

- `void setX(<neophodni_parametri>)`
- `<odgovarajuci_tip> getX()`
  Uglavnom, `<neophodni_parametri>` kod `setX` metode će biti samo nova vrednost za `x`, a `getX` metoda ne zahteva ni jedan parametar jer je njena uloga samo da vrati vrednost smeštenu u polje `x`.

Vratimo se nazad na klasu `Pacijent` i enkapsulirajmo sva njena polja; što podrazumeva:

- Postavljanje svih polja na `private`
- Kreiranje `set` metoda za svako polje
- Kreiranje `get` metoda za svako polje

Enkapsulacija podataka je vrlo univerzalan postupak do na izvedbu `set` i `get` metoda. Ove metode su najčešće postavljene sa

`public` ili `default` modifikatorom (mada ćemo se mi fokusirati samo na `public` modifikator u ovoj skripti).

```
package pacijent;

public class Pacijent {

    private String ime, prezime;
    private int godine;
    private String dijagnoza;

    public Pacijent(String ime, String prezime, int godine,
String dijagnoza) {
        this.ime = ime;
        this.prezime = prezime;
        this.godine = godine;
        this.dijagnoza = dijagnoza;
    }

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

    public void setPrezime(String prezime) {
        this.prezime = prezime;
    }

    public void setGodine(int godine) {
        this.godine = godine;
    }

    public void setDijagnoza(String dijagnoza) {
        this.dijagnoza = dijagnoza;
    }
```

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

    public String getPrezime() {
        return prezime;
    }

    public int getGodine() {
        return godine;
    }

    public String getDijagnoza() {
        return dijagnoza;
    }
}
```

Ovim putem, ne možemo *direktno* da pristupimo atributima nekog objekta, već moramo da pozivamo `get` metode svaki put kada želimo da prihvatimo neku vrednost atributa i da pozivamo `set` metodu sa odgovarajućim atributom ukoliko želimo da izmenimo vrednost tog atributa!

Sada, komanda:

```
p.godine = 15
```

izbacuje kompajlersku gresku, jer atribut `godine` nije dostupan ni jednoj klasi izvan `Pacijent` klase, pa ni klasi koju piše neobazrivi Neša kojaje čak izvan paketa u kome se nalazi `Pacijent` klasa.

S druge strane, ako ste obazirivi, ovim postupkom nismo ništa postigli ako Neša umesto priložene linije koda napiše:

```
p.setGodine(15);
```

Iako smo ilustrovali uopšteni postupak kako se podaći enkapsuliraju, vidimo da nismo zaobišli problem, već da smo samo malo otežali korisniku naše klase da unese nevalidan unos. To je zato što smo koristili naj opštije `settere`.
Jasno je da svi pacijenti stare kako vreme teče. Tako, prilikom postavljanja godina (na novi broj godina), treba se prvo zapitati da li je taj novi broj godina strogo veći od trenutnog, jer jedino to ima smisla. Dakle, korigovana enkapsulacija klase `Pacijent` izgleda:

```
package pacijent;

public class Pacijent {

    private String ime, prezime;
    private int godine;
    private String dijagnoza;

    public Pacijent(String ime, String prezime, int godine,
String dijagnoza) {
        this.ime = ime;
        this.prezime = prezime;
        this.godine = godine;
        this.dijagnoza = dijagnoza;
    }

    //Setteri
    public void setIme(String ime) {
        if (!ime.isBlank())
```

```
            this.ime = ime;
    }

    public void setPrezime(String prezime) {
        if (!prezime.isBlank())
            this.prezime = prezime;
    }

    public void setGodine(int godine) {
        if (godine > this.godine)
            this.godine = godine;
    }

    public void setDijagnoza(String dijagnoza) {
        if (!dijagnoz.isBlank())
            this.dijagnoza = dijagnoza;
    }

    //Getteri
    public String getIme() {
        return ime;
    }

    public String getPrezime() {
        return prezime;
    }

    public int getGodine() {
        return godine;
    }

    public String getDijagnoza() {
        return dijagnoza;
    }
}
```

Korigovali smo našu klasu na sledeći način:

- Ukoliko prosledjeno ime/prezime/dijagnoza nije prazna niska, postavi odgovarajuće polje na prosledjenu vrednost
- Samo ukoliko je prosledjen broj godina strogo veći od trenutnog, postavi broj godina na prosledjenu vrednost
- Inače, vrednosti ostaju takve kakve jesu

Sada, čak iako neobazrivi Neša pokuša da promeni broj godina pacijentu od recimo, 35 godina na 15, to se neće nikako odraziti na godine pacijenta, pošto će ostati stara vrednost od 35 godina, pa će funkcija rasporedjivanja pacijenta koja poziva funkciju `f` da rasporedi pacijenta `p` u odgovarajuću kliniku za punoletna lica.

Enkapsulacija podataka će često podrazumevati dodatan kod u telu svojih `set` metoda koji će se starati da prosledjen podatak zadovoljava semantičke kriterijume za taj atribut, pre menjanja vrednosti istog.

Od sada, makar to radili i najopštijem procesom enkapsulacije, sva polja naših klasa ćemo enkapsulirati sa `private` modifikatorom i kreirati odgovarajuće `set` i `get` metode.

Ključna reč static

Posmatrajmo naredni zadatak:

Zadatak 8.4 (Broj coveka)**.**

Napisati funkciju koja ispisuje koliko instanci klase `Pravougaonik` se kreiralo u programu.

Rešenje ovog problema naizgled deluje jednostavno. Potrebno je samo de definišemo jedan brojač, kojeg ćemo uvećavati za jedan svaki put kada se kreiramo novi objekat klase `Covek`.

```
package nekiPaket;

import pravougaonik.Pravougaonik;

public class NekaKlasaA {

    public static void brojPravougaonika(int br) {
        System.out.println("Kreirali smo " + br + "
objekata klase Pravougaonik");
    }

    public static void main(String[] args) {
        int brojObjekataPravougaonika = 0;
        Pravougaonik p = new Pravougaonik(3, 4);
        brojObjekataPravougaonika++;
        brojPravougaonika(brojObekataPravougaonika)
    }
}
```

```
OUTPUT: "Kreirali smo 1 objekata klase Pravougaonik"
```

Pre svega, ova funkcija `brojPRavougaonika` je redundantna jer prima jedan broj i ispisuje poruku sa tim brojem. Pored toga, šta ako postoji još jedan program koji kreira objekte klase `Pravougaonik`?

```
package nekiPaket;
```

```
import pravougaonik.Pravougaonik;
import nekiPaket2.NekaKlasaC;

public class NekaKlasaB {

    public static void main(String[] args) {
        Pravougaonik p = new Pravougaonik(5, 6);
        Pravougaonik c2 =
NekaKlasaC.kreirajMiPravougaonik();
    }
}
```

Očiglendo, broj obejkata klase `Pravougaonik` je 2, tj `brojObjekataPravougaonika` treba nekako da obraća pažnju na obe ove `main` metode, što nije moguće. Pored toga, u liniji 9 gornjeg koda vidimo da `NekaKlasaC` ima metodu koja kreira i vraća referencu na objekat klase `Pravougaonik` tokadje, koja u pozadini možda kreira i dodatna 3 objekata klase `Pravougaonik`, te ovaj naivni pristup jednostavno ne radi.
Da bi smo uspeli da rešimo ovakve probleme, potreban nam je **static** modifikator.

Definicija 8.12 (Ključna reč static)**.**

Statički modifikator `static` naglašava da metoda ili atribut uz koji stoji pripada samoj *klasi*, a ne *objektima* te klase.

Drugim rečima, ključna reč `static` definiše nedinamička polja i metode.
Videli smo da svaki naš objekat klase `Pravougaonik` ima sopstvene vrednosti za polja definisanih u klasi, kao i da ima sopstveno ponašanje metoda, koje dobijaju svoje konkretne vrednosti i

ponašanje prilikom kreiranja tih objekata.
S druge strane, kada bi smo želeli da dopunimo našu klasu `Pravougaonik` sa metodom koja daje tekstualni opis definicije pravougaonika, ta metoda bi bila identična za svaku klasu, pošto je opis "Pravougaonik predstavlja geometrijsku figuru sa dva para jednakih naspramnih medjusobno normalnih stranica.", univerzalan za svaki pravougaonik, te nema mnogo smisla da svaki objekat klase `Pravougaonik` ima posebnu instancu te metode. Umesto toga, takvu metodu možemo da proglasimo **statičkom** i na taj način vezati je za samo klasu `Pravougaonik`, a ne za njene objekte. U tom slučaju, imali bi smo samo jednu realizaciju te metode.

```
package pravougaonik;

public class Pravougaonik {

    double a, b;

    public Pravougaonik(double stranica1, double stranica2)
{
        a = stranica1;
        b = stranica2;
    }

    public Pravougaonik(double stranica) {
        a = stranica;
        b = stranica;
    }

    public Pravougaonik() {}

    public double obim() {
        double o = 2 * a + 2 * b;
        return o;
```

```
    }

    public double povrsina() {
        return a * b;
    }

    public static String opis() {
        return "Pravougaonik predstavlja geometrijsku
figuru sa dva para jednakih naspramnih medjusobno normalnih
stranica."
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();

        sb.append("Pravougaonik stranica
").append(a).append(" i ").append(b).append("\n");
        sb.append("Obim: ").append(obim());
        sb.append("Porvsina: ").append(povrsina());

        return sb.toString();
    }
}
```

Kako su statička polja i metode delovi samih klasa, a ne njenih objekata, to pristupanje tim poljima i metodama vršimo preko nazivom klase praćenim sa separatorom `.`

```
package pravougaonik;

public class TestPravougaonik {

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

```
        String opis = Pravougaonik.opis();
        System.out.println(opis);
    }
}
```

```
OUTPUT: "Pravougaonik predstavlja geometrijsku figuru sa
dva para jednakih naspramnih medjusobno normalnih
stranica."
```

Važno je napomentu da prilikom **prvog pominjanja klase** (bilo da je to zarad kreiranja objekata te klase, provervanja da li su neki objekti instance te klase, pokušaj pristupanju statičkim elementima ...) sve statičke metode se kreiraju i sva statička polja se inicijalizuju u memoriji!
Dakl,e prilikom prvog pominjanja klase `Pravouganik` u lini 6 sa `Pravouganik.`, metoda `opis()` se kreira, zatim i izvršava i njeno rešenje smešta u promenjivu `String opis`.
Primetimo da ukoliko nema drugih klasa van `Pravouganik` i `TestPravouganik`, nema ni jednog objekta klase `Pravouganik`, ali mi bez problema pristupamo statičkoj metodi `opis()`, što nam je moguće upravo zato što se statički elementi kreiraju u momentu prvog moninjanja klase u kojoj se nalaze. Zato, prirodno se nalažu dve važne posledice:

Posledica 8.1.

Nestatički elementi mogu da pristupaju statičkim elementima.

Posledica 8.2.

Statički elementi ne mogu da pristupaju nestatičkim elementima.

Ako bolje razmislimo, ove dve posledice imaju itekako smisla. Posledica 8.1 važi jer nestatički elementi pripadaju objektima klase, a da bi smo kreirali objekat neke klase, moramo pozvati konstruktor te klase, što znači da **pominjemo** tu klasu, te čak pre pokretanja traženog konstrukta, ta klasa inicijalizuje sva statička polja i kreira sve statičke metode.

Posledica 8.2 se takodje prirodno nameće. Naime, statičkim elementima, poput statičkoj metodi, ne možemo garantovati da će se ikada kreirati bilo kakav objekat, tj. nestatički element te klase; kao što smo mogli da vidimo i u gornjem primeru. Zbog toga, statičke metode mogu da pozivaju samo druge statičke metode.

Konačno možemo da dešifrujemo i objasnimo prvu liniju koda koju smo pisali:

```
public static void main(String[] args)
```

- Kreiramo javnu, statičku funkciju bez povratne vrednosti naziva `main` koja uzima niz niski kao jedini argument.

Očigledno, kako je `main` metoda glavan metoda, nema nikakvog smisla pozivati je od objekata klase, te ona treba biti statička. Štaviše, ona mora biti statička jer odnekle moramo krenuti - da nije bila, morali bi smo prvo nekako da kreiramo objekat klase, pa da pozovemo `main` metodu te klase, pre nego što pokrenemo program - što nema smisla.

Koristeći se ovim znanje, konačno možemo rešiti prethodni zadatak uvodjenjem jednog statičkog polja klase, koji će brojati koliko smo objekata kreirali tako što će se kao poslednja linija koda u konstruktoru uvećavati za jedan. Iako je po prirodi takvo polje `static`, njega takodje možemo enskapulirati i dozvoliti samo njegovo dohvatanje.

```
package pravougaonik;

public class Pravougaonik {

    private double a, b;
    private static brojacObjekata = 0;

    public Pravougaonik(double stranica1, double stranica2)
{
        a = stranica1;
        b = stranica2;
        brojacObjekata++;
    }

    public Pravougaonik(double stranica) {
        a = stranica;
        b = stranica;
        brojacObjekata++;
    }

    public Pravougaonik() {}

    public double obim() {
        double o = 2 * a + 2 * b;
        return o;
    }
```

```
    public double povrsina() {
        return a * b;
    }


    public static int getBrojacObjekata() {
        return brojacObjekata;
    }

    public double getA() {
        return a;
    }

    public double getB() {
        return b;
    }

    public void setA(double a)  {
        this.a = a;
    }

    public void setB(double b) {
        this.b = b;
    }

    public static String opis() {
        return "Pravougaonik predstavlja geometrijsku
figuru sa dva para jednakih naspramnih medjusobno normalnih
stranica."
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
```

```
        sb.append("Pravougaonik stranica
").append(a).append(" i ").append(b).append("\n");
        sb.append("Obim: ").append(obim());
        sb.append("Porvsina: ").append(povrsina());

        return sb.toString();
    }
}
```

```
package pravougaonik;

public class TestPravougaonik {

    public void brojPravougaonika() {
        System.out.println("Kreirali smo " +
Pravougaonik.getBrojacObejkata() + " objekata klase
Pravougaonik");
    }

    public static void main(String[] args) {
        Pravougaonik p = new Pravougaonik(3, 2);
        Pravougaonik p2 = new Pravougaonik(5, 5);
        brojPravougaonika();
    }
}
```

```
OUTPUT: "Kreirali smo 2 objekata klase Pravougaonik"
```

Vidimo da sada nema potrebe za dodatnim brojačem i da ma koliko različith klasa kreiralo objekte klase `Pravougaonik`, brojač koji to memoriše je vezan bas za klasu `Pravougaonik` i jedinstven za ceo projekat!

Pored ovoga, u Javi je moguće deklarisati i **statički blok** unutar klase.

Definicija 8.13 (Statički blok)**.**

Statički blok predstavlja blok koda označen sa ključnom rečju `static`, koji se izvršava samo jednom i to u momentu kada se data klasa **prvi put pomene** u programu.

Kreirajmo jedan statički blok u našoj klasi `Pravougaonik` čija je jedina svrha da ispiše poruku "Pomenula se klasa Pravougaonik":

```
package pravougaonik;

public class Pravougaonik {

    private double a, b;
    private static brojacObjekata = 0;

    static {
        System.out.println("Pomenula se klasa
Pravougaonik");
    }

    public Pravougaonik(double stranica1, double stranica2)
{
        a = stranica1;
        b = stranica2;
        brojacObjekata++;
    }

    public Pravougaonik(double stranica) {
        a = stranica;
```

```
        b = stranica;
        brojacObjekata++;
    }

    public Pravougaonik() {}

    public double obim() {
        double o = 2 * a + 2 * b;
        return o;
    }

    public double povrsina() {
        return a * b;
    }


    public static int getBrojacObjekata() {
        return brojacObjekata;
    }

    public double getA() {
        return a;
    }

    public double getB() {
        return b;
    }

    public void setA(double a)  {
        this.a = a;
    }

    public void setB(double b) {
        this.b = b;
    }
```

```
    public static String opis() {
        return "Pravougaonik predstavlja geometrijsku
figuru sa dva para jednakih naspramnih medjusobno normalnih
stranica."
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();

        sb.append("Pravougaonik stranica
").append(a).append(" i ").append(b).append("\n");
        sb.append("Obim: ").append(obim());
        sb.append("Porvsina: ").append(povrsina());

        return sb.toString();
    }
}
```

Sada, kada kreiramo dva objekta klase `Pravougaonik`, dodata poruka će se ispisati samo *jednom* i to pre kreacije prvog objekta:

```
package pravougaonik;

public class TestPravougaonik {

    public static void main(String[] args) {
        Pravougaonik p = new Pravougaonik(3, 2);
        Pravougaonik p2 = new Pravougaonik(5, 5);
    }
}
```

```
OUTPUT: "Pomenula se klasa Pravougaonik"
```

Statička polja i metode su jako korisne kod klasa poput `Math`, čije ponašanje ne varira od objekta do objekta. Tako na primer, `Math.PI` je jedno `public static final double` polje, dok je `Math.pow()` jedna `public static double` metoda. Štatviše, ako malo razmislimo, nema nikavog smisla kreirati objekte klase `Math` jer bi njihovo ponašanje bilo identično za svaki objekat - matematika je invarijantna na perspektivu. Zato, jedini (prazni) konstruktor u klasi `Math` je definisan kao `private`, s čime i zabranjujemo kreiranje objekata te klase.

```
Math m = new Math();
```

Klase kao polja klase

TODO

Liste klasa i polja liste klasa

TODO

---

*Ugnježdene klase

---

1. U poslednje vreme funkcionalno programiranje je u porastu i vodeća lica u sferi objektno orijentisanog programiranja pokušavaju da pronadju jednostavniji način programiranja jer OOP sa sobom vuče takozvani 'boiler plate' kod, odnosno kod koji nema nikakvu funkcijsku svrhu, već je

tu zbog sintaksnih ili sematičkih razloga. No, zamena OOP koncepata je idalje dalek, ako uopšte moguć koncept, pošto jev već toliko koda napisano u OOP stilu, koji se pokazao svakako efektivnim. ↩

2. Osim u slučaju statičkih atributa/metoda. Videti odeljak 'Ključna reč static' ↩ ↩
3. Alozno Church (1903 - 1995) - Tjuringov mentor koji je paralelno sa Tjuringom kreirao drugačiji formalizam programiranja koji se zasniva na anonimnim funkcijama koji nazivamo lambda računom ($\lambda-$račun) ↩
4. Heap - gomila, eng. ↩
5. null pokazivač je prvi kreirao Tony Hoare (1934 -), jedan od velikana u svetu računarstva za svoj programski jezik ALGOL. Ovaj pokazivač se ponaša nalik praznom skupu u matematici (skupu bez ijednog elementa, tj. $\{\}$). Nakon više decenija, kada je Hoare primetio kakve sve probleme null pokazivač povlači sa sobom, priznao je da se kaje što ga je ikada kreirao. ↩
6. Pattern matching - engleski izraz ↩
7. Kada želimo da naznačimo da je nešto `default` vidljivosti, ne smemo da koristimo `default` ključnu reč jer je ona rezervisana za `switch-case` kontrolu toka. ↩