



Funded by the
Erasmus+ Programme
of the European Union



This project has been funded with support from the European Commission. This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.



KI104 - JAVA 2: OBJEKTNO-ORIJENTISANO PROGRAMIRANJE

Objekti i klase

Lekcija 01

PRIRUČNIK ZA STUDENTE

KI104 - JAVA 2: OBJEKTNO-ORIJENTISANO PROGRAMIRANJE

Lekcija 01

OBJEKTI I KLASSE

- ✓ Objekti i klase
- ✓ Poglavlje 1: Klase i objekti
- ✓ Poglavlje 2: Definicija klase
- ✓ Poglavlje 3: Primena klasa iz Javine biblioteke klasa
- ✓ Poglavlje 4: Uvod u klasni model Power Designer
- ✓ Poglavlje 5: Domaći zadatak
- ✓ Poglavlje 6: Rezime kroz dodatne primere
- ✓ Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

UVOD

Koncept klase i objekata je osnov objektno-orijentisanog programiranja.

U ovom predavanju se zato obraća posebna pažnja na detalje koncepta klase i njenog definisanja u Javi, kao i na potpuno razumevanje objekata i njihovog konstruisanja u programu.

Ciljevi ove lekcije su:

- Opisivanje objekata i klasa i upotreba klasa radi modelovanja objekata
- Upotreba UML grafičkog označavanja klasa i objekata i njihovo modelovanje.
- Objašnjenje kako se definišu klase i kako se kreiraju objekti
- Kreiranje objekata upotrebom konstruktora
- Pristup objektima preko referentne promenljive objekta
- Definisanje referentne promenljive objekta upotrebom tipa reference
- Pristup podacima i metodima objektu upotrebom operatora za pristup metodama (.)
- Definisanje polja podataka referentnog tipa i dodeljivanje početnih vrednosti podataka
- Razlika između referentnih promenljivih objekata i primitivnih promenljivih objekata
- Razlika između objektnih i statičkih promenljivih i metoda
- Učaurjenje polja podataka radi lakše izrade klasa sa lakšim održavanjem.
- Razvoj metoda sa argumentima sa objektima i razlika između argumenata primitivnog tipa i argumenata sa objektima.

▼ Poglavlje 1

Klase i objekti

ŠTA JE OBJEKTNO-ORIJENTISANO PROGRAMIRANJE?

Objektno -orijentisano programiranje (OOP) je programiranje sa upotrebom objekata. Objektno-orijentisano programiranje omogućava vam efektivni razvoj velikih softverskih sistema i GUI.

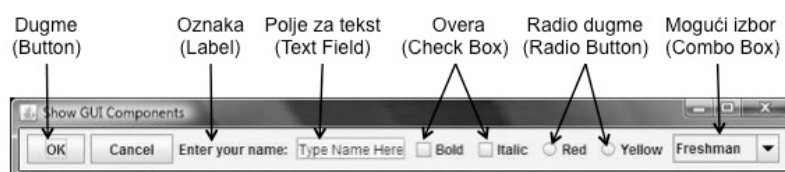
Objektno-orijentisano programiranje (OOP) predstavlja noviji pristup za rešavanje problema na računaru kojim se rešenje traži na prirodan način koji bi se primenio i u svakodnevnom životu. U starijem, proceduralnom programiranju je programer morao da identifikuje računarski postupak za rešenje problema i da napiše niz programskih naredbi kojim se realizuje taj postupak.

Naglasak u objektno-orijentisanom programiranju nije na računarskim postupcima nego na objektima - softverskim elementima sa podacima i metodima koji mogu da dejstvuju jedni na druge. Objektno-orijentisano programiranje se zato svodi na kreiranje različitih klasa objekata kojima se na prirodan način modelira problem koji se rešava. Pritom, softverski objekti u programu mogu predstavljati ne samo realne, nego i apstraktne entitete iz odgovarajućeg problemskog domena.

Objektno-orijentisane tehnike, u principu, mogu se koristiti u svakom programskom jeziku. To je posledica činjenice što objekte iz standardnog ugla gledanja na stvari možemo smatrati običnom kolekcijom promenljivih i procedura koje manipulišu tim promenljivim. Međutim, za efektivnu realizaciju objektno-orijentisanog načina rešavanja problema je vrlo važno imati jezik koji to omogućava na neposredan i elegantan način.

Programiranje u Javi bez korišćenja njenih objektno-orijentisanih mogućnosti nije svrsishodno, jer su za proceduralno rešavanje problema lakši i pogodniji drugi jezici..

Java se koristi za programiranje selekcija, petlji, metoda i nizova. Međutim, ovo nije dovoljno za razvoj grafičkog korisničkog interfejsa (Graphic User Interface - GUI) , kao što su GUI objekti prikazani na slici 1 .



Slika 1.1 GUI objekti kreirani iz GUI klasa

Primena objektno-orijentisanog programiranja olakšava razvoj velikih softverskih sistema, jer primenom manjih komponenata sistema, olakšava njegovo razumevanje, fazni i paralelni razvoj i jednostavne održavanje.

ŠTA JE OBJEKAT?

Objekat ima svoj jedinstveni identitet i svoja svojstva, kao što su stanje i ponašanje.

Na slici 1 prikazan je jedan auto crvene boje.



Slika 1.2 Auto, kao primer objekta

Vidimo da ima dvojna vrata, i naravno – ima i četiri točka. Ostala svojstva automobila se ne vide. Koliko troši goriva? Da li trenutno njegov motor radi? Ako neko pritisne papučicu za gas, auto počinje da se kreće. Ova reakcija je deo ponašanja automobila. Kada jedan objekat (čovjek) radi sa automobilom na određeni način, auto reaguje na taj način što počinje da se kreće. Trenutna brzina kretanja automobila je takođe, deo njegovog stanja, a ponašanje automobila se odražava na njegovo stanje. Vozač upravlja autom (objektom) upotrebom različitih uređaja za upravljanje (volan, menjač brzine, pedale, prekidači...) i time utiče na njegovo ponašanje.

Primenom različitih merača-instrumenata, vozač saznaje trenutno stanje automobila (objekta). Interfejs automobila čine uređaji za upravljanje radom automobila i merači, tj. instrumenti. To je veza vozača sa automobilom. Pošto dva automobila istog tipa, iste boje i sa istim dodacima izgledaju identično, oni moraju da imaju svoj sopstveni identifikacioni broj, tj. broj registracije. Znači, u realnosti, imamo dva ista automobila, te se oni moraju nekako razlikovati (pored registracije, proizvođač im dodeljuje jedinstven broj na karoseriji i na motoru).

Objekat ima svoj jedinstveni identitet i svoja svojstva, kao što su stanje i ponašanje.

- **Stanje objekta** (svojstva, atributi) predstavljaju polja podataka sa svojim trenutnim vrednostima.
- **Ponašanje objekta** (akcije, operacije) definišu se metodima. Da bi se pozvao neki metod objekta, mora se tražiti od objekta da izvrši određenu akciju (operaciju).

- **Identitet** nekog objekta se odnosi na neko njegovo svojstvo po kojem se on razlikuje od bilo kojeg drugog objekta. Svaki objekat je jedinstven, jer bilo koja dva ili više objekta, bez obzira na njihove sličnosti, imaju individualan identitet.

ŠTA JE KLASA?

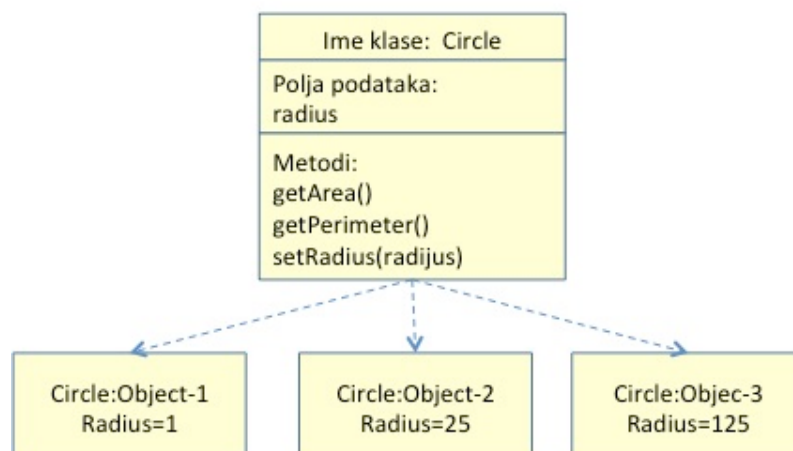
Klasa opisuje objekte sa sličnim karakteristikama i ponašanjem; ona je opšti opis nekog objekta

Objekti istog tipa se definišu zajedničkom klasom. **Klasa** je uzorak ili ugovor koji definiše polja podataka i metode koje će posedovati svi objekti klase. Klasa opisuje objekte sa sličnim karakteristikama i ponašanjem; ona je opšti opis nekog objekta.

Klasa definiše sadržaj objekta (karakteristike i ponašanja), ali ga ne popunjava, tj. ne daje konkretne vrednosti atributima. Klasa je softverski opis objekta. Ona definiše sadržaj i strukturu objekta koji se, na osnovu toga, može da kreira. Popunjavanjem sadržaja i strukture dobija se pojedinačni objekat. Dok klasa definiše od čega se sastoji neki objekat, objekat je stvarni (realni) primerak (instanca) klase.

Klasa u stvarnosti ne postoji. Samo objekti postoje u stvarnosti. Ako klasu zamislimo kao neki šablon za kreiranje svojih objekata, onda su objekti entiteti stvoreni na osnovu tog šablona. Na primer, recept za pravljenje pite od jabuka je analogno klasi, a sama pita od jabuka, koju možemo da jedemo, je objekat. Mi vidimo objekte i radimo sa njima, a klase, kao šabloni, iskorišćeni su samo za stvaranje (kreiranje) svojih objekata, ali u stvarnosti oni ne postoje, jer predstavljaju definišu strukture i svojstva objekata koje stvara.

Objekat je određen primerak ili instanca (engl. instance) određene klase. Možete kreirati puno objekata, tj. instanci (primerka) neke klase. Kreiranje neke instance je opredmećenje (engl. instantiation) klase. Termini objekat ili instanca se koriste za istu stvar. Na slici 2 prikazana je klasa za kreiranje krugova, pod nazivom **Circle** i tri objekta, tj. krugova sa različitim prečnicima.



Slika2 Klasa Circle i njeni objekti - krugovi

KREIRANJE KLASA

Video objašnjava šta je klasa i kako se kreira

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

UVOD U OBJEKTNO-ORIJENTISANO PROGRAMIRANJE (VIDEO)

Video daje pregled koncepata primenjenih u objektno-orijentisanom programiranju

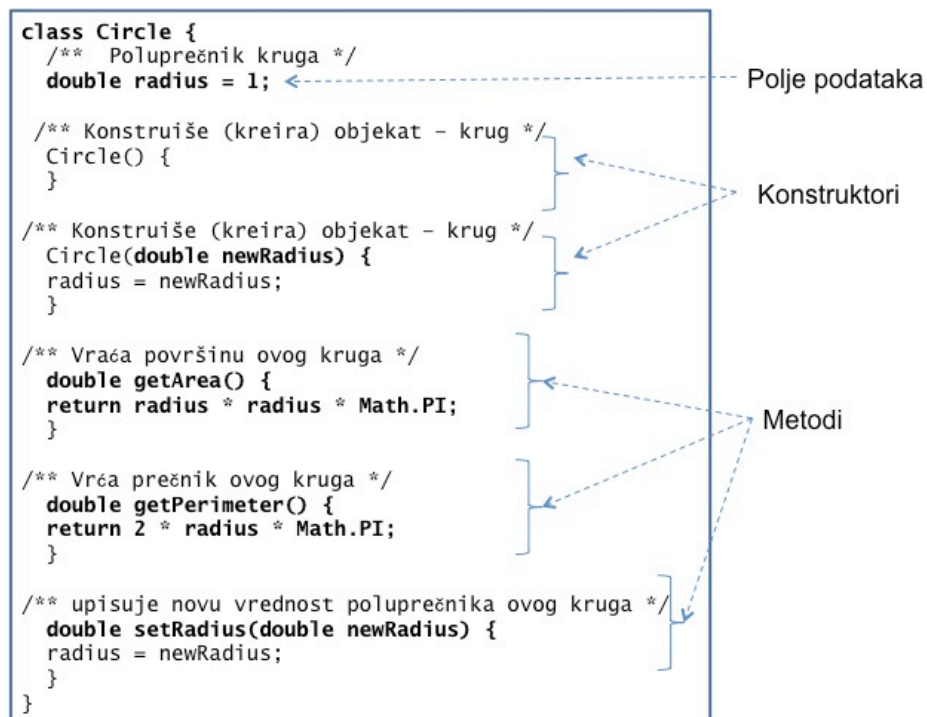
Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

PRIMER 1 - KLASA CIRCLE DEFINISANA U JAVI

Java klasa upotrebljava promenljive da bi definisala polja podataka i metode potrebne za njene akcije/operacije

Java klasa upotrebljava promenljive da bi definisala polja podataka i metode potrebne za njene akcije/operacije. Svaka klasa poseduje jedan ili više metoda posebnog tipa po nazivaju konstruktori (engl., constructors) koji pokreću kreiranje novog objekta svoje klase. Konstruktori inicijaliziraju početne vrednosti u poljima podataka objekata koje kreiraju. Na slici 3 dat je primer klase **Circle**, tj. klase za definisanje krugova i njihovo kreiranje uz pomoć njenih konstruktora.

Klasa Circle, kao što se vidi, ne poseduje metod **main()** te ne može da se izvrši samostalno. Ona samo definiše kružne objekte. Da bi se neki objektno-orijentisan program pisan u Javi mogao da izvrši, mora da ima metod **main()**, jer samo njegovim pozivanjem, tj. aktiviranjem, počinje izvršenje Java programa. Kako metodi ne postoje van klase, to klasa koja sadrži metod **main()** ima naziv **main klasa**.

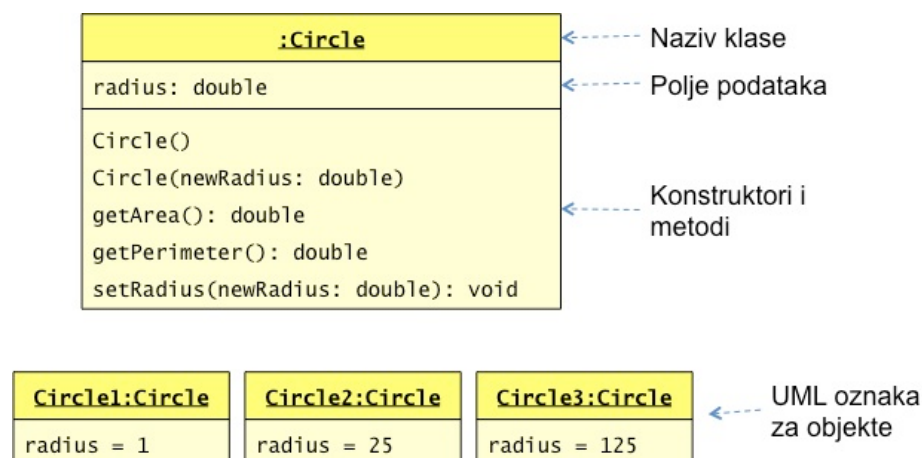


Slika 1.3 Izvorni kod klase Circle sa konstruktorima

PRIMER 1 - KLASA CIRCLE U UML-U

Klasa u UML-u se predstavlja pravougaonikom koji ima tri dela koji sadrže: naziv klase, nazive atributa i njihove tipove podataka, i metode sa svojim parametrima i rezultatom.

Na slici 4 prikazana je definicija klase i objekata na standardizovani način upotrebom univerzalnog jezika za modelovanje - **Universal Modeling Language (UML)**. Slika prikazuje UML dijagram klase, ili kraće, dijagram klase.



Slika 1.4 Klasa Circle i njeni objekti predstavljeni u UML

Klasa u UML-u se predstavlja pravougaonikom koji ima tri dela_Prvi deo sadrži ime (naziv) klase.Drugi deo sadrži atribut klase.Treći deo sadrži konstruktore i ostale metode klase. Radi pojednostavljenja, najčešće se ne navode get i set metode koji čitaju i menjaju vrednosti atributa, jer se podrazumevaju. :

Klasa ima svoj **naziv (ime)** i označen je sa:

:ImeKlase

tj. Iza dve tačke, sledi naziv klase, a sve to podvučeno

Atributi se označavaju sa:

imetPromenljive : tipPromenljive

Konstruktor se označava sa:

ImeKlase(nazivParametra : tipParametra)

Metod: se označava:

imeMetoda(imeParametra : tipParametra) : tipRezultata

Objekti se predstavljaju u UML-u slično kao i klase, samo što se ne prikazuje treći deo sa metodima.

Ime objekta se upisuje ispred imena klase kome pripada, tj. ispre dve tačke:

ImeObjekta :ImeKlase

Atributi kod objekata se upisuju sa konkretnim vrednostima:

imePromenljive = vrednostPromenljive

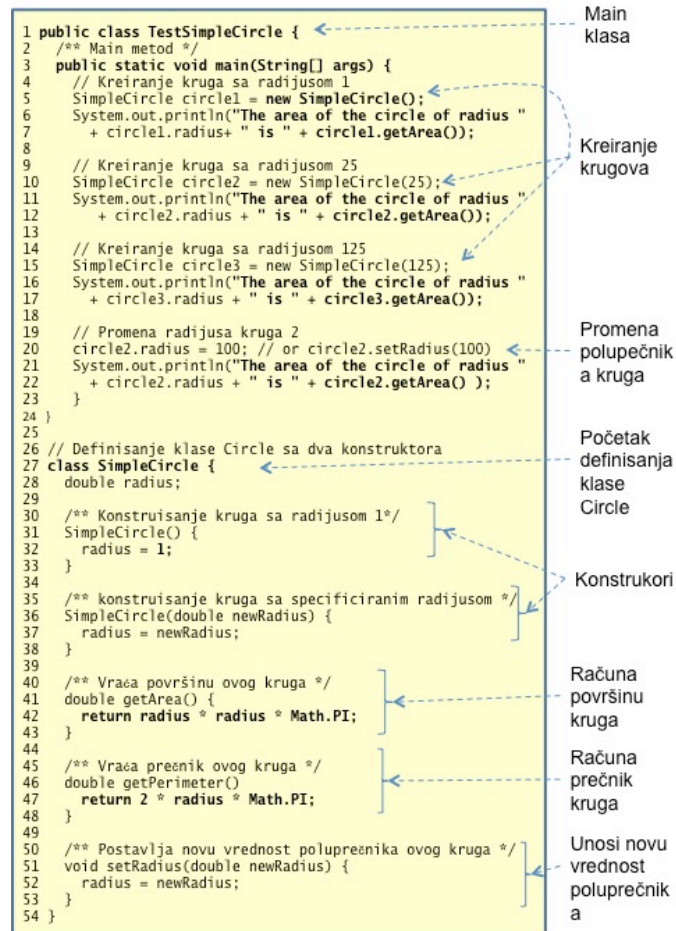
PRIMER 2 - DEFINISANJE MAIN KLASE ZA CRTANJE KRUGA

Samo main klasa je javna, tj. koristi modifikator public.

Ovde se navode dva primera definisanja klase i upotrebe klase za kreiranje njihovih objekata. Na slici 5 dat je prikaz programa, sa objašnjenjima, a koji definiše klasu **Circle** i koji je koristi za kreiranje njenih objekata (krugova). Program kreira tri kružna objekta sa poluprečnicima o 1, 25 i 125 i prikazuje poluprenik i površinu svakog od truh kružna objekta. Program zatim menja poluprečnik drugog kružnog objekta (umesto 25 sada je 100) i prikazuje njegov novi poluprečnik i njegovu novu (izračunatu) površinu.

Program sadrži dve klase. Prva klasa **TestSimpleCircle** je main klasa. Njena jedina svrha je da testira drugu klasu **SimpleCircle**. Program koji sadrži definicije klase se ponekad naziva klijentom klase. Kada izvršavate ovaj program, Java izvršni sistem poziva **main()** metod koji je dat u main klasi.

Možete obe klase da smestite u jedan fajl, ali samo jedna od klasa može biti javna klasa (public class). **Javna klasa** mora da ima isto ime kao što je i ime fajla. Prema tome, kako je klasa **TestSimpleCircle** javna, to se fajl mora da naziva: **TestSimpleCircle.java**.



Slika 1.5 Listing klase `TestSimpleCircle` sa objašnjenjima

PRIMER 2 - LISTINZI KLASA

Fajl sa izvornim kodom dobija naziv main klase i ima završetak .java sa svakom klasom u njoj, posle kompilacije dobija poseban fajl sa nazivom klase i završetkom .class.

Donjli listing sadrži kod obe klase: `TestSimpleCircle` i `SimpleCircle`.

```
public class TestSimpleCircle {
    /** Main metod */
    public static void main(String[] args) {
        // Kreiranje kruga sa radijusom 1
        SimpleCircle circle1 = new SimpleCircle();
        System.out.println("The area of the circle of radius "
            + circle1.radius+ " is " + circle1.getArea());

        // Kreiranje kruga sa radijusom 25
```

```

        SimpleCircle circle2 = new SimpleCircle(25);
        System.out.println("The area of the circle of radius "
            + circle2.radius + " is " + circle2.getArea());

        // Kreiranje kruga sa radijusom 125
        SimpleCircle circle3 = new SimpleCircle(125);
        System.out.println("The area of the circle of radius "
            + circle3.radius + " is " + circle3.getArea());

        // Promena radijusa kruga 2
        circle2.radius = 100; // or circle2.setRadius(100)
        System.out.println("The area of the circle of radius "
            + circle2.radius + " is " + circle2.getArea() );
    }
}

// Definisanje klase Circle sa dva konstruktora
class SimpleCircle {
    double radius;

    /** Konstruisanje kruga sa radijusom 1 */
    SimpleCircle() {
        radius = 1;
    }

    /** konstruisanje kruga sa specificiranim radijusom */
    SimpleCircle(double newRadius) {
        radius = newRadius;
    }

    /** Vraća površinu ovog kruga */
    double getArea() {
        return radius * radius * Math.PI;
    }

    /** Vraća prečnik ovog kruga */
    double getPerimeter()
        return 2 * radius * Math.PI;
    }

    /** Postavlja novu vrednost poluprečnika ovog kruga */
    void setRadius(double newRadius) {
        radius = newRadius;
    }
}

```

Glavna main klasa sadrži **main()** metod (linija 3 u listingu klase TestSimpleCircle) koji kreira tri objekta. Kreirajući jedan niz, operator new se koristi za kreiranje objekta pomoću konstruktora **new SimpleCircle()**. On kreira objekat (prvi krug) radijus 1 (linija 5). Konstruktor **new SimpleClass(25)** kreira drugi krug sa radijusom 25, a treći konstruktor **new SimpleCircle(126)** kreira objekat sa radijusom 125 (linija 15).

Ova tri objekta (označeni sa **circle1**, **circle2** i **circle3**) imaju različite podatke, ali koriste iste metode. Tako, koriste metod **getArea()** za proračun svojih površina. Poljima podataka se može pristupiti sa referencom (sa pozivanjem) objekata i upotrebom **circle1.radius**, **circle2.radius**, i **circle3.radius**. **Objekat može da pozove svoj metod preko reference objekata upotrebom **circle1.getArea()**, **circle2.getArea()**, i **circle3.getArea()**.**

Ova tri objekta su nezavisna. Radijus objekta **circle2** je promenjen na 100 u liniji 20. Sa ovim novim radijusom je izračinata njegova nova površina u linijama 21-22.

PRIMER 2: UMETO DVE, KORISTI SE SAMO JEDNA KLASA

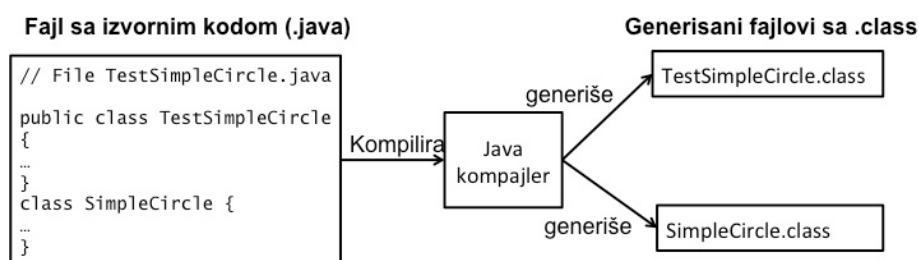
Postoji puno načina da se napiše Java program. Na primer, kombinovanjem dve klase u jednu, dobija se novi Java program.

Na monitoru se dobija sledeći prikaz (kao rezultat System.out.println() metoda):

```
The area of the circle of radius 1.0 is 3.141592653589793
The area of the circle of radius 25.0 is 1963.4954084936207
The area of the circle of radius 125.0 is 49087.385212340516
The area of the circle of radius 100.0 is 31415.926535897932
```

Slika 1.6 Prikaz rezultata na monitoru

Svaka klasa sa izvornim Java kodom ima produžetak u nazivu .java. Fajl svake Java klase posle kompilacije dobija produžetak naziva .class . Zbog toga, kompilacijom klase u fajlu TestSimpleCircle.java dobijaju se dva fajla sa produžecima (extensions) naziva: TestSimpleCircle.class i SimpleCircle.class, kao što je prikazano na slici 7 .



Slika 1.7 Posle kompilacije, svaka klasa se nalazi u posebnom fajlu, sa završetkom .class u bajkodu

Postoji puno načina da se napiše Java program. Na primer, kombinovanjem dve klase sa slike 5 u jednu, dobija se novi Java program čiji je listing dat dole.

Kako kombinovana klasa ima main metod, može se izvršiti sa interpreterom Jave. Ovde je main metod isti kao i u prethodnom slučaju, na slici 5 . Ovo pokazuje da možete da testirate neku klasu jednostavno, dodajući joj metod main. Sve u jednoj klasi.

```
public class SimpleCircle {
    /* Main metoda */
    public static void main(String[] args) {
```

```
// Kreira krug sa poluprečnikom 1
SimpleCircle circle1 = new SimpleCircle();
System.out.println("The area of the circle of radius "
+ circle1.radius + " is " + circle1.getArea());

// Kreira krug sa radijusom 25
SimpleCircle circle2 = new SimpleCircle(25);
System.out.println("The area of the circle of radius "
+ circle2.radius + " is " + circle2.getArea());

// Kreira krug sa radijusom 125
SimpleCircle circle3 = new SimpleCircle(125);
System.out.println("The area of the circle of radius "
+ circle3.radius + " is " + circle3.getArea());

// Menjanje poluprečnika kruga
circle2.radius = 100;
System.out.println("The area of the circle of radius "
+ circle2.radius + " is " + circle2.getArea());
}

double radius;

/** Kreira krug sa poluprečnikom 1 */
SimpleCircle() {
    radius = 1;
}

/** Kreiranje kruga sa specificiranim poluprečnikom */
SimpleCircle(double newRadius) {
    radius = newRadius;
}

/** Vraća površinu ovog kruga */
double getArea() {
    return radius * radius * Math.PI;
}

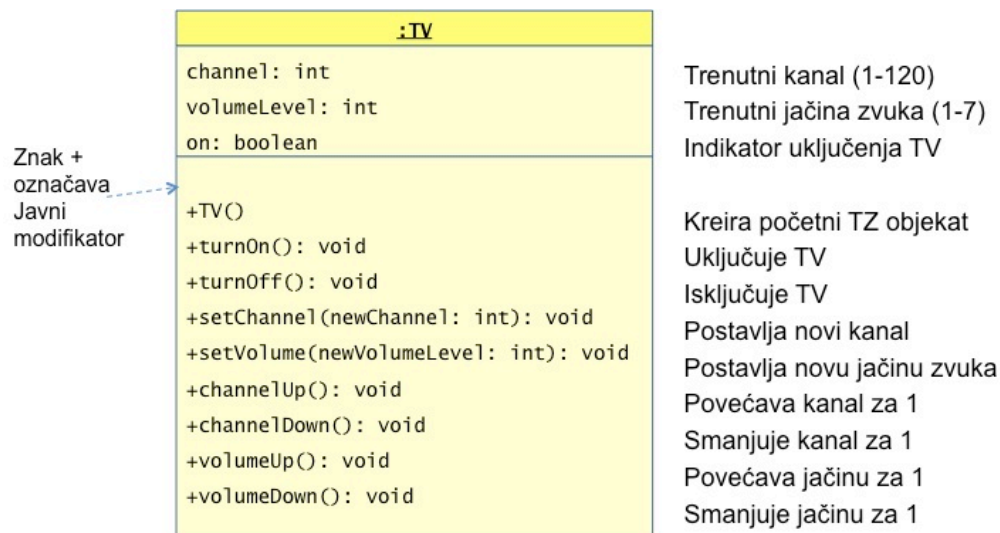
/** Vraća prečnik ovog kruga */
double getPerimeter() {
    return 2 * radius * Math.PI;
}

/** Postavlja novu vrednost poluprečnika ovog kruga */
void setRadius(double newRadius) {
    radius = newRadius;
}
}
```

PRIMER 3: KLASA TV U UML

Klasa TV ima tri atributa i više metoda

Svaki televizor je jedan objekat sa stanjima (sadašnji program, sadašnji nivo zvuka, uključen ili isključen) i sa ponašanjima (promena kanala, podešavanje jačine zvuka, uključenje/ isključenje). Na slici 8 prikazan je UML dijagram klase koja predstavlja model televizora.



Slika 1.8 UML model klase TV

PRIMER 3: KLASA TV U JAVI

Konstruktor je metod koji može biti preopterećen, tj. da ima više verzija, ali sa različitim argumentima.

Konstruktor i metodi u klasi TV su definisani kao javni, te im se može prići sa drugih klasa. Kao što vidite, pri isključenju i uključenju ne menja sa nivo zvuka i kanal. Pre nego što se ovi promene, njihove tekuće vrednosti se proveravaju da li su unutar datog oseg vrednosti.

```
public class TV {
    int channel = 1; // Default channel is 1
    int volumeLevel = 1; // Default volume level is 1
    boolean on = false; // TV is off

    public TV() { }

    public void turnOn() {
        on = true;
    }

    public void turnOff() {
        on = false;
    }
}
```



```
public void setChannel(int newChannel) {
    if (on &&&&&&&&&&&&&&&& newChannel >= 1
&&&&&&&&&&&&&&&&&& newChannel <= 120)
        channel = newChannel;
}

public void setVolume(int newVolumeLevel) {
    if (on &&&&&&&&&&&&&&&& newVolumeLevel >=
1 &&&&&&&&&&&&&&&&&& newVolumeLevel <= 7)
        volumeLevel = newVolumeLevel;
}

public void channelUp() {
    if (on &&&&&&&&&&&&&&&& channel < 120)
        channel++;
}

public void channelDown() {
    if (on &&&&&&&&&&&&&&&& channel > 1)
        channel--;
}

public void volumeUp() {
    if (on &&&&&&&&&&&&&&&& volumeLevel < 7)
        volumeLevel++;
}

public void volumeDown() {
    if (on &&&&&&&&&&&&&&&& volumeLevel > 1)
        volumeLevel--;
}
}
```

PRIMER - KLASA TESTTV

Metodima se pristupa sa izrazom `imeObjekta.imeAtributa`, a metodu `imeObjekta.imeMetoda`

Listing prikazuje program koji upotrebljava klasu TV, da bi kreirao dva njena objekta.

Program kreira dva objekta u linijama 3 i 8 i poziva metode objekata radi izvršenja akcija postavljanja kanala i nivoa jačine zvuka, kao i za povećanje kanala i jačine zvuka. Program prikazuje stanje objekata na linijama 14-17. Metodi se pozivaju upotrebom sintakse kao što je `tv1.turnOn()` (linija 4). Poljima podataka se pristupa upotrebom sintakse kao što je `tv1.channel` (linija 14).

```
public class TestTV {
    public static void main(String[] args) {
        TV tv1 = new TV();
        tv1.turnOn();
    }
}
```



```
        tv1.setChannel(30);
        tv1.setVolume(3);

        TV tv2 = new TV();
        tv2.turnOn();
        tv2.channelUp();
        tv2.channelUp();
        tv2.volumeUp();

        System.out.println("tv1's channel is " + tv1.channel
            + " and volume level is " + tv1.volumeLevel);

        System.out.println("tv2's channel is " + tv2.channel
            + " and volume level is " + tv2.volumeLevel);
    }
}
```

ZADACI ZA SAMOSTALNI RAD

Na osnovu predhodnih primera uraditi sledeće zadatke

Zadatak 1.1:

Kreirati klasu CPU koja ima sledeće atribute:

1. *nazivCPU*,
2. *brzinaUMhz*,
3. *threads*,
4. *cores* i
5. *proizvodjac*.

Zatim kreirati i klasu Proizvođač sa atributima

1. *imeProizvodjaca* i
2. *oblast*.

▼ Poglavlje 2

Definicija klase

UVOD

Klase opisuju određene objekte čijim se manipulisanjem u programu ostvaruje potrebna funkcionalnost.

U ovom delu u ćemo se upoznati sa:

- Definicijom klase
- Dužinom trajanja i oblašću važenja promenljivih
- Modifikatorima pristupa
- Promenljivima klasnog tipa
- Konstrukcijom, inicijalizacijom i uklanjanjem objekata
- Enkapsulacijom (učaurivanjem) i sakrivanjem podataka
- Ključnom reči this

▼ 2.1 Statički i objektni članovi klasa

KLASA

Klasa obuhvata attribute ili polja (globalne promenljive) i metode koji se jednim imenom nazivaju članovi klase. Klasa je šablon za konstruisanje objekata.

Metodi su programske celine za obavljanje pojedinačnih specifičnih zadataka u programu - na sledećem višem nivou složenosti programskih jedinica su **klase**. Naime, jedna klasa može obuhvatati kako promenljive (attribute ili polja) za čuvanje podataka, tako i više metoda za manipulisanje tim podacima radi obavljanja različitih zadataka.

Pored ove organizacione uloge klasa u programu, druga njihova uloga je da opišu određene objekte čijim se manipulisanjem u programu ostvaruje potrebna funkcionalnost, pa ćemo u ovom odeljku detaljnije govoriti o ovim ulogama klase.

Klasa obuhvata attribute ili polja (globalne promenljive) i metode koji se jednim imenom nazivaju članovi klase. Ovi **članovi klase** mogu biti statički, što se prepoznaje po modifikatoru **static** u definiciji člana, ili **nestatički (objektni)**.

S druge strane, govorili smo da klasa opisuje objekte sa zajedničkim osobinama i da svaki objekat koji pripada nekoj klasi ima istu strukturu koja se sastoji od atributa i metoda definisanih unutar njegove klase. Tačnije je reći da samo **nestatički deo** neke klase opisuje objekte koji pripadaju toj klasi, ali najtačnije iz programerske perspektive je da klasa služi za konstruisanje objekata.

Drugim rečima, neka **klasa je šablon za konstruisanje objekata**, pri čemu svaki konstruisan objekat te klase sadrži sve nestatičke članove klase.

Za objekat konstruisan na osnovu definicije neke klase se kaže da pripada toj klasi ili da je primerak ili instanca ili objekat te klase.

Nestatički članovi klase (atributi i metodi), koji ulaze u sastav konstruisanih objekata, nazivaju se i objektni ili instancni članovi.

Glavna razlika između klase i objekata je dakle to što se klasa definiše u tekstu programa, dok se objekti te klase konstruišu posebnim operatorom tokom izvršavanja programa. To znači da se klasa stvara prilikom prevođenja programa i zajedno sa svojim statičkim članovima postoji od početka do kraja izvršavanja programa.

S druge strane, objekti se stvaraju dinamički tokom izvršavanja programa i zajedno sa nestatičkim članovima klase postoje od trenutka njihovog konstruisanja, moguće negde u sredini programa, do trenutka kada više nisu potrebni.

ATRIBUTI

Statička promenljiva ne menja vrednost, tj. ne menja se. Nestatička (objektna) promenljiva ima različitu vrednost kod svakog objekta iste klase.

Da bismo bolje razumeli ove osnovne koncepte klase i objekata, posmatrajmo jednostavnu klasu koja može poslužiti za čuvanje osnovnih informacija o fakultetima

```
class Fakultet{
    static String ime;
}
```

Klasa **Fakultet** sadrži statički atribut **Fakultet.ime**, pa u programu koji koristi ovu klasu postoji samo jedan primerak promenljive **Fakultet.ime**.

Taj program može da modelira situaciju u kojoj postoji samo jedan fakultet u svakom trenutku, jer imamo memorijski prostor za čuvanje podataka o samo jednom fakultetu. Klasa **Fakultet** i njen statički atribut **Fakultet.ime** postoje sve vreme izvršavanja programa.

Posmatrajmo sada sličnu klasu koja sadrži skoro isti, ali nestatički atribut:

```
class Student{
    String ime;
}
```

U klasi `Student` je definisani **nestatički (objektni)** atribut `ime`, pa u ovom slučaju ne postoji promenljiva **`Student.ime`**. Ova klasa dakle ne sadrži ništa konkretno, osim šablona za konstruisanje objekata te klase. Ali, ova klasa može poslužiti za stvaranje velikog broja objekata, pri čemu će svaki konstruisani objekat ove klase imati svoj primerak atributa `ime`. Zato program koji koristi ovu klasu može modelirati više studenata, jer se po potrebi mogu konstruisati novi objekti za predstavljanje novih studenata.

Recimo, ako posmatramo softver za evidenciju aktivnih studenata, kad se student upiše na fakultet može se konstruisati nov objekat klase **`Student`** za čuvanje podataka o tom studentu. Ako se student ispiše sa fakulteta ili završi fakultet, objekat koji ga predstavlja u programu može se ukloniti.

Primetimo da ovi primeri ne znače da klasa može imati samo statičke ili samo nestatičke članove.

METODI

Statički metodi pripadaju klasi i postoje za sve vreme izvršavanja programa. Objektni metodi mogu se primenjivati samo za neki objekat koji je prethodno konstruisan.

U nekim primenama postoji potreba za obe vrste članova klase. Na primer, ako dopunimo definiciju klase **`Student`**:

```
class Student{
    static int ukupanBrojStudenata;
    String ime;
    int brojPoloženihIspita;

    void polaganjeIspita() {
        brojPoloženihIspita++;
        System.out.println("Položili ste: "
            + brojPoloženihIspita + ". ispit !")
    }
}
```

Ovom klasom se u programu mogu predstaviti studenti, pošto je ukupan broj studenata nepromenljiv za sve studente, nema potrebe taj podatak vezivati za svakog studenta i zato se može čuvati u jedinstvenom statičkom atributu **`Student.ukupanBrojStudenata`**.

S druge strane, `ime` i broj položenih ispita su karakteristični za svakog studenta, pa se ti podaci moraju čuvati u nestatičkim atributima `ime` i **`brojPoloženihIspita`**.

Obratite pažnju na to da se u definiciji klase određuju tipovi svih atributa, i statičkih i nestatičkih. Međutim, dok se aktuelne vrednosti statičkih atributa nalaze u samoj klasi, aktuelne vrednosti nestatičkih (objektnih) atributa se nalaze unutar pojedinih objekata, a ne klase. Slična pravila važe i za metode koji su definisani u klasi - statički metodi pripadaju klasi i postoje za sve vreme izvršavanja programa. Zato se statički metodi mogu pozivati u programu nezavisno od konstruisanih objekata njihove klase, pa čak i ako nije konstruisan nijedan objekat.

Definicije nestatičkih (objektnih) metoda se nalaze unutar teksta klase, ali takvi metodi logički pripadaju konstruisanim objektima, a ne klasi. To znači da se objektni metodi mogu primenjivati samo za neki objekat koji je prethodno konstruisan.

Na primer, u klasi Student je definisan objektni metod **polaganjelspita()** kojim se broj položenih ispita studenta uvećava za 1 i prikazuje koji ispit po redu je položen.

Metodi **polaganjelspita()** koji pripadaju različitim objektima klase Student obavljaju isti zadatak u smislu da prikazuju koji ispit po redu je položen različitim studentima. Ali njihov stvarni efekat je različit, jer broj položenih ispita za različite studente mogu biti različiti.

STATIČKI I OBJEKTNI ATRIBUTI I METODI

Statičkim delom definicije se navode članovi koji su deo same klase, dok se nestatičkim delom navode članovi koji će biti deo svakog konstruisanog objekta

Generalno govoreći, definicija objektnih metoda u klasi određuje zadatak koji takvi metodi obavljaju nad konstruisanim objektima, ali njihov specifični efekat koji proizvode može varirati od objekta do objekta, zavisno od aktuelnih vrednosti objektnih atributa različitih objekata.

Prema tome, statički i nestatički (objektni) članovi klase su vrlo različiti koncepti i služe za različite svrhe. Važno je praviti razliku između teksta definicije klase i samog pojma klase. Definicija klase određuje kako klasu, tako i objekte koji se konstruišu na osnovu te definicije. Statičkim delom definicije se navode članovi koji su deo same klase, dok se nestatičkim delom navode članovi koji će biti deo svakog konstruisanog objekta.

DEFINISANJE KLASA I KREIRANJE OBJEKATA (VIDEO)

Video objašnjava šta je klasa i kako se ona koristi za kreiranje objekata

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

PRIMER - KREIRANJE KLASA KVADRAT

Klasa Kvadrat opisuje jedan kvadrat i pokretačku klasu KvadratMain koja demonstrira sve funkcionalnosti klase Kvadrat

Tekst zadatka:

Napraviti klasu **Kvadrat** koja opisuje jedan kvadrat i pokretačku klasu **KvadratMain** koja demonstrira sve funkcionalnosti klase **Kvadrat**.

Klasa trebaju biti deo NetBeans projekta pod nazivom **KI104-V08** i deo paketa **cs101.v08.kvadrat**.

Za učitavanje podataka od korisnika treba koristiti objekat klase **Scanner**, a za ispit podataka objekat **System.out**.

Napomena:

S obzirom da u ovoj vežbi po prvi put kreiramo sve moguće funkcionalnosti jedne klase, od konstruktora, atributa i metoda do preopterećenih konstruktora, setter i getter metoda, statičkih atributa, statičkih metoda i pisanja dokumentacionih komentara, tekst zadatka ćemo detaljno definisati uporedo sa rešenjem u obliku isečaka programskog koda, po koracima:

PRIMER - KLASA KVADRAT - NASTAVAK 1

Klasa Kvadrat definiše veličinu stranice kvadrata i boju njegove unutrašnjosti, a klasa KvadratMain sadrži metod main za pokretanje aplikacije.

Korak 1:

Definisati klasu **Kvadrat** i kreirati sledeće javne attribute:

stranica, tipa int

bojaIvice, tipa String

bojaUnutrašnjosti, tipa String

Programski kod klase **Kvadrat**

```
package cs101.v07.kvadrat;

public class Kvadrat {
    public int stranica;
    public String bojaIvice;
    public String bojaUnutrasnjosti;

    public void ispiši(){
        System.out.println("Stranica kvadrata: \t\t" + stranica);
        System.out.println("Boja ivice kvadrata: \t\t" + bojaIvice);
        System.out.println("Boja unutrašnjosti kvadrata: \t" + bojaUnutrasnjosti);
    }
}
```

Programski kod klase **KvadratMain**:

```
package cs101.v07.kvadrat;

public class KvadratMain {
    public static void main(String[] args) {
        Kvadrat k1 = new Kvadrat();
        System.out.println("Objekat k1 klase Kvadrat: ");
        System.out.println("Stranica: " + k1.stranica + " Boja ivice: " +
```

```
k1.bojaIvice + " Boja unutrašnjosti: " + k1.bojaUnutrasnjosti);  
    }  
}
```

ZADACI ZA SAMOSTALNI RAD

Na osnovu prethodnih primera uraditi sledeće zadatke

Zadatak 2.1:

Na osnovu primera klase Student i zadatka 1 iz prethodnog poglavlja, dopunite klase CPU i Proizvodjac statičkim atributima

- ukupnoCPUa i
- ukupnoProizvodjaca

Napisati i metode

- povecajBrojCPU()
- povecajBrojProizvodjaca()

koje će kreiranjem novog CPU i Proizvodjac objekata uvećati gore navedene attribute za 1.

ZADACI ZA SAMOSTALNI RAD - 2

Proverite svoje znanje

Zadatak 2.2:

Dodaje ključnu reč *static* umesto ____ ako je potrebno.

```
public class Test {  
    int count;  
    public ____ void main(String[] args) {  
        ...  
    }  
    public ____ int getCount() {  
        return count;  
    }  
    public ____ int factorial(int n) {  
        int result = 1;  
        for (int i = 1; i <= n; i++)  
            result *= i;  
        return result;  
    }  
}
```

▼ 2.2 Modifikatori pristupa

TIPOVI MODIFIKATORA U JAVI

Modifikatori pristupa specificiraju dostupnost podataka - članova, metoda, konstruktora ili klase.

Programski jezika Java podržava dva tipa modifikatora:

- pristupni modifikatori;
- nepristupni modifikatori;

Postoje četiri modifikatora pristupa koji se intenzivno koriste u Java programima:

- **public;**
- **private;**
- **protected;**
- **default.**

Zadatak modifikatora pristupa jeste da specificiraju dostupnost (scope) podataka-članova, metoda, konstruktora ili klase.

Postoji mnogo nepristupnih modifikatora kao što su **static**, **abstract**, **synchronized**, **native**, **volatile**, **transient** itd. Ovde će akcenat biti na demonstraciji i analizi modifikatora pristupa.

Sledećom tabelom su prikazani modifikatori pristupa i oblasti u kojima su dostupni podaci obeleženi odgovarajućim modifikatorom.

Modifikator pristupa	unutar klase	unutar paketa	izvan paketa samo preko podklase	izvan paketa
Private	DA	NE	NE	NE
Default	DA	DA	NE	NE
Protected	DA	DA	DA	NE
Public	DA	DA	DA	DA

Slika 2.1.1 Modifikatori pristupa

Iz tabele se vidi da su privatni podaci (**private**) dostupni samo unutar klase u kojoj su kreirani. Javni (**public**) su dostupni svuda. Zaštićeni (**protected**) se vide u klasi, paketu i potklasama. Podrazumevani (**default**) modifikator je sličan zaštićenom ali je, kao što se vidi iz tabele, restriktivniji po pitanju vidljivosti u potklasama.

PRIMERI 1 I 2 - PRIMENA MODIFIKATORA PRISTUPA - PRIVATE I PUBLIC

Primeri koji pokazuju primenu modifikatora private i public.

Sledećim listingom je dat primer primene modifikatora pristupa **private**.

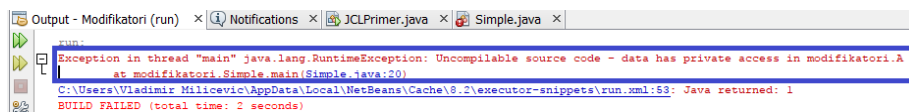

```
package modifikatori;

class A{
    private int data=40;
    private void msg(){
        System.out.println("Hello java");
    }
}

public class Simple{
    public static void main(String args[]){
        A obj=new A();

        //Compile Time Error se javlja
        // u sledećim linijama
        System.out.println(obj.data);
        obj.msg();
    }
}
```

Klasa A sadrži privatno polje *data* i privatnu metodu *msg()*. Ovako definisani članovi klase ne mogu da budu vidljivi izvan klase u kojoj su kreirani, pa tako ni u klasi *Simple* koja je kreirana u nastavku primera. Pokretanjem programa javiće se sledeća greška.



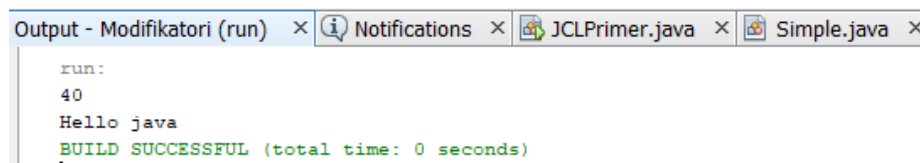
Slika 2.1.2 Pristup članovima klase obeleženim sa private - greška

Ukoliko bi, u aktuelnom primeru, u klasi A bila izvršena modifikacija i umesto modifikatora *private* bio upotrebljen *public*, pomenuto polje i metoda bi bili dostupni izvan klase u kojoj su kreirani. Slede novi listinzi klasa, a nakon toga i slika kojom je demonstrirano izvršavanje modifikovanog programa.

```
package modifikatori;

class A {
    public int data=40;
    public void msg(){
        System.out.println("Hello java");
    }
}

public class Simple{
    public static void main(String args[]){
        A obj=new A();
        System.out.println(obj.data);
        obj.msg();
    }
}
```



```
run:
40
Hello java
BUILD SUCCESSFUL (total time: 0 seconds)
```

Slika 2.1.3 Primena modifikatora public

PRIMERI 3 I 4 - PRIMENA MODIFIKATORA PRISTUPA - PROTECTED I DEFAULT

Ukoliko se član klase ne obeleži modifikatorom pristupa, Java program ga tretira kao da je obeležen modifikatorom default.

Ukoliko se član klase ne obeleži modifikatorom pristupa, Java program ga tretira kao da je obeležen modifikatorom default.

Sledi primer primene modifikatora pristupa **protected**. Kreirane su dve klase koje pripadaju različitim paketima, pri čemu je jedna klasa **super** klasa u odnosu na drugu. Sledi listing.

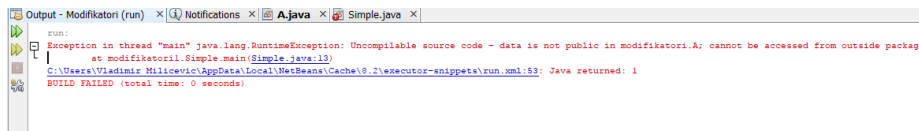
```
package modifikatori;

public class A {
    protected int data=40;
    protected void msg(){
        System.out.println("Hello java");
    }
}
*****
package modifikatoril;
import modifikatori.A;

public class Simple extends A {
    public static void main(String args[]){
        Simple obj=new Simple();
        System.out.println(obj.data);
        //Compile Time Error
        obj.msg();
        //Compile Time Error
    }
}
```

Modifikator pristupa **protected** dozvoljava da obeleženo članovi klase budu vidljivi i unutar drugog paketa ali u potklasi. Rezultat izvršavanja bi bio identičan kao na slici 3.

Ukoliko bi iz listinga bio izbrisan modifikator **protected** javila bi se greška prilikom izvršavanja programa. Tada bi konkretne članice klase imale status **default** koji je restriktivniji u odnosu na **protected**. To znači da članice klase ne bi bile vidljive izvan matičnog paketa, čak i u slučaju pozivanja iz potklase. U slučaju izvršavanja ovakvog programa, javila bi se greška kao na sledećoj slici.



Slika 2.1.4 Primena default modifikatora u klasama iz različitih paketa

PRIMER - KLASA KVADRAT - NASTAVAK 2

Klasa Kvadrat definiše veličinu stranice kvadrata i boju njegove unutrašnjosti, a klasa KvadratMain sadrži metod main za pokretanje aplikacije.

Korak 1:

Definisati klasu **Kvadrat** i kreirati sledeće javne attribute:

stranica, tipa int

bojaIvice, tipa String

bojaUnutrašnjosti, tipa String

Programski kod klase **Kvadrat**

```
package cs101.v07.kvadrat;

public class Kvadrat {
    private int stranica;
    private String bojaIvice;
    private String bojaUnutrasnjosti;

    public void ispiši(){
        System.out.println("Stranica kvadrata: \t\t" + stranica);
        System.out.println("Boja ivice kvadrata: \t\t" + bojaIvice);
        System.out.println("Boja unutrašnjosti kvadrata: \t" + bojaUnutrasnjosti);
    }
}
```

Programski kod klase **KvadratMain**:

```
package cs101.v07.kvadrat;

public class KvadratMain {
    public static void main(String[] args) {
        Kvadrat k1 = new Kvadrat();
        System.out.println("Objekat k1 klase Kvadrat: ");
        System.out.println("Stranica: " + k1.stranica + " Boja ivice: " +
            k1.bojaIvice + " Boja unutrašnjosti: " + k1.bojaUnutrasnjosti);
    }
}
```

ZADACI ZA SAMOSTALNI RAD

Na osnovu prethodnih primera uraditi sledeće zadatke

Zadatak 2.3:

Na osnovu primera i zadatka 2 iz prethodnog poglavlja, dodeilte modifikatore pristupa njihovim atributima i metodama:

Klasa CPU: svi atributi su privatni, osim atributa `ukupnoCPU` koji je javan

- svi atributi su privatni, osim atributa `ukupnoCPU` koji je javan
- metoda `povecajBrojCPU()` kojom se uvećava `ukupnoCPU` je privatna

Klasa Proizvodjac:

- svi atributi su zaštićeni, osim `ukupnoProizvodjaca` koji je javan
- metoda `povecajBrojProizvodjaca()` kojom se uvećava `ukupnoProizvodjaca` je privatna

▼ 2.3 Opšti oblik definisanja klase

OPŠTI OBLIK PREDSTAVLJANJA KLASE

Telo klase sadrži definicije statičkih i nestatičkih članova (atributa i metoda) klase

Opšti oblik definicije obične klase u Javi je vrlo jednostavan:

```
modifikatoriKlase class ImeKlase {  
    teloKlase  
}
```

Modifikatori klase na početku definicije klase nisu obavezni, ali se mogu sastojati od jedne ili više rezervisanih reči kojima se određuju izvesne karakteristike klase koja se definiše. Na primer, modifikator pristupa **public** ukazuje na to da se klasa može koristiti izvan svog paketa. Inače, bez tog modifikatora, klasa se može koristiti samo unutar svog paketa. Na raspolaganju su još tri modifikatora.

Ime klase se gradi na uobičajeni način, uz napomenu da sve reči imena klase počinju velikim slovom.

Telo klase sadrži definicije *statičkih i nestatičkih članova* (atributa i metoda) klase.

Na primer, ako izmenimo definiciju klase **Student**:

Nijedan od članova klase **Student** nema modifikator **static**, što znači da ta klasa nema *statičkih članova* i da je zato ima smisla koristiti samo za konstruisanje objekata.

Svaki objekat koji je instanca klase **Student** sadrži attribute ime, **brojPoloženihIspita** i **nizOcena**, kao i metod **prosekOcena()**. Ovi atributi u različitim objektima imaju generalno različite vrednosti.

Metod **prosekOcena()** primenjen za različite objekte klase Student daje različite rezultate, jer će se za *svaki objekat koristiti konkretne vrednosti njegovih atributa*. Ovaj efekat je zapravo tačno ono što se misli kada se kaže da **objektni metod** pripada pojedinačnim objektima, a ne klasi.

```
public class Student {
    String ime;
    int brojPoloženihIspita;
    int[] nizOcena;

    public double prosekOcena() {
        int sumaOcena = 0;
        for (int i=0; i<nizOcena.length; i++){
            sumaOcena+= nizOcena[i];
        }
        return (double)sumaOcena/nizOcena.length;
    }
}
```

DUŽINA TRAJANJA PROMENLJIVIH

Postojanje promenljive u programu se završava kada se oslobodi memorijski prostor koji je rezervisan za promenljivu.

Svaka promenljiva ima svoj "životni vek" u memoriji tokom izvršavanja programa. Postojanje neke promenljive u programu počinje od trenutka izvršavanja naredbe deklaracije promenljive. Time se rezerviše memorijski prostor odgovarajuće veličine za smeštanje vrednosti promenljive. Ovaj postupak se naziva **alociranje promenljive**.

Postojanje promenljive u programu se završava kada se oslobodi memorijski prostor koji je rezervisan za promenljivu, moguće da bi se iskoristio za neke druge potrebe. Ovaj postupak se naziva dealociranje promenljive, a momenat njegovog dešavanja tokom izvršavanja programa zavisi od vrste promenljive.

U telu nekog metoda se mogu koristiti tri vrste promenljivih:

- **lokalne promenljive** deklarisanе unutar metoda
- **parametri metoda**
- **globalne promenljive** deklarisanе izvan metoda, ali u istoj klasi kao i metod

Ove globalne promenljive su naravno drugo ime za attribute klase.

Lokalne promenljive se dealociraju čim se izvrši poslednja naredba metoda u kojem su deklarisanе i neposredno pre povratka na mesto gde je taj metod pozvan. Lokalne promenljive postoje samo za vreme izvršavanja svog metoda, pa se zato ni ne mogu koristiti

negde izvan svog metoda, jer tamo više ne postoje. Lokalne promenljive dakle nemaju nikakve veze sa okruženjem metoda, odnosno one su potpuno deo internog rada metoda.

Parametri metoda služe za prihvatanje ulaznih vrednosti metoda od argumenata kada se metod pozove. Kako parametri metoda imaju konceptualno istu ulogu kao lokalne promenljive, za parametre važi slično pravilo kao i za lokalne promenljive: parametri metoda se alociraju u momentu poziva metoda za izvršavanje i dealociraju u momentu završetka izvršavanja metoda.

TRAJANJE GLOBALNIH I LOKALNIH PROMENLJIVIH

Globalna promenljiva metoda postoji od momenta kada se njena klasa prvi put koristi pa sve do kraja izvršavanja celog programa. Lokalne promenljive dobijaju vrednost pre korišćenja.

Globalna promenljiva metoda koja je deklarisan unutar neke klase postoji, grubo rečeno, od momenta kada se njena klasa prvi put koristi pa sve do kraja izvršavanja celog programa.

Tačni trenuci alociranja i dealociranja globalnih promenljivih nisu toliko bitni, nego je važno samo to što su globalne promenljive u nekoj klasi nezavisne od metoda te klase i uvek postoje dok se metodi te klase izvršavaju. Zato globalne promenljive dele svi metodi u istoj klasi. To znači da promene vrednosti globalne promenljive u jednom metodu imaju prošireni efekat na druge metode: ako se u jednom metodu promeni vrednost nekoj globalnoj promenljivoj, onda se u drugom metodu koristi ta nova vrednost globalne promenljive ukoliko ona učestvuje u, recimo, nekom izrazu.

Dakle, globalne promenljive su zajedničke za sve metode neke klase, za razliku od lokalnih promenljivih (i parametara) koje pripadaju samo jednom metodu.

Treba istaći još par detalja u vezi sa inicijalizacijom promenljivih prilikom njihovog alociranja:

Lokalne promenljive metoda se u Javi ne inicijalizuju nikakvom vrednošću prilikom alociranja tih promenljivih na samom početku izvršavanja metoda. Stoga one moraju imati eksplicitno dodeljenu vrednost u metodu pre nego što se mogu koristiti.

Parametri metoda se inicijalizuju vrednostima argumenata, prilikom alociranja na početku izvršavanja metoda.

Globalne promenljive (atributi ili polja) se automatski inicijalizuju unapred definisanim vrednostima. Za numeričke promenljive, ta vrednost je nula; za logičke promenljive, ta vrednost je false; i za znakovne promenljive, ta vrednost je Unicode znak '\u0000', za objektne promenljive, ta inicijalna vrednost je specijalna vrednost null.

OBLAST VAŽENJA IMENA PROMENLJIVE

Programiranje se u svojoj suštini svodi na davanje imena, neko ime (ili identifikator) u programu može da označava razne konstrukcije: promenljive, metode, klase i tako dalje.

Ove konstrukcije se preko svojih imena koriste na raznim mestima u tekstu programa, tj. Dinamički se alociraju i dealociraju tokom izvršavanja programa. Teorijski je zato moguće da se neke programske konstrukcije koriste u tekstu programa iako one tokom izvršavanja programa fizički više ne postoje u memoriji. Da bi se izbegle ove vrste grešaka, u Javi postoje pravila o tome gde se uvedena (prosta) imena mogu koristiti u programu.

Oblast programskog koda u kome se može upotrebiti neko ime, radi korišćenja programske konstrukcije koju označava, naziva se oblast važenja imena ili domen (engl. scope).

Oblast važenja imena globalne promenljive (atributa) je cela klasa u kojoj je globalna promenljiva definisana. To znači da se globalna promenljiva može koristiti u tekstu cele klase, čak i eventualno ispred naredbe njene deklaracije.

Oblast važenja imena lokalne promenljive je od mesta njene deklaracije do kraja bloka u kome se njena deklaracija nalazi. Zato se lokalna promenljiva može koristiti samo u svom metodu, i to od naredbe svoje deklaracije do kraja bloka u kojem se nalazi ta naredba deklaracije.

Ova pravila za oblast važenja imena promenljivih su ipak malo složenija, jer je dozvoljeno lokalnoj promenljivoj ili parametru dati isto ime kao nekoj globalnoj promenljivoj. U tom slučaju, unutar oblasti važenja lokalne promenljive ili parametra, globalna promenljiva je zaklonjena. Da bismo ovo ilustrovali, razmotrimo klasu `Igra` čija definicija ima sledeći oblik:

```
class Igra{
    // Globalna promenljiva (atribut)
    static String pobednik;

    static void odigrajIgru() {
        // Lokalna promenljiva
        String pobednik;
        ...
        // Ostale naredbe metoda ...
        ...
    }
    ...
}
```

U telu metoda **odigrajIgru()**, ime pobednik se odnosi na lokalnu promenljivu. U ostatku klase **Igra**, ime pobednik se odnosi na globalnu promenljivu (atribut), sem ako to ime nije opet zaklonjeno istim imenom lokalne promenljive ili parametra u nekom drugom metodu. Ako se ipak mora koristiti statički atribut pobednik unutar metoda `odigrajIgru()`, onda se mora pisati njegovo puno ime `Igra.pobednik`.

PRIMER - VAŽENJE PROMENLJIVIH UNUTAR FOR PETLJE

Oblast važenja ovako deklarisanе kontrolne promenljive je samo kontrolni deo i telo for petlje, odnosno ne proteže se do kraja tela metoda koji sadrži for petlju

Ove komplikacije, kada lokalna promenljiva ili parametar imaju isto ime kao globalna promenljiva, uzrok su mnogih grešaka koje se najjednostavnije izbegavaju davanjem različitih imena radi lakšeg razlikovanja. To je upravo i preporuka dobrog stila programiranja koje se treba pridržavati.

Drugi specijalni slučaj oblasti važenja imena promenljivih pojavljuje se kod deklarisanja kontrolne promenljive petlje unutar for petlje:

```
for (int i = 0; i < n; i++) {
    ...
    // Telo petlje
    ...
}
```

U ovom primeru bi se moglo reći da je promenljiva i deklarisan lokalno u odnosu na metod koji sadrži for petlju. Ipak, oblast važenja ovako deklarisanе kontrolne promenljive je samo kontrolni deo i telo for petlje, odnosno ne proteže se do kraja tela metoda koji sadrži for petlju. Zbog toga se vrednost brojača for petlje ne može koristiti van te petlje, na primer, ovo nije dozvoljeno:

```
for (int i = 0; i < n; i++) {
    ...
    // Telo petlje
    ...
}
if (i == n) // GREŠKA: ovde ne važi ime i
    System.out.println("Završene sve iteracije");
```

Oblast važenja imena parametra nekog metoda je blok od kojeg se sastoji telo tog metoda. Nije dozvoljeno redefinisati ime parametra ili lokalne promenljive u oblasti njihovog važenja, čak ni u ugnježđenom bloku, na primer, ni ovo nije dozvoljeno:

```
void neispravanMetod(int n) {
    int x;
    while (n > 0) {
        int x; // GREŠKA: ime x je već definisano
        ...
    }
}
```

OBLAST VAŽENJA METODA

Izvan oblasti važenja lokalne promenljive ili parametra se njihova imena mogu ponovo davati. Oblast važenja nekog metoda je cela klasa u kojoj je metod definisan

Globalne promenljive se mogu redefinisati, dok se lokalne promenljive i parametri ne mogu redefinisati. S druge strane, izvan oblasti važenja lokalne promenljive ili parametra se njihova imena mogu ponovo davati, na primer:


```
void ispravanMetod(int n) {
    while (n > 0) {
        int x;
        ...
    } // Kraj oblasti važenja imena x
    while (n > 0) {
        int x; // OK: ime x se može opet davati
        ...
    }
}
```

Pravilo za oblast važenja imena metoda je slično onom za globalne promenljive: oblast važenja nekog metoda je cela klasa u kojoj je metod definisan. To znači da je dozvoljeno pozivati metod na bilo kom mestu u klasi, uključujući tačke u programskom tekstu klase koje se nalaze ispred tačke definicije metoda. Moguće je pozivati neki metod i unutar definicije samog metoda i u tom slučaju se radi, o rekurzivnim metodima.

Ovo opšte pravilo ima dodatni uslov s obzirom na to da metodi mogu biti statički ili objektni. Naime, *objektni članovi klase (metodi i atributi), koji pripadaju nekom objektu klase, ne moraju postojati u memoriji dok se statički metod izvršava*. Na primer, statički metod neke klase se može pozvati upotrebom njegovog punog imena u bilo kojoj tački programa, a do tada objekat kome pripadaju objektni članovi iste klase možda nije još bio ni konstruisan tokom izvršavanja programa. Zato se objektni metodi i polja klase ne mogu koristiti u statičkim metodima iste klase.

OBLAST VAŽENJA (SCOPE) PROMENLJIVIH (VIDEO)

Video objašnjava oblast važenja promenljivih i metoda klase

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

ZADACI ZA SAMOSTALNI RAD

Na osnovu prethodnih primera uraditi sledeće zadatke

Zadatak 2.5:

Na osnovu zadatka 2.3 iz prethodnog poglavlja, u klasama CPU i Proizvođač, dodati metodama *povecajBrojCPU()* i *povecajBrojProivodjaca()* lokalne promenljive.

Kreirati klasu Main i u njoj po dva objekata obe klase. Pristupiti loklanim promenljivama u metodma *povecajBrojCPU()* i *povecajBrojProivodjaca()*

ZADACI ZA SAMOSTALNI RAD - 2

Proverite svoje znanje

Zadatak 2.6:

Šta je izlaz iz sledećeg koda?

```
public class Test {  
    private static int i = 0;  
    private static int j = 0;  
  
    public static void main(String[] args) {  
        int i = 2;  
        int k = 3;  
        {  
            int j = 3;  
            System.out.println("i + j is " + i + j);  
        }  
        k = i + j;  
        System.out.println("k is " + k);  
        System.out.println("j is " + j);  
    }  
}
```

▼ 2.4 Promenljive klasnog tipa

REFERENCE KLASA

Klase predstavljaju novi tip podataka u programu. Promenljiva klasnog tipa ne sadrži neki objekat kao svoju vrednost, već referencu na taj objekat.

Jedan aspekt definicije neke klase je to što se time opisuje struktura svih objekata koji pripadaju klasi.

Drugi aspekt definicije neke klase je to što se time uvodi novi tip podataka u programu.

Vrednosti tog klasnog tipa su objekti same klase. To znači da se ime definisane klase može koristiti za tip promenljive u naredbi deklaracije, kao i za tip formalnog parametra i za tip rezultata u definiciji nekog metoda.

Na primer, ako imamo u vidu sledeću definiciju klase **Student**:

```
public class Student {  
    String ime;  
    int brojPoloženihIspita;  
  
    public double prosekOcena() {  
        return 8.0;  
    }  
}
```

u programu se može deklarirati promenljiva klasnog tipa Student:

```
Student s;
```

Kao što je to uobičajeno, ovom naredbom se u memoriji računara rezerviše prostor za promenljivu a radi čuvanja njenih vrednosti tipa Student. Međutim, iako vrednosti tipa Student jesu objekti klase Student, promenljiva s ne sadrži ove objekte. U Javi važi opšte pravilo da nijedna promenljiva nikad ne sadrži neki objekat.

Da bismo ovo razjasnili, moramo malo bolje razumeti postupak konstruisanja objekata. Objekti se konstruišu u specijalnom delu memorije programa koji se zove **hip memorija** (engl. heap). Pritom se, zbog brzine, ne vodi mnogo računa o redu po kojem se zauzima taj deo memorije - *novi objekat se smešta u hip memoriju tamo gde se nađe prvo slobodno mesto*, a i da se njegovo mesto prosto oslobađa kada nije više potreban u programu.

Naravno, da bi se moglo pristupiti objektu u programu, mora se imati informacija o tome gde se on nalazi u hip memoriji - ta informacija se naziva **referenca ili adresa objekta** i svodi se na adresu memorijske lokacije objekta u hip memoriji.

Promenljiva klasnog tipa ne sadrži neki objekat kao svoju vrednost, već referencu na taj objekat. Zato kada se u programu koristi promenljiva klasnog tipa, ona služi za indirektan pristup objektu na koga ukazuje aktuelna vrednost (referenca) te promenljive.

KONSTRUISANJE (STVARANJE) OBJEKTA

Konstruisanje objekata svake klase u programu se izvodi posebnim operatorom čija je oznaka ključna reč new

Konstruisanje objekata svake klase u programu se izvodi posebnim operatorom čija je oznaka ključna reč **new**. Pored konstruisanja jednog objekta, odnosno rezervisanja potrebnog memorijskog prostora u hipu, rezultat ovog operatora je i vraćanje reference na novokonstruisani objekat - to je neophodno da bi se kasnije u programu moglo pristupiti novom objektu i s njim uraditi nešto korisno. Zbog toga se vraćena referenca na novi objekat mora sačuvati u promenljivoj klasnog tipa, jer inače se novom objektu nikako drugačije ne može pristupiti.

Na primer, ako je promenljiva s tipa **Student** deklarirana kao u prethodnom primeru, onda izvršavanjem naredbe dodele:

```
s = new Student();
```

najpre bi se konstruisao novi objekat koji je instanca klase **Student**, a zatim bi se referenca na njega dodelila promenljivoj s. Prema tome, vrednost promenljive s je referenca na novi objekat, a ne sam taj objekat.

Nije zato sasvim ispravno kratko reći da je taj objekat vrednost promenljive s, mada je ponekad teško izbeći ovu kraću terminologiju, još manje je ispravno reći da promenljiva s sadrži taj objekat.

Ispravan način izražavanja je da promenljiva s ukazuje na novi objekat.

Iz definicije klase **Student** može se zaključiti da novokonstruisani objekat klase Student, na koga ukazuje promenljiva **s**, sadrži attribute ime **brojPoloženihIspita**. U programu se ovi atributi tog konkretnog objekta mogu koristiti pomoću standardne tačka-notacije: **s.ime** i **s.brojPoloženihIspita**. Tako, recimo, može se napisati sledeća naredba:

```
System.out.println("Student " + s.ime + " je položio: " + s.brojPoloženihIspita + " ispita.");
```

Ovom naredbom bi se na ekranu prikazalo ime i broj položenih ispita studenta predstavljenog objektom na koga ukazuje promenljiva **s**. Generalno, atributi **s.ime** i, recimo, **s.brojPoloženihIspita** mogu se koristiti u programu na svakom mestu gde je dozvoljena upotreba promenljivih tipa **String**, odnosno **int**. Na primer, ako treba odrediti broj znakova stringa u atributu **s.ime**, onda se može koristiti zapis:

```
s.ime.length()
```

REFERENCE OBJEKATA

*Članovima objekta se pristupa koristeći oznaku **imeObjekta.imeAtributa** i **imeObjekta.metod()**. Ovde je **imeObjekta** referenca na objekat.*

Slično, objektni metod **prosekOcena()** se može pozvati za objekat na koga ukazuje **s** zapisom **s.prosekOcena()**. Da ne bismo komplikovali objašnjavanje, napravili smo da ova metoda za sve studente vraća vrednost 8.0. Da bi se prikazala prosečna ocena studenta na koga ukazuje **s** može se pisati, na primer:

```
System.out.print("Prosečna ocene studenta "
+ s.ime);
System.out.println(" je: " + s.prosekOcena());
```

U nekim slučajevima je potrebno naznačiti da promenljiva klasnog tipa ne ukazuje ni na jedan objekat - to se postiže dodelom specijalne reference **null** promenljivoj klasnog tipa. Na primer, može se pisati naredba dodele:

```
s = null;
```

ili naredba grananja u obliku

```
if (s == null)
    ...
```

Ako je vrednost promenljive klasnog tipa jednaka null, onda ta promenljiva ne ukazuje ni na jedan objekat, pa se preko te promenljive ne mogu koristiti objektni atributi i metodi odgovarajuće klase. Tako, ako promenljiva **s ima vrednost null, onda bi bila greška koristiti, recimo, **s.prosekOcena()**.**

Da bi smo bolje razumeli vezu između promenljivih klasnog tipa i objekata, razmotrimo detaljnije sledeći programski fragment:

```
Student s1, s2, s3, s4;
s1 = new Student();
s2 = new Student();
s1.ime = "Milorad Novaković";
s2.ime = "Sonja Stošić";
s3 = s1;
s4 = null;
...
```

Prvom naredbom u ovom programskom fragmentu se deklariraju četiri promenljive klasnog tipa **Student**.

Drugom i trećom naredbom se konstruišu dva objekta klase **Student** i **reference** na njih se redom dodeljuju promenljivim **s1** i **s2**.

Narednim naredbama se atributu ime ovih novih objekata dodeljuju vrednosti odgovarajućih stringova.

Na kraju, pretposlednjom naredbom se vrednost promenljive **s1** dodeljuje promenljivoj **s3**, a poslednjom naredbom se referenca **null** dodeljuje promenljivoj **s4**.

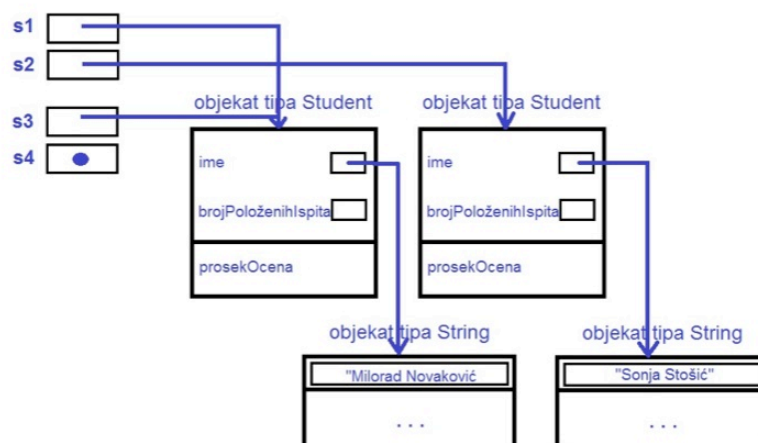
Stanje memorije posle izvršavanja svih naredbi u ovom primeru prikazano je na slici. 1 (sledeći slajd) Promenljive su prikazane u obliku malih pravougaonika, a objekti u obliku većih pravougaonika sa zaobljenim uglovima.

PRISTUP HIP MEMORIJI SA OBJEKTOM

Kada se jedna promenljiva klasnog tipa dodeljuje drugoj, onda se kopira samo referenca, ali ne i objekat na koga ta referenca ukazuje.

Ako promenljiva sadrži referencu na neki objekat, vrednost te promenljive je prikazana u obliku strelice koja pokazuje na taj objekat.

Ako promenljiva sadrži referencu **null**, ona ne pokazuje ni na jedan objekat, pa je vrednost te promenljive prikazana kao podebljana tačka.



Slika 2.2.1 Veza između promenljivih klasnog tipa i objekata

Sa slike se vidi da promenljive **s1** i **s3** ukazuju na isti objekat. Ta

činjenica je posledica izvršavanja prethodne naredbe dodele **s3 = s1**; kojom se kopira referenca iz **s1** u **s3**. Obratite pažnju na to da ovo važi u opštem slučaju: kada se jedna promenljiva klasnog tipa dodeljuje drugoj, onda se kopira samo referenca, ali ne i objekat na koga ta referenca ukazuje.

To znači da je vrednost atributa **s1.ime** posle izvršavanja naredbi u prethodnom primeru jednaka stringu "Milorad Novaković", ali i da je vrednost atributa **s3.ime** jednaka "Milorad Novaković".

U Javi se jednakost ili nejednakost promenljivih klasnog tipa može proveravati relacionim operatorima `==` i `!=`, ali pri tome treba biti obazriv, nastavljajući prethodni primer, ako se napiše, recimo:

```
if (s1 == s3)
    ...
```

onda se, naravno, ispituje da li su vrednosti promenljivih **s1** u **s3** jednake. Ali, te vrednosti su reference, tako da se time ispituje samo da li promenljive **s1** i **s3** ukazuju na isti objekat, ali ne i da li su jednaki objekti na koje ove promenljive ukazuju. To je u nekim slučajevima baš ono što je potrebno, ali *ako treba ispitati jednakost samih objekata na koje promenljive ukazuju, onda se mora pojedinačno ispitati jednakost svih atributa tih objekata.*

STACK I HEAP MEMORIJA

Video ukratko objašnjava šta je stack, a šta je heap memorija

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

PRIMER - RAZLIKA IZMEĐU REFERENCE NA OBJEKAT I SAMOG OBJEKTA

Objekti se fizički nalaze negde u hip memoriji, a promenljive klasnog tipa samo ukazuju na njih.

Kako su stringovi (nizovi oznaka) zapravo objekti klase **String**, a ne primitivne vrednosti, na slici su stringovi (nizovi oznaka) "Milorad Novaković" i "Sonja Stošić" prikazani kao objekti.

Neka promenljiva klasnog tipa **String** može dakle sadržati samo referencu na string, kao i referencu null, a ne i sam niz znakova koji čine string - to objašnjava zašto su na slici vrednosti dva primerka atributa ime tipa String prikazane strelicama koje pokazuju na objekte odgovarajućih stringova.

Primetimo da se u vezi sa stringovima često primenjuje neprecizna objektna terminologija, iako su stringovi pravi objekti kao i svaki drugi u Javi. Na primer, kažemo da je vrednost

atributa `s1.ime` baš string "Milorad Novaković", a ne pravilno - ali rogovatnije, da je vrednost atributa `s1.ime` referenca na objekat stringa "Milorad Novaković".

Spomenimo na kraju još dve logične posledice činjenice da promenljive klasnog tipa sadrže reference na objekte, a ne same objekte. Objekti se fizički nalaze negde u hip memoriji, a promenljive klasnog tipa samo ukazuju na njih.

Pretpostavimo da se vrednost promenljive klasnog tipa **x** prenosi kao argument prilikom poziva nekog metoda. Onda se, kao što je poznato, vrednost promenljive **x** dodeljuje odgovarajućem

parametru metoda i metod se izvršava. Vrednost promenljive **x** se ne može promeniti izvršavanjem metoda, ali pošto dodeljena vrednost odgovarajućem parametru predstavlja referencu na neki objekat, u metodu se mogu promeniti podaci u tom objektu. Nakon izvršavanja metoda, promenljiva **x** će i dalje ukazivati na isti objekat, ali podaci u tom objektu mogu biti promenjeni. Konkretnije, pod pretpostavkom da je definisan metod:

```
void promeni(Student s) {  
    s.ime = "Dragan Janković";  
}
```

i da se izvršava ovaj programski fragment:

```
Student x = new Student();  
x.ime = "Milorad Novaković";  
promeni(x);  
System.out.println(x.ime);
```

nakon njegovog izvršavanja se na ekranu prikazuje string "Dragan Janković" kao vrednost atributa **x.ime**, a ne "Milorad Novaković". Naime, izvršavanje naredbe poziva `promeni(x)` ekvivalentno je izvršavanju dve naredbe dodele:

```
s = x;  
s.ime = "Dragan Janković";
```

odnosno atribut `ime` objekta na koji ukazuje promenljiva **s**, a time i **x**, dobiće novu vrednost.

PREBACIVANJE VREDNOSTI I REFERENCI

Video objašnjava razliku prebacivanje podataka po vrednosti i preko reference

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

ZADACI ZA SAMOSTALNI RAD

Na osnovu predhodnih primera uraditi sledeće zadatke

Zadatak 2.7:

Na osnovu zadatka 2.3 iz poglavlja *Modifikatori pristupa*, kreirati (ukoliko niste u prethodnom primeru) klasu Main.

Kreirati po dva objekta klase CPU i Proizvodjac i dodeliti vrednost svim atributima.

Na kraju, ispisati na standaradan izlaz podatke sačuvane u tim objektima.

▼ 2.5 Konstrukcija, inicijalizacija i uklanjanje objekata

KONSTRUKCIJA OBJEKATA

Vrednosti klasnih tipova (reference) se moraju eksplicitno definisati u programu. Atributima se mogu dodeliti početne vrednosti u njihovim deklaracijama.

Klasni tipovi u Javi su vrlo različiti od primitivnih tipova: vrednosti primitivnih tipova su "ugrađene" u jezik, dok se vrednosti klasnih tipova, koje predstavljaju objekte odgovarajuće klase, moraju u programu eksplicitno konstruisati.

S druge strane, ne postoji suštinska razlika u postupanju sa promenljivim primitivnih i klasnih tipova, jer se razlikuju samo vrednosti koje mogu biti njihov sadržaj - kod jednih su to primitivne vrednosti, a kod drugih su to reference.

Postupak konstruisanja jednog objekta se sastoji od dva koraka:

- rezervisanja dovoljno mesta u hip memoriji za smeštanje objekta
- inicijalizacije njegovih polja podrazumevanim vrednostima

Programer ne može da utiče na prvi korak pronalaženja mesta u memoriji za smeštanje objekta, ali za drugi korak u složenijem programu obično nije dovoljna samo **podrazumevana inicijalizacija**. Podsetimo se da se ova automatska inicijalizacija sastoji od dodele vrednosti nula atributima numeričkog tipa (int, double, . . .), dodele vrednosti false logičkim atributima, dodele Unicode znaka 'u0000' znakovnim atributima i dodele reference null atributima klasnog tipa. Ukoliko podrazumevana

inicijalizacija nije dovoljna, atributima se mogu dodeliti početne vrednosti u njihovim deklaracijama, baš kao što se to može uraditi za bilo koje druge promenljive.

Radi konkretnosti, pretpostavimo da u programu za simuliranje neke društvene igre treba predstaviti kocku za bacanje. U nastavku je prikazana klasa **KockaZalgru** čiji su objekti računarske reprezentacije realnih objekata kocki za bacanje u društvenim igrama.

Ova klasa sadrži jedan objektni atribut čiji sadržaj odgovara broju palom pri bacanju kocke i jedan objektni metod kojim se simulira slučajno bacanje kocke.


```
public class KockaZaIgru {
    public int broj = (int)(Math.random()*6)+1;

    public void baci() {
        broj = (int)(Math.random()*6)+1;
    }
}
```

U ovom primeru klase **KockaZaIgru**, atribut broj dobija slučajnu vrednost svaki put kada se konstruiše objekat klase **KockaZaIgru**. Dakle, različiti objekti klase **KockaZaIgru** imaju verovatno različite početne vrednosti svojih primeraka atributa **broj**.

INICIJALIZACIJA OBJEKATA

Operator new kreira novi objekat navedene klase, tj. pozivom njenog konstruktora i vraća referencu na kreirani objekat.

Inicijalizacija statičkih atributa neke klase nije mnogo drugačija od inicijalizacije nestatičkih (objektnih) atributa, osim što treba imati u vidu da *statički atributi nisu vezani za pojedinačne objekte nego za klasu kao celinu*. Kako postoji samo jedan primerak statičkog atributa klase, on se inicijalizuje samo jednom i to kada se klasa po prvi put učitava od strane virtuelne mašine.

Konstruisanje objekata u programu se obavlja posebnim operatorom **new**, na primer, u programu koji koristi klasu **KockaZaIgru** može se pisati:

```
/* Deklaracija promenljive
   klasnog tipa KockaZaIgru */
KockaZaIgru kocka;

// Konstruisanje objekta klase
kocka = new KockaZaIgru();
```

Isti efekat se može postiti kraćim zapisom:

```
KockaZaIgru kocka = new KockaZaIgru();
```

U ovim primerima se izrazom:

```
new KockaZaIgru()
```

na desnoj strani znaka jednakosti konstruiše novi objekat klase **KockaZaIgru**, inicijalizuju se njegovi objektni atributi i vraća se referenca na taj novi objekat kao rezultat tog izraza.

KONSTRUKTORI

Konstruktor je metod koji stvara objekte svoje klase. Konstruktor ima ime jednako imenu klase, parametre i modifikator pristupa (public, private ili protected).

Primitimo da deo **KockaZalgru()** iza operatora new podseća na poziv metoda, zapravo, to je poziv specijalnog metoda koji se naziva konstruktor klase, iako se u definiciji klase KockaZalgru ne nalazi takav metod!

Međutim, svaka klasa ima bar jedan konstruktor, koji se automatski dodaje ukoliko nije eksplicitno definisan nijedan drugi konstruktor. Taj podrazumevani konstruktor ima samo formalnu funkciju i praktično ne radi ništa. Međutim, s obzirom da se konstruktor poziva odmah nakon konstruisanja objekta i inicijalizacije njegovih atributa podrazumevanim vrednostima, u klasi se može definisati jedan ili više posebnih konstruktora u klasi radi dodatne inicijalizacije novih objekta.

Definicija konstruktora klase se piše na isti način kao definicija običnog metoda, uz tri razlike:

- Ime konstruktora mora biti isto kao ime klase u kojoj se definiše
- Jedini dozvoljeni modifikatori su public, private i protected
- Konstruktor nema tip rezultata, čak ni void

S druge strane, konstruktor sadrži uobičajeno telo metoda u formi bloka naredbi unutar kojeg se mogu pisati bilo koje naredbe.

Takođe, konstruktor može imati parametre kojima se mogu preneti vrednosti za inicijalizaciju objekata nakon njihove konstrukcije.

U drugoj verziji klase KockaZalgru je definisan poseban konstruktor koji ima parametar za početnu vrednost broja koji pokazuje kocka

```
public class KockaZaIgru {  
    public int broj;  
    public KockaZaIgru(int n) {  
        broj = n;  
    }  
    public void baci() {  
        broj = (int)(Math.random() * 6) + 1;  
    }  
}
```

U ovom primeru klase KockaZalgru, konstruktor:

```
public KockaZaIgru(int n) {  
    broj = n;  
}
```

KORIŠĆENJE KONSTRUKTORA

Konstruktor ima isto ime kao klasa i nema tip rezultata, jer je uvek rezultat u obliku novog objekta date klase, a rezultat je referenca (adresa) tog objekta.

Konstruktor isto ime kao klasa, nema tip rezultata i ima jedan parametar. Poziv konstruktora, sa odgovarajućim argumentima, navodi se iza operatora `new`. Na primer:

```
KockaZaIgru kocka = new KockaZaIgru(1);
```

Na desnoj strani znaka jednakosti ove naredbe deklaracije promenljive `kocka` konstruiše se novi objekat klase **KockaZalgru**, poziva se konstruktor te klase kojim se atribut objekta inicijalizuje vrednošću argumenta `1` i, na kraju broj novokonstruisanog, referenca na taj objekat se dodeljuje promenljivoj `kocka`.

Podrazumevani konstruktor se dodaje klasi samo ukoliko nije eksplicitno definisan nijedan drugi konstruktor u klasi. Zato, na primer, izrazom **`new KockaZalgru()`** ne bi više mogao da se konstruiše objekat prethodne klase **KockaZalgru**. Međutim, i konstruktori mogu preopterećivati kao i obični metodi - to znači da **klasa može imati više preopterećenih konstruktora pod uslovom, naravno, da su njihovi potpisi različiti**.

Na primer, prethodnoj klasi **KockaZalgru** može se dodati konstruktor bez parametara koji atributu broj konstruisanog objekta početno dodeljuje slučajan broj:

```
public class KockaZaIgru {  
    public int broj;  
    public KockaZaIgru() {  
        baci();  
    }  
    public KockaZaIgru(int n) {  
        broj = n;  
    }  
    public void baci() {  
        broj = (int)(Math.random() * 6) + 1;  
    }  
}
```

Objekti ove klase **KockaZalgru** se mogu konstruisati na dva načina: bilo izrazom **`new KockaZalgru()`** ili izrazom **`new KockaZalgru(x)`**, gde je **`x`** izraz tipa **`int`**.

Na osnovu svega do sada rečenog može se zaključiti da su konstruktori specijalna vrsta metoda - oni nisu objektni metodi jer ne pripadaju objektima, već se pozivaju samo u trenutku konstruisanja objekata. Mada, oni nisu ni statički metodi klase, jer se za njih ne može koristiti modifikator **`static`**.

POSTUPAK KREIRANJA I INICIJALIZACIJE OBJEKTA

Konstruktori se mogu pozvati samo uz operator new. Prvo se analizom argumenata dodeljuje odgovarajući prostor u memoriji, a onda izvršavaju naredbe u telu konstruktora.

Za razliku od drugih metoda, konstruktori se mogu pozvati samo uz operator new u izrazu oblika:

```
new ImeKlase(listaArgumenata)
```

Rezultat ovog izraza je referenca na konstruisani objekat, koja se najčešće odmah dodeljuje promenljivoj klasnog tipa u naredbi dodele. Međutim, ovaj izraz se može pisati svuda gde to ima smisla, na primer kao neki argument u pozivu metoda ili kao deo nekog drugog većeg izraza. Zbog toga je važno razumeti tačan postupak izračunavanja ovog izraza, jer je njegov efekat zapravo rezultat izvršavanja redom ova četiri koraka:

1. Pronalazi se dovoljno veliki blok slobodne hip memorije za objekat navedene klase koji se konstruiše
2. Inicijalizuju se objektni atributi tog objekta. Početna vrednost nekog atributa objekta je ili ona izračunata iz deklaracije tog atributa u klasi ili podrazumevana vrednost predviđena za njegov tip
3. Poziva se konstruktor klase na uobičajen način: najpre se eventualni argumenti u pozivu konstruktora izračunavaju i dodeljuju odgovarajućim parametrima konstruktora, a zatim se izvršavaju naredbe u telu konstruktora
4. Referenca na konstruisani objekat se vraća kao rezultat izraza

Referenca na konstruisani objekat, koja je dobijena na kraju ovog postupka, može se zatim koristiti u programu za pristup atributima i metodima novog objekta.

KREIRANJE OBJEKATA POMOĆU KONSTRUKTORA (VIDEO)

Video objašnjava kako se pomoću konstruktora klasa kreiraju njeni objekti.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

PRIMER: BROJ BACANJA DVE KOCKE DOK SE NE POKAŽU ISTI BROJEVI

Klase napisane jednom mogu se iskoristiti u različitim programima u kojima je potrebna njihova funkcionalnost. Ovde je dat primer klase KockaZalgru

Jedna od prednosti objektno-orjentisanog programiranja je mogućnost višekratne upotrebe programskog koda. To znači da, recimo, klase napisane jednom mogu se iskoristiti u različitim programima u kojima je potrebna njihova funkcionalnost.

Na primer, poslednja klasa **KockaZaIgru** predstavlja fizičke kocke za igranje i obuhvata sve njihove relevantne attribute i mogućnosti. Zato se ta klasa može koristiti u svakom programu čija se logika zasniva na kockama za igranje. To je velika prednost, pogotovo za komplikovane klase, jer nije potrebno gubiti vreme na ponovno pisanje i testiranje novih klasa.

Da bismo ilustrovali ovu mogućnost, u nastavku je prikazan program koji koristi klasu **KockaZaIgru** za određivanje broja puta koliko treba baciti dve kocke pre nego što se dobije isti broj na njima

```
public class BacanjaDveKocke {  
    public static void main(String[] args) {  
        int brojBacanja = 0;  
        KockaZaIgru kocka1 = new KockaZaIgru();  
        KockaZaIgru kocka2 = new KockaZaIgru();  
        do {  
            kocka1.baci();  
            System.out.print("Na prvoj kocki je pao broj: ");  
            System.out.println(kocka1.broj);  
            kocka2.baci();  
            System.out.print("Na drugoj kocki je pao broj: ");  
            System.out.println(kocka2.broj);  
            brojBacanja++;  
        } while (kocka1.broj != kocka2.broj);  
        System.out.print("Dve kocke su bačene " + brojBacanja);  
        System.out.println(" puta pre nego što je pao isti broj.");  
    }  
}
```

UKLANJANJE OBJEKATA

U Javi, uklanjanje objekata se događa automatski i programer je oslobođen obaveze da brine o tome. Uklanja se objekat kada u programu nestane referenca na njega.

Novi objekat se konstruiše operatorom **new** i (dodatno) inicijalizuje konstruktorom klase - od tog trenutka se objekat nalazi u hip memoriji i može mu se pristupiti preko promenljivih koje sadrže referencu na njega. Ako nakon izvesnog vremena objekat više nije potreban u programu, postavlja se pitanje da li se on može ukloniti ? Odnosno, da li se radi uštede memorije može osloboditi memorija koju objekat zauzima ?

U nekim programskim jezicima sam programer mora voditi računa o uklanjanju objekata i u tim jezicima su predviđeni posebni načini kojima se eksplicitno uklanja objekat u programu.

U Javi, uklanjanje objekata se događa automatski i programer je oslobođen obaveze da brine o tome. Osnovni kriterijum na osnovu kojeg se u Javi prepoznaje da objekat nije više potreban je da ne postoji više nijedna promenljiva koja ukazuje na njega. To ima smisla, jer se takvom

objektu više ne može pristupiti u programu, što je isto kao da ne postoji, pa bespotrebno zauzima memoriju.

Da bismo ovo ilustrovali, posmatrajmo sledeći metod:

```
void novi() {
    Student s = new Student();
    s.ime = "Jovan Miletić";
    . . .
}
```

U metodu **novi()** se konstruiše objekat klase **Student** i referenca na njega se dodeljuje lokalnoj promenljivoj **s**. Nakon izvršavanja poziva metoda **novi()**, njegova lokalna promenljiva **s** se dealocira tako da više ne postoji referenca na objekat konstruisan u metodu **novi()**. Više dakle nema načina da se tom objektu pristupi u programu, pa se objekat može ukloniti i memorija koju zauzima osloboditi za druge namene. I sam programer može ukazati da neki objekat nije više potreban u programu, na primer:

```
Student s = new Student();
. . .
s = null;
. . .
```

U ovom primeru, nakon konstruisanja jednog objekta klase **Student** i njegovog korišćenja preko promenljive **s**, toj promenljivoj se eksplicitno dodeljuje referenca **null** kada taj objekat nije više potreban - time se gubi veza s tim objektom, pa se njemu više ne može pristupiti u programu.

OBJEKTI "OTPACI"

Objekti koji se nalaze u hip memoriji, ali se u programu više ne mogu koristiti jer nijedna promenljiva ne sadrži referencu na njih, popularno se nazivaju "otpaci".

Objekti koji se nalaze u hip memoriji, ali se u programu više ne mogu koristiti jer nijedna promenljiva ne sadrži referencu na njih, popularno se nazivaju "otpaci". U Javi se koristi posebna procedura sakupljanja otpadaka (engl. garbage collection) kojom se automatski s vremena na vreme "čiste otpaci", odnosno oslobađa memorija onih objekata za koje se nađe da je broj referenci na njih u programu jednak nuli.

U prethodna dva primera se može vrlo lako otkriti kada jedan objekat klase **Student** nije dostupan i kada se može ukloniti. U složenim programima je to obično mnogo teže. Ako je neki objekat bio korišćen u programu izvesno vreme, onda je moguće da više promenljivih sadrže referencu na njega. Taj objekat nije "otpadak" sve dok postoji bar jedna promenljiva koja ukazuje na njega.

Iako procedura sakupljanja otpadaka usporava izvršavanje samog programa, razlog zašto se u Javi uklanjanje objekata obavlja automatski, a ne kao u nekim jezicima ručno od strane

programera, jeste to što je vođenje računa o tome u složenijim programima vrlo teško i podložno greškama.

Prva vrsta čestih grešaka se javlja kada se nenamerno obriše neki objekat, mada i dalje postoje reference na njega. Ove greške visećih pokazivača dovode do problema pristupa nedozvoljenim delovima memorije.

Druga vrsta grešaka se javlja kada programer zanemari da ukloni nepotrebne objekte. Tada dolazi do greške curenja memorije koja se manifestuje time da program zauzima veliki deo memorije, iako je ona praktično neiskorišćena. To onda dovodi do problema nemogućnosti izvršavanja istog ili drugih programa zbog nedostatka memorije.

Do ovih grešaka ne može da se desi, jer Java automatski briše objekte za koje program više nema potrebe, tj. ne koristi više njihove reference (adrese).

PRIMER - KLASA KVADARAT - NASTAVAK 2

Podrazumevajući konstruktor inicira objekat sa podrazumevajućim vrednostima atributa

Korak 2:

Kreirati **podrazumevani (iliti prazan)** konstruktor koristeći ključnu reč `this`, tj. konstruktor koji ne prima argumente tako da postavlja podrazumevanu veličinu stranice na 1 i podrazumevane boje na vrednost "Crna"

Kreirati metodu **ispiši()**, koja ispisuje vrednosti svih atributa u sledećem formatu, sa tabulatorima:

Stranica kvadrata: 1

Boja ivice kvadrata:Crna

Boja unutrašnjosti kvadrata:Crna

Koristeći pokretačku klasu **KvadratMain** demonstrirati sve trenutne funkcionalnosti klase Kvadrat.

Programski kod klase **Kvadrat**:

```
package cs101.v07.kvadrat;
public class Kvadrat {
    private int stranica;
    private String bojaIvice;
    private String bojaUnutrasnjosti;

    public Kvadrat() {
        stranica = 1;
        bojaIvice = "Crna";
        bojaUnutrasnjosti = "Crna";
    }
    public void ispiši(){
```

```

        System.out.println("Stranica kvadrata: \t\t" + stranica);
        System.out.println("Boja ivice kvadrata: \t\t" + bojaIvice);
        System.out.println("Boja unutrašnjosti kvadrata: \t" + bojaUnutrasnjosti);
    }
}

```

Programski kod klase **KvadratMain**:

```

package cs101.v07.kvadrat;
public class KvadratMain {
    public static void main(String[] args) {
        Kvadrat k1 = new Kvadrat();
        k1.ispiši();
    }
}

```

ZADACI ZA SAMOSTALNI RAD

Na osnovu predhodnih primera uraditi sledeće zadatke

Zadatak 2.8:

Na osnovu zadatka 2.7 iz prethodnog poglavlja, u klasama CPU i Proizvodjac dodati:

1. Podrazumevani konstruktor
2. Konstruktor sa parametrima za sve attribute klase

Testirati rad konstruktora u Main klasi

ZADACI ZA SAMOSTALNI RAD - 2

Provera znanja

Zadatak 2.9 - a: Šta je pogrešno u sledećem kodu?

```

public class ShowErrors {
    public static void main(String[] args) {
        ShowErrors t = new ShowErrors(5);
    }
}

```

Zadatak 2.9 - b: Šta je pogrešno u sledećem kodu?

```

public class ShowErrors {
    public void method1() {
        Circle c;
        System.out.println("What is radius "
            + c.getRadius());
        c = new Circle();
    }
}

```



```
}
}
```

Zadatak 2.9 - c: Šta je pogrešno u sledećem kodu?

```
public class ShowErrors {
    public static void main(String[] args) {
        ShowErrors t = new ShowErrors();
        t.x();
    }
}
```

Zadatak 2.9 - d: Šta je pogrešno u sledećem kodu?

```
public class ShowErrors {
    public static void main(String[] args) {
        C c = new C(5.0);
        System.out.println(c.value);
    }
}

class C {
    int value = 2;
}
```

▼ 2.6 Učaurivanje

SKRIVENI ATRIBUTI OBJEKATA

*Svi objektni atributi klase se deklarišu i skrivaju sa modifikatorom **private**. Pristup vrednostima tih atributa iz drugih klasa je moguć samo preko javnih metoda.*

U prethodnim primerima su članovi klase uglavnom bili deklarirani s modifikatorom **public** kako bi bili dostupni iz bilo koje druge klase. Međutim, važno objektno-orjentisano načelo enkapsulacije (ili učaurivanja) nalaže da svi objektni atributi klase budu skriveni i deklarirani s modifikatorom **private**. Pritom, pristup vrednostima tih atributa iz drugih klasa treba omogućiti samo preko javnih metoda.

Jedna od prednosti ovog pristupa je to što su na taj način svi atributi (i interni metodi) klase bezbedno zaštićeni unutar "čaure" klase i mogu se menjati samo na kontrolisan način, što olakšava testiranje i pronalaženje grešaka.

Značajnija korist se ogleda u tome što se time mogu sakriti interni implementacioni detalji klase. Ako drugi programeri ne mogu koristiti te detalje, već samo elemente dobro definisanog interfejsa klase, onda se implementacija klase može lako promeniti usled novih okolnosti - to znači da je dovoljno zadržati samo iste elemente starog interfejsa u novoj

implementaciji, jer se time neće narušiti ispravnost nekog programa koji koristi prethodnu implementaciju klase.

Da bismo ovu opštu diskusiju učinili konkretnijom, razmotrimo primer jedne klase koja predstavlja profesore fakulteta, recimo, radi obračuna plata:

```
public class Profesor{
    private String ime;
    private long jmbg;
    private int staž;
    private double plata;

    public Profesor(String i, long id, int s, double p) {
        ime = i;
        jmbg = id;
        staž = s;
        plata = p;
    }

    public String getIme() {
        return ime;
    }

    public long getJmbg() {
        return jmbg;
    }

    public int getStaž() {
        return staž;
    }

    public void setStaž(int s) {
        staž = s;
    }

    public double getPlata() {
        return plata;
    }

    public void povećajPlatu(double procenat) {
        plata += plata * procenat / 100;
    }
}
. . .
```

GETER I SETER METODI

Metodi čija imena počinju sa get samo vraćaju vrednosti objektnih atributa, a metodi čija imena počinju sa set menjaju sadržaj objektnih atributa.

Obratite pažnju na to da su **svi objektni** atributi klase **Profesor** deklarirani da budu privatni - time je obezbeđeno da se oni izvan same klase Profesor ne mogu direktno koristiti. Na primer, u nekoj drugoj klasi se više ne može pisati:

```
Profesor pr1 = new Profesor("Nikola Dimitrijević", 111111, 3, 1000);
System.out.println(pr1.ime); // GREŠKA: ime je privatni atribut
pr1.staž = 10; // GREŠKA: staž je privatni atribut

. . .
```

Naravno, vrednosti atributa za konkretnog profesora se u drugim klasama programa moraju koristiti na neki način, pa su u tu svrhu definisani **javni metodi** sa imenima koja počinju rečima **get** i **set**. Ovi metodi su deo javnog interfejsa klase i zato se u drugoj klasi može pisati:

```
Profesor pr1 = new Profesor("Nikola Dimitrijević", 111111, 3, 1000);
System.out.println(pr1.getIme());
pr1.setStaž(10);

. . .
```

Metodi čija imena počinju sa **get** samo vraćaju vrednosti objektnih atributa i nazivaju se geteri ili geter metodi.

Metodi čija imena počinju sa **set** menjaju sadržaj objektnih atributa i nazivaju se seteri ili seter metodi. Postoji i termin mutator.

Nažalost, ova terminologija nije u duhu srpskog jezika, ali je uobičajena usled nedostatka makar približno dobrog prevoda - čak i da postoje bolji srpski izrazi, konvencija je da se ime geter metoda za neku promenljivu pravi dodavanjem reči "get" ispred imena promenljive napisanog velikim slovom. Tako, za promenljivu ime se dobija getIme, za promenljivu jmbg se dobija getJmbg i tako dalje.

Ime **geter metoda** za logički atribut se dozvoljava da počinje i sa is. Tako, da u klasi **Profesor** postoji atribut doktorirao tipa boolean, njegov geter metod bi se mogao zvati **isDoktorirao()** umesto **getDoktorirao()**.

Konvencija za ime seter metoda za neku promenljivu je da se ono pravi dodavanjem reči "set" ispred imena te promenljive napisanog velikim slovom. Zato je za promenljivu staž u klasi Profesor definisan seter metod **setStaž()**.

KOJA JE KORIST OD GETER I SETER METODA?

Eventualne greške u pristupu atributima se lakše nalaze pri korišćenju geter i seter metoda, a mogu se analitički pratiti pristupi atributima.

Neki aspekti naprednog Java programiranja se potpuno zasnivaju na prethodnoj konvenciji za imena getera i setera i zbog toga se preporučuje da se programeri pridržavaju konvencije za imena getera i setera, kako bi se olakšalo eventualno prilagođavanje klase naprednim

tehnikama o kojima ćemo govoriti u predmetu IT250 - Veb sistemi. Takođe, razvojno okruženje NetBeans poseduje mogućnost automatskog generisanja getera i setera u ovom formatu.

Postavljaju se pitanja:

Da li se nešto dobija deklarisanjem atributa **ime**, **jmbg**, **staž** i plata da budu privatna na račun dodatnog pisanja nekoliko naizgled nepotrebnih metoda u klasi Profesor ? Zar nije jednostavnije da su svi ovi atributi samo deklarisani da budu javni i tako bi se izbegle dodatne komplikacije ?

Prvo što se dobija za dodatno uloženi trud je lakše otkrivanje grešaka. Atributi ime i jmbg se ne mogu nikako menjati nakon početne dodele vrednosti u konstruktoru, čime je osigurano to da se u nijednom delu programa izvan klase **Profesor** ne može (slučajno ili namerno) ovim atributima dodeliti neka nekonzistentna vrednost. Vrednosti atributa staž i plata se mogu menjati, ali samo na kontrolisan način metodima **setStaž()** i

povećajPlatu(). Prema tome, ako su vrednosti tih atributa na neki način pogrešne, jedini uzročnici mogu biti ovi metodi, pa je zato veoma olakšano traženje greške - da su atributi staž i plata bili javni, onda bi se izvor greške mogao nalaziti bilo gde u programu.

Druga korist je to što geter i seter metodi nisu ograničeni samo na čitanje i upisivanje vrednosti odgovarajućih atributa. Ova mogućnost se može iskoristiti tako da se, na primer, geter metodi prate (broje) koliko puta se pristupa nekom atributu:

```
public double getPlata() {  
    plataBrojPristupa++;  
    return plata;  
}
```

GETER I SETER METODI (VIDEO)

Video objašnjava primenu seter i geter metoda za pristup podacima (atributima) u klasama.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

PROMENE UČAURENE KLAŠE I OSTALE KLAŠE

U seter metodu se može obezbediti da dodeljene vrednosti atributu budu samo one koje su smislene. Interna implementacija klase se može menjati bez posledica za ostale klase.

Slično, u seter metodu se može obezbediti da dodeljene vrednosti atributu budu samo one koje su smislene:

```
public void setStaž(int s) {  
    if (s < 0) {
```

```

        System.out.println("Greška: staž je negativan");
        System.exit(-1);
    }
    else
        staž = s;
}

```

Treća korist od skrivanja podataka je to što se može promeniti interna implementacija neke klase bez posledica na programe koji koriste tu klasu. Na primer, ako predstavljanje punog imena profesora mora da se razdvoji u dva posebna dela za ime i prezime, onda je dovoljno klasi **Profesor** dodati još jedan atribut za prezime, recimo:

```
private String prezime;
```

i promeniti getter metod **getIme()** tako da se ime profesora formira od dva dela:

```

public String getIme() {
    return ime + " " + prezime;
}

```

Ove promene su potpuno nevidljive za programe koji koriste klasu **Profesor** i ti programi se ne moraju uopšte menjati (čak ni ponovo prevoditi) da bi ispravno radili kao ranije. U opštem slučaju, geteri i seteri, pa i ostali metodi, verovatno moraju pretrpeti velike izmene radi prelaska sa stare na novu implementaciju klase. Ali, *poenta enkapsulacije je da stari programi koji koriste novu verziju klase ne moraju uopšte da se menjaju*.

PRIMER: PROGRAMSKI KOD IZMENJENE KLASA PROFESOR

Primena skrivanja atributa i njihovih setter i getter metoda izoluje njihovu implementaciju od ostalih klasa koje ih koriste.

```

public class Profesor {

    private String ime;
    private String prezime;
    private long jmbg;
    private int staž;
    private double plata;
    private static int plataBrojPristupa;

    public Profesor(String i, long id, int s, double p) {
        ime = i;
        jmbg = id;
        staž = s;
        plata = p;
    }

    public String getIme() {

```

```
        return ime + " " + prezime;
    }

    public long getJmbg() {
        return jmbg;
    }

    public int getStož() {
        return stož;
    }

    public void setStož(int s) {
        if (s < 0) {
            System.out.println("Greška: stož je negativan");
            System.exit(-1);
        }
        else
            stož = s;
    }

    public double getPlata() {
        plataBrojPristupa++;
        return plata;
    }

    public void povećajPlatu(double procenat) {
        plata += plata * procenat / 100;
    }
}
```

UČAURENJE (ENCAPSULATION) U JAVI (VIDEO)

Video objašnjava koncept učenja podataka u Java klasama

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

PRIMER - KLASA KVADRAT - NASTAVAK 3

Klasa Kvadrat sadrži, pored atributa i konstruktora, i svoje seter i geter metode.

Korak 3:

Izvršiti **enkapsulaciju** (učaurivanje) atributa kreiranjem odgovarajućih geter i seter metoda.

Kreirati **preopterećeni konstruktor** koji prima vrednosti svih atributa.

Kreirati **preopterećeni konstruktor kopije**, tj. konstruktor koji prima objekat klase **Kvadrat** kao argument.

Koristeći pokretačku klasu **KvadratMain** demonstrirati sve trenutne funkcionalnosti klase **Kvadrat**.

Ovde se prikazuje programski kod klase **Kvadrat**:

```
package cs101.v07.kvadrat;
public class Kvadrat {
    private int stranica;
    private String bojaIvice;
    private String bojaUnutrasnjosti;
    public Kvadrat() {
        this.stranica = 1;
        this.bojaIvice = "Crna";
        this.bojaUnutrasnjosti = "Crna";
    }
    public Kvadrat(int stranica, String bojaIvice, String bojaUnutrasnjosti) {
        this.stranica = stranica;
        this.bojaIvice = bojaIvice;
        this.bojaUnutrasnjosti = bojaUnutrasnjosti;
    }
    public Kvadrat(Kvadrat kvadratTemp) {
        this.stranica = kvadratTemp.stranica;
        this.bojaIvice = kvadratTemp.bojaIvice;
        this.bojaUnutrasnjosti = kvadratTemp.bojaUnutrasnjosti;
    }
    public int getStranica() {
        return stranica;
    }
    public void setStranica(int stranica) {
        this.stranica = stranica;
    }
    public String getBojaIvice() {
        return bojaIvice;
    }
    public void setBojaIvice(String bojaIvice) {
        this.bojaIvice = bojaIvice;
    }
    public String getBojaUnutrasnjosti() {
        return bojaUnutrasnjosti;
    }
    public void setBojaUnutrasnjosti(String bojaUnutrasnjosti) {
        this.bojaUnutrasnjosti = bojaUnutrasnjosti;
    }
    public void ispiši(){
        System.out.println("Stranica kvadrata: \t\t" + this.stranica);
        System.out.println("Boja ivice kvadrata: \t\t" + this.bojaIvice);
        System.out.println("Boja unutrašnjosti kvadrata: \t" +
this.bojaUnutrasnjosti);
    }
}
```

PRIMER - KLASA KVADRATMAIN - NASTAVAK 4

Klasa koja ima main program je javna.

Programski kod klase **KvadratMain**:

```
package cs101.v07.kvadrat;

public class KvadratMain {
    public static void main(String[] args) {
        Kvadrat k1 = new Kvadrat();
        Kvadrat k2 = new Kvadrat(100, "Zelena", "Bela");
        Kvadrat k3 = new Kvadrat(k2);

        k1.setBojaUnutrasnjosti("Bela");
        //k1.bojaUnutrasnjosti = "Bela"; // GREŠKA

        k1.ispiši();
        k2.ispiši();
        k3.ispiši();
    }
}
```

ZADACI ZA SAMOSTALNI RAD

Na osnovu predhodnih primera uraditi sledeće zadatke

Zadatak 2.10:

Na osnovu zadatka 2.8 iz prethodnog poglavlja, napisati getter i seter metode za sve atribute klase CPU i Proizvodjač.

Testirati rad metoda u Main klasi.

ZADACI ZA SAMOSTALNI RAD - 2

Proverite svoje znanje

Zadatak 2.11 - a: Šta je izlaz iz sledećeg programa?

```
public class Test {
    public static void main(String[] args) {
        int[] a = {1, 2};
        swap(a[0], a[1]);
        System.out.println("a[0] = " + a[0]
            + " a[1] = " + a[1]);
    }
    public static void swap(int n1, int n2) {
        int temp = n1;
```



```
        n1 = n2;
        n2 = temp;
    }
}
```

Zadatak 2.11 - c: Šta je izlaz iz sledećeg programa?

```
public class Test {
    public static void main(String[] args) {
        T t = new T();
        swap(t);
        System.out.println("e1 = " + t.e1
            + " e2 = " + t.e2);
    }
    public static void swap(T t) {
        int temp = t.e1;
        t.e1 = t.e2;
        t.e2 = temp;
    }
}

class T {
    int e1 = 1;
    int e2 = 2;
}
```

Zadatak 2.11 - b: Šta je izlaz iz sledećeg programa?

```
public class Test {
    public static void main(String[] args) {
        int[] a = {1, 2};
        swap(a);
        System.out.println("a[0] = " + a[0]
            + " a[1] = " + a[1]);
    }
    public static void swap(int[] a) {
        int temp = a[0];
        a[0] = a[1];
        a[1] = temp;
    }
}
```

Zadatak 2.11 - d : Šta je izlaz iz sledećeg programa?

```
public class Test {
    public static void main(String[] args) {
        T t1 = new T();
        T t2 = new T();
        System.out.println("t1's i = " +
            t1.i + " and j = " + t1.j);
        System.out.println("t2's i = " +
            t2.i + " and j = " + t2.j);
    }
}
```

```
    }  
}  
  
class T {  
    static int i = 0;  
    int j = 0;  
    T() {  
        i++;  
        j = 1;  
    }  
}
```

▼ 2.7 Ključna reč this

ŠTA ZNAČI THIS?

*Reč **this** označava promenljivu klasnog tipa koja u trenutku poziva metoda dobija vrednost reference na objekat za koji je metod pozvan*

Objektni metodi neke klase se primenjuju na pojedine objekte te klase i ti metodi tokom svog izvršavanja koriste konkretne vrednosti atributa onih objekata za koje su pozvani. Na primer, objektni metod **povećajPlatu()** klase **Profesor** iz prethodnog odeljka:

```
public void povećajPlatu(double procenat) {  
    plata += plata * procenat / 100;  
}
```

dodeljuje novu vrednost atributu plata koje pripada objektu za koji se ovaj metod pozove. Efekat poziva, recimo, mentor.**povećajPlatu(10)**; sastoji se u povećanju vrednosti atributu plata za 10% onog objekta na koji ukazuje promenljiva mentor. Drugim rečima, taj efekat je ekvivalentan izvršavanju naredbe dodele:

```
mentor.plata += mentor.plata * 10 / 100;
```

Poziv objektnog metoda **povećajPlatu()** sadrži dva argumenta. Prvi, implicitni, argument se nalazi ispred imena metoda i ukazuje na objekat klase **Profesor** za koji se metod poziva. Drugi, eksplicitni, argument se nalazi iza imena metoda u zagradama.

Poznato je da se parametri koji odgovaraju eksplicitnim argumentima poziva metoda navode u definiciji metoda.

S druge strane, parametar koji odgovara implicitnom argumentu se ne navodi u definiciji metoda, ali se može koristiti u svakom metodu - njegova oznaka u Javi je ključna reč **this**.

U ovom slučaju dakle, reč **this** označava promenljivu klasnog tipa koja u trenutku poziva metoda dobija vrednost reference na objekat za koji je metod pozvan. To znači da se, recimo, u metodu **povećajPlatu()** može pisati:

```
public void povećajPlatu(double procenat) {
    this.plata += this.plata * procenat / 100;
}
```

Primetimo da je ovde reč **this** uz atribut plata nepotrebna i da se podrazumeva, mada neki programeri uvek koriste ovaj stil pisanja, jer tako jasno razlikuju objektnu atributu klase od lokalnih promenljivih metoda.

KADA KORISTITI THIS?

Parametrima se upotrebom this mogu davati ista imena koja imaju atributi metoda.

Prilikom poziva konstruktora, implicitni parametar this ukazuje na objekat koji se konstruiše. Zbog toga se parametrima konstruktora često daju ista imena koja imaju objektni atributi klase, a u telu konstruktora se ti atributi pišu sa prefiksom this, na primer:

```
public Profesor(String ime, long jmbg,
int staž, double plata) {
    this.ime = ime;
    this.jmbg = jmbg;
    this.staž = staž;
    this.plata = plata;
}
```

Prednost ovog stila pisanja je to što ne moraju da se smišljaju dobra imena za parametre konstruktora kako bi se jasno ukazalo šta svaki od njih znači. Primetimo da se u telu konstruktora moraju pisati puna imena atributa sa prefiksom this, jer nema smisla pisati naredbu dodele, recimo:

```
ime = ime;
```

U stvari, to bi bilo pogrešno, jer se ime u telu konstruktora odnosi na parametar konstruktora, zato bi se naredbom:

```
ime = ime;
```

vrednost parametra ime opet dodelila njemu samom, a objektni atribut ime bi ostalo netaknuto.

U sledećem primeru promenljiva this se ona implicitno dodaje, pa je njeno isticanje više odlika ličnog stila:

```
public void povećajPlatu(double procenat) {
    plata += plata * procenat / 100;
}
```

U narednom primeru se može izbeći njena upotreba davanjem imena parametrima konstruktora koja su različita od onih koja imaju objektni atributi:

```
public Profesor(String ime1, long jmbg1,
int staž1, double plata1) {
    ime = ime1;
    jmbg = jmbg1;
    staž = staž1;
    plata = plata1;
}
```

KADA JE UPOTREBA THIS OBAVEZNA?

*Promenljiva **this** je obavezna u naredbi return this u prvoj grani if naredbe. Različiti konstruktori se mogu međusobno pozivati, ali se poziv jednog konstruktora unutar drugog piše s*

Ipak, u nekim slučajevima je promenljiva **this** neophodna i bez nje se ne može dobiti željena funkcionalnost.

Da bismo to pokazali, pretpostavimo da je klasi **Profesor** potrebno dodati objektni metod kojim se upoređuju dva profesora na osnovu njihovih plata. Tačnije, treba napisati metod **većeOd()** tako da se pozivom tog metoda u obliku, na primer, **p1.većeOd(p2)** kao rezultat dobija referenca na onaj objekat, od dva data objekta na koje ukazuju promenljive **p1** i **p2**, koji ima veću platu.

Ovaj metod mora koristiti promenljivu **this**, jer rezultat metoda može biti implicitni parametar metoda:

```
public Profesor višeOd(Profesor drugi) {
    if (this.getPlata() > drugi.getPlata())
        return this;
    else
        return drugi;
}
```

Obratite ovde pažnju na to da se u zapisu **this.getPlata()** može izostaviti promenljiva **this**, jer se bez nje poziv metoda **getPlata()** ionako odnosi na implicitni parametar metoda **većeOd()** označen promenljivom **this**.

S druge strane, promenljiva **this** jeste obavezna u naredbi return this u prvoj grani if naredbe, jer je rezultat metoda **većeOd()** u toj grani baš implicitni parametar ovog metoda.

Službena reč **this** ima još jedno, potpuno drugačije značenje od prethodnog. Naime, konstruktor klase je običan metod koji se može preopterećivati, pa je moguće imati više konstruktora sa različitim potpisom u istoj klasi. U tom slučaju se različiti konstruktori mogu međusobno pozivati, ali se poziv jednog konstruktora unutar drugog piše u obliku:

```
this(listaArgumenata);
```

Na primer, ukoliko klasi **Profesor** treba dodati još jedan konstruktor kojim se konstruiše profesor-pripravnik bez radnog staža i sa fiksnom platom, to se može uraditi na standardan način:

```
public Profesor(String ime, long jmbg) {
    this.ime = ime;
    this.jmbg = jmbg;
    this.staž = 0;
    this.plata = 100;
}
```

PRIMENA THIS U KONSTRUKTORU

*Prednost primene službene reči **this** je to što se zajedničke naredbe za konstruisanje objekata mogu pisati samo na jednom mestu u najopštijem konstruktoru.*

Ali umesto toga, novi konstruktor se može kraće pisati:

```
public Profesor(String ime, long jmbg) {
    this(ime, jmbg, 0, 100);
}
```

Ovde je zapisom:

```
this(ime, jmbg, 0, 100);
```

označen poziv prvobitnog konstruktora klase **Profesor** sa četiri parametra. Izvršavanje tog metoda sa navedenim argumentima je ekvivalentno baš onome što treba uraditi.

Prednost primene službene reči **this** u ovom kontekstu je dakle to što se zajedničke naredbe za konstruisanje objekata mogu pisati samo na jednom mestu u najopštijem konstruktoru. Onda se pozivom tog konstruktora pomoću **this** sa odgovarajućim argumentima mogu obezbediti drugi konstruktori za konstruisanje specifičnijih objekata.

Pritom treba imati u vidu i jedno ograničenje: *poziv nekog konstruktora pomoću **this** mora biti prva naredba u drugom konstruktoru*. Zato nije ispravno pisati:

```
public Profesor(String ime, long jmbg) {
    System.out.println("Profesor-pripravnik");
    this(ime, jmbg, 0, 100);
}
```

ali jeste ispravno:

```
public Profesor(String ime, long jmbg) {
    this(ime, jmbg, 0, 100);
    System.out.println("Profesor-pripravnik");
}
```

PRIMENA KLJUČNE REČI THIS U JAVI (VIDEO)

Video objašnjava primenu ključne reči u Javi

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

PRIMER - IZMENJEN KOD U KLASI PROFESOR

Primena this u klasi Profesor omogućava kraće pisanje konstruktora.

```
public class Profesor {

    private String ime;
    private String prezime;
    private long jmbg;
    private int staž;
    private double plata;
    private static int plataBrojPristupa;

    public Profesor(String i, long id, int s, double p) {
        this.ime = i;
        this.jmbg = id;
        this.staž = s;
        this.plata = p;
    }

    public Profesor(String ime, long jmbg) {
        this(ime, jmbg, 0, 100);
    }

    public String getIme() {
        return ime + " " + prezime;
    }

    public long getJmbg() {
        return jmbg;
    }

    public int getStaž() {
        return staž;
    }

    public void setStaž(int s) {
        if (s < 0) {
            System.out.println("Greška: staž je negativan");
            System.exit(-1);
        }
        else
            staž = s;
    }
}
```

```

    }

    public double getPlata() {
        plataBrojPristupa++;
        return plata;
    }

    public void povećajPlatu(double procenat) {
        this.plata += this.plata * procenat / 100;
    }

    public Profesor većeOd(Profesor drugi) {
        if (this.getPlata() > drugi.getPlata())
            return this;
        else
            return drugi;
    }
}

```

PRIMER 2 - KLASA KLINIKA

Ovaj primer ima za cilj provežbavanje kombinovanje klasa i metoda u Javi

Napisati klasu Klinika koja od atributa ima naziv klinike, adresu klinike, broj zaposlenih, broj pacijenata, tip i prosečnu platu zaposlenih. Tip klinike može biti PrivatnaKlinika ili DrzavnaKlinika.

U klasi Main kreirati proizvoljnu listu klinika. Od korisnika zahtevati da unese opciju 1 ili 2. Ukoliko unese opciju 1, prikazati kliniku sa najmanjom prosečnom zaradom. Ukoliko unese opciju 2 prikazati prikazati kliniku koja ima najveći broj pacijenata. Usuprotnom prikazati poruku o nevalidnoj opciji.

Objašnjenje: Na osnovu opisa podataka možemo zaključiti da su atributi naziv i adresa klinike stringovi, dok su broj zaposlenih, broj pacijenata tipa int. Prosečnu platu definišemo kao tip double.

Što se tiče atributa tipKlinike, kako je definisano da tip sam po sebi ima samo dve vrednosti: PrivatnaKlinika i DržavnaKlinika koristimo enum sa nazivom TipKlinike. Enum jer nam on omogućava da imamo predefinisane vrednosti kao tip podataka.

Klasa Klinika:

```

package zadatak4;

public class Klinika {

    private String naziv, adresa;
    private int brojZaposlenih, brojPacijenata;
    private double prosečnaPlata;
    private TipKlinike tip;
}

```

```
public Klinika() {  
}  
  
public Klinika(String naziv, String adresa, int brojZaposlenih, int  
brojPacijenata, double prosečnaPlata, TipKlinike tip) {  
    this.naziv = naziv;  
    this.adresa = adresa;  
    this.brojZaposlenih = brojZaposlenih;  
    this.brojPacijenata = brojPacijenata;  
    this.prosečnaPlata = prosečnaPlata;  
    this.tip = tip;  
}  
  
public String getNaziv() {  
    return naziv;  
}  
  
public void setNaziv(String naziv) {  
    this.naziv = naziv;  
}  
  
public String getAdresa() {  
    return adresa;  
}  
  
public void setAdresa(String adresa) {  
    this.adresa = adresa;  
}  
  
public int getBrojZaposlenih() {  
    return brojZaposlenih;  
}  
  
public void setBrojZaposlenih(int brojZaposlenih) {  
    this.brojZaposlenih = brojZaposlenih;  
}  
  
public int getBrojPacijenata() {  
    return brojPacijenata;  
}  
  
public void setBrojPacijenata(int brojPacijenata) {  
    this.brojPacijenata = brojPacijenata;  
}  
  
public double getProsečnaPlata() {  
    return prosečnaPlata;  
}  
  
public void setProsečnaPlata(double prosečnaPlata) {  
    this.prosečnaPlata = prosečnaPlata;  
}
```



```
public TipKlinike getTip() {
    return tip;
}

public void setTip(TipKlinike tip) {
    this.tip = tip;
}

@Override
public String toString() {
    return "Klinika{" + "naziv=" + naziv + ", adresa=" + adresa + ",
    brojZaposlenih=" + brojZaposlenih + ", brojPacijenata=" + brojPacijenata + ",
    prosecnaPlata=" + prosecnaPlata + ", tip=" + tip + '}';
}
}
```

PRIMER 2 - NASTAVAK PROGRAMSKOG KODA

Ovaj primer ima za cilj provežbavanje kombinovanje klasa i metoda u Javi i upotrebu enuma

Enum TipKlinike:

```
package zadatak4;

public enum TipKlinike {
    PRIVATNA_KLINIKA,
    DRZAVNA_KLINIKA
}
```

Main klasa:

```
package zadatak4;

import java.util.ArrayList;
import java.util.List;
import javax.swing.JOptionPane;

public class Zadatak4 {

    public static void main(String[] args) {
        // TODO code application logic here
        new Zadatak4();
    }

    public Zadatak4() {
        List<Klinika> klinike = new ArrayList<>();
        Klinika k1 = new Klinika("Klinika 1", "Adresa 1", 20, 150, 66000,
```

```

TipKlinike.PRIVATNA_KLINIKA);
    klinike.add(k1);
    Klinika k2 = new Klinika("Klinika 2", "Adresa 2", 20, 220, 76450,
TipKlinike.PRIVATNA_KLINIKA);
    klinike.add(k2);
    Klinika k3 = new Klinika("Klinika 3", "Adresa 3", 22, 800, 48500,
TipKlinike.DRZAVNA_KLINIKA);
    klinike.add(k3);
    Klinika k4 = new Klinika("Klinika 4", "Adresa 4", 28, 660, 62000,
TipKlinike.PRIVATNA_KLINIKA);
    klinike.add(k4);
    Klinika k5 = new Klinika("Klinika 5", "Adresa 5", 20, 980, 66000,
TipKlinike.DRZAVNA_KLINIKA);
    klinike.add(k5);
    String option = JOptionPane.showInputDialog(null, "Unesite opciju 1 ili 2");
    if (option.equals("1")) {
        JOptionPane.showMessageDialog(null, "Klinika sa najmanjom prosečnom
zaradom je " + najmanjaProsecnaPlata(klinike));
    } else if (option.equals("2")) {
        JOptionPane.showMessageDialog(null, "Klinika sa najviše pacijenata je "
+ najvisePacijenata(klinike));
    } else {
        JOptionPane.showMessageDialog(null, "Izabrali ste pogresnu opciju");
    }

}

public Klinika najmanjaProsecnaPlata(List<Klinika> klinike) {
    Klinika k = klinike.get(0);
    for (int i = 1; i < klinike.size(); i++) {
        if (klinike.get(i).getProsecnaPlata() < k.getProsecnaPlata()) {
            k = klinike.get(i);
        }
    }
    return k;
}

public Klinika najvisePacijenata(List<Klinika> klinike) {
    Klinika k = klinike.get(0);
    for (int i = 1; i < klinike.size(); i++) {
        if (klinike.get(i).getBrojPacijenata() > k.getBrojPacijenata()) {
            k = klinike.get(i);
        }
    }
    return k;
}
}

```

PRIMER - KLASA KVADRAT - NASTAVAK 5

Reč this se može koristiti u konstruktorima.

Korak 4:

Kreirati sledeće metode koristeći ključnu reč **this**:

- **računajPovršinu**, koja vraća površinu kvadrata
- **računajObim**, koja vraća obim kvadrata

Programski kod - klasa Kvadrat:

Dodatak u programskom kodu klase **Kvadrat**:

```
public double računajObim(){
    return this.stranica * 4;
}

public double računajPovršinu(){
    return this.stranica * this.stranica;
}
```

Programski kod **KvadratMain**:

```
package cs101.v07.kvadrat;

public class KvadratMain {
    public static void main(String[] args) {
        Kvadrat k1 = new Kvadrat();
        System.out.println(k1.računajPovršinu());
    }
}
```

PRIMER - KLASA KVADRAT - NASTAVAK 6

Automatsko povećanje broja kreiranja kvadrata pri svakom novom kreiranju kvadrata.

Korak 5:

Kreirati statički atribut **ukupanBrojKvadrata**, tipa **int**, tako da se ovaj atribut uvećava prilikom svakog kreiranja nove instance klase **Kvadrat**.

Ovde se daje isečak programskog koda klase **Kvadrat**:

```
public class Kvadrat {
    private int stranica;
    private String bojaIvice;
    private String bojaUnutrasnjosti;
    public static int ukupanBrojKvadrata = 0;

    public Kvadrat() {
        this.stranica = 1;
        this.bojaIvice = "Crna";
    }
}
```

```

        this.bojaUnutrasnjosti = "Crna";
        ukupanBrojKvadrata++;
    }

    public Kvadrat(int stranica, String bojaIvice, String bojaUnutrasnjosti) {
        this.stranica = stranica;
        this.bojaIvice = bojaIvice;
        this.bojaUnutrasnjosti = bojaUnutrasnjosti;
        ukupanBrojKvadrata++;
    }

    public Kvadrat(Kvadrat kvadratTemp) {
        this.stranica = kvadratTemp.stranica;
        this.bojaIvice = kvadratTemp.bojaIvice;
        this.bojaUnutrasnjosti = kvadratTemp.bojaUnutrasnjosti;
        ukupanBrojKvadrata++;
    }
    // Isto kao u prethodnom delu
    ...
}

```

PRIMER - KLASA KVADRAT - NASTAVAK 7

*Metod **slučajanKvadrat** vraća objekat tipa **Kvadrat** sa slučajno određenim veličinama stranice kvadrata, boje ivice i boje unutrašnjosti kvadrata.*

Kreirati statičku metodu **slučajanKvadrat()** koja vraća kvadrat sa slučajnim vrednostima veličine stranice, boje ivice i boje unutrašnjosti pri čemu treba da važi:

- veličina stranice je u opsegu [1, 10]
- boja ivice je neka od sledećih vrednosti: "Bela", "Crvena", "Zelena", "Plava", "Crna"
- boja unutrašnjosti je neka od sledećih vrednosti: "Bela", "Crvena", "Zelena", "Plava", "Crna"

Isečak programskog koda klase Kvadrat:

```

public static Kvadrat slučajanKvadrat(){
    Random r = new Random();
    String[] nizBoja = {"Bela", "Crvena", "Zelena", "Plava", "Crna"};
    int vel = 1 + r.nextInt(10);
    int indeks1 = r.nextInt(nizBoja.length);
    int indeks2 = r.nextInt(nizBoja.length);
    Kvadrat kvadratTemp = new Kvadrat(vel, nizBoja[indeks1], nizBoja[indeks2]);
    return kvadratTemp;
}

```

PRIMER - KLASA KVADRAT - NASTAVAK 8

Dodate su x i y koordinate gornjeg levog temena kvadrata, primeniti enkapsulaciju za koordinate i prilagoditi konstruktore tako da podrazumevana početna koordinata bude (0, 0).

Dodati x i y koordinate za tačku koja predstavlja početak kvadrata, tj. gornju levu koordinatu, primeniti enkapsulaciju za koordinate i prilagoditi konstruktore tako da podrazumevana početna koordinata bude (0, 0), takođe, prilagoditi metodu ispiši() tako da ispisuje vrednosti svih atributa i metodu slučajanKvadrat() tako da generiše slučajne koordinate veće ili jednake 0 i manje ili jednake 800.

Kreirati preopterećeni konstruktor koji prima sve attribute klase Kvadrat uključujući i x i y koordinatu početne tačke.

```
package cs101.v07.kvadrat;
import java.util.Random;

public class Kvadrat {
    private int xKoordinata;
    private int yKoordinata;
    private int stranica;
    private String bojaIvice;
    private String bojaUnutrasnjosti;
    public static int ukupanBrojKvadrata = 0;
    public Kvadrat() {
        this.xKoordinata = 0;
        this.yKoordinata = 0;
        this.stranica = 1;
        this.bojaIvice = "Crna";
        this.bojaUnutrasnjosti = "Crna";
        ukupanBrojKvadrata++;
    }
    public Kvadrat(int stranica, String bojaIvice, String bojaUnutrasnjosti) {
        this.xKoordinata = 0;
        this.yKoordinata = 0;
        this.stranica = stranica;
        this.bojaIvice = bojaIvice;
        this.bojaUnutrasnjosti = bojaUnutrasnjosti;
        ukupanBrojKvadrata++;
    }
    public Kvadrat(int xKoordinata, int yKoordinata, int stranica, String
bojaIvice, String bojaUnutrasnjosti) {
        this.xKoordinata = xKoordinata;
        this.yKoordinata = yKoordinata;
        this.stranica = stranica;
        this.bojaIvice = bojaIvice;
        this.bojaUnutrasnjosti = bojaUnutrasnjosti;
    }
    public Kvadrat(Kvadrat kvadratTemp) {
```

```
        this.xKoordinata = kvadratTemp.xKoordinata;
        this.yKoordinata = kvadratTemp.yKoordinata;
        this.stranica = kvadratTemp.stranica;
        this.bojaIvice = kvadratTemp.bojaIvice;
        this.bojaUnutrasnjosti = kvadratTemp.bojaUnutrasnjosti;
        ukupanBrojKvadrata++;
    }
    public int getStranica() {
        return stranica;
    }
    public void setStranica(int stranica) {
        this.stranica = stranica;
    }
    public String getBojaIvice() {
        return bojaIvice;
    }
    public void setBojaIvice(String bojaIvice) {
        this.bojaIvice = bojaIvice;
    }
    public String getBojaUnutrasnjosti() {
        return bojaUnutrasnjosti;
    }
    public void setBojaUnutrasnjosti(String bojaUnutrasnjosti) {
        this.bojaUnutrasnjosti = bojaUnutrasnjosti;
    }
    public int getXKoordinata() {
        return xKoordinata;
    }
    public void setxKoordinata(int xKoordinata) {
        this.xKoordinata = xKoordinata;
    }
    public int getyKoordinata() {
        return yKoordinata;
    }
    public void setyKoordinata(int yKoordinata) {
        this.yKoordinata = yKoordinata;
    }
    public void ispiši() {
        System.out.println("X koordinata početne tačke kvadrata: \t" +
this.xKoordinata);
        System.out.println("Y koordinata početne tačke kvadrata: \t" +
this.yKoordinata);
        System.out.println("Stranica kvadrata: \t\t\t" + this.stranica);
        System.out.println("Boja ivice kvadrata: \t\t\t" + this.bojaIvice);
        System.out.println("Boja unutrašnjosti kvadrata: \t\t" +
this.bojaUnutrasnjosti);
    }
    public double računajObim() {
        return this.stranica * 4;
    }
    public double računajPovršinu() {
        return this.stranica * this.stranica;
    }
}
```

```

    public static Kvadrat slučajanKvadrat() {
        Random r = new Random();
        String[] nizBoja = {"Bela", "Crvena", "Zelena", "Plava", "Crna"};
        int vel = 1 + r.nextInt(10);
        int xTemp = r.nextInt(801);
        int yTemp = r.nextInt(801);
        Kvadrat kvadratTemp = new Kvadrat(xTemp, yTemp, vel, nizBoja[indeks1],
nizBoja[indeks2]);
        return kvadratTemp;
    }
}

```

PRIMER - KLASA KVADRAT - KOMPLETAN PROGRAMSKI KOD

Klasa Kvadrat predstavlja klasu kojom opisujemo kvadrate.

U ovom odeljku ćemo prikazati kompletan programski kod za klase **Kvadrat** i **KvadratMain** sa dokumentacionim komentarima:

```

package cs101.v07.kvadrat;

import java.util.Random;

/**
 * Klasa Kvadrat predstavlja klasu kojom opisujemo kvadrate.
 * Klasa Kvadrat sadrži:
 * - attribute koji opisuju: X koordinatu, Y koordinatu, veličinu stranice, boju
ivice i boju unutrašnjosti, i adekvatne seter i geter metode za svaki atribut
 * - statički atribut kojim se opisuje ukupan broj kreiranih primeraka klase Kvadrat
 * - podrazumevani (prazan) konstruktor, dva konstruktora za inicijalizaciju
atributa i tzv. konstruktor kopije
 * - metode za računanje obima i površine kvadrata i ispisivanje svih atributa
 * - statičku metodu koja vraća objekat klase kvadrat sa slučajnim vrednostima
atributa
 *
 * @author Nikola
 */
public class Kvadrat {

    private int xKoordinata;
    private int yKoordinata;
    private int stranica;
    private String bojaIvice;
    private String bojaUnutrasnjosti;
    public static int ukupanBrojKvadrata = 0;

    /**
     * Podrazumevani konstruktor koji postavlja inicijalne vrednosti atributa
     */
    public Kvadrat() {
        this.xKoordinata = 0;
    }
}

```

```

        this.yKoordinata = 0;
        this.stranica = 1;
        this.bojaIvice = "Crna";
        this.bojaUnutrasnjosti = "Crna";
        ukupanBrojKvadrata++;
    }

    /**
     * Konstruktor koji postavlja veličinu stranice, boju ivice i boju unutrašnjosti
na date vrednosti, a X i Y koordinatu na vrednost 0
     * @param stranica data veličina stranice
     * @param bojaIvice data boja ivice
     * @param bojaUnutrasnjosti data boja unutrašnjosti
     */
    public Kvadrat(int stranica, String bojaIvice, String bojaUnutrasnjosti) {
        this.xKoordinata = 0;
        this.yKoordinata = 0;
        this.stranica = stranica;
        this.bojaIvice = bojaIvice;
        this.bojaUnutrasnjosti = bojaUnutrasnjosti;
        ukupanBrojKvadrata++;
    }

    /**
     * Konstruktor koji postavlja X i Y koordinatu, veličinu stranice, boju ivice i
boju unutrašnjosti na date vrednosti
     * @param xKoordinata data X koordinata
     * @param yKoordinata data Y koordinata
     * @param stranica data veličina stranice
     * @param bojaIvice data boja ivice
     * @param bojaUnutrasnjosti data boja unutrašnjosti
     */
    public Kvadrat(int xKoordinata, int yKoordinata, int stranica, String
bojaIvice, String bojaUnutrasnjosti) {
        this.xKoordinata = xKoordinata;
        this.yKoordinata = yKoordinata;
        this.stranica = stranica;
        this.bojaIvice = bojaIvice;
        this.bojaUnutrasnjosti = bojaUnutrasnjosti;
    }

    /**
     * Konstruktor kopije, tj. konstruktor koji prima objekat klase Kvadrat i
vrednosti atributa postavlja na vrednosti tog objekta
     * @param kvadratTemp dati objekat klase Kvadrat
     */
    public Kvadrat(Kvadrat kvadratTemp) {
        this.xKoordinata = kvadratTemp.xKoordinata;
        this.yKoordinata = kvadratTemp.yKoordinata;
        this.stranica = kvadratTemp.stranica;
        this.bojaIvice = kvadratTemp.bojaIvice;
        this.bojaUnutrasnjosti = kvadratTemp.bojaUnutrasnjosti;
        ukupanBrojKvadrata++;
    }

```



```
}

/**
 * Vraća veličinu stranice kvadrata
 * @return veličina stranice kvadrata
 */
public int getStranica() {
    return stranica;
}

/**
 * Vraća veličinu stranice kvadrata na datu vrednost
 * @param stranica data veličina stranice
 */
public void setStranica(int stranica) {
    this.stranica = stranica;
}

/**
 * Vraća boju ivice kvadrata
 * @return boja ivice kvadrata
 */
public String getBojaIvice() {
    return bojaIvice;
}

/**
 * Postavlja boju ivice kvadrata na datu vrednost
 * @param bojaIvice data boja ivice
 */
public void setBojaIvice(String bojaIvice) {
    this.bojaIvice = bojaIvice;
}

/**
 * Vraća boju unutrašnjosti kvadrata
 * @return boja unutrašnjosti kvadrata
 */
public String getBojaUnutrasnjosti() {
    return bojaUnutrasnjosti;
}

/**
 * Postavlja boju unutrašnjosti kvadrata na datu vrednost
 * @param bojaUnutrasnjosti data boja unutrašnjosti
 */
public void setBojaUnutrasnjosti(String bojaUnutrasnjosti) {
    this.bojaUnutrasnjosti = bojaUnutrasnjosti;
}

/**
 * Vraća X koordinatu kvadrata
 * @return X koordinata kvadrata
```

```
*/
public int getXKoordinata() {
    return xKoordinata;
}

/**
 * Postavlja X koordinatu kvadrata na datu vrednost
 * @param xKoordinata data X koordinata
 */
public void setxKoordinata(int xKoordinata) {
    this.xKoordinata = xKoordinata;
}

/**
 * Vraća Y koordinatu kvadrata
 * @return Y koordinata kvadrata
 */
public int getYKoordinata() {
    return yKoordinata;
}

/**
 * Postavlja Y koordinatu kvadrata na datu vrednost
 * @param yKoordinata data Y koordinata
 */
public void setyKoordinata(int yKoordinata) {
    this.yKoordinata = yKoordinata;
}

/**
 * Vraća ukupan broj kreiranih primeraka klase Kvadrat
 * @return ukupan broj kreiranih primeraka klase Kvadrat
 */
public static int getUkupanBrojKvadrata() {
    return ukupanBrojKvadrata;
}

/**
 * Postavlja ukupan broj kreiranih primeraka klase Kvadrat
 * @param ukupanBrojKvadrata ukupan broj kreiranih primeraka klase Kvadrat
 */
/*
public static void setUkupanBrojKvadrata(int ukupanBrojKvadrata) {
    Kvadrat.ukupanBrojKvadrata = ukupanBrojKvadrata;
}
*/

/**
 * Ispisuje vrednosti svih atributa klase Kvadrat: X i Y koordinatu, veličinu
stranice, boju ivice i boju unutrašnjosti
 */
public void ispiši() {
```

```

        System.out.println("X koordinata početne tačke kvadrata: \t" +
this.xKoordinata);
        System.out.println("Y koordinata početne tačke kvadrata: \t" +
this.yKoordinata);
        System.out.println("Stranica kvadrata: \t\t\t" + this.stranica);
        System.out.println("Boja ivice kvadrata: \t\t\t" + this.bojaIvice);
        System.out.println("Boja unutrašnjosti kvadrata: \t\t" +
this.bojaUnutrasnjosti);
    }

    /**
     * Vraća obim kvadrata
     * @return obim kvadrata
     */
    public double računajObim() {
        return this.stranica * 4;
    }

    /**
     * Vraća površinu kvadrata
     * @return površina kvadrata
     */
    public double računajPovršinu() {
        return this.stranica * this.stranica;
    }

    /**
     * Vraća objekat klase Kvadrat sa slučajnim vrednostima atributa
     * @return objekat klase Kvadrat sa slučajnim vrednostima atributa
     */
    public static Kvadrat slučajanKvadrat() {
        Random r = new Random();
        String[] nizBoja = {"Bela", "Crvena", "Zelena", "Plava", "Crna"};
        int vel = 1 + r.nextInt(10);
        int indeks1 = r.nextInt(nizBoja.length);
        int indeks2 = r.nextInt(nizBoja.length);
        int xTemp = r.nextInt(801);
        int yTemp = r.nextInt(801);
        Kvadrat kvadratTemp = new Kvadrat(xTemp, yTemp, vel, nizBoja[indeks1],
nizBoja[indeks2]);
        return kvadratTemp;
    }
}
package cs101.v07.kvadrat;

public class KvadratMain {
    public static void main(String[] args) {
        // Demonstracija korišćenja preopterećenih konstruktora
        Kvadrat k1 = new Kvadrat();
        Kvadrat k2 = new Kvadrat(10, "Bela", "Bela");
        Kvadrat k3 = new Kvadrat(50, 50, 10, "Crna", "Plava");
        Kvadrat k4 = new Kvadrat(k1);
        // Demonstracija korišćenja statičke metode slučajanKvadrat
    }
}

```

```
Kvadrat slucajanKvadrat = Kvadrat.slucajanKvadrat();

// Demonstracija korišćenja setter metode za boju ivice
System.out.println("Boja ivice objekta k1 klase Kvadrat je: " +
k1.getBojaIvice());

// Demonstracija korišćenja metoda za računanje obima i površine
k3.računajObim();
k3.računajPovršinu();

// Demonstracija korišćenja metode za ispisivanje vrednosti atributa
System.out.println("Vrednosti atributa slucajnog kvadrata su: ");
slucajanKvadrat.ispiši();
}
}
```

ZADACI ZA SAMOSTALNI RAD

Na osnovu prethodnih primera uraditi sledeće zadatke

Zadatak 2.12:

Na osnovu zadatka 2.10 iz prethodnog poglavlja, u klasama CPU i doraditi konstruktore, kao i getter i setter metode dodavanjem ključne reči **this**.

U Main klasi testirati rad dorađenih klasa.

ZADACI ZA SAMOSTALNI RAD - 2

Proverite svoje znanje

Zadatak 2.13:

Šta je pogrešno u sledećem kodu?

```
public class Test {
    private int id;
    public void m1() {
        this.id = 45;
    }
    public void m2() {
        Test.id = 45;
    }
}
```

Zadatak 2.14:

Šta je pogrešno u sledećem kodu?

```
public class C {
    private int p;
```

```
public C() {  
    System.out.println("C's no-arg"  
        + " constructor invoked");  
    this(0);  
}  
  
public C(int p) {  
    p = p;  
}  
  
public void setP(int p) {  
    p = p;  
}  
}
```

▼ Poglavlje 3

Primena klasa iz Javine biblioteke klasa

JAVINA BIBLIOTEKA KLASA

Java platforma obezbeđuje koristan i kompletan skup standardnih Java biblioteka klasa.

Javina biblioteka klasa ([Java Class Library](#) - JCL) predstavlja skup biblioteka koje dinamički mogu biti učitane tokom vremena izvršavanja programa. Iz razloga što je Java platformski nezavisan programski jezik, Java programi ne mogu da se oslanjaju na prirodne biblioteke platformi na kojima se izvršavaju. Umesto toga, Java platforma obezbeđuje koristan i kompletan skup standardnih Java biblioteka klasa koje sadrže funkcije koje su opšte za savremene operativne sisteme.

JCL obezbeđuje tri glavne funkcionalnosti Java platforme:

- Obezbeđuje skup dobro poznatih korisnih alata, poput kontejnerskih klasa i procesiranja regularnih izraza;
- Obezbeđuje apstraktni interfejs za zadatke koji bi, u normalnim okolnostima, čvrsto bili povezani sa hardverom i operativnim sistemom, na primer pristup mreži i datotekama;
- Neke platforme ne daju punu podršku alatima neophodnim za razvoj Java programa. U tom slučaju, implementacija neke biblioteke može da emulira ove alate ili da obezbedi konzistentan način provere prisustva specifičnog alata.

Javina biblioteka klasa sadrži skup predefinisanih klasa, napisanih Java jezikom, koje pokrivaju širok spektar problema. Sledećim linkom je moguće detaljno sagledati brojne klase iz JCL: <https://docs.oracle.com/javase/7/docs/api/>.

Sledećom slikom su prikazane neke od često korišćenih standardnih Java klasa.

Java Class Libraries

- I/O
- Networking
- Math
- Collections
- Regular Expressions
- Logging
- Graphics and UI
- Text Formatting
- XML
- Remote Method Invocation
- Security
- Databases
- Reflection
- Brojne druge...

Slika 3.1 Često korišćene standardne Java klase

PRIMER 1 - PRIMENA JCL KLASE

*Instrukcijom **import**, iza koje se navodi pun naziv klase, JCL klasa se čini dostupnom za konkretan program.*

Da bi Java program, koji se kreira, mogao da koristi klasu iz Javine biblioteke klasa, neophodno je datu klasu, zajedno sa paketom kojem pripada, uključiti u program. Instrukcijom **import**, iza koje se navodi pun naziv klase (uključuje naziv paketa i klasu), JCL klasa se čini dostupnom za konkretan program.

Sledećom slikom je prikazano korišćenje **import** instrukcija u jednostavnom Java primeru.

```
1  /*
2   * To change this license header, choose License Headers in Project Properties.
3   * To change this template file, choose Tools | Templates
4   * and open the template in the editor.
5   */
6  package jclprimer;
7
8  import java.util.Calendar;
9
10 /*
11  *
12  * @author Vladimir Milicevic
13  */
14 public class JCLPrimer {
15
16     /**
17      * @param args the command line arguments
18      */
19     public static void main(String[] args) {
20         System.out.println ("Trenutno vreme je:" + Calendar.getInstance().getTime());
21     }
22 }
23
```

Slika 3.2 Primena import instrukcije

Iz prikazanog listinga se primećuje da je zadatak kreiranog programa prikazivanje trenutnog vremena. Ovaj zadatak obavlja objekat klase **Calendar** (linija koda broj 20) koja pripada standardnoj Javinoj biblioteci klasa. Da bi program mogao da koristi ovu klasu, neophodno je dodati liniju koda broj 8:

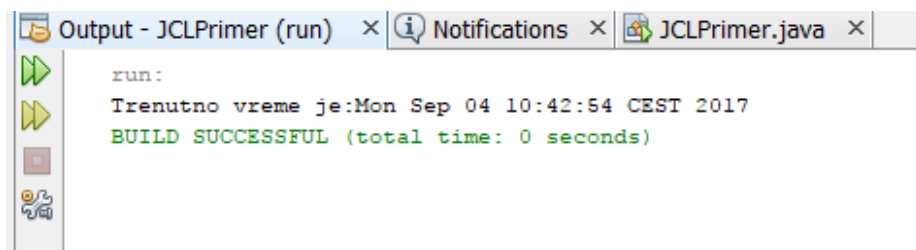
```
import java.util.Calendar;
```

Ovom linijom koda je omogućeno korišćenje ove klase u programu. Važno je napomenuti da se klasa navodi sa punim nazivom, zajedno sa paketom kojem pripada (u ovom slučaju to je paket **java.util**).

Ukoliko se koristi više klasa iz istog paketa, moguće je koristiti džoker znak "*" koji zamenjuje sve klase iz datog paketa, na primer:

```
import java.util.*;
```

Sledećom slikom je prikazan rezultat pokretanja kreiranog programa.



Slika 3.3 Primena klase iz Java biblioteke klasa

PRIMER 2

Primena JCL - ArrayList

Kreirati klasu Student koja od atributa ima ime, prezime, broj indeksa, datum rođenja, godinu koju je poslednju upisao na fakultetu kaolistu koji broje maksimalno 32 ocene. Ime i prezime treba biti tipa String, broj indeksa, godina na fakultetu i niz ocena treba biti tipa int, a datum rođenja može biti tipa String ili Date. Za sve attribute kreirati prazan i preopterećen konstruktor, getere, setere kao i toString metodu. Implementirati i sledeće metode:

- Metodu studentBrukos() koja vraća true ukoliko je student brucosh i false ukoliko nije
- Metodu dodajPolozenIsplit() koja kao argument prima ocenu i dodaje je u listu/niz ocena studenta.
- Metodu racunajProsek() koja računa i vraća prosečnu ocenu studenta zaokruženu na dve decimale.

U klasi Main kreirati proizvoljan niz studenata i za svakog prikazati da li je brucosh ili ukoliko nije njegovu prosečnu ocenu.

Klasa Student:

```
package zadatak2;  
  
import java.util.ArrayList;  
import java.util.List;  
import javax.swing.JOptionPane;
```



```
public class Student {

    private String ime, prezime, datumRodjenja;
    private int BrIndeksa, godina;
    private List<Integer> ocene = new ArrayList<>();

    public Student() {
    }

    public Student(String ime, String prezime, String datumRodjenja,
int BrIndeksa, int godina) {
        this.ime = ime;
        this.prezime = prezime;
        this.datumRodjenja = datumRodjenja;
        this.BrIndeksa = BrIndeksa;
        this.godina = godina;
    }

    public String getIme() {
        return ime;
    }

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

    public String getPrezime() {
        return prezime;
    }

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

    public String getDatumRodjenja() {
        return datumRodjenja;
    }

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

    public int getBrIndeksa() {
        return BrIndeksa;
    }

    public void setBrIndeksa(int BrIndeksa) {
        this.BrIndeksa = BrIndeksa;
    }

    public int getGodina() {
        return godina;
    }
}
```

```
}

public void setGodina(int godina) {
    this.godina = godina;
}

public List<Integer> getOcene() {
    return ocene;
}

public void setOcene(List<Integer> ocene) {
    this.ocene = ocene;
}

public void dodajPolozenIspit(int ocena) {
    if (ocene.size() < 32) {
        if (ocena < 5 || ocena > 10) {
            JOptionPane.showMessageDialog(null,
                "Ocena nije validna!");
        } else {
            ocene.add(ocena);
        }
    } else {
        JOptionPane.showMessageDialog(null,
            "Lista ocena sadrzi maksimalan broj elemenata!");
    }
}

public double racunajProsek() {
    int sum = 0;
    for (int ocena : ocene) {
        sum += ocena;
    }

    return (double) sum / ocene.size();
}

public boolean studentBrucos() {
    return (this.getGodina() == 1);
}

@Override
public String toString() {
    return "Student{" + "ime=" + ime + ", prezime="
        + prezime + ", datumRodjenja=" + datumRodjenja
        + ", BrIndeksa=" + BrIndeksa + ", godina=" + godina
        + ", ocene=" + ocene + '}';
}
}
```

PRIMER 2- NASTAVAK

Primena JCL - ArrayList 2

U klasi Main kreirati proizvoljan niz studenata i za svakog prikazati da li je brucos̃ ili ukoliko nije njegovu prosečnu ocenu.

Main Klasa:

```
package zadatak2;

import java.util.ArrayList;
import java.util.List;
import javax.swing.JOptionPane;

public class Zadatak2 {

    public static void main(String[] args) {
        // TODO code application logic here
        new Zadatak2();
    }

    public Zadatak2() {
        List<Student> studenti = new ArrayList<>();
        Student s1 = new Student("Marko", "Markovic", "01-01-1990", 123, 1);
        s1.dodajPolozenIspit(10);
        s1.dodajPolozenIspit(9);
        s1.dodajPolozenIspit(9);
        s1.dodajPolozenIspit(7);
        studenti.add(s1);

        Student s2 = new Student("Marija", "Petrovic", "01-11-1992", 1234, 2);
        s2.dodajPolozenIspit(10);
        s2.dodajPolozenIspit(8);
        s2.dodajPolozenIspit(8);
        s2.dodajPolozenIspit(7);
        studenti.add(s2);

        Student s3 = new Student("Marko", "Markovic", "01-01-1990", 12345, 1);
        s3.dodajPolozenIspit(10);
        s3.dodajPolozenIspit(10);
        s3.dodajPolozenIspit(6);
        s3.dodajPolozenIspit(7);
        studenti.add(s3);

        Student s4 = new Student("Marko", "Markovic", "01-01-1990", 123456, 3);
        s4.dodajPolozenIspit(9);
        s4.dodajPolozenIspit(6);
        s4.dodajPolozenIspit(8);
        s4.dodajPolozenIspit(7);
        studenti.add(s4);
    }
}
```

```

        for (Student s : studenti) {
            if (s.studentBrucos()) {
                System.out.println("Student je brucos");
            } else {
                System.out.println("Prosecna ocena je " + s.racunajProsek());
            }
        }
    }
}

```

ZADACI ZA SAMOSTALNI RAD

Na osnovu predhodnih primera uraditi sledeće zadatke

Zadatak 3.1:

Na osnovu klase CPU, u Main klasi prikazati rad ArrayList-a.
Kreirati 10 objekata klase CPU i sačuvati ih u listi na nazivom **procesori**.

Zadatak 3.2:

Na osnovu klase CPU, u Main klasi prikazati rad ArrayList-a.
Kreirati 5 objekata klase Proizvođač i sačuvati ih u listi na nazivom **proizvođači**.

Zadatak 3.3 (*):

Stack je vrsta memorije u kojoj se smeštaju određene vrednosti. Treba napraviti simulaciju Stack-a putem Java koji koristi LIFO metod (Last in – First out). Treba napraviti klasu Stack i klasu Element. Element od atributa ima vrednost. Klasa Stack treba da ima listu Elemenata i metodu push koja dodaje element na listu. Pored push metode treba da sadrži pop metodu koja daje poslednji element te liste, a isti element samim tim izbacuje iz liste i metodu peek koja daje poslednji element sa liste, ali ga ne uklanja sa iste.. Pored ovih metoda treba još napraviti i metodu getStackSize koja vraća trenutnu veličinu Stack liste. Prikazati rad Stack-a preko Main-a.

▼ Poglavlje 4

Uvod u klasni model Power Designer

INSTALACIJA POWER DESIGNER ALATA

Kratke informacije koje se odnose na instalaciju Power Designer alata

Power Designer razvojni alat možete preuzeti sa sledećeg linka

ftp://ftp.metropolitan.ac.rs/studenti%20install/PowerDesigner161_Evaluation.exe.

Ukoliko niste u mogućnosti da preuzmete sa sajta program, možete isti tražiti takođe od IT službe ili asistenta.

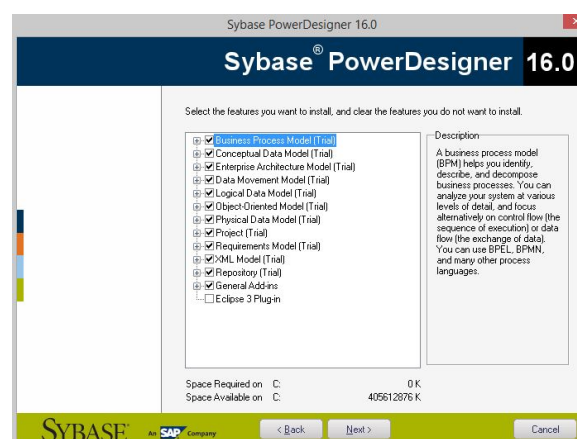
Potrebno je pokrenuti instalaciju i prilikom instalacije odabrati sve tipove dijagrama kao što je prikazano na slici 1 . Ukoliko ste uspešno instalirali Power Designer razvojni alat možete ga pokrenuti i dobićete prikaz kao na slici 2 .

Ukoliko želite registrovati licencu potrebno je da uradite sledeće:

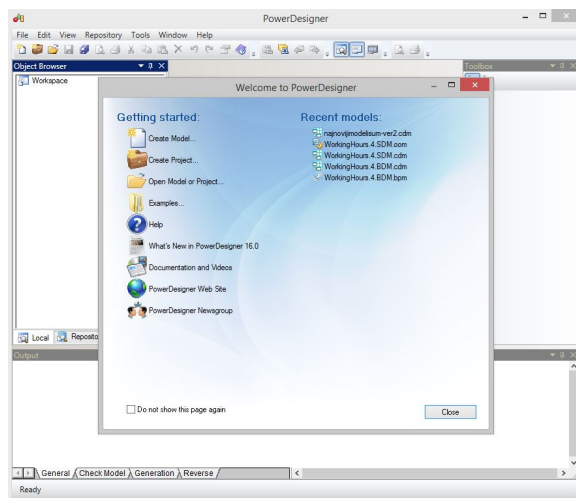
1. Nakon jednostavne instalacije, potrebno je odabrati iz menija **Tools -> Licence parameters** opciju **Floating licence**
2. U polje **Computer name of the licence server** uneti: **sybase.metropolitan.edu.rs,**
3. U polje **TCP port number** treba uneti: **27000**
4. Nakon toga, kliknuti na dugme **Next**, a zatim na dugme **Finish**, čime dobijate licencu za korišćenje PowerDesigner-a.

Uputstvo za registrovanje lincence takođe možete pogledati na sledećem linku:

<http://sybase.metropolitan.ac.rs/>



Slika 4.1 Odabir modela



Slika 4.2 Završena instalacija

UML KLASNI DIJAGRAM

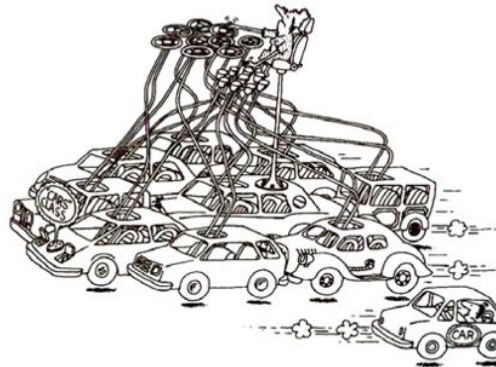
Uvod i upotreba UML klasnog dijagrama, objekata, klasa i veza

Koncept objekta i klase se međusobno stalno prepliću, tako da ne možemo govoriti o nekom objektu, a da ne pomenemo njegovu klasu. Između ova dva termina ipak, postoje bitne razlike. Dok objekat predstavlja konkretan entitet, koji postoji u vremenu i prostoru, klasa je samo apstrakcija, suština objekta. Grady Booch daje sledeću definiciju klase: "Klasa je skup objekata koji imaju zajedničku strukturu i ponašanje"

Na slici 3 prikazani su automobili i njihova zajednička svojstva:

1. Kreću se na četiri točka
2. Njima vozač upravlja preko volana, i papučicama za gas, za kvačilo i za kočnicu;
3. Svi imaju menjač i brzine

Te zajedničke karakteristike svih automobila (kao objekata) svrstavaju ih u isti tip objekata, tj. u klasu "automobil". Jedan objekat predstavlja primerak neke klase. Pogledajte primerak neke knjige; taj primerak ima svoje osobine: može biti sasvim nov ili pohaban; možda ste na prvoj stranici napisali svoje ime; možda pripada biblioteci, pa ima svoju jedinstvenu šifru. Sa druge strane, osnovni elementi knjige, kao što su: naslov, izdavač, autor i sadržaj su određeni opisom koji je primenljiv na svaki zaseban primerak: naslov knjige je taj i taj; izdavač je taj i taj; itd. Taj skup osobina ne definiše objekat, već klasu objekata (ponekad se koristi i termin *tip objekata*).



Slika 4.3 Zajednička svojstva

UML SIMBOL KLASA

Prikaz simbola klase i njegova značenja

Da bismo lakše opisali klase i objekte, kao i njihove veze koji svi zajedno čine OO modele, korist ćemo njihove grafičke prikaze u skladu sa tzv. UML jezikom za OO modeliranje (UML - Unified Programming Language).

UML klasni dijagram ili dijagram klasa prikazuje skup klasa, interfejsa i njihovih relacija odnosno opisuje strukturu sistema. Klase predstavljaju abstrakciju koja određuje zajedničku strukturu i ponašanje niza objekata. Objekti su instance klase, koje se stvore, modifikuju svoje stanje i uništavaju tokom izvođenja sistema. Svaki objekat ima stanje koje obuhvata vrednost atributa i njenih veza sa drugim objektima.

Svaka klasa sastoji se od imena, atributa i metoda(operacija). Na slici 2 prikazan je primer klase korišćenjem PowerDesigner razvojnog alata.

Kao što se može primetiti na slici jedan klasa kupac ima tri celine. Prva celina predstavlja naziv same klase (u primeru datom na slici 4 naziv klase je Kupac), drugi deo predstavlja mesto za deklarisanje svih promenljivih odnosno atributa klase i njihovih tipova podataka. U primeru datom na slici klasa Kupac ima tri atributa (ime, adresa, datumRodjenja) i oni pre početka naziva atributa stoje odgovarajući simboli koji definišu vidljivost(pristup) samog atributa. Treća sekcija predstavlja definisanje metoda odnosno operacija (funkcija). Pre naziva metode i dalje stoje simboli za enkapsulaciju koji određuju vidljivost metode, a nakon imena same metode posle dve tačke stoji tip podatka koji ta metoda vraća. U primeru datom na slici 4 klasa Kupac ima jednu metodu izracunavanjeKredita() koja je public i koja vraća double tip podataka.



Slika 4.4 Prikaz UML klase kupac

Da bi se ograničilo pravo pristupa nekom atributu ili metodu nekog objekta, koriste se tri stepena zaštićenosti, tj. kontrole pristupa:

public (+) – javan, dozvoljen pristup sa drugih objekata;

private (-) – privatan, dozvoljen pristup samo iz te klase;

protected (#) – zaštićen, dozvoljen pristup i iz podklasa ove klase

Kontrola pristupa definiše vidljivost klase, promenljive, ili metoda – od strane drugih objekata.

VEZE IZMEĐU OBJEKATA(KLASA)

Tipovi veza na klasnom dijagramu

Objektno-orijentisani model se definiše objektima i vezama koje postoje između objekata. Kao što se može primetiti sa prethodnog primera korišćeni su neki od tipova veza (generalizacija i asocijacija). U nastavku teksta govoriće se detaljnije o vezama objekata.

Postoji pet elementarnih tipova veza (ili relacija) između objekata:

1. Asocijacija – jedan objekat upotrebljava servise drugog;
2. Agregacija – jedan objekat sadrži u sebi drugi objekat;
3. Kompozicija – vlasnik je odgovoran za kreiranje ili uništavanje delova objekata;
4. Generalizacija – podklasa ima sve karakteristike klase;
5. Usavršavanje – dodavanje prethodno nespecificiranih aspekata nekompletno specificiranom entitetu.

Zavisnost



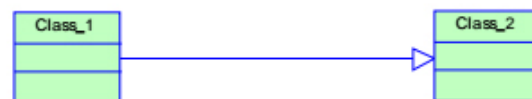
Asocijacija



Agregacija



Generalizacija



Realizacija



Slika 4.5 Generalni prikaz veza među klasama

ASOCIJACIJA

Objašnjenje veze asocijacije na klasnom dijagramu

Kod veze tipa asocijacije jedan objekat upotrebljava servise drugog, ali ga ne poseduje. Asocijacije su relacije koje se manifestuju u vreme izvršavanja (realizacije) time što dozvoljavaju razmenu poruka među objektima. Upotrebljavaju se kada:

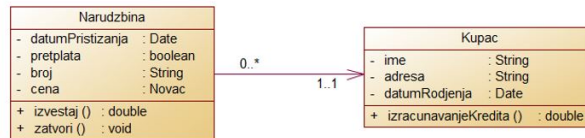
- Jedan objekat upotrebljava servise drugog, ali on nije sadržan u njemu.
- Životni ciklus upotrebene klase ne zavisi od klase koja je koristi.
- Asocijacija među objektima je labavija od agregacije.
- Asocijacija je veza tipa klijent-server.
- Upotrebljen objekat se deli podjednako sa puno drugih objekata.

Na slici je prikazana asocijacija dva objekta pomoću UML jezika. Asocijacija ima svoj naziv koji se upisuje na sredini linije, a uz objekte se upisuju dva podatka:

- uloga koju objekat igra u odnosu na objekat sa kojim je povezan

- multiplikator (ili kardinalost) koji pokazuje broj mogućih objekata koji mogu biti u vezi.

Objekti šalju informaciju drugim objektima slanjem parametara. Asocijacije označavaju veze objekata koji razmenjuju poruke. Kada se šalje parametar iz objekta A, poziva se metod objekta B koji prima parametar. Definicija klase treba da odredi da takva veza postoji i koji tip informacije se može preneti.



Slika 4.6 PD primer asocijativnosti



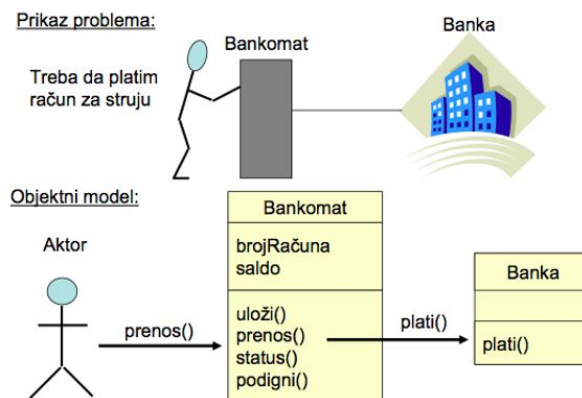
Slika 4.7 Uopšteni primer veze asocijacije

PRIMER VEZE ASOCIJACIJE

Primena veze asocijacije na primeru

Objekti šalju informaciju drugim objektima slanjem parametara. Asocijacije označavaju veze objekata koji razmenjuju poruke. Kada se šalje parametar iz objekta A, poziva se metod objekta B koji prima parametar. Definicija klase treba da odredi da takva veza postoji, i koji tip informacije se može preneti.

Objekti i njihove veze opisuju situacije u realnom životu koje želimo da modeliramo, da bismo kasnije sačinili i računarski program. Na slici 8 prikazan je jedan takav primer. Jedna osoba želi da preko bankomata (ATM mašine) plati račun za struju. Ona mora da izabere opciju na meniju bankomata koja joj dozvoljava da prenese određenu sumu novca sa svog računa – na račun Elektrodistribucije, u skladu sa računom za struju koji je dobila. Ovde imamo tri entiteta: Čovek, Bankomat i Banka. Meni bankomata aktivira metod prenos() objekta Bankomat, a ovaj poziva metod plati() objekta Banka, koji prebacuje novac sa jednog na drugi račun. Prema tome, pozivanjem metoda ovih objekata, podstaklo se njihovo ponašanje, tj. oni su uradili ono što se od njih tražilo. Prvi je poslao poruku drugom da prebaci novac sa jednog na drugi račun. Ova veza između dva objekta je veza tipa asocijacija, jer se njome vrši slanje poruka. Svaka poruka podstiče (poziva, aktivira) neki metod u objektu koji prima tu poruku, čime se on aktivira da se ponaša onako kako je definisano metodom koji je porukom podstaknut, tj. aktiviran.



Slika 4.8 Primer asocijacije

AGREGACIJA I KOMPOZICIJA

Primena veze agregacije i kompozicije na UML klasnom dijagramu

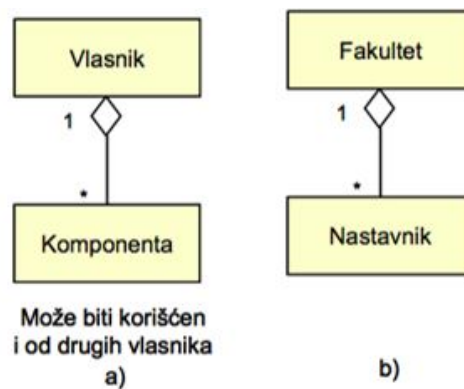
Agregacija

Agregacija se primenjuje kada jedan objekat fizički ili konceptijski sadrži drugi.

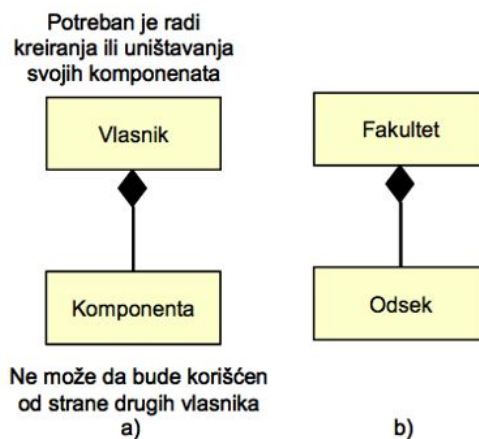
Na slici 9 a prikazan je UML model veze tipa agregacija između dva objekta. Objekat-vlasnik sadrži objekat-komponentu, ali objekat-komponentu mogu istovremeno da koriste i druge klase, tj. objekat-komponenta može da bude i u vlasništvu više različitih klasa-vlasnika. Na slici 9 b prikazan je primer ovakve veze. Na fakultetetu radi više nastavnika. Kako veza tipa agregacije dozvoljava da objekat – komponenta ima odnose i sa drugim klasama, to se jasno vidi iz ovog dijagrama da nastavnik može da radi i na nekom drugom fakultetu, tj. da deli svoje radno vreme na dva ili više fakulteta, odnosno, da ima više poslodavaca.

Kompozicija

Kompozicija je jak oblik agregacije. Upotrebljava se za aktivne objekte – objekte koji su izvori upravljanja. Ona kreira proces upravljanja, i njene komponente izvršavaju taj proces. Veza tipa agregacije se u UML obeležava *izokrenutim rombom*, koji je popunjen crnom bojom (slika 10). Opet, kao primer, navodimo fakultet. Fakultet može da ima više odseka, ali za razliku od nastavnika, ti odseci ne mogu da budu i delovi drugih fakulteta ili organizacija. Zato je ovde korišćen jači oblik agregacije, a to je kompozicija. Ona ne dozvoljava komponenti da bude deo nekog drugog objekta ili da ima veze tipa agregacije sa drugim objektima.



Slika 4.9 Demonstracija agregacije



Slika 4.10 Demonstracija kompozicije

GENERALIZACIJA I NASLEĐIVANJE

Šta je generalizacija, čemu služi i kako se primenjuje

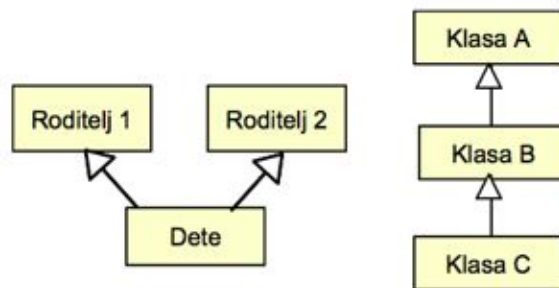
Dve klase su povezane tako što je jedna klasa-roditelj ili **nadklasa**, a druga je klasa-dete ili **podklasa**. Klasa-roditelj ima sva svojstva svojih klasa- dece, tj. klasa-dete nasleđuje sva svojstva (atribute i metode) klase- roditelja. Klasa-dete može da sadrži dodatne atribute i ponašanja (metode). U UML-u, ovakva veza se označava linijom sa trouglom na mestu strelice, kojim se označava klasa-roditelj kao što je prikazano na slici.

Nasleđivanje je ono što razlikuje objektno-orijentisane jezike od jezika koji su samo “bazirani na objektima”. Koristi se radi modeliranja odnosa koji postoje između različitih, ali vrlo sličnih klasa, koji na ovaj način postaju delovi iste familije. Nasleđivanje je veza tipa “kao što je”.

Klasa (ili neko njeno svojstvo) je vidljiva, kada su njena svojstva pristupačna nekoj spoljnoj klasi. Nasleđivanje kod klasa otvara još jednu mogućnost kontrole pristupa. Pristup se dozvoljava unutar klase i svim podklasama, ali ne i spoljnim klasama. To je slučaj tzv. zaštićen pristup (#, tj. protected).

Postoji i mogućnost tzv. višestrukog nasleđivanja (slika 11). To je slučaj kada jedna klasa ima više klasa-roditelja (nadklasa). Tada ona nasleđuje svojstva svojih klasa-roditelj, te sa njima

deli svojstva. Ponovljeno nasleđivanje je slučaj kada jedna klasa ima više klasa-roditelj, ali oni su u istoj liniji.



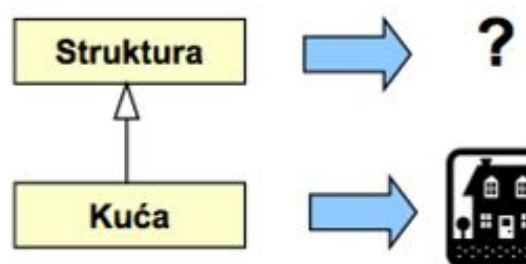
Slika 4.11 Prikaz višestrukog nasleđivanja

APSTRAKTNE KLASSE

Pojam apstrakcije i apstraktnih klasa i njihova primena

Apstraktne klase definišu vrlo široku lepezu klasa entiteta. Tako postavljen opšti tip klase se koristi kao superklasa koja sadrži svojstva svih svojih podklasa, koje imaju i svoja specifična svojstva – karakteristična za objekte koje definišu. Zbog svoje velike uopštenosti (generalisanja) apstraktna klasa nema svoje primerke u svetu objekata, tj. nema svoju instancu, objekat. Apstraktna klasa predstavlja neki koncept, a ne objekte. Na primer, mnogi različiti objekti mogu imati strukturu (velike i male zgrade, antene, statue i drugo). Struktura je koncept i ne može se predstaviti nekim objektom (slika 12).

Drugi primer apstraktne klase prikazan je na slici 13 Geometrijske figure sa nazivima trougla, pravougaonika i kruga imaju izvesna zajednička svojstva. Da se ne bi stalno prepisivala, ona se u takvim slučajevima, definišu u jednoj superklasi koju smo najavili – Oblik, tako da ih mehanizmom nasleđivanja dobijaju sva tri ova elementa. Međutim, postavlja se pitanje, kako izgleda taj geometrijski oblik koji predstavlja *klasa Oblik*? Odgovor je – ne izgleda nikako! Takav opšti oblik u stvarnosti ne postoji, te je očigledno da ta superklasa ne može da ima svoju instancu, tj. iz nje se ne može dobiti oblik. Zato je ona apstraktna klasa. Ona nema mogućnost da ima svoje instance (objekte). Ona samo služi za deljenje zajedničkih svojstava između njenih podklasa, i sama za sebe – ne predstavlja nikakav realan objekat.



Slika 4.12 Prikaz apstrakcije



Slika 4.13 Prikaz apstraktne klase

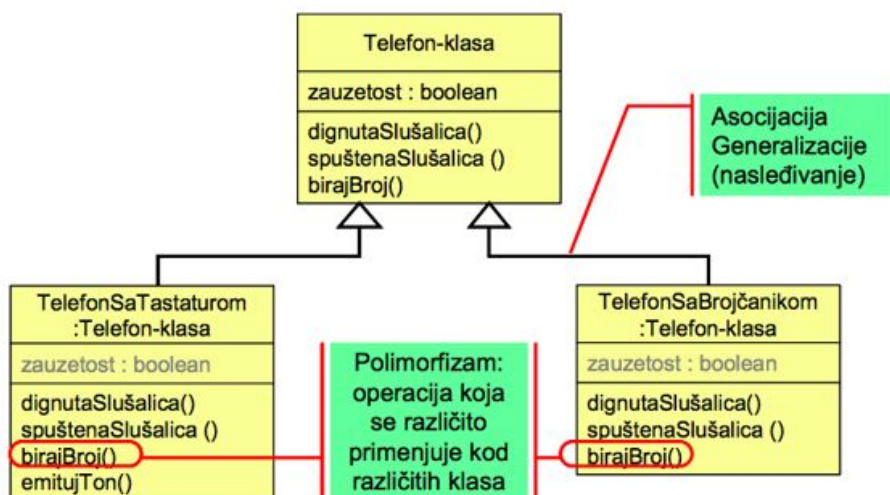
POLIMORFIZAM

Razjašnjenje pojma polimorfizma i njegovog nastanka

Polimorfizam (grčka reč) označava “mnoge forme”. U OO modeliranju označava tehniku korišćenja različitih klasa iz iste hijerarhije klasa (familije) koju može da koristi bilo koji član familije klasa. Poliformizam dozvoljava programeru da koristi bilo koju podklasu na mestu u kojem se zahteva korišćenje superklase. Suprotno, (da superklasa zameni podklasu) nije moguće.

Na slici 14 dat je primer polimorfizma. Dve klase: TelefonSaBrojčanicom i TelefonSaTastaturom imaju zajedničku superklasu koja sadrži njihova zajednička svojstva. Da bi bilo jasnije, nasleđeni metodi super klase koja se naziva Telefon-klasa, su navedeni i u podklasama, što nije bilo neophodno, jer se njihovo prisustvo podrazumeva. Metod birajBroj() po ovom mehanizmu nasleđivanja nalazi se u obe podklase. Međutim, implementacija tog metoda je različita. On podržava unos brojeva preko tatsature kod telefona sa tastaturom, a kod telefona sa brojčanicom – podržava unos brojeva preko brojčanika koji se okreće. Iako imaju isto ime, ova dva metoda se razlikuju u programskoj implementaciji. Iako se aktiviraju na isti način, pozivom metoda birajBroj(), ova dva objekta različito reaguju.

U jednom slučaju se aktivira program koji radi sa tastaturom, a u drugom – sa okretnim brojčanicom. To je primer polimorfizma. Neki metod se poziva na identičan način, a on izaziva različita ponašanja u različitim objektima. Slično je i sa metodom print() koja se primenjuje kod štampača. Postoje različiti štampači. Ista komanda, zbog poliformizma, koji dozvoljava objektno- oriujentisan model, aktivira štampanje dokumenata na različitim računarima. Za svaki od njih, naravno, prethodno je napravljena drugačija implementacija metoda print().

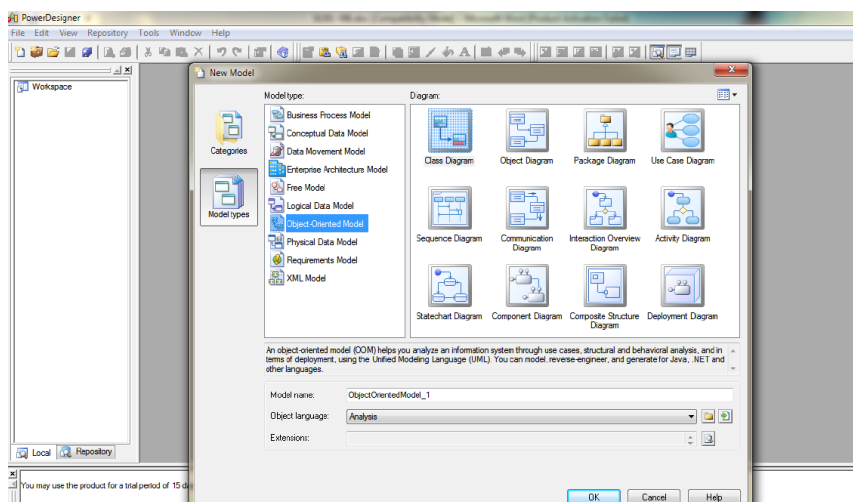


Slika 4.14 Prikaz polimorfizma

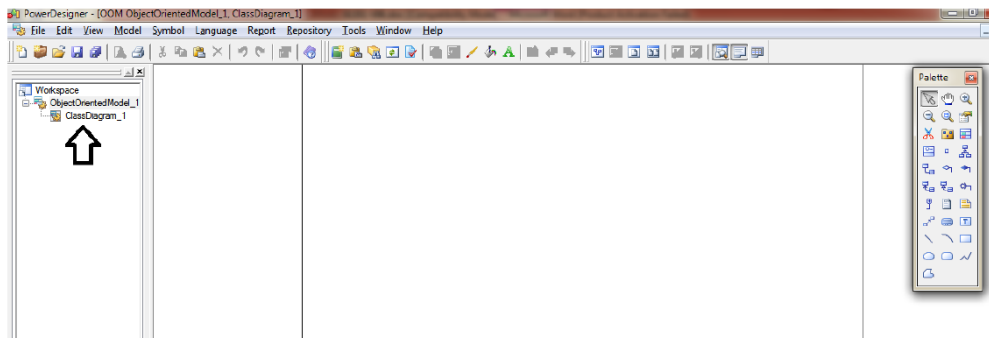
PRIMER 1 - OO MODELOVANJE

Drugi primer koji demonstrira OO modelovanje upotrebom klasnog dijagrama u okviru Power Designer alata

OO modelovanje u PowerDesigner-u se vrši preko kreiranja novog OO modela kao što je prikazano na slici 15 . Kao što se vidi na slici nakon odabira OO modela, prikazuje se lista svih mogućih UML dijagrama koje pripadaju OO modelima. U ovom slučaju mi izabiramo klasni dijagram.



Slika 4.15 Prikaz odabira OO modela korišćenjem Power Designer-a



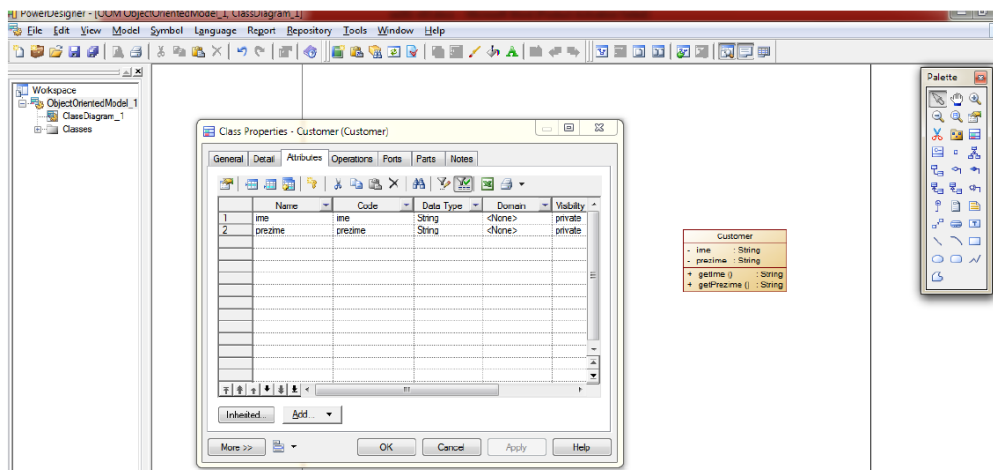
Slika 4.16 Prikaz klasnog dijagrama u okviru OO modela

PRIMER 1 - OO MODELOVANJE - NASTAVAK

Sledeći primer koji demonstrira OO modelovanje upotrebom klasnog dijagrama u okviru Power Designer alata

Sledeći korak je kreiranje nove klase Customer koja ima nekoliko atributa i metoda. Prilikom kreiranja klase

- sledi se jedan od principa OO modeliranja – Sakrivanje informacija. Generalno, atributi su private (sakriveni) i

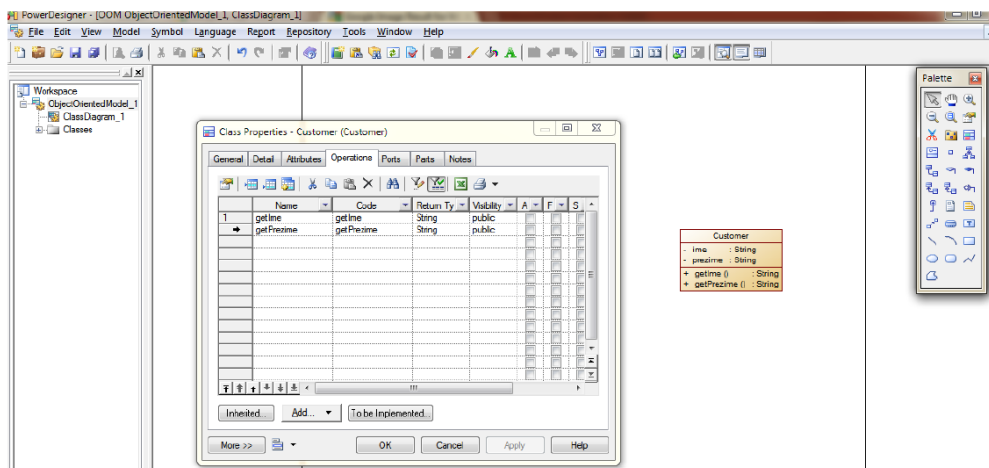


Slika 4.17 Kreiranje atributa klase Customer

PRIMER 1 - OO MODELOVANJE - NASTAVAK 2

Sledeći primer koji demonstrira OO modelovanje upotrebom klasnog dijagrama u okviru Power Designer alata 2

Kako su atributi private (sakriveni) --- pristupa im se spolja preko odgovarajucih metoda koji su u glavnom vidjivi van klase (public).

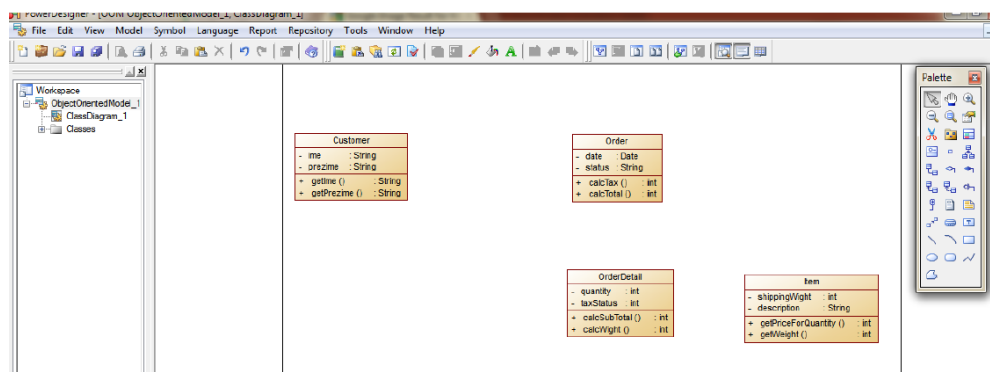


Slika 4.18 Kreiranje operacija (metoda) klase Customer

PRIMER 1 - OO MODELOVANJE - NASTAVAK 3

Sledeći primer koji demonstrira OO modelovanje upotrebom klasnog dijagrama u okviru Power Designer alata 3

Na sličan način dodajemo još 3 klase: Order, OrderDetail i Item.

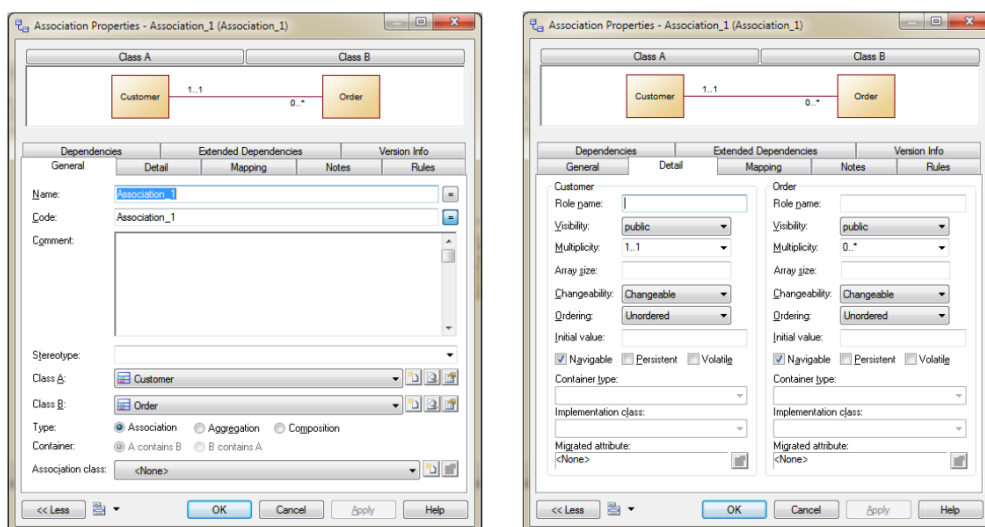


Slika 4.19 Prikaz kreiranih klasa

PRIMER 2 - OO MODELOVANJE

Dodavanje relacija između klasa - relacija i navigacija

Sad dodajemo relacije između klasa. Relacija između Customer i Order je asocijacija (slika levo), multiplicity je one-to-many i navigacija je sa obe strane (slika desno).

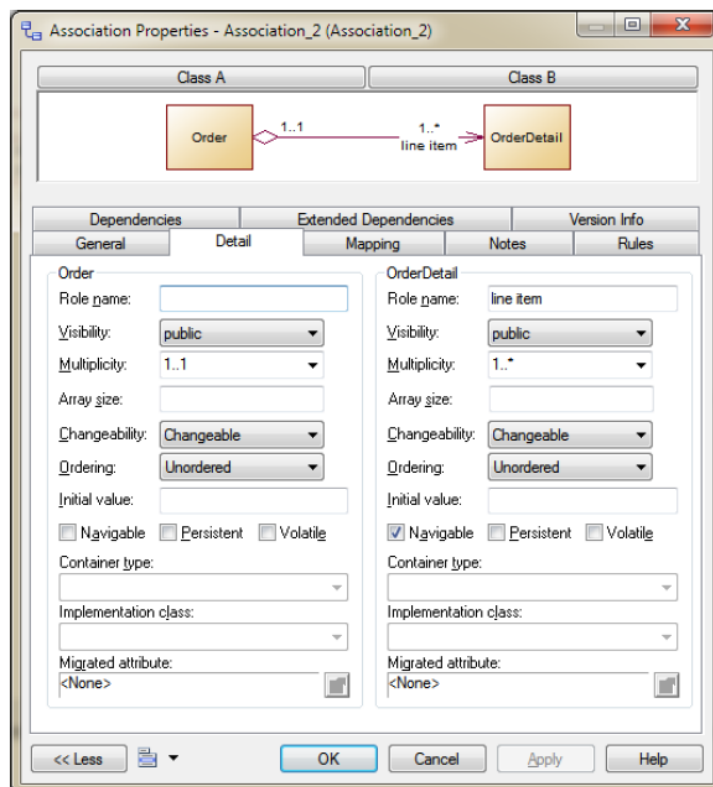


Slika 4.20 Relacija i navigacija

PRIMER 2 - OO MODELOVANJE - NASTAVAK

Dodavanje relacija između klasa - agregacija

Order i OrderDetail su povezani agregacijom (Order sadrži OrderDetail), multiplicity je 1 -1..* i OrderDetail ima ulogu u ovoj relaciji koje se zove line item

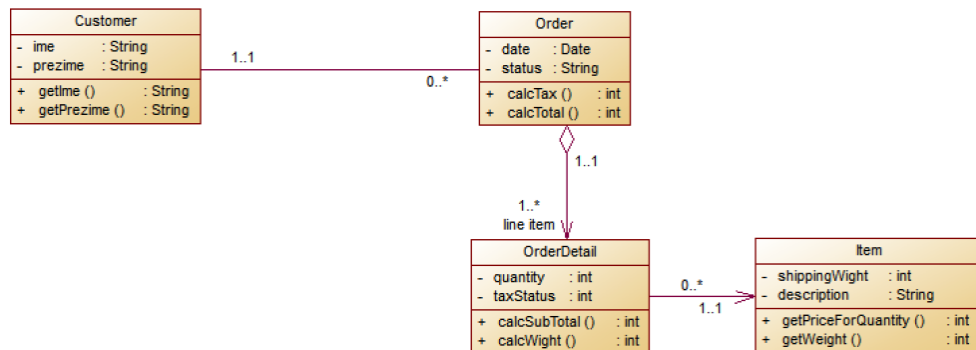


Slika 4.21 Agregacija

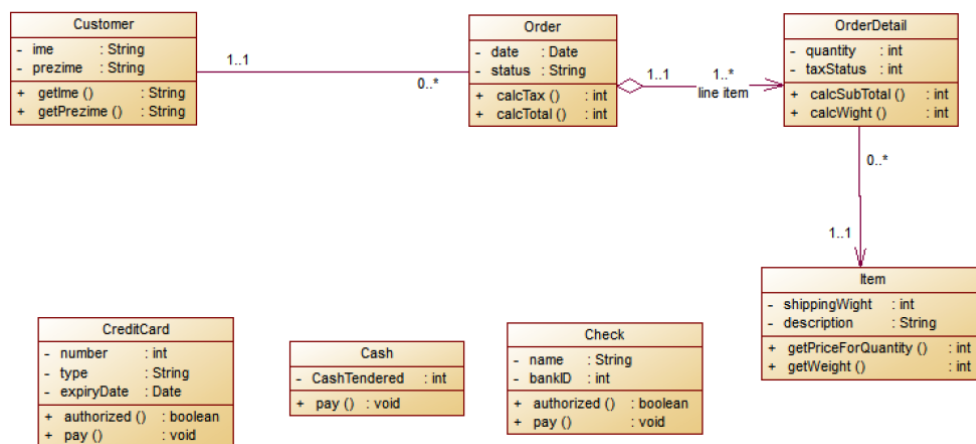
PRIMER 3 - KREIRANJE I KORIŠĆENJE DIJAGRAMA KLASA

Prikaz klasnih dijagrama

Trenutno stanje dijagrama prikazano je na slici 12. Sad bi hteli da modelujemo način plaćanja tako što bi dodali tri nove klase: CreditCard, Cash i Check – za svaki način plaćanja.



Slika 4.22 Prikaz klasnog dijagrama

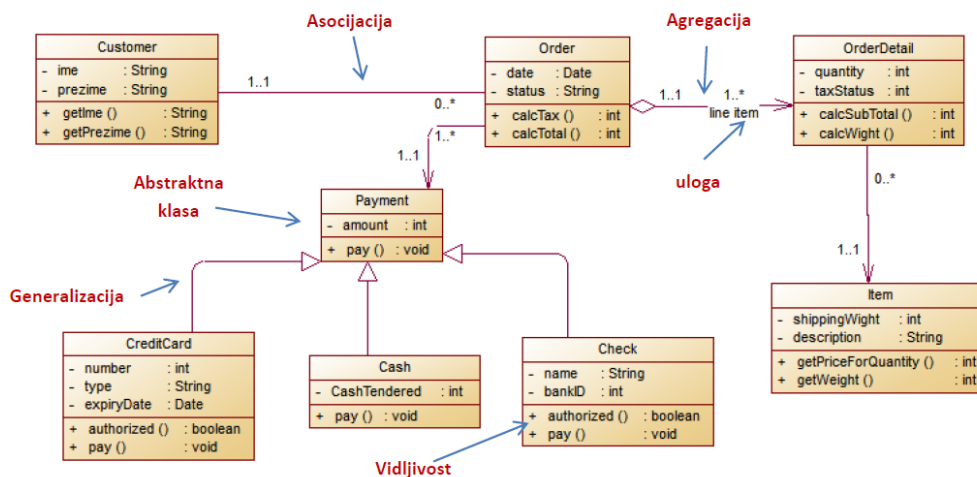


Slika 4.23 Dopuna klasnog dijagrama

PRIMER 3 - KREIRANJE I KORIŠĆENJE DIJAGRAMA KLASA - NASTAVAK

Prikaz klasnih dijagrama - Nastavak

S obzirom da sve tri klase predstavljaju tip plaćanja, tu možemo primeniti dva druga principa OO modeliranja abstrakciju i generalizaciju, tako što bi uvelu novu abstraktnu klasu Payment koja sadrži abstraktnu metodu pay() i generalizovali bi ove tri klase tj. tri tipa plaćanja. Tako da kranji dijagram izgleda kao na slici 24.



Slika 4.24 Prikaz završenog klasnog dijagrama sa vezama

KREIRANJE DIJAGRAMA KLASA SA POWER DESIGNER-OM

Video objašnjava korišćenje SAP Power Designer softverskog alata u kreiranja dijagrama klasa.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

SIMULACIJA KORIŠĆENJA POWER DESIGNER-A (VIDEO)

Ovaj video (bez zvuka) pokazuje kako koristiti Sybase Power Designer u kreiranju dijagrama klasa.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

PRIMER 4

Cilj primer je provežbavanje kreiranja klasnog modela

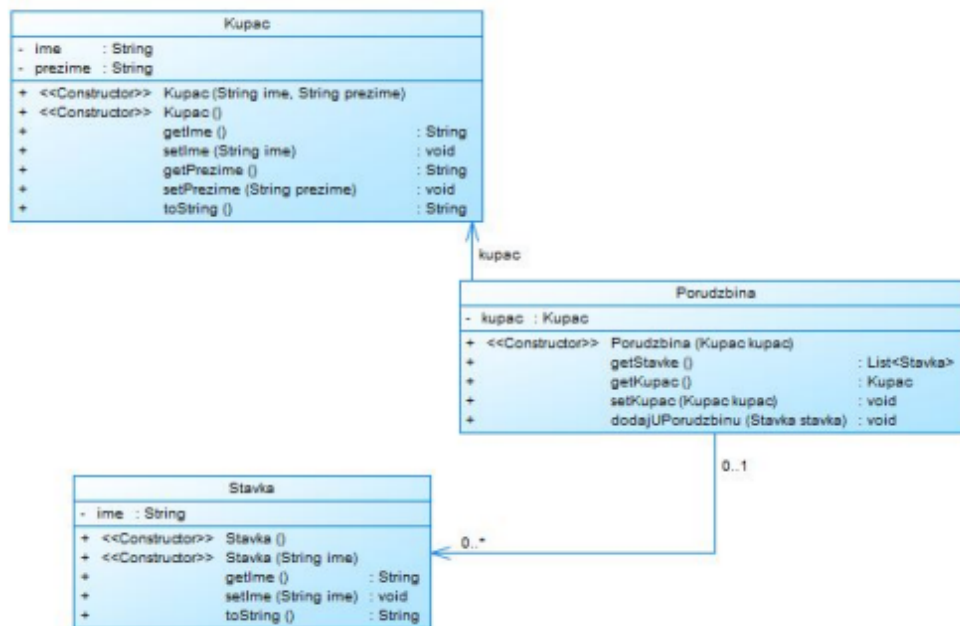
Na osnovu sledećih zahteva napraviti klasni model i implementaciju u Javi. Potrebno nam je da se napravi aplikacija za naručivanje. Mi imamo naše kupce koje pamtimo po imenu i prezimenu. Svaka porudžbina kupca ima više stavki. Svaka stavka (Proizvod) ime svoje ime.

U okviru Main klase kreirati jednu porudžbinu sa nekoliko stavki i prikazati podatke o njoj na konzoli.

Objašnjenje:

Na osnovu opisa problema, uočavamo da klasa Porudzbina sadrži atribut tipa Kupac i listu objekata Stavka (Proizvod).

Po vezama na klasnom modelu zaključujemo da svaka porudžbina ima jednog kupca i 0 ili više stavki. Svaka stavka pripada nijednoj ili jednoj porudžbini.



Slika 4.25 - Rešenje primera 4 u PowerDesigner-u

PRIMER 4 - PROGRAMSKI KOD

Cilj primera je provežbavanje kreiranja klasnog modela i prevođenja istog u Java kod

Klasa Kupac:

```

package zadatak5;

public class Kupac {

    private String ime, prezime;

    public Kupac() {
    }

    public Kupac(String ime, String prezime){
        this.ime = ime;
        this.prezime = prezime;
    }
}

```

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

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

public String getPrezime() {
    return prezime;
}

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

@Override
public String toString() {
    return "Kupac{" + "ime=" + ime
        + ", prezime=" + prezime + '}';
}

}
```

Klasa Stavka:

```
package zadatak5;

public class Stavka {

    private String ime;

    public Stavka() {
    }

    public Stavka(String ime) {
        this.ime = ime;
    }

    public String getIme() {
        return ime;
    }

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

    @Override
    public String toString() {
        return "Stavka{" + "ime="
            + ime + '}';
    }

}
```

Klasa Porudzbina:

```
package zadatak5;

import java.util.ArrayList;
import java.util.List;

public class Porudzbina {

    private List<Stavka> stavke = new ArrayList<Stavka>();
    private Kupac kupac;

    public Porudzbina(Kupac kupac) {
        this.kupac = kupac;
    }

    public List<Stavka> getStavke() {
        return stavke;
    }

    public Kupac getKupac() {
        return kupac;
    }

    public void setKupac(Kupac kupac) {
        this.kupac = kupac;
    }

    public void dodajUPorudzbinu(Stavka stavka) {
        stavke.add(stavka);
    }

    @Override
    public String toString() {
        return "Porudzbina{" + "stavke=" + stavke
            + ", kupac=" + kupac + '}';
    }

}
```

Main klasa:

```
package zadatak5;

public class Zadatak5 {

    public static void main(String[] args) {
        // TODO code application logic here

        Kupac k = new Kupac("Stefan", "Petrovic");

        Stavka s1 = new Stavka("s1");
        Stavka s2 = new Stavka("s2");
    }

}
```

```

        Stavka s3 = new Stavka("s3");
        Stavka s4 = new Stavka("s4");

        Porudzbina p = new Porudzbina(k);
        p.dodajUPorudzbinu(s1);
        p.dodajUPorudzbinu(s2);
        p.dodajUPorudzbinu(s3);
        p.dodajUPorudzbinu(s4);

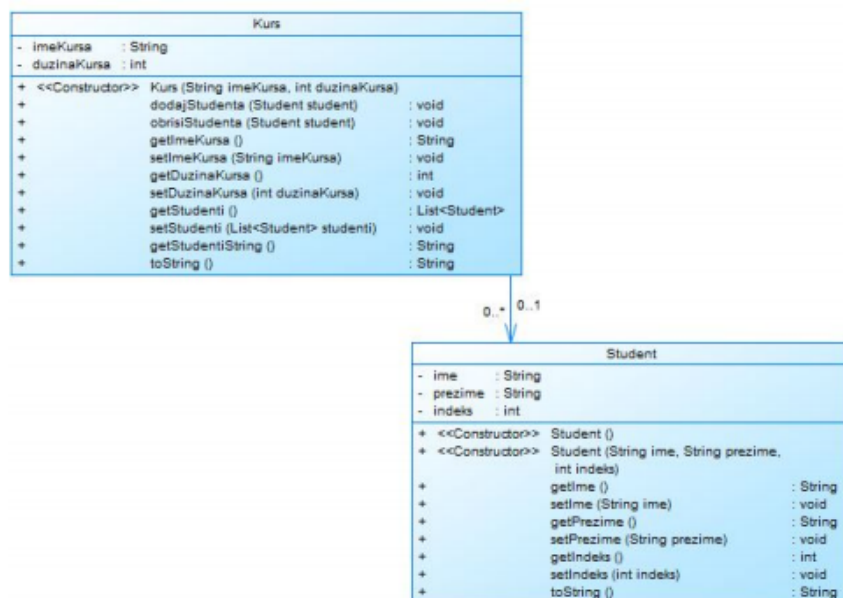
        System.out.println(p);
    }
}

```

PRIMER 5

Cilj primera je prevođenje modela u java kod

Na osnovu klasnog dijagrama sa slike, napisati Java program. U okviru Main klase testirati metode.



Slika 4.26 - Rešenje primera 5 u PoweDesigner-u

Objašnjenje:

Na osnovu dijagrama možemo zaključiti da klasa **Student** od atributa ima ime, prezime i indeks. Svaki student polaže kurseve. Svaki kurs od atributa ima ime kursa kao i dužinu kursa. Svaki kurs može da sluša neograničeno studenata.. Klasa kurs takođe ima metodu `dodajStudenta` koja dodaje studenta na kurs i `obrisiStudenta` koja briše studenta sa kursa

PRIMER 5 - PROGRAMSKI KOD

Cilj zadatka je prevođenje datog modela u Java kod

Klasa Student:

```
package zadatak6;

public class Student {

    private String ime;
    private String prezime;
    private int indeks;

    public Student() {
    }

    public Student(String ime, String prezime, int indeks) {
        this.ime = ime;
        this.prezime = prezime;
        this.indeks = indeks;
    }

    public String getIme() {
        return ime;
    }

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

    public String getPrezime() {
        return prezime;
    }

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

    public int getIndeks() {
        return indeks;
    }

    public void setIndeks(int indeks) {
        this.indeks = indeks;
    }

    @Override
    public String toString() {
        return ime + " " + prezime + " " + indeks;
    }
}
```

```
}  
}
```

Klasa Kurs:

```
package zadatak6;  
  
import java.util.ArrayList;  
import java.util.List;  
  
public class Kurs {  
  
    private String imeKursa;  
    private int duzinaKursa;  
    List<Student> studenti = new ArrayList<Student>();  
  
    public Kurs(String imeKursa, int duzinaKursa) {  
        this.imeKursa = imeKursa;  
        this.duzinaKursa = duzinaKursa;  
    }  
  
    public void dodajStudenta(Student student) {  
        studenti.add(student);  
    }  
  
    public void obrisiStudenta(Student student) {  
        studenti.remove(student);  
    }  
  
    public String getImeKursa() {  
        return imeKursa;  
    }  
  
    public void setImeKursa(String imeKursa) {  
        this.imeKursa = imeKursa;  
    }  
  
    public int getDuzinaKursa() {  
        return duzinaKursa;  
    }  
  
    public void setDuzinaKursa(int duzinaKursa) {  
        this.duzinaKursa = duzinaKursa;  
    }  
  
    public List<Student> getStudenti() {  
        return studenti;  
    }  
  
    public void setStudenti(List<Student> studenti) {  
        this.studenti = studenti;  
    }  
}
```

```

    public String getStudentiString() {
        String returnStudenti = "";
        for (Student std : studenti) {
            returnStudenti += std + "\r\n";
        }
        return returnStudenti;
    }

    @Override
    public String toString() {
        return imeKursa + " Duzina kursa: " + duzinaKursa;
    }
}

```

Main klasa:

```

package zadatak6;

public class Zadatak6 {
    public static void main(String[] args) {
        // TODO code application logic here
        Student student1 = new Student("Marko", "Markovic", 1250);
        Student student2 = new Student("Petar", "Markovic", 1240);

        Kurs kurs = new Kurs("CS101", 15);
        kurs.dodajStudenta(student1);
        kurs.dodajStudenta(student2);
        System.out.println("Velicina je:"
            + kurs.getStudenti().size());

        kurs.obrisiStudenta(student1);
        System.out.println("Velicina je:"
            + kurs.getStudenti().size());
    }
}

```

ZADACI ZA SAMOSTALNI RAD

Na osnovu predhodnih primera uraditi sledeće zadatke

Zadatak 4.1

Napraviti klasu Životinja. Životinja mora imati makar 3 atributa po slobodnom izboru i jedan atribut tip koji može imati vrednosti: Domaća životinja, odnosno Divlja životinja.. Napraviti model u Power Designer-u i nakon toga implementirati model u Javi.

Zadatak 4.2:

Napraviti sledeći model u Power Designeru: Naša firma se bavi proizvodnjom računara. Naše računare delimo na brze i spore. Pored računara proizvodimo još i web kamere, miševe kao i ostale periferene usb uređaje.

Dobijeni model implementirati u Javi.

Zadatak 4.3:

Potrebno je u Power Designeru napraviti model aparata za kafu. Aparat može da pravi više vrsta kafa i pića. Vrste kafa i pića su: Kapučino, Late, Nes kafa, Topla čokolada, Mleko.

Dobijeni model implementirati u Javi.

▼ Poglavlje 5

Domaći zadatak

DOMAĆI ZADATAK 1

Zadatke treba samostalno uraditi i poslati asistentu na pregled

Zadatak 1:

Napraviti klasni model u **PowerDesigner**-u i implementaciju modela u **Javi** za sledeći primer:
Naša firma se bavi proizvodnjom prozora . Prozore delimo na drvene, metalne i aluminiumske.
Svaki prozor ima svoju cenu, firmu kao i naziv.

O firmama sa kojim sarađujemo obično čuvamo informacije kao što su naziv firme, pib i država firme.

Dobijeni model implementirati u Javi.

Zadatak 2:

Napraviti simulaciju igrice u kojoj se trkaju dva motocikla.

Treba napraviti klasu Motocikl i klasu Trka.

Svaki Motocikl treba da ima naziv, maksimalnu brzinu kao i ubrzanje. Klasa Trka treba da predstavlja klasu u kojoj se odvija sama trka. Od atributa klasa treba da ima dva motocikla, dužinu staze i, vreme trke, kao i koji je motocikl pobednik na kraju trke.

Koristeći while petlju napraviti simulaciju trkanja tako da se vreme povećava za jedan a brzina motocikla ja jednaka njegovom ubrzanju pomnoženom sa vremenom. Paziti da brzina motocikla ne može preći maksimalnu brzinu.

▼ Poglavlje 6

Rezime kroz dodatne primere

DODATNI PRIMERI

Primeri za dodatnu vežbu pređenog gradiva

U ovoj sekciji su dati dodatni primeri kojim opisujemo gradivo iz prethodnih sekcija

PRIMER 1

Simulacija igre sa pet igrača.

Tekst zadatka:

Napraviti simulaciju igre od pet igrača u kojoj svaki igrač baca pikado, imena igrača su: Jovan, Milorad, Dragana, Milena i Petar.

Broj poena za svako bacanje može da bude u intervalu od 0 do 90, pri čemu su vrednosti deljive sa 10, na primer: 10, 20, 30..., 100.

Ukupan broj poena se uvećava za vrednost trenutnog bacanja za svakog igrača u svakom krugu.

Pobednik igre je onaj igrač koji prvi skupi 1000 ili više poena.

Program treba da simulira ovu igru tako što prikazuje broj trenutnog kruga, broj poena dobijen u trenutnom bacanju i ukupan broj poena za svakog igrača u trenutnom bacanju.

Na kraju simulacije program treba da prikaže ime pobednika korišćenjem standardnog izlaza.

Ime klase treba da bude Simulacija u okviru paketa cs101.v08.

Analiza problema:

Najjednostavniji način da uradimo ovaj zadatak je da kompletan programski kod smestimo u statičku klasu main, pri čemu će nam biti potrebne sledeće promenljive:

5 promenljivih tipa String koje će čuvati **imena igrača**, pri čemu ih inicijalizujemo na početku na date vrednosti - **ime1, ime2, ime3, ime4, ime5**

5 promenljivih tipa int koje će čuvati **trenutni broj poena igrača**, pri čemu ih inicijalizujemo na početku na vrednost 0 - **brojPoena1, brojPoena2, brojPoena3, brojPoena4, brojPoena5**

1 promenljiva tipa int koja će čuvati trenutni broj krugova, pri čemu je inicijalizujemo na vrednost 1 - **brojKrugova**

1 promenljiva tipa int koja će čuvati vrednost dobijenu prilikom nekog bacanja, pri čemu je inicijalizujemo na vrednost 0 - **trenutnoBacanje**

1 promenljiva tipa Random koja će nam služiti kao generator slučajnih brojeva - **rand**

PRIMER 1- ANALIZA PROBLEMA

Sve dok važi uslov da nijedan od igrača nema više od 1000 poena, biće potrebna petlja koja će ovo da realizuje.

S obzirom da se bacanja odvijaju sve dok važi uslov, tj. nijedan od igrača nema više od 1000 poena, biće nam potrebna petlja koja će ovo da realizuje. U svakoj iteraciji petlje ćemo:

- prikazati broj trenutnog bacanja
- generisati vrednost slučajnog bacanja za sve igrače korišćenjem klase **Random** i dodati dobijene vrednosti broju poena za svakog igrača
- prikazati vrednost slučajnog bacanja i broj poena za svakog igrača u tom krugu

Nakon izlaska iz petlje ćemo odrediti maksimalan broj poena i na osnovu toga ispisati ime igrača koji je pobedio.

Kao što možete primetiti u ovom slučaju se javlja se problem ako odlučimo da dodamo još 10 igrača u simulaciju igre, programski kod bi se povećao za 80 redova. Takođe, veći deo koda se ponavlja za različite igrače.

Prvi način za poboljšanje programskog koda je da napravimo dva niza koji će čuvati imena igrača i brojeve poena - nizImena i **nizPoena**, pri čemu ćemo elemente niza nizImena inicijalizovati na konkretne vrednosti, a elemente niza nizPoena na vrednosti 0.

Uvođenjem niza, u petlji možemo da proverimo da li uslov važi, tj. da li i-ti igrač ima manje od 1000 poena - ako uslov ne važi petlja se prekida. Ovo ćemo najlakše realizovati uvođenjem logičke promenljive uslov koja će u početku imati vrednost true, što označava da uslov važi, a ako i-ti igrač ima više ili jednako 1000 poena logička promenljiva uslov dobija vrednost false, a pobednik je i-ti element niza imena.

PRIMER 1 - KLASA SIMULACIJA

Klasa Simulacija sadrži metod main i omogućava simulaciju igre.

Programski kod klase Simulacija:

```
package cs101.v07;  
  
import java.util.Random;
```

```
public class Simulacija {
    public static void main(String[] args) {
        Random r = new Random();
        String[] nizImena = {"Jovan", "Milorad", "Milena", "Dragana", "Petar"};
        int[] nizPoena = {0, 0, 0, 0, 0};
        boolean uslov = true;
        int brojKrugova = 1;
        String pobednik = "";
        int trenutnoBacanje = 0;
        while (uslov) {
            System.out.println("Krug: " + brojKrugova);
            for (int i = 0; i < nizImena.length; i++) {
                trenutnoBacanje = r.nextInt(10) * 10;
                nizPoena[i] += trenutnoBacanje;
                System.out.print("Igrac " + nizImena[i] + " je u ovom krugu
osvojio: " + trenutnoBacanje + " poena. ");
                System.out.println("Igrac " + nizImena[i] + " ima ukupno: " +
nizPoena[i] + " poena.");
                if (nizPoena[i] >= 1000){
                    uslov = false;
                    pobednik = nizImena[i];
                }
            }
            brojKrugova++;
            System.out.println();
        }
        System.out.println("Pobednik je: " + pobednik);
    }
}
```

KLASA IGRAC

Klasa Igrac definiše ime igrača i broj poena koje je on osvojio, pored metoda koji koriste ove atribute.

Javlja se problem ako odlučimo da dodamo još 5 svojstava za svakog igrača, onda bi morali da pravimo još 5 nizova. Takođe, ovo rešenje nije objektno-orjentisano, jer igrači nisu opisani klasom Igrač, već se simuliraju nizovima.

Svaki igrač treba da ima atribute ime i brojPoena i konstruktor kojim se prosleđuje ime, a broj poena se postavlja na vrednost 0. Takođe u klasi Igrač možemo da realizujemo i metodu ispiši koja će u svakom krugu generisati vrednost slučajnog bacanja, dodavati dobijenu vrednost broju poena za tog igrača i ispisivati vrednosti za slučajno bacanje i broj poena u tom krugu.

Sledi programski kod za klasu Igrač:

```
package cs101.v07;
import java.util.Random;

public class Igrac {
```



```

private String ime;
private int brojPoena;
public Igrac(String ime) {
    this.ime = ime;
    this.brojPoena = 0;
}
public String getIme() {
    return ime;
}
public void setIme(String ime) {
    this.ime = ime;
}
public int getBrojPoena() {
    return brojPoena;
}
public void setBrojPoena(int brojPoena) {
    this.brojPoena = brojPoena;
}
public void ispisi(){
    Random r = new Random();
    int bacanje = r.nextInt(10) * 10;
    System.out.print("Igrac " + this.ime + " je u ovom krugu osvojio: " +
    bacanje + " poena. ");
    this.brojPoena += bacanje;
    System.out.println("Igrac " + this.ime + " ima ukupno:          : " +
    this.brojPoena + " poena.");
}
}

```

PRIMER 1 - KLASA SIMULACIJA OOP

Završna verzija programskog koda na objektno-orijentisan način data je u klasi SimulacijaOOP

U glavnom programu ćemo umesto da imamo po jedan niz za svako svojstvo igrača, imati niz igrača. Sledi treća, i završna, verzija programskog koda na objektno-orijentisan način data u klasi **SimulacijaOOP**:

```

package cs101.v07;

public class SimulacijaOOP {
    public static void main(String[] args) {
        Igrac[] nizIgraca = {new Igrac("Jovan"), new Igrac("Milorad"),
        new Igrac("Milena"), new Igrac("Dragana"), new Igrac("Petar")};
        boolean uslov = true;
        String pobednik = "";
        int brojKrugova = 1;
        while (uslov) {
            System.out.println("Krug: " + brojKrugova);
            for (int i = 0; i < nizIgraca.length; i++) {

```

```
nizIgraca[i].ispisi();
if (nizIgraca[i].getBrojPoena() >= 1000){
    uslov = false;
    pobednik = nizIgraca[i].getIme();
}
}
brojacKrugova++;
}
System.out.println("Pobednik je: " + pobednik);
}
}
```

ZADATAK ZA SAMOSTALNI RAD STUDENTA

Ovaj zadatak svaki student treba samostalno da uradi.

Zadatak za samostalni rad:

Napraviti simulaciju igre od 10 igrača u kojoj svaki igrač baca tri kocke za jamb, imena igrača odabrati po izboru.

Broj poena za svako bacanje može da bude u intervalu od 3 do 18.

Ukupan broj poena se uvećava za vrednost trenutnog bacanja za svakog igrača u svakom krugu.

Pobednik igre je onaj igrač koji prvi skupi 200 ili više poena.

Program treba da simulira ovu igru tako što prikazuje broj trenutnog kruga, broj poena dobijen u trenutnom bacanju i ukupan broj poena za svakog igrača u trenutnom bacanju.

Na kraju simulacije program treba da prikaže ime pobednika korišćenjem standardnog izlaza.

Zadatak prvo uraditi na proceduralni način, ime klase treba da bude Simulacija u okviru paketa cs101.iv08.

Zatim uraditi zadatak na objektno-orjentisani način, ime klasa treba da budu Simulacija i IgracJamb u okviru paketa cs101.iv08

PRIMER 2

Korišćenje klase Student i klase StudentMain

Tekst zadatka:

Napraviti klasu **Student** koja opisuje jednog studenta i pokretačku klasu **StudentMain** koja demonstrira sve funkcionalnosti klase **Student**.

Klasa trebaju biti deo NetBeans projekta pod nazivom KI104-V08 i deo paketa cs101.v08.student.

Za učitavanje podataka od korisnika treba koristiti objekat klase Scanner, a za prikaz podataka objekat System.out.

Napomena:

S obzirom da u ovoj vežbi po prvi put kreiramo sve moguće funkcionalnosti jedne klase, od konstruktora, atributa i metoda do preopterećenih konstruktora, seter i getter metoda i pisanja dokumentacionih komentara, tekst zadatka ćemo detaljno definisati uporedo sa rešenjem u obliku isečaka programskog koda, po koracima, a u video materijalu ćemo sve prikazati korišćenjem razvojnog okruženja NetBeans.

Korak 1:

Definisati klasu Student i kreirati sledeće atribute:

- ime, tipa String
- brojIndeksa, tipa int

Izvršiti enkapsulaciju atributa kreiranjem odgovarajućih getter i setter metoda.

Napomena:

Razvojno okruženje NetBeans poseduje opciju za automatsko kreiranje getter i setter metoda na osnovu atributa, može se koristiti opcija Source - Insert Code ili prečica Alt + Insert, zatim opcija Getter and Setter... , nakon čega odaberemo attribute redom i kliknemo na dugme Generate.

PRIMER 2 - KORAK 1: KLASA STUDENT I STUDENTMAIN

Listing klase Student i klase StudentMain

Programski kod klase **Student**: i klasa **StudentMain**

```
package cs101.v07.student;
public class Student {
    private String ime;
    private int brojIndeksa;
    public String getIme() {
        return ime;
    }
    public void setIme(String ime) {
        this.ime = ime;
    }
    public int getBrojIndeksa() {
        return brojIndeksa;
    }
    public void setBrojIndeksa(int brojIndeksa) {
        this.brojIndeksa = brojIndeksa;
    }
}
```

```
public class StudentMain {
    public static void main(String[] args) {
        Student student1 = new Student();
        student1.setIme("Milorad Novakovic");
        student1.setBrojIndeksa(2222);
    }
}
```

PRIMER 2 - KORAK 2

Konstruktor koji koristi ključnu reč this prima vrednosti atributa klase u kojoj je konstruktor definisan.

Kreirati podrazumevani (ili prazan) konstruktor koristeći ključnu reč this, tj. konstruktor koji ne prima argumente, tako da postavlja ime na vrednost "Student X", a broj indeksa na vrednost 3000.

Kreirati preopterećeni konstruktor koji prima vrednosti svih atributa.

Kreirati preopterećeni konstruktor kopije, tj. konstruktor koji prima objekat klase Student kao argument.

Kreirati metodu ispisi(), koja vraća vrednosti svih atributa u sledećem formatu:

Ime studenta: Student X

Broj indeksa studenta: 3000

Koristeći pokretačku klasu StudentMain demonstrirati sve trenutne funkcionalnosti klase Student.

Napomena:

Za kreiranje podrazumevanog konstruktora možemo koristiti opciju Insert Code ili prečicu Alt + Insert, zatim opcija Constructor i kliknemo na dugme Generate. Slično je i za kreiranje konstruktora koji prima argumente, ali pritom treba izabrati vrednosti atributa koje će biti postavljene na vrednosti argumenata ovog konstruktora

PRIMER 2 - KLASA STUDENT

Klasa Student sadrži zajedničke attribute za sve studente i metod za korišćenje tih atributa.

Programski kod klase Student

```
package cs101.v07.student;
public class Student {
    private String ime;
    private int brojIndeksa;
```

```
public Student() {
    this.ime = "Student X";
    this.brojIndeksa = 3000;
}
public Student(String ime, int brojIndeksa) {
    this.ime = ime;
    this.brojIndeksa = brojIndeksa;
}
public Student(Student studentTemp) {
    this.ime = studentTemp.ime;
    this.brojIndeksa = studentTemp.brojIndeksa;
}
public String getIme() {
    return ime;
}
public void setIme(String ime) {
    this.ime = ime;
}
public int getBrojIndeksa() {
    return brojIndeksa;
}
public void setBrojIndeksa(int brojIndeksa) {
    this.brojIndeksa = brojIndeksa;
}
public void ispisi(){
    System.out.println("Ime studenta: " + this.ime);
    System.out.println("Broj indeksa studenta: " + this.brojIndeksa);
}
}
```

PRIMER 2 - KLASA STUDENTMAIN

Main klasa sadrži main metod

Programski kod klase StudentMain

```
package cs101.v07.student;

public class StudentMain {
    public static void main(String[] args) {
        Student s1 = new Student();
        Student s2 = new Student("Milorad Novakovic", 2222);
        Student s3 = new Student(s2);
        s1.ispisi();
        s2.ispisi();
        s3.ispisi();
    }
}
```

PRIMER 2 - KORAK 3

Promeniti programski kod tako da vrednosti atributa ne mogu da se menjaju nakon kreiranja. Ovo se realizuje brisanjem seter metoda

Promeniti programski kod tako da vrednosti atributa ne mogu da se menjaju nakon kreiranja. Ovo ćemo realizovati tako što ćemo seter metode obrisati ili staviti pod komentare, tako da ih kompajler ne prepozna kao deo programskog koda.

Koristeći pokretačku klasu StudentMain demonstrirati sve trenutne funkcionalnosti klase Student.

Napisati JavaDoc dokumentacione komentare za klase Student i StudentMain.

Ovde se daje kod za klasu Student:

```
package cs101.v07.student;

/**
 * Klasa Student predstavlja klasu kojom opisujemo studente.
 * Klasa Student sadrži:
 * - attribute koji opisuju: ime studenta i broj indeksa studenta i adekvatne seter
i getter metode za svaki atribut
 * - podrazumevani (prazan) konstruktor, konstruktor za inicijalizaciju atributa i
tzv. konstruktor kopije
 *
 * @author Nikola
 */

public class Student {
    private String ime;
    private int brojIndeksa;

    /**
     * Podrazumevani (prazan) konstruktor koji postavlja inicijalne vrednosti
atributa
     */
    public Student() {
        this.ime = "Student X";
        this.brojIndeksa = 3000;
    }

    /**
     * Konstruktor koji postavlja ime i broj indeksa na date vrednosti
     * @param ime data vrednost za ime
     * @param brojIndeksa data vrednost za broj indeksa
     */
    public Student(String ime, int brojIndeksa) {
        this.ime = ime;
        this.brojIndeksa = brojIndeksa;
    }
}
```

```
* Konstruktor kopije, tj. konstruktor koji prima objekat klase Student i
vrednosti atributa postavlja na vrednosti tog objekta
* @param studentTemp dati objekat klase Student
*/
public Student(Student studentTemp) {
    this.ime = studentTemp.ime;
    this.brojIndeksa = studentTemp.brojIndeksa;
}
/**
 * Vraća ime studenta
 * @return ime studenta
 */
public String getIme() {
    return ime;
}
/*
public void setIme(String ime) {
    this.ime = ime;
}
*/
/**
 * Vraća broj indeksa studenta
 * @return broj indeksa studenta
 */
public int getBrojIndeksa() {
    return brojIndeksa;
}
/*
public void setBrojIndeksa(int brojIndeksa) {
    this.brojIndeksa = brojIndeksa;
}
*/
/**
 * Ispisuje vrednosti svih atributa klase Student: imena i broja indeksa
 */
public void ispisi() {
    System.out.println("Ime studenta: " + this.ime);
    System.out.println("Broj indeksa studenta: " + this.brojIndeksa);
}
}
```

ZADATAK ZA SAMOSTALNI RAD SA KLASOM STUDENTMAIN

Klasa StudentMain predstavlja pokretačku klasu kojom demonstriramo sve trenutne funkcionalnosti klase Student

Programski kod klase StudentMain

Zadatak za samostalni rad:

Na osnovu vežbe kreirati klasu Profesor , tako da sadrži sve elemente koje ima klasa Student i pokretačku klasu ProfesorMain koja demonstrira sve funkcionalnosti ovih klasa. Svaki profesor je opisan imenom, prezimenom, zvanjem i brojem godina.

Klasa treba da bude deo NetBeans projekta pod nazivom KI104-IV08 i deo paketa cs101.iv08.

Za učitavanje podataka od korisnika treba koristiti objekat klase Scanner, a za ispit podataka objekat System.out.

Napisati dokumentacione komentare za sve klase

```
package cs101.v07.student;

/**
 * Klasa StudentMain predstavlja pokretačku klasu kojom demonstriramo
 * sve trenutne funkcionalnosti klase Student
 *
 * @author Nikola
 */
public class StudentMain {
    /**
     * Statička pokretačka metoda main
     * @param args
     */
    public static void main(String[] args) {
        Student s1 = new Student();
        Student s2 = new Student("Milorad Novakovic", 2222);
        Student s3 = new Student(s2);

        // s1.setIme("Promenjen student"); // GREŠKA

        s1.ispisi();
        s2.ispisi();
        s3.ispisi();
    }
}
```


▼ Zaključak

REZIME URAĐENOG

U predavanju smo se upoznali sa:

- Definicijom klase
- Dužinom trajanja i oblašću važenja promenljivih
- Promenljivima klasnog tipa
- Konstrukcijom, inicijalizacijom i uklanjanjem objekata
- Enkapsulacijom (učaurivanjem) i sakrivanjem podataka
- Ključnom reči this

U vežbi smo prikazali:

Kreiranje klase **Student** i njenih atributa, metoda, setter i getter metoda, preopterećenih konstruktora i dokumentacionih komentara i kreiranje pokretačke klase StudentMain koja demonstrira funkcionalnosti klase **Student**

Kreiranje klase **Kvadrat** i njenih atributa, statičkih atributa, metoda, statičkih metoda, setter i getter metoda, preopterećenih konstruktora i dokumentacionih komentara i kreiranje pokretačke klase **KvadratMain** koja demonstrira funkcionalnosti klase **Kvadrat**

Kreiranje programa za simulaciju igre bacanja pikada za pet igrača na proceduralni način

Kreiranje programa za simulaciju igre bacanja pikada za pet igrača na objektno-orjentisani način