**OBJEKTNO ORIJENTISANO**

**PROGRAMIRANJE 1**

ODGOVORI NA PITANJA ZA

USMENI DEO ISPITA

JANUAR 2021

Git: <https://github.com/NikolaVetnic/OOP1>

Objekti i klase.

01

**OBJEKTNO ORIJENTISAN PROGRAM:** ideja – pakovanje podataka i funkcija koje ih obrađuju u jednu celinu predstavljenu jednom promenljivom, tj. objektom sačinjenom od atributa/polja (promenljive, opisuju stanje) i metoda (funkcije, definišu ponašanje); OO program je skup objekata u međusobnim interakcijama (npr. objA poziva metoda nad objB), a konkretno Java program je kolekcija **1)** klasa (apstraktne, konkretne – finalne i nefinalne, anonimne, lokalne) i **2)** interfejsa (ugovor koji kaže da ako klasa implementira interfejs tada ona implementira sve metode propisane interfejsom) koji se mogu grupisati po **3)** paketima (prostori imena, mehanizam hijerarhijske dekompozicije – klase u istom paketu imaju različita, a u različitim paketima mogu imati ista imena), koji obično sadrže pakete, srodne klase i interfejse.

**KLASA:** definicija skupa istorodnih objekata (ista specifikacija stanja i ponašanja, tj. atributi i metode), klase su referencijalni tipovi objekata (tip određuje skup vrednosti i skup operacija nad promenljivom); pisanje OO programe podrazumeva definiciju klasa i instanciranje (kreiranje) objekata definisanih klasa (primer klase i objekata sa crteža: – klasa kao „prototip“, objekti kao konkretne instance sa različitim stanjima).

Deklaracija klase koja nije ugnježdena može početi nekim od modifikatora: public, abstract, final, strictfp (reprezentacija i operacije sa realnim brojevima po IEEE 754 standardu); deklaracija polja može početi modifikatorima: public, protected, private, final, static, transient (polje se ne koristi pri serijalizaciji i deserijalizaciji objekata date klase), volatile (sinhronizovano polje, atomičke operacije kod višenitnih aplikacija); zaglavlje metoda može početi modifikatorima: public, protected, private, abstract, final, static, synchronized (sinhronizovana metoda), native (apstraktan metod implementiran u nekom drugom jeziku u nekoj od biblioteka koje se dinamički učitavaju), strictfp (striktne floating point operacije po IEEE 754).

**IMPERATIVNI PJ:** program je formalni opis procesa (imperativni PJ) ili specifikacije (deklarativni PJ) izračunavanja u PJ, ostvaren kroz skup naredbi (naredbe dodele, kontrole toka, itd.) koje menjaju vrednosti skupa promenljivih (odnosno stanje programa). Kategorije: **1)** proceduralni – dekompozicija u funkcije i procedure, lokalne i globalne promenljive; **2)** modularni – dekompozicija u module (logički srodne definicije promenljivih, funkcija i tipova, sa privatnim i javnim delom); **3)** OO – dekompozicija u klase (logički srodne def. promenljivih i funkcija, klase kao tipovi, privatni i javni deo, nasleđivanje klasa).

**KONSTRUKTORI I DESTRUKTORI:** konstruktor – specijalni metod koji se poziva pri instanciranju klase radi inicijalizacije stanja objekta, destruktor – specijalni metoda koji se poziva kada se objekat uništava. U Java PJ: **1**) instanciranje operatorom new; **2)** rezultat je referenca na instancirani objekat; **3)** moguće je više referenci na isti objekat; **4)** nema destruktora, automatsko uništavanje objekata (*garbage collector*) kada više nema referenci na objekat.

Apstrakcija, enkapsulacija i sakrivanje informacija.

02

**APSTRAKCIJA:** suština PJ je da omoguće mehanizme: **1)** apstrakcije - sakrivanje (zanemarivanje, odlaganje, zaboravljanje) nebitnih (tehničkih, implementacionih) detalja, što smanjuje kompleksnost; **2)** dekompozicije – razlaganje entiteta (problema) u pod-entitete koji su manje kompleksnosti. Paradigme: **1**) proceduralni PJ – dekompozicija u procedure, gde korisnik zna samo zaglavlje a ne i telo procedure da bi je koristio; **2)** OO PJ – dekompozicija u objekte, gde korisnik zna samo zaglavlja javnih metoda objekta.

**SAKRIVANJE INFORMACIJA:** public i private delovi objekta, kroz modifikatore vidljivosti/pristupa prilikom definicije atributa i metoda klase – privatni atributi vidljivi samo u klasi i van nje su dostupni samo preko metoda (primer: get i set metode), privatne metode se pozivaju samo iz iste klase, javnim atributima i metodama se može pristupati iz drugih klasa.

**ENKAPSULACIJA:** dizajn objekta koji skriva detalje nebitne za korišćenje od korisnika (ako su svi atributi privatni tada objekat ima potpunu kontrolu nad svojim stanjem), a takođe promena atributa klase ne utiče na druge klase kod održavanja i modifikacije programa (primer: class Tacka sa private double x, y poljima).

Agregacija i kompozicija klasa.

03

**AGREGACIJA I KOMPOZICIJA:** osnova za interakciju objekata – objekat a klase A ima referencu na objekat b klase B, odnosno u definiciji klase A postoji atribut p tipa B; ukoliko klasa A instancira p tada su A i B u **relaciji kompozicije**, budući da je b deo a koji ne može da postoji bez a i uništavanjem objekta a uništava se i objekat b (primer: A = Čovek, B = Srce); ako klasa A ne instancira p tada su A i B u **relaciji agregacije**, jer b može da postoji nezavisno od a (primer: A = FudbalskiKlub, B = Fudbaler).

Promenljive referencijalnog tipa, referenca this.

04

**PROMENLJIVE:** postoje različite vrste promenljivih: atributi klase, parametri metoda, lokalne promenljive u telu metoda; leksička pravila opsega – za neku pojavu identifikatora u kodu tražimo toj pojavi najbližu definiciju tipa, i to prvo u tekućem bloku a onda dalje redom po nadgnježdenim blokovima, listi parametara metoda i na kraju atributima klase.

**PROMENLJIVE REFERENCIJALNOG TIPA:** vrednost promenljive čiji je tip neka klasa je referenca na objekat te klase, ili null (tada ona ne pokazuje ni na jedan objekat) – vrednost koja se može dodeliti bilo kojoj promenljivoj ref. tipa; budući da su klase ref. tipovi, one mogu biti tipovi parametara i povratne vrednosti metoda; promenljive ovog tipa se implicitno inicijalizuju na null; moguće je kreirati nizove objekata čiji je tip neka klasa;

**REFERENCA this:** svaki objekat ima implicitno definisano polje koje se zove this i to je upravo referenca na samog sebe, što je zgodna referenca kada postoji kolizija na nivou imena (primer: inicijalizacija atributa u konstruktoru).

**REFERENCA super:** još jedno polje implicitno prisutno u svim objektima, super referencira delove objekta koji su nasleđeni; koristeći super može se pristupiti nasleđenim metodama i atributima u slučaju da su redefinisani u izvedenoj klasi.

**OPERATORI:** operatorom = se kopira vrednost reference, ne objekat; operatorima == i != se porede reference a ne vrednosti objekta;

Konstruktori i operator new.

05

**KONSTRUKTORI:** konstruktor (ctor) je poseban tip metode koja se poziva prilikom instanciranja klase; klasa može definisati više ctor-a koji se tada moraju razlikovati po broju parametara ili bar jedan mora biti različitog tipa; ctor takođe može pozvati drugi ctor, u kom slučaju se koristi ključna reč this iza koje se zadaju vrednosti parametara (na osnovu tipova kojih se određuje koji ctor se poziva) – ovo mora biti prva naredba u ctor-u; ctor takođe može koristiti super čime se poziva odgovarajući ctor nadklase, što se opet određuje na osnovu parametara; kod nasleđivanja ako bazna klasa definiše bar jedan ctor tada izvedena mora imati bar jedan ctor sa super pozivom, a ako to nije slučaj dodaje joj se podrazumevani ctor sa naredbom super(); ako klasa definiše ctor i prva naredba nije super() ili this() poziv kompajler automatski dodaje super() kao prvu naredbu (greška prilikom kompajliranja ako bazna klasa nema ctor bez argumenata).

Primer, prolazi kompajliranje:

public class Foo {

private int x;

public Foo(int x) {

this.x= x; } }

public class Bar extends Foo {

private int y;

public Bar(int x, int y) {

super(x);

this.y= y; }

public Bar(int y) {

this(0, y); } }

Primer, ne prolazi kompajliranje jer **1)** u Bar(int, int) kompajler automatski dodaje super() ali Foo nema definisan konstruktor bez parametara, a **2)** nasleđeno polje x je privatno i nije vidljivo u klasi Bar:

public class Foo {

private int x;

public Foo(int x) {

this.x = x; } }

public class Bar extends Foo {

private int y;

// ne prolazi kompajliranje!

public Bar(int x, int y) {

**this.x = x;**

this.y = y; } }

Statički atributi i metodi, *singleton* *pattern*.

06

**STATIČKI ATRIBUTI I METODI:** definisani korišćenjem ključne reči static, oni nisu vezani za objekte nego za klasu tako da im se pristupa preko imena klase, ali moguće je i preko objekta; budući da postoje nezavisno od objekata klase, statički atributi i klase postoje čak i kada klasa nije nijednom instancirana; statički metodi ne mogu pristupati nestatičkim atributima, dok nestatičke metode mogu pristupati i jednim i drugim; svaki objekat ima svoju (nezavisnu) kopiju nestatičkih atributa, a svi objekti dele iste statičke atribute (koji se ne kopiraju):

public class Ucenik {

private int id;

private String ime;

**private static int idVal;**

pub. stat. v. initID(int start) {

idVal = start; }

public Ucenik(String ime) {

**id = idVal++;**

this.ime = ime; }

... }

**SINGLETON PATTERN:** reč je o šemi u dizajnu softvera kojom se ograničava instanciranje neke klase na jedan objekat:

class Brojac {

private int val = 0;

// singleton brojac

**private static Brojac singleton = null;**

// sakrivamo konstruktor

**private Brojac() {}**

public static Brojac getInstance() {

if (singleton == null)

singleton = new Brojac();

return singleton;

}

public void inc() { val++; }

public int getVal() { return val; }

}

Korisno je kada je tačno jedan objekat potreban da koordiniše akcije u čitavom programu.

Konstruktori i nasleđivanje.

07

**KONSTRUKTORI I NASLEĐIVANJE:** odgovor pokriven odgovorom na pitanje 05 i druga.

Redefinisanje nasleđenih atributa i metoda

08

**REDEFINISANJE NASLEĐENIH ATRIBUTA I METODA:** odgovor pokriven odgovorom na pitanje 09 i druga.

Nasleđivanje, Liskov princip supstitucije i dinamičko vezivanje.

09

**NASLEĐIVANJE:** klase se mogu međusobno nasleđivati (ključna reč extends) i to tako da ako A nasleđuje B tada: **1**) A nasleđuje sve atribute i metode klase B (nasleđuju se i privatni članovi ali nisu vidljivi bez modifikatora protected), **2**) objekat klase A sadrži atribute i metode definisane u klasi B, **3**) u klasi A se mogu redefinisati nasleđeni atributi i metode (*overriding*) i naravno dodati novi, **4**) A – izvedena klasa / podklasa / klasa dete, **5**) B – bazna klasa / nadklasa / super klasa / klasa roditelj; ako A nasleđuje B tada su one u relaciji **„*is a*“** (primer: Mačka „*is a*“ Životinja); postoji direktno i indirektno nasleđivanje, a ukoliko klasa ne nasleđuje nijednu drugu ona implicitno nasleđuje Object iz java.lang; izvedena klasa može redefinisati nasleđene atribute i metode.

Nasleđivanje klasa je mehanizam ponovnog iskorišćenja koda (*code reuse*) koji omogućava: **1**) specijalizaciju – izvedena klasa dodaje nove atribute i nove metode, **2**) proširenje postojeće funkcionalnosti – izvedena klasa dodaje nove metode, **3**) modifikaciju postojeće funkcionalnosti – izvedena klasa modifikuje nasleđene metode; na ovaj način se kod iz bazne klase ne kopira, pa tako nema klonova u kodu, čije su pak posledice glomazniji programi koji se teže održavaju (primer: class Kvadrat extends Pravougaonik).

**LISKOV PRINCIP SUPSTITUCIJE:** ako klasa S nasleđuje klasu T tada objekat klase S može da se koristi gde god se očekuje objekat klase T (primer: ekran.nacrtaj(pravougaonik / kvadrat)):

T a = new T();

T b = new S();

I a i b su reference tipa T, što znači da referenca tipa T može da pokazuje na objekte klase T ali i objekte klasa izvedenih iz T (tip u vremenu kompajliranja i tip u vremenu izvršavanja):

public class Pravugaonik {

protected double a, b;

public Pravugaonik(double a, double b) {

this.a = a;

this.b = b; }

public boolean podudaranSa(P-nik p) {

return a == p.a && b == p.b; } }

p. c. Kvadrat extends Pravugaonik { ... }

=-=-=-=

Pravugaonik p = new Pravugaonik(4, 4);

Kvadrat k = new Kvadrat(4);

if (p.podudaranSa(k))

System.out.println("Podudarni");

„*Svaki kvadrat je pravougaonik, nije svaki pravougaonik kvadrat*“.

**POLIMORFIZAM I DINAMIČKO VEZIVANJE:** polimorfizam – različito ponašanje operatora ili metode (*overloaded* operatori i metodi) u zavisnosti od tipova argumenata (primer: ’+’ kao znak za sabiranje i konkatenaciju); statičko vezivanje – određivanje se vrši u vremenu kompajliranja, dinamičko vezivanje – određivanje se vrši u vremenu izvršavanja; polimorfizam u Javi – koja od preopterećenih metoda se poziva se određuje u vremenu izvršavanja na osnovu tipova referenci u vremenu izvršavanja (a ne kompajliranja):

Figure f1 = new Square();

double d1 = f1.calcSurface();

Figure f2 = new Circle();

double d2 = f2.calcSurface();

U različitim PJ postoji jednostruko i višestruko nasleđivanje (jedna klasa nasleđuje više klasa); u Javi postoji samo jednostruko nasleđivanje.

**ANOTACIJA @Override:** anotacije su dodatne informacije o klasi i delovima klase; počinju sa @ i daju se pre definicije klase, metoda, atributa; nemaju nikakav efekat na izvršavanje ali mogu biti zgodne kompajleru prilikom detekcije nekih vrsta grešaka, kao i za zanemarivanje nekih upozorenja; konkretno @Override data pre definicije metoda ukazuje da će taj metod redefinisati nasleđeni metod (ako metod takvog zaglavlja ne postoji u baznoj klasi generiše se greška u vreme kompajliranja).

**KLJUČNA REČ final:** ako je klasa definisana sa ključnom rečju final tada se ona ne može nasleđivati; ako je metod definisan sa final on se ne može redefinisati; ako je atribut definisan sa final on ne može menjati vrednost (konstanta) nakon inicijalizacije.

instanceof operator i eksplicitne konverzije referenci.

10

**instanceof OPERATOR:** u pitanju je binarni infiksni (što znači da dolazi između operanada) operator kojim proveravamo da li je objekat instanca neke klase – p instanceof C, gde je p promenljiva referencijalnog tipa i C ime klase (ime ref. tipa); rezultat je true/false; operator instanceof uzima u obzir nasleđivanje, odnosno vraća true za proveru svih klasa koje direktno ili indirektno nasleđuje klasa objekta koji se proverava.

**EKSPLICITNE KONVERZIJE REFERENCI:**

class Figura {}

class Pravougaonik extends Figura {}

class Kvadrat extends Pravougaonik {}

Figura f = new Kvadrat();

Kvadrat k = (Kvadrat) f;

Pravougaonik p = (Pravougaonik) f;

Klasa Object.

11

**KLASA Object:** klasa na vrhu hijerarhije nasleđivanja, sadrži metode:

1) boolean equals(Object o): proverava da li su dva objekta identična po sadržaju,

2) String toString(): vraća string reprezentaciju objekta, poziva se kada konkateniramo string objekat sa objektom nekog drugog tipa, automatski se poziva od S.O.P.(),

3) int hashCode(): vraća *hash* kod objekta, dva objekta identična po sadržaju moraju imati isti *hash* kod.

Apstraktne klase.

12

**APSTRAKTNE KLASE:** apstraktne klase se deklarišu ključnom reči abstract, sadrže apstraktne metode (metodi koji nisu implementirani i kod kojih je dato samo zaglavlje) i one se ne mogu instancirati (ne može se primeniti operator new); smisao apstraktnih klasa je **1**) obezbeđenje neke opšte funkcionalnosti, kao i **2**) da se specifične funkcionalnosti implementiraju izvedenim klasama, budući da se **apstraktne uvek nasleđuju** (primer: abstract class Figura koja definiše metode obim() i toString()).

Apstraktne klase dakle pored apstraktnih metoda (samo zaglavlje bez implementacije) sadrže i konkretne (sa zaglavljem i telom), ali takođe ne moraju da sadrže nijednu apstraktnu metodu (ako klasa sadrži bar jednu apstraktnu metodu ona mora biti apstraktna); hijerarhija klasa u OO programima – od apstraktnih ka specifičnim klasama (primer: niz nastavnika od kojih su neki asistenti a neki profesori).

U apstraktnim klasama implementiraju se opšte funkcionalnosti – specifične funkcionalnosti u apstraktnim klasama su apstraktne i implementiraju se tek u izvedenim klasama; opšte funkcionalnosti mogu koristiti specifične funkcionalnosti – konkretan metod iz apstraktne klase može pozvati apstraktan metod u kom slučaju se izvršava neki konkretan metod iz neke od izvedenih klasa po pravilu dinamičkog vezivanja; ako klasa nasleđuje apstraktnu klasu tada ona mora implementirati sve nasleđene apstraktne metode ili i sama mora biti apstraktna.

Modifikatori vidljivosti.

13

**MODIFIKATORI VIDLJIVOSTI:** public – klasa/interfejs vidljivi u svim paketima, član klase vidljiv za sve klase svih paketa; protected – član klase vidljiv za sve klase svog paketa i za nasleđene klase, ili interfejse iz bilo kog paketa; default (bez modifikatora) – klasa/interfejs vidljivi samo u okviru svog paketa, član klase vidljiv za sve klase u istom paketu; private – član klase vidljiv samo u okviru svoje klase.

Inicijalizacija statičkih atributa klase.

14

**STATIČKI ATRIBUTI:** mogu se inicijalizovati direktno prilikom deklaracije atributa, ili u statičkom inicijalizatoru (kojih može biti više) oblika static { .. }:

public class Foo {

priv. st. d. dvaPi = 2.0 \* Math.PI;

priv. st. d. cexp = M.log10(3.4) + dvaPi;

private static Covek[] ljudi;

static{

ljudi = new Covek[10];

for (int i= 0; i < ljudi.length; i++)

ljudi[i] = newCovek("Covek "+ i); } }

Prilikom direktne inicijalizacije statičkog atributa i u statičkom inicijalizatoru možemo koristiti samo statička polja i metode.

**STATIČKI INICIJALIZATORI:** može ih biti više, izvršavaju se u redosledu u kojem su navedeni:

public class Bar {

private static int x, y;

static {

System.out.println("Inicijalizujem x");

x = 1; }

static {

System.out.println("Inicijalizujem y");

y = 2; }

public static void main(String[] args) {

System.out.println(x+ ", "+ y); } }

Statički inicijalizatori se izvršavaju tačno jednom (pri prvom referenciranju klase, kada se class fajl učita u memoriju JVM), a služe za inicijalizaciju kompleksnih statičkih atributa klase; statički inicijalizator pristupa samo statičkim elementima klase; svi inicijalizatori se izvršavaju u redosledu u kom su navedeni.

Inicijalizacija nestatičkih atributa klase.

15

**NESTATIČKI ATRIBUTI:** mogu se inicijalizovati direktno, u konstruktoru i u nestatičkim inicijalizatorima; nestatički inicijalizatori su blok naredbe bez zaglavlja i sa njima ne treba preterivati:

public class FooBaz {

private int x, y, z;

{ System.out.println("Inicijalizator 1");

x = 10; }

public FooBaz() {

System.out.println("Konstruktor");

y = 20; }

{ System.out.println("Inicijalizator 2");

z = 30; }

public static void main(String[] args) {

new FooBaz(); } }

Nestatički inicijalizator izvršava se pri svakom instanciranju klase; pristupa i statičkim i nestatičkim elementima; svi inicijalizatori se izvršavaju u redosledu u kom su navedeni, a nestatički inicijalizatori se izvršavaju pre konstruktora klase, a nakon što se završi izvršavanje konstruktora iz bazne klase.

Ugnježdene statičke klase.

16

**UGNJEŽDENE KLASE:** u pitanju su klase definisane unutar neke druge klase kada imamo potrebu za pomoćnom klasom koja se ne koristi van te klase (povećana enkapsulacija i čitljivost koda), ili kada je klasi prirodno mesto unutar neke druge klase (npr. kod klasa koje definišu kontejner prirodno je tip elementa definisati unutar njega).

**UGNJEŽDENE STATIČKE KLASE:** razlika u odnosu na unutrašnje se ogleda u instanciranju van spoljašnje klase i mogućnostima pristupa elementima spoljašnje klase; drugi referencijalni tipovi (interfejsi i nabrojivi) takođe se mogu definisati unutar neke klase i implicitno su statički ugnježdeni tipovi.

Budući da su statički elementi nezavisni od instanci klase tako i statičke ugnježdene klase se mogu instancirati nezavisno od instanci spolje klase – tada statičku ugnježdenu klasu van spoljne referenciramo punim (kanoničkim) imenom:

public class Spoljasnja {

public static class Ugnjezdena {

public Ugnjezdena(int x) { . } }

priv. Ugnjezdena u = new Ugnjezdena(4); }

public class FooBar {

**private Spoljasnja.Ugnjezdena ug**

**= newSpoljasnja.Ugnjezdena(2);** }

Postoje takođe i puna i relativna imena ugnježdenih statičkih klasa (primer).

**VIDLJIVOST:** ako je ugnježdena statička klasa public tada je ona vidljiva koliko je vidljiva njena spoljašnja klasa, a ako je private tada nije vidljiva van spoljašnje; ako je protected tada je vidljiva u paketu najviše spoljašnje klase ako su sve njene spoljašnje klase javne ili deklarisane bez modifikatora pristupa, i u klasama izvedenim iz spoljašnje klase; ista pravila važe i za ugnježdene nestatičke klase, ugnježdene interfejse i ugnježdene nabrojive tipove; ugnježdene statičke klase mogu pristupati samo statičkim elementima spoljašnje klase, uključujući i privatne.

**NASLEĐIVANJE:** ugnježdene statičke klase se mogu nasleđivati, iako se u praksi to uglavnom ne radi (gotovo nikad van spoljašnje klase); realizuje se kao nasleđivanje običnih klasa (koristi se puno ime ako je nasleđivanje van spoljašnje klase).

Unutrašnje klase.

17

**UGNJEŽDENE NESTATIČKE (UNUTRAŠNJE) KLASE:** razlika u odnosu na ugnježdene statičke se ogleda u instanciranju van spoljašnje klase i mogućnostima pristupa elementima spoljašnje klase; dve specijalne vrste unutrašnjih (ugnježdenih nestatičkih) klasa su **1**) lokalna klasa – klasa deklarisana u nekom bloku naredbi, i **2**) anonimna klasa – singleton klasa bez imena koja se definiše prilikom instanciranja singleton objekta.

Instance unutrašnje klase su vezane za instance spoljašnje klase – unutrašnje klase se van spoljašnje klase instanciraju pozivajući operator new nad objektom spoljašnje klase:

public class Outer {

public class Inner {

public void hello(String x) {

S.O.P.(x); } }

public void m() {

Inner i = new Inner();

i.hello("Outer.m()"); } }

public class OuterInner {

public static void main(String[] args) {

Outer o = new Outer();

o.m();

Outer.Inner i = o.new Inner();

i.hello("OuterInner.main()"); } }

Još jedan primer:

public class SA {

public static class SB {

public static class SC {} } }

public class A {

public class B {

public classC {} } }

public class FooBar {

public void m() {

SA.SB sb = new SA.SB();

A.B b = new A().new B();

SA.SB.SC cs = new SA.SB.SC();

A.B.C c2 = newA().newB().newC(); } }

Instance unutrašnje klase su vezane za instance spoljašnje klase i mogu pristupati svim njenim elementima (uključujući i privatne); unutrašnja klasa može deklarisati atribut/metod istog imena kao atribut/metod iz spoljašnje klase (*shadowing*), pa se tada “zasenjenom” identifikatoru iz spoljašnje klase može pristupiti koristeći kvalifikovani this izraz oblika ImeKlase.this (u sledećem primeru C.this je referenca na atribute/metode spoljašnje klase C, a kval. izraz C.super je referenca na atribute/metode bazne klase spoljašnje klase C):

public class A {

private int x = 10;

public class B {

private int x = 20;

public class C {

private int x = A.this.x + B.this.x; } } }

U unutrašnjoj klasi nije moguće definisati statičke metode, a statička polja je moguće definisati samo ako je statičko i finalno (kod ugnježdenih statičkih nema ovih ograničenja).

Primer agregacije (klase Radnik i RadnaOrganizacija) i kompozicije (klase Cvor i ListaBrojeva) – u prvom slučaju unutrašnja klasa ima smisla bez spoljašnje, a u drugom ne.

**VIDLJIVOST:** pokriveno odgovorom na pitanje 16.

**NASLEĐIVANJE:** unutrašnje klase se takođe mogu nasleđivati, iako se u praksi to uglavnom ne radi (gotovo nikad van spoljašnje klase); kod nasleđivanja van spoljašnje klase mora se pozvati njen konstruktor nad vezanim objektom spoljašnje klase:

public class A {

public A (int x) {}

public class B {

public B (char c) {} } }

class C extends A.B {

public C (A a, char c) {

a.super(c); } }

Lokalne i anonimne klase.

18

**LOKALNE KLASE:** tekst