Objekat i klasa. Članovi objekata.

1. Šta je objekat, a šta klasa? Kreiranje objekta.

Klasa je tip, objekat je primerak tipa.

Klasom se opisuju objekti sa tim karakteristikama (podaci clanovi - atributi) i ponasanjem (metodama).

Atributi - Svaki objekat ima sopstvene vrednosti podatka clanova. Trenutne vrednosti podataka objekata cine trenutno stanje objekta.

Metodama se definisu ponasanja objekta.

Objekat je taj koji sadrzi u sebi realne vrednosti za podatke, a klasa je samo neka vrsta sablona.

Sablon podrazumeva opis sta ce svaki objekat koji pripada klasi, tj. tom tipu, morati da ima od podataka i sta ce umeti da radi tj. koje ce metode imati.

Klasa je grupa objekata koji imaju zajednicke osobine. Klasa je sablon ili nacrt po kojem se objekti kreiraju. Klasa u Javi moze sadrzati: Podatak-clan, metodu, konstruktor, blok, klasu i interfejs.

Objekat je bilo koji entitet koji ima stanje i ponasanje. npr olovka stolica.. Moze biti fizicki i logicki.

Kreiranje objekta

Klasa ime_objekta = new Konstruktor (...) --- Light lt = new Light();

Pozivamo operator **new** i navodimo **konstruktor**.

Desni deo ce dati kao rezultat rezervaciju memorijskog prostora za taj jedan objekat i taj prostor se zauzima na heap-u, ne na steku. Kada rezervise memorijski prostor i izvrsi inicijalizaciju on vrati adresu memorijske lokacije na kojoj se objekat nalazi, a lt ce da pamti tu adresu. lt nije objekat to je samo referenca na objekat. Kompajler, ako izostavimo kreiranje objekta, ce nam izbaciti gresku, jer promenljivoj nismo dodeli vrednost tj. izvrsili inicijalizaciju. Ako izostavimo liniju iznad reci ce nam da nismo promenljivoj lt dodelili vrednost pa je ne mozes koristiti.

Tacka A;

Tacka B = **new Tacka();**

A i B su na steku, a Tacka(0,0) na hipu, B sadrzi adresu nekog realnog objekta, a A ce imati vrednost NULL

double z = A.getX(); // greska, objekat ne postoji

double w = B.getX(); // OK

A = new Tacka(); A.x = 12; A.y = 3;

Sad i A pokazuje na neki objekat. tj sadrzi adresu objekta Tacka (12,3)

A = B; -- Ne kreira se novi objekat, vec A postaje referenca na vec postojeci objekat tj. A pokazuje na ono na sta pokazuje B.

Dok na objekat na koji je pokazivao A (Tacka(12,3)) nece ukazivati ni jedna referenca.

Brojevi zapisani decimalno se gledaju kao double. Da bi tacno oznacili da je to float dodamo f (1.0f)

Automatske promenljive su one promenljive koje deklarisemo unutar metoda i njih moramo da inicijalizujemo jer se one automatski ne inicijalizuju. Promenljive koje pripadaju objektima tj. koje su definisane unutar klase one ce biti inicijalizovani na null.

2. Oblast važenja (scope) referencne promenljive. Životni vek objekata.

```
{
      int x = 12; /* samo je x dostupno*/
      {
            int q = 96;      /* x i q su dostupni */
      }
      /* samo x je dostupno a q 'ne postoji' */
}
{
      int x = 12;
      {
            int x = 96; /* nepravilno*/
      }
}
```

Ne moze se deklarisati istoimena automatska promenljiva kao iz spoljnog bloka.

Referenca nakon zavrsenog bloka u kom je deklarisana izlazi iz oblasti vazenja.

Tacka A = new Tacka(); -- Kada napravimo ovako nesto, **A** je referenca tipa Tacka, napravimo objekat i upisemo njegovu adresu u A -- A (referenca) ce se nalaziti na steku, a Tacka (objekat cija adresa se nalazi u A) ce se nalaziti na hipu.

Kada funkcija zavrsi sa radom ili se zavrsi blok u okviru kog je A deklarisano, na steku referenca A nece postojati, ali ce na hipu postojati objekat.

Kada objekat više nije potreban, tj. nije referenciran ni jednom referencom onda biva automatski oslobođen garbage collector-om.

Objekte uklanja **garbage collector**. Objekat nece nestati onog trenutka kada neka promenljiva prestane da pokazuje na njega, to zavisi od toga kada ce garbage collector doci na red da izvrsi svoje poslove.

Sve reference i lokalne promenljive su na steku. Na hipu se nalaze objekti. U posebnom delu memorije se nalaze programski kodovi i komande metoda koje su definisane u raznim klasama.

Kako metod zna sa kojim objektom treba da radi? Svaki metod koji pripada objektu (koji nije static tipa) ima implicitni paramatar **this** (ovo mi ne pisemo kompajler ce sam to da odradi).

3. Koliko objekata je kreirano u datom primeru?

Šta će biti rezultat izvršavanja sledećeg koda?

```
Tacka t = newTacka(1, 1);
Tacka t2 = t;
t2.x = 12;
Console.WriteLine(t.x);.
```

-- I t i t2 pokazuju na isti objekat. tj, t2 pokazuje na ono nasta pokazuje i t.

Kreiran je samo jedan objekat.

Rezultat: 12.

4. Kako se definiše konstanta u C#? Kada joj se može definisati vrednost?

Konstanta (promenljiva koja ne menja vrednost tokom trajanja programa) se deklarise rezervisanom recju **const**. (public const int x = 2;).

Samo ugradjeni tipovi podataka mogu da budu deklarisani kao konstante. Kompleksni tipovi podataka mogu imati fiksnu vrednost samo ako se deklarisu kao **readonly (u C# ne u Javi).**

5. Da li se automatske promenljive moraju definisati?

Incijalizacija automatskih promenljivih je obavezna, tj. forsirana od strane kompajlera.

To su lokalne promenljive koje deklarisemo unutar metoda i njih moramo inicijalizovati. Kompajler nas nece pustiti da ih koristimo dok ih ne inicijalizujemo. One **nisu deklarisane unapred kao promenljive u klasi** (null, 0, false).

6. Koje su podrazumevane vrednosti za primitivne i referencne tipove atributa objekta?

Podrazumevane vrednosti za primitivne tipove su **nula (0)**, **false** ili **u0000**. Za sve primitivne **numericke promenljive to je nula, za Boolean je false**, a za clear? je u0000.

Za sve ostale, **referencne tipove,** podrazumevana vrednost je **null**.

Kompajler, interpreter, garbage collector

KOMPAJLERI I INTERPERTERI

- Program pisan u nekom od viših programskih jezika potrebno je prevesti na mašinski jezik, ne bi li bio izvršen. To prevođenje vrši **kompajler (compiler)** odgovarjućeg programskog jezika. Nakon što je program jednom preveden, program u mašinskom jeziku se može izvršiti neograničen broj puta, ali naravno, samo na određenoj vrsti računara.
- Postoji alternativa. Umesto kompajlera, koji odjednom prevodi čitav program, moguće je koristiti **interpreter**, koji prevodi naredbu po naredbu prema potrebi. Interpreter je program koji se ponaša kao CPU s nekom vrstom dobavi-i-izvrši ciklusa. Da bi izvršio program, interpreter radi u petlji u kojoj uzastopno čita naredbe iz programa, odlučuje šta je potrebno za izvršavanje te naredbe, i onda je izvršava (oni se mogu koristiti za izvršavanje mašinskog programa pisanog za jednu vrstu računara na sasvim različitom računaru).

- Projektanti Jave su se odlučili za upotrebu **kombinacije kompajliranja i interpretiranja.** Programi pisani u Javi se prevode u mašinski jezik virtuelnog računara, tzv. **Java Virtual Machine**.
- **Mašinski jezik za Java Virtual Machine se zove Java bytecode.**
- Sve što je računaru potrebno da bi izvršio Java bajt kod jeste interpreter. Takav interpreter oponaša Java virtual Machine i izvršava program.

- Svaka Java aplikacija mora sadržati barem jednu klasu sa metodom main(String[] args)
- Počinje svoje izvršavanje pozivom metoda main
- Ovako napisan program se prevodi izvršavajući **javac HelloWorld.java**
- Ako nema grešaka prevodilac **javac** kreira datoteku **HelloWorld.class** koja sadrži bytecode instrukcije za JVM.
- A pokreće se pozivom JVM uz prosledjivanje bajtkoda **java HelloWorld**

7. Da li se za prevođenje Java kodova koristi kompajler ili interpreter?

Za rad java programa potrebna je kombinacija kompajlera i interpertera. Java kod koji je pisao programmer se prvo kompajlira u byte code (.class file), a zatim JVM interpretira taj kod i izvrsava ga.

8. Šta je JVM, a šta JRE?

JVM (Java Virtual Machine) je virtuelna (**apstraktna**) **masina** (program) **koja** sluzi za izvrsavanje bytecode-a (**obezbedjuje runtime okruzenje u kojem Java bytecode moze biti izvrsen**).

JRE (Java Runtime Environment) je softver namenjen izvrsavanju Java programa tj. **obezbedjuje runtime okruzenje. To je implementacija JVM**. Sastoji se iz JVM-e, standardnih biblioteka i drugih fajlova koje JVM koristi u vreme izvrsavanja (runtime), alata za konfiguraciju,...

9. Šta je Java bajt kod?

Java bajt kod je kod nekog Java programa (JVM) koji se "nalazi" izmedju izvrsnog i masinskog koda tog programa. Dobija se iz izvornog java koda (**.java file)** preko kompajlera (npr. javac), nalazi u **.class** fajlu, a interpretira ga JVM do masinskog koda.

Da bi se izvrsio potreban je interpreter, a dobije se prevodjenjem programa pomocu kompajlera.

10. Šta je garbage collector i čemu služi?

Grabage collector je primer Deamon thread-a (nit) i uvek se izvrsava u pozadini. Glavni zadatak mu je da cisti (oslobadja) memoriju (heap) unistavanjem objekata kojima se ne moze pristupiti (npr. zbog gubljenja reference).

Deamon thread je nit niskog prioriteta koja se neprestano izvrsava u pozadini.

Kada objekat više nije potreban, tj. nije referenciran ni jednom referencom onda biva automatski oslobođen garbage collector-om.

Nije klasican proces vec nit. Ima zadatak da proverava stanje na hipu i posto za svaki objekat postoji informacija, citava tabela u kojoj se vodi evidencija o svim kreiranim objektima i tome koliko i ko ukazuje na te objekte tj. koliko referenci i promenljivih pokazuje na taj objekat i garbage collector proveri tabelu i vidi da li postoje objekti na koje niko ne ukazuje i ako takvi postoje on ih ukloni. Kada garbage collector dodje na red da izvrsi svoje poslove tada ce ukloniti te objekte.

Konstrukcija objekata

11. Šta je konstruktor? Šta je default-ni konstuktor?

Konstruktor je posebna (specijalna) vrsta metoda koji se koristi iskljucivo pri konstrukciji objekata.

Specificni metodi

- Prva i osnovna karakteristika **zovu se kao i klasa** tj ima isto ime (casesensitive - mora da isprati velika i mala slova).
- **Poziva se iskljucivo pri instanciranju objekata** (operator **new**) znaci: new Tacka();

Dozvoljeno je i new Tacka().getx(); -- poziv je u redu ako nam objekat ne treba za posle.

Pravi objekat klase Tacka i cim vrati referencu, preko nje pozivam metod getx

- **Nema povratnu vrednost.**

Klasa moze imati vise konstruktora, overloading se primenjuje i na konstruktore.

default-ni konstuktor je konstruktor bez parametara.

Ako u klasi nije definisan ni jedan konstruktor onda difoltni postoji.

Ako postoje neki nasi konstruktori onda difoltni ne postoji, ali moze da se napravi.

public void Tacka() -- Kompajler nece javiti gresku nego ce ovo tretirati kao metod Tacka. Ovo moze da napravi problem kada hocemo da koristimo difoltni konstruktor, jer ce nas kompajler obavestiti da

difoltnog konstruktora nema, jer konstruktor ne moze da ima povratnu vrednost ili void. Ako ima nesto od toga vodice se kao metoda.

12. Da li svaka klasa mora da ima konstruktor?

Svaka klasa mora da ima konstruktor, iako on nije eksplicitno naveden. U slucaju kada je kod izostavljen, default-ni konstruktor ce biti generisan.

13. Kada ne postoji default-ni kontruktor?

Default-ni konstruktor ne postoji ako je definisan neki drugi konstruktor u datoj klasi, a default-ni nije naveden.

14. Na koji način je moguće sprečiti poziv konstruktora neke klase van njene definicije? Ako je to uspešno izvedeno, kako bi rešili korišćenje objekata takve klase u aplikaciji?

Stavi se da je konstruktor private, pa se napravi static funkcija koja vraca objekte te klase???

Vidljivost konstruktora staviti da je private. Neka funkcija slicna setter funkciji, koja ce biti vidljiva (public, protected ili default zavisi gde treba da se vidi i odakle se zove) i kojoj cemo slati argumente. Ona bi pozivala kontruktor sa this (...). Getter koji bi vracao referencu na taj objekat.

15. Kako obezbediti da postoji samo jedan objekat neke klase? (Kako definisati Singleton klasu?)

Napravi se klasa sa private konstruktorom.

Definise se static metod koji vraca tip objekta date klase.

Definise se static instanca date klase.

U metodi se pita da li je instanca null, pa ako jeste poziva se konstruktor i vraca njegova povratna vrednost, a ako nije null vraca se postojeca istanca.

```
class Singleton {
        private Singleton () {...};    // Konstruktor
        private static Singleton instanca = null;
        public static Singleton instanciraj() {
                if(instanca == null)
                        instanca = new Singleton();
                return instanca;
        }
}
```

To bi se radilo koriscenjem singelton klase. Singelton klasa nije zapravo prava klasa vec dizan same klase.

Primer:

```
public class Singelton {
        private Singleton () {...};    // Konstruktor
        private static class SingletonInit{
                private static final Singleton
                referenca = new Singelton();
        }
        public static Singleton getSingleton() {
                return SingletonInit.referenca;
        }
}
```

Potrebno da klasa poseduje:

- private static atribut za kreirani objekat.
- public static metod za prosledjivanje reference na kreirani objekat.
- Inicijalizacija pri prvom koriscenju.
- Svi konstruktori protected ili private.
- Klijenti mogu samo da koriste getter reference.

Overloading - Preopterecivanje metoda

16. Šta je overloading? Po čemu se moraju razlikovati overloadovani metodi?

Overloading je mogucnost definisanja vise metoda sa istim imenom, ali razlicitim potpisom unutar jedne klase. Istoimeni metodi se moraju razlikovati po broju ili tipu (moze biti i drugaciji raspored) argumenata.

Dozvoljeno je da ovakvi metodi vracaju razlicite tipove dok god se razlikuju po broju ili tipu argumenata.

--Overloading je mehanizam koji je omogucen od strane kompajlera i ako kompajler podrzava overloading on podrzava mogucnost da se u jednoj klasi moze definisiati vise metoda sa istim imenom ali razlicitim potpisom (broj argumenata i raspored tipa tih argumenata (int,int,double je razlicito od double,int, int)).

17. Da li overloadovani metodi mogu vraćati razlicite tipove?

Jedan da vraca int drugi double?

Mogu ali se oni po tome ne razlikuju kompajler ih smatra istim ako imaju isti broj argumenata i isti tip a samo vracaju razlicite tipove kao povratne vrednosti i onda ce to biti greska.

Dozvoljeno je da vraćaju različite tipove, ali pod uslovom da se razlikuju po argumentima.

This

18. This – šta predstavlja i ko ga poseduje?

Implicitni parametar koji sadrzi adresu objekta ciji je metod pozvan.

This je rezervisana rec koja oznacava referencu trenutnog objekta date klase.

Specijalna referenca na objekat kojem se upravo pristupa. **Moze se koristiti unutar nestatickih metoda.**

This ima razna svojstva:

- Upucuje na instance objekta trenutne klase.
- Moze da sluzi za implicitni poziv metoda date klase.
- This() moze da se koristi za poziv konstruktora date klase.
- Moze da se prosledi kao argument metode date klase.
- Moze da se prosledi kao argument konstruktora date klase.
- Moze da sluzi za vracanje instance objekata date klase iz metoda.

Ko ga poseduje? -- This je referenca na datu klasu pa je svaka klasa poseduje i moze je koristiti, osim static metoda. Mora biti prva naredba u konstruktoru. Tada ne moze postojati i eksplicitni poziv super().

19. Šta se dobija pozivom this()

Pozivom this() unutar neke klase poziva se konstruktor te klase bez argumenata. Ako klasa ima vise konstruktora, onda od poslatih argumenata zavisi koji ce biti pozvan.

this(<lista argumenata>) -- Ovime se poziva kontruktor ciji argumenti odgovaraju tipu, broju i redosledu argumenata iz liste koja je data.

Static

20. Šta su članovi objekata, a šta članovi klasa?

Klasa predstavlja samo definiciju objekata, dok su objekti konkretni primerci klase.

Objekti u sebi sadrze **podatke** (opisuju stanje) i **metode** (opisuju ponasanje) koje su definisane u klasi kojoj taj objekat pripada.

Podaci clanovi objekta predstavljaju promenljive sa konkretnim vrednostima.

Da bi neki podaci odnosno metode pripadali klasi, a ne instanci te klase, koristi se rezervna rec **static** u njihovoj definiciji.

Ispred deklaracije promenljivih i metoda se može navesti static modfikator.

Static-om se obeležavju članovi zajednički svim instancama date klase.

- **Staticki podaci i metodi su podaci/metodi clanovi klase.**
- **Nestaticki podaci i metodi su podaci/metodi clanovi objekta.**

21. Kako se definišu i koriste statički podaci i metodi?

static int Mem2 -- Ideja kod statickih podataka je da ako deklarisemo static int Mem2, onda ce u memoriji postojati samo jedna jedina promenljiva Mem2, za koju se zna da je deklarisana u klasi D i svi objekti te klase ce gledati na tu jednu jedinu promenljivu, za razliku od nestatickih podataka gde svaki objekat ima svoj primerak i gde i d1 ima svoj mem1 i d2 ima svoj mem1, ali static mem2 je svima isti.

Static promenljive -- Jedna promenljiva zajednicka svim objektima klase. Tim clanovima se moze pristupiti (pod uslovom da im se moze pristupiti tj. da su dostupni za pristup iz spoljnog sveta (javni ili u istom paketu, zavisi odakle se zovu)) tj. moze im se prici **preko imena klase** D.Mem2 = 3;

To znaci da nam ne treba objekat da bi pristupili statickoj promenljivoj.

Za staticke podatke se alocira poseban memorijski prostor onog trenutka kada se ucita definicija klase.

Moze se prestupiti i preko imena objekta d1.Mem2 =3;. U c# ne moze (csarpu)

Staticki podaci i metodi se definisu pomocu rezervisane reci **static.** npr. static int x; ili static void fun();

Static-om se obelezavaju clanovi zajednicki svim instancama date klase.

Static clanovima klase se pristupa preko imena klase.

Sta znaci da metod pripada klasi, a ne objektu?

To znaci da nema nikakvu svest o objektima te klase, a to znaci da nema this. Ne dobija uopste this jer ga ne sadrzi i zato mozemo da pozovemo neki staticki metod tako sto cemo navesti ime klase i pozvati sam metod sa argumentima **(imeKlase).(imeStaticeMetode)**

npr: x = Math.sqrt(2); sqrt() je staticki metod.

Upotreba statickih metoda. Zasto su uvedeni?

U cisto objektnom jeziku kao sto je java nemamo mogucnost da definisemo nekakve nezavisne funkcije. Klasa za staticke metode glumi kontejner za tu staticku metodu (funkciju) i nista drugo.

```
class Test {
        int x;    // nestatička javna varijabla
        static int y;
        public void print(){        // nestatička metoda
                System.out.println("Hello, World");
        }
```

```
	public static void print_static() {		// statička metoda
		System.out.println("Hello, World");
	}
	public static void main(String[] args) {
		x=3;		//error
		print();	//error
		print_static();	//OK
		new Test().x=5;	//OK
		Test test=new Test(); //OK
		test.print();	//OK
	}
}
```

Posto staticki metodi ne dobijaju this, ne mozemo da upotrebimo x direktno zato sto nema this, y moze zato sto je static.

Staticki metodi mogu da koriste staticke promenljive unutar klase u kojoj su definisani, ali ne mogu da pristupe nestatickim podacima niti da pozovu nestaticke metode direktno, iskljucivo to mogu da urade tako sto u statickoj metodi napravimo objekat te klase i da nam da tu vrednost za x ili pozovi svoju funkciju print.

Treba nam numeracija svih objekata neke klase koji ce biti napravljeni (prvi redni broj 1 drugi redni broj 2...) To cemo uraditi tako sto cemo da napravimo staticku promenljuvu koja ce zapamtiti koji je prvi identifikator koji nije zauzet (pocetna vrednost je 0)

```
class Predmet {
	char oznaka;
	int id;
	static int nextID = 0;
	Predmet() {
		id = nextID;
		nextID++;
	}
	int getId() {
		return id;
	}
	char getOznaka{
		return oznaka;
	}
}
```

22. Definisanje ID objekta kao int podatka čija je vrednost jednaka rednom broju kreiranog objekta.

U klasi datog objekta definise se int ID promenljiva i jos jedna static int promenljiva (brojac) koja se postavi na nulu.

U konstruktoru date klase izvrsiti inkrementaciju static int promenljive i njenu vrednost dodeliti ID promenljivoj.

```
class Student {
	int ID;
	static int brojac = 0;
```

```
    public Student() {
        ID = brojac;
        brojac++;
    }
}
```

23. Šta svi nestatički metodi poseduju, a statički ne?

Ako govorimo o memoriji staticki metodi imaju fiksnu memoriju u RAM-u dok kod nestatickih memorija nije fiksna.

Staticki metodi ne dobijaju this tj. rerferencu na objekat klase kojem se upravo pristupa.

Zbog toga sto staticki metodi pripadaju klasi, a ne konkretnom instanciranom objektu te klase, oni ne mogu da koriste this ili super.

24. Koja su ograničenja u korišćenju static metoda?

Staticki metodi ne mogu koristiti nestaticke podatke ili pozivati nestaticke metode iste klase direktno, moze im se pristupiti samo indirektno.

Staticki metodi mogu da koriste samo staticke podatke.

Staticki metod direktno moze da pristupa samo statickim poljima i statickim metodama klase.

Staticki metodi ne mogu biti override-ovani.

25. Da li nestatički metodi definisani u klasi mogu pozivati statičke/nestatičke metode te iste klase?

Mogu ukoliko je vidljivost tih statickih metoda public, to mogu da urade i nestaticke metode van te klase.

26. Da li se u dete klasi može definisati statički metod koji ima isti potpis kao statički metod definisan u roditeljskoj klasi?

Da mogu. Staticki metodi ne mogu biti overridovani, ali se taj metod nece gledati kao metod roditelja nego kao metod za sebe.

Nece doci do greske, definisanje dva ista static metoda u srodnim klasama nece biti primer polimorfizma.

27. Da li je moguće definisati statičku klasu?

Zavisi. Samo ako su klase ugnjezdene**, unutrasnja klasa moze biti staticka**. Ako govorimo o top-level klasama onda ne, one ne mogu biti staticke.

Unutar neke klase je moguce definisati saticku klasu i to se naziva **ugnjezdena klasa.**

Nije moguce da klasa koja nije unutar druge klase bude staticka.

Instancu ugnjezdene klase ne mozemo kreirati dok nije inicijalizovana instanca klase u kojoj se ugnjezdena klasa nalazi.

Paketi

Ideja je da skup klasa koje sluze istom poslu spakujemo u takozvani paket i onda ce svaka klasa unutar tog paketa biti prepoznata ne samo na osnovu njenog imena vec i na osnovu paketa kojem pripada.

Svaka klasa u javi je deo nekog paketa, ako se nismo izjasnili o paketu, onda nasa klasa pripada difoltnom paketu, koji nije pametno koristiti, osim za testnu klasu, sve ostalo sto funkcionalno organizujemo treba smestati u pakete.

Paket je kolekcija srodnih tipova (klasa i interfejsa).

Razlozi za korišćenje paketa:

- **Naznačavanje srodnosti** određenih tipova koji se nalaze u paketu.
- **Olakšavanje pronalaženja željenog tipa** (fokusiranjem na samo jedan paket).
- **Otklanjanje potencijalnih duplikata u nazivima** (jedinstveni prostor imena).
- **Kontrolisanje pristupa** (klase u okviru istog paketa mogu da imaju neograničen pristup jedna drugoj, a spoljne ne).

Navođenje imena paketa kome klasa pripada: **package** geometry**;**

package mora biti prva naredba u fajlu (ne računajući prazne linije i komentare).

- U fajlu može postojati samo jedna deklaracija paketa.
- Jedan tip može pripadati samo jednom paketu.
- **Unutar jednog paketa ime tipa je jedinstveno**, npr. u paketu geometry pomenutog u primeru, može postojati samo jedna klasa sa imenom **Sphere**.
- Ime paketa može biti složeno, npr.**geometry.shapes3D** sadržaj ovog paketa ne mora da ima veze sa sadržajem paketa **geometry.**

UPOTREBA PAKETA

Dva načina upotreba tipova definisanih u nekom paketu:

- Navođenjem punog imena tipa: **<imepaketa>.<imetipa>**

  public class Ball {

  geometry.Sphere b = new **geometry.Sphere();**

  ...}

- Izvršiti uvoz tipa ili svih tipova paketa

  import geometry.Sphere; // ili **import geometry.*;**

  public class Ball {

  Sphere b = new Sphere();

  ...}

Kompajler i Interpreter

- Kompajler mora da zna gde da nađe import-ovane klase.
- Interpreter mora da zna gde da nađe neku klasu i njene metode.
- Kompajleru su potrebne informacije o svakom tipu koji se koristi.

U trenutku kada naiđe na naziv tipa čiju definiciju nema u tekućem fajlu, kompajler traži izvorni ili bajt kod(dovoljno mu je da jedno pronađe) u kome je definisan tip i to prvo trži u:

- Tekućem direktorijumu, pa u (moze da bude java fajl ili class fajl)
- lib direktorijumu Java Runtime Enviroment-a (JRE), a zatim u
- u tzv. user class path-u (-classpath "C:\moji paketi"), tj. korisnički definisanim putanjama do korisničkih klasa. Korisnički class path može biti pročitan na dva načina:
    - U CLASSPATH envoronment varijabli.
    - Direktno iz opcija pri kompajliranju javac -classpath "C:\moji paketi" Line.java

Prva dva su podrazumevana to ce se svakako desiti, a ako mi hocemo da dodamo jos nesto, onda cemo morati da navedemo putanju pri pozivu javac kompajlera ili navodjenjem classpath environment varijabli (vrednosti svih putanja).

Pri potrazi za definicijom tipa, javac može pronaći:

- class fajl, ali ne i izvorni (java) fajl: tada kompajler direktno koristi bajtkod koji je pronašao.
- Izvorni, ali ne i class fajl: tada kompajler kompajlira pronađeni izvorni fajl i koristi tako dobijeni bajt kod.
- Nalazi i izvorni i class fajl: tada kompajler prvo utvrđuje da li je class fajl out of date (zastareo). Ako je class fajl stariji od izvornog koda, tada izvorni kod biva kompajliran i class fajl zamenjen novim. U suprotnom, kompajler koristi postojeći bajt kod.

Sto se tice interpretera tj. pokretanja bajtkoda takodje mozemo navesti korisnicki definisane putanje do tipova.

Pokretanje pri upotrebi korisnički defnisanih paketa je takođe drugačije:

java –classpath"**.**;C:\mojipaketi" TryPackage Tacka . je tekuca putanja i mora se navesti

Jedna je tekuca, a druga je mojipaketi, ako tako nesto ne budemo uradili onda ce biti ignorisano sve sto se nalazi u tekucem direktorijumu iz kojeg pozivamo java fajl i onda nam se dize excepton.

jar sluzi tome da sve sto smo iskompajlirali i dobili od class fajlova i napravili jedan paket ili citav projekat, smestiti u jedan jedini fajl. Kada neko hoce da koristi nasu biblioteku mi je spakujemo u jar fajl i korisnik biblioteke nece morati da se bavi raspakivanjem, kompajler ce sam znati da pronadje sve tipove koji se nalaze u jar fajlu.

jar sluzi i za pakovanje i za ekstrakciju.

- Ako hocemo da pakujemo koristimo svic c - kreira se nova arhiva
- v - Znaci da ce nam pricati sta radi.
- f - Znaci da cemo navesti ime, a bez toga moze i on sam da napravi automatski.

.jar – java archive.

Jar arhive sadrže kompresovane class fajlove sa kompletno sačuvanom direktorijumskom strukturom i obezbeđuju jednostavnost u prenosu i upotrebi većeg broja korisnički definisanih paketa.

Kreiranje jedne .jar arhive: C:\Beg Java Stuff>jar cvf Geometry.jar Geometry*.class

Kreiranje .jar arhive: jar [options] [manifest] destination input-file [input-files]

Nasleđivanje (Inheritance)

Mogućnost uvođenja novih tipova/klasa proširivanjem osobina i ponašanja postojećih tipova/klasa.

- **Klasa koja nasleđuje (proširuje)** se naziva **IZVEDENA, PODKLASA ili DETE-KLASA** i predstavlja **uži tip** podatka u odnosu na klasu iz koje je izvedena.
- **Klasa koja biva proširena,** tj. čije osobine i ponašanja nasleđuje dete-klasa se naziva **SUPERKLASOM, NADKLASOM ili RODITELJSKOM KLASOM** i predstavlja **širi tip** podatka u odnosu na klasu iz koje je izvedena. Siri tip u smislu Gradjanini su i student i penzioneri... a podklasa je student zato je to uzi
- **Svaka klasa može da ima neograničen broj podklasa.**
- Podklase nisu ograničene na promenljive, konstruktore i metode klase koje nasleđuju od svoje roditeljske klase.
- **Podklase mogu dodati i neke druge promenljive i metode ili predefinisati stare metode.**
- U deklaraciji podklase navode se razlike između nje i njene superklase.
- Dete nasledjuje sve sto ima roditeljska klasa.

```
class B extends A {
      //telo klase B
}
```

Kažemo da klasa B nasleđuje klasu A, ako:

- su objekti klase B jedna vrsta objekata klase A, odnosno
- objekti klase B imaju sve osobine A (i još neke sebi svojstvene).

Dakle, objekti klase B su vrsta objekata klase A, a za vezu klase B sa klasom A kažemo da je tipa a-kind-of.

Primer: Svi koji su studenti su tipa gradjanin.

JEDNOSTRUKO NASLEĐIVANJE -- Nije dozvoljeno visestruko nasledjivanje

- **U Javi klasa može da ima samo jednu nadklasu tj. roditeljsku klasu.**
- **Sve klase** (i sistemske i naše) u Javi su direktno ili indirektno **izvedene iz klase Object.**

Ako se eksplicitno ne navede nadklasa neke klase, onda je ona implicitno izvedena iz **java.lang.Object.**

class HelloWorld extends Object { ... }

- protected Object **clone()** -- Sluzi da ako jednom objektu kazemo da se klonira taj metod vraca reference na novi objekat koji ima potpuno isto stanje kao i objekat kojem smo rekli da se klonira.
- **equals()** -- Ideja je da ako jednom objektu kazemo t1 equals t2 on poredi reference ako nista ne definisemo ali ideja je da sami napravimo pordjenje i da uporedimo stanje objekata. Za odredjeni broj klasa je napisan kako treba i zaista poredi stanje objekata ali po difoltu radi poredjenje referenci.

Kako se utvrdjuje da li se iza objekta tj. iza njegove reference krije neka klasa (npr. Tacka)?

- **Instanceof** vraca true i ako je objekat tipa Tacka ali i ako je objekat tipa Tacka2D koja je izvedena iz klase tacka. Ako nam treba precizna informacija da li objekat zaista pripada bas nekoj klasi onda koristimo metod **getClass()** koji vraca precizno kojoj klasi pripada objekat.
- **toString()** vraca string reprezentaciju objekta i on ima definiciju za klasu Object i ako nam se ta definicija ne svidja onda cemo ga prepisati. Ta definicija podrazumeva da ce nam toString metod

za bilo koji metod vratiti naziv klase kojoj taj object pripada i njegov haskod (VM ce svakom objektu pri njegovom instanciranju dodeliti jedinstven hashkod da bi vodila evidencuju na koje objekte ukazuju reference, koje sve objekte poseduje i koliko referenci pokazuje na njega da bi garbage kolektor posle mogao da ocisti)

- **hashCode()** vraca hashkod objekta.

KONSTRUKCIJA OBJEKATA

Kada se kreira objekat, njegovi atributi se postavljaju na podrazumevanu vrednost:

- **Nula (numerički)**
- **false (boolean)**
- **null (referenca)**

Podrazumevano ponasanje svakog Java kompajlera, pri izvrsavanju konstruktora neke klase se implicitno kao prva linija koda poziva difoltni konstruktor nadklase.

Objekat klase B ce sadrzati podatke koje je nasledio od A i svoje neke podatke koji su definisani samo u klasi B. Moze da se desi da inicijalizacija tih podataka nasledjenih iz klase A mogu da budu nedostupni.

Ako su nam ti podaci nedostupni onda ne mozemo nista da uradimo sa tim, i onda kompajler resava problem pozivanjem konstruktora nadklase i sve podatke koje smo nasledili sam inicijalizuje, a ja nakon toga u svom konstruktoru resim sto sam definisao u svojoj klasi.

Ako nadklasa nema konstruktor dobicemo obavestenje od kompajlera da ne postoji difoltni konstruktor klase A i nece da nas pusti da iskompajlira jer nema sta da pozove. Ako zelimo ili moramo da pozovemo konstruktor koji nije difoltni onda cemo koristiti super i navesti od argumenata sta nedifoltni konstruktor ocekuje.

Super mora da bude prva linija koda, ne moze nesto da bude ispred njega.

this je moguce koristiti za pozivanje konstruktora unutar iste klase.

Poziv konstruktora iz iste klase mora da bude prva naredba u konstruktoru ali to znaci da necemo moci da pozovemo super u tom konstruktoru, da li ce nas kompajler pustiti da to uradimo, hoce jer cemo super pozvati u ovom konstruktoru ispod. Pozivamo difoltni koji poziva nas konstruktor koji ce da pozove super.

```
class Robot{
	int rbr;
	Robot(){
		this(1);
		}
	Robo (int rbr) {
		this.rbr=rbr;
		}
	}
```

28. Šta je prepisivanje metoda? Overriding

Prepisivanje (override) je definisanje metoda u podklasi sa istim imenom, povratnom vrednoscu i argumentima kao metod u nadklasi.

Posto dete nasledjuje ponasanje roditelja, postoje situacije u kojima je potrebno da to ponasanje bude malo promenjeno, tj. potreban nam je metod od roditelja, sa malo drugacijim izvodjenjem.

Podklasa ce imati izmenjen metod iz nadklase.

Ukoliko se u podklasi definiše metoda sa istim imenom, povratnom vrednošću i argumentima kao i metoda super klase (nadklase) tada se ovom metodom prepisuje (override-uje) metoda superklase.

Za pristup originalnim metodama osnovne klase (nadklase) se koristi **super.**

```
float getPlata(){
                    return super.getPlata()+bonus;
            }
```

Ako imamo iste nazive promenljivih i u roditelj klasi i u dete klasi, onda u dete klasi postoje dve promenljive sa istim nazivom, osim sto je iz roditeljske sakrivena ali se moze pozvati tj. doci do nje sa super. Kakvu god funkciju u dete klasi da pisemo podrazumeva se da je to promenljiva iz dete klase, a drugu pozivamo sa super.imepromenljive.

Polimorfizam

29. Šta je polimorfizam? Dati primer kada se javlja polimorfno ponašanje. Uslovi za polimorfizam.

Polimorfizam je mogucnost da varijablom odredjenog tipa referencira objekte razlicitih tipova i da automatski pozivamo metode koje su specificne za tip objekta na koji varijabla referencira.

Primer: Osoba

Student Profesor

Ako imamo klasu Student i klasu Profesor koje nasledjuju klasu Osoba i ako svaka od njih ima override-ovanu metodu OglasiSe() koja stampa “Ja sam ” + tip objekta, onda:

`Osoba x = new Student();` **x** i **y** su instance tipa Osoba, ali pokazuju

`Osoba y = new Profesor();` na objekte tipa Student i Profesor.

x.OglasiSe() Dobija se izlaz: “Ja sam Student!”

y.OglasiSe() Dobija se izlaz: “Ja sam Profesor!”

Sta ovo konkretno znaci?

Iako imamo dve Osobe, one imaju tacno definisano ponasanje u zavisnosti koja je tacno vrsta osobe u pitanju. I Student i Profesor su osobe, ali svaki od njih se oglasava na svoj nacin.

Pojava kada se objekat nekog tipa ponasa drugacije u zavisnosti od situacije primer je za polimorfizam.

Uslovi za polimorfizam:

- Poziv metoda podklase kroz varijablu bazne klase (varijabla tipa Osoba zove metodu tipa Student).
- Pozvana metoda mora biti i clan bazne klase (klasa Osoba sadrzi metodu OglasiSe() kao i njene podklase).
- Signatura metode i povratni tip moraju biti isti i u baznoj i u izvedenoj klasi (metoda OglasiSe() je override-ovana kod Studenta i Profesora iz klase Osoba).
- Atribut pristupa ne sme biti restriktivniji u izvedenoj klasi nego sto je u baznoj klasi.
- Polimorfizam se odnosi samo na metode. Referenca baznog tipa moze se koristiti samo za pristup podacima baznog tipa (preko istance x ne moze se pristupiti podacima Studenta kao sto je smer, godina upisa itd. ali se moze pristupiti podacima Osobe kao sto su ime,godine,pol,...).

30. Da li se polimorfno ponašanje može primeniti na statičke metode? Dati i primer.

Ne moze! Polimorfno ponasanje se ne moze primeniti na staticke metode. Jedan od razloga je taj sto **staticke metode ne mogu da budu override-ovane**, a i staticke metode se pozivaju preko imena klase, a ne preko njene instance.

Pozivom staticke metode uvek ce se izvrsiti metoda iz klase sa cijom je instancom pozvana.

Primer: Neka imamo klase A i B koje su podklase klase C. Svaka od tih klasa ima static metodu napisiNesto() koja ispisuje ime klase. Iako su ove metode iste po signaturi i parametrima one se ne smatraju override-ovanim, niti mogu biti pozvane preko instance vec preko same klase.

Tako da poziv metoda A.napisiNesto(), B.napisiNesto() i C.napisiNesto() nisu primer za polimorfizam, jer se ne uocava polimorfno ponasanje. (eksplicitno je navedena klasa cija ce metoda biti pozvana).

Primer:

```
class Gradjanin {
	public static void jaSamStatic(){
		System.out.println("Static: Ja sam gradjanin");
	}
}
class Student extends Gradjanin {
public static void jaSamStatic() {
		System.out.println("Static: Ja sam student");
	}
}
public class test {
	public static void main(String[] args) {
		Student petar = new Student();
		Gradjanin komsija = petar;// siri tip = uzi tip

		Gradjanin.jaSamStatic();  // Static: Ja sam gradjanin
		Student.jaSamStatic();     // Static: Ja sam student

		komsija.jaSamStatic();     // Static: Ja sam gradjanin
		petar.jaSamStatic();      // Static: Ja sam student
	}
}
```

Da ovo nisu staticke metode onda bi se desio polimorfizam.

31. Pozivi kojih metoda se vezuju statički, a koji dinamički?

Poziv sa kodom koji treba da bude izvrsen. Veza poziva metoda sa telom metode se naziva povezivanje.

Staticki metodi -- Staticko vezivanje (rano povezivanje) (pri kompajliranju).

Nestaticki metodi -- Dinamicko vezivanje (kasno povezivanje) (pri izvrsavanju).

Kada je tip objekta odredjen u vremenu kompajliranja (od strane kompajlera), to je poznato kao staticko povezivanje. Ako u klasi postoji bilo koja private, final ili static metoda, postoji i staticko povezivanje.

32. Da li vidljivost metoda kojim se prepisuje roditeljski metod može biti veća/manja od vidljivosti prepisanog (roditeljskog metoda)?

Vidiljivost metoda koji se prepisuje mora biti ista ili veca u podklasi, ne sme biti manja.

33. Upotreba super.

Super je rezervisana rec koja:

- **Moze da se koristi kao referenca na objekat nadklase.**
- **Moze da sluzi za pozivanje metoda nadklase.**
- **Super() predstavlja poziv konstruktora nadklase (za poslate argumente ili bez njih (default)).**

Konstruktor svake klase mora da pozove jedan od konstruktora nadklase. Ako se taj konstruktor eksplicitno ne pozove preko super() bice pozvan implicitno default konstruktor nadklase.

Poziv super() mora biti prva komanda u konstruktoru podklase.

34. Šta se implicitno poziva kao prva komanda svakog konstruktora?

Default-ni konstruktor nadklase (super();)

Ukoliko nije eksplicitno pozvan, u svakom konstruktoru bice implicitno pozvan default konstruktor nadklase komandom super();

35. Da li privatni podatak definisan u roditeljskoj klasi postoji u objektu dete klase? Da li je moguće dobiti njegovu vrednost?

Ne postoji! Svi podaci definisani sa private postoje samo u klasi u kojoj su definisani, to jest ne nasledjuju se. Podklasa ne moze pristupiti vrednosti private podatka svoje nad klase direktno, ali moze preko public ili protected metoda tj. getera i setera nadklase (roditeljske klase).

36. Implicitna i eksplicitna konverzija referecnih tipova.

Konverzija **prosirivanja** (podtip u nadtip) se odvija **implicitno**, ne mora da se kastuje.

Osoba osoba = new Osoba();

Student student = new Student();

osoba = student; (Osoba osoba = new Student())

Konverzija **suzavanja** (nadtip u podtip) se odvija **eksplicitno** tj. mora da se kastuje.

student = (Student)osoba;

Kast je moguc ukoliko klase pripadaju istom lancu nasledjivanja, dakle, jedan tip mora da bude podtip drugog u bilo kom redosledu.

Java zadržava sve informacije o originalnoj klasi kojoj objekat pripada !!!

```
Spaniel aPet = new Spaniel("Fang");  // Kreiraj objekat tipa Spaniel
Animal theAnimal = (Animal)aPet;    // Nije potreban kast
Animal theAnimal = aPet;
Dog aDog = (Dog)theAnimal;
```

aDog se može koristiti **SAMO** za poziv overrided metoda iz klase Spaniel. Dakle, ako Spaniel ima i dodatne metode, oni neće biti dostupni sa aDog refence.

Ako je potrebno pozvati metod specifičan za Spaniel klasu koristeći referencu aDog, **NEOPHODAN** je cast aDog-a na Spaniel: **((Spaniel)aDog).specmetod();**

```
Gradjanin dobrovoljac = newGradjanin();
Student diplomac = (Student)dobrovoljac;        // run-time greska
diplomac.studiram();
```

Gradjanin je siri tip i on ne moze da zna sta se nalazi uzem tipu (Studentu) i zato dolazi do greske.

Access modifiers (vidljivost)

UČAURIVANJE ENKAPSULACIJA

Učaurivanje predstavlja mehanizam sakrivanja informacija.

Sakrivanje implementacije tipa.

Ideja je da se ne dozvoli pristupanje podacima direktno nego preko funkcija - getera i setera.

Trebalo bi da postoje za sve podatke cije bi vrednosti trebalo da budu javne.

Ne zelimo da dozvolimo da korisnik bude upoznat sa tim kakvog su tipa ti podaci unutar same klase (indeks u jednom trenutku hocu da predstavim kao string, a u drugom sa dva integera).

Proces sakrivanja informacija koje se cuvaju unutar samih objekata se naziva ucaurivanje tj. enkapsulacija.

To se izvodi tako sto se koriste modifikatori vidljivosti (pristupa) i njima se odredjuje opseg dostupnosti elemenata koda tj. iz kog dela koda mogu da vidim polja metode ili tipove.

Svaki element ima definisanu vidljivost, implicitno ili eksplicitno definisanu.

Modifikatori pristupa se mogu primeniti na:

- Tipove
- Elemente tipova – polja (podatke) i metode.

Ne mogu na varijble definisane unutar metoda. To su lokalni podaci i nebitni su za spoljni svet.

Nasledjivanje:

- Objekat podklase uvek sadrži kompletan objekat superklase klase, ali to ne znači da su svi članovi superklase dostupni metodama koje su specifične samo za podklasu!
- Nasleđivanje: uključivanje članova bazne klase u izvedenu klasu na način da su dostupni (accessible) u izvedenoj klasi.
- Nasleđeni član bazne klase je onaj koji je dostupan u izvedenoj klasi.

  - Metode koje sačinjavaju alat za komunikaciju sa spoljašnjim svetom/klasama se definišu kao public.
  - Podaci članovi ne treba da budu public osim konstanti namenjenih za opštu upotrebu.
  - Ako očekujete da će drugi ljudi koristiti vaše klase za izvođenje sopstvenih tada podatke članove definišite kao private, ali obezbedite public get-ere i set-ere.

37. Koji modifikatori vidljivosti su predviđeni u Javi? Na koji način svaki od modifikatora vidljivosti utiče na vidljivost članova objekata?

Od modifikatora vidljivosti u Javi postoje:

private -- Vidljivo samo u klasi.

protected -- Vidljivo u klasi, podklasi i paketu.

public -- **Vidljivo svuda.**

default -- **Vidljivo u klasi i paketu.** (Podrazumevan je i ispred metode ili promenljive se ne pise nista).

38. Da li se modifikatori vidljivosti mogu primeniti na statičke članove?

Da! Modifikatori vidljivosti imaju u potpunosti isto znacenje i za staticke i za nestaticke clanove objekta/klase.

39. Koja je podrazumevana vidljivost članova objekata/klasa u Javi?

Podrazumevana vidljivost je **default** (package friendly) vidljivost. Clanovi objekta/klase sa ovim modifikatorom su vidljivi u samoj klasi i u paketu, a nisu u podklasi (ako je van paketa) i van paketa.

40. Koja je podrazumevana vidljivost članova interfejsa?

Podaci u interfejsu su po default-u **public static final**, a metodi **public** i abstract.

41. Da li se modifikator pristupa/vidljivosti može primeniti na automatske promenljive?

Ne! Automatske (lokalne) promenljive postoje samo u jednom bloku (npr. metodu) i nemaju nikakve veze sa "spoljasnjim svetom", pa bi svaki modifikator vidljivosti za ovakve promenljive bio nelogican i nema potrebe (i ne moze) da ga ima.

Apstraktni metodi i tipovi

42. Kako se deklarišu apstraktni metodi?

Apstraktne metode je moguce deklarisati samo u apstraktnoj klasi. Deklarisu se pomocu rezervisane reci **abstract.**

Primer: abstract void nacrtaj(); -- **abstract tip_povratne_vred ime_metode(<argumenti>);**

Apstraktni metodi nemaju implementaciju (definisano telo funkcije), vec samo deklaraciju.

Moraju dobiti implementaciju u podklasi koja ih nasledjuje ili ce i ta klasa morati da bude apstraktna.

43. Da li se za apstraktan metod može postaviti bilo koja vrsta vidljivosti? Ako postoje ograničenja, obrazložiti.

Ne! Od svih modifikatora vidljivosti **nije dozvoljen samo private** (public, protected i default su dozvoljeni)**, jer private metode nisu polimorfne, a to u slucaju abstract klasa nema smisla** (taj private metod bi bio vidljiv samo u klasi koja je apstraktna, a ne moze da bude instancirana).

44. Šta su apstraktne klase? Navesti glavne karakteristike njene upotrebe. Dati primer definicije jedne apstraktne klase.

Apstraktne klase su klase koje se ne mogu instancirati, ali referenca ovakve klase moze postojati.

Uglavnom sadrze apstraktne metode (metode bez implementacije) (klasa mora da bude apstraktna da bi sadrzala apstraktne metode) **ali mogu sadrzati i konkretne metode koje nisu apstraktne.**

Deklaracija apstraktnih klasa: **<modifikatori> abstract class ImeKlase {...}**

Pri nasledjivanju apstraktne klase, njene podklase moraju prepisati i dati svoju implementaciju njenih apstraktnih metoda.

Apstraktne klase mogu sadrzati konstruktore koji ce biti pozvani prilikom poziva konstruktora nize podklase.

Apstraktne klase mogu imati metode tipa final (final metode se ne mogu prepisati).

Ne moze postojati objekat tipa apstraktne klase tj. ne moze se istancirati.

Primer:
```
public abstract class Figura {
        public abstract double Povrsina();
}
Apstraktna a = new Apstraktna();          // Nije dozvoljeno
Apstraktna a = b;          // Gde je b dete klasa, ovo je dozvoljeno
```

Karakteristike:

- Objedinjuje karakteristike veceg broja klasa.
- Deklarise se sa abstract.
- Ne moze biti instancirana, ali se moze prosiriti (i mora jer inace ne bi imala smisla).
- Ako klasa ima apstraktan metod, tada je i ona sama apstraktna, obrnuto ne vazi, tj. apstraktna klasa ne mora imati apstraktne metode.
- Ako podklasa nema implementirane sve nasledjene apstraktne metode onda i sama podklasa mora biti proglasena apstraktnom.
- Apstraktna klasa ne moze biti instacirana ali mogu deca klase i reference tipa apstraktne klase.

45. Da li je moguće definisati referencu tipa apstraktne klase? Na objekte kojih klasa ona može da ukazuje?

Da! Moguce je imati referencu tipa apstraktne klase, ali ne i konkretan objekat te klase.

Referenca tipa apstraktne klase moze da pokazuje na konkretne objekte podklasa te apstraktne klase.

46. Šta je potrebno uraditi da bi klasa izvedena iz apstraktne klase bila konkretna? Koje klase su konkretne?

Konkretna klasa (concrete class) je klasa koja ima implementaciju za sve svoje metode. Ovakva klasa moze biti instancirana. Drugim recima: Svaka klasa koja nije apstraktna je konkretna.

Da bi klasa koja je izvedena iz apstraktne klase bila konkretna mora da implementira sve apstraktne metode svoje (apstraktne) nadklase. Ako bilo koja apstraktna metoda ostane neimplementirana, ta klasa ce morati da bude apstraktna takodje.

Interfejsi

47. Šta su interfejsi? Navesti glavne karakteristike. Dati primer definicije interfejsa.

Interfejsi su **tipovi reference** bas kao i klase. Pripada apstraktnim tipovima.

Oni sadrze **metode** koje su implicitno **public i abstract** i **podatke** koji su implicitno **public, static i final.**

Interfejsi se ne mogu istancirati, ali mogu biti prosireni tj. izvedeni iz drugih interfejsa.

public **interface** prvi_interf **extends** drugi_interf {...}

Njihova konkretna namena je simulacija **Visestrukog nasledjivanja.**

Kada neka klasa **implementira (implements)** interfejs, tada ona mora implementirati i sve metode tog interfejsa. (kao neka vrsta ugovora kojim se klasa koja ga implementira obavezuje tj garantuje da ima odredjene metode).

Primer: public **interface** Merljivo {

int tezina();

}

class Covek **implements** Merljivo {...}

48. Da li interfejsi mogu sadržati podatke članove?

Da! Interfejsi mogu sadrzati podatke clanove**, ali su oni public, static i final. Drugim recima to su konstante.** Najcesce se ovakvi interfejsi izbegavaju.

49. Da li klasa može da implementira više interfejsa?

Da! Klase u Javi mogu da implementiraju beskonacno mnogo (figurativno receno) interfejsa, tako i uspevaju da simuliraju visestruko nasledjivanje, ali mogu naslediti (extends) samo jednu klasu.

50. Implicitna vidljivost članova interfejsa.

Metode definisane u interfejsu su implicitno **public.** Isto vazi i za podatke clanove interfejsa.

51. Da li se u interfejsu može definisati statički metod?

Da! Interfejsi mogu imati staticke metode, ali one moraju da imaju implementaciju u datom interfejsu (ne mogu biti abstract).

Ovakve metode se mogu pozivati samo pomocu imena datog interfejsa, jer pripadaju istom interfejsu.

Ovakve metode se ne mogu prepisivati u klasama koje implementiraju (nasledjuju) dati interfejs.

52. Da li je moguće definisati referencu tipa interfejsa? Na objekte kojih klasa ona može da ukazuje?

Da! Referenca tipa interfejsa se moze definisati, a ona moze da ukazuje na sve klase koje implementraju dati interfejs.

53. Razlika između apstraktnih klasa i interfejsa.

Interfejs moze da sadrzi samo apstraktne metode, dok apstraktna klasa moze i apstraktne i neapstraktne (konkretne) metode.

Svi podaci clanovi interfejsa su final i static, dok apstraktna klasa moze sadrzati podatke koji nisu final.

Interfejsi sadrze samo final i static varijable. Apstraktne klase mogu i ne final i nestaticke promenljive.

Apstraktna klasa moze da implementira interfejs, ali **interfejs ne moze** da bude izveden iz apstraktne klase tj. ne moze da **nasledi** apstraktnu ili bilo koju drugu klasu.

Klase mogu da implementiraju vise interfejsa, ali mogu da naslede samo jednu apstraktnu (ili bilo koju drugu) klasu. Interfejs moze da nasledi samo interfejs.

Apstraktne klase ne podrzavaju visestruko nasledjivanje, interfejsi podrzavaju.

Clanovi interfejsa su po default-u public, dok kod apstraktne klase mogu biti public, protected i default.

Final

54. Šta je posledica primene modifikatora final na atribut, metod, klasu?

Definicija elemenata na koji se primenjuje final je konacna!

- **Ako je promenljiva definisana kao final njena vrednost ne moze biti promenjena (postaje konstanta).**
- **Ako je klasa definisana kao final onda se ona ne moze prosiriti (naslediti).**
- **Ako je metod definisan kao final onda on ne moze biti prepisan (override).**
- **Ako je argument metode final ne moze biti menjan unutar metode.**

final modifikator može biti primenjen na klase, varijable i metode. Uopšteno, povlači za sobom tvrdnju da je definicija elementa na koji se primenjuje konačna.

♣final varijabla –varijabla predstavlja konstantu i ne menja vrednost

Inicijalizacija ne mora da bude obavljena na mestu gde je konstanta deklarisana, ali je vrednost moguće samo jednom postaviti.

♣final klasa–definicija tipa je konačna, pa klasa ne može biti proširena, tj. ne može biti nasleđena.

♣final metod–ne može biti prepisan (overrided)

♣Final može da bude i argument metoda i tada se ta varijabla ne može menjati unutar metoda

Primer static final moze da bude pi iz math klase.

OOP koncepti

55. Sta je enkapsulacija (ucaurivanje)?

Enkapsulacija je jedan od principa Objektno-orijentisanog programiranja. Ona nam govori da **podaci treba da se sakriju od spoljnih pristupa** (da moraju da budu privatni)**.** Ako neko ima potrebu da pristupi podacima, to mozemo da obezbedimo preko javnih metoda za pristup podacima (geterima i seterima).

To je mehanizam sakrivanja informacija upotrebom modifikatora vidljivosti, kojima se odredjuje opseg dostupnosti elemenata koda (tipova, atributa i metoda). Svaki element ima definisanu vidljivost implicitno ili eksplicitno.

Kada enkapsuliramo podatke i odredjene metode obezbedjujemo da nas objekat ima strogo kontrolisan pristup.

Ovim metodom poboljsavamo stabilnost aplikacije i obezbedjujemo da nam jedna klasa bude celina za sebe. Obezbedjuje se da svaki objekat bude celina za sebe i da ume da se stara o svojim podacima.

Enkapsulaciju u Javi vrsimo pomocu modifikatora vidljivosti private, protected, default.

56. Na koji način se postiže enkapsulacija u C#.

U C# kao i u ostalim OOP jezicima enkapsulacija se postize preko modifikatora vidljivosti.

Sve promenljive neke klase se postave na private, a pristup njima od spolja se kontrolise preko metoda (getera i setera), koji su obicno public.

57. Koji mehanizam u OO jezicima omogućava reusability koda?

Mehanizam **nasledjivanja (inheritance)** omogucava **code reuse.**

Nasledjivanje je mogucnost uvodjenja novih tipova/klasa prosirivanjem osobina i ponasanja postojecih tipova/klasa.

58. Overriding

Jedna od odlika OO jezika koja **omogucava podklasi da obezbedi sopstvenu implementaciju metoda koji je vec implementiran u nadklasi.**

Kad metod u podklasi ima isto ime, argumente i povratnu vrednost, kao i metod u nadklasi onda kazemo da metod u podklasi override-uje (prepisuje) metod nadklase.

Ovo je jedan od nacina za postizanje polimorfizma. Staticki metodi se ne mogu prepisati.

59. Mehanizam nasleđivanja (Inheritance).

Nasledjivanje (inheritance) je mogucnost uvodjenja novih tipova/klasa prosirivanjem osobina i ponasanja postojecih tipova/klasa. Klasa koja je izvedena nazive se **podklasa** (dete-klasa), a klasa cije se osobine nasledjuju **nadklasa** (roditeljska klasa).

Podklasa je **uzi**, a nadklasa **siri** tip.

Svaka klasa moze da ima neogranicen broj podklasa.

Podklase mogu dodati neke nove (svoje) promenljive i metode ili predefinisati stare.

U javi **klasa moze da ima samo jednu nadklasu**. (Nije moguce visestruko nasledjivanje).

Sve klase su direktno ili indirektno izvedene iz klase **Object.**

60. Da li Java podržava višestruko nasleđivanje?

Ne podrzava! Java ne podrzava visestruko nasledjivanje, ali nudi nacin da se nasledi samo ugovor i to implementiranjem interfejsa. Posto jedna klasa moze da implementira vise interfejsa mozemo da kazemo da je visestruko nasledjivanje u Javi delimicno omoguceno.

Podrzava da svaka klasa moze da ima neogranicen broj podklasa. C++ podrzava visestruko nasledjivanje.

Izuzeci

Da bi obezbedili razdvajanje koda koji se izvršava kada program teče glatko, od obrade 'neregularne' situacije, objektni jezici uvode:

1. posebnu vrstu upravljačkog bloka – try/catch/finally

2. poseban mehanizam obaveštavanja da se i šta desilo – instanciranjem objekata specijalnog tipa i njegovim prosleđivanjem onom delu koda koji je u stanju da ga "obradi".

61. Try/catch/finally – objasniti na primeru šta se i kojim redosledom izvršava.

Try/catch/finally blok sluzi da razdvoji delove koda koji se izvrsava kada program tece glatko, od obrade "neregularne" situacije.

Try/catch je posebna vrsta upravljackog bloka cija je uloga nadgledanje dela koda i definisanje obrade izuzetaka.

Ako se desi greska u try bloku, izvrsavaju se naredbe catch bloka, u suprotnom catch blok se ne izvrsava.

finally blok se uvek izvrsava, bez obzira na izhod izvrsavanja try bloka.

Ako dodje do greske u try bloku i ona bude obradjena, program nastavlja sa normalnim radom (ostatak koda se izvrsava), u suprotnom program prestaje sa radom nakon pojave greske.

Primer:

```
try {
      x.metod1(a);            //nadgledani blok
      y.metod2(b);            //nadgledani blok
}  catch (Greska g) {
      //Obrada greske
}  finally {
      //Komande
}
//Ostatak koda
```

U delu try/catch bloka

```
catch(ExceptionType1 identifier) {
      // ...
}
```

kod koji obrađuje izuzetak se poziva za **ExceptionType1 ili bilo koju njegovu podklasu.**

Ako je u nekom try/catch bloku navedeno nekoliko catch blokova sa nekoliko tipova izuzetaka u istoj klasnoj hijearhiji, potrebno je blokove postaviti tako da se prvo hvata izuzetak najniže podklase, pa redom prema najvišoj superklasi.

// neispravna sekvenca catch blokova

// neće se prevesti

```
try {
// try block code
}
catch(Exception e){ ... }
catch(ArithmeticException e){ ... }
Treba da budu obrnuto postavljeni prvo ArithmeticException pa Exception
```

FINALLY BLOK

- Koristi se za pospremanje (clean-up).
- Asociran s određenim try blokom (kao i catch blok).
- Može se koristiti i s try blokom koji sadrži kod koji ne baca nikakav izuzetak:
    - za kod s višestrukim break ili return elementima,
    - vrednosti koje vratimo s return u finally bloku će pregaziti bilo koji return izvršen u try bloku.

```
int metod(){
      try {
            //...return 1;
      }
      finally {
            return 2;
      }
      // nedohvatljiv deo koda, kompajler ne bi dozvolio
}
```

Metod iz primera vraća 2.

62. Koji su proveravani, a koji neproveravani izuzeci?

Na osnovu toga da li prevodilac insistira njihovom proveravanju ili ne, izuzeci se dele u dve grupe:

- **Proveravani izuzeci (checked)**

Oni koji su izvedeni iz klase Exception i svi koji nisu u lancu RunTimeException klase. Ako metoda baca neki proveravani izuzetak, poziv te metode mora da bude uokviren try/catch blokom koji hvata taj izuzetak, a metoda mora da bude označena ključnom reči **throws** i nazivom klase izuzetka koji baca.

- **Neproveravani izuzeci (unchecked)**

Oni koji su izvedeni iz RunTimeException. Klase koje su navedene u tabeli Javinih predefinisanih izuzetaka, uglavnom, nasleđuju klasu RunTimeException pa pripadaju grupi neproveravanih izuzetaka. Ako metoda baca neki neproveravani izuzetak, poziv te metode može, ali ne mora biti uokviren try/catch blokom koji hvata taj izuzetak.

GUI

63. Šta su Layout maneger-i i čemu služe?

Layout menager-i se koriste za rasporedjivanje grafickih komponenti na poseban nacin

64. Šta su komponente, a šta kontejneri?

65. Dati primer definisanja anonimne klase.

Anonimne klase su posebna vrsta lokalne klase koja se instancira na mestu na kom se definise, pa joj se ne navodi ime.

Anonimna klasa prosiruje drugu klasu ili implementira neki interfejs.

Anonimne klase ne mogu imati konstruktor, a ako je potreban konstruktor nadklase, iza imena apstraktne klase se navode argumenti.

Primer:

```
public static Iterator obilazak() {
        return new Iterator() {
                private int p = 0;
                public boolean LiasNext() {
                        return p < object.length;
                }
                public void next() {
                        p++;
                }
                public void remove() { }
        };
}
```

Telo anonimne klase. Ono predstavlja prosirenje na vec postojecu nadklasu Iterator.

Kada bi bio potreban poziv konstruktora nadklase sa argumentima, ti argumenti bi bili navedeni u zagradama posle imena nadklase. npr return new Iterator (int a) { ... }

Ugnježdeni tipovi

66. Koja je razlika između ugnježnenih i unutrašnjih klasa/tipova?

67. Kako se instancira objekat ugnježdene/unutrašnje klase?

68. Primer pristupa članovima objekata unutrašnje/ugnježdene klase.

69. Anonimne klase.

Niti

70. Šta su niti? Kako se mogu definisati u Javi? Dati primer.

Nit (thread) predstavlja jedno izvrsavanje nekog dela koda unutar adresnog prostora nekog procesa kreiranog unutar OS-a.

Niti su u Javi definisane klasom **Thread** u standardnoj biblioteci. Aktivan objekat (sadrzi vlastitu nit) se kreira na slican nacin: **Thread nit = new Thread();**

Posle kreiranja nit se moze konfigurisati ili pokrenuti.

Metod **start** aktivira novu nit kontrole (nit je spremna), a zavrsava se pozivom njene **run** metode.

Postoje vise stanja niti: **new (nit je kreirana), runnable (nit ceka na izvrsenje), runing (nit se izvrsava) i dead (nit je mrtva (zavrsava sa radom)).**

Drugi nacin za kreiranje niti je implementacijom interfejsa **Runnable.**

Runnable je apstrakcija koncepta aktivnog objekta - izvrsava neki kod konkurentno sa drugim takvim objektima.

Klasa Thread implementira Runnable.

Ovaj interfejs deklarise samo metod run.

Primer: preko klase Thread

```
class Multi extends Thread {
      public void run() {
            System.out.println(“Thread is running...”);
      }
}
public class Multi2 implements Runnable {
      public void run() {
             System.out.println(“Thread is running...”);
      }
}
public class Test {
      public static void main (String args[]) {
            Multi t1 = new Multi();
            t1.start();
            Multi2 m1 = new Multi2();
            Thread t2 = new Thread(m1);
            t2.start();
      }
}
```

71. Stanja niti.

Postoji vise stanja niti:

new (nit je kreirana) (Multi t1 = new Multi())

runnable (nit ceka na izvrsenje) (t1.start)

runing (nit se izvrsava) (t1.start)

dead (nit je mrtva (zavrsava sa radom)).

72. Objasniti mehanizam zaključavanja. Šta se 'zaključava'? Šta se dešava kada nit poziva sihronizovan metod objekta na koji je u trenutku poziva postavljen katanac?

Mehanizam zakljucavanja

Ako postoje dve niti koje vrse rad nad istim objektima potrebno je njihov rad sinhornizovati. Ovo se postize implementacijom koncepta **monitora.**

Svaki Java objekat poseduje jedan implicitni **katanac (lock).**

Kada neka nit zakljuca objekat (dobije njegov katanac) samo ta nit moze da pristupi objektu. Ako za to vreme druga nit zatrazi katanac istog objekta bice **blokirana** i moci ce da nastavi sa radom tek kada prva nit otpusti katanac.

Ako nit poziva **synhronized** metod za objekat koji je pod katancem, ta nit ce biti blokirana dok se dati objekat ne oslobodi.

73. Na šta se može primeniti syncronized?

Synhornized se moze primeniti na metode (staticke i nestaticke) i blokove naredbi unutar metoda.

74. Šta se dešava pozivom metoda wait(), a šta pozivom notify()?

Metod **wati()** oslobadja objekat (monitor), a sama nit koja je pozvala ovaj metod postaje blokirana.

Metod **notify()** obavestava jednu od niti koje cekaju, da nastavi sa izvrsavanjem. Kada se nit ponovo pokrene, brava se ponovo zakljuca. Ovaj metod budi samo jednu nit i to onu koja je najduze cekala.

75. Sinhronizacija i nasleđivanje.

Kada se redefinise metod u izvedenoj klasi, osobina synhronized NECE biti nasledjena.

Redefinisan metod moze biti sinhronizovan ili ne, nezavisno od odgovarajuceg metoda roditeljske klase.

Novi metod ako je nesinhronizovan nece ukinuti sinhronizovano ponavljanje metoda roditeljske klase.

76. Primer zadatka (Pogledati I zadatke sa kolokvijuma od ranijih godina)

Neka su definisana dva tipa niti, od kojih jedna koristi objekat skladišta iz kog čita njegovo stanje u obliku celobrojne vrednosti, a druga dodeljuje celobrojnu vrednost kao stanje sklasišta.

```
class Proizvodjac extends Thread {
	private Skladiste mojeskladiste;
	private int broj;
	 private int k;

	public Proizvodjac(Skladiste sk, int broj) {
		mojeskladiste = sk;
		this.broj = broj;
	}

	public void run() {
		for (int i = 0; i < 10; i++) {
			mojeskladiste.stavi(i);
		}
	}
}
class Korisnik extends Thread {
	private Skladiste mojeskladiste;
	private int broj;
```

```
	public Korisnik(Skladiste sk, int broj) {
		mojeskladiste = sk;
		this.broj = broj;
	}
	public void run() {
		int procitani_broj = 0;
		for (int i = 0; i < 10; i++) {
			procitani_broj = mojeskladiste.uzmi();
		}
	}
}
```

Definisati klasu skladište tako da bude thread-safe, onemogući istovremeni pristup/ažuriranje stanja skladišta, obezbedi naizmenično pisanje i čitanje.

Napisati telo main metoda testne klase u kom se instanciraju po dve niti korisnika i potrošača i startuju

---

IZUZECI

Resavali na OP pomocu ifova mana je sto nam se mesa kod tj delovi za regularan tok programa i delovi koji obradjuju neku gresku

U OOP-u se uvodi Try catch final blok cime se to regulise i u tom slucaju se javlja poseban vid obavestenja da se desila odgovarajuca greska tj da imamo neki neregularan deo koda

Kada dodje do greske tipa deljenje nulom Java nam ispisuje taj exception cak kaze i koja je greska i opis tog izuzetka (deljenje nulom)

Pitanje je da li je ova greska obradjena ili nije?

Ako probamo da nastavimo program to nece biti moguce

Znaci kad dodje do greske dalji tok programa se ne nastavlja i ovo je neobradjena greska

Java pomocu try catch bloka gde u try bloku pokusavamo da resimo sporni deo koda

try blok je nadgledani region

Ako imamo neki deo koda koji je problematican ili moze da nas dovede do neke greske mi ga stavljamo u nadgledani deo u okviru try bloka gde pokusavamo da izvrsimo odgovarajucu metodu ili naredbu(aritmeticku operaciju) i ukoliko moze da se izvrsi taj deo on ce se nesmetano izvrsiti a ukoliko dodje do greske catch deo sluzi da obradi tu gresku

Imamo nekoliko koraka sta se desava u tom trenutku:

Kako dolazi do obavestavanja da se desila neka greska?

Obavestenje o samoj gresci vrsi metod tokom cijeg izvrsavanja se desila ta nepredvidjena situacija i obavestenje se vrsi tako sto se generise jedan objekat specijalnog tipa a to je tip Exception gde mi na osnovu tog objekta u catch delu vrsimo obradu greske

I treci deo je preuzimanje tog obavesternja i ispisivanje odgovarajuce poruke

I u catch delu mozemo sami da napisemo sta je greska

I program je posle tog try catch dela nastavio sa programom

Pozeljno je da ne ispisujemo sami poruke sa opisom odgovarajuce greske to ce da radi sam objekat Exception koji u sebi sadrzi informacije tj metode koje ce nam omoguciti stampanje citave greske. e.printStackTrace();

printStackTrace() -- Stampa Kojoj klasi pripada napravljeni objekat pri generisanju izuzetka i koji je tip greske

Ili metoda koja nam vraca poruku o tipu greske: syso(e.getMessage()); da vidimo opis koja je greska u pitanju

Ako znamo tacno koji izuzetak postojeci jurimo ne moramo imati referencu opste klase Exception vec mozemo reci ArithemeticException posto znamo da ce taj izuzetak da se desi

Da bi kreirali nas sopstveni izuzetak moramo da napravimo klasu koja mora biti izvedena iz klase Exception

class DeljenjeNulom extends Exception{

bitno je da imamo neku poruku koja nam govori do kakve greske je doslo

}