Ovo je priča koju vrlo rado pričam,
to je priča o OOP-u…
jedni ga hvale, drugi žale, treći kažu: ”E, moj brale…!”

UPUTSTVO ZA POČETNIKE

(one koji na časovima nisu bili, ili jesu al' k'o da nisu)

verzija 1.2
14. jun 2009.

Marija Milanović

Predgovor

Tekst je zamišljen kao sredstvo za što brže osposobljavanje studenta za adekvatno praćenje nastave (npr. kad ja na času kažem: ”Dobro, to je bila klasa Sfera, a sad ćemo pisati klasu Duz, pa pošto je svaka duž određena svojim dvema krajnjim tačkama, prvo nam treba klasa Tacka”, posle 5 minuta svi dobiju po plusić za radioaktivnost na času jer su umeli da napišu klasu Tacka)

Ovo nikako nije zamena za pohađanje vežbi, niti za slajdove sa moje web-stranice!

Ako ste razumeli o čemu se radi, da počnemo...

Polazne pretpostavke

1. Seli ste da učite kod kuće za predstojeći kolokvijum

2. Računar imate, i uključen je

while (nije ispunjena pretpostavka 2.)
;

3. Instalirali ste JDK i prekopirali Eclipse na hard disk
Ako niste, možete skinuti neophodne fajlove sa web-strane www.matf.bg.ac.yu/~biljana (link OOP) ili ih prekopirati sa Desktop-a nastavničkog računara koji se nalazi u sali 718.

4. Znate da pokrenete Eclipse, napravite projekat, u njemu paket, a u njemu klasu (ako ne, nađite uputstvo kako se to radi u prezentaciji ”java2.ppt”)

5. Veliko predznanje C-a nije potrebno.
Treba znati kontrolne strukture (if, for, while, do, switch) jer u Javi imaju istu sintaksu, pa zato nije posvećeno vreme njihovom objašnjavanju.

Ovo već i vrapci oko fakulteta znaju

```
public class ZdravoSvete{

	public static void main(String[] args){

		System.out.println(”Zdravo, svete!”);

	}

}
```

Kad hoćemo nešto da ispišemo na standardni izlaz, koristimo System.out.println(), a u zagradicama pod dvostrukim navodnicima navedemo to što želimo da ispišemo.

Iz ovog primera već vidimo da ćemo imati posla sa klasama (class u prvom redu). Šta su klase?

Objekti i klase

Osnovni pojam nam je ***objekat***.

Objekat može biti bilo šta: cipela, drvo, cvet, učenik, student, osoba, životinja, pas, tačka, duž, sfera,...
Ali kad kažemo student, uvek mislimo na nekog konkretnog studenta (znamo koje mu je ime, prezime, indeks, smer...). Ili, ako kažemo cipela, znamo da li je muška ili ženska, leva ili desna, koji joj je broj, proizvođač, od kog materijala je napravljena, koje je boje, koji joj je vlasnik, kada je kupljena... Sve ove pobrojane stvari su neki atributi cipele koji je opisuju.

Dalje, kad imamo neke objekte, onda uvek želimo da ti objekti nešto rade ili da mi nešto sa njima radimo. Npr. želimo da student uči, da položi ispit, da se za vreme časa ne loguje na facebook i slično. A cipela može da se obuje, izuje, očisti, možda može i da se zaveže pertla (ako ih cipela ima)...

I na kraju, u Javi, objekat je uvek konkretan primerak neke ***klase***.
Šta to praktično znači? To znači da moramo da imamo definiciju klase da bismo mogli da pravimo objekte te klase. U definiciji klase mi kažemo koje atribute će imati objekti koje budemo pravili i šta ćemo moći sa njima da radimo.

Klasa nam dođe nešto kao šablon za objekte. Npr. klasa Ucenik opisuje koje atribute ima svaki konkretan učenik i šta on sve može da radi. Svi objekti neke klase imaće sve atribute koji su navedeni u njenoj definiciji (samo će vrednosti tih atributa biti specifične za svaki konkretan objekat) i sa svim objektima klase ćemo moći da radimo sve što u definiciji klase piše da može sa njima da se radi. I ništa više od toga.

Sa programerske tačke gledišta, **atributi** su promenljive (npr. za broj cipele imaćemo neku promenljivu tipa int, a ona može da uzima vrednosti 37, 38, i druge), dok su operacije koje mogu da se izvode nad objektima funkcije (u oo-terminologiji koristi se termin **metodi**)

Iz definicije problema koji hoćemo da rešavamo možemo i sami da odredimo šta će biti atributi klase koju pišemo. Npr. ako u zadatku kaže da radimo sa studentima, i pominje se njihovo ime, prezime, indeks i smer, jasno je da nećemo staviti da nam atribut u klasi Student bude boja očiju studenta ili broj cipela koje on nosi. Ili, ako radimo sa tačkama u 2D, atributi klase Tacka biće x- i y- koordinata, a ako radimo sa tačkama u 3D, biće nam potrebna još i z-koordinata kao atribut klase Tacka.

Šta nam treba od metoda? Tu već dolaze do izražaja neke specifičnosti Jave. Npr. ako nam za rešavanje problema treba da odredimo rastojanje između dve tačke, u klasi Tacka napisaćemo metod rastojanje() koji nam računa to rastojanje. Ali, moraćemo da pišemo i neke druge metode, npr. konstruktore, metod toString(), get*() i set*() metode... Kako se to radi i zašto, biće rečeno malo kasnije.

Postoje neka pravila igre koja moramo da savladamo da bismo pisali funkcionalne programe. Zato će u početku imati malo više teksta, a manje koda.

Statički i nestatički atributi i metodi

I atributi i metodi mogu biti statički i nestatički.

Atributi se u definiciji klase navode kao kada u C-u hoćemo da deklarišemo promenljivu. S tim da se ispred statičkih navodi i ključna reč **static**.
Mi ćemo uvek pisati još i ključnu reč **private** ispred svakog atributa (time kažemo da je samo kodu naše klase dopušteno da direktno pristupa vrednostima tih atributa i da ih menja, a ako kod neke druge klasa to pokuša, neće mu biti dopušteno. Na taj način imamo mogućnost da kontrolišemo ko sem nas može da vidi vrednosti atributa naše klase i ko može da ih menja, a kad nekome dopustimo da ih menja, možemo da proverimo da li je nova vrednost koju hoće da postavi validna za odgovarajući atribut.
Kada nam je atribut private, i hoćemo da i druge klase mogu da *čitaju* njegovu vrednost, pisaćemo public metod get*() koji služi samo da dohvati njegovu vrednost.
Kada hoćemo da dopustimo da druge klase *menjaju* vrednost našeg private atributa, pisaćemo public metod set*() koji služi samo da vrednost tog atributa postavi na neku zadatu vrednost)

Primer:

```
public class Sfera{
    private double x;  // x-koordinata centra sfere
    private double y;  // y-koordinata centra sfere
    private double z;  // z-koordinata centra sfere
    private double radius; // poluprecnik sfere
    private static int brojac;
}
```

Kog tipa mogu biti atributi (i statički i nestatički)?
Proizvoljnog primitivnog tipa (int, double, char, boolean,...), ili tipa neke klase (iz Javine biblioteke – npr. String; ili tipa neke naše klase koja je prethodno definisana; pa čak i tipa klase koju upravo definišemo).

Na mestu gde navodimo definiciju atributa, možemo da navedemo i inicijalizacionu vrednost:
npr. mogli smo da napišemo u gornjem primeru:

```
private double radius=1; // poluprecnik sfere
```

Ukoliko prilikom definisanja atributa ne navedemo inicijalizacione vrednosti, oni će biti inicijalizovani automatski na podrazumevane vrednosti koje zavise od tipa atributa. Numerički atributi (tipa int, double,...) biće inicijalizovani na 0, atributi tipa boolean na false, dok će nizovske promenljive i promenljive tipa neke klase biti inicijalizovane na **null** .

Statički atribut
Kada je atribut statički, to je nešto što je karakteristično za klasu kao celinu, ne za pojedinačne njene objekte. Na vežbama smo naveli dobar primer statičkog atributa: brojač kreiranih objekata klase. Rečeno je već da je atribut promenljiva. Kada je reč o statičkom atributu, postoji samo jedan primerak te promenljive u memoriji bez obzira koliko je objekata klase kreirano. Statički atribut postoji čak i kada nijedan objekat klase nije kreiran. Npr. u tom slučaju naš brojač treba da ima vrednost 0.

Nestatički atributi (instancne promenljive) opisuju konkretne objekte klase (to je ono sa početka: ime, prezime, indeks, smer studenta, x i y (eventualno i z) koordinata tačke ...). Svaki objekat klase koji bude kreiran imaće svoj sopstveni primerak svakog od nestatičkih atributa koji su definisani u klasi. Recimo, svaka tačka imaće svoju x i y koordinatu (tj. svoj primerak promenljive x i svoj primerak promenljive y) i one će biti nezavisne od x i y koordinata drugih kreiranih tačaka. Nestatički atributi daju objektima individualnost, oni

su ono po čemu se objekti medjusobno razlikuju (svaki objekat ima neke svoje vrednosti nestatičkih atributa).

Kakva je situacija sa metodima?

I metodi mogu biti statički i nestatički.

Oni koji su statički, mogu se pozivati i kada nije kreiran nijedan objekat klase. Statički metodi imaju to ograničenje da u svom telu mogu da koriste isključivo statičke članove klase (statičke atribute i druge statičke metode). (onda kada nije kreiran nijedan objekat, ne postoji nijedan primerak nestatičkog atributa)

Metodi koji nisu statički mogu koristiti i statičke i nestatičke članove klase. Oni se uvek pozivaju za neki konkretan objekat klase (to znači: moramo prvo da napravimo neki objekat i tek onda možemo za njega da pozovemo neki nestatički metod). Kada se u telu nestatičkog metoda koriste nestatički atributi, koriste se zapravo vrednosti onih primeraka nestatičkih atributa koji pripadaju tom objektu.

Metodi će nam gotovo uvek biti **public** (javno dostupni, tako da mozemo da ih pozivamo iz metoda drugih klasa koje budemo pisali).

Metod je funkcija:

- ima svoj povratni tip (int, double, void, String...),
- ima svoje ime
- a onda se unutar para zagradica ()
  navodi lista parametara, svaki parametar je oblika <tip><ime> a ako ih je više, razdvajaju se zapetama. Metod može da bude i bez parametara, ali () se mora navesti
- iza desne zagradice) unutar para vitičastih zagrada {} navodi se telo metoda (naredbe)

primer:

```
public static void main(){
    System.out.println(”Zdravo, svete!”);
}
```

Metodi koje će imati svaka naša klasa (sem test-klasa)

Konstruktori

Konstruktori su posebna vrsta metoda.

Svaki put kada se kreira novi objekat klase, poziva se konstruktor.

Njegova primarna svrha jeste da za objekat koji se kreira inicijalizuje vrednosti nestatičkih atributa (po potrebi može da menja i vrednosti statičkih. Ako imamo brojač kreiranih objekata, pravo mesto za njegovo uvećanje je upravo u konstruktoru jer se on poziva prilikom kreiranja novog objekta)

Ime konstruktora ne možemo sami da biramo. Konstruktor se uvek zove isto kao i klasa kojoj pripada.

Konstruktor nikada nema povratni tip, čak ni void!

U klasi možemo imati veći broj metoda koji se isto zovu, ali koji imaju različit "potpis".
Dva metoda imaju isti potpis ako se:

- isto zovu
- imaju isti broj parametara
- odgovarajući parovi parametara su im istih tipova

Za potpis metoda nije bitan njegov povratni tip, kao ni imena parametara.

Primer:
metodi

```
String  obrada(int a, String b) {}
double  obrada(int c, String d) {}
```

imaju isti potpis, jer se isto zovu, imaju po 2 parametra, prvi parametar je u oba metoda tipa int, a drugi je u oba metoda tipa String. To što prvi vraća String, a drugi double i što se prvi parametar u prvom metodu zove a, a u drugom c, i

drugi parametar prvog metoda se zove b, a drugog d, za potpis metoda nije od značaja.

Potpisi svih metoda klase moraju da se razlikuju da bi kompajler na osnovu poziva metoda mogao tačno da odredi koji metod mi želimo da pozovemo.

Recimo da smo mi definisali gornja dva metoda sa istim potpisom (to kažem hipotetički, jer nas Eclipse neće ni pustiti da to tako ostavimo, pojaviće se crvene linijice i upozorenje da imamo duplikat istog metoda) i da se negde nađe sledeći poziv:

```
obrada(10, ”Pera”);
```

Kompajler iz njega nikako ne bi mogao da dokuči da li smo mi hteli da pozovemo onaj kome se prvi argument zove a, a drugi b ili onaj kome se prvi argument zove c, a drugi d. Eto zato potpisi metoda moraju da budu različiti, i zato imena parametara nisu od značaja za potpis metoda.

Cela ova priča oko potpisa metoda ispiričana je, jer ona važi i za konstruktore. U klasi možemo imati veći broj konstruktora (svi će oni imati isto ime, tj. zvaće se isto kao i klasa) pod uslovom da nikoja dva među njima nemaju isti potpis.

Konstruktori mogu imati proizvoljan broj argumenata.

Podrazumevani konstruktor

Konstruktor se zove podrazumevani ako nema argumente.

Npr.

```
Sfera(){
    /* kreira jedinicnu sferu sa centrom u koordinatnom
     * pocetku
     */
    radius=1;
    brojac++;
   /* x, y, z ce biti automatski inicijalizovani na 0,
    * jer su numerickog tipa (double)
    */
}
```

Ako sami eksplicitno ne napišemo nijedan konstruktor, kompajler će automatski generisati podrazumevani konstruktor sa praznim telom:

```
Sfera()
{}
```

Kada napišemo bar jedan konstruktor, kompajler ne generiše podrazumevani sa praznim telom, pa ako nam je i on potreban, moramo ga sami napisati. (To bi značilo da nećemo imati podrazumevani koji u telu ima neke naredbe) (ova priča oko podrazumevanog konstruktora i toga da li ga ima ili nema i zašto će ga ovaj generisati ako nema nikakvog konstruktora, postaće mnogo jasnija kada bude bilo reči o nasleđivanju)

Unutar jednog konstruktora možemo pozvati neki drugi konstruktor iste klase korišćenjem ključne reči this umesto imena metoda, i to mora biti prva naredba u telu konstruktora:

npr.

```
Sfera(double x, double y, double z){
    this();  /* postavi nam radius na 1, i uveca brojac */
             /* this za poziv konstruktora */
    this.x=x;   /* this za pristup istoimenom atributu */
    this.y=y;
    this.z=z;
}
```

Dopušteno je parametre metoda nazvati istim imenima koja su upotrebljena za neke od atributa klase (parametri x, y i z prethodnog konstruktora, a imamo i atribute x, y i z klase). Kada napišemo samo x, misli se na parametar x metoda, a da bismo pristupili istoimenom atributu x, moramo da koristimo ključnu reč this: dakle, this.x je atribut x tekućeg objekta.

Definicija tog drugog konstruktora kojeg pozivamo pomoću ključne reči this može se u definiciji klase nalaziti i ispod definicije konstruktora u kome je taj poziv.

Ključna reč **this** za pristup atributima se može koristiti samo unutar nestatičkih metoda i predstavlja referencu na tekući objekat sa kojim taj nestatički metod barata.

this u statičkim metodima nema nikakvog smisla jer oni ne barataju pojedinačnim objektom, već isključivo statičkim atributima, pa za njih nije jasno šta bi to bio ”tekući objekat”.

Referenca

Šta je referenca?
U Javi ne postoji operator * (koji smo u C-u koristili da pristupimo onome na šta pokazuje neki pokazivač)

Ali: promenljiva tipa neke klase i objekat te klase koji je u vezi sa njom (mada promenljiva ne mora uvek da bude u vezi sa nekim objektom) su dve sasvim različite stvari.

Kada deklarišemo promenljivu tako da bude tipa neke klase, npr.

```
Sfera s;
```

mi time nismo napravili neki novi objekat s klase Sfera. Nikakav objekat tu nije kreiran niti je bilo kakva memorija izdvojena za smeštanje objekta. Samo smo rekli da je s promenljiva koja može da čuva referencu na objekat klase Sfera. Izdvojeno je jedno malecno parče memorije koje je dovoljno za čuvanje reference (adrese) objekta.

Da bismo kreirali novi objekat klase, moramo da koristimo ključnu reč **new** koju prati poziv odgovarajućeg konstruktora, npr.

```
new Sfera();
```

I šta se desi nakon ovoga?
Izdvoji se memorija neophodna za smeštanje objekta klase Sfera tj. kreira se po jedan primerak svakog od nestatičkih atributa klase: x, y, z i radius; izvrši se telo konstruktora da bi se oni inicijalizovali (telo ovog konstruktora još i uveća onaj naš statički atribut brojač) i rezultat je referenca na tako dobijeni objekat.

Tu referencu možemo da sačuvamo u promenljivoj tipa Sfera koju smo prethodno deklarisali:

```
s=new Sfera();
```

Ovim smo uspostavili vezu između promenljive s i upravo kreiranog objekta klase Sfera. Sada se u promenljivoj s nalazi referenca na taj objekat. Ako želimo da raskinemo tu vezu, možemo da kažemo:

s=null;

sada promenljiva s nije u vezi ni sa jednim objektom (ne pokazuje ni na šta)

Ili smo mogli da je povežemo sa nekim drugim objektom:

s=new Sfera(1,1,1,17);

(ovo možemo ako u klasi postoji konstruktor sa 4 argumenta)

Ako imamo i ovako nešto:

Sfera t=new Sfera(0,0,0,5);

Pa kažemo:

s=t;

time smo ono što piše u t (a to je referenca na sferu sa centrom u koordinatnom početku i poluprečnika 5) prepisali u promenljivu s, pa sada imamo 2 promenljive (s i t) koje obe referišu na isti objekat u memoriji. Ako nakon toga koristeći promenljivu s promenimo poluprečnik objekta na koji ona referiše na 7, objekat na koji referišu i promenljiva s i promenljiva t biće sfera sa centrom u koordinatnom početku i poluprečnika 7.

Kopi-konstruktor

Kada želimo da napravimo novi objekat koji je ”kopija” nekog postojećeg, zvaćemo tzv. kopi-konstruktor.
Kada kažemo ”kopija” postojećeg objekta, mislimo da će taj novi objekat u trenutku kreiranja imati vrednosti svih nestatičkih atributa jednake vrednostima odgovarajućih nestatičkih atributa objekta čija će kopija on biti.

Npr. ako imamo neku sferu sa centrom u tački (1,2,3) i poluprečnika 5, pa napravimo njenu kopiju koristeći kopi-konstruktor, dobićemo novu sferu koja će imati centar isto u tački (1,2,3) i poluprečnik će joj biti isto 5 kao u onoj polaznoj.

Ali nova sfera i polazna sfera su dva nezavisna objekta. Ako nakon kreiranja sfere-kopije, promenimo poluprečnik polazne sfere na 7 (bio je 5), poluprečnik sfere-kopije neće se promeniti (ostaće i dalje 5).

Kopi-konstruktor uvek ima samo jedan argument i on je tipa klase kojoj pripada.

Primer:

```
Sfera(Sfera s){
        x=s.x;
        y=s.y;
        z=s.z;
        radius=s.radius;
        brojac++;
}
```

public String toString()

Ako u svojoj klasi napišemo implementaciju metoda toString(), pri čemu on MORA biti public i povratni tip mu MORA biti String, moći ćemo da ”sabiramo” babe i žabe ☺

Evo o čemu se radi:

```
int x=5;
int y=4;
```

I sad ja vas pitam šta je i koliko je

```
x+y
```

Ne sumnjam da znate da je to int i ima vrednost 9.

A šta ako imamo ovako:

```
String baba=”OOP, cas ”;
int zaba=1;
```

šta će biti

```
baba+zaba
```

Sabiramo String i int!
Evo rešenija:

to će biti String ”OOP, cas 1”

Zašto???
Za svaki primitivni tip (a int je jedan od njih) u Javinoj biblioteci postoji po jedna odgovarajuća klasa, tzv. wrapper ili omotač-klasa (za tip int ta klasa iz Javine biblioteke se zove Integer) i svaka od tih wrapper-klasa za primitivne tipove ima implementiran metod toString(), koji je public i povratni tip mu je String.
I taj metod toString() za neku konkretnu vrednost primitivnog tipa vraća njenu String-reprezentaciju, tj. ako imamo vrednost 5 tipa int, metod toString() klase Integer će nam vratiti String u kome piše ”5”, ako imamo vrednost 123 tipa int, vratiće nam String u kome piše ”123” itd.

Dalje, ako imamo operator +, i oba operanda su numeričkog tipa, vršiće se najobičnije sabiranje njihovih vrednosti, što nas nije začudilo kada smo sabirali 5 i 4 i dobili 9, je l' tako?

Ako su oba operanda za operator + tipa String, vršiće se nadovezivanje drugog String-a na prvi i rezultat će biti String, npr:

String s1="vra";
String s2="bac";
s1+s2 je String u kome piše "vrabac".

I ostao je još slučaj kad je jedan operand String, a drugi primitivna vrednost. Onda se za tu primitivnu vrednost implicitno pozove metod toString() odgovarajuće wrapper-klase, tako dobijemo ekvivalentnu String-reprezentaciju te primitivne vrednosti, pa se slučaj sveo na "sabiranje" dva String-a, a videli smo da je rezultat toga String dobijen njihovim nadovezivanjem.

Lepo. A još lepše je to da kada u svojoj klasi napišemo metod public String toString() moći ćemo da sabiramo String i objekat naše klase, u kom slučaju će se implicitno pozvati metod toString() naše klase da vrati String-reprezentaciju objekta u onom obliku koji mi želimo.
Npr.
u klasi Tacka koja ima atribute x i y napišemo:

```
public String toString(){
    return "(" + x + ", " + y + ")";
}
```

i recimo da smo negde kreirali neku konkretnu tačku:

```
Tacka t=new Tacka(10,10);
```

pa nam se kasnije pojavi:

```
System.out.println("Ovo je tacka: " + t);
```

biće ispisano:

```
Ovo je tacka: (10, 10)
```

jer će implicitno biti pozvan metod toString() klase Tacka za objekat t. To je kao da smo sami napisali:

```
System.out.println("Ovo je tacka: " + t.toString());
```

get*() metodi

Rekli smo već da će nam atributi klasa uvek biti private i da to znači da kod nijedne druge klase neće moći direktno da pristupa njihovim vrednostima i da ih menja.

Ako želimo da dopustimo da kod neke druge klase može da pročita vrednost našeg private atributa, napisaćemo odgovarajući public get*() metod koji služi samo da dohvati vrednost tog atributa.

Ako se radi o nestatičkom atributu, odgovarajući get*() metod biće takođe nestatički.
get*() metod za statički atribut biće takođe statički.

get*() metod nema argumente, a povratni tip mu je isti kao tip atributa čiju vrednost dohvata. U telu mora imati return-naredbu kojom vraća vrednost atributa.

Primeri:

za našu klasu Sfera:

```
public double getX(){
        return x;
}

public static int getBrojac(){
       return brojac;
}
```

set*() metodi

Ako želimo da dopustimo da kod neke druge klase može da promeni vrednost našeg private atributa, napisaćemo odgovarajući public set*() metod koji služi samo da promeni vrednost tog atributa na neku zadatu vrednost.

Ako se radi o nestatičkom atributu, odgovarajući set*() metod biće takođe nestatički.
set*() metod za statički atribut biće takođe statički.

Povratni tip set*() metoda je void, i ima jedan argument čiji tip je isti kao tip atributa čiju vrednost menja. U telu mora imati naredbu dodele kojom se vrednost atributa postavlja na zadatu.

Primeri:

za našu klasu Sfera:

```
public void setX(double x){
     this.x=x;
}

/* ovaj nemamo u klasi Sfera
 * jer nam nije bio potreban
 */
public static void setBrojac(int b){
      brojac=b;
}
/* kod ovog metoda bi bilo malo problema ako bi se
 * argument umesto b zvao brojac
 * jer u statickom metodu ne moze se koristi kljucna
 * rec this. Onda bismo statickom clanu brojac klase
 * Sfera morali da se obratimo sa Sfera.brojac
 * dok bi nam samo brojac bio istoimeni argument
 * metoda
 */
```

Korišćenje atributa i metoda

Unutar definicije klase kada želimo da koristimo vrednost nekog atributa ili pozovemo neki metod (bez obzira da li se radi o statičkom ili nestatičkom članu) dovoljno je da prosto napišemo njegovo ime (osim za atribut u slučaju kada imamo istoimeni parametar metoda, pa onda moramo da dodamo još i this.)

Izvan definicije klase, tj. u metodima nekih drugih klasa, kada želimo da koristimo atribute i metode, pravimo razliku između statičkih i nestatičkih članova.

statički članovi:

navodimo ime klase pa tačku pa statički član kome pristupamo, npr.

```
Sfera.getBrojac();
```

ili

```
Math.PI       (pi  3.14...)
```

ili

```
Math.sqrt(4);
```

nestatički članovi:

MORAMO imati referencu na neki konkretan objekat, npr.

```
Sfera jedinicna = new Sfera();
```

korišćenje nestatičkih članova klase onda ide sa:

```
jedinicna.zapremina();
jedinicna.setRadius();
```

itd.

Ukratko o nizovima i Stringovima

Budući da je na kolokvijumu bilo problema sa korišćenjem nizova, evo par reči o tome.

NIZOVI

Niz je konačan skup podataka istog tipa koji se u memoriji čuva na uzastopnim lokacijama. Ako znamo koji je po redu podatak koji nam treba, pristupamo mu indeksiranjem. Indeksi nizova kreću od nule.

U Javi se baratanje nizovima donekle razlikuje od C-a. Naime, postoji tzv. *nizovska promenljiva* i njen je zadatak da čuva referencu na niz podataka određenog tipa. Nizovska promenljiva i sam niz čiju referencu ona čuva su dve odvojene stvari. Zašto je to nama uopšte bitno?

Pa ovako, ako želimo da radimo nešto sa nizom, za početak moramo znati kog su tipa elementi niza i moramo imati nizovsku promenljivu koja će pamtiti gde se u memoriji nalaze elementi tog niza kako bismo uopšte mogli da im pristupamo.

Npr. neka elementi budu tipa *int*. Deklarišemo nizovsku promenljivu, *a*, za čuvanje reference na niz *int*-ova na sledeći način:

int a[]; ili int []a; što je ekvivalentno.

Još uvek se ne zna koliko elemenata će imati niz i nije izdvojena nikakva memorija na koju će *a* referisati. Na ovom mestu niz NE POSTOJI! Postoji samo promenljiva *a* koja može čuvati referencu na niz *int*-ova i u ovom trenutku ona nije u vezi ni sa kakvim konkretnim nizom.

Sada kada smo obezbedili nekoga ko ume da pamti gde nam počinje niz, možemo da odvojimo parče memorije dovoljno veliko da se smeste svi elementi našeg niza i adresu početka

tog parčeta memorije zapamtimo u nizovskoj promenljivoj *a*. Da bismo to uradili, treba nam neki konkretan broj za broj elemenata niza. Npr. može korisnik da zada dimenziju niza ili da mi zadamo neko gornje ograničenje broja elemenata. Hajde recimo da korisnik unese broj elemenata sa standardnog ulaza:

```
Scanner sc=new Scanner(System.in);
System.out.println("Unesite broj elemenata niza: ");
int n=sc.nextInt();

int a[]=new int[n]; /* ovde je u jednoj naredbi
                       deklarisana nizovska promenljiva a,
                       izdvojeno parce memorije dovoljno
                       za cuvanje n int-ova i adresa
                       pocetka tog dela memorije
                       upamcena u promenljivoj a
                    */
```

Tek sada imamo niz od n *int*-ova kome možemo pristupati pomoću promenljive *a* na uobičajeni način. Npr. a[0] je prvi element tog niza, ..., a[n-1] je poslednji.

U zavisnosti od tipa elemenata, vrši se njihova automatska inicijalizacija. Elementi numeričkog tipa inicijalizuju se na 0, dok se elementi tipa neke klase inicijalizuju vrednošću *null*.

Uobičajeno je da se elementi niza dalje obrađuju u *for*-petlji, pri čemu se broj elemenata niza na koji referiše nizovska promenljiva *a* može dobiti sa *a.length*

Primer:

```
for(int i=0; i<a.length; i++)
        a[i]=i*i;
```

Kada želimo samo da prođemo kroz čitav niz bez da menjamo njegove elemente, npr. da bismo ispisali sadržaj niza ili da na osnovu vrednosti njegovih elemenata

sračunamo neku veličinu, možemo koristiti i tzv. collection based for-petlju:

```
for(int el: a)
    System.out.println(el+” ”);
```

Sintaksa collection based for-petlje je sledeća:
unutar oblih zagrada nakon ključne reči *for* navodi se tip elemenata niza - *int*, zatim neko ime za tekući element niza - *el*, dvotačka, pa ime niza kroz koji prolazimo – *a*. U telu *for*-petlje tekućem elementu niza obraćamo se imenom koje smo mu upravo dodelili (*el*).

Posebno je, sa stanovišta studenata, interesantan slučaj niza objekata neke klase. Po pravilu, na kolokvijumu je potrebno napraviti niz nekakvih objekata koji će služiti za poziv polimorfnih metoda, o čemu će više reči biti u nastavku teksta.

Stringovi

sledi popis najčešće korišćenih metoda za rad sa objektima klase *String* (paket *java.lang* Javine biblioteke)

1. Poređenje dva Stringa, metod *equals*:

```
String string1=”prvi”;
String string2=”drugi”;
string1.equals(string2);  // false

string2=”prvi”;
string1.equals(string2);  // true

string2=”Prvi”;
string1.equals(string2);  // false
```

Metod *equals* pravi razliku između velikih i malih slova.

2. Poređenje do na veličinu slova, metod *equalsIgnoreCase*:

```
String string1 = ”prvi”;
String string2 = ”PrVi”;
string1.equalsIgnoreCase(string2);  // true
```

3. Metodi *startsWith()* i *endsWith()* proveravaju da li tekući String počinje/završava se stringom koji je dat kao argument metoda

```
String string1 = ”Ovo je string”;
string1.startsWith(”Ovo”);  // true
string1.endsWith(”string”); // true
```

4. Leksikografsko poređenje Stringova, metod *compareTo()* vraća int: 0 ako su tekući i String argument jednaki,
<0 ako je tekući String leksikografski manji od argumenta i >0 inače.

string1.compareTo(string2)

5. Određivanje dužine Stringa, metod *length()*

string1.length()

6. Karakter na traženoj poziciji u Stringu, metod *charAt()*

string1.charAt(0); // prvi karakter Stringa string1
indeksi (pozicije) kreću od 0
(može izbaciti izuzetak tipa
StringIndexOutOfBoundsException)

7. Pretraga Stringova, metodi *indexOf()* i *lastIndexOf()*

pogledati prototipove u dokumentaciji, moguće je u tekućem Stringu tražiti prvu (indexOf()) i poslednju (lastIndexOf()) pojavu datog karaktera ili Stringa; od početka, odnosno kraja Stringa, kao i od neke zadate pozicije. Vraćaju odgovarajući indeks ako je traženje uspelo, a -1 inače. Upotreba je ilustrovana primerom sa vežbi koji broji pojavljivanje reči ”the” i ”and” u nekom tekstu.

8. Izdvajanje podstringa, metod *substring()*

2 verzije:

- od neke pozicije, pa do kraja Stringa

```
String mesto=”Novi Sad”;
String poslednjaRec=mesto.subString(5); // ”Sad”
```

- zadavanjem indeksa prvog karaktera podstringa i prvog karaktera iza poslednjeg karaktera podstringa:

```
String deo=mesto.subString(1,4); // ”ovi”
```

9. Odsecanje belina, metod *trim()*

odseca beline sa početka i sa kraja Stringa

```
string1.trim();
```

Najbitniji metodi **StringBuffer** klase promenljivih Stringova: (za detalje pogledati dokumentaciju)

1. dopisivanje na kraj, metod *append()*

2. umetanje, metod *insert()*

3. obrtanje niske, metod *reverse()*

4. konvertovanje u String objekat, metod *toString()*

Bitni koncepti: nasleđivanje, polimorfizam, interfejsi

Držaćemo se poznatih primera sa vežbi. Iako deluju dosta naivno, na njima je moguće ilustrovati sve što treba bez zatrpavanja nepotrebnom gomilom koda koja vas može odvući od suštine. Kada shvatite principe na kojima počivaju ovi koncepti, znate dovoljno Jave da rešavate najrazličitije probleme. Da li ćete ih rešiti? Možda da, možda ne, ali to onda zavisi isključivo od specifičnosti samog problema i vaše sposobnosti da sa tim izađete na kraj. Npr. zavisi od toga da li ćete uspeti da uočite vezu između indeksa polja na šahovskoj tabli prilikom kretanja raznih figura, da li znate kako da saberete cifre celog broja, da proverite da li je neki datum pre nekog drugog i slično, što mnogo više ima veze sa vašom moći logičkog rasuđivanja nego sa tim na kom ste primeru učili polimorfizam.

NASLEĐIVANJE

Nasleđivanje je postupak kojim se iz postojećih prave (izvode) nove klase. Nova (izvedena) i postojeća (bazna) ili superklasa nalaze se u specifičnom odnosu.

Zašto izvodimo klase iz postojećih?

Nama je glavni cilj da rešimo neki problem. Naravno, imaćemo klase i objekte jer videli smo da Java bez toga ne može.

Ti objekti mogu biti nešto apstraktno (tačke, krugovi, duži), a mogu biti i nešto opipljivije (životinje, mačke, buve,...).

I može da postoji nekakva hijerarhija među objektima i klasama sa kojima se radi. Npr. možemo imati klasu *GeometrijskiObjekat* i klase *Trougao*, *Kvadrat*, *Krug*,.... I jasno je da je trougao geometrijski objekat, kao i kvadrat i krug...

Ili npr. ako imamo klase *Zivotinja*, *Macka*, *Pas*,...
Opet, mačka je životinja, pas je takođe životinja, itd.

Kod nasleđivanja vrlo je bitna ta reč ”JE”.
Mačka ”JE” životinja.
To znači da ima sva svojstva životinje (sve atribute i ponašanja zajednička za sve životinje). Ali ima i nešto više, nešto što je svojstveno samo mačkama, nešto zbog čega je izdvojena u posebnu životinjsku vrstu (mačke predu, i slično).

U takvim situacijama, kada su objekti dvaju ili većeg broja klasa na neki način specijalizacija objekata neke klase, prirodno je da ta klasa bude bazna, a da se ove ostale izvedu iz nje. U baznoj klasi treba da se nađu atributi i metodi zajednički za objekte svih tih klasa, dok u izvedene klase treba smestiti ono po čemu se njihovi objekti međusobno razlikuju.

Kako reći da je jedna klasa izvedena iz druge?
To se čini ključnom rečju *extends* koja se navodi u prvom redu definicije klase. Npr.

```
public class Pas extends Zivotinja{

  …
}
```

Ovde se kaže da je klasa *Pas* izvedena iz klase *Zivotinja*.

To dalje znači da će svaki objekat klase *Pas*, pored atributa (i metoda) koji se nalaze u definiciji ove klase sadržati i sve atribute (i metode) definisane u baznoj klasi *Zivotinja*. Posebno je važno naglasiti da je prilikom kreiranja objekta izvedene klase (*Pas*) neophodno izvršiti inicijalizaciju i tih atributa bazne klase (*Zivotinja*) koji su prisutni u objektu izvedene klase.

```
public class Zivotinja{
  private String vrsta;

  public Zivotinja(String vrsta){
     this.vrsta=vrsta;
  }

  public String toString(){
     return “Ovo je ” + vrsta;
  }
```

```
public class Pas extends Zivotinja{
    private String ime;
    private String rasa;

    public Pas(String ime, String rasa){
        super(“Pas”);
        this.ime=ime;
        this.rasa=rasa;
    }
    public String toString(){
        return super.toString() + “ “ + ime + “, “ + rasa;
    }
}
```

(Svaki objekat klase *Pas* pored atributa *ime* i *rasa*, imaće i atribut *vrsta*)

Takođe, vrlo bitna stvar je da, iako je primer mali, zgodan i ilustrativan, ne treba to bukvalizovati i terati analogiju i gde treba i gde ne treba, pa u svakom mogućem zadatku dodavati u baznu klasu atribut tipa String koji će ”glumiti” *vrstu* i u *toString()* metod bazne klase gurati reči ”Ovo je ”. To samo pokazuje koliko ništa ne razmišljate.

Kod nasleđivanja uvodi se i pojam tzv. ***nasleđenih članova bazne klase***. To nisu svi članovi (atributi i metodi) bazne klase iako su svi oni, kao što upravo rekosmo, prisutni u objektu izvedene klase. Kada se bazna i izvedena klasa nalaze u istom paketu, ne nasleđuju se samo oni članovi koji su u baznoj klasi definisani sa *private* pristupnim atributom. Ako se pak, bazna i izvedena klasa nalaze u različitim paketima, neće biti nasleđeni ni oni članovi bazne klase koji imaju paketno pravo pristupa. (*public* i *protected* članovi nasleđuju se uvek, bez obzira na razmeštaj klasa po paketima)

A šta dobijemo kada nasledimo neki član bazne klase?
Pa to je kao da se definicija tog člana nalazi u definiciji izvedene klase. Praktično, nasleđenom članu bazne klase možemo pristupati samo navođenjem njegovog imena. On je ravnopravan sa članovima koji su definisani u samoj izvedenoj klasi.

Odavde, između ostalog, sledi: kako nam po pravilu atributi klasa (što će se odnositi i na baznu) imaju *private* pristupni atribut, oni nisu nasleđeni članovi bazne klase. Za njihovu inicijalizaciju koristi se poziv konstruktora bazne klase na način koji će uskoro biti opisan, dok ako su nam potrebne njihove vrednosti, moramo ih dohvatiti pozivom odgovarajućih *get*()* metoda bazne klase.

Može se (slučajno) desiti da postoji atribut sa istim imenom i u baznoj i u izvedenoj klasi (tip i pristupni atribut u ovoj priči nisu od značaja). Tada je atribut bazne klase skriven istoimenim atributom izvedene klase. Navođenjem samo imena atributa obraćamo se atributu izvedene klase. Da bismo referisali istoimeni nasleđeni atribut bazne klase, moramo njegovo ime kvalifikovati ključnom rečju ***super***.

Konstruktori bazne klase se nikada ne nasleđuju! To znači da ih u izvedenoj klasi ne možemo zvati na način na koji to činimo npr. u test-klasi (navođenjem njihovog imena i liste argumenata unutar oblih zagrada). Ali to ne znači da ih ne možemo zvati uopšte. Štaviše, prilikom kreiranja svakog novog objekta izvedene klase neophodno je pozvati konstruktor bazne klase koji će izvršiti inicijalizaciju atributa bazne klase prisutnih u objektu izvedene klase.
Ako ne pozovemo konstruktor bazne klase iz konstruktora izvedene, kompajler će pokušati da to uradi za nas.
Poziv konstruktora bazne klase vrši se ključnom rečju *super* kao imena metoda i to mora biti prva naredba u telu konstruktora izvedene klase (pogledati gornji primer). Ako to nije slučaj, kompajler će implicitno ubaciti poziv podrazumevanog konstruktora (*super();*), a ukoliko podrazumevani konstruktor ne postoji u baznoj klasi, to će rezultovati greškom pri kompajliranju.

Predefinisanje (overriding) metoda bazne klase

Može se definisati metod u izvedenoj klasi koji ima isti potpis kao i neki metod bazne klase. (pretpostavka je da su nama, kao i obično, metodi u klasama *public*, pa će ova priča važiti. Inače, priča važi kada pristupni atribut metoda u izvedenoj klasi nije restriktivniji od pristupnog atributa metoda bazne klase koji ima isti potpis).

Kada pozovemo metod prosto navođenjem njegovog imena, pozivaće se metod izvedene klase, a ako želimo da pozovemo baznu verziju metoda, moramo koristiti ključnu reč super. (pogledati primer, to je slučaj sa metodom toString().)

JOŠ JEDAN PRIMER NASLEĐIVANJA

Npr. neka je bazna klasa *Osoba* koja ima 2 atributa tipa *String* (*ime* i *prezime*), a iz nje izvodimo klasu *OsobaD* (poznat je još i datum rođenja osobe, i to je atribut tipa *Datum* – pretpostavka je da smo napisali klasu *Datum*).

Pogledajmo neke osnovne detalje ovih klasa:

```
public class Osoba{
   private String ime;
   private String prezime;

   public Osoba(String ime, String prezime){
      this.ime=ime;        // kada se radi o String-u, nema potrebe:  new String(ime);
      this.prezime=prezime;   // jer su String-ovi nepromenljivi objekti
   }

   public Osoba(final Osoba o){
     this(o.ime, o.prezime);
   }

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

}
```

```
public class OsobaD extends Osoba{
  private Datum datum;

  public OsobaD(String ime, String prezime, Datum datum){
       super(ime, prezime);
       this.datum=new Datum(datum);
  }

  public OsobaD(final OsobaD od){
       super(od);   // poziv kopi-konstruktora bazne klase
       // ovde se vrsi implicitno kastovanje objekta od tipa OsobaD
       // u tip bazne klase Osoba (pogledati nize odeljak o kastovanju)

       // ili: super(od.getIme(), od.getPrezime());
       // onda moraju da postoje getIme() i getPrezime() u klasi Osoba

       this.datum=new Datum(datum);
  }


  public String toString(){
       return super.toString() + ", " + datum;
  }

}
```

POLIMORFIZAM

Polimorfizam obezbeđuje da se poziv metoda ponaša različito u zavisnosti od tipa objekta na koji je primenjen.

Postoje 2 načina da se dobije polimorfno ponašanje metoda. Jedan je pomoću nasleđivanja (tu opet ima 2 varijante), a drugi pomoću interfejsa.

Nasleđivanje i polimorfizam (verzija 1)

Imamo hijerarhiju klasa (npr. *Zivotinja*, *Pas*, *Macka*, *Patka*, *Sarplaninac* iz primera sa vežbi).

Bitno je da promenljiva koja će se koristiti za poziv polimorfnog metoda bude tipa bazne klase (*Zivotinja*).
U promenljivoj tipa bazne klase može se čuvati referenca kako na objekat te klase (*Zivotinja*), tako i na objekat proizvoljne izvedene klase (*Pas*, *Macka*, *Patka*, *Sarplaninac*).
Za polimorfizam, mora se u promenljivoj tipa bazne klase čuvati referenca na objekat izvedene klase.

Metod koji će se polimorfno pozivati mora biti deklarisan i u baznoj i u izvedenim klasama za čije se objekte poziva taj metod. (kod nas u primeru to je metod *zvuk()*)

Metod u svim tim klasama mora imati isti potpis (plus u izvedenim pristupni atribut ne sme biti restriktivniji; kod nas je pristupni atribut svuda *public*, pa uslov za pristupne atribute važi)

I onda, kada se taj metod pozove, biće izvršena verzija metoda iz one klase na čiji se objekat čuva referenca u promenljivoj baznog tipa.
Pa tako, ako se u promenljivoj *ljubimac* tipa *Zivotinja* čuva referenca na objekat tipa *Macka*, poziv

```
ljubimac.zvuk();
```

uzrokovaće ispis poruke ”Mijau” (izvršava se metod *zvuk()* klase *Macka*), dok ako se čuva referenca na objekat tipa *Pas*, isti poziv

```
ljubimac.zvuk();
```

ispisaće ”Av, av!” itd.

Vratimo se, za trenutak, na priču o nizovima objekata i pogledajmo kako bi to izgledalo na ovom primeru.

```
Scanner sc=new Scanner(System.in);
System.out.println("Unesite broj objekata");
int n=sc.nextInt();
Zivotinja zivotinje[]=new Zivotinja[n];
/* Sada treba u petlji uneti odgovarajuce podatke,
 * napraviti pozivom odgovarajuceg konstruktora
 * objekat i smestiti ga u niz
 */
for(int i=0; i<zivotinje.length; i++){
  System.out.println("Unesite 1 za psa, 2 za macku");
  int izbor=sc.nextInt();

  switch(izbor){
        case 1:
              //… unose se potrebni podaci
              // za poziv konstruktora
              zivotinje[i]=new Pas(…); //poziv konstr.
              break;
        case 2:
             //… unose se potrebni podaci
             // za poziv konstruktora
             zivotinje[i]=new Macka(…);//poziv konstr.
             break;
        default:
             System.out.println("Pogresna opcija");
             i--;
      }
}
```

Ovde isto na kolokvijumu ljudi pišu svašta bez razmišljanja i udubljivanja u smisao toga što rade. Uzmu neki rešeni primer, prosto preprave malo imena klasa i promenljivih i tako pozivaju (nepostojeće) konstruktore sa neodgovarajućim brojem argumenata i/ili neodgovarajućih tipova.

Nasleđivanje i polimorfizam (verzija 2)
Apstraktne klase

U prethodnoj verziji, metod *zvuk()* je u baznoj klasi imao prazno telo, iz prostog razloga što je prisustvo tog metoda u baznoj klasi neophodno, jer to zahteva polimorfizam, a ne znamo kako bismo ga implementirali, jer ne postoji univerzalni način oglašavanja životinja.

Alternativa je da nam taj metod bude apstraktan u baznoj klasi. Čim klasa ima bar jedan apstraktni metod i sama postaje apstraktna (ključna reč *abstract* u definiciji klase i metoda).

Nije moguće praviti konkretne objekte apstraktnih klasa.

Može se definisati promenljiva tipa apstraktne klase.

Ako imamo klasu koja nasleđuje apstraktnu klasu, da bi bilo moguće praviti konkretne objekte te izvedene klase, ona mora imati implementaciju svih metoda koje je nasledila od svoje (apstraktne) bazne klase.

KLONIRANJE
(nekom drugom prilikom, preživećete i sa kopi-konstruktorom)

getClass()

Rečeno je već kod polimorfizma da se u promenljivoj tipa bazne klase može čuvati referenca na objekat tipa te klase ili proizvoljne izvedene klase.

Za svaku korišćenu klasu (i interfejs) u našem programu postoji po jedan *Class* objekat koji možemo koristiti za proveru da li se u nekoj promenljivoj čuva referenca na

objekat tipa te klase. Referenca na *Class* objekat neke klase dobija se kada se na ime te klase nadoveže ".class"

Npr.

```
Zivotinja ljubimac;
...
if(ljubimac.getClass()==Pas.class)
      System.out.println("To je pas");
```

KASTOVANJE

moguće je kastovati objekat u tip (direktne ili indirektne) natklase ili potklase, dakle samo naviše i naniže kroz neku hijerarhiju klasa.

Kastovanje naviše vrši se implicitno. To je upravo ono što imamo kod polimorfizma kada objekat tipa izvedene klase čuvamo u promenljivoj tipa bazne.

```
Zivotinja zivotinja=new Sarplaninac("Sarko");
```

Kastovanje naniže kroz hijerarhiju klasa, tj. kastovanje u tip izvedene klase mora se izvršiti eksplicitno.
Da bi kastovanje radilo, klasa u koju kastujemo mora biti stvarna klasa objekta, ili superklasa objekta!

Dakle, nakon gornje naredbe, nije dobro (izbaciće se izuzetak)

```
(Macka)zivotinja
```

a dobro je

```
(Pas)zivotinja
```

kao i

```
(Sarplaninac)zivotinja
```

Kada kastujemo naniže kroz hijerarhiju klasa?
Kastujemo onda kada smo npr. zbog polimorfizma referencu na objekat morali da čuvamo u promenljivoj tipa bazne klase, a želimo da izvršimo metod specifičan za klasu objekta (dakle metod koji nismo nasledili od bazne klase)
To ilustruje primer sa vežbi u kome imamo metod *lezeJaja()* u klasi *Patka*. Klasa *Zivotinja* je bazna za klasu *Patka* i ako se u promenljivoj *zivotinja* tipa bazne klase nalazi referenca na objekat tipa *Patka*, da bismo za taj objekat izvršili metod *lezeJaja()*, neophodno je prvo izvršiti kastovanje u tip *Patka*:

```
Zivotinja zivotinja=new Patka(”Daca”, ”pekinska”);
// pre kastovanja uvek u programu treba proveriti
// da li je to kastovanje legitimno!!!
// ovde bi trebalo da dodje ta provera
((Patka)zivotinja).lezeJaja();
```

instanceof operatorom moguće je proveriti da li je legitimno kastovanje neke promenljive u tip neke klase.
Npr.

```
Zivotinja zivotinja;
Patka patka;
...
if(zivotinja instanceof Patka){  // ovo je provera
                                 // koja gore nedostaje
    patka=(Patka)zivotinja;
    patka.lezeJaja();
}
```

instanceof vraća *true* ako je objekat referisan promenljivom istog tipa kao i desni operand ili ako je tipa proizvoljne njegove potklase. U tom slučaju, dopušteno je kastovanje promenljive (*zivotinja*) u tip desnog operanda (*Patka*).

INTERFEJSI

(samo najvažnije) (gledati primer sa vežbi)
sadrže samo deklaracije metoda, bez njihove implementacije.
Za metode se podrazumeva da su ***public*** i ***abstract*** i te ključne reči se u definiciji interfejsa za njih ne navode.

Klasa može ”implementirati” jedan ili veći broj interfejsa.

Da bismo mogli da pravimo konkretne objekte klase koja implementira neki interfejs, klasa mora sadržati implementaciju svih metoda navedenih u definiciji interfejsa. Inače, nasleđuje od interfejsa bar jedan apstraktni metod, pa je i sama apstraktna.

Moguće je deklarisati promenljivu tipa interfejsa, a u njoj se može čuvati referenca na objekat proizvoljne klase koja implementira taj interfejs.

interfejsi i polimorfizam

- u interfejsu se navedu deklaracije polimorfnih metoda
- sve klase za koje hoćemo da koriste polimorfizam napišemo tako da implementiraju taj interfejs
- deklarišemo promenljivu tipa interfejsa u kojoj čuvamo reference na objekte tipa tih klasa
- i polimorfizam funkcioniše

UGNJEŽDENE KLASE

to su klase čija se definicija nalazi unutar definicije neke druge klase

ugnježdene klase mogu biti statičke i nestatičke

ono gde mi koristimo ugnježdene klase jeste za osluškivanje događaja kada pišemo programe sa grafičkim korisničkim interfejsom i te klase su po pravilu nestatičke pa sada pričamo samo o njima

unutar nestatičkog metoda spoljašnje klase objekat nestatičke ugnježdene klase (npr. *ControlHandler*) može se kreirati pozivom njenog konstruktora na uobičajeni način, npr.

```
ControlHandler ch=new ControlHandler();
```

(statički metod spoljašnje klase ne može kreirati objekte nestatičke ugnježdene klase, ali baš nas briga, nama to kod osluškivača događaja i ne treba)

nestatička ugnježdena klasa ne može sadržati statičke članove

nestatička ugnježdena klasa ima direktan pristup svim članovima (i statičkim i nestatičkim) spoljašnje klase (može im pristupati samo navođenjem njihovog imena).

GENERIČKI TIPOVI

(ne morate prorađivati, koristićemo postojeće iz Javine biblioteke)

Java Collections Framework
(Posebno obratiti pažnju na upotrebu vektora i steka)

paket *java.util*

radi se o kolekcijama objekata proizvoljnog tipa čiji broj ne mora unapred biti poznat

Iteratori
za jedan prolazak kroz sve elemente kolekcije, od prvog do poslednjeg, može se koristiti iterator
(ima negde i primer)
Topla preporuka je da, gde god je to moguće, koristite collection-based for petlju.

Vektor (klasa Vector<>) (primer: Guzva.java)
konstruktori (najbitniji):
- podrazumevani:
  Vector<String> a=new Vector<String>();
kreira prazan vektor a String-objekata

- konstruktor koji pravi vektor od neke postojeće kolekcije objekata
(kao jedini argument prima tu kolekciju. Običan niz jeste kolekcija)

objekat se u vektor dodaje metodom *add()*

ako znamo poziciju(indeks) nekog objekta u vektoru, možemo mu pristupiti metodom *get()*
  a.get(4);
(indeksi kreću od 0)

uklanjanje objekta iz vektora vrši se metodom *remove()*

...

Sortiranje kolekcije - Collections.sort() metod

klasa Collections je iz paketa *java.util*

Prosto pozovemo ovaj metod i prosledimo mu kao argument kolekciju objekata koju želimo da sortiramo.

Neophodno je da klasa kojoj pripadaju ti objekti implementira *Comparable*<> interfejs (da ima implementaciju *compareTo()* metoda koji služi da utvrdi odnos tekućeg i objekta koji se prosledi kao argument ovog metoda). Metod *compareTo()* vraća vrednost tipa *int*!

0 ako su tekući i objekat-argument jednaki
<0 ako je tekući objekat manji od objekta-argumenta
>0 inače

(primer: isto Guzva.java)

Stek (klasa Stack<>)

osnovne operacije: stavljanje na vrh steka (*push*)
skidanje sa vrha steka (*pop*)
konstruktor: samo podrazumevani koji pravi prazan stek
Stack<Karta> spil=new Stack<Karta>();

Mešanje kolekcije – Collections.shuffle() metod

Prosto se pozove metod i prosledi mu se kao argument kolekcija koju želimo da promešamo. Ništa dodatno nije potrebno.

Primer: DeljenjeKarata
u primeru se koriste i enumeracije, pa nije zgoreg malo se pozabaviti i time.

Enumeracije

Enumeracija je konačan skup imenovanih konstanti (tipa *int* koje po default-u imaju redom vrednosti 0,1,2,...) (*Boja.java* i *Vrednost.java* primera DeljenjeKarata)

- Atribut klase može biti tipa enumeracije (klasa *Karta*)
- String reprezentacija konstante enumeracije je ime koje smo pridružili toj konstanti
- metodom *compareTo()* možemo utvrditi odnos dve konstante enumeracije
- *Boja.values()* vraća kolekciju koja sadrži sve konstante enumeracije *Boja*, pa možemo jednom collection-based for petljom proći kroz sve te konstante i obraditi ih na neki način (to koristimo u konstruktoru špila karata kako bismo generisali karte svih boja i svih vrednosti)

ČITANJE IZ FAJLA I UPIS U FAJL

Naći primere sa vežbi (!!!), u njima postoje svi neophodni komentari kako se to radi.

Znate l' priču o GUI-ju?

(uglavnom su pobrojane stvari na koje treba obratiti pažnju, nemam sad vremena sve detaljno da pišem, manje-više jasno je o čemu se radi)

GUI – grafički korisnički interfejs (prozori, dugmad, polja za unos teksta...)

paketi:

java.awt i *javax.swing*

Kao prozor aplikacije koristićemo objekat klase *JFrame* ili neke njene potklase koju ćemo izvesti iz *JFrame* i prilagoditi svojoj aplikaciji.

Za aplete se koristi potklasa klase *JApplet*.

KREIRANJE PROZORA

```
public class TestWindow{
  static JFrame prozor=new JFrame(”Naslov prozora”);

  public static void main(String[] args){
       int sirinaProzora=400;
       int visinaProzora=150;

       prozor.setBounds(50,100,sirinaProzora,visinaProzora);
       prozor.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       prozor.setVisible(true);

  }

}
```

metod *setBounds()* ima 4 argumenta: prva dva određuju koordinate gornjeg levog ugla prozora u koordinatnom sistemu ekrana (izraženo brojem piksela), a zatim slede, redom, širina i visina prozora, takođe izraženo brojem piksela.

prozor.setDefault...() uvek se stavlja za prozor aplikacije da se klikom na x u gornjem desnom uglu prozora, završi izvršavanje aplikacije na način na koji smo navikli

bez *prozor.setVisible(true)*; ništa! Ovom naredbom prozor koji smo prethodnim naredbama formirali u memoriji postaje vidljiv i na ekranu.

BITNO: Komponente za svoj GUI ne dodajemo direktno *JFrame* ili *JApplet* objektu, već u njihov content pane objekat (objekat koji predstavlja površ prozora).

referenca na content pane objekat može se dobiti pozivom metoda

getContentPane()

povratni tip metoda je *Container*.

neke bitne karakteristike komponenti:

- foreground i background color (boja kojom se npr. ispisuju slova na dugmetu, i boja pozadine dugmeta)

- font

- cursor (izgled kursora iznad površi komponente)

- enable (da li komponenta izgleda ”normalno” ili sivo)

- visible (vidljivost komponente na ekranu)

isVisible(), *isEnabled()*
setVisible(), *setEnabled()*

setPreferredSize()

sredstvo za određivanje veličine ekrana

```
import java.awt.Toolkit;
import java.awt.Dimension;
...
Toolkit theKit=prozor.getToolkit();
Dimension velicinaProzora=theKit.getScreenSize();
```

pogledati primer u celini koji ilustruje centriranje prozora na ekranu

VIZUELNE KARAKTERISTIKE KOMPONENTI

void setBackground(Color color)
Color getBackground()
void setForeground(Color color)
Color getForeground()
void setFont(Font font)
Font getFont()
void setCursor(Cursor cursor)

BOJA

java.awt.Color

RGB model se koristi

konstante:

- Color.WHITE
- Color.PINK
- Color.BLACK
- Color.BLUE
- Color.GREEN
- Color.GRAY itd.

int getRGB()
vraća vrednost tipa *int* koja je kombinacija komponenata boje

2 boje se porede na jednakost
metodom equals()
ili sa
if(bojaA.getRGB()==bojaB.getRGB())

KURSOR

java.awt.Cursor

lista konstanti kojima je određen izgled kursora može se naći na jednom od slajdova prve prezentacije sa vežbi koja se bavi GUI-jem

DEFAULT_CURSOR
WAIT_CURSOR
HAND_CURSOR ...

KREIRANJE KURSORA

pozivom konstruktora:

```
Cursor myCursor=new Cursor(Cursor.TEXT_CURSOR);
```

alternativni način, pozivom statičkog metoda:

```
Cursor myCursor=Cursor.getPredefinedCursor(Cursor.TEXT_CURSOR);
```

primer:

```
import java.awt.Color;
import java.awt.Cursor;
...
prozor.setCursor(Cursor.getPredefinedCursor(Cursor.TEXT_CURSOR);
prozor.getContentPane().setBackground(Color.PINK);
```

FONT

Font myFont=new Font(”Serif”, Font.ITALIC, 12);

prvi argument: **ime fonta** (logičko ili fizičko,
preporuka:
logičko koristiti.
Opcije: Serif, Dialog, DialogInput, Monospaced, SansSerif)

drugi: **stil** (ITALIC, PLAIN, BOLD,
može i Font.ITALIC+Font.BOLD)

treći: **veličina** (broj piksela)

getStyle()
getSize()

isPlain() *isBold()* *isItalic()*

SWING-KOMPONENTE, NABRAJANJE KLASA KOJE SU NAM OD ZNAČAJA

JButton – obično dugme
JCheckBox – čekboks
JRadioButton – radio-dugme (radio-dugmad se koriste u grupi, u svakom trenutku tačno jedno radio-dugme iz grupe je selektovano. Objekat koji vodi računa o tome je objekat tipa ***ButtonGroup***. To je objekat koji nema grafičku reprezentaciju. Svu radio-dugmad koja pripadaju istoj grupi je, osim u content-pane potrebno dodati i u *ButtonGroup* objekat)

JLabel labela (pasivna komponenta, korisnik je ne može editovati)
JTextField polje za unos jednog reda teksta
JTextArea polje za unos većeg broja redova teksta
(ako se želi omogućiti skrolovanje, koristi se u kombinaciji sa *JScrollPane* objektom na način koji se može videti u rešenju jednog od zadataka sa drugog kolokvijuma prošle godine (Osobe))

JPanel često korišćena komponenta za razmeštanje komponenti po površi prozora na željeni način

Svakoj komponenti može se dodati granica (**border**) radi lepšeg vizuelnog utiska. Postoji nekih 8 vrsta granica.

komponente se dodaju metodom ***add()***.

layout manager-i
(to su objekti koji kontrolišu raspored dodatih komponenti)

FlowLayout – komponente se dodaju po redovima (kao reči kada se piše u nekom editoru, npr. Notepad-u); u novi red se prelazi kada se popuni tekući
(default za JPanel objekat)

BorderLayout – 4 strane sveta, plus centralno polje (dakle, može smestiti samo 5 komponenti (ali te kompoenente mogu biti npr. tipa *JPanel*)
(default za JFrame i JApplet objekte)

CardLayout – sve komponente se smeštaju na stek i u svakom trenutku vidljiva je samo ona sa vrha steka

GridLayout – mreža (vrste i kolone)

BoxLayout – vrsta ili kolona

Layout manager-i automatski podešavaju veličinu i pozicije komponenata tako da se uklope u raspoloživi prostor.

layout manager objekat (za kontejnerski objekat, a kontejnerski je onaj koji može sadržati u sebi druge komponente) postavlja se pozivom metoda

setLayout()

```
FlowLayout flow=new FlowLayout();
prozor.getContentPane().setLayout(flow);
```

FlowLayot manager (ima primerčić sa vežbi)
po default-u, sadržaj redova je centriran

a komponente su po horizontali i po vertikali udaljene jedne od drugih za po 5 piksela

to se dobije pozivom podrazumevanog konstruktora FlowLayout()

postoji konstruktor sa 3 argumenta: poravnanje redova, horizontalni i vertikalni razmak izmedju komponenti

setHgap(), setVgap()

BorderLayout (opet ima primerčić)

CardLayout (primerčić...)
za ovaj primer neophodno je znanje obrade događaja klika na dugme (addActionListener()...)

a za objašnjenje kako se koristi CardLayout pročitati nekoliko slajdova sa kraja prve prezentacije sa vežbi koja se bavi GUI-jem

GridLayout (primerčić...) i poslednji slajd te prezentacije

SKETCHER-prvi primer

- definisanje prozora kao objekta naše klase izvedene iz *JFrame*

APLETI

Java-programi koji se izvršavaju u kontekstu web-browser-a. To nameće ozbiljna ograničenja na ono što oni smeju da rade.

...

aplet ne može imati pristup fajlovima na lokalnom računaru

...

potklasa klase **JApplet**

public void init();

smatrajte: što je main() za aplikaciju, to je init() za aplet u smislu: ono što biste u aplikaciji stavili u main(), to kod apleta stavite u init().

OBRADA DOGAĐAJA

događaj je kada npr. korisnik klikne na dugme ili pritisne neku tipku tastature...
mi želimo da isprogramiramo kod koji će se izvršavati kada se desi neki događaj

koristićemo mehanizam tzv. osluškivača događaja

uvek postoji objekat koji je **izvor događaja** (u našem primeru to je dugme)

kada se desi događaj, generiše se objekat koji predstavlja taj **događaj** (kada se radi o kliku na dugme, taj događaj je tipa ***ActionEvent***)

taj objekat se prosleđuje objektu koji će odreagovati na događaj (to je objekat koji se zove **osluškivač događaja**)

Objekat koji predstavlja izvor događaja i objekat koji predstavlja osluškivač događaja nad tim izvorom na neki način se moraju dovesti u vezu, tj. mora se staviti do znanja koji je objekat osluškivač zainteresovan da osluškuje događaje sa tim izvorom. To se čini pozivom metoda **addActionListener()** za objekat izvor događaja, a kao argument metoda prosleđuje se referenca na objekat osluškivač.

Osluškivač događaja može biti objekat proizvoljne klase koja implementira odgovarajući listener-interfejs (u slučaju klika na dugme, taj interfejs je **ActionListener**. U ovom interfejsu deklarisan je samo jedan metod i to je: **public void actionPerformed(ActionEvent e);**) U implementaciji metoda listener-interfejsa treba staviti onaj kod koji želimo da se izvršava kada se desi događaj.

Kada se desi događaj vezan za objekat izvor, **automatski** se poziva odgovarajući metod listener-interfejsa koji smo implementirali u klasi osluškivača.

U primerima Loto, LotoAlternativa1, Loto... prikazani su razni načini da se definiše objekat osluškivač (otvoriti paralelno primer i poslednju prezentaciju sa vežbi i čitati kod i objašnjenje)

IZBEGAVANJE DEADLOCK-a
(**SwingUtilities.invokeLater()**)

Celu ovu pričicu zašto i kako možete da preskočite ako vas ne interesuje, bitno je samo da zovete *SwingUtilities.invokeLater()* tamo gde treba.

postoji nešto što se zove nit. Niti su procesi koji su međusobno nezavisni pa se mogu pseudo-paralelno izvršavati. Bitno je da se obrada događaja izvršava u posebnoj niti (tzv. niti za obradu događaja) koja je različita od glavne (main) niti. Ako se nakon prikazivanja GUI-ja vrše neke izmene na njemu u main-niti, to može uzrokovati da main-nit čeka da se izvrši nešto iz niti za obradu događaja da bi nastavila svoje izvršavanje, i obrnuto, da nit za obradu događaja čeka da se izvrši nešto u main-niti pa da nastavi svoje izvršavanje, što se naziva dead-lock.

Izbegavanje dead-lock-a je jednostavno i postiže se time što se stavi da se sav kod za kreiranje GUI-ja izvršava u niti za obradu događaja. To se postiže pozivom metoda *SwingUtilities.invokeLater()*. Ovaj metod očekuje jedan argument tipa *Runnable*, tj. tipa klase koja implementira *Runnable* interfejs. U tom interfejsu deklarisan je samo jedan metod:

public void run();

i u njegovo telo treba staviti onaj kod koji želimo da se izvršava u niti za događaje nakon obrade svih događaja koji čekaju. Mi napravimo metod *createGUI()* u koji stavimo sav

kod za kreiranje GUI-ja i u metodu *run()* samo pozovemo ovaj metod.

Argument metoda *invokeLater()* zgodno je napraviti pomoću anonimne klase:

```
new Runnable(){
    public void run(){
      createGUI();
    }
}
```

DOGAĐAJI NISKOG NIVOA

miš, tastatura, prozor

MouseEvent, *WindowEvent* (tip objekta događaja)

getSource() !!! vraća objekat koji je izvor događaja kao tip Object

MouseListener, *WindowListener* (listener interfejsi)

System.exit() završavanje aplikacije

ADAPTERSKE KLASE – klase koje implementiraju odgovarajuće listener-interfejse metodima sa praznim telom, pa umesto da nam objekat osluškivač bude objekat klase koja implementira listener-interfejs, pravimo klasu koja nasleđuje odgovarajuću adaptersku klasu i implementiramo samo metode za koje smo zainteresovani

zgodno je onda klasu osluškivača implementirati kao ugnježdenu jer onda može da pristupa članovima spoljašnje klase samo navođenjem njihovog imena.

MouseAdapter, *WindowAdapter* (adapterske klase)

LOTO-primeri

LotoDetalji:

statički inicijalizacioni blok (inicijalizacija statičkog niza)

atribut klase: niz dugmadi (dakle, niz komponenti) – zgodno kada imamo veći broj komponenti koje istovetno izgledaju i ponašaju se na isti način!

metod getNumbers() – primer metoda koji vraća niz!!!

žuta dugmad – primer komponenti koje same osluškuju sopstvene događaje

kontrolna dugmad – jedna klasa osluškivača za oba dugmeta, ali pravi se poseban objekat osluškivač za svako dugme, svakom dugmetu je pridružen jedinstveni identifikator - meni lično ovaj način nije nešto naročito prirastao za srce ☺

LotoAlternativa1

sve isto, samo se razlikuje kako je realizovana obrada događaja za kontrolnu dugmad:

pomoću metoda ***getSource()*** koji vraća referencu na objekat koji je izvor događaja.

Sada se pravi jedan objekat koji će biti osluškivač za oba kontrolna dugmeta.

Da bismo utvrdili koje je od dva dugmeta izvor događaja, reference na njih moraju biti dostupne metodu *actionPerformed()* u klasi osluškivača, pa te reference proglašavamo atributima klase apleta.

Odlučujemo koje dugme je izazvalo događaj poređenjem onoga što nam vrati getSource() i ovih referenci.

LotoAlternativa2

posebna klasa za svaki osluškivač
anonimne klase
(zgodan način obrade događaja)

LotoAlternativa3
ubačena je i obrada događaja niskog nivoa
promena izgleda kursora miša kada pređe preko žute dugmadi

LotoZutaDugmadAnonimneKlase
ovde je i obrada događaja za žutu dugmad realizovana pomoću anonimnih klasa

RadioDugmad

primer korišćenja radio dugmadi

SkraćivanjeRazlomka

tekstualna polja
onemogućavanje unosa u tekstualno polje