Objektno orijentisano programiranje 2

Klase i objekti

Klase

- Klasa je osnovna jedinica programiranja na jeziku Java
- Klase sadrže:
  - polja (atribute, podatke članove), nestatička i statička
  - metode (operacije, funkcije članice), nestatičke i statičke
  - konstruktore (specijalne funkcije za stvaranje objekata)
  - inicijalizacione blokove, nestatičke i statičke
  - ugnežđene tipove
- Polja
  - čine strukturu podataka
  - čuvaju stanje objekta ili klase (klase – statička polja)
- Metodi
  - sadrže naredbe (izvršni kod)
  - određuju ponašanje objekta ili klase (klase – statički metodi)

Ugovor i implementacija

- OOP striktno razdvaja pojmove:
  - šta se radi
  - kako se radi
- "Šta" je opisano ugovorom – skupom deklaracija javnih metoda
  - deklaracije uključuju potpis, tip rezultata, tipove bačenih izuzetaka
  - potpis uključuje ime metoda, broj i tipove parametara
  - deklaracija podrazumeva i semantiku metoda
  - deklaracije metoda u Javi – samo u apstraktnim klasama i interfejsima
  - klase u Javi nemaju razdvojene deklaracije od implementacije metoda
- "Kako" je opisano određenom implementacijom – skupom:
  - podataka – polja i konstanti
  - definicija metoda koji rade nad datim podacima
- Javni podaci su i deo implementacije, ali i ugovora
  - treba da se izbegavaju (osim imenovanih konstanti)

Primer jednostavne klase

- Klasa `NebeskoTelo` čuva podatke o nebeskim telima:

```
class NebeskoTelo {
  private static long sledeciID = 0;
  private long id=sledeciID++;
  private String ime="nepoznato";
  private NebeskoTelo kruziOko;
  public long citajId(){return id;}
  public NebeskoTelo(){}
  public NebeskoTelo(String ime){this.ime=ime;}
  public NebeskoTelo(String ime, NebeskoTelo centar){
    this(ime);
    kruziOko = centar;
  }
}
```

- Definicija klase uvodi novi naziv tipa (`NebeskoTelo`)

Stvaranje objekata

- Referenca na objekat klase se definiše na sledeći način:
  `NebeskoTelo merkur;`
- Gornja definicija reference na objekat ne stvara objekat
- Ako je polje vrednost je `null`; ako je lokalna promenljiva – nedefinisana je
- Primer dela modela solarnog sistema:

```
public static void main(String[] args){
  NebeskoTelo sunce=new NebeskoTelo("Sunce");
  NebeskoTelo zemlja=new NebeskoTelo("Zemlja", sunce);
  NebeskoTelo mesec=new NebeskoTelo("Mesec", zemlja);
}
```

- Alokator `new` stvara objekat i vraća referencu na njega
- Navodi se klasa i lista argumenata konstruktruktora (makar prazna)
- Objekat se stvara u memoriji za dinamičku dodelu (*heap*)
- Ako nema dovoljno memorije javlja se izuzetak `OutOfMemoryError`

Metod `toString`

- Metod `toString()` služi za konverziju proizvoljnog objekta u `String`
- Metod `toString()` se automatski poziva kada se referenca "nadovezuje" na nisku
- I primitivni podaci se automatski konvertuju u `String` kada se nadovezuju na nisku
- Primer (u klasi `NebeskoTelo`) :

```
public String toString(){
  String opis = id + " (" + ime + ")";
  if (kruziOko != null) opis+= " centar rotacije: " + kruziOko;
  return opis;
}
// … u metodu main:
System.out.println("Telo " + sunce);
System.out.println("Telo " + zemlja);
System.out.println("Telo " + mesec);
```

Izlaz:

```
Telo 0 (Sunce)
Telo 1 (Zemlja) centar rotacije: 0 (Sunce)
Telo 2 (Mesec) centar rotacije: 1 (Zemlja) centar rotacije: 0 (Sunce)
```

Polja

- Promenljive u klasama se nazivaju poljima (atributima)
- Primeri: `id`, `ime`, `kruziOko`, `sledeciID`
- Polje može da se inicijalizuje u definiciji klase
  - ne može kroz listu inicijalizatora u zaglavlju konstruktora (kao u C++)
- Dve vrste polja:
  - nestatička polja: jedno po objektu (`id`, `ime`, `kruziOko`)
  - statička polja: jedno po klasi (`sledeciID`)
- Svaki objekat ima svoju vlastitu kopiju nestatičkog polja
- Svi objekti dele jednu kopiju statičkog polja
- Statička polja se nazivaju i “promenljive klase”
- Statička polja se inicijalizuju čim se klasa učita u memoriju
- Nestatička polja se inicijalizuju kad se stvori objekat

Kontrola pristupa

- Članovi su pristupačni u celom kodu klase u kojoj se nalaze
- Prava pristupa članu iz drugih klasa se određuju modifikatorom:
- `private`
  - član je pristupačan samo u klasi gde je definisan
- (bez modifikatora pristupa, podrazumevano, paketsko pravo)
  - član je pristupačan samo u kodu datog paketa
- `protected`
  - član je pristupačan u izvedenim potklasama i u kodu celog paketa
- `public`
  - član je pristupačan sa proizvoljnog mesta
- Svako više pravo uključuje prethodna prava

Metodi

- Sadrže kod koji manipuliše stanjem objekta
- Pozivaju se preko referenci na objekte koristeći operator `.`
  - *referenca.metod(argumenti)*
- Svaki parametar ima naveden tip
- Parametri ne mogu da imaju podrazumevane vrednosti argumenata
- Svi argumenti metoda se prenose isključivo po vrednosti
  - vrednosti parametara u telu metoda su kopije stvarnih parametara
  - nalaze se na steku
- Referenca se prenosi po vrednosti, ali objekat po referenci
  - referenca sadrži adresu, prenosi se kao i argumenti primitivnih tipova
- Povratni tip se deklariše ispred imena metoda
  - ako metod ne vraća nikakvu vrednost – tip "rezultata" je `void`
- Ne postoje statičke lokalne promenljive

Promenljiv broj argumenata

- Parametar za promenljiv broj argumenata podržan od verzije 5

```
void metod(Object ... par);
```

  - podrška za formatirani izlaz sa promenljivim brojem argumenata:

```
System.out.printf("%s %3d", ime, godine);
```

- Primer:

```
public class Parametri {
    static void m(int ... p){ for(int i: p) System.out.print(i); }
    public static void main(String[] args) { m(1); m(2,3); m(4,5,6); }
}
```

  Izlaz: `123456`

- Parametar za promenljiv broj argumenata može da se indeksira
- Ne može da mu se dohvati broj argumenata (npr. `par.lenght`)

Primer prenosa po vrednosti

- Primer:

```
public static void main(String[] argumenti){
  double jedan = 1.0;
  System.out.println("pre: jedan="+jedan);
  prepolovi(jedan);
  System.out.println("posle: jedan="+jedan);
}
public static void prepolovi(double arg){
  arg /= 2.0; System.out.println("funkcija: pola="+arg);
}
```

- Izlaz:

```
pre: jedan=1
funkcija: pola=0.5
posle: jedan=1
```

Primer prenosa po referenci

- Primer:

```
public static void main(String[] argumenti){
  NebeskoTelo venera = new NebeskoTelo("Venera", sunce);
  System.out.println("pre: "+venera);
  drugoIme(venera);
  System.out.println("posle: "+venera);
}
public static void drugoIme(NebeskoTelo telo){
  telo.ime = "Zvezda Danica";
  telo = null; // nema značaja
}
```

- Izlaz:

```
pre: 1 (Venera) centar rotacije: 0 (Sunce)
posle: 1 (Zvezda Danica) centar rotacije: 0 (Sunce)
```

Pristupni metodi i preklapanje imena

- Loša praksa je da polja imaju javni pristup
  - sprečava promenu implementacije klase
- Pristupni (*accessor*) metodi regulišu pristup privatnim podacima
- Na primer, da bi se osigurao *read-only* pristup polju `id`, ono je privatno, a odgovarajući javni metod `citajId()` ga samo čita
  - nema načina da korisnik klase `NebeskoTelo` modifikuje polje `id`
- Preklapanje imena (*name overloading*):
  - 2 metoda mogu da imaju isto ime
  - potrebno je da njihovi potpisi sadrže različit broj ili tipove argumenata
- Primer dva pristupna metoda sa preklopljenim imenima:

```
public NebeskoTelo orbitira() {return kruziOko;}
public void orbitira(NebeskoTelo centar)
  { kruziOko = centar; }
```

Referenca `this`

- Specijalna referenca na objekat kojem se upravo pristupa
- Može da se koristi unutar nestatičkih metoda
- Način da se prosledi referenca na tekući objekat kao parametar drugim metodima:

```
lista.dodaj(this);
```

- Takođe, može da se koristi za pristup sakrivenim članovima:

```
public NebeskoTelo(String ime){this.ime=ime;}
```

- Sakrivanje imena polja imenom parametra
  - dobra praksa samo u konstruktorima i pristupnim metodima
- Implicitan `this.` se dodaje ispred reference člana:

```
class Ime {
  String s;
  Ime(){ s="bezimeni"; } // isto što: this.s="bezimeni"
}
```

Inicijalizacija objekta

- Inicijalna vrednost polja može da se navede u definiciji klase
  - inicijalizator, izraz koji se izračunava
  - primer:

```
public String ime="nepoznato";
```

- Ako se vrednost ne pridruži eksplicitno polju, biće "nula": `0`, `+0.0f`, `+0.0`, `\u0000`, `false` ili `null`
- Ako je potrebna netrivijalna operacija za određivanje početnog stanja objekta – konstruktori
- Za jednostavnu inicijalizaciju bez argumenata – inicijalizacioni blokovi

Konstruktori

- Konstruktori – specijalne funkcije za inicijalizaciju objekata
  - imaju isto ime kao klasa kojoj pripadaju
  - nula ili više parametara (kao metodi)
  - nemaju povratnu vrednost (za razliku od metoda)
- Konstruktori se izvršavaju nakon što se:
  - poljima pridruže njihove podrazumevane vrednosti
  - izvrše eksplicitni inicijalizatori i inicijalizacioni blokovi
- Specifični konstruktori
  - konstruktor bez argumenata (*no-arg*) – za podrazumevanu inicijalizaciju
  - implicitni *no-arg* konstruktor – ne radi ništa (prazno telo)
    - obezbeđen samo ako ne postoji drugi konstruktor
    - *public* – ako je klasa javna, odnosno nije javni ako klasa nije javna
  - sa ograničenim pravom pristupa – ograničava ko može da stvara objekte
    - privatni, paketski i zaštićeni

Inicijalizacioni blokovi

- Blok za inicijalizaciju – blok naredbi u telu klase
- Izvršavaju se pre konstruktora, po redosledu navođenja
  - kao da su sastavni deo na početku svakog konstruktora
- Primer (u klasi `NebeskoTelo`): umesto u inicijalizatoru polja `id`
  `{ id = sledeciID++; }`
- Blokovi se koriste za jednostavnu inicijalizaciju:
  - kada nisu potrebni argumenti pri inicijalizaciji objekta
- Redosled izvršavanja inicijalizatora i inicijalizacionih blokova
  - sleva u desno, odozgo naniže, bez obzira šta su
- Inicijalizacioni blok može da baci izuzetak
  samo ako su svi konstruktori deklarisali da bacaju taj izuzetak

Statička polja

- Statičko polje (promenljiva klase) ima samo jednu pojavu po klasi
  - tačno jedna promenljiva bez obzira na broj (čak 0) objekata klase
- Statičko polje se inicijalizuje po učitavanju klase, a pre nego što se:
  - bilo koje statičko polje u toj klasi koristi
  - bilo koji metod te klase počne izvršavanje
- Primer:

```
class Inicijalizacija{
  public void init() {y=x;}
  private static double x = 10.0;
  private double y;
  ...
} // x je definisano polje u trenutku korišćenja
```

Statički metodi

- Statički metod može da obavlja opšti zadatak za sve objekte klase
- Statički metod može direktno da pristupa samo
  - statičkim poljima
  - statičkim metodima klase
- Nestatičkim poljima i metodima može da se pristupa samo indirektno
  - korišćenjem reference na neki objekat čijem se polju/metodu pristupa
- Ne postoji `this` referenca (nema tekućeg objekta nad kojim se radi)
- Izvan klase – statičkom članu se pristupa koristeći ime klase i operator `.`
- U klasi `ProstiBrojevi` (naredni slajd) su definisani:
  - statički metod: `sledeciProstBroj()`
  - statički niz: `prviProstiBrojevi`
- Korišćenje izvan klase *klasa.statičkiČlan*:

```
prostBroj = ProstiBrojevi.sledeciProstBroj();
n = ProstiBrojevi.prviProstiBrojevi.length;
```

Statički inicijalizacioni blokovi

- Služe za inicijalizaciju statičkih polja ili drugih stanja
- Primer:

```
class ProstiBrojevi{
  static int[] prviProstiBrojevi=new int[100];
  static int sledeciProstBroj() {...}
  static {
    prviProstiBrojevi[0]=2;
    for (int i=1; i< prviProstiBrojevi.length; i++)
      prviProstiBrojevi[i]=sledeciProstBroj();
  }
  // ...
}
```

- Redosled statičke inicijalizacije: sleva-udesno i odozgo-naniže
- U toku izvršenja inicijalizatora statičkih polja još nije pripremljena obrada izuzetaka
  - inicijalizatori ne smeju da pozivaju metode koji deklarišu da mogu da bacaju izuzetke
- Statički blok sme da poziva metode koji bacaju izuzetke, samo ako može i da ih hvata

Problem ciklične statičke inicijalizacije

- Problem:
  - ako statički inicijalizator u klasi A poziva statički metod u klasi B, a statički inicijalizator u klasi B poziva statički metod u klasi A
  - ne može da se otkrije u vreme prevođenja
    - B može još da ne bude napisana kada se A prevodi i obrnuto
- Ponašanje:
  - inicijalizatori A se izvršavaju do tačke poziva metoda klase B
  - pre nego što se izvrši metod klase B, inicijalizatori B se izvršavaju
  - kada inicijalizator B pozove metod klase A ovaj se izvrši (iako nije završena inicijalizacija klase A)
  - završavaju se inicijalizatori klase B
  - izvršava se pozvani metod klase B
  - konačno, završavaju se inicijalizatori klase A

Primer ciklične statičke inicijalizacije

```
class A{
   static {
     System.out.println("Izvršenje statičkog bloka A počelo");   // (1)
     B.metod();
     System.out.println("Izvršenje statičkog bloka A završava"); // (6)
   }
   static void metod(){ System.out.println("A.metod"); }         // (3)
}
class B{
   static {
     System.out.println("Izvršenje statičkog bloka B počelo");   // (2)
     A.metod();
     System.out.println("Izvršenje statičkog bloka B završava"); // (4)
   }
   static void metod(){ System.out.println("B.metod");  }        // (5)
}
class T{ public static void main(String[] args){ A a  = new A(); } }
```

Objekti na koje ne upućuju reference

- Java eliminiše potrebu da se objekti uništavaju eksplicitno
- Kada ni jedna referenca ne upućuje na objekat, prostor koji ovaj zauzima može da se oslobodi
- Da bi se prostor oslobodio potrebno je da nema referenci na objekat:
  - ni u jednom statičkom podatku
  - ni u jednoj promenljivoj bilo kog tekuće izvršavanog metoda
  - ni u jednom polju ili elementu niza do kojeg bi se moglo stići počevši od statičkih podataka ili promenljivih izvršavanih metoda
- Ako na objekat upućuju reference samo iz objekata na koje više ne upućuju reference – može da se i taj izbaci

Sakupljač đubreta

- Prostor se oslobađa ako je potrebno još prostora
  - sakupljač đubreta se aktivira da se izbegne situacija *out of memory*
- Đubre se sakuplja bez akcija programera, ali sakupljanje đubreta uzima vreme
- Programi treba da se projektuju tako da ne budu previše produktivni u stvaranju objekata
- Sakupljač đubreta ne garantuje da će uvek da bude dovoljno memorije raspoloživo za nove objekte
- Na ovaj način Java rešava probleme
  - visećih referenci i
  - curenja memorije
- Automatsko uklanjanje đubreta otklanja potrebu za destruktorima
  - destruktori u C++ prvenstveno služe za eksplicitnu razgradnju objekta

Metod `finalize()`

- Koncept sakupljanja đubreta oslobađa programera od brige za oslobađanje memorije
- Primarna uloga destruktora u jeziku C++: oslobađanje memorije
- Posledica: u jeziku Java klase nemaju destruktore
- C++ destruktori su obavljali i oslobađanje nememorijskih resursa
- Ideja: metod `finalize()` u klasi `Object` za takve potrebe
- Metod se poziva iz sakupljača đubreta pri uklanjanju objekta
- Problem: za objekte koji „dožive“ kraj programa metod se ne izvrši
- Čak i za „mrtve“ objekte metod se ne izvrši automatski
  - potreban poziv: `System.gc(); // ni tad nema garancije`
- Metod je „zastareo, za uklanjanje“ (od v9, ali još nije uklonjen)

Nadjačavanje `finalize()`

- Metod `finalize()` je definisan u klasi `Object`
- Svaka klasa može da nadjača metod `finalize()`
  - iz tog metoda treba pozvati metod `finalize()` natklase
  - poziv istoimenog metoda natklase – preko reference `super`

```
public void finalize() throws Throwable {
  super.finalize();
  //...
}
```

Primer `finalize()`

- Klasa vrši obradu nekog toka podataka i obezbeđuje njegovo zatvaranje

```
public class ObradaToka{
  private Tok tok;
  public ObradaToka(String ime){
    tok=new Tok(ime);
  }
  //...
  public void zavrsi(){
    if(tok!=null){tok.zatvori(); tok=null;}
  }
  public void finalize() throws Throwable
    { super.finalize(); zavrsi(); }
}
```

- Metod `zavrsi()` je korektan i za višestruke pozive
- Problem: neizvesno je implicitno zatvaranje toka (pozivanjem metoda `finalize()`)

Izuzeci, izlaz iz aplikacije i `finalize()`

- Izuzeci i `finalize()`
  - telo `finalize()` može da koristi `try/catch` da obradi izuzetke metoda koje on pozove
  - sve neuhvaćeni izuzeci koji su se pojavili za vreme izvršenja `finalize()` se ignorišu
  - razlog: metod `finalize()` se poziva iz niti sakupljača đubreta
- Izlaz iz aplikacije i `finalize()`
  - ne poziva se metod `finalize()` za „žive“ objekte
  - ako se prethodno pozove `System.gc()` poziva se metod `finalize()` „mrtvih“ objekata (ali ne garantovano)
  - neke greške mogu da spreče da se svi `finalize()` metodi izvrše
    - na primer: pri završavanju zbog nedostatka memorije (*out of memory*)

Oživljavanje objekata

- Metod `finalize()` može da reanimira objekat postavljajući ponovo referencu na njega
  - na primer, dodajući tu referencu u neku statičku listu
- Nije preporučljivo takvo ponašanje
- Umesto oživljavanja – „umirući“ objekat treba da se klonira
  - klon (novi objekat) preuzima stanje umirućeg objekta
- Metod `finalize()` se pokreće najviše jednom za svaki objekat
- Ako je `finalize()` oživeo objekat, neće se ponovo izvršiti kad se objekat bude stvarno uklanjao
- To znači da objekat može samo jednom da bude reanimiran

Metod `main`

- Metod `main()` mora da ima:
  - modifikatore `public` i `static`
  - tip `void`
  - jedan parametar tipa `String[]`
- Aplikacija može da ima više `main()` metoda
  - svaka klasa može da ima po jedan `main()` metod
- Stvarno korišćeni `main()` pri pokretanju aplikacije:
  - specificiran imenom klase u komandnoj liniji
- Preporuka:
  - svaka klasa treba da ima `main()` metod za potrebe testiranja

Primer metoda `main()`

- Ispisuju se argumenti uneti preko komandne linije pri pokretanju:

```
class Eho {
  public static void main(String[] argumenti) {
    for(String a: argumenti) System.out.print(a+" ");
    System.out.println();
  }
}
```

- Iz komandne linije `Eho` može da se pozove na način:

```
java Eho se prikazuje
```

- Rezultat će da bude:

```
se prikazuje
```