Reference na objekte

Od ranije se sećate da se u memoriji računara nalaze obrasci britova. Konkretan obrazac ima značenje samo znate koji tip vrednosti on predstavlja i koja se šema koristi za predstavljanje. Šema koja se koristi u konkretnom delu memorije je tip podatka.

U Javi postoji puno ugrađenih tipova podataka, a vi kao programer, možete definisati onoliko novih tipova koliko vam je potrebno.Ako nisu primitivni tipovi podataka, svi tipovi su klase. Drugim rečima, podataka je ili primitivni podatak ili klasa. Jedini tip podatka koji programer može da definiše je klasa. Svaki objekat u Javi je primerak neke klase. Pre nego što se napravi objekat, mora postojati definicija klase.

<table>
<tr><td colspan="2">Svi podaci</td></tr>
<tr><td>primitivni tipovi</td><td>objekti</td></tr>
</table>

Primitivne promenljive

Evo jednog malog programa koji koristi primitivni tip podatka:

```
class egLong{
  public static void main ( String[] args ) {
    long vrednost;
    vrednost = 18234;
    System.out.println(vrednost);
  }
}
```

Promenljiva vrednost u ovom programu predstavlja ime za 64-bitni deo memorije koji se koristi za smeštanje celih brojeva. Iskaz

```
vrednost = 18234;
```

postavlja konkretan obrazac bitova u deo memorije.

Kod primitivnih tipova podataka promenljiva je deo memorije koji je rezervisan za vrednost konkretnog tipa. Ako na primer, kažemo long value, 64 bitova memorije se rezerviše za ceo broj. Ako kažemo int sum, 32 bita memorije se rezervišu za ceo broj.

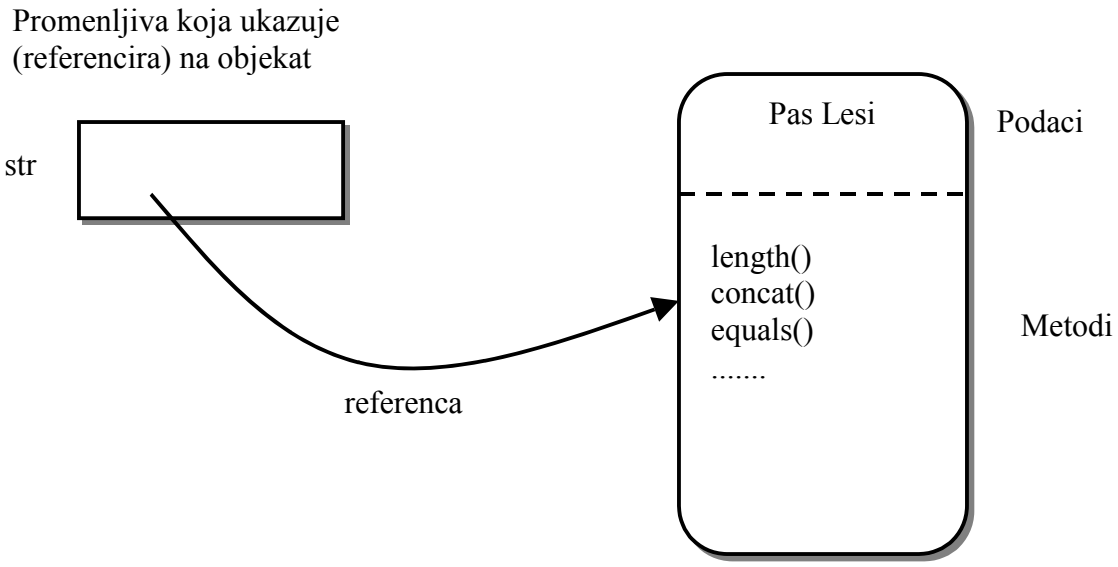
Promenljive koje ukazuju na objekte se ne ponašaju na ovaj način.

Objekti

Objekti su veliki, komplikovani i promenljive veličine. Prilikom deklaracije promenljive koja ukazuje na objekat, nećete automatski dobiti i objekat. Sve što dobijate je ime za budući objekat. Pogledajmo sledeći program:

```
class egString{
  public static void main ( String[] args ) {
    String str;
    str = new String( "Pas Lesi" );
    System.out.println( str );
  }
}
```

Objekat sadrži podatke i metode (stanje i ponašanje). Objekat klase String možete vizuelno predstaviti na sledeći način:



Objekat klase String

Deo objekta sa podacima sadrži karaktere. Deo sa metodima sadrži puno metoda. (Ova slika je pojednostavljenje realne situacije. Java sistem radi nešto efikasnije, ali logički ekvivalentno.)

Ako se prisetimo operatora dodele, kada se on izvrši dešavaju se dve stvari:

1. Određuje se vrednost izraza sa desne strane znaka =
2. Rezultat tog izračunavanja se dodeljuje promenljivoj sa leve strane znaka =.

To izgleda otprilike ovako:

```
String str;    // ime budućeg objekta

str =      new String( "Pas Lesi" );
--+--      ----------------+--------------
  |                        |
  |
  |                        1.  Napravljen je objekat
  |                            Java sistem zna kako da prati
  |                            i pronađe objekat

  2. Ovo je način da pronađete objekat smešten u promenljivoj str
```

Referenca na objekat je informacija kako da pronađete konkretan objekat. Objekat je deo glavne memorije. Referenca na objekat je način da se pristupi tom delu memorije. Promenljiva str ne sadrži objekat, ali sadrži informacije o tome gde se objekat nalazi.

Objekti se kreiraju tokom rada programa. Svaki objekat ima jedinstvenu referencu, koja se koristi da bi ga program pronašao. Kada se referenca objekta dodeli promenljivoj, onda promenljiva zna kako da pronađe objekat.

Referenca na objekat se u dijagramima obično prikazuje strelicom od promenljive prema objektu.

U Javi postoje dve vrste promenljivih. Tabelarno se to može predstaviti na sledeći način:

	Karakteristike
primitivna promenljiva	sadrži stvarne podatke
referenca	sadrži informacije o tome gde da se pronađe objekat

Pogledajmo sada malo proširenu verziju prethodnog programa (dodata je nova promenljiva primitivnog tipa):

```
class egString2{
  public static void main ( String[] args )  {
    String str;
    long   vrednost;
    str   = new String( "Pas Lesi" );
    vrednost = 32912;
    System.out.println( str   );
    System.out.println(vrednost);
  }
}
```

Kada se izvrši iskaz

```
str   = new String( " Pas Lesi " );
```

pravi se novi objekat i njegova referenca se postavlja u promenljivu str. Promenljiva str sada ukazuje na objekat tipa String.

Dve vrste operatora dodele

U skladu sa prethodnim napomenama, treba primetiti da postoji razlika između iskaza:

```
vrednost = 32912;
```

i

```
str   = new String( "Pas Lesi" );
```

U prvom iskazu je vrednost primitivan tip, tako da iskaz dodele postavlja podatak direktno u nju. U drugom slučaju, str je referenca na objekat, tako da se u tu promenljivu postavlja referenca na objekat. Upamtite da promenljiva nikad ne sadrži objekat.

Tabelarno se ovo može predstaviti na sledeći način:

Vrsta promenljive	Informacije koje sadrži	Kada je sa leve strane znaka =
Primitivna promenljiva	Sadrži stvarne podatke	Prethodni podaci se zamenjuju novim podacima
referenca	Sadrži informacije o tome kako da se pronađe objekat	Stara referenca se zamenjuje novom

Da vidimo kako se ove dve vrste promenljivih koriste.

Ako ponovo pogledamo primer videćemo da kada se izvrši iskaz

```
System.out.println( str );
```

pronalazi se objekat na koji ukazuje str, a onda se podaci iz tog objekta prikazuju na monitoru.

Kada se izvrši iskaz

```
System.out.println( vrednost );
```

primitivna vrednost u promenljivoj vrednost se koristi direktno. (Ona se prevodi u niz karaktera i štampa na ekranu.)

Naša dva tipa promenljivih se u izrazima ponašaju različito:

	Karakteristike	Kada se koristi u izrazu
primitivna promenljiva	Fiksan broj bitova. Sadrži stvarne podatke.	Koristi podatke u promenljivoj
referenca	Sadrži informacije o tome kako pronaći objekat	Vrednost u promenljivoj se koristi da se pronađe objekat

Tip promenljive određujete prilikom njene deklaracije. Na primer,

```
String str;
```

govori da je str referenca, za koju se očekuje da sadrži referencu na objekat tipa String. Deklaracija

```
long value;
```

kaže da je vrednost promenljiva koja sadrži primitivan tip long. Prilikom kompilacije programa, ove deklaracije govore kompajleru kakave informacije se nalaze u promenljivoj, tako da on svaku od promenljivih koristi na odgovarajući način.

Da još malo promenimo prethodni program:

```
class egString3{
  public static void main ( String[] args )  {
    String str;
    str   = new String("Pas Lesi");
    System.out.println(str);
    str   = new String("Maca Tom");
    System.out.println(str);
   }
}
```

Program radi baš ono što biste očekivali. Štampa sledeće:

```
Pas Lesi
Maca Tom
```

Da objasnimo sada neke detalje.

Iskaz	Akcija
`str = new String("Pas Lesi");`	Kreira se prvi objekat. Referenca na objekat se postavlja u str.
`System.out.println(str);`	Koristi se referenca na prvi objekat. Uzimaju se podaci iz objekta i štampaju.
`str = new String("Maca Tom");`	Kreira se drugi objekat. U str se postavlja referenca na drugi objekat. U ovom trenutku ne postoji referenca na prvi objekat. Prvi objekat je sad "otpad".
`System.out.println(str);`	Koristi se referenca na drugi objekat. Uzimaju se podaci iz tog objekta i štampaju.

Obratite pažnju na sledeće:

1. Svaki put kada se upotrebi operator new pravi se novi objekat.
2. Svaki put kada se pravi objekat, pravi se i referenca na njega.
3. Svaki objekat koji postoji u računaru ima jedinstvenu referencu.
4. Referenca se obično čuva u promenljivoj.
5. Referenca iz promenljive se kasnije koristi za pristup objektu.
6. Ako se u istu promenljivu stavi nova referenca ona zamenjuje prethodnu.
7. Ako ne postoji promenljiva koja sadrži referencu na objekat, onda nema načina da pronađete objekat, tako da on postaje otpad.

Izraz otpad se koristi za objekte na koje ne ukazuje nijedna referenca (promenljiva). Pošto program ne može da pronađe objekat, za njega taj objekat i ne postoji. Ova situacija se često dešava i obično nije greška. Objekti se kreiraju tokom

izvršenja programa. Kada reference na njih više nisu potrebne, oni se odbacuju. U Javi postoji deo sistema, tzv. skupljač otpada (garbage collector) koji izbacuje izgubljene objekte, tako da memorija koju su oni zauzimali može da se ponovo koristi.

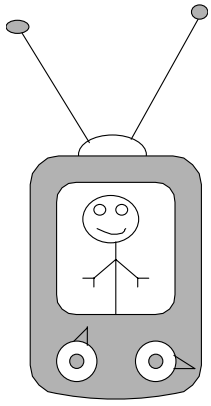
```
class egString4{
  public static void main ( String[] args )  {
    String strA;  // referenca na prvi objekat
    String strB;  // referenca na drugi objekat

    strA   = new String( "Pas Lesi" );    // kreira se prvi objekat i
                                        // Zapisuje se njegova referenca

    System.out.println(strA); // koristi se referenca na prvi objekat
                              // i štampaju se njegovi podaci

    strB   = new String( "Maca Tom");// Kreira se drugi objekat i
                                     // zapisuje njegova referenca

    System.out.println(strB); // koristi se referenca na drugi objekat
                            // i štampaju njegovi podaci

    System.out.println(  strA  ); // Koristi se referenca na prvi
                                  //  objekat i štampaju njegovi podaci

  }
}
```

U programu postoje dve promenljive sa referencama, strA i strB. Program pravi dva objekta i referencu na njih stavlja u odgovarajuće promenljive. Pošto svaki objekat ima svoju promenljivu sa referencom, nijedna referenca se ne gubi, tako da nema objekta koji bi bio otpad (sve dok program ne završi rad).

Učaurenje

Jedan od osnovnih principa objektno orijentisanog programiranja jeste pravilo sakrivanja informacija ili učaurenja (eng. encapsulation).

Učaurenje se može se može uporediti sa TV aparatom. Detalji o unutrašnjem radu su skriveni od korisnika. Korisnik sa aparatom komunicira preko definisanih kontrolnih dugmadi. Ta dugmad se ponekad nazivaju i korisničkim interfejsom.

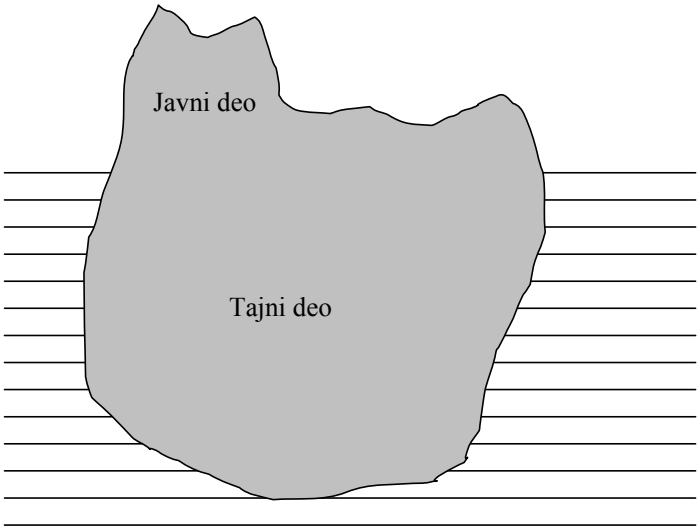


Ovo pravilo se prvenstveno odnosi na poboljšanu modularnost programa. Ono kaže da autor svakog modula mora da izabere podskup svojstava modula koja predstavljaju zvanični opis modula. Taj opis koriste autori klijentskih modula koji pristupaju tom modulu.

Opis modula treba da sadrži samo neka svojstva. Sve ostalo treba da ostane tajna. Javna svojstva modula se nazivaju interfejsom modula.

Osnovni razlog koji stoji iza skrivanja informacija je kriterijum neprekidnosti. Pod pretpostavkom da se modul promeni, ali da se promene odnose samo na njegove tajne elemente, a da javni ostanu nepromenjeni, onda promene ne utiču na druge module koji ga koriste, na klijente. Što je manji javni deo modula, veća je šansa da će promene modula biti ograničene samo na njegov tajni deo.

Modul koji poštuje skrivanje informacija se može opisati kao ledeni breg. Samo njegov vrh, interfejs je vidljiv za klijente.



Ovo se može bolje razumeti na primeru. Pogledajmo na primer, proceduru za dobijanje atributa objekta, koji se nalaze u nekoj tabeli i organizovani su preko ključeva. Svaki objekat ima jedinstveni ključ i osobine koje pripadaju tom ključu i objektu. Procedura koja vadi te osobine na osnovu ključa može biti različita u zavisnosti od toga kako je tabela u kojoj se atributi i objekti čuvaju. To može biti sekvencijalna datoteka, heš tabela, binarno stablo itd. Skrivanje informacija ukazuje na to da upotreba procedure treba da bude nezavisna od konkretne implementacije. Na taj način klijent neće pretrpeti nikakve promene usled promene implementacije.

Koja svojstva modula treba da budu javna, a koja tajna? Opšte pravilo bi bilo da javni deo treba da sadrži specifikaciju funkcija modula, a da sve što se odnosi na implementaciju tih funkcija, treba da ostane tajno, da bi se ostali moduli sačuvali od kasnijih promena implementacije.

Vrlo je bitno izbeći nesporazum do kojeg često dovodi ime skrivanje informacija. Skrivanje informacija se ne odnosi na zaštitu podataka u smislu bezbednosti, na primer fizičko sprečavanje autora klijentskih modula od pristupa internom tekstu autorskog modula. Tehnički, skrivanje informacija znači da klijentski moduli treba da se baziraju samo na javnim svojstvima snabdevača.

Modifikatori vidljivosti

Kako u programskom jeziku ostvariti princip skrivanja informacija? U Javi se to radi preko modifikatora vidljivosti.

Modifikator vidljivosti private

Kada je član klase deklarisan kao privatan, on se može koristiti samo u metodima te klase.

Sledeći primer pokazuje kako to izgleda u praksi.

```
class Sfera{
   private double poluprecnik;
   final double pi = 3.14159265;
   double Zapremina(){
          v = 4 * poluprecnik * poluprecnik * poluprecnik  * pi / 3;
          return v;
   }
}
```

Promenljivoj poluprecnik mogu da pristupaju samo metodi klase Sfera.

Pošto je u pitanju sfera postavlja se pitanje kako biste u prethodnom slučaju zadali koliki je poluprečnik konkretne sfere. Kako se spolja ovom atributu klase ne može pristupiti, sledi da se mora napraviti neki metod koji će biti dostupan spolja, a koji će zadavati vrednost poluprecnika. Sledi dopunjena verzija prethodne klase.

```
class Sfera{
   private double poluprecnik;
   final double pi = 3.14159265;
   double setPoluprecnik(double r){
          poluprecnik = r;
   }
   Sfera(){
          poluprecnik = 1.;
   }
   double Zapremina(){
          v = 4 * poluprecnik * poluprecnik * poluprecnik  * pi / 3;
          return v;
   }
}
```

Metodi kojima se pristupa privatnim članovima klase se često nazivaju metodima pristupa. Ovi metodi su vidljivi drugim klasama. Metodi pristupa obično ne menjaju vrednosti atributa klase, ali se može i to desiti, ako je potrebno da se vrše neka izračunavanja.

```
class Primer{
   public static void main( String[] args ){
     Sfera sf = new Sfera();// poziva se konstruktor klase Sfera
     sf.setPoluprecnik(10.);
     System.out.println( sf.Zapremina());// stampa se zapremina
                                         //sfere
   }
}
```

Metod main iz prethodnog primera ne može da vidi privatne članove klase. Zbog toga je morao da im pristupa preko posebnih metoda (setPoluprecnik). Ako bismo probali da direktno pristupimo članovima klase definisanim kao u prethodnom primeru, kompajler bi javio grešku.

Može vam izgledati čudno da se prvo atribut klase učini privatnim i tako se spreči pristup do njega, a da se onda napravi metod koji omogućava da mu se ipak pristupi spolja. Razlog je u tome što je u ovom drugom slučaju pristup do tog atributa konzistentan. Zna se da svi koji koriste taj objekat pristupaju njegovim atributima na isti način, preko unapred definisanih metoda. Ako poželimo da kasnije promenimo implementaciju tih metoda, to neće uticati na druge objekte, koji koriste promenjeni.

U prethodnom primeru, bismo mogli, na primer, da promenimo metod setPoluprecnik(), tako da se proverava da li je poluprecnik veći od 0, odnosno da li je poslata vrednost koja se može realno dogovoditi. Takva promena neće uticati na objekte klijente, koji koriste taj metod za zadavanje vrednosti poluprecnika.

```
class Sfera{
   private double poluprecnik;
   final double pi = 3.14159265;
   double setPoluprecnik(double r){
          if(r < 0.)
                 return;
          else
                 poluprecnik = r;
   }
   Sfera(){
          poluprecnik = 1.;
   }
   double Zapremina(){
```

```
            v = 4 * poluprecnik * poluprecnik * poluprecnik  * pi / 3;
            return v;
    }
}
```

Sada ne može da se zada poluprečnik sfere koji je manji od 0. Ovo naravno nije potpuno funkcionalan program. Možda bi bolje bilo da se javi poruka o tome da je pokušan upis poluprečnika manjeg od 0, itd., ali to ćemo ostaviti za kasnije.

Pored toga što članovi klase mogu biti deklarisani kao privatni i metodi se mogu deklarisati kao privatni. Koji bi bio razlog da se neki metod označi kao privatan?

Ako u svojoj klasi treba da obavite neki zadatak koji se stalno ponavlja na različitim mestima u programu, onda u skladu sa principom dekompozicije softvera ima smisla da taj kod izdvojite u posebnu celinu, poseban metod i onda ga samo pozivate kad vam zatreba. Ako ste sigurni da se taj metod neće pozivati spolja, odnosno da neće biti potreban klijentima klase, onda taj metod može i treba da bude privatan.

Modifikator vidljivosti public

Modifikator private spre;ava spoljašnji svet da pristupa unutrašnjoj implementaciji objekta. Koristite ga da biste sačuvali implementaciju. Sa druge strane objekat ne postoji sam za sebe. On je u programu da bi komunicirao sa drugim objektima. Rečeno je da se ta komunikacija odvija preko interfejsa koji je unapred definisan i koji ne bi trebalo da se mnogo menja.

Pristupanje objektu se vrši preko njegovih metoda. Članovi klase (atributi i metodi) se mogu definisati kao javni, odnosno svima dostupni. U tom cilju se koristi identifikator public. Ovaj identifikator eksplicitno označava da se metodu ili promenljivoj može pristupati izvan objekta.

```
class Racun{
  private String brojRacuna;
  private String vlasnik;
  private int    stanje;
  private int    upotrebaRacuna = 0;

  public Racun ( String brRac, String vl, int start )
      { . . . . }
  private void sledecaUpotreba() { . . . . }
  public int trenutnoStanje() { . . . . }
  public void  staviNaRacun( int suma ) { . . . . }
  public void uzmiSaRacuna( int suma) { . . . . }
  public void prikazi() { . . . . }
}
```

Programer je taj koji određuje koji će metodi i atributi biti javni, a koji privatni. Dobra programerska klasa nalaže da se atributi klase deklarišu kao privatni, dok su metodi javni. Od ovog pravila, naravno, postoje izuzeci, kao što ste videli kod privatnih metoda, ali svaki izuzetak potvrđuje pravilo.

Podrazumevana vidljivost

U programima koje ste pisali na početku, niste definisali da li je neka promenljiva ili metod privatna ili javna. Ako se eksplicitno ne deklariše kakva je vidljivost nekog člana klase, onda on ima podrazumevanu vidljivost.

Podrazumevana vidljivost kaže da su metod ili atribut vidljivi samo unutar same klase, ali i svih drugih klasa kojesu članice istog paketa. Paketi su način organizacije klasa u logičke celine, koji se koristi u Javi i o tome će kasnije biti više reči.

Pošto još nismo govorili o paketima, možete da prihvatite da je podrazumevana vidljivost, za naše programe, u stvari public. I pored toga što je približno isto ne deklarisati vidljivost i reći da je ona javna, ipak je bolje, da ako želite da naznačite vidljivost, napišete ključnu reč public i time jasno stavite do znanja šta ste želeli.

Pored navedenih tipova modifikatora vidljivosti, postoji i modifikator protected. Ovaj modifikator se koristi zajedno sa nasleđivanjem, pa će o njemu biti više reči kada budemo govorili o nasleđivanju.

Mehanizmi apstrakcije

Apstrakcije omogućavaju ljudima da se koncentrišu na opšte ideje, a ne na specifične manifestacije tih ideja. Osnove apstrakcije potiču iz filozofije i matematike i ona se koristi i u mnogim drugim oblastima, uključujući i računarske nauke. U sistemskoj analizi apstrakcija je disciplina putem koje se koncentrišemo na najvažnije aspekte problema koji rešavamo, a zanemarujemo aspekte koji nisu bitni. Na primer, ako projektujemo sistem za kontrolu avionskog saobraćaja, koncentrisaćemo se na detalje koji su bitni za rad sistema (kao što su tip aviona i znak upozorenja), a zanemarićemo nevažne detalje, kao što su boja aviona, ili mena osoblja i putnika.

U programiranju apstrakcija ukazuje na razliku koju pravimo između onog šta program radi i kako to radi. Na taj način se mogu odvojiti napori programera koji koriste neki program i onih koji taj program implementiraju. Pored toga možemo da koristimo programske jedinice nižeg nivoa i da preko njih implementiramo programske jedinice višeg nivoa, a da ove koristimo za implementaciju još viših programa. Odvajanje posla i višestruki nivoi apstrakcije su najvažniji alati kod izgradnje velikih programskih sistema.

Proceduralna apstrakcija se bavi najjednostavnijim programskim jedinicama, procedurama.

Funkcije i procedure

Procedura je entitet koji obavlja izračunavanje. Između procedure i funkcije se pravi razlika. Funkcija obuhvata izraz koji treba da se izračuna, dok procedura obuhvata komande koje treba da se izvrše. Izračunavanja koja su ugrađena u proceduru će se izračunavati kad god se pozove procedura.

Često se dešava da jedan programer pravi proceduru, a da je drugi programeri koriste. Ovi programeri imaju različite uglove gledanja na tu proceduru. Programeri koji samo koriste proceduru se brinu samo o spoljašnjem ponašanju te procedure, odnosno o njenom pozivanju. Oni koji proceduru implementiraju se brinu o tome kako da procedura uradi ono što je potrebno, odnosno brinu o algoritmu.

Efektivnost procedura se dalje proširuje parametrizacijom.

Metodi u objektno orijentisanim programskim jezicima su u suštini procedure, koje se drugačije zovu. Jedina razlika je da je svaki metod pridružen određenoj klasi ili objektu.

Funkcije

Funkcija obuhvata izraz koji treba da se izračuna. Kada se pozove funkcija daje vrednost, koja se naziva rezultatom funkcije. Programer koji koristi funkciju brine samo o tom rezultatu, a ne o tome kako je on dobijen.

Funkcija u jeziku C ili C++ ima sledeći oblik:

T identifikator (FAD1, ... FADN) B

identifkator je identifikator funkcije, FAD je deklaracija formalnog argumenta, T je tip rezultata, a B je blok komandi koje čine telo funkcije. Ovaj blok mora da sadrži najmanje jedan iskaz return E, gde je E izraz tipa T. Funkcija se može pozvati na sledeći način:

identifkator(SA1, .. SAN) gde su SA stvarni argumenti. Ovakav poziv funkcije dovodi do izvršenja bloka B, u kome se na kraju izvršava neki od iskaza return, koji vraćaju rezultat.

Pogledajmo definiciju funkcije u C++-u:

```
float stepen (float x, int n) {
   float p = 1.0;
   for (int i = 1; i <= n; i++)
   p *= x;
   return p;
}
```

Ova funkcija vezuje identifikator stepen sa dva formalna argumenta (x i n) i rezultatom tipa float. Funkcija inače služi za izračunavanje n-og stepena od x, sa pretpostavkom da n nije negativan broj. Obratite pažnju na to da se za izračunavanje koriste lokalne promenljive i iteracija.

Ako se ista funkcija napiše primenom rekurzije, ona izgleda ovako:

```
float power (float x, int n) {
```

```
    if (n == 0)
           return 1.0;
    else
           return x * power(x, n-1);
}
```

Definicije funkcija u C-u ili C++-u mogu da zbune. Telo funkcije je skup komandi, tako da se programeri ohrabruju da prave funkcije sa spoljašnjim efektima. Rezultat funkcije se definiše iskazom return, ali se može desiti da se stigne do kraja funkcije, a da se nije stiglo do iskaza return, u kom slučaju poziv funkcije ne uspeva. Osnovni problem je da telo funkcije mora da vrati vrednost, kao i izraz, ali da je sintaktički u pitanju komanda.

Mnogo je prirodnije da je telo funkcije i sintaktički izraz. To je slučaj u funkcionalnim jezicima, kao što su ML ili HASKELL.

Evo kako izgleda definicija funkcija u HASKELL-u.

```
stepen (x: Float, n: Int) =
if n = 0
then 1.0
else x * power(x, n-1)
```

Ova funkcija vezuje identifikator stepen sa funkcijom. Funkcija vraća isti rezultat kao i ona napisana u C-u, ali je njeno telo sada izraz.

Generalno se poziv funkcije može posmatrati sa dva različita stanovišta:

- Sa stanovišta programera koji funkciju samo koristi. Za njega funkcija samo povezuje argumente i odgovarajući tip rezultata.
- Sa stanovišta programera koji funkciju implementira. Za njega funkcija treba da izvrši komande definisane u telu funkcije i to preko formalnih argumenata. On se brine o algoritmu koji se izvršava u funkciji.

Ako pogledamo dve C++ funkcije koje smo prethodno definisali, onda programer koji koristi funkciju nju vidi samo kao način da se poveže par (x,n) i rezultat. Onaj koji funkciju implementira treba da brine o tome kako da izračuna stepen nekog broja i da vrati rezultat. Druga funkcija (napravljena preko rekurzije) je za onog ko koristi funkciju potpuno ista, ali se za onog ko je implementira pogled menja, jer on sada mora da problem reši na drugačiji način (putem rekurzije).

Procedure

Procedura obuhvata skup komandi koje treba da se izvrše. Prilikom poziva ona ažurira promenljive. Programer koji koristi proceduru brine samo o tim ažuriranjima, a ne o tome kako je to stvarno urađeno.

Procedura se u C-u i C++-u definiše na sledeći način:

void identifkator (FAD1, .. FADn) B

gde je identifikator identifikator te procedure, a FAD formalni argmenti. B je blok komandi koje čine telo funkcije. Procedura se poziva preko iskaza

identifkator(SA1, .. SAn)

gde su SA stvarni argumenti. Ovakav poziv dovodi do toga da se izvrši telo procedure.

Primetićete da u jezicima C i C++, odakle je ovaj primer, procedura predstavlja samo specijalan slučaj definicije funkcije. Razlika je samo u povrtanom tipu, koji je kod procedure void.

Pogledajmo definiciju procedure za sortiranje:

```
void sort (int a[], int l, int r) {
   int minpos; int min;
   for (int i = l; i < r; i++) {
          minpos = i; min = a[minpos];
   for (int j = i+1; j <= r; j++) {
          if (a[j] < a[i]) {
                 minpos = j; min = a[minpos];
```

```
            }
            if (minpos != i) {
                  a[minpos] = a[i]; a[i] = min;
            }
      }
}
```

Ovim se identifikator sort vezuje za proceduru. Programera koji proceduru koristi interesuje samo niz a, koji će biti sortiran po rastućem redosledu. Programer koji treba da napravi ovu proceduru mora da primeni odgovarajući algoritam (u ovom slučaju sortiranje sa selekcijom).

Ako kasnije programer koji je pravio proceduru odluči da promeni algoritam i primeni neki efikasniji, na primer quicksort, on to može uraditi bez problema. Programer koji koristi tu proceduru neće biti ni svestan promena.

Generalno se poziv procedure može posmatrati na sličan način kao poziv funkcije:

- Sa stanovišta programera koji proceduru samo koristi. Za njega procedura samo povezuje argumente i ažuriranje izvesnih promenljivih.
- Sa stanovišta programera koji implementira proceduru. Za njega procedura treba da izvrši komande definisane u telu i to pomoću formalnih argumenata. On se brine o algoritmu koji se izvršava u proceduri.

U objektno orijentisanom programiranju se procedure i funkcije nazivaju metodima. Razlika u odnosu na funkcije u tradicionalnim jezicima je u tome da metodi pripadaju objektima.

Parametri (argumenti) metoda

Objekat sadrži i promenljive i metode. Kada koristite metod često je potrebno da mu date neke informacije, koje govore šta on to treba da uradi. Pogledajmo sledeći primer:

```
Point pt = new Point();
pt.move( 14, 22 );        // pomera tacku pt u x=14, y=22
```

Metod move() se mora koristiti sa dva parametra, koji označavaju x i y koordinatu nove lokacije.

Metodi bez argumenata

Pogledajmo sledeći primer:

```
String str = new String("Probni tekst");
int    len = str.length();
              |     |   |
              |     |   +---- nema argumenata
              |     |
              |     +---- ime metoda koji treba da se izvrši
              |
              +---- referenca na objekat čiji je to metod
```

Drugi iskaz sadrži poziv metoda. Ovo je zahtev da se izvrši metod objekta. Objekat kome se pristupa preko promenljive str sadrži metod length(), koji kada se pozove, vraća broj karaktera stringa.

Metod length() ne traži nikakve argumente. Lista argumenata je u ovom slučaju prazna.

Vratimo se na metod move() klase Point. Ako bismo napisali

```
Point pt = new Point();  // construct a point at x=0, y=0
pt.move();               // pomeranje na novu lokaciju
```

Postavlja se pitanje da li je ovaj kod ispravan? Nije, jer metod nije pozvan na pravi način.

Deklaracija metoda move() u klasi Point izgleda ovako:

```
public void move(int x, int y);   // menja (x,y) objekta tačka
```

Ovaj opis metoda kaže da metod traži dva argumenta:

1. Prvi argument je tipa int. On će postati nova vrednost za x.
2. Drugi argument je tipa int i on će postati nova vrednost za y.

Tačka se koristi da se definiše objekat čiji se metod poziva, kao i metod koji treba pozvati. Lista argumenata snabdeva metod potrebnim podacima. Evo nekoliko primera:

```
Point pointA = new Point();
Point pointB = new Point( 94, 172 );

pointA.move( 45, 82 );

pointB.move( 24-12, 34*3 - 45 );

int col = 87;
int row = 55;
pointA.move( col, row );

pointB.move( col-4; row*2 + 34 );
```

U listi argumenata se može naći i izraz, sve dok je vrednost tog izraza onog tipa koji metod očekuje. Naravno da se pre početka izvršenja metoda prvo izračun vrednost izraza.

Da prođemo još jednom kroz poziv ovog metoda:

```
pointB.move( 24-12, 34*3 - 45 );

      je isto što i:

pointB.move( 12, 34*3 - 45 );

      je isto što i:

pointB.move( 12, 102 - 45 );

      je isto što i:

pointB.move( 12, 57 );
```

U ovom trenutku počinje da se izvršava metod move(). On sada sadrži dva cela broja koja su mu potrebna za rad.

Kada metod počne sa radom, on mora imati tačan broj parametara i svaki parametar mora biti traženog tipa. Ponekad se dešava da se pre početka rada metoda neki od argumenata konvertuju u traženi tip. Ta konverzija se može desiti na jedan od dva načina:

- Eksplicitno, kada programer traži da se vrednost konvertuje, preko operatora koverzije
- Implicitno, kada kompajler može da uradi konverziju, bez gubitka informacija, ili sa vrlo malim gubitkom preciznosti, on će to i uraditi.

Operator eksplicitne konverzije radi na sledeći način:

```
(traženiTip)(izraz)
```

(traženiTip) je nešto poput (int). (izraz) je običan izraz. Ako je to samo jedna promenljiva ne morate je stavljati u zagrade. Sledeći primer pokazuje kako se koristi eksplicitna koverzija. Na engleskom se za eksplicitnu konverziju tipova koristi termin type casting.

```
import java.awt.*;  // uvozi se biblioteka klasa u kojoj je defnisana
                    //  klasa Point
class typeCastEg{
```

```
  public static void main ( String arg[] )  {
    Point pointB = new Point();     // pravi se tačka
    pointB.move( (int)14.305, (int)(34.9-12.6) );
    System.out.println("Nova lokacija:" + pointB );
  }
}
```

U ovom primeru je potrebna eksplicitna konverzija za oba argumenta, pošto se konvertuje realan broj u ceo broj. Kod konverzije realnog broja u ceo broj se decimalni deo odbacuje (ne zaokružuje se).

U prethodnom primeru je konverzija realnog broja u ceo broj dovodila do gubitka informacija, tako da je programer morao da eksplicitno zatraži konverziju.

Kada se konverzija iz jednog u drugi tip može obaviti bez gubitka informacija, kompajler to radi automatski. Na primer, metod move() traži dva parametra tipa int:

```
public void move(int x, int y);  // menja se (x,y) objekta tačka
```

Vrednost tipa int sadrži 32 bita. Vrednost tipa short ima 16 bitova i može se konvertovati u 32-bitnu vrednost bez gubitka informacija. Sledeći program će prema tome, raditi kako treba:

```
import java.awt.*;      //
class autoConvEg1{
  public static void main ( String arg[] )  {
    Point pointB = new Point();     // pravi se tačka u x=0 y=0
    short a=12, b=42;
    pointB.move( a, b );    // vrednosti u listi parametara se
                            // automatski konvertuju u traženi tip int
    System.out.println("Nova lokacija:" + pointB );
  }
}
```

Promenljive a i b se ne menjaju. One su samo upotrebljene da kažu koje se vrednosti konvertuju.

Pravila konverzije

Generalno govoreći, kada postoji mogućnost da se informacije izgube, konverzija iz jednog tipa u drugi se ne može obaviti automatski. Konverzija tipa koji koristi N bitova u podatak koji koristi manje od N bitova nosi rizik gubitka informacija i neće se obaviti automatski. Kompajler odluku o konverziji donosi na osnovu tipa, a ne na osnovu vrednosti koje su tog trenutka prisutne.

Komajler će automatski izvršiti konverziju u sledećim slučajevima:

- Ako se konvertuje celobrojni tip u drugi celobrojni tip, koji koristi više bitova.
- Ako se konvertuje tealan broj u drugi realan broj koji koristi više bitova.
- Ako se konvertuje celobrojni tip u realan broj koji koristi isti broj bitova, može se desiti gubitak preciznosti, ali se to ipak radi automatski.
- Ako se konvertuje celobrojni tip u realan tip sa više bitova.

"Gubitak preciznosti" znači da neke od manje značajnih cifara mogu postati nule, ali će najvažnije cifre i veličina broja ostati isti.

Kompajler neće automatski izvršiti konverziju u sledećim slučajevima:

- Kada se konvertuje celobrojni tip u drugi celobrojni tip sa manje bitova.
- Kada se konvertuje realan tip u drugi realan tip sa manje bitova.
- Kada se konvertuje realan tip u celobrojni tip, sa mogućim gubitkom i preciznosti i vrednosti.
- Konverzija u i iz tipa boolean nikad nije dozvoljena (čak ni eksplicitna)

Što se tiše veličine tipova, evo podsetnika:

tip	broj bitova
byte	8
short	16
int	32
long	64
float	32
double	64

Sledeći primer pokazuje kako biste metodu prosledili argument tipa double. U Javi postoji pripremljena klasa Math u kojoj se nalazi veliki broj matematičkih metoda, kao što su trigonometrijske metode sin, cos, tan itd. U toj klasi postoji statički metod cos, koji prima argument tipa double i računa kosinus ugla. Deklaracija metoda izgleda ovako:

```
public static double cos( double num )
```

Metod bi se pozvao na sledeći način:

```
class CosEg{
  public static void main ( String arg[] )  {
    double x = 0.0;
    System.out.println( "kosinus je:" + Math.cos( x ) );
  }
}
```

Sećate se da pošto je u pitanju statički metod, ne morate imati primerak klase da biste mogli da ga pozovete.

Argumenti metoda

Argumenti koji se šalju u metod su vidljivi samo u metodu za koji su deklarisani. Svaka promenljiva ima svoj domen, odnosno oblast važenja. Oblast važenja argumenata metoda je samo taj metod. Ako pogledamo sledeći primer:

```
class Racun{  . . . .
  private int    stanje;

  . . . .
  void  staviNaRacun( int suma){ // ovde pocinje domen
                                    //promenljive suma
    stanje = stanje + suma;
    // ovde završava domen promenljive suma
  }

  void prikazi() {
    System.out.println(balance+"\t" + suma);//sintaktička greška
  }
}
```

U metodu prikazi ne može se pristupiti argumentu suma, pošto je on izvan domena te promenljive. Kompajler neće kompajlirati ovaj program.

U okviru klase može postojati više metoda sa istim argumentima. To je dozvoljeno jer se domeni ovih promenljivih ne preklapaju.

```
class Racun{  . . . .
  private int    stanje;
```

```
  . . . .
  void  staviNaRacun( int suma){ // ovde pocinje domen
                                      //promenljive suma
    stanje = stanje + suma;
    // ovde završava domen promenljive suma
  }

void skiniSaRacuna( int suma ){
    // domen promenljive suma počinje ovde
    int skini = 15;
    stanje =  stanje - suma - skini ;
    // domen promenljive suma ovde završava
  }
}
```

Dva metoda koriste isti identifikator, suma, za dva različita argumenta. Svaki metod ima svoj argument potpuno odvojen od drugog metoda. Domeni se ne preklapaju tako da iskazi iz jednog metoda ne mogu da vide promenljivu iz drugog.

Lokalne promenljive

Može se desiti da je za rad metoda potrebno da deklarišete i neke nove promenljive. Promenljive koje se definišu unutar samog metoda se nazivaju lokalnim promenljivim. Domen lokalnih promenljivih je metod u kome su deklarisane. Izvan tog metoda im se ne može pristupiti. Te promenljive postoje samo u toku izvršenja metoda. Posle napuštanja metoda one se brišu iz memorije.

U prethodnom primeru je u metodu skiniSaRacuna() deklarisana lokalna promenljiva skini, koja se koristi samo u tom metodu (označava cenu usluga banke). Oblast važenja te promenljive je samo metod skiniSaRacuna().

U okviru metoda možete definisati lokalnu promenljivu sa istim imenom, kao što je neka promenljiva deklarisana na nivou objekta.

```
class Racun{  . . . .
  private int    stanje;

  . . . .
  void  staviNaRacun( int suma){ // ovde pocinje domen
                                      //promenljive suma
    int stanje = 0;
    stanje = stanje + suma;
    // ovde završava domen promenljive suma
  }
```

Ovo nije sintaktička greška (iako je verovatno logička greška). Kompajler će ovaj kod iskompajlirati. Druga deklaracija promenljive stanje (plave boje) kreira lokalnu promenljivu u metodu staviNaRacun(). Domen ove promenljive počinje njenom deklaracijom i završava se na kraju metoda. To znači da se u narednom redu koristi lokalna promenljiva, a ne promenljiva objekta.

Pravilo: Skoro uvek je greška da se koristi lokalna promenljiva sa istim imenom kao promenljiva definisana na nivou klase. Ipak, ovo nije sintaktička greška, tako da vam kompajler neće pomoći u otkrivanju takvih grešaka.

Prosleđivanje parametara po vrednosti

Ako je tip argumenta koji se šalje u metod neki od primitivnih tipova, onda promene koje napravite nad tim argumentom u metodu, nemaju uticaja izvan samog metoda. Taj argument je lokalna kopija, onog što je prosleđeno u metod. Promene utiču samo na tu lokalnu kopiju.

```
class Racun{  . . . .
  private int    stanje;

  . . . .
```

```
  void  staviNaRacun( int suma){ // ovde pocinje domen
                                      //promenljive suma
    stanje = stanje + suma;
    // ovde završava domen promenljive suma
  }

  void prikazi() {
    System.out.println(balance+"\t" + suma);//sintaktička greška
  }
}
class CheckingTester{
  Racun act;
  public static void main ( String[] args )  {
    int suma = 5000;
    act = new Racun( "123-345-99",
        "Mirko Petrovic",  100000 );

    // štampa se "5000"
    System.out.println( "suma:" + suma );

    // poziva se staviNaRacun sa sumom 5000
    act.staviNaRacun(suma);

    // štampa "5000" --- "suma" se nije promenila
    System.out.println( " suma:" + suma);

  }
}
```

Evo kako funkcioniše prosleđivanje po vrednosti:

- Kada se metod pozove, na mestu poziva se zadaje lista vrednosti (stvarne vrednosti argumenata).
- Kada pozvani metod opčne sa radom, te vrednosti se vezuju za listu imena (formalni argumenti) koja je definisana u metodu.
- Pozvani metod koristi te vrednosti u svom radu
- Pozvani metod ne može da preko argumenata pošalje vrednost nazad do mesta poziva.

Prosleđivanje po referenci

Argumenti metoda mogu biti i objekti. Prosleđivanje argumenata funkcioniše slično kao kod prosleđivanja promenljivih primitivnog tipa. Razlika je u tome što se sada šalju reference na objekat. Pošto pozvani metod ima referencu na objekat, on može da pristupa tom objektu i da ga koristi.

Pogledajmo sledeći primer:

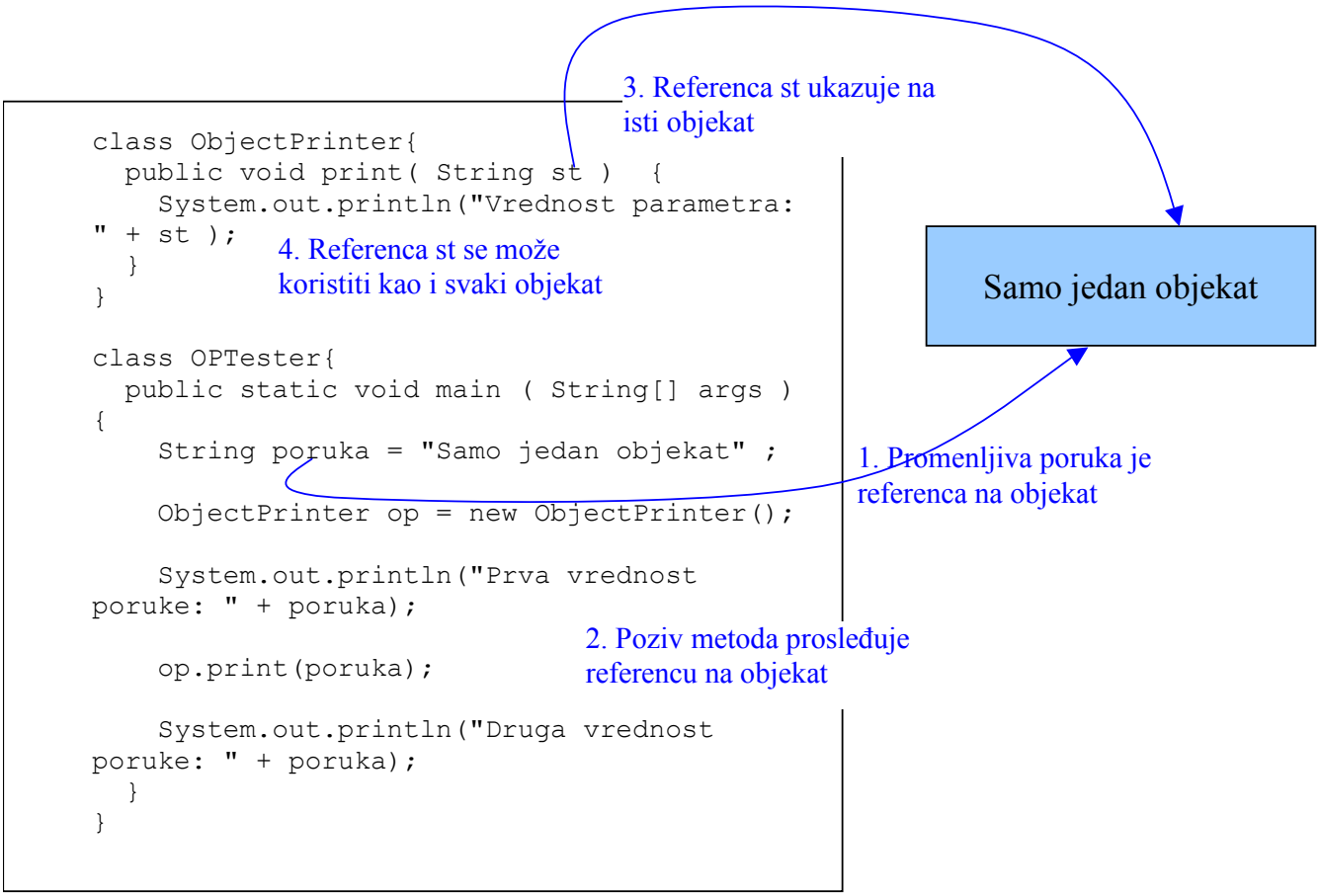
```
class ObjectPrinter{
  public void print( String st )  {
    System.out.println("Vrednost parametra: " + st );
  }
}

class OPTester{
  public static void main ( String[] args )  {
    String poruka = "Samo jedan objekat" ;
```

```
    ObjectPrinter op = new ObjectPrinter();

    System.out.println("Prva vrednost poruke: " + poruka);
    op.print(poruka);
    System.out.println("Druga vrednost poruke: " + poruka);
  }
}
```

Sva tri puta će se odštampati tekst "Samo jedan objekat". To je i logično očekivati jer sva tri puta pristupamo istom objektu.

U prethodnom primeru, kada je u metod print poslat objekat poruka, u stvari je poslata referenca na objekat tipa string. Iako se pravi kopija reference i ona šalje u metod, u pitanju je referenca koja ukazuje na isti objekat.

Vizuelno se prethodni primer može predstaviti ovako:



Pravilo: Ako su objekti argumenti metoda oni se u metod šalju po referenci. To znači da se u metodu može da pristupi originalnom objektu. Promene stanja objekta u metodu dovode do promena i u originalnom objektu, koji je postojao na mestu poziva. Ovo ne važi za argumente koji su primitivni tipovi.

Pogledajmo sada jedan primer u kome se u metodu menja stanje objekta koji je poslat kao argument.

```
class Tacka{
  public int x=3, y=5 ;
  public void print(){
    System.out.println("x = " + x + "; y = " + y );
  }
}

class DupliranjeTacke{
```

```
  public void dvaPuta( Tacka parm )  {
    System.out.println("Ulaz u DupliranjeTacke");
    parm.print() ;
    parm.x = parm.x * 2 ;
    parm.y = parm.y * 2 ;
    parm.print() ;
    System.out.println("Napustanje metoda dvaPuta");
  }
}

class Test{
  public static void main ( String[] args )  {
    Tacka pt = new Tacka ();
    DupliranjeTacke dbl = new DupliranjeTacke ();
    pt.print();
    dbl.dvaPuta( pt );
    pt.print();
  }
}
```

Ako se ovaj program izvrši dobija se sledeći izlaz:

```
x = 3; y = 5
Ulaz u DupliranjeTacke
x = 3; y = 5
x = 6; y = 10
Napustanje metoda dvaPuta
x = 6; y = 10
```

Kao što se vidi u metodu dvaPuta, klase DupliranjeTacke doslo je do promene stanja objekta, koji je poslat kao argument. Ta promena stanja je promenila originalan objekat (to je u stvari isti objekat), što se vidi iz poslednjeg štampanja.

Stvar je programera da li će u metodu menjati objekat koji je prosleđen kao argument. Ponekad to može biti potrebno, a ponekad ne. Programer samo treba da bude svestan da te promene utiču i na originalan objekat i da se ponaša u skladu sa time.

Preklapanje metoda

Preklapanje metoda nastaje kada dva ili više metoda u klasi imaju isto ime, a različitu listu argumenata. Koji metod će se pozvati određuje se u trenutku izvršenja uparivanjem liste deklarisanih argumenata, sa listom stvarnih argumenata, koji se u tom trenutku prosleđuju.

Pogledajmo sledeći primer:

```
class Racun{  . . . .
  private int    stanje;

  . . . .
  void  staviNaRacun( int suma){
    stanje = stanje + suma;
  }
  void  staviNaRacun( int suma, int cenaUsluge){
    stanje = stanje + suma - cenaUsluge;
  }

}
```

Ovo je promenjena klasa od ranije. Pretpostavka je da su nam bila potrebna dva metoda za stavljanje na račun. Jedan se koristi za uobičajeno postavljanje novca na račun, kada banka ne skida novac za svoje usluge i drugi za ostale slučajeve kada se banka uzima svoj deo. Sledeći kod pokazuje kako bi izgledala upotreba ovih metoda:

```
class Test{
  public static void main( String[] args )  {
    Racun bobsAccount = new Racun( "999", "Pera", 100 );
    perinRacun.staviNaRacun( 200 );       // iskaz A
    perinRacun. staviNaRacun ( 200, 25 );   // iskaz B
  }
}
```

Iskaz A poziva prvi metod staviNaRacun, a iskaz B drugi.

Kada nekoliko metoda imaju isto ime, pravilo za određivanje koji se metod poziva glasi: Koristiće se onaj metod čiji stvarni argumenti odgovaraju deklarisanim argumentima.

Na primer poziv

```
    perinRacun.staviNaRacun( 200 );       // iskaz A
```

odgovara deklaraciji

```
    void  staviNaRacun( int suma)
```

pošto broj i tip stvarnih argumenata odgovara broju i tipu deklarisanih argumenata.

Prethodno pravilo se može izraziti i na sledeći način: Pozvaće se onaj metod čija signatura odgovara.

Signaturu metoda čine:

- ime
- broj i tip argumenata

Signatura metoda u jednoj klasi mora biti jedinstvena. Signature dva metoda staviNaRacun() su:

```
    void  staviNaRacun( int suma)
    void  staviNaRacun( int suma, int cenaUsluge)
```

Povratni tip nije deo signature, isto kao ni identifikatori argumenata u deklaraciji metoda. To znači da nije dovoljno da se dva metoda u istoj klasi razlikuju samo po povratnom tipu.

Pogledajmo sledeća dva metoda:

```
int racunaSumu(int a)
float racunaSumu(int b)
```

Pošto je signatura ova dva metoda ista (povratni tip nije deo signature) kompajler neće dozvoliti kompilaciju, odnosno prijaviće grešku.

U prethodnom primeru treba obratiti pažnju na to da imena argumenata nisu deo signature, tako da nije bitno što su u pitanju promenljive a i b.

Preklapanje konstruktora

Sećate se da su konstruktori specijalni metodi koji se pozivaju prilikom kreiranja novih objekata klase. Oni su specijalni po tome što imaju isto ime kao klasa i što se za njih ne zadaje povratni tip. Osim ovih razlika, konstruktori su isti kao i drugi metodi. To znači da i oni mogu da se preklapaju. Preklapanje konstruktora se jako često koristi, jer to predstavlja način da se na različit način inicijalizuju atributi objekta.

Pogledajmo sledeći primer:

```
class Tacka{
  public int x, y ;
  Tacka(int xu, int yu){// prvi konstuktor
          x = xu;
```

```
            y = yu;
    }
    Tacka(Tacka A){// drugi konstruktor
            x = A.x;
            y = A.y;
    }
}
```

Kao što se vidi klasa Tacka ima dva konstruktora. Jedan se koristi za kreiranje nove instance na osnovu dva cela broja, koji predstavljaju koordinate nove tačke, a drugi konstruktor se koristi za kreiranje novog objekta kopiranjem starog. Koji konstruktor će se pozvati određuje se u trenutku poziva na osnovu prosleđenih argumenata.

```
class Test{
   public static void main( String[] args )  {
            Tacka A = new Tacka(2, 3);
            Tacka B = new Tacka(A);
    }
}
```

Kod kreiranja tačke A se poziva prvi konstruktor (sa dva cela broja). Kod kreiranja tačke B se poziva drugi konstruktor, koji novu tačku pravi na osnovu koordinata stare.