ava podsetnik

Upotreba tastature

Preporučujem da koristite američki raspored, čak i ako su vam dirke obeležene srpskim ili nemačkim slovima.

Znaci [] su desno od slova P. Znaci {} su isti tasteri, ali sa shiftom.

Znaci ;’:” su desno od L.

Znaci <>/? su desno od M.

Ceremonija

Za sad će nam svi programi izgledati ovako:

public class ZdravoSvete {

public static void main(String args[]) {

**// ovde dolazi glavni deo našeg programa**

}

}

Ispisivanje

System.out.println("tekst");

System.out.print("tekst");  // bez prelaska u novi red

System.out.println("5 + 3");

System.out.println(5 + 3);

Promenljive

String ime = "Žika";

System.out.print("Ime glasi: ");

System.out.println(ime);

int a = 5;

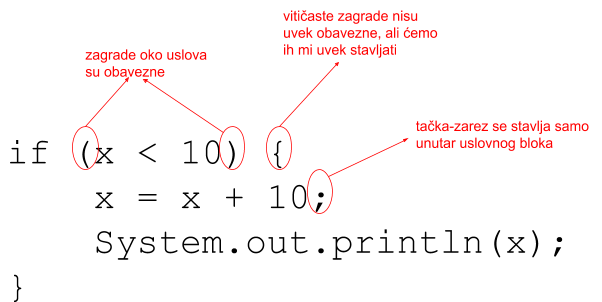
int b = a + 3;

a = b + 10;

b = b + 10 \* a;

poruka = "Korisnik se zove " + ime;

Uslovno grananje



Opšta struktura grananja:

|  |
| --- |
| **if (**uslov**) {**  iskaz 1**;**   iskaz 2**;**  iskaz 3;  **} else {**  iskaz 1**;**  iskaz 2**;**  iskaz 3**;**  **}** |

Obratite pažnju na to gde idu zagrade i tačka-zarezi. U if naredbi nigde nema tačka-zareza van vitičastih zagrada.

Unos podataka od korisnika

Za unos smo koristili Scanner:

|  |
| --- |
| **import java.util.Scanner;**  public class ZdravoSvete {  **static Scanner sc = new Scanner(System.in);**     public static void main(String args[]) {       int x;       System.out.print("Unesite broj: ");  **x = sc.nextInt();**       System.out.println("Uneli ste broj: " + x);     }  } |

Za sada tretirajmo Scanner kao formulu koja se mora uneti na ovakav način.

Linija sa "nextInt" je ona koja sačeka da korisnik programa unese neki broj, i onda smesti taj broj u promenljivu (npr. x).

Programi sa Scannerom su naši prvi programi u kojima (kao programeri) *ne znamo* koje će vrednosti biti upisane u promenljivu. Pišemo programe tako da korisnik može da unese razne brojeve, i da program uvek bude ispravan.

Uslovi za if

Za poređenje brojeva koristimo ove operacije:

|  |  |
| --- | --- |
| veće? | x > y |
| veće ili jednako? | x >= y |
| manje? | x < y |
| manje ili jednako? | x <= y |
| jednako? | **x == y** |
| nejednako? | x != y |

Obratite naročito pažnju na pretposlednji red.

U Javi, jedan znak jednakosti se koristi samo za upisivanje početne ili nove vrednosti u promenljivu, nikada za poređenje dve vrednosti. Za poređenje koristimo dvostruki znak jednakosti. I jako iskusni programeri nekad pogreše oko ovoga!

Kaskadno grananje

Nekad imamo potrebu da testiramo niz uslova jedan za drugim, gde se svaki proverava samo ako prethodni nije bio ispunjen. Ovo je samo poseban slučaj ugneždenih grananja, tako da ga možemo tako i izraziti:

|  |
| --- |
| **if** (n < 10) {     System.out.println("n je jednocifren");  } **else** {  **if** (n < 100) {         System.out.println("n je dvocifren");     } **else** {  **if** (n < 1000) {             System.out.println("n je trocifren";         } **else** {             System.out.println("n ima bar 4 cifre");         }     }  } |

Ali će u ovakvom slučaju biti čitkije ako odstupimo od nekih pravila kojih smo se držali za vizuelno uređivanje koda:

|  |
| --- |
| **if** (n < 10) {     System.out.println("n je jednocifren");  } **else if** (n < 100) {     System.out.println("n je dvocifren");  } **else** **if** (n < 1000) {     System.out.println("n je trocifren";  } **else** {     System.out.println("n ima bar 4 cifre");  } |

Ovaj kod ima potpuno isto značenje. Vizuelna razlika je u tome što smo, isključivo u slučaju kad else grana ima samo jedan iskaz, i taj iskaz je novi (ugneždeni) if, odlučili da ugneždeni iskaz nastavimo u istom redu, neuokviren vitičastim zagradama, i da kod koji je njemu podređen ne uvlačimo dodatno u desno.

Neki tipovi podataka

int, s kojim smo se već susreli, je ceo broj.

float i double su realni brojevi; koristićemo samo double.

char je pojedinačni znak unutar stringa.

boolean je logički tip koji može da ima samo vrednosti true (tačno) ili false (netačno).

Nad brojevima možemo da vršimo operacije + - \* /  
Nad celim brojevima imamo i operaciju % koja računa ostatak pri deljenju (dok / računa samo celobrojni deo količnika).

Nad logičkim vrednostima primenjujemo operacije && (i), || (ili) i ! (negacija).

Poređenje brojeva sad možemo da shvatimo i kao operaciju, na isti način na koji su oduzimanje ili množenje operacije: uzmemo dva broja, izvršimo npr. operaciju <= i dobijemo rezultat, koji doduše nije broj nego je logička vrednost "tačno" ili "netačno", zavisno od toga da li je prvi broj stvarno bio manji-ili-jednak drugom.

Uslov koji smeštamo u if naredbu može biti bilo šta logičkog (tj. boolean) tipa: npr. logička promenljiva, ili rezultat operacije && ili rezultat nekog metoda koji vraća logičku vrednost.

Metodi (ili funkcije)

|  |
| --- |
| static **int** **kvadrat(int n)** {     int pomocna;     pomocna = n;     pomocna = pomocna \* n;  **return** pomocna;  } |

Ključnu reč static za sad stavljamo ispred svakog metoda.

int koji sledi posle toga je tip vrednosti koju metod *vraća*.

Sledi ime koje dajemo metodu.

Iza toga u zagradi sledi ono što metod prihvata kao ulaznu vrednost. Ima istu strukturu kao deklaracije promenljive: prvo tip, zatim ime.

Unutar vitičastih zagrada navodimo sve korake koji se izvršavaju kad se metod *pozove*. (U ovom primeru namerno ima više koraka nego što je stvarno potrebno.)

Kao poslednji korak dajemo posebni iskaz return *neki\_izraz*;  - Ovo znači da se dalje izvršavanje metoda prekida, i da se vrednost tog izraza *vraća* glavnom programu, da s njom uradi šta god hoće.

Gde god se u glavnom programu pojavi izraz poput kvadrat(3) ili kvadrat(x \* y - 1), vrednost tog izraza će biti ono što metod vrati.

Metod može da se napiše tako da traži više od jednog parametra na ulazu, ili nijedan parametar. Vrednost koju metod vraća je samo jedna (i tad pišemo njen tip) ili nijedna (i tad umesto tipa pišemo reč void).

I main je metod.

Java program se u stvari izvršava tako što se potraži metod koji se upravo tako zove (*glavni* metod programa) i pozove. Drugi metodi će se pozvati samo iz main-a, posredno ili neposredno.

I System.out.println je metod.

Nismo ga mi napisali, ali je on deo Jave. Dajemo mu kao parametar ono što treba ispisati, on to izvrši i vrati kontrolu glavnom programu (bez ikakve povratne vrednosti).

(Tačke između reči ćemo objasniti uskoro.)

Kako se piše while petlja

|  |
| --- |
| **while** (uslov) {     iskaz1;     iskaz2;     iskaz3;  } |

Po strukturi dosta liči na if (bez else-a). Telo petlje se izvršava više puta, dok god uslov ne postane netačan. Pošto se uslov proverava pre izvršavanja petlje, može se desiti da nikad ne bude ispunjen i da se telo ne izvrši nijednom.

Da bi petlja imala smisla, potrebno je da uslov bar nekad počne kao tačan; u protivnom, petlja se nikad neće ni izvršiti. Potrebno je i da se unutar petlje stvari promene na takav način da uslov koji je počeo kao tačan pre ili kasnije postane netačan; u protivnom, imaćemo beskonačnu petlju iz koje se nikad ne izlazi.

Još jedan način da se izađe iz petlje

|  |
| --- |
| while (uslov) {     iskaz1;     if (drugi\_uslov) {         iskaz2;  **break;**     }     iskaz3;  } |

Ključna reč break iskače iz petlje.

Obično stoji unutar if-a, kao u ovom primeru, jer bi *bezuslovno* iskakanje iz petlje značilo da se drugi deo petlje nikad i ne izvrši.

Brojačka (for) petlja

Ona je samo skraćeni način da se zapiše jedan poseban oblik while petlje, koji se jako često susreće u praksi:

|  |  |
| --- | --- |
| formula | for (**početni izraz; kontrolni izraz; brojački izraz**) {     telo petlje ... ;  } |
| isto što i | **početni izraz;**  while (**kontrolni izraz**) {     telo petlje ... ;  **brojački izraz**;  } |
| npr. | for (**i = 1; i <= 20; i = i + 1**) {     System.out.println(i);  } |
| isto što i | **i = 1**;  while (**i <= 20**) {     System.out.println(i);  **i = i + 1;**  } |

Bilo koji od tri izraza se može izostaviti. (Izostavljen kontrolni izraz znači da se ne izlazi iz petlje - mora se iskoristiti break, inače bi petlja bila beskonačna.)

Izrazi mogu biti bilo koji. Na primer, umesto i = i + 1 možemo staviti i = i \* 2.

Skraćeni oblici i++ odnosno ++i imaju isti efekat na promenljivu kao i = i + 1.

do-while petlja

|  |
| --- |
| **do** {     iskazi; ...  } **while (**uslov**);** |

Ovo je varijacija while petlje koja proverava uslov na kraju. Neke razlike su:

* mora se početi rečju do
* telo petlje će se sigurno izvršiti bar jednom (jer se uslov ne proverava dok se ne prođe kroz telo)
* na kraju je neophodan tačka-zarez, jer while klauzula dolazi posle {} bloka.

Klase

|  |
| --- |
| public **class ImeKlase** {     // atributi/polja/članovi se deklarišu isto kao promenljive:     int brojEkrana;     double potrošnjaPoSatu;     String nazivStadiona;     // konstruktor:     public **ImeKlase**(parametri) {         // inicijalizacija         // nema return!     }     // drugi metodi:     void snimi() {         // ...     }     int najmanjiBrojač(parametri) {         // ...     }  }  // ← ovde završava definicija klase |

Promenljivu tipa neke klase deklarišemo kao:

   ImeKlase imePromenljive;

Novi objekat (primerak, instancu) te klase stvaramo pozivom operatoru new i konstruktoru:

   new ImeKlase(parametri)

Da bismo stvorili promenljivu i u nju smestili novokonstruisani objekat, kombinujemo:

  ImeKlase imePromenljive = new ImeKlase(parametri);

Ali objekat možemo i dobiti kao povratnu vrednost nekog metoda (bez neposrednog konstruisanja), ili možemo novokonstruisani objekat poslati kao parametar drugom metodu (bet smeštanja u promenljivu).

Koja korist od klasa?

Već kad smo napisali jednu klasu, imamo prednost da su svi detalji njenog funkcionisanja objedinjeni na jednom mestu i da se lako mogu naći.

Drugi delovi programa se mogu pisati bez detaljnog znanja o tome kako klasa funkcioniše iznutra - treba samo da znaju koje poslove i upite mogu da joj povere i kakve će rezultate dobiti.

Kada menjamo način rada klase, ili dodajemo neku novu funkcionalnost, dovoljno je da to uradimo na jednom mestu. Ne moramo detaljno da poznajemo sve ostale delove svih programa koji je koriste.

Vidljivost

Koristili smo na dosta mesta ključnu reč public, koja označava da datoj klasi, polju ili metodu može pristupiti bilo koji deo Java programa. Kad želimo da ograničimo pristup, za sad smo koristili reč private koja znači da samo ta klasa ima pristup:

|  |
| --- |
| public class Krug {  **private** double radius;     public **getRadius**() {         return radius;     }     public **setRadius**(double radius) {         this.radius = radius;     }  } |

Upotreba "pristupnih metoda" *(accessor methods,* odnosno *getters and setters)* je nekad zamorna, i ne treba je nekritički ubacivati svuda, ali nam ovde omogućava da od drugih klasa koje koriste našu "sakrijemo" detalje implementacije - one se ne smeju oslanjati na ideju da naša klasa interno ("ispod haube") koristi prečnik ili poluprečnik, nego samo treba da se oslanjaju na to da mogu spolja dobiti ili postaviti vrednost bilo prečnika bilo poluprečnika.

Postoje još dva međunivoa pristupa, koje za sad nismo koristili: protected znači da pristup ima sama klasa i sve njene *potklase*, dok odsustvo bilo kakve oznake pristupa znači da pristup ima svaki deo programa koji je u istom *paketu*. Ovim odnosima ćemo se pozabaviti kasnije.

Nizovi

Promenljive i vrednosti nizovnog tipa sadrže ne jednu nego više vrednosti nekog osnovnog tipa. Promenljivu deklarišemo kao:

   String**[]** niz1;  
    String niz2**[]**;

(Oba načina su ispravna.) Na isti način se izražava da parametar ili povratna vrednost metoda treba da bude niz.

"Vrednost" koja čini ceo niz moramo kreirati slično objektima:

   new String[10]

Dužina niza ne mora biti fiksan broj - može biti bilo koja (celobrojna) promenljiva ili izraz.

Spajanjem ove dve formule dobijamo punu deklaraciju koja stvara novi niz i "pamti" ga u datoj promenljivoj:

   String niz[] = new String[10];

Pojedinačnom članu niza pristupamo preko njegovog *indeksa* tj. rednog broja: niz[5]. Možemo mu dodeliti novu vrednost ili pročitati tekuću, isto kao s nezavisnom promenljivom. Indeksi kreću od 0, tako da niz od 20 elemenata ima indekse od 0 do 19. Dužinu niza možemo saznati čitanjem njegovog atributa length.

Tipičan šablon prolaska kroz sve elemente niza koristi for petlju:

|  |
| --- |
| for (int i = 0; i < **niz.length**; i++) {     // neke radnje nad članom **niz[i]**  } |

(Primetite da je unutar petlje i uvek manje od length, nikad jednako.)

Pojednostavljena (ili "napredna") for petlja

Za najčešći oblik prolaženja kroz nizove nam stoji na raspolaganju još jedna forma for petlje. Kao što je osnovni for poseban slučaj while petlje, tako je i napredni for poseban slučaj osnovne for petlje:

|  |
| --- |
| String boje[] = {"crvena", "plava", "zelena"};  for (**String b : boje)** {     System.out.println(**b**);  } |

Ova petlja obavezno prolazi kroz sve članove, redom od početnog do krajnjeg, tako da nemamo mogućnost da prođemo samo kroz deo niza ili drugačijim redosledom. Unutar petlje nemamo pristup brojaču ili indeksu. Jedini "brojač" je promenljiva koja uzima pojedinačne vrednosti *članova* niza. (Dakle, u gornjem primeru "crvena", "plava", "zelena", a ne 0, 1, 2).

Nasleđivanje

Nova klasa može da *nasledi* (ili *proširi*) drugu, postojeću klasu:

|  |
| --- |
| class Korisnik {     public String ime, prezime;     public String punoIme() { ... }     public void prijavi() { ... }     public void odjavi() { ... }     // ...  } |
| class Administrator **extends Korisnik** {     int nivoOvlašćenja;     public zaustaviSistem() { ... }     public pokreniSistem() { ... }  } |

Klasu koja nasleđuje zovemo *naslednik, dete-klasa, potklasa* (eng. *subclass*)*.*

Klasu koja je nasleđena zovemo *osnovna, roditelj-klasa*ili *natklasa* (eng. *superclass*).

Naslednička klasa automatski sadrži sva polja i metode roditeljske klase, tako da treba dodati samo elemente koji su potrebnu za novu ili drugačiju funkcionalnost. U gornjem primeru, svaki *Administrator* je ujedno i *Korisnik*, tako da npr. već ima ime, i može se prijaviti. Ali Administrator ima i neka svojstva i oblike funkcionalnosti koje ostali Korisnici nemaju.

Koja korist od nasleđivanja?

Nasleđivanje je glavna tehnika *ponovne upotrebe koda* (eng. *code reuse*) u Javi. Ako smo već na jednom mestu definisali kako funkcioniše prijavljivanje na sistem za Korisnike, ne želimo da to ponavljamo za Administratore. Glavna prednost nije ušteda rada, nego osiguravanje da se zaista koristi isti kod, tako da se lakše izbegnu ili pronađu greške, a i ako se u budućnosti proces menja, da bude dovoljno da se promeni na jednom mestu.

U bilo koju promenljivu, parametar ili povratnu vrednost metoda koja je deklarisana sa tipom natklase (npr. Korisnik) može se uvek upisati objekat koji u stvari pripada potklasi (npr. Administrator). Kôd koji koristi tu promenljivu/parametar/itd. ne može da pristupa onim njenim poljima i metodima koji važe samo za Administratore (jer on "ne zna" da je taj korisnik u stvari i Administrator) ali može da ga potpuno tretira kao Korisnika, jer Administrator garantovano ima sva svojstva Korisnika:

|  |
| --- |
| **Korisnik** k = new **Administrator**( ... );  System.out.println(**k.prezime**);  // ispravno, jer *k* jeste i *Korisnik* i stoga ima *prezime*  System.out.println(**k.nivoOvlašćenja**);  // neispravno, jer iako taj objekat ima polje *nivoOvlašćenja,*  // kôd koji mu pristupa kao promenljivoj tipa *Korisnik* to ne zna -  // on se samo sme osloniti na to da *k* ima svojstva *Korisnika.* |

Nasleđivanje i konstruktori

Konstruktori se ne nasleđuju, ali se očekuje da se konstruktori nasledničke klase oslone na roditeljske:

|  |
| --- |
| class Administrator extends Korisnik {     // ...     public Administrator(String ime, String prezime) {         // pozovi konstruktor kao za Korisnika:  **super**(ime, prezime);         // obavi ostatak inicijalizacije, koji         // važi samo za Administratora:         nivoOvlašćenja = 0;     }  } |

Ključna reč super znači "ovaj objekat, ali posmatran kao da pripada roditeljskoj klasi". Ovako upotrebljena u konstruktoru ima efekat da se pozove roditeljski konstruktor s datim parametrima.

Ako konstruktor nasledničke klase ne počne super pozivom, tretira se kao da je postojao poziv super() - tj. roditeljski konstruktor bez parametara. Ako takav roditeljski konstruktor ne postoji, Java prijavljuje grešku, tako da tad moramo ručno dodati poziv *nekom* roditeljskom konstruktoru.

Zamena metoda

Pored dodavanja novih metoda, naslednička klasa može i da zameni *("override")* neke od nasleđenih metoda:

|  |
| --- |
| class Korisnik {     // ...     String **punoIme**() {         return ime + " " + prezime;     }  } |
| class Administrator extends Korisnik {     // ...  **@Override**     String **punoIme**() {         return ime + " " + prezime + " (admin.)";         // ili još bolje:         // return super.punoIme() + " (admin.)";     }  } |

Anotacija @Override nije obavezna, ali pomaže da se uhvate greške (npr. u imenu metoda).

Ako je klasa zamenila nasleđene metode, onda će pozivanje tih metoda nad objektima te klase uvek da izvrši nad njima onaj metod koji je sama njihova klasa definisala, *bez obzira na to* da li onaj ko poziva metod zna kojoj klasi objekat zaista pripada. Na primer:

|  |
| --- |
| public static ispišiPunoIme(**Korisnik** k) {     System.out.println(k.**punoIme**());  } |

Ovaj pomoćni metod posmatra svoj parametar samo kao običnog korisnika, i ne može znati kojoj on klasi zaista pripada. Ali kad pozove metod punoIme, to ne znači da će pozvati metod iz klase Korisnik - pozvaće se metod koji pripada stvarnoj klasi kojoj objekat pripada.

Možemo reći da sam objekat "zna" koja je njegova klasa i njegov metod, a pozivajući metod *nad* nekim objektom (dakle kao objekat.metod() a ne pomoćniMetod(objekat)) program poverava objektu da sam izvrši onu verziju metoda koja mu odgovara.