1. ***Osnovni pojmovi: programiranje, program, programski jezik. Programi nižeg i višeg nivoa. Računar, računarska platforma. Računari i programi: podela sadržaja memorije, (hipotetički) primer mašinskog/asemblerskog programa***

\* Pod (računarskim) **programiranjem** obično se podrazumeva čovekova aktivnost (odnosno, aktivnost **programera**) usmerena ka rešenju nekog konkretnog problema na računaru

\* Problem se obično rešava konstrukcijom **algoritma** za rešenje problema, a algoritam se zapisuje posredstvom **programskog jezika**

\* Po jednoj od najopštijih definicija programskog jezika njegova uloga je da obezbedi konstrukcije (i načine) za organizovanje “izračunavanja” na računaru

\* Tako organizovano izračunavanje nazivamo program izračunavanja, odnosno kraće: **program**

\* Kao sastavni deo rešenja problema programi obično sadrže interakciju sa korisnikom, kao i drugim računarima i uređajima, jednim terminom zvani **ulazno-izlazne operacije**

\* Interakcija sa korisnikom obavlja se kroz **korisnički interfejs**

- Konzolni (*console*, *command prompt*)

- Grafički (*graphical user interface* – GUI)

\* Komunikacija sa drugim računarima i uređajima uključuje:

- Rad sa fajlovima

- Razmenu informacija preko mreže

- Štampanje

\* Razlikujemo programe **nižeg i višeg nivoa** (tačnije, hijerarhiju nivoa gde se viši nivoi oslanjaju na usluge nižih)

\* U programe **nižeg nivoa** spadaju operativni sistemi (Windows, Linux, Android...) i njihovi delovi, drajveri, itd.

- Programi “bliži” hardveru računara

- Operativni sistemi se obrađuju na istoimenim kursevima (OS 1 i 2)

\* U programe **višeg nivoa** spadaju korisnički programi (aplikacije), npr. Word, Skype, Grand Theft Auto...

- Programi “dalji” i nezavisniji od hardvera

- Na ovom kursu se fokusiramo na ovu vrstu programa, sa konzolnim korisničkim interfejsom

- GUI će se raditi na drugim kursevima (Objektno-orijentisano programiranje 1 i 2, Web dizajn...)0

\* Pojam **računar** (*computer*) obuhvata sve uređaje koji se sastoje od tri osnovne komponente

- Centralni procesor (*central processing unit* – CPU)

- Memorija

- Mogućnost ulaza/izlaza (*input*/*output* – IO)

\* Uz to postoje dodatni zahtevi koji se tiču mogućnosti procesora u smislu izračunljivosti (kursevi: Teorijske osnove informatike 2, Analiza algoritama)

**\* Računarska platforma**: svaki uređaj koji ima gore navedene elemente i poseduje operativni sistem koji omogućava pravljenje i instalaciju korisničkih aplikacija širokih mogućnosti

Primeri računara:

- Personalni (desktop) i notebook računari

- Tableti

- “Pametni” telefoni i televizori

\* Nisu računari:

- Klasični kalkulatori

- Stari telefoni i televizori

\* Na granici (jesu računari, ali ne i platforme):

- Ruteri

- Specijalizovani industrijski kontroleri

\* Osnovna podela sadržaja memorije i IO uređaja:

- Programski kod (instrukcije) u mašinskom jeziku

-Podaci

\* U osnovi, SVI sadržaji su predstavljeni binarnim brojevima (nizovima 0 i 1)

\* Program (na primer, .exe fajl) se učitava sa ulaznog uređaja, CPU ga izvršava koristeći memoriju, učitavajući podatke sa ulaznih uređaja i šaljući rezultat(e) na izlaz

Jedan prost (hipotetički) mašinski program od 3 instrukcije:

00000100111100111010001010

00000101000100100111100010

00101100101010111111001111

\* Isti program u (hipotetičkom) asemblerskom jeziku:

**LOAD** A

**ADD** B

**STORE** C

\* Program odgovara naredbi dodele A = B + C, gde su A, B i C imena memorijskih lokacija

***2. Programski jezici: mašinski, asemblerski, viši. Java, JVM, izvršavanje Java programa***

**Mašinski jezik** je jedini jezik koji računar “razume”, tj. prograne napisane u mašinskom jeziku jedine direktno može da izvršava

\* Prvi programeri (1950-ih) su pisali programe u mašinskom jeziku

**\* Nedostaci programiranja u mašinskom jeziku** su brojni:

- Užasno je teško pisati programe

- Takođe je teško razumeti ih, nalaziti greške, i menjati

\* Programerima teško da čitaju sopstveni programski kod nakon izvesnog vremena, kamoli tuđ

- Svaki model računara ima drugačiji mašinski jezik

\* Da bi se program mogao izvršavati na drugom modelu računara, treba ga u osnovi ponovo napisati

\* Ubrzo su se pojavili **asemblerski jezici**

- Pojedinačne binarne instrukcije, pa i grupe instrukcija, zamenjene simboličkim, a programi se pišu u formi teksta

- Poseban program, **asembler** (*assembler*) zadužen je za prevođenje programa pisanog u asemblerkom jeziku u mašinski jezik

- Programiranje donekle olakšano

- I dalje različiti modeli računara imaju različite asemblerske jezike

\* Ovaj “niski nivo” računara i programiranja obrađuje se na kursu Organizacija računara

\* Asemblerski jezici su pokrenuli proces **apstrakcije** programa i programiranja od arhitekture konkretnog računara

**\* Viši programski jezici** predstavljaju sledeći korak u tom smeru

\* Konstrukcije viših jezika su u potpunosti (ili bar u velikoj meri) nezavisne od računara, a kao i kod asemblerskih jezika programi su u formi teksta

- Npr. naredba dodele A = B + C će u memorijsku lokaciju (promenljivu) sa imenom A smestiti broj dobijen sabiranjem brojeva sadržanih u memorijskim lokacijama sa imenima B i C

\* Poseban program, **kompajler** (*compiler*) zadužen je za prevođenje programa pisanog u višem programskom jeziku u mašinski jezik

\* Broj viših programskih jezika u aktivnoj upotrebi meri se u hiljadama

\* Primeri viših programskih jezika: FORTRAN, C, C++, BASIC, Pascal, Modula-2, LISP, Smalltalk, Python, **Java**...

**\* Java** je veoma popularan i rasprostranjen viši programski jezik, zvanično predstavljen javnosti 23. maja 1995. godine

\* Razvio ga je *Sun Microsystems* u okviru projekta *Green*, kao jezik i platformu za kontrolisanje rada potrošačkih elektronskih uređaja

\* U Sun-u su uskoro uvideli da je novi jezik pogodan i za Web

\* Glavni prodor je napravljen kada je *Netscape* dobio licencu na Javu i u septembru 1995. godine predstavio svoj novi brauzer *Netscape Navigator 2.0* sa podrškom za Java aplete

\* Korporaciju Sun je u međuvremenu preuzeo softverski gigant *Oracle* (poznat po sistemima za upravljanje bazama podataka)

\* Rasprostranenosti Jave veoma doprinosi Android operativni sistem, gde su programi (*app*-ovi) bazirani na Javi

\* Kao i kod drugih viših programskih jezika, Java kod se prevodi u mašinski jezik, i to poseban jezik nazvan Java **bajtkod** (*bytecode*)

\* Međutim, ne postoji fizički računar koji “razume” Java bajtkod, već se radi o “virtuelnom” računaru koji se zove **Javina virtuelna mašina** (*Java Virtual Machine* – JVM)

\* Fizički računar zamenjuje program koji ga simulira (rad CPU, memorije, ulaz/izlaz) i naziva se **JVM interpreter**, ili prosto JVM

\* Implementacije JVM postoje za sve značajnije računarske platforme

**\* Prednost virtuelnog pristupa:** Java programi kompajlirani u bajtkod na jednoj platformi (trebalo bi da) se mogu izvršavati bez izmena na svakoj platformi za koju postoji implementacija JVM

**\* Mana virtuelnog pristupa:** manja efikasnost programa

\* Različiti aspekti kompajliranja, interpretiranja, virtuelnih mašina itd. obrađuju se na kursevima Programski jezici i Konstrukcija kompajlera

\* U dijagramu, “javac” i “java” su konzolni programi koji dolaze u sastavu Java Development Kit (JDK)

\* Moderna okruženja za razvoj programa(*integrated development environment* – IDE) obično sakrivaju ove korake pod jednim dugmetom “Run”, ali oni su i dalje prisutni

***3. Osnovni elementi Java programa, struktura Java programa, jedinica prevođenja***

***Osnovni elementi Java programa***

\* Osnovni elementi jezika Java:

- Reči (lekseme) • Identifikatori • Literali • Specijalni simboli • Rezervisane reči

- Praznine

- Komentari

***Identifikatori***

\* Identifikatori služe za imenovanje promenljivih, metoda, klasa, i drugih konstrukcija u programu

- Primeri: broj, main, System, out

\* Razlikuju se velika i mala slova

- Primer: identifikatori broj i Broj nisu isti

\* Identifikatori se grade od slova, cifara i nekoliko specijalnih znakova:

- Počinju slovom, ili znakom \_ ili $

\* Rezervisane reči (npr. class, void, int, itd.) ne mogu se koristiti kao identifikatori

\* Identifikatori mogu biti proizvoljne dužine

\* Java-slovo je svaki znak za koji metod Character.isJavaLetter vraća vrednost true

\* Java-cifra je svaki znak za koji metod Character.isJavaLetter vraća vrednost false, a metod Character.isJavaLetterOrDigit vraća vrednost true

***Konvencije za Java identifikatore***

\* Pored pravila za formiranje ispravnih Java identifikatora, postoje i konvencije (preporuke) kako imenovati različite konstrukcije jezika

\* Imena klasa treba da počinju velikim slovom

Primer: OvoJeIspravnoImeKlase

\* Imena promenljivih, metoda i paketa treba da počinju malim slovom Primeri: broj, x, deli

\* Ako je identifikator sastavljen od više reči, koristiti velika slova za njihovo odvajanje (sem eventualno kod prve reči)

Primeri: charArray, fileNumber, ClassName

\* Izbegavati korišćenje \_ kao prvog znaka identifikatora

Primeri: \_read, \_write

***Literali***

**\* Literali** su konstantne vrednosti koje direktno uključujemo u program

\* Tipovi literala u Javi:

- **Brojevi**

•Celi brojevi (*integer*) •Realni brojevi (brojevi sa zarezom, *floating point*)

- **Logički literali** (*boolean*)

- **Znakovi** (*character*)

- **Stringovi** (*string*)

***Literali: celi brojevi***

\* Celi brojevi se mogu navoditi u nekoliko različitih brojnih sistema

\* Decimalni zapis (sa osnovom 10) – standardan zapis

Primer: 42

\* Heksadecimlni zapis (osnova 16) – počinje znakovima 0x

Primer: 0x2A

\* Oktalni zapis (osnova 8) – počinje sa 0

Primer: 052

\* Binarni zapis (osnova 2) – počinje sa 0b

Primer: 0b101010

***Literali: realni brojevi***

\* Strogo uzevši, *racionalni* brojevi

\* Dve vrste zapisa: standardni i “naučni” (*scientific*)

\* Standardni zapis – decimalni broj, tačka, decimalni broj

Primer: 583.45

\* Naučni zapis – mantisa + eksponent

Primer: 5.8345E2 (5.8345 ∙ 102)

***Literali: logički***

\* Dve moguće vrednosti: true i false

***Literali: znakovi***

\* Znakovni literali predstavljaju znakove iz Unicode rasporeda

\* Unicode znakovi su reprezentovani sa 16 bita (2 bajta, odnosno 65536 mogućih vrednosti), tako da pored slova engleske abecede, cifara i standardnih znakova interpunkcije (stari 8-bitni ASCII raspored) mogu da se predstave znakovi iz drugih jezika

\* Znakovni literal navodi se kao znak okružen jednostrukim navodnicima

Primer: znakovni literal za slovo a se piše **'**a**'**

\* Specijalni znakovi koji se ne mogu jednostavno “otkucati” na tastaturi se predstavljaju *escape* znacima

Primeri: **'**\n**'** znak za novi red

**'**\b**'** znak za brisanje (*backspace*)

***Literali: stringovi***

\* String literali su sekvence znakova između dvostrukih navodnika

\* Primeri: **"**Ovo je string**"**, **"**Uvod u programiranje**"**

\* Escape znaci unutar stringova i znakovnih literala:

**\n** *line feed* LF

**\r** *carriage return* CR

**\t** horizontalni tabulator HT

**\b** *backspace* BS

**\"** dvostruki navodnici **"**

**\'** jednostruki navodinici **'**

**\\** *backslash* **\**

**\broj** redni broj znaka (ASCII raspored, od 0 do 255) u oktalnom zapisu

**\u*xxxx,*** gde je ***xxxx*** heksadecimalni broj – redni broj Unicode znaka

***Specijalni simboli i rezervisane reči***

**\* Specijalni simboli** su operatori i separatori u jeziku, tj. znaci ili nizovi znakova koji imaju neko posebno značenje u programu

-Neki operatori : < > + += ++ &&

-Separatori : { } [ ] () ; , .

**\* Rezervisane** (ili **ključne**) **reči** su posebne lekseme jezika koje imaju tačno određeno značenje

\* Ključne reči se koriste za opis naredbi i drugih konstrukcija programa programa i deo su sintakse programskog jezika

***Rezervisane reči***

\* Rezervisane reči se ne mogu koristiti kao identifikatori u programu

\* Rezervisane reči u Javi:



***Praznine***

**\* Praznine** (engl. *whitespace*) u Javi su posebni znaci u koje spadaju:

-Oznaka za kraj reda

-Oznaka za tabulator

-“Prazan” znak (*space*)

-Kontrolni znak za štampač (*form feed*)

\* Zanemaruju se prilikom prevođenja

\* Strogo uzevši, služe samo za razdvajanje leksema

\* Utiču na lepši izgled i bolju preglednost programa

***Komentari***

**\* Komentari** su delovi programa koji ne utiču na smisao i suštinu programa i zanemaruju se prilikom prevođenja (kompajler se prema njima odnosi kao da su praznine)

\* U Javi postoje **tri vrste komentara**:

-**Obični komentari** - tekst koji se nalazi između simbola /\* i \*/

-**Jednolinijski komentari** - tekst koji se nalazi između simbola // i kraja reda

-**Dokumentacioni komentari** - tekst koji se nalazi između simbola /\*\* i \*/. Ovaj komentar služi za automatsko generisanje dokumentacije programa (pomoću alata Javadoc) pa je bitno gde se navodi komentar, i tekst komentara mora biti strukturiran po određenim pravilima

***Primeri komentara***

**\* Obični komentari:**

/\* Ovo je komentar

koji se prostire na vise linija \*/

/\* A ovaj komentar je u jednoj liniji \*/

**\* Jednolinijski komentari:**

// Ovo je jednolinijski komentar

// A ako zelimo vise linija

// moramo pre svake linije komentara staviti //

**\* Dokumentacioni komentar:**

/\*\* Slican je obicnom komentaru, s tim da moze

sadrzati tagove, npr.

@author Milos Radovanovic

@version 1.0

koji imaju posebno znacenje pri generisanju

dokumentacije \*/

***Struktura Java programa***

\* Program u Javi se sastoji od deklaracija klasa, interfejsa i nabrojivih tipova podataka

-Klase, interfejsi i nabrojivi tipovi su referencijalni tipovi podataka

\* Mora biti deklarisana bar jedna klasa sa metodom čije je zaglavlje

public static void main(String[] args)

\* Sve deklaracije se mogu nalaziti u jednom fajlu ili mogu biti raspoređene u više fajlova

\* Fajl koji čini Java program nazivamo jedinica prevođenja

\* Jedinica prevođenja treba da se imenuje isto kao i klasa koja sadrži main() metod

***Jedinica prevođenja***

**\* Paketi** se uvode radi grupisanja referencijlnih tipova

-Primer: package uup;

-Na početku fajla (jedinice prevođenja) svi referencijalni tipovi deklarisani u tom fajlu svrstavaju se u isti paket sa zadatim imenom (ako se ovaj deo ne navede ref. tipovi će pripasti tzv. anonimnom paketu)

\* U delu **uvoz** navodimo ime paketa i ime koje se iz njega uvozi

-Primer: import java.util.HashSet;

**\* Deklaracija tipa** – deklarisanje klasa, interfejsa i nabrojivih tipova

***4. Značaj tipova podataka, prosti tipovi u Javi, promenljive***

***Značaj tipova podataka***

**\* Veća preglednost i jasnoća programa** - programer radi sa apstraktnim pojmovima brojeva, znakova, i sl., ne opterećujući se načinom njihovog unutrašnjeg predstavljanja

**\* Provera usklađenosti tipova** - prevodilac proverava kako se koriste konstante i promenljive u izrazima, u skladu sa operatorima koji su dozvoljeni za pojedine tipove podataka

**\* Veća efikasnost prevedenog koda** *-* pošto je usklađenost tipova već proverena, Java prevodilac može generisati efikasniji kod koji ne vrši nepotrebne provere u toku izvršavanja programa

***Prosti tipovi podataka u Javi***

\* U Javi postoji osam prostih (primitivnih) tipova podataka:

-boolean (logičke vrednosti)

-char (znakovi)

-byte (celi brojevi)

-short (celi brojevi)

-int (celi brojevi)

-long (celi brojevi)

-double (realni brojevi)

-float (realni brojevi)

***Tip boolean***

\* Tip boolean predstavlja dve logičke vrednosti: true(tačno) i false(netačno)

\* Primer: boolean nasao = true;

\* U gornjem primeru se deklariše promenljiva sa imenom nasao, tipa boolean, i dodeljuje joj se vrednost true

\* Ovo je tip podataka sa najmanje mogućih vrednosti, što mu ne umanjuje važnost

\* Logičke vrednosti se dobijaju i kao rezultat primene relacionih operatora na vrednosti celobrojnih ili realnih tipova podataka

***Tip char***

\* Tip char (skraćeno od engl. *character*) predstavlja jedan znak iz Unicode rasporeda

\* Literali ovog tipa navode se korišćenjem jednostrukih navodnika ' '

\* Primeri:

'a' // Slovo a ' ' // Razmak '\t' // Znak za tabulator

'\'' // jednostruki navodnik '\"' // dvostruki navodnik

\* Znakovni literali obrađeni su na prethodnom predavanju

\* Tip String, koji služi za predstavljanje sekvenci znakova, nije prost tip (u pitanju je klasa)

-Kod njegovih literala koriste se dvostruki navodnici " "

\* Operatori nad tipom char su isti kao za celobrojne tipove, štaviše char se pri primeni operatora i tretira kao celobrojni tip (sa vrednostima koje prestavljaju redne brojeve znakova u Unicode rasporedu)

***Celobrojni tipovi podataka***

**\* byte** – 1 bajt (8 bita), celi brojevi iz intervala

[-27, 27-1] = [-128, 127]

**\* short** – 2 bajta (16 bita)

[-215, 215-1] = [-32768, 32767]

**\* int** – 4 bajta (32 bita)

[-231, 231-1] = [-2147483648, 2147483647]

**\* long** – 8 bajtova (64 bita)

[-263, 263-1] = [-9223372036854775808, 9223372036854775807]

\* Ako je n broj bitova kojim se predstavlja ceo broj, tada su moguće vrednosti celobrojnog tipa [-2n-1, 2n-1-1]

***Celobrojni literali***

\* Celi brojevi se mogu navoditi u nekoliko različitih brojnih sistema, sa osnovama 10, 16, 8 i 2

\* Decimalni zapis (sa osnovom 10) – standardan zapis

Primer: 42

\* Heksadecimlni zapis (osnova 16) – počinje znakovima 0x

Primer: 0x2A

\* Oktalni zapis (osnova 8) – počinje sa 0

Primer: 052

\* Binarni zapis (osnova 2) – počinje sa 0b

Primer: 0b101010

\* Ako želimo da naglasimo da je literal tipa long, to radimo dodavanjem L ili l na kraj literala (pri čemu se ne preporučuje l zbog sličnosti sa cifrom 1)

Primeri: 42L 0x2AL 052L 0b101010L

\* Inače se podrazumeva da je literal tipa int (mada su usklađeni i sa drugim brojevnim tipovima)

***Operatori nad celobrojnim tipovima***

**\* Relacioni operatori**: < , <=, > , >= , == , !=

**\* Aritmetički operatori**: + , - , \* , / , % , ++ , --

**\* Operatori nad bitovima**: << , >> , >>> , ~ , & , | , ^

**\* Operatori dodele**: = , += , -= , \*= , /= , %= , <<= , >>=

>>>= , &= , |= , ^=

**\* Uslovni operator** ? :

**\* “*Cast*” operatori za konverziju tipova:** (imeTipa)

**\* Operator konkatenacije stringova** +

**\* Klase za rad sa tipovima**: Character, Byte, Short, Integer, Long**,** klasa Math – matematičke funkcije

***Tipovi realnih brojeva***

**\*** Uzimaju vrednosti iz skupa racionalnih brojeva

-float – jednostruka preciznost (4 bajta)

-double – dvostruka preciznost (8 bajtova)

**\*** Zapis realnih literala:

-Standardni zapis – decimalni broj, tačka, decimalni broj

Primer: 583.45

-Naučni zapis – mantisa + eksponent

Primer: 5.8345E2 (5.8345 ∙ 102)

-Na kraju može da stoji oznaka tipa (D, d, F, f)

Primeri: 583.45D 5.8345E2F

**\*** ako se oznaka ne navede podrazumeva se tip double

***Tipovi realnih brojeva***

**\* Operatori** (kao i kod celobrojnih tipova) – relacioni operatori, aritmetički operatori, operatori nad bitovima, operatori dodele, uslovni operator, “*cast*” operatori, operator konkatenacije stringova

**\*** Tri **posebne konstante** (nastaju samo kao rezultat izvršavanja nekih matematički nedefinisanih operacija) :

-Infinity (minus beskonačno)

Infinity (beskonačno)

NaN (*'Not a number'* – nije broj)

**\*** Klase Float i Double – korisne metode i način predstavljanja realnih tipova preko klasa

***Promenljive***

**\* Promenljiva** predstavlja jedan podatak u memoriji računara, kojem se pristupa preko imena (identifikatora)

**\*** Svaka promenljiva ima:

-Ime (po kojem se promenljiva prepoznaje)

-Vrednost (čuva se u memoriji)

-Tip podataka (određuje vrednosti koje promenljiva može imati)

**\* Imena** se grade od slova, cifara i znakova \_ i $ , po pravilima zapisivanja identifikatora sa prošlog predavanja

**\* Tip podataka** može biti bilo koji prosti tip, ili referencijalni tip (klasa, niz, nabrojivi tip)

***Deklaracija i inicijalizacija***

**\*** U Javi, svaka promenljiva se pre korišćenja mora deklarisati, na primer:

int br**;** // Ime: br, tip: int

double x**;** // Ime: x, tip: double

String s**;** // Ime: s, tip: String

boolean nasao; // Ime: nasao, tip: boolean

**\*** Moguće je promenljivama dodeliti početnu vrednost prilikom deklaracije:

int br = 42**;** double x = 5.5**;**

String s = "Neki tekst"**;** boolean nasao = true;

**\*** A moguće je i deklarisati/inicijalizovati više promenljivih odjednom:

int br, n, i, j**;**

double x = 1.0, y = 2.0, z = 3.0**;**

**\*** Ako se pri deklaraciji ne inicijalizuje, promenljiva prostog tipa imaće podrazumevanu (*default*) vrednost tog tipa koja odgovara broju 0, odnosno (imeTipa)0: 0, 0L, 0.0, '\0', false...

(sem ako je u pitanju lokalna promenljiva, o njima više kasnije)

***Deklaracija i inicijalizacija: preporuke***

**\*** Dobra je praksa inicijalizovati promenljive prilikom deklaracije

**\*** Imena promenljivih treba birati tako da opisuju sadržaj promenljive, odnosno ono što promenljiva predstavlja u rešenju problema

-Primer: promenljivu koja sadrži konačnu ocenu studenta na ispitu je preporučljivo imenovati ocena, ili još bolje konacnaOcena, a ne recimo o, x ili ejfvbewjhrb

**\*** Preporučuje se da se svaka promenljiva deklariše u posebnom redu

-Primer: preferira se

double ispit = 0.0**;**

double test = 10.0**;**

double konacnaOcena = 0.0**;**

u odnosu na

double ispit = 0.0, test = 10.0,konacnaOcena= 0.0**;**

***Štampanje***

**\*** Vrednost promenljive (odnosno izraza) može se štampati pozivom standardnih metoda

System.out.println()

**\*** Kao argument u zagradama prosleđuje se vrednost koja se štampa, a koja može biti bilo kog tipa podataka

**\*** Metode štampaju odgovarajuću reprezentaciju date vrednosti kao String

**\*** Razlika između metoda je što System.out.println() dodaje znak za kraj reda ('\n') na kraj ispisa.

***5. Operatori: relacioni, aritmerički, nad bitovima***

***Operatori***

**\* Relacioni operatori (operatori poređenja)**

**==** (jednakost)

**!=** (nejednakost)

**<** (manje)

**<=** (manje ili jednako)

**>** (veće)

**>=** (veće ili jednako)

**\* Aritmetički operatori**

**+** (unarni plus)

**-** (unarni minus)

**\*** (množenje)

**/** (deljenje)

**%** (ostatak pri deljenju)

**+** (sabiranje)

**-** (oduzimanje)

**++** (prefiksni ili postfiksni operator uvećanja)

**--** (prefiksni ili postfiksni operator smanjenja)

***Aritmetički operatori ++ i --***

**\*** Unarni operatori:

-Uvećanja (inkrementacija): ++

-Umanjenja (dekrementacija): --

**\*** Koriste se uz promenljivu nekog brojevnog tipa i uvećavaju, odnosno umanjuju vrednost promenljive za 1

**\*** Primer:

++broj

radi isto što i

broj = broj + 1

**\*** Operatori ++ i -- mogu da se koriste u prefiksnom i postfiksnom obliku

-Primeri: prefiksni oblik ++broj --broj

postfiksni oblik broj++ broj--

**\*** Efekat na vrednost promenljive je isti

**\*** Razlika je u vrednosti celog izraza (koji uključuje ime promenljive i operator)

-Postfiksni operator: vrednost izraza je originalna vrednost promenljive

-Prefiksni operator: vrednost izraza je uvećana/umanjena vrednost promenljive

**\*** Treba paziti da izrazi u kojima se koriste operatori ++ i -- budu kratki i jasni

class IncDecOperatori2

**\* Operatori nad bitovima**

**~** negacija (invertovanje) bitova

**<<** pomeranje bitova ulevo

**>>** pomeranje bitova udesno vodeći računa o predznaku broja

**>>>** pomeranje bitova udesno ne vodeći računa o predznaku broja

*Primeri:*

**System.out.println(~0); // štampa -1**

**System.out.println(~7); // štampa -8**

**System.out.println(1 << 3); // štampa 8**

**System.out.println(-1 << 3); // štampa -8**

**System.out.println(17 >> 3); // štampa 2**

**System.out.println(-17 >> 3); // štampa -3**

**System.out.println(17 >>> 3); // štampa 2**

**System.out.println(-17 >>> 3); // štampa 536870909**

**System.out.println(1 << 35); // štampa 8**

***Relacioni operatori: primer***

class RelacioniOperatori {

public static void main(String[] args) {

int i = 37;

int j = 42;

int k = 42;

System.out.println("Vrednosti promenljivih:");

System.out.println(" i = " + i);

System.out.println(" j = " + j);

System.out.println(" k = " + k);

// Vece

System.out.println("Vece:");

System.out.println(" i > j = " + (i > j)); // false

System.out.println(" j > i = " + (j > i)); // true

System.out.println(" k > j = " + (k > j)); // false

// Vece ili jednako

System.out.println("Vece ili jednako:");

System.out.println(" i >= j = " + (i >= j)); // false

System.out.println(" j >= i = " + (j >= i)); // true

System.out.println(" k >= j = " + (k >= j)); // true

// Manje

System.out.println("Manje:");

System.out.println(" i < j = " + (i < j)); // true

System.out.println(" j < i = " + (j < i)); // false

System.out.println(" k < j = " + (k < j)); // false

// Manje ili jednako

System.out.println("Manje ili jednako:");

System.out.println(" i <= j = " + (i <= j)); // true

System.out.println(" j <= i = " + (j <= i)); // false

System.out.println(" k <= j = " + (k <= j)); // true

// Jednako

System.out.println("Jednako:");

System.out.println(" i == j = " + (i == j)); // false

System.out.println(" k == j = " + (k == j)); // true

// Nejednako

System.out.println("Nejednako:");

System.out.println(" i != j = " + (i != j)); // true

System.out.println(" k != j = " + (k != j)); // false

}

}

***Izlaz:***

Vrednosti promenljivih:

i = 37

j = 42

k = 42

Vece:

i > j = false

j > i = true

k > j = false

Vece ili jednako:

i >= j = false

j >= i = true

k >= j = true

Manje: i < j = true

j < i = false

k < j = false

Manje ili jednako:

i <= j = true

j <= i = false

k <= j = true

Jednako:

i == j = false

k == j = true

Nejednako:

i != j = true

k != j = false

***Aritmetički operatori: primer***

class AritmetickiOperatori {

public static void main(String[] args) {

int i = 37;

int j = 42;

double x = 27.475;

double y = 7.22;

System.out.println("Vrednosti promenljivih:");

System.out.println(" i = " + i);

System.out.println(" j = " + j);

System.out.println(" x = " + x);

System.out.println(" y = " + y);

// Sabiranje

System.out.println("Sabiranje:");

System.out.println(" i + j = " + (i + j));

System.out.println(" x + y = " + (x + y));

// Oduzimanje

System.out.println("Oduzimanje:");

System.out.println(" i - j = " + (i - j));

System.out.println(" x - y = " + (x - y));

// Mnozenje

System.out.println("Mnozenje:");

System.out.println(" i \* j = " + (i \* j));

System.out.println(" x \* y = " + (x \* y));

// Deljenje

System.out.println("Deljenje:");

System.out.println(" i / j = " + (i / j));

System.out.println(" x / y = " + (x / y));

// Ostatak pri deljenju

System.out.println("Ostatak pri deljenju:");

System.out.println(" i % j = " + (i % j));

System.out.println(" x % y = " + (x % y));

// Mesanje tipova

System.out.println("Mesanje tipova:");

System.out.println(" j + y = " + (j + y));

System.out.println(" i \* x = " + (i \* x));

}

}

***Izlaz:***

Vrednosti promenljivih:

i = 37

j = 42

x = 27.475

y = 7.22

Sabiranje:

i + j = 79

x + y = 34.695

Oduzimanje:

i - j = -5

x - y = 20.255000000000003

Mnozenje:

i \* j = 1554

x \* y = 198.36950000000002

Deljenje:

i / j = 0

x / y = 3.805401662049862

Ostatak pri deljenju:

i % j = 37

x % y = 5.815000000000002

Mesanje tipova:

j + y = 49.22

i \* x = 1016.575

***6. Operatori: logički, dodele, specijalni, ostali***

***Operatori***

**\* Logički operatori**

**!** logička negacija

**&&** logička konjunkcija **&** konjunkcija nad bitovima

**||** logička disjunkcija **|** disjunkcija nad bitovima

**^** ekskluzivna disjunkcija nad bitovima

**\*** Operator ! je unaran, dok su svi ostali binarni

**\*** Operatori s leve strane primenljivi su isključivo na operandima tipa boolean

**\*** Operatori s desne strane su istovremeno i logički operatori (primenljivi na tip boolean), i operatori koji rade na bitovima (primenljivi na brojevnim tipovima)

**\*** Neka je a operand (promenljiva ili izraz) tipa boolean

**\*** Istinitosna tablica za operator !:

!a

|  |  |
| --- | --- |
| a | rezultat |
| true | false |
| false | true |

**\*** Operator ! daje suprotan rezultat u odnosu na operand

**\*** Neka su a i b operandi (promenljive ili izrazi) tipa boolean

**\*** Istinitosne tablice za operatore && i &, odnosno || i |

a&&b a||b

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| a | b | rezultat |  | a | b | Rezultat |
| true | true | True |  | true | true | True |
| false | true | False |  | false | true | True |
| true | false | False |  | true | false | True |
| false | false | false |  | false | false | false |

**\*** Za tip boolean operatori && i &, odnosno || i | daju iste rezultate

**\*** Postoje važne **razlike**:

- && i || se primenjuju samo na tip boolean

- & i | se mogu primeniti na svim brojevnim tipovima, i tada rade nad

pojedinačnim bitovima, na primer:

System.out.println(7 & 9); // 1

System.out.println(7 | 9); // 15

- && i || se evaluiraju **lenjim izračunavanjem** (*short-circuiting*), što znači da ako se vrednost izraza može zaključiti na osnovu prvog operanda, drugi operand se neće ni izračunavati; kod & i | se uvek izračunavaju svi operandi (**vredno izračunavanje**)

**\*** Istinitosna tablica za operator ^

a^b

|  |  |  |
| --- | --- | --- |
| a | b | rezultat |
| True | True | False |
| True | False | True |
| False | False | True |
| False | True | False |

**\*** Operator ekskluzivne disjunkcije ^ daje rezultat true ako je tačno jedan operand true, a drugi false

**\*** Pošto je neophodno izračunati vrednosti oba operanda da bi se došlo do rezultata, lenjo izračunavanje nema smisla (i zato ne postoji operator ^^)

***Operatori dodele***

**=** dodela

**+=** operator dodele sa prethodnom primenom operatora +

**-=** operator dodele sa prethodnom primenom operatora -

**\*=** operator dodele sa prethodnom primenom operatora \*

**/=** operator dodele sa prethodnom primenom operatora /

**%=** operator dodele sa prethodnom primenom operatora %

**<<=** operator dodele sa prethodnom primenom operatora <<

**>>=** operator dodele sa prethodnom primenom operatora >>

**>>>=** operator dodele sa prethodnom primenom operatora >>>

**&=** operator dodele sa prethodnom primenom operatora &

**|=** operator dodele sa prethodnom primenom operatora |

**^=** operator dodele sa prethodnom primenom operatora ^

**\*** Osnovni operator = koristi se u obliku: promenljiva = vrednost

**\*** Promenljiva sa leve strane znaka = mora biti već deklarisana (ili se operator koristi prilikom deklaracije)

**\*** Vrednost sa desne strane može biti neki **literal** (npr. 2), **promenljiva**, **poziv metoda**, odnosno u opštem slučaju **izraz** koji kombinuje literale, promenljive i pozive metoda

**\*** Način izvršavanja: prvo se izračuna vrednost izraza sa desne strane =, zatim se ta vrednost dodeli promenljivoj sa leve strane

**\*** Takođe, cela konstrukcija promenljiva = vrednost predstavlja izraz koji ima vrednost jednaku dodeljenoj vrednosti

Primer: int i;

int j = (i = 22) + 8;

-Ovu mogućnost u principu treba izbegavati, jer može dovesti do grešaka koje se lako prave a teško uočavaju

**\*** Operatori oblika op= , gde je op neki od navedenih operatora (+, -, ...) koriste se u obliku: promenljiva op= vrednost

**\*** Izvršavaju se isto kao promenljiva = promenljiva op vrednost

**\*** Primer:

int i = 2;

i += 2; // i = i + 2;

i \*= 3; // i = i \* 3;

i %= 5; // i = i % 5;

**\* Specijalni operatori**

**?:** uslovni operator

**(*imeTipa*)** eksplicitna konverzija tipa

**+** konkatenacija stringova

***Uslovni operator ?:***

**\*** Jedini **ternarni operator** u Javi

-Potrebno mu je proslediti tri operanda

**\*** Koristi se u obliku:

izraz1 ? izraz2 : izraz3

gde je izraz1 tipa boolean, a ostali izrazi mogu biti bilo kog tipa

**\*** Izvršava se na sledeći način:

-Izračuna se vrednost izraz1

-Ako je ta vrednost true, vrednost celog izraza dobija se izračunavanjem vrednosti izraz2

-U protivnom, vrednost celog izraza dobija se izračunavanjem vrednosti izraz3

**\*** Izračunavanje izraz2 i izraz3 se radi po potrebi, tj. “lenjo”

***Ostali operatori***

**instanceof** – pripadnost referencijalnom tipu

**.** (tačka) – pristup članu klase, paketa...

**[]** (uglaste zagrade) – pristup elementu niza

**new** – kreiranje instance klase

***7. Prioritet operatora. Konverzija prostih tipova***

**Operatori: prioritet**

\* Svi operatori razvrstani su po prioritetu, tako da je za svaki dobro formiran izraz tačno poznato kojim se redosledom izračunavaju vrednosti operanada

\* Prioritet operatora se menja korišćenjem zagrada ( i )

\* Primer: vrednost izraza

6 % 2 \* 5 + 4 / 2 + 88 - 10

izračunava se kao da su zagrade stavljene na sledeći način:

((((6 % 2) \* 5) + (4 / 2)) + 88) - 10

Redosled izračunavanja može se potpuno promeniti:

6 % ((((2 \* 5) + 4) / (2 + 88) - 10))

|  |  |
| --- | --- |
| **Operator** | **Komentar** |
| . [] *new pozivMetoda()* | Operatori najveceg prioriteta |
| -- ++ | Postfiksni operatori |
| *(imeTipa)* ~ ! -- ++ + - | Unarni operatori, prefiksni operatori |
| \* / % | Mnozenje, deljenje, ostatak |
| + - | Sabiranje, konkatenacija, oduzimanje |
| << >> >>> | Pomeranje bitova |
| < > <= >= instanceof | Relacioni operatori |
| == != | Ispitivanje jednakosti |
| & | Konjunkcija |
| ^ | Ekskluzivna disjunkcija |
| | | Disjunkcija |
| && | Logicka konjunkcija |
| || | Logicka disjunkcija |
| ?: | Uslovni operator |
| = \*= /= %= += -= <<= >>= >>>= &= ^= |= | Operatori dodele imaju najmanji prioritet |

***Konverzija prostih tipova***

\* Često je u programu (tj. izrazima) potrebno “mešati” vrednosti različitih tipova podataka

-Ovo je rađeno i u mnogim primerima koje smo videli, kao što je recimo izraz 2e13f + 7 \* 9.8

\* Ista vrednost (npr. broj 7, posmatran kao broj a ne Java literal) se različito predstavlja u memoriji (na nivou bitova) u različitim brojevnim tipovima podataka

-Kao int: 0000000000000000000000000000111

-Kao float: 1000000111000000000000000000000

\* S obzirom da vrednost izraza može biti samo jednog tipa, neophodno je da se vrednosti različitih tipova podataka **konvertuju**, tj. prebace iz jedne reprezentacije u drugu

\* Razlikujemo dve vrste konverzije prostih tipova:

-Proširujuće proste konverzije

-Sužavajuće proste konverzije

\* **Proširujuće proste konverzije** su:

-iz tipa **byte** u tipove **short**, **int**, **long**, **float** ili **double**,

-iz tipa **short** u tipove **int**, **long**, **float** ili **double**,

-iz tipa **char** u tipove **int**, **long**, **float** ili **double**,

-iz tipa **int** u tipove **long**, **float** ili **double**,

-iz tipa **long** u tipove **float** ili **double**,

-iz tipa **float** u tip **double**.

\* Ne dolazi do gubitka informacija, jer važi sledeće:

-tipovi su međusobno kompatibilni,

-ciljni tip je veći od izvornog tipa

\* **Sužavajuće proste konverzije** su:

-iz tipa **byte** u tip **char**

-iz tipa **short** u tipove **byte** ili **char**

-iz tipa **char** u tipove **byte** ili **short**

-iz tipa **int** u tipove **byte**, **short** ili **char**

-iz tipa **long** u tipove **byte**, **short**, **char** ili **int**

-iz tipa **float** u tipove **byte**, **short**, **char**, **int** ili **long**

-iz tipa **double** u tipove **byte**, **short**, **char**, **int**, **long** ili **float**

\* Može doći do gubitka informacija, pri konverziji se odseca decimalni deo (konverzija realni-celi) i viši bajtovi broja (konverzije celi-celi)

\* Ove konverzije programer mora eksplicitno naznačiti korišćenjem cast operatora

-Primer: int n = (int)53.7;

\* Proširujuću prostu konverziju po pravilu Java kompajler može da reguliše automatski, bez intervencije programera, ali treba biti svestan njenog postojanja i pravila

-Pogrešna očekivanja mogu dovesti do grešaka u kodu

-Nepotrebna konverzija može usporiti program

\* Primer: 2e13f + 7 \* 9.8

-Literal 7 je tipa int, a literal 9.8 tipa double, pa je podizraz 7 \* 9.8 tipa double i vrši se konverzija iz int u double

-Literal 2e13f je tipa float, i konvertuje se u double da bi tip celog izraza bio tipa double

-Da je izraz bio zapisan 2e13 + 7.0 \* 9.8 ne bi bilo konverzije

**Pravila za automatsko unapređenje (promociju) tipova** koja se primenjuju u izrazima:

\* **Pravilo unarne numeričke promocije** (primena unarnih operatora)**:** vrednost tipa byte, short ili char se menja proširujućom prostom konverzijom u tu istu vrednost tipa int

\* **Pravilo binarne numeričke promocije** (primena binarnih operatora)**:**

-Ako je jedan operand tipa double onda se i drugi konvertuje u tip double

-Inače, ako je jedan operand tipa float onda se i drugi konvertuje u tip float

-Inače, ako je jedan operand tipa long onda se i drugi konvertuje u tip long

-Inače, ako oba operanda već nisu tipa int, onda se konvertuju u tip int

**Primer:**

short s = 42;

s = -s;

s = s + 1;

Šta nije u redu sa datim kodom?

Zbog automatske promocije tipova, mora se upotrebiti cast operator:

s = (short)-s;

s = (short)(s + 1);

zato što su izrazi -s i s + 1 promovisani u tip int

***8. Naredbe: izraz, izraz-naredba, blok, prazna naredba. Blok: deklaracija i vidljivost lokalne promenljive***

***Naredbe***

\* Izraz, izraz-naredba, blok

\* Prazna naredba

\* Naredbe grananja: if, if-else, switch

- Omogućuju grananje izvršavanja, odnosno izbor dela koda koji će da se izvrši

\* Naredbe ponavljanja (petlje): while, do-while, for

- Omogućuju da se deo koda izvrši više puta

\* Naredbe skoka: break, continue, return

- Omogućuju prekid izvršavanja koda i nastavak izvršavanja na drugom mestu

***Izraz***

\* Izraz u Javi je:

-Literal

-Lokalna promenljiva

-Element niza

-Poziv metoda

-Pristup polju klase ili objekta

-Kreiranje novog objekta pomoću new operatora

-Primena ključne reči this

-Izraz u običnim zagradama

-Primena nekog od operatora (osim operatora new) na operande koji su izrazi odgovarajućeg tipa

-Pristup objektu tipa Class kojim se predstavlja neki tip

***Izraz-naredba***

\* Neke vrste izraza u Javi mogu da se nalaze u programu na mestu gde se očekuje naredba (npr. kao jedna od naredbi u telu metoda main)

\* Tako korišćene izraze nazivamo izraz-naredbe

\* Izraz-naredbe u Javi su:

-Naredba dodele

\* Primena operatora dodele: osnovni =, i sa prethodnom primenom drugih operatora +=, \*=, ...

-Primena operatora ++ i -- prefiksno i postfiksno

-Poziv metoda

-Kreiranje novog objekta pomoću new operatora

\* Delovi izraz-naredbi (npr. deo posle operatora dodele) mogu biti i proizvoljni izrazi, ali osnova mora biti neka od navedenih vrsta izraza

\* Izračunata vrednost izraza koji se koristi kao naredba se gubi – izraz se izračunava zbog efekta koji izračunavanje ima na promenljive, itd.

***Prazna naredba***

\* Ne proizvodi nikakvu akciju niti efekat

\* U nekim situacijama može biti korisna kod složenih naredbi (for, while...) kada moramo da navedemo narebu, a ne želimo

\* Primere ispravnog korišćenja navešćemo kasnije, ali je u principu treba izbegavati zbog mogućnosti pravljenja suptilnih grešaka

***Blok***

\* Blok se sastoji od sekvence naredbi i deklaracija promenljivih (i deklaracija lokalnih klasa) navedenih između vitičastih zagrada {}

\* Blok se u programu može naći na bilo kom mestu gde se očekuje (jedna) naredba (i otuda oblici bez i sa {} naredbi if, for, itd.)

\* Dve osnovne namene bloka:

-Grupisanje naredbi

-Ograničenje oblasti važenja lokalnih promenljivih

***Blok: deklaracija lokalne promenljive***

\* U Javi, svaka promenljiva se pre korišćenja mora deklarisati

\* Lokalna promenljiva se deklariše u okviru nekog bloka

\* Takve promenljive smo do sad i koristili:

int br**;** double x**;**

String s**;** boolean nasao;

\* Moguće je promenljivama dodeliti početnu vrednost prilikom deklaracije, gde se sa desne strane operatora = može nalaziti izraz:

int br = 45 - 3**;** double x = 5.5 + 2.7**;**

String s = "Neki tekst"**;** boolean nasao = true;

\* A moguće je i deklarisati/inicijalizovati više promenljivih odjednom:

int br, n, i, j**;**

double x = 1.0, y = 2.0, z = 3.0**;**

\* Lokalne promenljive se moraju inicijalizovati pre prvog korišćenja njihovih vrednosti

\* Pre deklaracije promenljive može se staviti ključna reč final

\* Takva promenljiva se može inicijalizovati samo jednom (ne obavezno pri deklaraciji, mada je preporučljivo)

\* Tako se u stvari dobija **konstanta**, odnosno promenljiva čija se vrednost ne menja

\* Primeri: final double PI = 3.14;

final double DVA\_PI = 2 \* PI;

final int THE\_ANSWER = 42;

final String JA = "Taj i taj";

***Blok: vidljivost lokalne promenljive***

\* Lokalna promenljiva (promenljiva deklarisana u nekom bloku) se može koristiti (vidljiva je) od mesta na kojem se deklariše, uključujući i deo eventualnog odeljka za inicijalizaciju, pa sve do kraja bloka.

***9. Naredbe grananja***

\* Kontrola toka programa se grana u zavisnosti od nekih uslova, tj. rezultata izračunavanja nekog izraza

\* Naredbe grananja u Javi:

-if naredba

-if-else naredba

-switch naredba

***Naredbe grananja: if***

\* Specificira da će se data naredba (ili blok naredbi) izvršiti ako i samo ako je vrednost datog logičkog izraza jednaka true

\* Efekat je da tok izvršavanja programa može da “krene” nekom “granom” ili ne

\* Oblici:

if (izraz) if (izraz) {

naredba; naredba;

...

naredba;

}

***Naredbe grananja: if***

\* Primeri:

if (x == 2)

System.out.println("x je dva");

if ('a' <= c && c <= 'z') {

System.out.println("malo slovo");

brSlova++;

}

***Naredbe grananja: if-else***

\* Specificira koja će se od dve date naredbe (ili blokova naredbi) izvršiti u zavisnosti od vrednosti datog logičkog izraza

\* Efekat je da tok izvršavanja programa “bira” jednu od ponuđenih grana na osnovu logičkog izraza

\* Oblici:

if (izraz) if (izraz) {

naredba; naredba;

else ...

naredba; naredba;

}

else {

naredba;

...

naredba;

}

\* Moguće je i kombinovati pojedinačnu naredbu u jednoj grani i blok u drugoj

\* Primeri:

if (x < 0)

absX = -x;

else

absX = x;

if (i > 10 || i < -10) {

System.out.println("van opsega");

i = 0;

j++;

}

else {

System.out.println("unutar opsega");

j--;

}

***Naredbe grananja: preporuke***

\* Naredbe navedene u granama treba uvući u odnosu na samu if / if-else naredbu (2, 4 znaka razmaka, tabulator... važna je konzistentnost)

\* Da bi se izbegla konfuzija, preporučljivo je uvek koristiti blokove (tj. navoditi naredbe u granama u vitičastim zagradama {}), čak i kad u grani želimo da navedemo samo jednu naredbu

-Primer:

if (x < 0) if (x < 0) {

absX = -x; absX = -x;

else }

absX = x; else {

absX = x;

}

***Naredbe grananja: ugneždavanje***

\* *Jedna od* naredbi u prvoj grani if(-else) naredbe može opet da bude if(-else) naredba, čime se postiže ugneždavanje

\* Primer:

if (delilac != 0) {

if (deljenik > 1000) {

System.out.println("Deljenik je van opsega");

}

}

else {

System.out.println("Delilac ne sme biti 0");

}

***Naredbe grananja: if-else-if***

\* *Jedina* naredba u drugoj grani if-else naredbe može da bude nova if(-else) naredba, čime se na koncizan način postiže kaskadno vezivanje if-else naredbi (lestvica)

\* Primer:

if (bodovi >= 85) {

System.out.println("Odlicno!!!");

}

else if (bodovi >= 75) {

System.out.println("Vrlo dobro!");

}

else if (bodovi >= 65) {

System.out.println("Dobro.");

}

else if (bodovi >= 55) {

System.out.println("Dovoljno.");

}

else {

System.out.println("Zao mi je...");

}

***Naredbe grananja: switch***

\* Specificira koje će se naredbe izvršiti na osnovu vrednosti celobrojnog izraza

\* Efekat je da tok izvršavanja programa direktno “skoči” na deo koda koji odgovara zadatoj vrednosti izraza

\* Osnovni oblik:

switch (izraz) {

case vrednost1:

naredba;

...

naredba;

break;

...

case vrednostN:

naredba;

...

naredba;

break;

}

\* Vrednosti se navode u okviru “slučajeva”, odnosno labela (ključna reč case) kao konstantni izrazi, tj. izrazi koje kompajler može da izračuna u toku prevođenja programa

\* Naredba break označava izlazak iz switch naredbe, odnosno skok na prvu naredbu posle }

\* Nakon poslednjeg niza naredbi break nije neophodan

\* Ako izraz nije jednak ni jednoj od zadatih vrednosti , neće se izvršiti ni jedna od datih naredbi

\* Oblik sa default granom:

switch (izraz) {

case vrednost1:

naredba;

...

naredba;

break;

...

case vrednostN:

naredba;

...

naredba;

break;

default:

naredba;

...

naredba;

}

\* Ako izraz nije jednak ni jednoj od zadatih vrednosti, izvršiće se naredbe iz default grane

\* Primer switch naredbe i if naredbe sa istim efektom:

switch (ocena) {

case 5: if (ocena == 5) {

System.out.println("Odlican"); System.out.println("Odlican");

break; }

case 4: else if (ocena == 4) {

System.out.println("Vrlo dobar"); System.out.println("Vrlo dobar");

break; }

case 3: else if (ocena == 3) {

System.out.println("Dobar"); System.out.println("Dobar");

break; }

case 2: else if (ocena == 2) {

System.out.println("Dovoljan"); System.out.println("Dovoljan");

break; }

default: else {

System.out.println("Nedovoljan"); System.out.println("Nedovoljan");

} }

\* Glavni izraz switch naredbe mora biti tipa char, byte, short ili int

-Glavni izraz takođe može biti nabrojivog tipa i tipa String

\* U pravljenju konstantnih izraza moguće je primeniti samo sledeće operatore:

-eksplicitne konverzije u tip String ili u neki prost tip

-unarne operatore +, -, ~ i !

-binarne operatore \*, /, %, +, -, <<, >>, >>>, <, <=, > , >=, ==, !=, &, ^, |, && i ||

-ternarni operator ?:

\* switch naredba je efikasnija od niza ugnježđenih if naredbi

\* Nije obavezno da se niz naredbi uz svaku labelu završi naredbom break

-Efekat je da izvršavanje naredbi “propadne” u sledeći slučaj

-Nenamerno izostavljanje break naredbe je česta greška

\* Niz naredbi uz labelu može da bude prazan

-Omogućava da se jedna ista grupa naredbi izvrši u različitim slučajevima bez dupliranja koda

\* Primer switch naredbe sa pogrešno izostavljenim break naredbama:

class SwitchNoBreak {

public static void main(String[] args) {

System.out.print("Unesite ceo broj: ");

int broj = Svetovid.in.readInt();

switch (broj) {

case 0:

System.out.println("nula");

case 1:

System.out.println("jedan");

case 2:

System.out.println("dva");

case 3:

System.out.println("tri");

default:

System.out.println("manji od 0 ili veci od 3");

}

}

}

***10. Naredbe ponavljanja***

\* Omogućavaju ponavljanje izvršavanja date naredbe ili bloka naredbi

\* Još se nazivaju i “petlje” (eng. *loop*)

\* U Javi postoje tri naredbe ponavljanja:

- while naredba

- do-while naredba

- for naredba

***Naredbe ponavljanja: while***

\* Omogućava ponavljanje izvršavanja date naredbe ili bloka naredbi dokle god je vrednost datog logičkog izraza jednaka true

\* Vrednost logičkog izraza se izračunava pre prvog ponavljanja, što znači da se date naredbe ne moraju izvršiti ni jednom

\* Važno je da naredbe koje se ponavljaju (ili samo izračunavanje vrednosti logičkog izraza) utiču na sledeće izračunavanje logičkog izraza, u protivnom izvršavanje programa može da uđe u “beskonačnu petlju”

\* Oblici:

while (izraz) while (izraz) {

naredba; naredba;

...

naredba;

}

\* Primer:

class HelloWhile {

public static void main(String[] arguments) {

int i = 1;

while (i <= 5) {

System.out.println("Hello, world!");

i = i + 1;

}

}

}

\* Sav “posao” petlje može da se obavi prilikom izračunavanja logičkog izraza, pri čemu se kao telo petlje stavlja prazna naredba (;). Sledeći primer pronalazi sredinu intervala između i i j:

class Sredina {

public static void main(String[] args) {

int i = 100;

int j = 200;

// sredina intervala izmedju i i j, ako je u startu i < j

while (++i < --j);

System.out.println("Sredina je: " + i);

}

}

***Naredbe ponavljanja: do-while***

\* Kao i while naredba, omogućava ponavljanje izvršavanja date naredbe ili bloka naredbi dokle god je vrednost datog logičkog izraza jednaka true

\* Vrednost logičkog izraza se izračunava *posle* prvog ponavljanja, što znači da će se date naredbe izvršiti bar jednom

\* Kao i kod while naredbe, važno je uticati na sledeća izračunavanja logičkog izraza

\* Oblici:

do do {

naredba; naredba;

while (izraz); ...

naredba;

} while (izraz);

\* Primer:

class HelloDoWhile {

public static void main(String[] arguments) {

int i = 1;

do {

System.out.println("Hello, world!");

i++;

} while (i <= 5);

}

}

***Naredbe ponavljanja: for***

\* Naredba for prvenstveno služi da ponavlja naredbu (ili blok naredbi) za određene vrednosti brojačke (kontrolne) promenljive

-Na primer, za vrednosti celobrojne promenljive i redom 1, 2, 3, 4, 5

\* Naredba for dozvoljava i šire primene od gore opisane

\* Oblik:

for (pocetak; izraz; korak) {

naredba;

...

naredba;

}

\* pocetak: inicijalizacija brojačke promenljive ili promenljivih (može i deklaracija sa inicijalizacijom), u slučaju više promenljivih inicijalizacije se odvajaju sa ,

\* izraz: ako ovaj logički izraz ima vrednost false, izlazi se iz petlje (obično provera da li je brojačka promenljiva dostigla neku vrednost)

\* korak: naredba (ili naredbe razdvojene sa ,) koja se izvršava u svakoj iteraciji (obično ažuriranje brojačke promenljive)

\* Primer:

class HelloFor {

public static void main(String[] arguments) {

for (int i = 1; i <= 5; i = i + 1) {

System.out.println("Hello, world! " + i + ". put");

}

}

}

\* Svaki od tri dela (pocetak, izraz, korak) se može izostaviti

\* Primer ekvivalentnih for naredbi:

for (int i = 1; i <= 5; i++) {

System.out.println("Hello, world! " + i + ". put");

}

int i = 1;

for (; i <= 5; i++) {

System.out.println("Hello, world! " + i + ". put");

}

for (int i = 1; i <= 5;) {

System.out.println("Hello, world! " + i + ". put");

i++;

}

\* Delovi pocetak i korak mogu sadržati više naredbi odvojenih zarezima, ne obavezno posvećenih inicijalizaciji, odnosno ažuriranju vrednosti

\* Primer:

class RazniFor {

public static void main(String[] arguments) {

for (int i = 1; i <= 5; i++) {

System.out.println(i);

}

for (int i = 1; i <= 5; System.out.println(i), i++);

int i = 1;

for (System.out.println("Start!"), i = 1;

i <= 5;

System.out.println(i++));

}

}

\* Naredba for može istovremeno da radi sa više brojačkih promenljivih

\* Primer:

class ForViseBrojaca {

public static void main(String[] args) {

int a, b;

for(a = 1, b = 4; a < b; a++, b--) {

System.out.println("a = " + a);

System.out.println("b = " + b);

}

}

}

\* Izlaz:

a = 1

b = 4

a = 2

b = 3

\* Naredba for ima i unapređenu (eng. *enhanced*) varijantu, koja se još naziva i for-each naredba

\* Koristi se kod nizova i kolekcija da bi se na jednostavan način iteriralo (prošlo) kroz sve elemente niza/kolekcije

\* Oblik:

for (elem : nizIliKolekcija) {

naredba;

...

naredba;

}

\* elem: ime ili deklaracija promenljive

\* nizIliKolekcija: ime objekta koji predstavlja niz ili kolekciju čiji su elementi istog tipa kao elem

\* Efekat je da će promenljiva elem redom dobijati vrednosti elemenata iz nizIliKolekcija, i za svaku tu dodelu biće izvršene navedene naredbe

***11. Naredbe sa labelom. Naredbe break, continue i return***

\* Labele (oznake) se koriste da bi se pomoću naredbi break i continue uticalo na tok izvršavanja programa

\* Unutar označene naredbe, naredbama break i continue postiže se skok toka programa na željeno mesto

\* Oblik navođenja labele:

identifikator: naredba;

***Naredba break***

\* Naredba break ima efekat “skoka” toka izvršavanja programa, i ima tri namene:

-Izlazak iz switch naredbe (gde je praktično neizbežna)

-Izlazak iz naredbi ponavljanja (gde se ređe koristi)

-Kao “civilizovana” varijanta goto naredbe (gde se najređe koristi)

\* Naredba break ima dva oblika:

-Bez labele -Sa labelom

\* Naredba break bez labele:

-Može da se nalazi samo unutar naredbi switch, while, do-while i for

-Prebacuje tok izvršavanja programa posle prve (najunutrašnije) naredbe switch, while, do-while ili for koja sadrži break

\* Naredba break sa labelom:

-Ne mora da se nalazi unutar naredbi switch, while, do-while i for

-Naredba koja sadrži break mora biti označena labelom navedenom uz break naredbu

-Prebacuje tok izvršavanja programa posle naredbe označene labelom

-Najčešće služi za izlazak iz više ugneždenih petlji

\* Primer:

class BreakFor {

public static void main(String[] arguments) {

for (int i = 1; i <= 5; i++) {

System.out.println("i = " + i);

if (i == 3) break;

}

}

}

\* Izlaz:

i = 1

i = 2

i = 3

***Naredba continue***

\* Slična naredbi break

\* Može da se koristi samo u okviru naredbi ponavljanja (while, do-while i for)

\* Efekat je da se tekuća iteracija petlje preskače, i tok izvršavanja programa nastavlja sa sledećom iteracijom

\* Kao i naredba break, naredba continue ima dva oblika:

-Bez labele -Sa labelom

\* Naredba continue bez labele:

-Prebacuje tok izvršavanja programa na početak sledeće iteracije prve (najunutrašnije) naredbe while, do-while ili for koja sadrži continue

\* Naredba continue sa labelom:

-Naredba koja sadrži continue mora biti označena labelom navedenom uz continue naredbu

-Prebacuje tok izvršavanja programa na sledeću iteraciju petlje koja je označena tom labelom, i ne mora biti najunutrašnjija (koristi se kod više ugneždenih petlji)

\* Primer:

class Prosti {

public static void main(String[] arguments) {

System.out.print("Prosti brojevi od 1 do 100 su: ");

System.out.print(2);

prvi:

for (int i = 3; i <= 100; i++) {

if (i % 2 == 0) continue;

for (int j = 3; j <= (int)Math.sqrt(i); j++)

if (i % j == 0) continue prvi;

System.out.print(", " + i);

}

System.out.println();

}

}

\* Izlaz:

Prosti brojevi od 1 do 100 su: 2, 3, 5, 7, 11, ..., 89, 97

***Naredba return***

\* Koristi se samo u telu metoda (i konstruktora klasa)

\* Efekat je da se izvršavanje tela metoda prekida, i tok programa nastavlja nakon mesta gde je metod pozvan

\* Naredba return ima dva oblika:

-Bez izraza: return;

\* Povratak iz metoda koji nema povratnu vrednost (vraća tip void)

-Sa izrazom: return izraz;

\* Povratak iz metoda koji ima povratnu vrednost (istog tipa kao izraz)

\* O naredbi return biće još reči kada budemo obrađivali metode

\* Primer:

class PrekidMetoda {

public static void main(String[] args) {

boolean t = true;

System.out.println("pre return-a");

if (t) return;

System.out.println("nece se izvrsiti");

}

}

\* Izlaz:

pre return-a

***12. Referencijalni tipovi, šta znači "referencijalni", klase, polja***

\* Pored prostih tipova, u Javi postoje **referencijalni tipovi**

\* Referencijalni tipovi u Javi su:

-Klase

-Interfejsi

-Nizovi

-Nabrojivi tipovi (uvedeni u Javi 5)

-Lambda izrazi (uvedeni u Javi 8)

\* Sem nizova, svi ovi tipovi su **uvedeni** (ili **korisnički**) tipovi, sto znači da korisnik sam može da definiše konkretne tipove, ili koristi već definisane tipove

***Šta znači “referencijalni”?***

\* Kod prostih tipova, sa vrednošću promenljive u memoriji se radi **direktno**: preko imena promenljive pristupa se vrednosti, vrednost se može menjati naredbom dodele, itd.

\* Kod referencijalnih tipova, vrednosti se u memoriji skladište **indirektno**: preko imena promenljive pristupa se **referenci** (memorijskoj adresi) preko koje se pristupa vrednosti smeštenoj u memoriji na nekom drugom mestu

-Mesto (deo memorije) gde se smeštaju vrednosti kojima se pristupa preko referenci naziva se hrpa (engl. **heap**)

\* Razliku između prostih i referencijalnih tipova ilustrovaćemo preko tipa String, koji je u stvari klasa

\* Posmatrajmo dve promenljive tipa int i String:

int num = 10; // prost tip

String name = "Hello"; // referencijalni tip

\* Jednostavno rečeno, kod prostih tipova ime promenljive je zamena za **adresu**, a kod referencijalnih tipova ime promenljive je zamena za **adresu adrese**

\* Promenljive prostog tipa predstavljaju jedan vid **apstrakcije**: promenljive svojim imenom apstrahuju memorijske adrese (skrivaju detalje o vrednostima adresa, organizaciji memorije, itd.)

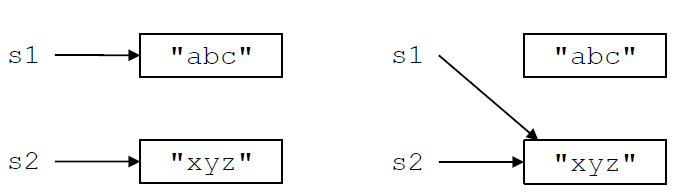
\* Promenljive referencijalnih tipova ovu apstrakciju podižu na viši (meta) nivo

-Omogućava pravljenje dugačkih “lanaca” referenci, čime se mogu praviti složene strukture podataka

String s1 = "abc"; String s1 = "abc";

String s2 = "xyz"; String s2 = "xyz";

s1=s2;



***Klase (1. deo)***

\* Klase su osnovni referencijalni tip u Javi

\* Svi ostali referencijalni tipovi su “u pozadini” realizovani kao klase, mada se to na prvi pogled ne vidi

\* Vrednosti promenljivih tipa neke klase su reference na **objekte** ili **instance**, i moraju se kreirati korišćenjem operatora new

-Izuzetak je klasa String, čije se instance mogu kreirati i navođenjem literala: String name = "Hello";

\* Mnogi programski jezici dizajnirani su po principu da je neki koncept osnovni, tzv. “građanin 1. reda”, oko kog se “sve vrti”

-U slučaju Jave, taj koncept je **klasa**

\* Klase se deklarišu pomoću ključne reči class, nakon čega sledi ime klase, i u vitičastim zagradama navedeni članovi klase

\* Od članova klase, na početku ćemo obraditi:

-Polja

-Statičke metode

\* Do sada nam je svaki program u stvari bio klasa sa jednim statičkim metodom main, koji ima poseban status jer se on izvršava pri pokretanju programa u JVM

\* Mi u klasi možemo deklarisati i druge statičke metode i pozivati ih iz metoda main

\* Pri tom ćemo podatke (prostih i referencijalnih tipova podataka) prosleđivati statičkim metodama kao argumente, čime ćemo dobiti programe pisane **proceduralnim stilom programiranja**

***Polja klase***

\* Polja klase predstavljaju podatke, i deklarišu se na isti način kao lokalne promenljive, samo na drugom mestu:

-Lokalne promenljive se deklarišu u okviru metoda i blokova

-Polja klase se deklarišu u okviru klase

\* Polja klase “vezuju” se za objekte, odnosno instance klase, i postoje nezavisno u okviru svakog objekta (tako da u stvari postaju *polja objekta*)

\* Poljima objekta pristupa se pomoću operatora **.**

\* Proširićemo definiciju pojma **promenljiva**, koja će sad da uključuje:

-Lokalne promenljive

-Polja

\* Klase predstavljaju jedan od načina da se, pomoću polja, podaci grupišu i “spakuju” u logične celine (objekte)

***Polja klase: primer***

class Tacka {

double x, y;

}

class TackaTest {

public static void main(String[] args) {

Tacka t1 = new Tacka();

Tacka t2 = new Tacka();

t1.x = 1.0;

t1.y = 2.0;

t2.x = 5.0;

System.out.println("t1 = (" + t1.x + ", " + t1.y + ")");

System.out.println("t2 = (" + t2.x + ", " + t2.y + ")");

}

}

\* Izlaz:

t1 = (1.0, 2.0)

t2 = (5.0, 0.0)

***Polja klase: anti-primer***

\* U ovom “anti-primeru” ne grupišemo promenljive u klase/objekte kao polja, već uvodimo posebne nezavisne (lokalne) promenljive za svaki podatak. Ovakav pristup postaje nezgodan čim se program malo poveća

class TackaBad {

public static void main(String[] args) {

double t1x = 1.0;

double t1y = 2.0;

double t2x = 5.0;

double t2y = 0.0;

System.out.println("t1 = (" + t1x + ", " + t1y + ")");

System.out.println("t2 = (" + t2x + ", " + t2y + ")");

}

}

\* Izlaz:

t1 = (1.0, 2.0)

t2 = (5.0, 0.0)

***13. Klase: kreiranje objekata, literal null, brisanje objekata, lanci polja, nasleđivanje, statički članovi klase***

***Kreiranje objekata operatorom new***

\* U prethodnom primeru smo videli kako se kreira nova instanca klase:

Tacka t1 = new Tacka();

\* Pri izračunavanju izraza new Tacka() dešava se sledeće:

-Rezerviše se (alocira) memorijski prostor na heap-u potreban da se smesti objekat sa svojim članovima

-Polja objekta se inicijalizuju na podrazumevane vrednosti:

\* 0, 0L, 0.0, '\0', false, **null**...

\* Ili na vrednosti eksplicitno navedene kod polja korišćenjem operatora =

-Izvršava se konstruktor (kod u koji se mogu staviti neke posebne inicijalizacije i slično, više o njima kasnije)

-Referenca na napravljeni objekat se vraća kao vrednost izraza

***Literal null***

\* Promenljive referencijalnog tipa mogu se inicijalizovati i literalom null, koji je podrazumevana vrednost za referencijalne tipove (kao 0 za int, itd.):

Tacka t1 = null;

\* Literal null označava “praznu” referencu, odnosno ref. koja ne ref. “nigde”

\* Drugim rečima, dodeljivanjem vrednosti null promenljivoj referencijalnog tipa kažemo da ne postoji objekat u memoriji na koji ta promenljiva referencira

\* **Opasnost:** ako objekat nije inicijalizovan, odnosno ako je null, pokušaj pristupa nekom članu izazvaće grešku u toku izvršavanja programa

***Brisanje objekata***

\* U Javi ne postoji poseban operator kojim se objekti brišu iz memorije, tj. ne postoji pandan operatoru new koji radi suprotnu operaciju

\* U Javi se brisanje objekata radi automatski, po potrebi, o čemu u principu programer ne bi trebao/la da brine

\* Da bi objekat mogao u jednom trenutku da bude obrisan, neophodno je da na njega ne pokazuje ni jedna referenca

-Ovo se postiže dodelom promenljivoj reference na neki drugi objekat, ili null

-Ne postoji način da se u programu “povrati” pristup ovakvim objektima

\* Za brisanje ovakvih objekata zadužen je poseban proces, tzv. sakupljač đubreta (engl. *garbage collector*)

\* Ovaj proces pokreće se automatski, a moguće je eksplicitno zatražiti njegovo pokretanje pozivom metoda System.gc();

***Polja klase: lanci polja***

\* Za početak ćemo posmatrati klase i objekte samo kao strukture podataka koje sadrže polja

-Analogno tipovima podataka struct, record i sl. u nekim drugim programskim jezicima

\* Kao i sve promenljive, polja klasa mogu biti bilo kog tipa podataka, pa i referencijalnog, odnosno mogu biti tipa neke klase

\* Time se može postići “ulančavanje” polja objekata, jer polje objekta može biti objekat čije neko polje takođe može biti objekat, itd.

\* Dakle, možemo imati lanac pristupa poljima preko operatora **.**

***Polja klase: lanci polja - primer***

class Automobil {

String marka, proizvodjac;

int godinaProiz;

String boja;

int brKonja, brVrata = 5;

String regBroj;

}

class Vlasnik {

String ime, prezime, JMBG;

Automobil auto;

}

class Automobili {

public static void main(String[] args) {

Vlasnik pera = new Vlasnik();

pera.ime = "Pera";

pera.prezime = "Peric";

pera.JMBG = "0101900800001";

pera.auto = new Automobil();

pera.auto.marka = "Yugo Koral 55";

pera.auto.proizvodjac = "Crvena zastava";

pera.auto.godinaProiz = 1989;

pera.auto.boja = "crvena";

pera.auto.brKonja = 55;

pera.auto.brVrata = 3;

}

}

***Nasleđivanje klasa***

\* Jedan od fundamentalnih koncepata objektno-orijentisanog programiranja je **nasleđivanje** klasa

\* Kada se deklariše da klasa B nasleđuje klasu A, to znači da B preuzima članove od A, eventualno dodajući nove

\* Na ovom kursu se konceptom nasleđivanja nećemo detaljno baviti, ali moramo biti svesni sledećih posledica:

-Promenljivoj tipa A (“nadklase”) mogu se dodeljivati objekti tipa B (“podklase”)

-Sve klase (i ostali ref. tipovi) implicitno nasleđuju specijalnu klasu Object

***Statički i nestatički članovi***

\* Statički članovi klase - metodi i polja koja su deklarisana pomoću ključne reči static

\* Mogu se koristiti i bez prethodnog kreiranja instanci klase, oni se zapravo i ne odnose na instance klase, već na samu klasu

\* Svaki objekat klase sadrži svoju kopiju svih polja i metoda klase, osim onih polja i metoda koji su statički, jer statička polja i metodi pripadaju klasi a ne njenim instancama

***Statička polja***

\* Do sada smo posmatrli samo nestatička polja, gde svaki objekat ima svoju kopiju polja

\* Slično kao kod metoda, i polja mogu da se deklarišu sa ključnom rečju static, čime postaju statička, odnosno jedinstvena za ceo program, tj. klasu

-Jednom rečju, polja postaju **globalna**

\* Imenima statičkih polja se pristupa analogno kao imenima statičkih metoda

\* Treba imati na umu da je statičko polje uvek jedno isto, uprkos tome što mu se može pristupati preko različitih objekata

***Statička polja: primer***

class Tacka {

double x, y;

static final int NULA = 0;

}

class StaticTest {

static final int JEDAN = 1;

public static void main(String[] args) {

Tacka t = new Tacka();

System.out.println("Nule: " + Tacka.NULA + ", " + t.NULA);

System.out.println("Jedan: " + JEDAN);

}

}

***14. Nizovi: deklaracija i inicijalizacija, pristup elementima, polje length, odnos sa klasom Object***

***Nizovi***

\* Nizovi predstavljaju grupu elemenata istog tipa koje se pojavljuju pod istim imenom

\* Niz je specijalna vrsta objekta i sastoji se od elemenata kojima se pristupa pomoću njihovih indeksa

\* Ubuduće, kada kažemo “niz” mislićemo na objekat nizovnog tipa podataka ili sam nizovni tip (biće jasno iz konteksta)

\* Indeks prvog elementa u nizu je uvek 0, dok je indeks poslednjeg elementa za 1 manji od ukupnog broja elemenata u nizu

\* Tip elemenata niza može biti bilo koji tip, uključujući i tip niza

\* Ako su elementi niza nizovi, tada takav niz zovemo više-dimenzionalan niz, inače se radi o jednodimenzionalnom nizu

***Nizovi: deklaracija***

\* Promenljive tipa jednodimenzionalnog niza deklarišemo navođenjem tipa, imena promenljive i jednog para uglastih zagrada i može se inicijalizovati odmah prilikom deklaracije

-Par uglastih zagrada može se navesti posle imena tipa ili posle imena promenljive

\* Prilikom deklaracije niza ne navodi se veličina niza - ona se zadaje tek prilikom njegovog kreiranja new operatorom

\* Nakon kreiranja, inicijalne vrednosti elemenata niza su nule (brojevni nizovi) , false vrednosti (logički nizovi) ili null ako su elementi niza referencijalnog tipa

***Nizovi: deklaracija - primeri***

int[] nizCelih1;

int nizCelih2[];

String[] imena1;

String imena2[];

Lopta[] lopte;

Object objekti[];

...

nizCelih1 = new int[10];

imena1 = new String[] {"aca", "ceca", "daca"};

objekti = new Object[8];

\* U primerima se mogu uočiti dva vida inicijalizacije nizova, koji se međusobno isključuju:

-Navođenjem broja elemenata

-Navođenjem samih elemenata

\* Kod načina sa navođenjem elemenata dozvoljeno je izostaviti deo new TipElementa[], kad se inicijalizacija radi pri deklaraciji, u protivnom se taj deo mora navesti

-Dakle, kod sledećeg koda kompajler prijavljuje grešku:

int[] celi3;

celi3 = {1, 2, 3};

***Nizovi: pristup elementima***

\* Elementima niza se pristupa tako što se prvo navede ime niza ili izraz čija je vrednost niz (pri čemu to ne sme biti izraz kreiranja niza), nakon čega se u uglastim zagradama navodi celobrojni izraz čija vrednost je indeks elementa kojem pristupamo

\* Svaki niz ima i polje length koje sadrži broj elemenata niza zadat pri inicijalizaciji

***Nizovi: pristup elementima - primer***

class FibonaciNiz {

public static void main(String[] args) {

int[] fib = new int[4];

fib[0] = 0;

fib[1] = 1;

fib[2] = fib[0] + fib[1];

fib[3] = fib[1] + fib[2];

System.out.println("3. Fibonacijev broj je " + fib[3]);

}

}

\* Izlaz:

3. Fibonacijev broj je 2

***Polje length***

\* Nakon kreiranja niza, broj njegovih elemenata je fiksan i više se ne može promeniti

-Ako se niz ponovo inicijalizuje operatorom new, u stvari se pravi novi niz, ne proširuje se stari

\* Broj elemenata niza se može dobiti pomoću length polja niza. Ovo polje je konstantno (final), tj. njegova vrednost se ne može modifikovati

\* Polje length može da se koristi da bismo saznali koliko argumenata je korisnik naveo prilikom poziva programa

\* Navedenim argumentima pristupamo pomoću jedinog parametra metoda main, tipa String[]

***Nizovi i klasa Object***

\* Rekli smo da su svi referencijalni tipovi “u pozadini” realizovani kao klase, što je slučaj i sa nizovima

\* Svaki niz direktno nasleđuje klasu Object. To znači da svaki niz sadrži sve članove klase Object, a takođe i da referenca bilo kog niza može biti dodeljena promenljivoj tipa Object

\* Primer:

class ObjectNiz {

public static void main(String[] args) {

int[] x = {2, 4, 6};

Object obj = x;

int[] y = (int[])obj;

x[0] = 1;

for (int i = 0; i < y.length; i++)

System.out.println(y[i]);

}

}

\* Izlaz:

1 4 6

***15. Višedimenzionalni nizovi***

\* Višedimenzionalni nizovi su nizovi čiji su elementi takođe nizovi

\* Broj dimenzija niza se određuje na sledeći način:

-Niz čiji elementi nisu nizovi ima jednu dimenziju

-Niz čiji su elementi nizovi ima za jedan veću dimenziju od dimenzije njegovog elementa

\* Promenljiva čiji je tip višedimenzionalni niz se deklariše navođenjem imena tipa koji nije nizovski tip i onoliko parova otvorenih i zatvorenih uglastih zagrada koliki je broj dimenzija niza, nakon čega se navodi ime promenljive

\* Pri inicijalizaciji višedimenzionalnih nizova ne moraju biti inicijalizovane sve dimenzije: mora se inicijalizovati samo prva dimenzija, a proizvoljan broj ostalih dimenzija može ostati neinicijalizovan

-Prvo se navode uzastopne dimenzije koje se inicijalizuju brojevima elemenata (mora postojati bar jedna)

-Zatim se navode neinicijalizovane dimenzije praznim parovima uglastih zagrada

*Primeri: Deklaracije i inicijalizacije*

int[][] matrica1;

int[] matrica2[];

int matrica3[][];

boolean[][] logTabela1;

logTabela1 = new boolean[][] {{true, false}, {false, true}};

boolean[][] logTabela2 = {{true, false}, {false, true}};

*Primeri: Kreiranje višedimenzionalnih nizova na drugi način*

int[][] matrica1;

matrica1 = new int[3][4];

int[] matrica2[] = new int[2][5]; // i deklaracija i kreiranje niza

System.out.println(matrica2.length); // 2

System.out.println(matrica2[0].length); // 5

System.out.println(matrica2[1].length); // 5

int matrica3[][] = new int[3][];

System.out.println(matrica3.length); // 3

System.out.println(matrica3[0]); // null

matrica3[0] = new int[4]; // podnizovi mogu biti razlicitih duzina

matrica3[1] = new int[6];

matrica3[2] = new int[5];

System.out.println(matrica3[0].length); // 4

System.out.println(matrica3[1].length); // 6

System.out.println(matrica3[2].length); // 5

***Višedimenzionalni nizovi: primer***

class ZbirMatrica {

public static void main(String[] args) {

double[][] A = { {1.1, 2.2, 3.3, 4.1},

{0.4, -2.1, 1.9, 8.7},

{4.1, 2, 44, 23.2} };

double[][] B = { {7.3, 12, 33.2, 6.2},

{0.0, 3.1, 2.7, 9.3},

{13.1, 3.8, 4.4, 23.8} };

double[][] rez = new double[3][4];

for (int i = 0 ; i < 3; i++)

for (int j = 0; j < 4; j++)

rez[i][j] = A[i][j] + B[i][j];

System.out.println("Zbir matrica je:");

for (int i = 0 ; i < 3; i++) {

for (int j = 0; j < 4; j++)

System.out.print(rez[i][j] + "\t");

System.out.println();

}

}

}

***16. Operatori nad referencijalnim tipovima***

**Operatori nad referencijalnim tipovima**

**=** dodela

**(*imeTipa*)** eksplicitna konverzija tipa

**==** ispitivanje jednakosti

**!=** ispitivanje nejednakosti

**?:** uslovni operator

**.** pristup članu

**instanceof** ispitivanje tipa objekta

**+** konkatenacija stringova

**new** kreiranje instance

**[]** pristup elementu niza

***Dodela***

**\*** Operator dodele = je binarni operator kojim se promenljivoj sa leve strane operatora dodeljuje vrednost izraza sa desne strane operatora

**\*** Vrednost promenljive referencijalnog tipa nije sam objekat, već **referenca objekta** - zbog toga se dodelom vrednosti jedne promenljive drugoj ne pravi nova kopija objekta, već se kopira samo referenca na objekat, tako da posle dodele obe promenljive pokazuju na isti objekat

**\*** Vrednost koja se dodeljuje promenljivoj referencijalnog tipa može biti:

-Literal null ili

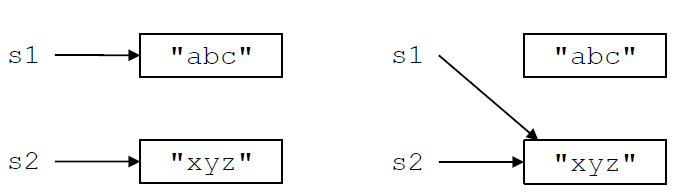
-Referenca objekta čiji je tip jednak tipu promenljive ili

-Referenca objekta čiji tip je moguće konvertovati u tip promenljive korišćenjem neke od proširujućih referencijalnih konverzija

String s1 = "abc"; String s1 = "abc";

String s2 = "xyz"; String s2 = "xyz";

s1=s2;



***Dodela: proširujuće konverzije***

**\*** Neke od proširujućih referencijalnih konverzija su:

1.Konverzija iz bilo koje klase P u bilo koju klasu N, pod uslovom da je P podklasa klase N

2.Konverzija vrednosti null u bilo koji referencijalni tip,

3.Konverzija iz bilo kog niza u klasu Object,

4.Konverzija iz bilo kog niza A[] u bilo koji niz B[], pod uslovom da su A i B referencijalni tipovi i da postoji proširujuća referencijalna konverzija iz tipa A u tip B

**\*** Proširujuće referencijalne konverzije biće detaljnije obrađene na kursu OOP1

**\*** Primeri:

Object objekat = "abc"; // 1. String -> Object

String[] boje = {"crvena", "zelena", "plava"};

Object[] nizObj;

nizObj = boje; // 4. String[] -> Object[]

objekat = nizObj; // 3. Object[] -> Object

boje = null; // 2. null -> String[]

***Eksplicitna konverzija (kasting)***

**\*** Vrednost nekog tipa se može eksplicitnom konverzijom pretvoriti u odgovarajuću vrednost drugog tipa

**\*** Ime ciljnog tipa se navodi u običnim zagradama i koristi se kao unarni operator

**\*** Eksplicitna konverzija tipa se najčešće koristi prilikom dodele vrednosti promenljivoj kada se tip promenljive i tip vrednosti razlikuju i kada se oni ne mogu izjednačiti implicitnom proširujućom referencijalnom konverzijom

**\*** Pravila koja definišu kad je moguća eksplicitna konverzija referencijalnih tipova su dosta složena i radiće se na kursu OOP1

**\*** Za nas će biti dovoljno da se podsetimo primera:

int[] x = {2, 4, 6};

Object obj = x;

int[] y = (int[])obj;

**\*** Dakle, ako je na neki objekat primenjena proširujuća konverzija (npr. int[] -> Object), dodela u “suprotnom smeru” mora da se radi pomoću kastinga

***Ispitivanje (ne)jednakosti***

**\*** Operator ispitivanja jednakosti == i operator ispitivanja nejednakosti != se pored primene na vrednosti prostih tipova mogu primeniti i na vrednosti referencijalnih tipova

-Njima se ispituje da li dve reference pokazuju ili ne pokazuju na isti objekat

**\*** Rezultat primene operatora != je uvek suprotan od rezultata primene operatora ==

**\*** Ovim operatorima nikada ne treba proveravati da li su dva objekta jednaka (po sadržaju), već za to treba koristiti specijalno napravljene metode

-(standardan način je da se za tu svrhu redefiniše metod equals koji potiče iz klase Object)

***Uslovni operator***

**\*** Ternarni operator ?: može biti primenjivan i na referencijalne vrednosti

**\*** Prvi operand ovog operatora je uvek logičkog tipa, a druga dva operanda mogu biti oba prostog tipa, ali mogu biti i oba referencijalnog tipa

**\*** Ako su druga dva operanda ovog operatora istog referencijalnog tipa, tada će i rezultat operatora biti tog tipa. Kada su druga dva operanda različitog referencijalnog tipa, na primer jedan operand je tipa A a drugi operand je tipa B, tada za te tipove mora da važi sledeće:

-Vrednost tipa A je moguće dodeliti promenljivoj tipa B, ili

-Vrednost tipa B je moguće dodeliti promenljivoj tipa A

**\*** Uslovni operator ?: se sa referencijalnim vrednostima koristi slično kao i sa prostim

**\* Primer:**

Object o = 1\*2\*3 != 1+2+3 ? new Object() : "abc";

***Pristup članu referencijalnog tipa***

**\*** Operatorom **.** (tačka) se pristupa poljima, metodima i drugim članovima referencijalnih tipova

**\*** Moguće je pristupiti samo onim članovima koji su u datom kontekstu vidljivi (više o vidljivosti kasnije)

**\*** Nestatičkim članovima klase se pristupa navođenjem imena objekta, tačke i imena člana

**\*** Statičkim članovima klase se pristupa navođenjem imena klase, tačke i imena člana, a moguće im je pristupiti i na isti način kao nestatičkim članovima, navođenjem imena nekog objekta te klase, tačke i imena statičkog člana

***Operator instanceof***

**\*** Binarni operator instanceof se koristi samo kod referencijalnih tipova, njime se ispituje da li je tip prvog operanda jednak drugom operandu

**\*** Prvi operand može biti samo neki objekat ili null

**\*** Drugi operand je ime nekog referencijalnog tipa

**\*** Ako je tip prvog operanda moguće eksplicitnom konverzijom konvertovati u tip naveden u drugom operandu, tada je vrednost izraza true, a inače je false

**\*** Primer:

int[] x = {2, 4, 6};

Object obj = x;

if (obj instanceof int[])

System.out.println("Moguca konverzija");

int[] y = (int[])obj;

***Konkatenacija stringova***

**\*** Konkatenacija (spajanje) stringova se vrši binarnim operatorom **+**

**\*** Ako su oba operanda stringovi, onda je rezultat novi string koji je jednak stringu koji bi nastao spajanjem stringova operanada

**\*** Ako je samo jedan operand tipa String a drugi je nekog drugog tipa, onda se najpre vrednost operanda nestringovskog tipa konvertuje u tip String nakon čega se rezultat kreira isto kao u slučaju kada su oba operanda tipa String

**\*** Konvertovanje u tip String je uvek moguće izvršiti:

-Prosti tipovi – uobičajena konverzija

-Literal null – u string "null"

-Bilo koji objekat – u string nastao pozivom metoda toString() koji je deklarisan u klasi Object, pa ga sve klase nasleđuju

**\* Primeri:**

System.out.println("abc" + 1 + 2); // dve konkatenacije,

// ispisuje abc12

System.out.println(1 + "abc" + 2); // dve konkatenacije,

// ispisuje 1abc2

System.out.println(1 + 2 + "abc"); // sabiranje i konkatenacija,

// ispisuje 3abc

***17. Klase: metodi***

\* Metodi nam daju način da kod koji bismo često ponavljali u programu “spakujemo” u jednu celinu i svedemo na jednu naredbu - poziv metoda

\* Metode deklarišemo kao članove klasa - u istom odeljku gde deklarišemo i polja

\* Ako se deklariše sa ključnom rečju static, metod je statički - jedinstven za ceo program, tj. klasu kojoj pripada

\* U protivnom svaki objekat tipa klase sadrži svoju kopiju metoda

\* Prvo ćemo se fokusirati na statičke metode

**Deklaracija metoda Primer:** Recimo da u programu često imamo potrebu da štampamo sve cele brojeve iz određenog intervala. Da ne bismo ponavljali vrlo sličan kod (for petlju) svaki put, možemo deklarisati metod stampajInterval

static void stampajInterval(int a, int b) {

for (int i = a; i <= b; i++) {

System.out.print(" " + i);

}

}

**Primer:** Recimo da u programu često imamo potrebu da proveravamo da li je znak jednak nekom od znakova iz skupa operacija koji nas zanima. Umesto da ponavljamo isti složen logički izraz, možemo deklaristi metod:

static boolean jeOperacija(char c) {

return c == '+' || c == '-' || c == '\*' || c == '/';

}

\* Deklaracija metoda sadrži:

-Zaglavlje metoda

-Telo metoda

\* Zaglavlje metoda počinje povratnim tipom, tj. tipom vrednosti koju metod vraća prilikom poziva, koji može biti:

-Bilo koji tip, ili

-Ključna reč void, čime označavamo da metod nema povratnu vrednost

\* Zatim sledi ime metoda (identifikator), i u običnim zagradama lista formalnih parametara (argumenata):

-0 ili više deklaracija formalnih parametara, odvojene zarezima

\* Formalni parametar se deklariše kao i promenljiva: navođenjem tipa i imena

\* Telo metoda je blok, odnosno niz naredbi i deklaracija lokalnih promenljivih između { }

\* Formalni parametri metoda su vidljivi u telu metoda (i samo u telu metoda) i mogu se koristiti kao i sve druge promenljive

\* Štaviše, proširićemo opet definiciju promenljive, tako da sad promenljiva može biti:

-Lokalna promenljiva

-Polje objekta/klase

-Formalni parametar metoda

***Poziv metoda***

\* Ako metod ima povratnu vrednost, poziv metoda može da se pojavi:

-U izrazu (na mestu gde je dozvoljena vrednost povratnog tipa metoda)

-Kao naredba (tada se povratna vrednost zanemaruje)

\* Ako je metod deklarisan sa void, tada poziv metoda može da se pojavi samo kao naredba

\* Poziv metoda počinje navođenjem imena metoda, nakon čega se u običnim zagradama navode **stvarni parametri** metoda

\* Stvarni parametri su vrednosti (izrazi) koji se pri pozivu metoda vezuju za formalne parametre, nakon čega se izvršava telo metoda, ali tako da se formalni parametri sada odnose na prosleđene stvarne parametre

class MetodTest {

static void stampajInterval(int a, int b) {

for (int i = a; i <= b; i++) {

System.out.print(" " + i);

}

}

static boolean jeOperacija(char c) {

return c == '+' || c == '-' || c == '\*' || c == '/';

}

public static void main(String[] args) {

stampajInterval(5, 10);

System.out.println();

System.out.print("Unesite znak operacije (+, -, \*, /): ");

char c = Svetovid.in.readChar();

if (jeOperacija(c)) {

System.out.println("Uneli ste znak operacije" + c +", hvala!");

}

else {

System.out.println("Niste uneli znak operacije. C c c...");

}

}

}

***Naredba return***

\* Naredba return izaziva vraćanje toka izvršavanja na mesto gde je metod pozvan

\* Ako je metod deklarisan da ima povratnu vrednost, tada:

- Vrednost odgovarajućeg tipa se navodi nakon ključne reči return

-Izvršavanje naredbe return u telu metoda je **obavezno**, i to u svakoj mogućoj grani izvršavanja

\* Ako je metod deklarisan sa void, tada:

-Nakon ključne reči return se ne navodi ništa

-Izvršavanje naredbe return u telu metoda **nije obavezno**, već se izvršavanje metoda može završiti i izvršavanjem poslednje naredbe u telu metoda

***Poziv metoda: prosleđivanje parametara***

\* U Javi se prosleđivanje parametara pri pozivu metoda radi **po vrednosti** (engl. *call by value*)

\* To znači da će u formalni parametar biti **kopirana vrednost** stvarnog parametra

\* Posledica za proste tipove je da menjanjem vrednosti formalnog parametra u telu metoda ne utičemo na vrednost stvarnog parametra

-Npr. ako metodu prosledimo promenljivu x kao stvaran parametar, a odgovarajući formalni parametar menjamo u telu metoda, nakon izvršavanja metoda promenljiva x će ostati neizmenjena

\* **Napomena:** formalni i stvarni parametar mogu se zvati isto, ali to su *uvek* dve posebne stvari

***Poziv metoda: prosleđivanje parametara***

**Primer:**

static void stampajInc(double x) {

x++; // menja vrednost formalnog parametra,

// ali ne i stvarnog

System.out.println("metod: x = " + x);

}

\* Telo metoda main:

double x = 5.0;

stampajInc(x);

System.out.println("main: x = " + x);

\* Izlaz:

metod: x = 6.0

main: x = 5.0

***Poziv metoda: prosleđivanje parametara***

\* Kopiranje vrednosti stvarnog u formalni parametar za referencijalne tipove znači da će biti kopirana **referenca**

\* Posledica je da promena vrednosti formalnog parametra referencijalnog tipa (tj. promena same reference) naredbom dodele i slično neće imati efekta na stvarni parametar (kao i kod prostih tipova)

\* Međutim, promena **sadržaja objekta** (polja, elemenata nizova...) preko kopirane reference će biti oslikana i na stvarnom parametru

**Poziv statičkog metoda**

\* Prilikom poziva statičkog metoda, ime metoda može se navesti u nekoliko oblika:

-Ime klase, tačka, identifikator metoda

Primer: ParametarTest.menjajNesto(t1);

-Ime objekta, tačka, identifikator metoda

Primer: ParametarTest pt = new ParametarTest();

pt.menjajNesto(t1);

-Samo identifikator metoda (ako je poziv u okviru iste klase)

Primer: menjajNesto(t1);

**Nestatički metodi**

\* Nestatički metodi deklarišu se bez ključne reči static

\* Svaki objekat ima svoju kopiju metoda (koji se mehanizmom nasleđivanja može menjati)

\* Koncept nestatičkih metoda je jedan od najosnovnijih u objektno-orijentisanom programiranju, jer omogućava da se u jednu celinu (objekat) “spakuju” ne samo podaci (polja), već i programski kod (nestatički metodi)

\* Po pravilu, nestatički metod treba da se odnosi na i operiše nad objektom preko kog se poziva

***Nestatički metodi: primer***

class Tacka {

double x, y;

void translirajX(double pomeraj) {

x += pomeraj;

}

void translirajY(double pomeraj) {

y += pomeraj;

}

}

class TackaTestOO {

public static void main(String[] args) {

Tacka t1 = new Tacka();

Tacka t2 = new Tacka();

t1.x = 1.0; t1.y = 2.0; t2.x = 5.0;

t1.translirajY(10.0);

t2.translirajX(5.0);

System.out.println("t1 = (" + t1.x + ", " + t1.y + ")");

System.out.println("t2 = (" + t2.x + ", " + t2.y + ")");

}

}

***Poziv nestatičkog metoda***

\* Prilikom poziva nestatičkog metoda, ime metoda može se navesti u nekoliko oblika:

-Ime objekta, tačka, identifikator metoda

**Primer:** t1.translirajY(10.0);

-Samo identifikator metoda, ako je poziv u okviru nestatičkog metoda iz iste klase. Tada se pozivaju metodi vezani za isti objekat:

**Primer:** Nov metod u klasi Tacka:

void translirajXY(double pomeraj) {

translirajX(pomeraj);

translirajY(pomeraj);

}

***18. Konstruktori, modifikatori pristupa, paketi***

\* Konstruktori se mogu posmatrati kao specijalna vrsta nestatičkih metoda koji se u životu objekta mogu izvršiti samo jednom, i to pri kreiranju objekta pomoću operatora new (vidi prethodno predavanje)

\* Deklarišu se navođenjem imena klase, liste formalnih argumenata u običnim zagradama, i tela u vidu bloka

\* Primer:

Tacka(double xInit, double yInit) {

x = xInit; y = yInit;

}

Tada se objekat tipa Tacka kreira na sledeći način:

Tacka t1 = new Tacka(1.0, 2.0);

\* U klasi možemo deklarisati proizvoljan broj konstruktora, koji moraju imati različite liste argumenata (broj i/ili tipove)

\* Ako ne deklarišemo ni jedan konstruktor, u klasi će implicitno biti deklarisan tzv. podrazumevani (engl. *default*) konstruktor, bez argumenata i sa praznim telom

\* Ako deklarišemo bar jedan konstruktor, podrazumevani konstruktor se gubi

\* U našem primeru, ako i dalje želimo mogućnost da kreiramo objekte klase Tacka bez navođenja argumenata (kao ranije), moramo konstruktor bez argumenata eksplicitno deklarisati:

Tacka() { }

***Modifikatori pristupa***

\* Modifikatori pristupa (engl. *access modifiers*) su ključne reči koje se dodeljuju članovima klase i određuju njihovu vidljivost, tj. mogućnost da se tim članovima pristupi iz različitih delova koda

\* U Javi postoje 4 modifikatora pristupa članovima: public, private, protected, i odsustvo modifikatora (*default*)

\* Modifikator public označava da se članu može pristupiti iz bilo kog dela programa

\* Modifikator private označava da se članu može pristupiti samo u kodu klase kojoj pripada

\* Preostala dva modifikatora su “negde između” ova dva ekstrema

\* Modifikatori pristupa će se detaljnije raditi na predmetu OOP1

***Paketi***

\* Paketi su jednostavan ali efektan način organizovanja referencijalnih tipova u Javi

\* Svaka Java klasa (kao i drugi uvedeni referencijalni tipovi) pripada nekom paketu

\* Svaki paket predstavljen je folderom istog imena kao paket, u koji se smeštaju .java fajlovi sa jedinicama prevođenja, i podfolderi koji predstavljaju pod-pakete

\* Da bi se u jedinici prevođenja naznačilo kom paketu pripada, na početku .java fajla se stavlja ključna reč package i ime paketa

-Ako se ovo izostavi (kao što mi radimo) klasa pripada tzv. podrazumevanom (*default*) paketu koji odgovara tekućem folderu

\* Da bi se u kodu neke klase (ref. tipa) koristili ref. tipovi iz drugih paketa, potrebno je uraditi nešto od sledećeg:

-Koristiti puno kvalifikovano ime ref. tipa, navođenjem imena paketa, svih podpaketa, i na kraju imena tipa, razdvojenih tačkom, svuda u jedinici prevođenja (.java fajlu) gde se ref. tip koristi **Primer:** java.io.File

-Na početku jedinice prevođenja pomoću ključne reči import uvesti ime referencijalnog tipa, nakon čega se pri navođenju imena tipa ne moraju koristiti imena paketa

**Primer:** Na početku .java fajla: import java.io.File;

U ostatku .java fajla navodimo samo: File

-Ako želimo da koristimo više ref. tipova iz istog paketa, možemo sva imena iz paketa uvesti korišćenjem znaka \*

**Primer:** Na početku .java fajla: import java.io.\*;

U ostatku .java fajla postaju dostupna sva imena iz paketa java.io: File, Reader, Writer...

***19. Važnije API klase u Javi***

\* Uz Javu, odnosno Java Development Kit (JDK) dolazi i velika biblioteka gotovih klasa

- Application Programming Interface (API)

\* Opisaćemo mali broj važnijih klasa:

- Object

- String

- Referencijalni ekvivalenti prostih tipova

\* Sve ove klase pripadaju paketu java.lang koji ima specijalan status: ne mora se navoditi u kvalifikovanom imenu, niti se njegov sadržaj mora uvoziti

***Klasa Object***

\* Osnovna klasa u hijerarhiji svih Java klasa, odnosno referencijalnih tipova

\* Svi referencijalni tipovi implicitno nasleđuju klasu Object, što znači da nasleđuju njene članove

\* Pomenućemo dva bitna člana - metoda:

public boolean equals(Object obj): poredi tekući objekat sa obj.

\* Osnovna implementacija poredi reference operatorom ==, ali je namena metoda da se redefiniše u drugim klasama i radi poređenje *sadržaja* objekata

public String toString(): vraća tekstualnu reprezentaciju tekućeg objekta.

\* Osnovna implementacija vraća interni kod objekta, a očekuje se da svaka klasa za koju postoji potreba za String reprezentacijom redefiniše ovaj metod. Metod toString ima specijalan status - implicitno se poziva pri konverzijama objekata u string (pri štampanju, konkatenaciji stringova i sl.)

***Klasa Object: primer metoda toString()***

class Tacka {

double x, y;

...

public String toString() {

return "(" + x + ", " + y + ")";

}

}

...

System.out.println("t1 = " + t1.toString());

// .toString() se moze izostaviti

System.out.println("t2 = " + t2);

...

***Klasa String***

\* Osnovna klasa za predstavljanje teksta u Javi

\* Objekti tipa String su **nepromenljivi** (engl. *immutable*): jednom kreiran, sadržaj stringa ne može se menjati, već se uvek kreira novi objekat

\* Klasa String ima specijalan status u Javi, po dva osnova:

-Nove instance se mogu kreirati bez korišćenja operatora new, prostim navođenjem literala

-Operator + može da se koristi za konkatenaciju stringova

\* Bitni metodi:

public boolean equals(Object obj): ako je obj instanceof String, poredi sadržaj tekućeg stringa sa obj i vraća true ako su jednaki, u protivnom vraća false

public boolean equalsIgnoreCase(String anotherString): poredi sadržaj tekućeg stringa sa anotherString i vraća true ako su jednaki, ignorišući razliku između malih i velikih slova, u protivnom vraća false

public int length(): vraća dužinu (broj karaktera) tekućeg stringa

public char charAt(int i): vraća znak na poziciji i u tekućem stringu (pozicije, odnosno indeksi kreću od 0)

***Klasa String: primer***

class StringTest {

public static void main(String[] args) {

System.out.print("Unesite jedan string: ");

String s1 = Svetovid.in.readLine();

System.out.print("Unesite jos jedan: ");

String s2 = Svetovid.in.readLine();

System.out.println("Duzine stringova su " + s1.length() +

" i " + s1.length());

System.out.println("Stringovi pocinju znacima " + s1.charAt(0) + " i " + s2.charAt(0));

System.out.println("Stringovi " + (s1.equals(s2) ? "JESU" : "NISU") + " jednaki");

System.out.println("Ako se zanemari velicina slova,stringovi " + (s1.equalsIgnoreCase(s2) ? "JESU" :"NISU")+ " jednaki");

}

}

***Referencijalni ekvivalenti prostih tipova***

\* Za svaki prost tip, Java API sadrži odgovarajuću klasu koja predstavlja njegov referencijalni ekvivalent

\* Ovo su tzv. *wrapper* klase, odnosno klase koje služe da se vrednost prostog tipa “umota” u objekat

\* Svrha ovih klasa je da omoguće korišćenje prostih tipova i u situacijama kad je neophodno koristiti referencijalni tip

-Na primer, u neke strukture podataka kao što su standardne kolekcije iz Java API mogu da se smeštaju samo objekti

\* Uz to, ove klase definišu korisne metode za rad sa prostim tipovima

***Prost tip Referencijalni tip***

boolean Boolean

char Character

byte Byte

short Short

int Integer

long Long

double Double

float Float

**Primer:**

Integer pet = new Integer(5); // boxing

int p = pet.intValue(); // unboxing

int pedespet = Integer.parseInt("55");

Character a = new Character('a');

char c = a.charValue();

System.out.println("Brojevi: " + pet + ", " + p + ", " + pedespet);

System.out.println("Znak " + c + " " +

(Character.isLetter(c) ? "JESTE" : "NIJE") + " slovo");

System.out.println("Znak " + c + " " +

(Character.isDigit(c) ? "JESTE" : "NIJE") + " cifra");

System.out.println("Znak " + c + " " +

(Character.isUpperCase(c) ? "JESTE" : "NIJE") + " veliko slovo");

System.out.println("Znak " + c + " " +

(Character.isLowerCase(c) ? "JESTE" : "NIJE") + " malo slovo");

Izlaz:



***20. Nabrojivi tip***

\* Podsetimo se da tip podataka određuje skup vrednosti koje promenljiva može da ima

\* Ako postoji potreba za malim skupom konstantnih vrednosti koji se neće menjati, sa jednostavnim operacijama koje će se primenjivati (poređenje jednakosti, i ne puno više od toga), može se koristiti nabrojivi tip (engl. *enumerated type*, skraćeno *enum*)

\* Nabrojivi tip se definiše pomoću ključne reči enum, nakon čega se navodi ime nabrojivog tipa, i unutar {} imenovane konstante razdvojene zarezima

\* **Primer:**

enum OsnovneBoje {

CRVENA, ZELENA, PLAVA

}

\* Lista konstanti može se završiti i znakovima ; i ,

enum OsnovneBoje {

CRVENA, ZELENA, PLAVA

}

enum Oblici {

TROUGAO, KVADRAT, PETOUGAO, SESTOUGAO, KRUG

}

class EnumTest {

public static void main(String[] args) {

OsnovneBoje boja = OsnovneBoje.ZELENA;

Oblici oblik = Oblici.KVADRAT; // mora kvalifikovano

if (boja == OsnovneBoje.PLAVA)

System.out.println(boja + ": volim tu boju");

else

System.out.println(boja + ": ne volim je");

}

}

\* Iako je referencijalni tip, nad nabrojivim tipom se ne primenjuje operator new: sve moguće instance su već kreirane i reference na njih dodeljene nabrojanim konstantama

\* Nabrojivi tip može se koristiti u switch naredbi:

switch (oblik) {

case TROUGAO: // mora nekvalifikovano

System.out.println("Vidim trougao!");

break;

case KVADRAT:

System.out.println("Vidim kvadrat!");

break;

default:

System.out.println("Vidim nesto drugo...");

}

\* Statički metod values() svakog nabrojivog tipa vratiće niz nabrojanih konstanti, što omogućava jednostavnu iteraciju kroz sve moguće vrednosti nabrojivog tipa, u redosledu u kom su nabrojane:

OsnovneBoje[] sveBoje = OsnovneBoje.values();

for (OsnovneBoje b : sveBoje) {

System.out.println(b);

}

\* Izlaz:

CRVENA

ZELENA

PLAVA

***21. Rekurzija, faktorijel***

\* Rekurzijom se, u opštem slučaju, naziva način definisanja (ili objašnjavanja) nekog pojma posredstvom istog tog pojma

\* U programiranju se rekurzijom naziva (neposredni ili posredni) poziv jednog metoda iz tela tog istog metoda

\* Rekurzija je jedan od čestih načina rešavanja problema u programiranju

\* Njome se prvobitan (težak) problem deli na manje probleme, koji su po svojoj prirodi istovetni sa polaznim, ali su jednostavniji

\* Novi (jednostavniji) problem se dalje deli na nove jednostavnije i tako redom

\* Međutim, jasno je da se postupak pojednostavljivanja ne može nastavljati beskonačno, i da se u jednom momentu mora zaustaviti

\* Nizom pojednostavljenja polazni problem se svodi na trivijalni, koji se lako rešava

\* Rešenje trivijalnog problema uslovljava povratni lanac formiranja rešenja od jednostavnijih ka složenijim sve do rešenja početnog problema

\* Rekurzivno rešenje je najprirodnije ako su po svojoj prirodi rekurzivni

-Problem koji se rešava, ili

-Struktura podataka koja se koristi u rešenju problema

\* Rekurzivno rešenje, naravno, nije uvek ni jedino moguće, niti najefikasnije rešenje problema

\* Na primer, za svako rekurzivno rešenje može se naći odgovarajuće nerekurzivno (iterativno) - koje od ta dva rešenja je bolje, ponekad je teško prosuditi

\* Često je međutim rekurzivno rešenje elegantnije, kraće i čitljivije, a nerekurzivno efikasnije

\* Konkretna efikasnost zavisi od računara na kom se program izvršava, od realizacije programskog jezika (kompajlera, interpretera), kao i od samog rešenja

\* U Javi (i ostalim programskim jezicima koji dozvoljavaju rekurziju), razlikujemo dva tipa rekurzije:

-Direktna rekurzija (samorekurzija), kada metod neposredno poziva samog sebe

-Indirektna (uzajamna) rekurzija, kada metod poziva samog sebe posredno preko drugih metoda koje poziva

***Rekurzija: rešavanje problema***

\* U svakom rekurzivnom metodu mora da postoji:

-Rešenje opšteg problema njegovom dekompozicijom na manje probleme koji se rešavaju rekurzivnim pozivom (pozivima),

-Rešenje trivijalnog slučaja kojim se problem rešava direktno bez daljih rekurzivnih poziva,pri čemu dekompozicija opšteg problema mora postepeno svoditi početni problem na trivijalan slučaj (ili trivijalne slučajeve)

\* Posmatrajmo za početak način funkcionisanja i efekte rekurzije na jednom jednostavnom, standardnom i nezaobilaznom primeru: izračunavanju faktorijela nekog prirodnog broja

***Rekurzija: faktorijel***

\* Izračunavanje faktorijela broja *n* (u oznaci *n*!) je problem koji se prirodno rešava rekurzijom. Po definiciji,

što znači da pri izračunavanju *n*! rešenje svodimo na izračunavanje (*n*–1)!, potom rešavanje (*n*–1)! svodimo (na isti način) na rešavanje (*n*–2)! itd.

\* Trivijalan slučaj je ako je *n* = 1, kada je rešenje 1

\* Ispunjen je i uslov da se dekompozicijom početni problem postepeno svodi na trivijalan slučaj

class Faktorijel {

static int fakt(int n) {

if (n == 1) {

return 1;

}

else {

return fakt(n - 1) \* n;

}

}

public static void main(String[] args) {

int n;

do {

System.out.print("Unesite broj n za koji se racuna n! (n>0): ");

n = Svetovid.in.readInt();

} while (n <= 0);

System.out.println("n! = " + fakt(n));

}

}

\* Možemo primetiti da se nizom rekurzivnih poziva problem pojednostavljuje i izračunavanje odlaže, a da se, kada je rešen trivijalni problem, u nizu povrataka iz rekurzivnih poziva, izračunava konačno rešenje

\* Memorijski prostor se zauzima za svaki novi rekurzivni poziv procedure, on je aktuelan dok se procedura izvršava i oslobađa se kada je rekurzivni poziv procedure završen

\* Rezultat izračunavanja iz jednog poziva prenosi se u prethodni poziv

***22. Hanojske kule***

\* Za ovaj problem je vezano mnogo legendi, ali jedna od njih naročito slikovito prikazuje koliko je jednostavno neke izuzetno teške probleme rešiti kada se razmišlja rekurzivno

\* Legenda kaže da je pre mnogo godina, negde u okolini grada Hanoj, živeo car koji je tražio novog dvorskog mudraca. Pošto je i sam bio mudar, želeo je da pronađe što boljeg mudraca, pa je rešio da uzme onoga koji da najbolje rešenje postavljenog problema (zagonetke):

Dato je tri štapa i *n* diskova, različitog prečnika. Svi diskovi su postavljeni na prvom štapu tako da su uvek manji disk nalazi iznad većeg. Potrebno je prebaciti sve diskove sa izvornog štapa na ciljni, premeštajući jedan po jedan disk, koristeći treći štap kao pomoćni, ali da se ni u jednom momentu disk većeg prečnika ne nađe na disku manjeg prečnika. Mnogi mudraci iz cele zemlje dolazili su pred cara sa raznim rešenjima, ali su rešenja bila ili nerazumljiva ili predugačka. „Mora da postoji jednostavniji način“, razmišljao je car. Jednog dana je pred cara stigao Buda, i rekao da je problem toliko jednostavan da se rešava sam od sebe. Svoje rešenje Buda je izložio na sledeći način.

1. Ako postoji samo jedan disk, pomeramo ga sa izvornog na ciljni štap, i to je toliko jednostavan posao da ga može uraditi svaka seoska luda.

2. Ako pak ima više od jednog diska postupak je sledeći

Premestimo prvo *n*–1 diskova sa izvornog na pomoćni štap, koristeći ciljni kao pomoćni . Pošto je *n*–1 diskova na pomoćnom štapu, a najveći je i dalje ostao na izvornom, problem se svodi na tačku 1, tj. treba prebaciti taj jedan disk sa izvornog na ciljni štap. Potom treba *n*–1 diskova, sa pomoćnog štapa prebaciti na ciljni po istom postupku (sada koristeći izvorni štap kao pomoćni). Kada je Buda završio sa pričom, car ga je pitao kada će konačno reći svoje rešenje. Buda se samo nasmešio i otišao.

class HanojskeKule {

static void hKule(int n, int izvor, int cilj, int pomoc) {

if (n == 1) { // jedan disk

System.out.println("Prebaci disk sa stapa " + izvor " na stap " + cilj);

}

else { / / vise diskova

hKule(n-1, izvor, pomoc, cilj);

// n-1 diskova sa izvor na pomoc, koristeci cilj kao pomocni

hKule(1, izvor, cilj, pomoc);

// prebaci jedini disk sa izvor na cilj

hKule(n-1, pomoc, cilj, izvor);

// n-1 diskova sa pomoc na cilj, koristeci izvor kao pomocni

}

}

public static void main(String[] args) {

int n;

do {

System.out.print("Unesite broj diskova n (n>0): ");

n = Svetovid.in.readInt();

} while (n <= 0);

hKule(n, 1, 2, 3);

}

}

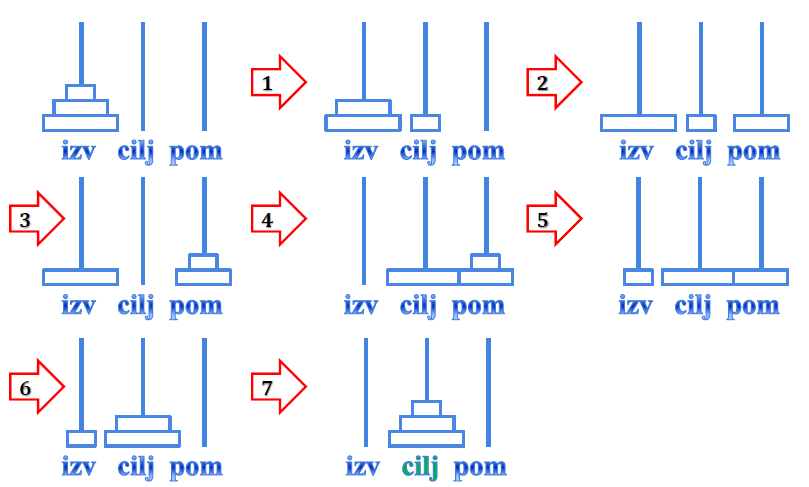
Izlaz:



\* U okviru metoda hKule uočavamo tri rekurzivna poziva istog metoda

\* Kako se u svakom pozivu metoda mogu pojaviti tri nova poziva, uočava se “eksplozija” broja rekurzivnih poziva

\* Ovo rešenje proizvodi 2*n* – 1 poteza za *n* diskova



***23. Sporedni efekti, vidljivost imena i život promenljivih***

***Sporedni efekti***

\* Posmatrajmo statičke metode, za koje smo rekli da su jedinstvene celine za ceo program (klasu)

\* Za “komunikaciju” metoda sa spoljašnjim okruženjem postoje tri mogućnosti:

1.Sve vrednosti se u metod unose i sve vrednosti se iz metoda iznose posredstvom “spoljašnjih” promenljivih (npr. statičkih polja klase)

2.Sve vrednosti se u metod unose i iz njega iznose posredstvom parametara i (eventualno) povratne vrednosti (za “iznošenje” vrednosti)

3.Kombinacijom prethodna dva načina

\* Metodi koji koriste 2. način su logičke celine koje ne zavise od globalnih vrednosti i trenutnog stanja okruženja programa - njihovo ponašanje je u potpunosti određeno formalnim parametrima

\* Pored toga, takvi statički metodi se lako i bez izmena mogu koristiti i u drugim programima

\* Kad god je to moguće treba koristiti upravo ovakve statičke metode, jer su opštiji i “prenosiviji”, a programi su čitljiviji, jasniji i omogućavaju lakše otkrivanje grešaka

\* Rekli smo da nije dobro ako ponašanje metoda zavisi od vrednosti “spoljašnjih” promenljivih

\* Još je manje dobro ako metodi menjaju vrednosti spoljašnjih promenljivih, jer se tako stvaraju programi čije se izvršavanje teško prati

\* Menjanje spoljašnjih promenljivih iz metoda se često naziva sporednim efektom (engl. *side effect*) i generalno se smatra lošom praksom u programiranju

\* Menjanje spoljašnjih promenljivih iz statičkih metoda može često biti uzrok grešaka u radu programa koje se vrlo teško otkrivaju

\* Prirodno je, bar za matematičare, da (na primer) izraz f(x)+g(x) ima istu vrednost kao i izraz g(x)+f(x), jer je + komutativna operacija

\* Međutim, kada se programira uz pomoć sporednih efekata, to više ne mora biti tako

\* U primeru koji sledi, kao rezultat poziva f(vred)+g(vred) i g(vred)+f(vred) dobiće se različite vrednosti, bez obzira što su polazna okruženja u kojima su se izračunavanja vršila bila ista

\* Primer je jednostavan i pomalo veštački, ali je čak i u njemu teško pratiti šta se i kako menja

\* Ako je program duži i promene promenljivih nesmotrene, kasnije menjanje programa će biti otežano, a greške nastale usled sporednih efekata teško će se pronaći

***Sporedni efekti: primer***

class SideEffect {

static int vred;

static int f(int x) {

vred += x;

return vred;

}

static int g(int x) {

return vred + x;

}

public static void main(String[] args) {

vred = 1;

System.out.println("f(vred) + g(vred) = " + (f(vred) + g(vred)));

vred = 1;

System.out.println("g(vred) + f(vred) = " + (g(vred) + f(vred)));

}

}

\* Kod nestatičkih metoda važi sličan princip

\* Razlika je u tome što u tom slučaju “jedinstveni entitet” ne treba da bude sam metod kao kod statičkih metoda, već objekat kome nestatički metod pripada

\* Tako da je dozvoljeno i poželjno da nestatički metod zavisi od i utiče na vrednosti polja objekta kom pripada

\* Međutim, zavisnost od i uticaj na bilo koje promeljive “spoljašnjije” od polja istog objekta nisu poželjni

***Vidljivost imena i život promenljivih***

\* Oblast dejstva, doseg ili vidljivost imena (engl. *scope*) jesu sinonimi koji označavaju delove programa u kome važi deklaracija imena, tj. u kome ime može da se koristi (u kome se “vidi”)

\* Život (ili trajanje) promenljive (engl. *extent*) je deo programa u kome promenljiva fizički postoji (“živi”), bez obzira da li je trenutno vidljiva ili ne

\* U Javi, osnovni mehanizmi za određivanje vidljivosti imena i života promenljivih su paketi, klase (odnosno ref. tipovi), metodi i blokovi

\* Paketi utiču samo na vidljivost imena referencijalnih tipova

\* Na život promenljive mogu uticati dva mehanizma:

-Blokovi (uključujući tela metoda)

-Kreiranje objekata, odnosno postojanje reference na objekat kreiran u memoriji (heap)

\* Lokalne promenljive žive i vidljive su u bloku u kom su deklarisane, i to od deklaracije do kraja bloka

-Memorijski prostor za vrednost lokalne promenljive kreira se prilikom deklaracije, a po izlasku iz bloka se oslobađa

-Pri tom, ako je lokalna promenljiva referencijalnog tipa, ovo važi samo za vrednost promenljive što je referenca - kreiran objekat na koji referenca pokazuje može da živi “posebnim životom”

\* Argumenti metoda žive i vidljivi su u telu metoda (bloku) do njegovog kraja

\* Do kraja diskusije pod pojmom “klasa” podrazumevamo “referencijalni tip”

\* Statička polja klase žive celim tokom izvršavanja programa koji koristi klasu

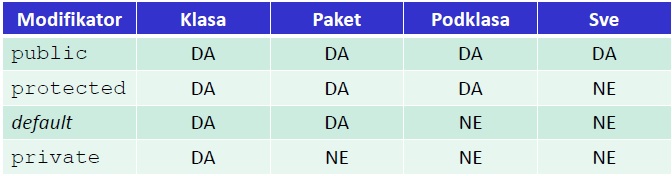
\* Na vidljivost statičkih polja utiču modifikatori vidljivosti:

-Polja public vidljivosti su vidljiva svuda (gde je vidljiva i klasa)

-Polja protected vidljivosti su vidljiva u svim klasama iz istog paketa i svim podklasama klase

-Polja *default* vidljivosti su vidljiva u svim klasama iz istog paketa

-Polja private vidljivosti su vidljiva samo u okviru svoje klase



\* Nestatička polja klase žive dokle god živi objekat kom pripadaju

-Objekat živi ako je kreiran (operatorom new i sl.) i na njega pokazuje bar jedna referenca u programu (tj. živi bar jedna promenljiva čija je vrednost referenca na taj objekat)

\* Za vidljivost nestatičkog polja objekta neophodan preduslov je da je moguće doći do reference na taj objekat

-Tada za vidljivost nestatičkog polja važe ista pravila kao za statička polja

\* Ključna reč this može da se koristi u svakom nestatičkom kontekstu (konstruktori, nestatički metodi, inicijalizacija nestatičkih polja), gde označava referencu na tekući objekat

\* Dozvoljeno je da formalni parametri metoda i konstruktora, kao i lokalne promenljive u datoj klasi imaju isto ime kao i polja, čime je ime polja u metodu/konstruktoru/bloku redefinisano

\* U takvim slučajevima i dalje je moguće pristupiti poljima klase

-Za nestatička polja: navođenjem this, tačke, i imena polja

-Za statička polja:

\* Isto kao za nestatička polja, ili

\* Navođenjem imena klase, tačke, i imena polja

class Tacka {

static final double JEDAN = 1.0;

double x, y;

Tacka(double x, double y) {

this.x = x; this.y = y;

}

double testX() {

int x = 2;

final int JEDAN = -1;

return this.x \* x + Tacka.JEDAN + this.JEDAN;

}

public String toString() {

return "(" + this.x + ", " + this.y + ")";

}

}

class ThisTest {

public static void main(String[] args) {

Tacka t = new Tacka(1.0, 2.0);

double x = t.testX();

System.out.println("t.testX() = " + x); // 4.0

}

}

***24. Algoritmi, kriterijumi razlikovanja/izbora. Rekurentni nizovi jedne promenljive,*** ***primer: parni brojevi***

\* Sastavni delovi svakog programa su **algoritmi** i **strukture podataka**

\* **Algoritam** je diskretan, jednoznačan, čisto mehanički postupak, koji je definisan konačnim brojem pravila i koji dovodi do rešenja svakog konkretnog problema iz nekog skupa problema u konačno mnogo koraka

\* Reč “algoritam” potiče od latiniziranog imena arapskog matematičara Muhameda ibn-Muse al-Hvarizmija koji je živeo u IX veku i između ostalog napisao knjigu o “računu Indijaca”, tj. o računanju sa arapskim ciframa

\* Prvobitno je reč “algoritam” označavala pravila za sabiranje, oduzimanje, množenje i deljenje, a kasnije se pojam algoritma proširio da obuhvata sve mehaničke postupke

\* Jedini sačuvani latinski prevod odlomka te knjige počinje rečima “Dixit Algorismi...” (Algorizmi je rekao...), po čemu je i današnji pojam algoritam dobio ime

\* Svi programi koji se izvode na računarima su realizacija algoritama - da bi se problem mogao rešiti na računaru mora da bude algoritamski rešiv

\* Ukoliko neki problem ima algoritamsko rešenje, tada obično ima više različitih načina za njegovo rešavanje pa se prirodno postavlja pitanje koji je od mogućih algoritama za rešavanje problema bolji

**Kriterijumi po kojima se algoritmi razlikuju, i po kojima se može vršiti izbor pravog algoritma za dati problem, su sledeći:**

\* **Korektnost:** algoritam treba da ispravno reši postavljeni problem. Ovaj uslov zahteva da problem bude dovoljno precizno postavljen da bismo uopšte bili u stanju da analiziramo da li je problem rešen korektno

\* **Količina memorije** potrebna za rešavanje problema. Ovaj kriterijum vremenom gubi na značaju jer je memorija postala dovoljno jeftina, tako da se često ne isplati gubiti vreme na skraćivanje programa ili optimizaciju potrošene memorije na uštrb jasnoće koda

\* **Jasnoća i jednostavnost.** Ovaj kriterijum sve više dobija na značaju, jer se danas potencira zahtev da se problem što pre reši na što jasniji način, te da to rešenje bude što shvatljivije i lako za izmene i prilagođavanja

\* **Efikasnost.** Pretpostvalja se da je efikasniji algoritam bolji, a da bi se definisalo šta se pod efikasnim algoritmom podrazumeva, uvodi se pojam složenosti algoritma. Kao mera složenosti uzima se broj koraka koji je potrebno izvesti da bi se do rešenja došlo. Broj koraka za rešavanje jednog problema najčešće zavisi i od veličine početnih podataka pa se broj koraka izražava kao funkcija veličine polaznih podataka

***Rekurentni nizovi jedne promenljive***

\* U mnogim oblastima primene računara javlja se potreba za izračunavanjem elemenata matematičkog niza

\* Jedna vrsta matematičkih nizova pogodnih za izračunavanje su rekurentni nizovi, tj. nizovi {*sn*}, gde je *n*-ti element niza, *sn*, definisan preko prethodnih elemenata niza (*sn*-1, *sn*-2, ...)

\* Posmatraćemo prvo najjednostavniji slučaj kada je element niza definisan preko tačno jednog prethodnog:



gde funkcija *f* može da ima još neke parametre koji zavise od *n*

\* Zbog toga što elementi niza zavise od prethodnih kaže se “**rekurentni**”, a za gornji slučaj pošto element zavisi od tačno jednog prethodnog kaže se “**jedne promenljive**”

\* Indekse (matematičkog) niza predstavićemo Javinim tipom int, a elemente tipom double

\* Funkciju *f* za računanje sledećeg elementa niza predstavićemo metodom:

static double f(double stari, int n)

\* Metod f će za dati indeks n i prethodni element niza *sn*-1 prosleđen u parametru stari vratiti izračunatu vrednost novog elementa *sn*

\* Moguće je da metod f osim indeksa n koristi i još neke dodatne parametre koji su zbog jednostavnosti izostavljeni

\* Uz navedene pretpostavke dobija se sledeći metod za računanje elemenata niza:

static double s(int n, double pocetnaVrednost) {

double tekucaVrednost = pocetnaVrednost;

for (int i = 1; i <= n; i++) {

tekucaVrednost = f(tekucaVrednost, i);

}

return tekucaVrednost;

}

\* Sledećim programskim fragmentom ispisuje se vrednost izračunatog elementa niza:

System.out.print("Unesite n: ");

int n = Svetovid.in.readInt();

System.out.print("Unesite pocetnu vrednost: ");

double a = Svetovid.in.readDouble();

System.out.println("s(n) = " + s(n, a));

***Primer: niz parnih prirodnih brojeva***

\* Niz parnih prirodnih brojeva može se definisati na sledeći način:



\* Lako se vidi da je *s*0 = 0, *s*1 = 2, *s*2 = 4, ...

\* Kad se primeni opisana metodologija dobija se sledeći program za računanje *n*-tog elementa ovog niza

***Primer: niz parnih prirodnih brojeva***

class Parni {

static double f(double stari, int n) {

return stari + 2;

}

static double s(int n, double pocetnaVrednost) {

double tekucaVrednost = pocetnaVrednost;

for (int i = 1; i <= n; i++) {

tekucaVrednost = f(tekucaVrednost, i);

}

return tekucaVrednost;

}

public static void main(String[] args) {

System.out.print("Unesite n: ");

int n = Svetovid.in.readInt();

System.out.println("s(n) = " + s(n, 0));

}

}

***25. Rekurentni nizovi jedne promenljive, primer: binomni koeficijenti***

***Rekurentni nizovi jedne promenljive***

\* U mnogim oblastima primene računara javlja se potreba za izračunavanjem elemenata matematičkog niza

\* Jedna vrsta matematičkih nizova pogodnih za izračunavanje su rekurentni nizovi, tj. nizovi {*sn*}, gde je *n*-ti element niza, *sn*, definisan preko prethodnih elemenata niza (*sn*-1, *sn*-2, ...)

\* Posmatraćemo prvo najjednostavniji slučaj kada je element niza definisan preko tačno jednog prethodnog:



gde funkcija *f* može da ima još neke parametre koji zavise od *n*

\* Zbog toga što elementi niza zavise od prethodnih kaže se “**rekurentni**”, a za gornji slučaj pošto element zavisi od tačno jednog prethodnog kaže se “**jedne promenljive**”

\* Indekse (matematičkog) niza predstavićemo Javinim tipom int, a elemente tipom double

\* Funkciju *f* za računanje sledećeg elementa niza predstavićemo metodom:

static double f(double stari, int n)

\* Metod f će za dati indeks n i prethodni element niza *sn*-1 prosleđen u parametru stari vratiti izračunatu vrednost novog elementa *sn*

\* Moguće je da metod f osim indeksa n koristi i još neke dodatne parametre koji su zbog jednostavnosti izostavljeni

\* Uz navedene pretpostavke dobija se sledeći metod za računanje elemenata niza:

static double s(int n, double pocetnaVrednost) {

double tekucaVrednost = pocetnaVrednost;

for (int i = 1; i <= n; i++) {

tekucaVrednost = f(tekucaVrednost, i);

}

return tekucaVrednost;

}

\* Sledećim programskim fragmentom ispisuje se vrednost izračunatog elementa niza:

System.out.print("Unesite n: ");

int n = Svetovid.in.readInt();

System.out.print("Unesite pocetnu vrednost: ");

double a = Svetovid.in.readDouble();

System.out.println("s(n) = " + s(n, a));

***Primer: binomni koeficijenti***

\* Prethodni primer je jednostavan i ne ilustruje punu snagu pristupa preko rekurentnih nizova,jer postoji trivijalan odnos između indeksa i elementa niza: *sn* = 2*n*

-Stoga je u ovom slučaju pristup preko rekurentnih nizova previše komplikovan i neefikasan

\* Zato ćemo obraditi i malo složeniji primer gde se bolje vide prednosti ovakvog pristupa - računanje vrednosti binomnih koeficijenata

\* Za date prirodne brojeve *n* i *k*, binomni koeficijent je broj koji označava koliko različitih podskupova od *k* elemenata je moguće izdvojiti iz skupa od *n* elemenata

-Standardna oznaka:



\* Formula po kojoj se računa binomni koeficijent:



\* Vrednost binomnog koeficijenta se može izračunati na jednostavan način: izračunaju se vrednosti svih faktorijela, pa se pomnože/podele

\* Međutim, ovaj “naivni” pristup ima jedan veliki nedostatak

-Faktorijeli su veliki brojevi, pa lako dolazi do prekoračenja opsega brojevnih tipova pri njihovom računanju

-S druge strane, binomni koeficijenti nisu toliko veliki brojevi kao faktorijeli, i bilo bi poželjno izbeći rad sa velikim brojevima

\* Zato ćemo binomne koeficijente posmatrati kao rekurentni niz, gde će nam *n* biti fiksan broj, a indeksi niza biće po *k*, tako da će element niza biti:



\* Preostalo je da odredimo:

-Početnu vrednost rekurentnog niza, *s*0

-Funkcionalnu zavisnost ( *f* ) između elemenata *sk* i *sk*-1 za *k* ≥ 1

***Primer: binomni koeficijenti***

\* Početna vrednost:



\* Funkcionalnzu vezu *sk* i *sk*-1 je najjednostavnije naći preko količnika *sk* / *sk*-1 :



tako da je



\* Početna vrednost:

\* Funkcionalnzu vezu *sk* i *sk*-1 je najjednostavnije naći preko količnika *sk* / *sk*-1 :

tako da je

static double f(double stari, int n, int k) {

return stari \* (n - k + 1) / k;

}

static double bk(int n, int k) {

if (n >= 0 && k >= 0) {

if (k > n) {

return 0;

}

else {

if (k > n - k) k = n - k;

double tekucaVrednost = 1;

for (int i = 1; i <= k; i++) {

tekucaVrednost = f(tekucaVrednost, n, i);

}

return tekucaVrednost;

}

}

else {

return -1;

}

}

***26. Sume i proizvodi***

Važan specijalan slučaj rekurentnih nizova su sume i proizvodi:



Predstavićemo prvo dva rekurzivna načina da se izračuna vrednost sume (računanje proizvoda je analogno)

Pretpostavimo da je definisan metod koji računa vrednost sabirka *an*:

static double sabirak(int n)

Izračunavanje sume može se implementirati sledećim rekurzivnim metodom:

static double suma(int n) {

if (n == 0)

return 0.0;

else

return suma(n-1) + sabirak(n);

}

Međutim, ova implementacija je neefikasna, jer se čuvaju svi međurezultati izračunavanja u lancu rekurzivnih poziva

Moguće je rekurzivno izračunavanje sume organizovati tako da se u samom metodu ne radi ništa nego se izračunavanje izvodi u zaglavlju metoda pri prenošenju parametara

Ovako korišćeni parametri nazivaju se **akumulirajući parametri**

static double suma(double zbir, int i, int n) {

if (i > n)

return zbir;

else

return suma(zbir + sabirak(i), i + 1, n);

}

public static void main(String[] args) {

System.out.print("Unesite n: ");

int n = Svetovid.in.readInt();

System.out.println("suma(n) = " + suma(0.0, 1, n));

}

U većini programskih jezika sume i proizvodi se izračunavaju efikasnije korišćenjem petlji, najčešće varijacijom sledećih opštih postupaka:

suma = 0.0; proizvod = 1.0;

i = 1; i = 1;

while (i <= n) { while (i <= n) {

izracunati sabirak ai; izracunati cinilac bi;

suma += ai; proizvod \*= bi;

i++; i++;

} }

ili korišćenjem odgovarajuće for ili do-while petlje

***Suma recipročnih vrednosti***

Treba napisati metod koji računa sumu recipročnih vrednosti prvih *n* prirodnih brojeva (*n* ≥ 1)

Pošto je izračunavanje sabiraka jednostavno, nećemo za to koristiti posebnan metod

Koristićemo for petlju

Ako korisnik unese *n* koje nije u dozvoljenom opsegu, metod će vratiti -1 i tako signalizirati grešku

static double sumaRV(int n) {

final int DONJA\_GR = 1;

if (n >= DONJA\_GR) {

double suma = 1.0;

for (int i = 2; i <= n; i++) {

suma += 1.0 / i;

}

return suma;

}

else {

return -1.0;

}

}

***Suma kvadrata***

Treba napisati metod koji računa sumu kvadrata prvih *n* prirodnih brojeva (0 ≤ *n* ≤ 2000)

Pošto je izračunavanje sabiraka jednostavno, opet nećemo za to koristiti posebnan metod

Koristićemo while petlju

Ako korisnik unese *n* koje nije u dozvoljenom opsegu, metod će vratiti -1 i tako signalizirati grešku

Takođe, metod će vratiti -1 ako dođe do prekoračenja opsega pri računanju sume

-Sabirci ne mogu prekoračiti opseg, jer je 20002 < Integer.MAX\_VALUE

static int sumaKvad(int n) {

final int DONJA\_GR = 1;

final int GORNJA\_GR = 2000;

if (DONJA\_GR <= n && n <= GORNJA\_GR) {

boolean ok = true;

int i = 1;

int suma = 0;

while (ok && i <= n) {

int sabirak = i \* i;

ok = Integer.MAX\_VALUE - suma > sabirak;

if (ok) suma += sabirak;

i++;

}

if (ok) return suma;

else return -1;

}

else {

return -1;

}

}

***27. Izračunavanje an2***

\* Treba napisati metod koji računa za dati realan broj *a* i ceo broj *n* računa *an*2 koristeći množenje

\* Kako je stepenovanje specijalan slučaj proizvoda, problem možemo rešiti jednostavnim programskim fragmentom:

int rez = 1;

for (int i = 1; i <= n\*n; i++) {

rez \*= a;

}

\* Ovaj postupak jeste jednostavan, ali zahtena *n*2 množenja

\* Ukoliko primenimo opšti postupak za izračunavanje elemenata rekurentnog niza jedne promenljive, *sn* = *f*(*sn*-1), treba naći funkcijsku vezu *f*

\* U ovom slučaju (kao što smo imali i kod binomnih koeficijenata) pogodno je tražiti vezu oblika *sn* = *y sn*-1, za neko nepoznato *y* koje treba odrediti

\* Za *a* ≠ 0 dobijamo:



pa je:



\* Ako se sad i izračunavanje *a*2*n*-1 realizuje postupkom za izračunavanje elemenata rekurentnog niza, dobijamo:





\* Konačno se dobija da se *an*2 izračunava pomoću dva rekurentna niza {*sn*} i {*xn*}

\* Za izračunavanje je potrebno samo 2*n* množenja, umesto *n*2 množenja iz prvobitne verzije

\* Radi jednostavnosti nećemo proveravati prekoračenje opsega

static double an2(double a, int n) {

if (a == 0.0) {

return 0.0;

}

else {

n = Math.abs(n);

double s = 1.0;

double x = 1.0 / a;

double cinilac = a \* a;

for (int i = 1; i <= n; i++) {

x \*= cinilac;

s \*= x;

}

return s;

}

}

***28. Stepenovanje uzastopnim kvadriranjem, rekurzivan metod***

***Izračunavanje an***

\* S obzirom da smo uspeli pronaći način za efikasnije računanje *an*2, postavlja se pitanje da li je slično moguće uraditi i u opštem slučaju kod računanja *an*

-Tačnije, da li postoji način da se izračuna *an* korišćenjem (mnogo) manje od *n* operacija množenja?

\* Odgovor je potvrdan

\* Ranije smo pri računanju *an* u stvari koristili funkcionalnu vezu *an* = *an*-1 *a*

\* Sad je ideja da koristimo vezu *an* = *an*-*k ak*

\* Postupak je obično najefikasniji za *k* = *n* / 2, pa dobijamo dekompoziciju

*an* =*an*/2 *an*/2 = (*an*/2)2

\* Postupak gde računanje *an* svodimo na računanje *an*/2 kog zatim kvadriramo, naziva se **uzastopno kvadriranje**

\* S obzirom da je izložilac *n* ceo broj mora se voditi računa o njegovoj parnosti, a stepen se dobija rekurzivno sledećim izračunavanjem:

if (n % 2 == 1)

return a \* sqr(rekStepen(a, n/2));

else

return sqr(rekStepen(a, n/2));

\* Trivijalan slučaj i izlaz iz rekurzije je *a* za *n* = 1

\* Na osnovu ove ideje može se napisati rekurzivan metod i ako se ne vodi računa o eventualnom prekoračenju, dobija se sledeći kod

static double sqr(double a) {

return a \* a;

}

static double rekStepen(double a, int n) {

if (n == 1)

return a;

else if (n % 2 == 1)

return a \* sqr(rekStepen(a, n/2));

else

return sqr(rekStepen(a, n/2));

}

static double stepen(double a, int n) {

if (a == 0.0 && n <= 0)

return Double.NaN;

else {

if (a == 0.0)

return 0.0;

else if (n == 0 || a == 1.0)

return 1.0;

else {

if (n < 0) {

a = 1.0 / a;

n = Math.abs(n);

}

return rekStepen(a, n);

}

}

}

***29. Stepenovanje uzastopnim kvadriranjem, iterativan metod***

***Izračunavanje an***

\* S obzirom da smo uspeli pronaći način za efikasnije računanje *an*2, postavlja se pitanje da li je slično moguće uraditi i u opštem slučaju kod računanja *an*

-Tačnije, da li postoji način da se izračuna *an* korišćenjem (mnogo) manje od *n* operacija množenja?

\* Odgovor je potvrdan

\* Ranije smo pri računanju *an* u stvari koristili funkcionalnu vezu *an* = *an*-1 *a*

\* Sad je ideja da koristimo vezu *an* = *an*-*k ak*

\* Postupak je obično najefikasniji za *k* = *n* / 2, pa dobijamo dekompoziciju

*an* =*an*/2 *an*/2 = (*an*/2)2

\* Postupak gde računanje *an* svodimo na računanje *an*/2 kog zatim kvadriramo, naziva se **uzastopno kvadriranje**

\* S obzirom da je izložilac *n* ceo broj mora se voditi računa o njegovoj parnosti, a stepen se dobija rekurzivno sledećim izračunavanjem:

if (n % 2 == 1)

return a \* sqr(rekStepen(a, n/2));

else

return sqr(rekStepen(a, n/2));

\* Trivijalan slučaj i izlaz iz rekurzije je *a* za *n* = 1

\* Na osnovu ove ideje može se napisati rekurzivan metod i ako se ne vodi računa o eventualnom prekoračenju, dobija se sledeći kod

static double stepen(double a, int n) {

if (a == 0.0 && n <= 0)

return Double.NaN;

else {

if (a == 0.0)

return 0.0;

else if (n == 0 || a == 1.0)

return 1.0;

else {

if (n < 0) {

a = 1.0 / a;

n = Math.abs(n);

}

double stepen = 1.0;

while (n > 0) {

if (n % 2 == 1)

stepen \*= a;

n /= 2;

a \*= a;

}

return stepen;

}

}

}

***30. Opšti rekurentni nizovi***

\* Potpuno analogno rekurentnom nizu jedne promenljive definisanim sa *sn* = *f*(*sn*-1), može se definisati opšti postupak za izračunavanje elemenata niza {*sn*} datog rekurentnom relacijom *r*-tog reda (*r* ≥ 1)

\* Elementi niza izračunavaju se rekurentnim izrazom sa fiksnim brojem od *r* argumenata, tj. izrazom oblika:



gde su *pvi*, 0 ≤ *i* ≤ *r* – 1, date početne vrednosti

\* Neka su elementi niza tipa double i neka je R zadat kao konstanta: static final int R = 5;

\* Pretpostavićemo da je funkcionalna zavisnost realizovana metodom: static boolean f(double[] s)

\* Za razliku od metoda f kod rekurentnih nizova jedne promenljive, parametar s sadrži (Javin) niz od R + 1 elementa:

-Prvih R elemenata niza s predstavljaju R elemenata rekurentnog niza koji prethode elementu sa indeksom *n* kog računamo

-Poslednji element niza s sadržaće novoizračunati element: s[R] = *f*(s[R-1], s[R-2], …, s[0])

-Metod f vratiće logičku vrednost koja označava da li je došlo do neke greške

\* Neka su početne vrednosti zadate nizom pv, tako da je

pv[0] = *pv*0, …, pv[R-1] = *pvR*-1

\* Za prirodan broj *n* iz nekog dozvoljenog intervala (0 ≤ *n* ≤ *granica*) *n*-ti element rekurentnog niza može se izračunati sledećim programskim fragmentom

if (0 <= n && n <= GRANICA) {

ok = true;

if (n < R)

rezultat = pv[n];

else {

for (int i = 0; i < R; i++) s[i] = pv[i];

ok = f(s);

if (ok) {

if (n == R)

rezultat = s[R];

else {

int i = R;

do {

for (int j = 1; j <= R; j++)

s[j-1] = s[j];

ok = f(s);

i++;

} while (i < n && ok);

if (ok) rezultat = s[R];

}

else {

ok = false;

***31. Izračunavanje Fibonačijevih brojeva, osnovni i unapređen metod***

***Opšti rekurentni nizovi***

\* Često je *r* mali broj, pa se tada neke petlje mogu zameniti nizom ekvivalentnih naredbi

\* Takođe, izračunavanje narednog elementa često je jednostavno, pa se metod f može izostaviti

\* Kao primer opšteg rekurentnog niza navodimo Fibonačijeve brojeve

\* Svaki naredni Fibonačijev broj je zbir prethodna dva Fibonačijeva broja

***Fibonačijevi brojevi***

\* Rekurentni niz Fibonačijevih brojeva definisan je sa:



\* Primenićemo opšti postupak za izračunavanje rekurentnih nizova od *r* promenljivih, uz nekoliko pojednostavljenja:

-Nećemo definisati metod f zbog jednostavnosti izračunavanja novih elemenata

-Nećemo koristiti niz početnih vrednosti, nego ćemo ih odmah staviti u niz f

-Izbacićemo neke nepotrebne dodele promenljivoj ok, jer ćemo povratnom vrednošću -1 signalizirati grešku

-Elementi niza biće tipa int

\* Dobija se sledeći metod

static int fibonacci(int n) {

final int GRANICA = 50;

int[] f = new int[3];

boolean ok = true;

if (0 <= n && n <= GRANICA) {

f[0] = 0;

f[1] = 1;

f[2] = f[1] + f[0];

if (n <= 2)

return f[n];

else {

int i = 2;

do {

f[0] = f[1];

f[1] = f[2];

ok = Integer.MAX\_VALUE - f[1] > f[0];

if (ok) f[2] = f[1] + f[0];

i++;

} while (i < n && ok);

if (ok) return f[2];

}

}

return -1;

}

\* U slučaju Fibonačijevih brojeva izračunavanje može da se organizuje malo efikasnije koristeći činjenicu da je veza među njima jednostavna

\* Naime, ako se sabiranje vrši i u f[0] i u f[1], tako da u f[0] budu Fibonačijevi brojevi sa parnim indeksima, a u f[1] sa neparnim, tada se korak u petlji može povećavati za dva, što će skratiti broj prolazaka kroz petlju na pola

\* Takođe, ne mora se koristiti niz, nego obične promenljive

\* Radi jednostavnosti, eliminisaćemo i proveru gornje granice

static int fibonacci1(int n) {

int f0, f1;

boolean ok = true;

if (0 <= n) {

f0 = 0;

f1 = 1;

int i = 1;

while (i < n && ok) {

ok = Integer.MAX\_VALUE - f1 - f1 > f0;

// dva puta oduzimamo f1 jer cemo ga dva puta dodati

if (ok) {

f0 = f0 + f1;

f1 = f0 + f1;

i += 2;

}

}

if (ok)

if (n % 2 == 1) return f1;

else return f0;

}

return -1;

}

***32. Izračunavanje Fibonačijevih brojeva, rekurzivno preko definicije i korišćenjem*** ***akumulirajućeg parametra***

\* Fibonačijevi brojevi mogu da se izračunavaju i rekurzivno, direktnim korišćenjem definicije:

static int fibonacci2(int n) {

if (0 <= n)

return fib(n);

else

return -1;

}

static int fib(int n) {

if (n <= 1)

return n;

else

return fib(n-1) + fib(n-2);

}

\* Prethodni način izračunavanja je vrlo neefikasan, jer se rekurzivnim pozivima metoda fib prethodni elementi rekurentnog niza nepotrebno izračunavaju više puta

\* Npr. za izračunavanje *f*25 = 75025 izvršava se 242785 poziva metoda fib, dok se iterativnim postupkom u metodi fibonacci1 koristi samo 26 sabiranja (ne računajući proveru prekoračenja opsega i povećanje brojača)

\* Korišćenjem tehnike akumulirajućih parametara moguće je rekurzivno rešenje po efikasnosti, a i po prirodi, približiti iterativnom

-Za računanje *fn* biće potrebno samo *n*+1 poziva metoda

-Metodom fib ćemo u stvari simulirati while petlju, jer će u svakoj “iteraciji” (rekurzivnom pozivu) parametar f1 dobiti vrednost f0 + f1, parametar f0 će dobiti vrednost f1, a poslednji parametar će igrati ulogu brojača

static int fibonacci3(int n) {

if (0 <= n)

return fib(1, 0, n);

else

return -1;

}

static int fib(int f1, int f0, int n) {

if (n == 0)

return f0;

else

return fib(f0+f1, f1, n-1);

}

***33. Izračunavanje Fibonačijevih brojeva pomoću matrica***

\* Jedan od najefikasnijih načina za izračunavanje Fibonačijevih brojeva je da se iskoristi činjenica da se Fibonačijevi brojevi dobijaju kao elementi stepena matrice *F*:



što se lako dokazuje matematičkom indukcijom

\* Za *k* = 1 tvrđenje je trivijalno tačno. Neka je tačno za *k* = *n*, tada je:



pa je



\* Posmatrajmo matricu *G* = *F*2, i raspišimo rekurentni niz sa elementima *Gk* = (*F*2) *k*.

Dobija se:



gde je matrica *G* definisana kao:



\* Uočimo da su elementi donje vrste matrice *Gk* jednaki *f*2*k* i *f*2*k*+1

\* Za izračunavanje stepena matrice *G* nije potrebno pamtiti sva 4 elementa, dovoljna su dva, npr. elementi donje vrste *g*1 = *g*21 i *g*2 = *g*22 jer:

-*g*12 = *g*21 (matrica je simetrična)

-*g*11 = *g*22 – *g*21 = *g*2 – *g*1 (elementi matrice su Fibonačijevi brojevi)

\* Za efikasno stepenovanje matrice G koristićemo postupak uzastopnog kvadriranja, pa treba uzračunati vrednosti elemenata donje vrste kvadrata matrice *Gk*, odnosno matrice *GkGk*

\* Pošto smo izrazili elemente matrice *Gk* preko *g*1 i *g*2, imamo:



\* Ako se uvede pomoćna promenljiva temp, tada se nove vrednosti *g*1 i *g*2 izračunavaju iz starih sledećim programskim fragmentom:

temp = g1\*g1;

g1 = 2\*g1\*g2 - temp;

g2 = g2\*g2 + temp;

\* Analogan postupak se primenjuje na izračunavanje matrice *P*, u kojoj se akumulira proizvod

\* Pamte se samo elementi donje vrste matrice *P*, označimo ih *p*1 i *p*2

\* Proizvod matrica *P* i *G* izračunava se na uobičajen način (vrsta puta kolona), s tim da se elementi prve vrste matrice *P* ne izračunavaju (pa ih obeležavamo sa \*)



\* Koristeći pomoćnu promenljivu temp, u kojoj se čuva *p*1*g*1, proizvod matrica *P* i *G* se izračunava sledećim programskim fragmentom:

temp = p1\*g1;

p1 = p1\*g2 + p2\*g1 - temp;

p2 = p2\*g2 + temp;

static int fibonacci4(int n) {

if (0 <= n) {

int p1 = 0, p2 = 1, g1 = 1, g2 = 2, temp;

boolean izbor = (n % 2 == 1);

n /= 2;

while (n > 0) {

if (n % 2 == 1) { // P = P \* G

temp = p1\*g1;

p1 = p1\*g2 + p2\*g1 - temp;

p2 = p2\*g2 + temp;

}

n /= 2;

if (n > 0) { // G = G \* G

temp = g1\*g1;

g1 = 2\*g1\*g2 - temp;

g2 = g2\*g2 + temp;

}

}

if (izbor) return p2; else return p1;

}

return -1;

}

***34. Apstraktni tipovi podataka, klasa Polinom, operacije nad polinomima, proceduralni pristup***

***Apstraktni tipovi podataka***

\* Jedna od moćnih i ključnih tehnika rešavanja problema u programiranju (i uopšte) je **apstrakcija**

\* Pod apstrakcijom podrazumevamo zanemarivanje nepotrebnih detalja i koncentrisanje na bitne elemente rešenja

\* Na taj način o rešenju problema razmišljamo posredstvom globalnih i opštih operacija, čije detalje za početak zanemarujemo

\* Posle, odnosno nezavisno od globalnog rešenja problema realizuju se i (prethodno “apstrahovane”) operacije

\* Apstrakciju je moguće uspešno primeniti i na strukture podataka u programiranju

\* Možemo reći da svaka struktura podataka ima dva aspekta:

-**spoljašnji** i

-**unutrašnji**

\* Spoljašnji aspekt su one karakteristike strukture podataka koje opisuju šta struktura predstavlja - šta su elementi strukture podataka i koje su operacije sa njima dozvoljene

\* Unutrašnji aspekt je način na koji su struktura podataka i operacije nad njom realizovane

\* Pošto spoljašnji aspekt strukture podataka ne zavisi od realizacije, on se često naziva i **apstrakcijom** strukture podataka, dok se unutrašnji aspekt često naziva **implementacijom** apstraktne strukture podataka

\* Tipovi podataka se karakterišu:

-skupom vrednosti i

-skupom operacija nad tim vrednostima

\* Ako apstraktnu strukturu podataka “snabdemo” skupom vrednosti koje ta struktura može da predstavi i skupom operacija nad tim vrednostima, dobijamo **apstraktni tip podataka**

\* Apstraktni tip podataka je konačan **skup vrednosti**, koji se naziva domen, i **skup operacija** i relacija definisanih nad tim skupom vrednosti

\* Pri tom operacije i relacije treba shvatiti u širem smislu - dopuštaju se operacije sa argumentima koji mogu biti i iz nekog drugog skupa, a ni rezultat operacije ne mora biti u domenu

\* Skup vrednosti mora biti konačan, jer će se implementacija apstraktnog tipa podataka izvršavati na računaru

\* Za rešavanje problema apstraktni tip podataka je važniji od njegove realizacije jer se rešenje problema formuliše kao algoritam u terminima apstraktnog tipa podataka (*šta* a ne *kako* raditi sa vrednostima, odnosno *kojim* operacijama, a ne *kako* su one realizovane)

\* Implementacija apstraktnog tipa podataka se može ostaviti za kasnije, a ako je apstraktni tip podataka već implementiran, tada se taj aspekt može trajno zanemariti

\* Programski jezik Java pomoću koncepta referencijalnih tipova, odnosno klasa, omogućava kreiranje korisničkih apstraktnih tipova podataka (u daljem tekstu samo: apstraktnih tipova podataka)

\* Za definisanje i implementaciju apstraktnih tipova podataka u Javi, glavni mehanizam su klase

\* U klasama se definiše struktura podataka koju apstraktni tip ima, kao i operacije realizovane metodima

\* Demonstriračemo dva pristupa ogranizaciji i implementaciji apstraktnog tipa podataka:

-Proceduralni

-Objektno-orijentisani (OO)

\* Kod proceduralnog pristupa struktura podataka koja prestavlja tip je “odvojena” od operacija, koje u Javi implementiramo statičkim metodima

-Metodi su “nezavisni” od objekata na koje se primenjuju, koji se metodima prosleđuju preko parametara

\* Kod OO pristupa su podaci i operacije “spakovani” u jednu strukturu (klasu) kao polja i nestatički metodi

-Metodi se odnose na objekat preko kog se pozivaju

\* Akcenat stavljamo na proceduralni pristup

\* Napomenimo ovde da nam apstraktni tipovi podataka omogućavaju da prevaziđemo i jedno ograničenje programskog jezika vezano za tipove podataka

\* Ugrađeni prosti tipovi podataka imaju unapred definisani skup vrednosti i unapred definisani skup operacija nad tim vrednostima

\* Tipovi koje smo do sad uglavnom uvodili sadržali su samo podatke, čime smo im zadavali skup mogućih vrednosti, ali ne i skup operacija

\* Najzad, apstraktni tipovi podataka omogućavaju potpunu slobodu: programer može da odabere i skup vrednosti i skup operacija nad tim vrednostima - pod uslovom da ih sam i realizuje

**Polinom - statička implementacija**

\* **Polinom** *p*(*x*) je funkcija po promenljivoj *x* definisana sledećim (matematičkim) izrazom:



za date:

-ceo broj *n* ≥ 0: **stepen** polinoma

-realni brojevi *a*0, *a*1, ..., *an*: **koeficijenti** polinoma

\* ako je *n* > 0, tada mora da važi *an* ≠ 0

\* Izraze *aixi*, za *i* ϵ {0, 1, ..., *n*} nazivamo **monomi** polinoma

\* Koeficijenti polinoma su obično brojevi. Mi ćemo raditi sa realnim koeficijentima, ali koeficijenti mogu da budu i celi brojevi, kompleksni brojevi, itd.

Jedan uobičajen skup osnovnih operacija sa polinomima je:

Anulirati polinom, tj. staviti da je polinom jednak nuli

Kopirati polinom

Naći stepen polinoma

Izračunati vrednost polinoma za neku vrednost promenljive *x*

Učitati polinom

Štampati polinom

Sabrati dva polinoma

Oduzeti dva polinoma

Pomnožiti polinom konstantom

Pomnožiti dva polinoma i

Podeliti polinom polinomom

\* Polinom ćemo predstaviti nizom koeficijenata tipa double, i to tako da je indeks elementa u nizu jednak indeksu koeficijenta u polinomu i jednak stepenu monoma čiji je to koeficijent

\* Koristićemo nizove fiksne veličine sa unapred zadatom maksimalnom vrednošću stepena (maxSt) za sve polinome. Za takve nizove (i uopšte strukture podataka) kažemo da su **statički**

-(Nisu dinamički jer im se veličina i struktura ne mogu menjati u toku izvršavanja programa)

-Zato kažemo da je ova implementacija polinoma statička

\* Ako se monom nekog stepena ne pojavluje u polinomu,odgovarajući koeficijent je 0

\* Pošto je za svaki polinom karakterističan stepen, odlučujemo se da ga u reprezentaciji polinoma posebno čuvamo

\* Takođe želimo da napravimo razliku između nula-polinoma i konstantnog polinoma. Razlikovaćemo ih dogovorom da će nula polinom imati stepen -1, a konstantan polinom različit od nule imaće stepen 0

\* Prema tome, pogodna struktura za predstavljanje polinoma je:

class Polinom {

static final int maxSt = 100;

double[] k = new double[maxSt+1];

int st = -1;

}

\* Polje maxSt je static final, što znači da predstavlja jedinstvenu konstantu za sve polinome

\* Niz koeficijenata k sadrži redom *a*0, *a*1, ..., *an*, gde 0 ≤ *n* ≤ maxSt

\* Polje st sadrži *n*, a na početku ga inicijalizujemo na -1 da bi objekat tipa Polinom predstavljao nula-polinom

-(elementi niza k se automatski inicijalizuju na 0.0)

\* Operacije nad polinomima realizovaćemo kao statičke metode u klasi PolinomN:

class PolinomN {

/\* Anulira polinom p \*/

static void anuliraj(Polinom p) { ... }

/\* Kreira i vraca kopiju polinoma p \*/

static Polinom kopiraj(Polinom p) { ... }

/\* Pronalazi stepen polinoma p i smesta ga u strukturu \*/

static void nadjiStepen(Polinom p) { ... }

/\* Izracunava vrednost polinoma p za dato x \*/

static double izracunaj(double x, Polinom p) { ... }

/\* Ucitava polinom \*/

static Polinom ucitaj() { ... }

/\* Stampa polinom p \*/

static void stampaj(Polinom p) { ... }

/\* Sabira polinome p1 i p2 vracajuci zbir \*/

static Polinom saberi(Polinom p1, Polinom p2) { ... }

/\* Oduzima polinom p2 od polinoma p1 vracajuci razliku \*/

static Polinom oduzmi(Polinom p1, Polinom p2) { ... }

/\* Mnozi broj c sa polinomom p vracajuci proizvod \*/

static Polinom brojPuta(Polinom p, double c) { ... }

/\* Mnozi polinom p1 sa p2 vracajuci proizvod \*/

static Polinom puta(Polinom p1, Polinom p2) { ... }

/\* Deli dva polinoma, vracajuci kolicnik i ostatak u nizu \*/

static Polinom[] deli(Polinom p1, Polinom p2) { ... }

}

***35. Klasa Polinom, anuliranje (anuliraj), kopiranje (kopiraj), nalaženje stepena polinoma (nadjiStepen), proceduralni pristup***

\* Pogodna struktura za predstavljanje polinoma je:

class Polinom {

static final int maxSt = 100;

double[] k = new double[maxSt+1];

int st = -1;

}

\* Polje maxSt je static final, što znači da predstavlja jedinstvenu konstantu za sve polinome

\* Niz koeficijenata k sadrži redom *a*0, *a*1, ..., *an*, gde 0 ≤ *n* ≤ maxSt

\* Polje st sadrži *n*, a na početku ga inicijalizujemo na -1 da bi objekat tipa Polinom predstavljao nula-polinom

-(elementi niza k se automatski inicijalizuju na 0.0)

/\* Anulira polinom p \*/

static void anuliraj(Polinom p) {

if (p != null) {

p.st = -1;

for (int i = 0; i <= p.maxSt; i++)

p.k[i] = 0.0;

}

}

/\* Kreira i vraca kopiju polinoma p \*/

static Polinom kopiraj(Polinom p) {

if (p == null)

return null;

Polinom q = new Polinom();

q.st = p.st;

for (int i = 0; i <= p.maxSt; i++)

q.k[i] = p.k[i];

return q;

}

/\* Pronalazi stepen polinoma p i smesta ga u strukturu \*/

static void nadjiStepen(Polinom p) {

if (p != null) {

final double eps = 1.0E-5;

p.st = p.maxSt;

while (p.st > -1 && (Math.abs(p.k[p.st]-0.0) < eps)) {

**/\* Postavlja stepen polinoma za prvi koeficijent**

**razlicit od 0, ili na -1 ako su svi koeficijenti**

**0. Pri poredjenju sa nulom pitamo se da li**

**je koeficijent dovoljno blizu nule**

**\*/**

p.st--;

}

}

}

***36. Klasa Polinom, izračunavanje vrednosti polinoma (izracunaj), učitavanje polinoma (ucitaj), množenje polinoma brojem (brojPuta), proceduralni pristup***

\* Pogodna struktura za predstavljanje polinoma je:

class Polinom {

static final int maxSt = 100;

double[] k = new double[maxSt+1];

int st = -1;

}

\* Polje maxSt je static final, što znači da predstavlja jedinstvenu konstantu za sve polinome

\* Niz koeficijenata k sadrži redom *a*0, *a*1, ..., *an*, gde 0 ≤ *n* ≤ maxSt

\* Polje st sadrži *n*, a na početku ga inicijalizujemo na -1 da bi objekat tipa Polinom predstavljao nula-polinom

-(elementi niza k se automatski inicijalizuju na 0.0)

\* Vrednost polinoma se najjednostavnije izračunava pomoću Hornerove šeme, tj. programiranjem sledećeg postupka:

/\* Izracunava vrednost polinoma p za dato x \*/

static double izracunaj(double x, Polinom p) {

if (p == null)

return Double.NaN;

double rezultat;

if (p.st == -1)

rezultat = 0.0;

else {

rezultat = p.k[p.st];

for (int i = p.st - 1; i >= 0; i--)

rezultat = rezultat \* x + p.k[i];

}

return rezultat;

}

\* Učitavanje polinoma svodi se na učitavanje stepena polinoma i koeficijenata svih monoma

\* Treba paziti da učitani stepen polinoma bude u dozvoljenim granicama

\* Takođe, ne treba dozvoliti da koeficijent monoma najvećeg stepena bude 0, osim ako je u pitanju nula-polinom

-Zato ćemo učitavanje koeficijenta monoma najvećeg stepena realizovati posebno

/\* Ucitava polinom \*/

static Polinom ucitaj() {

Polinom p = new Polinom();

int pom;

double koef;

do {

System.out.print("Stepen polinoma (>= 0, <= " + p.maxSt + "): ");

pom = Svetovid.in.readInt();

} while (pom < 0 || pom > p.maxSt);

p.st = pom;

do {

System.out.print("Koeficijent uz x^" + p.st + ": ");

koef = Svetovid.in.readDouble();

} while (p.st > 0 && koef == 0.0);

p.k[p.st] = koef;

if (p.st == 0 && p.k[p.st] == 0.0) {

p.st = -1;

}

else {

for (int i = p.st - 1; i >= 0; i--) {

System.out.print("Koeficijent uz x^" + i + ": ");

koef = Svetovid.in.readDouble();

p.k[i] = koef;

}

}

return p;

}

/\* Mnozi broj c sa polinomom p vracajuci proizvod \*/

static Polinom brojPuta(Polinom p, double c) {

if (p == null)

return null;

Polinom rezultat = new Polinom();

if (p.st != -1 && c != 0.0) {

rezultat.st = p.st;

for (int i = 0; i <= p.st; i++)

rezultat.k[i] = c \* p.k[i];

}

return rezultat;

}

***37. Klasa Polinom, štampanje polinoma (stampaj), sabiranje i oduzimanje polinoma (saberi, oduzmi) , proceduralni pristup***

\* Pogodna struktura za predstavljanje polinoma je:

class Polinom {

static final int maxSt = 100;

double[] k = new double[maxSt+1];

int st = -1;

}

\* Polje maxSt je static final, što znači da predstavlja jedinstvenu konstantu za sve polinome

\* Niz koeficijenata k sadrži redom *a*0, *a*1, ..., *an*, gde 0 ≤ *n* ≤ maxSt

\* Polje st sadrži *n*, a na početku ga inicijalizujemo na -1 da bi objekat tipa Polinom predstavljao nula-polinom

-(elementi niza k se automatski inicijalizuju na 0.0)

\* Polinom se može štampati po opadajućim ili rastućim stepenima monoma

\* Sledeći metod štampa polinom po opadajućim stepenima

\* Ako je koeficijent monoma 0, taj monom ne treba štampati

\* Metod je najzgodnije podeliti na tri dela:

-najpre štampati **vodeći monom**, jer se ne štampa vodeći predznak +,

-zatim štampati sve **“unutrašnje” monome**, tj. monome u kojima treba štampati *x* (pri tom kod *x*1 ne štampati 1),

-na kraju štampati **slobodan član** (ne štampati *x*).

/\* Stampa polinom p \*/

static void stampaj(Polinom p) {

if (p != null) {

if (p.st > 0) { /\* ako polinom ima x stampati: \*/

if (p.k[p.st] < 0.0) /\* najpre vodeci monom \*/

System.out.print("-");

if (Math.abs(p.k[p.st]) != 1.0)

System.out.print(Math.abs(p.k[p.st]));

if (p.st > 1)

System.out.print("x^" + p.st);

else

System.out.print("x");

/\* stampati ostale monome sa x \*/

for (int i = p.st - 1; i >= 1; i--) {

if (p.k[i] != 0.0) {

if (p.k[i] > 0.0)

System.out.print("+");

else

System.out.print("-");

if (Math.abs(p.k[i]) != 1.0)

System.out.print(Math.abs(p.k[i]));

if (i > 1)

System.out.print("x^" + i);

else if (i == 1)

System.out.print("x");

}

}

/\* na kraju stampati slobodan clan \*/

if (p.k[0] != 0.0) {

if (p.k[0] > 0.0)

System.out.print("+" + p.k[0]);

else

System.out.print("-" + Math.abs(p.k[0]));

}

}

else { /\* ako polinom nema x \*/

if (p.k[0] < 0.0)

System.out.print("-" + Math.abs(p.k[0]));

else

System.out.print(p.k[0]);

}

}

}

\* Sabiranje i oduzimanje polinoma može da se realizuje jednim istim metodom, jer je postupak vrlo sličan: pri sabiranju polinoma koeficijenti uz monome istog stepena se sabiraju, a kod oduzimanja oduzimaju

\* Jedino na šta treba paziti je da na kraju treba odrediti stepen rezultata, jer je moguće da je stepen rezultata manji od stepena sabiraka

\* Uvedimo dodatan parametar op (operacija) koji može biti PLUS ili MINUS, što su konstante nabrojivog tipa Znak:

enum Znak {

PLUS, MINUS

}

private static Polinom sab(Polinom p1, Polinom p2, Znak op) {

if (p1 == null || p2 == null)

return null;

Polinom zbir = new Polinom();

if (p1.st > p2.st)

zbir.st = p1.st;

else

zbir.st = p2.st;

if (op == Znak.PLUS)

for (int i = 0; i <= zbir.st; i++)

zbir.k[i] = p1.k[i] + p2.k[i];

else

for (int i = 0; i <= zbir.st; i++)

zbir.k[i] = p1.k[i] - p2.k[i];

nadjiStepen(zbir);

return zbir;

}

/\* Sabira polinome p1 i p2 vracajuci zbir \*/

static Polinom saberi(Polinom p1, Polinom p2) {

return sab(p1, p2, Znak.PLUS);

}

/\* Oduzima polinom p2 od polinoma p1 vracajuci razliku \*/

static Polinom oduzmi(Polinom p1, Polinom p2) {

return sab(p1, p2, Znak.MINUS);

}

***38. Klasa Polinom, množenje polinoma (puta), proceduralni pristup***

\* Pogodna struktura za predstavljanje polinoma je:

class Polinom {

static final int maxSt = 100;

double[] k = new double[maxSt+1];

int st = -1;

}

\* Polje maxSt je static final, što znači da predstavlja jedinstvenu konstantu za sve polinome

\* Niz koeficijenata k sadrži redom *a*0, *a*1, ..., *an*, gde 0 ≤ *n* ≤ maxSt

\* Polje st sadrži *n*, a na početku ga inicijalizujemo na -1 da bi objekat tipa Polinom predstavljao nula-polinom

-(elementi niza k se automatski inicijalizuju na 0.0)

\* Množenje se može realizovati na sličan način, samo što se sada množi svaki monom jednog polinoma sa svakim monomom drugog polinoma

\* Treba množiti polinome ako su oba činioca različita od nule, a ako je bar jedan činilac nula-polinom, tada je množenje završeno

\* Ukoliko je stepen proizvoda veći od maksimalnog dozvoljenog stepena, metod vraća null

\* Množenje dva monoma se realizuje tako što se proizvod koeficijenata ta dva monoma doda na koeficijent monoma u proizvodu čiji je stepen jednak zbiru stepena monoma koji se množe

/\* Mnozi polinom p1 sa p2 vracajuci proizvod \*/

static Polinom puta(Polinom p1, Polinom p2) {

if (p1 == null || p2 == null)

return null;

int proizvodSt = p1.st + p2.st;

if (proizvodSt > Polinom.maxSt)

return null;

Polinom proizvod = new Polinom();

if (p1.st != -1 && p2.st != -1) {

proizvod.st = proizvodSt;

for (int i = 0; i <= p1.st; i++)

for (int j = 0; j <= p2.st; j++)

proizvod.k[i+j] += p1.k[i] \* p2.k[j];

}

return proizvod;

}

***39. Klasa Polinom, deljenje polinoma (deli), proceduralni pristup***

\* Pogodna struktura za predstavljanje polinoma je:

class Polinom {

static final int maxSt = 100;

double[] k = new double[maxSt+1];

int st = -1;

}

\* Polje maxSt je static final, što znači da predstavlja jedinstvenu konstantu za sve polinome

\* Niz koeficijenata k sadrži redom *a*0, *a*1, ..., *an*, gde 0 ≤ *n* ≤ maxSt

\* Polje st sadrži *n*, a na početku ga inicijalizujemo na -1 da bi objekat tipa Polinom predstavljao nula-polinom

-(elementi niza k se automatski inicijalizuju na 0.0)

\* Deljenje se realizuje tako što se imitira poznati algoritam deljenja polinoma “peške”

\* Ako je delilac jednak nula-polinomu, metod će vratiti null i time signalizirati grešku

/\* Deli dva polinoma, vracajuci kolicnik i ostatak u nizu \*/

static Polinom[] deli(Polinom p1, Polinom p2) {

if (p1 == null || p2 == null)

return null;

if (p2.st == -1)

return null;

int i, j, m, l;

Polinom ostatak = kopiraj(p1);

Polinom kolicnik = new Polinom();

if (p1.st >= p2.st) {

kolicnik.st = p1.st - p2.st;

for (j = p1.st; j >= p2.st; j--) {

i = j - p2.st;

kolicnik.k[i] = ostatak.k[j] / p2.k[p2.st];

for (m = 0; m <= p2.st; m++) {

l = m + i;

ostatak.k[l] -= kolicnik.k[i] \* p2.k[m];

}

}

nadjiStepen(ostatak);

}

Polinom[] rezultat = {kolicnik, ostatak};

return rezultat;

}

***40. Klasa Polinom, anuliranje (anuliraj), kopiranje (kopiraj), nalaženje stepena polinoma (nadjiStepen), OO pristup***

\* Statička implementacija polinoma koju smo predstavili urađena je u proceduralnom stilu, gde su podaci (polja klase Polinom) odvojeni od operacija (metodi klase PolinomN)

\* Uz male modifikacije implementacija može da se organizuje u objektno-orijentisanom stilu:

-Metodi pripadaju klasi Polinom, i nisu statički

-Umesto da se svi polinomi prosleđuju metodima kao parametri, poziv metoda odnosi se na tekući objekat preko koga se metod poziva, tako da se (prvi) parametar metoda tipa Polinom izostavlja

\* Primeri: PolinomN.anuliraj(p) postaje p.anuliraj() r = PolinomN.saberi(p, q) postaje r = p.saberi(q)

\* Java prvenstveno podržava OO stil, tako da je u Javi ovaj pristup implementaciji prirodniji

class Polinom {

static final int maxSt = 100;

double[] k = new double[maxSt+1];

int st = -1;

/\* Anulira polinom \*/

void anuliraj() {

st = -1;

for (int i = 0; i <= maxSt; i++)

k[i] = 0.0;

}

/\* Kreira i vraca kopiju polinoma \*/

Polinom kopiraj() {

Polinom q = new Polinom();

q.st = st;

for (int i = 0; i <= maxSt; i++)

q.k[i] = k[i];

return q;

}

/\* Pronalazi stepen polinoma i smesta ga u strukturu \*/

void nadjiStepen() {

final double eps = 1.0E-5;

st = maxSt;

while (st > -1 && (Math.abs(k[st]-0.0) < eps)) {

st--;

}

}

}

***41. Klasa Polinom, izračunavanje vrednosti polinoma (izracunaj), učitavanje polinoma (ucitaj), množenje polinoma brojem (brojPuta), OO pristup***

\* Statička implementacija polinoma koju smo predstavili urađena je u proceduralnom stilu, gde su podaci (polja klase Polinom) odvojeni od operacija (metodi klase PolinomN)

\* Uz male modifikacije implementacija može da se organizuje u objektno-orijentisanom stilu:

-Metodi pripadaju klasi Polinom, i nisu statički

-Umesto da se svi polinomi prosleđuju metodima kao parametri, poziv metoda odnosi se na tekući objekat preko koga se metod poziva, tako da se (prvi) parametar metoda tipa Polinom izostavlja

\* Primeri: PolinomN.anuliraj(p) postaje p.anuliraj() r = PolinomN.saberi(p, q) postaje r = p.saberi(q)

\* Java prvenstveno podržava OO stil, tako da je u Javi ovaj pristup implementaciji prirodniji

class Polinom {

static final int maxSt = 100;

double[] k = new double[maxSt+1];

int st = -1;

/\* Anulira polinom \*/

void anuliraj() {

st = -1;

for (int i = 0; i <= maxSt; i++)

k[i] = 0.0;

}

/\* Izracunava vrednost polinoma za dato x \*/

double izracunaj(double x) {

double rezultat;

if (st == -1)

rezultat = 0.0;

else {

rezultat = k[st];

for (int i = st - 1; i >= 0; i--)

rezultat = rezultat \* x + k[i];

}

return rezultat;

}

\* Metod ucitaj ostavićemo da bude statički, jer prirodno omogućava kreiranje i inicijalizaciju polinoma u okviru jednog poziva metoda:

/\* Ucitava polinom (napomena: static metod)\*/

static Polinom ucitaj() { ... }

\* Učitavanje polinoma svodi se na učitavanje stepena polinoma i koeficijenata svih monoma

\* Treba paziti da učitani stepen polinoma bude u dozvoljenim granicama

\* Takođe, ne treba dozvoliti da koeficijent monoma najvećeg stepena bude 0, osim ako je u pitanju nula-polinom

-Zato ćemo učitavanje koeficijenta monoma najvećeg stepena realizovati posebno

/\* Ucitava polinom \*/

static Polinom ucitaj() {

Polinom p = new Polinom();

int pom;

double koef;

do {

System.out.print("Stepen polinoma (>= 0, <= " + p.maxSt + "): ");

pom = Svetovid.in.readInt();

} while (pom < 0 || pom > p.maxSt);

p.st = pom;

do {

System.out.print("Koeficijent uz x^" + p.st + ": ");

koef = Svetovid.in.readDouble();

} while (p.st > 0 && koef == 0.0);

p.k[p.st] = koef;

if (p.st == 0 && p.k[p.st] == 0.0) {

p.st = -1;

}

else {

for (int i = p.st - 1; i >= 0; i--) {

System.out.print("Koeficijent uz x^" + i + ": ");

koef = Svetovid.in.readDouble();

p.k[i] = koef;

}

}

return p;

}

/\* Mnozi broj c sa polinomom vracajuci proizvod \*/

Polinom brojPuta(double c) {

Polinom rezultat = new Polinom();

if (st != -1 && c != 0.0) {

rezultat.st = st;

for (int i = 0; i <= st; i++)

rezultat.k[i] = c \* k[i];

}

return rezultat;

}

***42. Klasa Polinom, štampanje polinoma (stampaj), sabiranje i oduzimanje polinoma (saberi, oduzmi) , OO pristup***

\* Statička implementacija polinoma koju smo predstavili urađena je u proceduralnom stilu, gde su podaci (polja klase Polinom) odvojeni od operacija (metodi klase PolinomN)

\* Uz male modifikacije implementacija može da se organizuje u objektno-orijentisanom stilu:

-Metodi pripadaju klasi Polinom, i nisu statički

-Umesto da se svi polinomi prosleđuju metodima kao parametri, poziv metoda odnosi se na tekući objekat preko koga se metod poziva, tako da se (prvi) parametar metoda tipa Polinom izostavlja

\* Primeri: PolinomN.anuliraj(p) postaje p.anuliraj() r = PolinomN.saberi(p, q) postaje r = p.saberi(q)

\* Java prvenstveno podržava OO stil, tako da je u Javi ovaj pristup implementaciji prirodniji

class Polinom {

static final int maxSt = 100;

double[] k = new double[maxSt+1];

int st = -1;

/\* Anulira polinom \*/

void anuliraj() {

st = -1;

for (int i = 0; i <= maxSt; i++)

k[i] = 0.0;

}

}

\* Štampanje polinoma realizovaćemo uz pomoć metoda toString nasleđenog iz klase Object

\* Iskoristićemo Javinu klasu StringBuilder, koja predstavlja stringove koji mogu da se modifikuju. Nama će biti potreban samo metod append koji spaja tekući string sa stringom prosleđenim kao parametar

\* Pored deklaracije i inicijalizacije objekta s klase StringBuilder, jedina razlika u odnosu na stari metod stampaj je da umesto System.out.print pišemo s.append, i na kraju vratimo s.toString()

public String toString() {

StringBuilder s = new StringBuilder();

if (st > 0) { /\* ako polinom ima x stampati: \*/

if (k[st] < 0.0) /\* najpre vodeci monom \*/

s.append("-");

if (Math.abs(k[st]) != 1.0)

s.append(Math.abs(k[st]));

if (st > 1)

s.append("x^" + st);

else

s.append("x");

/\* stampati ostale monome sa x \*/

for (int i = st - 1; i >= 1; i--) {

if (k[i] != 0.0) {

if (k[i] > 0.0)

s.append("+");

else

s.append("-");

if (Math.abs(k[i]) != 1.0)

s.append(Math.abs(k[i]));

if (i > 1)

s.append("x^" + i);

else if (i == 1)

s.append("x");

}

/\* na kraju stampati slobodan clan \*/

if (k[0] != 0.0) {

if (k[0] > 0.0)

s.append("+" + k[0]);

else

s.append("-" + Math.abs(k[0]));

}

}

else { /\* ako polinom nema x \*/

if (k[0] < 0.0)

s.append("-" + Math.abs(k[0]));

else

s.append(k[0]);

}

return s.toString();

}

\* Sad kad imamo metod toString, implementacija metoda stampaj je trivijalna

\* Prosleđivanjem this kao parametra metodi System.out.print implicitno će biti pozvan metod this.toString() tekućeg objekta, i rezultat će biti odštampan

/\* Stampa polinom \*/

void stampaj() {

System.out.print(this);

}

\* Nabrojivi tip Znak i pomoćni metod sab ostaju isti, samo ćemo nabrojivi tip staviti da bude privatan član klase Polinom:

private enum Znak {

PLUS, MINUS

}

/\* Sabira polinom sa polinomom p vracajuci zbir \*/

Polinom saberi(Polinom p) {

return sab(this, p, Znak.PLUS);

}

/\* Oduzima polinom p od polinoma vracajuci razliku \*/

Polinom oduzmi(Polinom p) {

return sab(this, p, Znak.MINUS);

}

private static Polinom sab(Polinom p1, Polinom p2, Znak op) {

if (p1 == null || p2 == null)

return null;

Polinom zbir = new Polinom();

if (p1.st > p2.st)

zbir.st = p1.st;

else

zbir.st = p2.st;

if (op == Znak.PLUS)

for (int i = 0; i <= zbir.st; i++)

zbir.k[i] = p1.k[i] + p2.k[i];

else

for (int i = 0; i <= zbir.st; i++)

zbir.k[i] = p1.k[i] - p2.k[i];

nadjiStepen(zbir);

return zbir;

}

***43. Klasa Polinom, množenje polinoma (puta), OO pristup***

\* Statička implementacija polinoma koju smo predstavili urađena je u proceduralnom stilu, gde su podaci (polja klase Polinom) odvojeni od operacija (metodi klase PolinomN)

\* Uz male modifikacije implementacija može da se organizuje u objektno-orijentisanom stilu:

-Metodi pripadaju klasi Polinom, i nisu statički

-Umesto da se svi polinomi prosleđuju metodima kao parametri, poziv metoda odnosi se na tekući objekat preko koga se metod poziva, tako da se (prvi) parametar metoda tipa Polinom izostavlja

\* Primeri: PolinomN.anuliraj(p) postaje p.anuliraj() r = PolinomN.saberi(p, q) postaje r = p.saberi(q)

\* Java prvenstveno podržava OO stil, tako da je u Javi ovaj pristup implementaciji prirodniji

class Polinom {

static final int maxSt = 100;

double[] k = new double[maxSt+1];

int st = -1;

/\* Anulira polinom \*/

void anuliraj() {

st = -1;

for (int i = 0; i <= maxSt; i++)

k[i] = 0.0;

}

}

/\* Mnozi polinom polinomom p vracajuci proizvod \*/

Polinom puta(Polinom p) {

if (p == null)

return null;

int proizvodSt = st + p.st;

if (proizvodSt > maxSt)

return null;

Polinom proizvod = new Polinom();

if (st != -1 && p.st != -1) {

proizvod.st = proizvodSt;

for (int i = 0; i <= st; i++)

for (int j = 0; j <= p.st; j++)

proizvod.k[i+j] += k[i] \* p.k[j];

}

return proizvod;

}

***44. Klasa Polinom, deljenje polinoma (deli), OO pristup***

\* Statička implementacija polinoma koju smo predstavili urađena je u proceduralnom stilu, gde su podaci (polja klase Polinom) odvojeni od operacija (metodi klase PolinomN)

\* Uz male modifikacije implementacija može da se organizuje u objektno-orijentisanom stilu:

-Metodi pripadaju klasi Polinom, i nisu statički

-Umesto da se svi polinomi prosleđuju metodima kao parametri, poziv metoda odnosi se na tekući objekat preko koga se metod poziva, tako da se (prvi) parametar metoda tipa Polinom izostavlja

\* Primeri: PolinomN.anuliraj(p) postaje p.anuliraj() r = PolinomN.saberi(p, q) postaje r = p.saberi(q)

\* Java prvenstveno podržava OO stil, tako da je u Javi ovaj pristup implementaciji prirodniji

class Polinom {

static final int maxSt = 100;

double[] k = new double[maxSt+1];

int st = -1;

/\* Anulira polinom \*/

void anuliraj() {

st = -1;

for (int i = 0; i <= maxSt; i++)

k[i] = 0.0;

}

}

/\* Deli polinom polinomom p, vracajuci kolicnik i ostatak u nizu \*/

Polinom[] deli(Polinom p) {

if (p == null)

return null;

if (p.st == -1)

return null;

int i, j, m, l;

Polinom ostatak = kopiraj();

Polinom kolicnik = new Polinom();

if (st >= p.st) {

kolicnik.st = st - p.st;

for (j = st; j >= p.st; j--) {

i = j - p.st;

kolicnik.k[i] = ostatak.k[j] / p.k[p.st];

for (m = 0; m <= p.st; m++) {

l = m + i;

ostatak.k[l] -= kolicnik.k[i] \* p.k[m];

}

}

ostatak.nadjiStepen();

}

Polinom[] rezultat = {kolicnik, ostatak};

return rezultat;

} // Kraj klase Polinom