OBJEKTNO ORIJENTISANO PROGRAMIRANJE-USMENI ISPIT

I Objektno orijentisana paradigma

1. Osnovni pojmovi objektno-orijentisanog programiranja (objekat, klasa, ucaurenje, nasledjivanje). Primeri

-Objektno-orijentisano programiranje i objektno orijentisani jezici su sustinski zasnovani na modulima, tj na skupu objekata koji imaju medjusobnu interakciju.

-U OOP, osnovni koncept je objekat. Njihova namena je da predstavljaju stvari i pojmove iz realnog sveta(zivotinje, studenti, automobili, ...).Svi objekti imaju dve zajednicke karakterstike:stanje i ponasanje.Stanja se predstavljaju poljima objekta, a ponasanja implementiraju metode. Vazno je identifikovati stanja i ponasanja objekta iz realnog sveta.Objekti variraju po slozenosti(siromasnija ili brojnija stanja i/ili ponasanja).Objekti u sebi mogu sadrzati druge objekte.Metodi izvrsavaju aktivnosti nad stanjem objekta predtavljenim preko polja i osnovni su mehanizam komuniciranja izmedju objekata.

Metod je funkcija koja je sastavni deo objekta i opisuje njegovo ponasanje.

-Vazan koncept OOP je "skrivanje podataka"(enkapsulacija, ucaurenje), gde se stanja objekta skrivaju u smislu da im se ne dopusta direktan pristup i menjanje vrednosti, vec se svaki vid interakcije objekta i promene stanja objekta odvija preko metoda.Sa odgovarajucim vrednostima stanja i metodima za pristup i promenu tekuceg stanja, uspostavljena je kompletna kontrola i nacin pristupa i menjanja stanja objekta kao reakcije na poruke iz okruzenja.Pakovanje koda zajedno sa stanjem u jedinstvenu celinu, pruza citav niz prednosti:

a)Modularnost: Izvorni kod razlicitih objekata moze da se pise medjusobno nezavisno.

b)Sakrivanje informacija

v)Ponovno koriscenje koda

g)Jednostavna promena

-Klase:Svaki objekat je napravljen iz istog modela/sheme/prototipa i sadrzi iste komponente/osobine.Realni svet sadrzi mnogo pojedinacnih objekata za koje se moze reci da su iste vrste.Dakle, klasa je model/shema/plan/prototip od koje se prave pojedinacni, konkretni objekti.Kreiranje i koriscenje objekata nastalih od klasa je obično posao nekih drugih klasa u okviru vece softverske aplikacije.

-Instanca(primerak klase) je objekat neke klase

Primer:

Class Student koji u sebi ima (ime,rollNo,setName(),setRollNo()) je klasa a Jenna R005 je objekat te klase, za svaki objekat primerak kao sto se vidi, instanca promenljiva ima konkretnu vrednost

-Nasledjivanje: Razlicite vrste objekata pored ociglednih razlika cesto imaju neke zajednicke karakteristike koje su bitne i mogu biti koriscene u procesu dizajna programa. Jedan od osnovnih koncepata OOP je da klasa moze da nasledi stanje i ponasanje neke druge klase. U Javi, svaka klasa moze da ima proizvoljan broj podklasa, kao i jednu direktnu nadklasu. Ovakvo nasledjivanje se naziva i jednostruko nasledjivanje. Nasledjivanje dozvoljava definisanje nove klase na osnovu vec postojece klase. Klasa koja se nasledjuje se naziva nadklasa, a klasa koja nasledjuje se naziva podklasa.

Nasledjivanje opisuje odnose: 1)jeste konkretizacija, 2)jeste uopstenje

Sadrzavanje opisuje odnose:1)je deo od, 2)sadrzi

Primer:

Ovde je zivo bice uopstenje zivotinje, a zivotinja je konkretizacija zivog bica

Nasledjivanje pojednostavljuje proces modeliranja hijerarhije objekata iz realnog sveta:

a)opsta svojstva i funkcionalnosti se implementiraju u osnovnoj klasi i time se omogucava bolje ponovno koriscenje koda.

b)podklase onda prosiruju, specijalizuju ili menjaju funkcionalnosti nadklase.

Primer:

Sintaksa u Javi za kreiranje podklase:

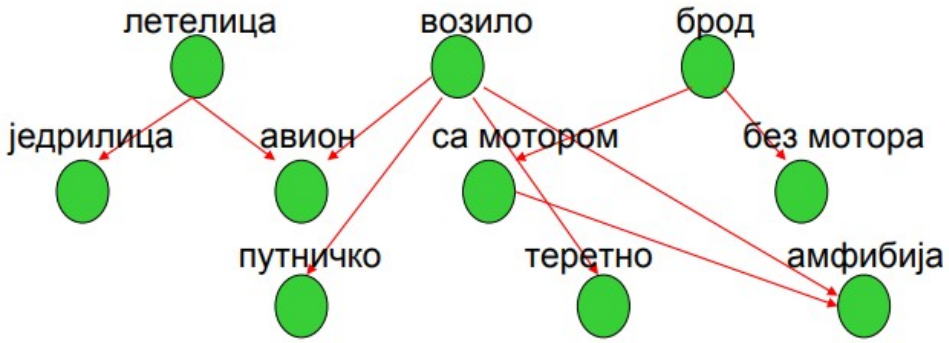
```
class ImePodklase extends ImeNadklase{
    ....
}
```

Kompozicije i agregacije omogucavaju konstruisanje kompleksnih klasa koje inkorporiaju druge objekte.

2. Napredni pojmovi OOP(visestruko nasledjivanje, apstraktne klase, intefejsi)

-Visestruko nasledjivanje je mehanizam koji omogucava da jedna klasa A moze imati vise nadklasa, odnosno moze da nasledjuje vise nadklasa B1, B2, B3, ..., Bn. Na taj nacin podklasa prakticno spaja svojstva vise nadklasa u jednu celinu. Ovakvo nasledjivanje moze dovesti do konflikta imena u klasi A, ako jedna ili vise nadklasa ima svojstva sa istim imenom. Java ne podrzava visesturko nasledjivanje.

Primer:



Konflikt imena se razresava na sledece nacine:

a)redosled u kom su podklase navedene u toku nasledjivnja definise koje je svojstvo vazece i koje ce biti izabrano, a ostala svojstva sa istim imenom u narednim klasama ostaju skrivena.

b)podklasa razresava konflikt tako sto odredjuje iz koje klase se koristi svojstvo sa tim imenom.

Specijalan konfilkt imena se javlja ako klasa D nasledjuje klase B i C koje nasledjuju klasu A(problem dijamanta).Neki progrmski jezici ovaj problem razresavaju tako da klasa D nasledjuje:

*klasu A

*Svojstva klase B i svojstva klase C BEZ svojstava koje su nasledile iz klase B

-Interfejsi: kad se posmatraju samo definicije metoda, moze se reci da metodi formiraju interfejs prema spoljnom okruzenju, tako sto specifiraju kako ce objekat i spoljni svet medjusobno komuicirati.U Javi, interfejs se definise grupa metoda sa praznim telom, odnosno bez koda.Pre nego sto se interfejs koristi, mora se inmplementirati:

```
class ImeKlase implements ImeInterfejsa{

    ...

}
```

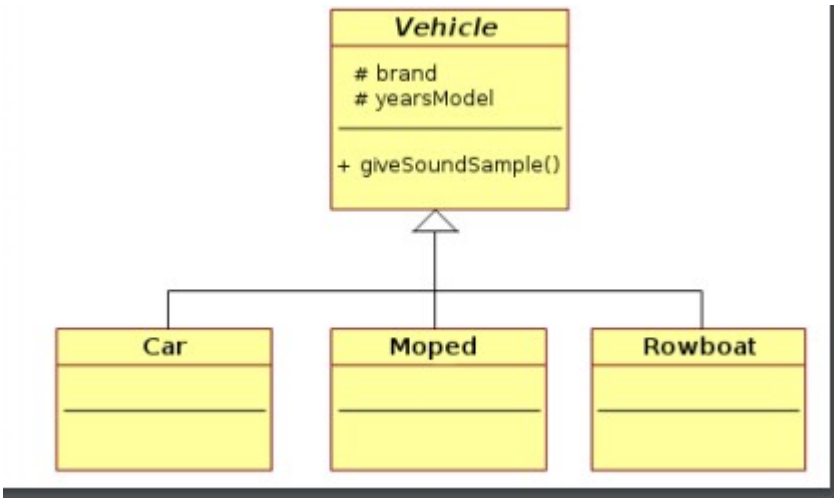
Da bi klasa koja implementira bila funkcionalna, mora implementirati sve metode iz intefejsa.

-Apstraktne klase:Apstraktna klasa je klasa u kojoj postoji bar jedan metod koji nema telo vec samo njegovu deklaraciju.Takav metod se naziva apstraktan metod.Apstraktna klasa se ne moze instancirati.Potklasa apstraktne klase moze realizovati sve apstraktne metode iz nadklase, a ako ih ne implementira onda i sama postaje apstraktna.Java programer koristi apstraktne klase u situaciji kada zeli da obaveze da sve podklase moraju da realizuju apstraktne metode.

Primer:

Gledamo na dijagram dole, metod pomeriSe, znamo da bi zivotinja trebala da moze da se pomera, ali se ne pomeraju sve zivotinje na isti nacin, koristi se kada programer zeli da obaveze da sve podklase date klase realizuju neko ponasanje, to ponasanje se definise u podklasi, dok se u apstraktnoj klasi samo deklarise.

Mogao bi i primer za oblike, da se povrsina kruga, kvadrata i trougla ne racuna na isti nacin



3. Prednosti i mane OOP

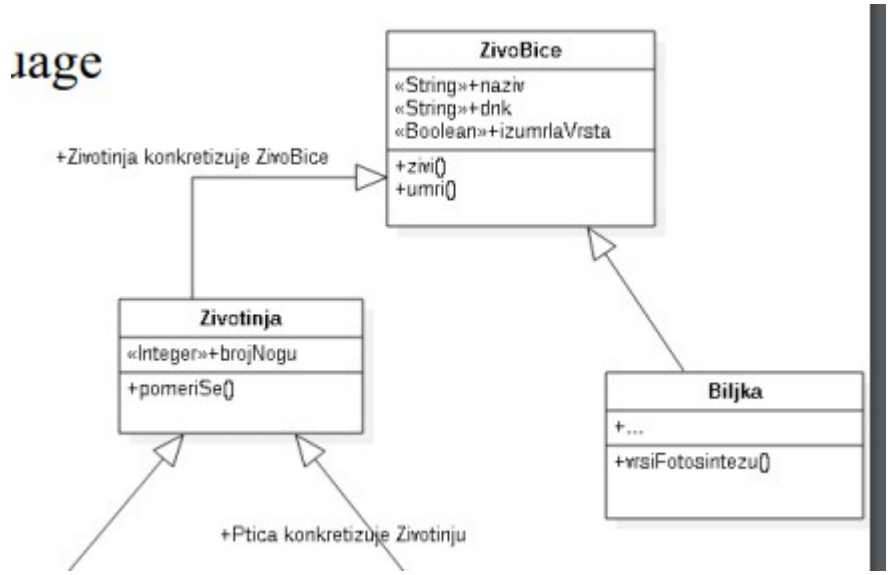
-Prednosti: pogodan je za analizu, projektovanje i programiranje. Olaksano odrzavanje softvera, omogucava lako i jednostavno uklapanje modula, ponovna iskoristivost softvera, najpogodniji za simuliranje dogadjaja, ...

-Mane: veca velicina programa, sporiji programi, nije namenjen za sve vrste problema, treba dosta vremena da se nauci

4. UML dijagrami klasa. Primeri

Uml je standardizovani format za modeliranje softvera koriscenjem grafickih oblika i teksta.

Primer:



II Karakteristike programskog jezika Java

1. Nastanak i razvoj programskog jezika Java

-1991.Preteca Jave namenjena mreznom kucnom okruzenju

1994.Usmerenje prema internetu

1995.Jezik Java lansiran na SunWorld konferneciju(IBM dobija licencu, Javu pocinje da koristi i microsoft)

1996.Sun razvija JDK 1.0(kodno ime Oak)

1997.Pojavljuje se JDK 1.1

1997.Druga konferencja o javi sa 10000 ucesnika

1998.Krajem godine novi JDK oznacen sa J2SE 1.2(Swing graficka biblioteka, kolekcije)

2000.Pojavljuje se J2SE 1.3(Rad sa zvukom, rad u distribuiranom okruzenju)

2002.Pojavljuje se J2SE 1.4(Ulancavanje izuzetaka, biblioteke za logovanje gresaka, manipulacija slikama)

2004.Pojavljuje se J2SE 5.0(1.5)(Genericki tipovi, anotacije, paralelno programiranje)

2006.Sun objavljuje Java SE(ubrzavanje performansi jezgra i rada sa grafikom, poboljsavani veb servisi, poboljsan rad pri povezivanju sa bazama podataka, poboljsanja na java virtuelnoj masini)

2006.Veliki deo Jave postaje slobodan i otvoren kod pod GPL licencom

2010.Java postaje vlasnistvo kompanije Oracle

2011.Pojavljuje se Java SE 7(JVM podrska z adinamicke jezike, ubrzavanje iscrtavanja sa modernim grafickim karticama, jednostavnije deklarisanje metoda sa promenljivim brojem parametara, simultano hvatanje vise vrsta izuzetaka)

2014.Pojavljuje se Java SE 8(Rad sa lambda izrazima, Intuitivniji rad sa vremenom i kalendarima, poboljsan modul za rad sa JavaScript bibliotekom)

There were five primary goals in the creation of the Java language:

1. It should use the object-oriented programming methodology.
2. It should allow the same program to be executed on multiple operating systems.
3. It should contain built-in support for using computer networks.
4. It should be designed to execute code from remote sources securely.
5. It should be easy to use by selecting what was considered the good parts of other object-oriented languages.

2. Karakteristike programskog jezika Java

-Java je od svog nastanka pa do danas svrstavana u najperspektivnije i najcesce koriscene jezike. Njene glavne osobine su:

*Jednostavnost-jednostavna sintaksa koja lici na C i C++ sintaksu, svi koncepti su lako shvatiljivi i nisu kompleksni kao u C i C++ jeziku, sigurnije jezicke konstrukcije u odnosu na C i C++

*Platformska nezavisnost-racunarsku platformu cine tip racunara i OS. Java ne zavisi od racunarskih platformi, odnosno da ako Java program radi na jednoj platformi, radice na svakoj

*Objektna orijentisanost-Java je OO jezik.

*Prenosivost-zbog nezavisnosti od platforme, Java program se lako prenosi sa platforme na platformu

*Distribuiranost-Java je pogodna za mrezno programiranje i pravljenje distribuiranh apliakcija. Distribuirane aplikacije se sastoje od vise programa koji se izvrsavaju na vise razlicitih racunara koji su spojeni racunarskom mrezom

*Interpretacija-Izvorni kod se ne prevodi direktno u masinski kod racunara, vec se prevodi u platformski nezavistan kod(Java bajt kod).Java bajt kod se interpretira na Java virtuelnoj masini

*Niti-Java podrzava niti

*Efikasnost-Java je jako efikasan jezik, u poredjenju sa ostalim jezicima koji se interpretiraju. Programski jezici ciji se kod ne prevodi u masinski kod racunara su sporiji, pa je Java sporija od npr. C++.

*Robusnost-Vecina gresaka u Java programu se otkrije jos u fazi prevodjenja, Java ne dozvoljava programeru primenu opasnih operacija, poput pokazivacke aritmetike u C++.Java programer ne pristupa direktno memoriji.

*Dinamisko vezivanje-Java dozvoljava nasledjivanje klasa

*Sigurnost-Ako programi salju jedan drugom poverljive informacije one moraju biti zasticene, pa Java nudi efikasne sigurnosne mehanizme.

3. Mogucnosti jezika Java-odgovori dati u prethodnom pitanju

4. Tipovi Java aplikacija

-Aplikacije sa grafickim korisnickim interfejsom-Java moze biti koriscena za razvoj portabilnih GUI apliakcija na svim podrzanim platformama, postoji veci broj biblioteka koje olaksavaju programiranje GUI u Javi(AWT, SWT, Java FX).

-Aplikacije iz komandne linije-ove aplikacije ne koriste graficke komponente, unos i ispis se vrse preko komandne linije umesto putem tekstualnih polja, labela, ...

-Aplikacije za mobilne uredjaje-koriste funkcionalanost biblioteka za programiranje mobilnih uredjaja koje se isporucuju uz Javu, koristi se Java FX, cilj je da se razvoj aplikacija za sve uredjaje uopsti, pored toga postoje i Java alati za razvoj android Aplikacija(Android SDK), Java FX podrzava kako Java ME apliakcije, tako i standardni Java API

-Apleti-veb programiranje na klijentnoj strani-

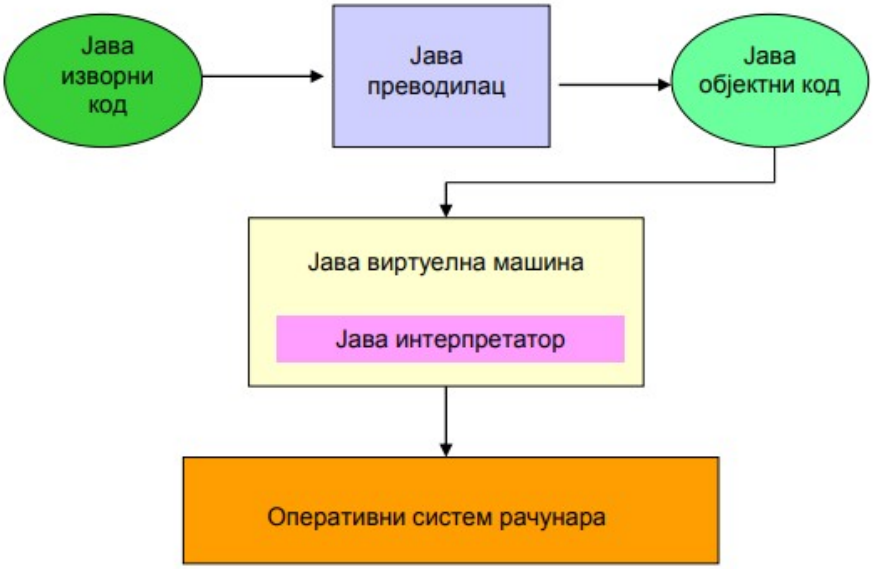
-Servleti i Java serverske strane(veb programiranje na serverskoj strani)-Na veb serveru se izvrsavaju naredbe, potom veb server napravi HTML datoteku i posalje je klijentu(ova vrsta programiranja se naziva programiranje na serverskoj strani), ovakvi programi se jos nazivaju viseslojne aplikacije sa mrsavim klijetnom

-Veb servisi-Java omogucuje kreiranje i mdoifikaciju veb servisa, komunikacija se vrse preko HTTP protokola, veb servisi omogucavaju komunikaciju izmedju aplikaicja koje se izvrsavaju na razlicitim platofrmama, opisi veb servisa su zadati najcesce koa XML datoteka sto omogucava njihovu prosirivost i dinamicnost, pozeljno je graditi hijerarhiju veb servisa u kojoj oni sofisticiraniji koriste usluge jednostavnijih

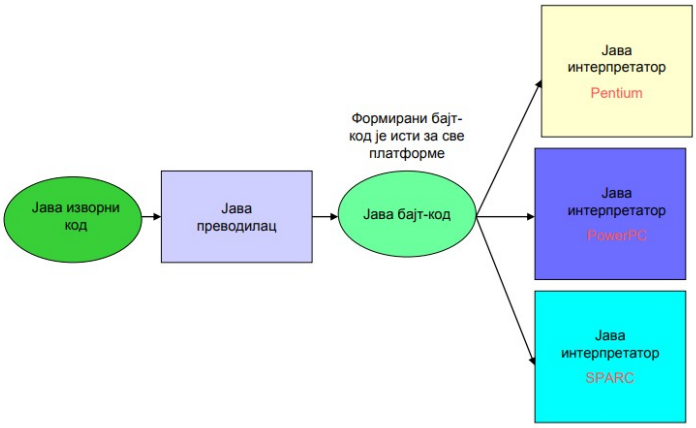
-Biblioteke klasa-Java omogucava da programer koristi klase iz JDKa kao i da oformi sopstvene klase, programer svoje klase moze spakovati u biblioteke klasa, u toj formi se one dalje koriste, biblioteka obicno sadrzi veci broj klasa i srodne klase se obicno grupisu u pakete

5. Proces izvrsavanja Java programa

-Java je jezik koji se prevodi i interpretira.Napisani izvorni kod Java programa se prvo prevede koriscenjem Java kompajlera javac u tzv. bajt-kod.Potom se prevedeni bajt-kod izvrsava uz pomoc Java interpretatora java.Kreirani bajt-kod je binaran i arhitektonski neutralan(platformski neutralan).Java okruzenje za izvrsavanje postoji posebno za svaku platofrmu i ono je nadlezno za prevodjenje bajt-koda do izvsrnog koda.Java izvorni kod i Java bajt-kod ostaje isti bez obzira na kojoj se platformi izvrsava.Koriscenjem Jave se postize da postoji jedinstveni izvorni Java kod, a da program radi na raazlicitim platformama.Kombinacija kompilacije i interpretacije je pozitivna iz nekoliko razloga:obezbedjuje sigurnost i stabilnost(java sadrzi element nazvan povezivac koji proverava podatke koji dolaze na JVM, da ne bi ostetile datoteke), ova kombinacija resava probleme neuklapanja verzija.



Креирање Јава извршног кода од изворног програма – превођење и интерпретација



Drugi jezici imaju samo izvorni kod, prevodilac, i izvrsni kod.

6. Prevodjenje i interpretacija Java programa, Java prevodilac

Java, being a platform independent programming language, doesn't work on one-step-compilation. Instead, it involves a two-step execution, first through an OS independent compiler; and second, in a virtual machine (JVM) which is custom-built for every operating system. The two principle stages are explained below:

Compilation:

First, the source '.java' file is passed through the compiler, which then encodes the source code into a machine independent encoding, known as Bytecode. The content of each class contained in the source file is stored in a separate '.class' file. While converting the source code into the bytecode, the compiler follows the following steps:

- Parse: Reads a set of *.java source files and maps the resulting token sequence into AST (Abstract Syntax Tree)-Nodes.
- Enter: Enters symbols for the definitions into the symbol table.
- Process annotations: If Requested, processes annotations found in the specified compilation units.

- Attribute: Attributes the Syntax trees. This step includes name resolution, type checking and constant folding.
- Flow: Performs dataflow analysis on the trees from the previous step. This includes checks for assignments and reachability.
- Desugar: Rewrites the AST and translates away some syntactic sugar.
- Generate: Generates '.Class' files.

-Veb strane sa apletima se obicno ucitavaju i prikazuju duze od klasicnih,javina prenosivost izaziva gubitak performansi,koji je smanjen koriscenjem JIT kompajlera jer on prevodi Java metode u masinski kod za konretnu patofrmu.(Nisam siguran da je ovo dovoljan odgovor).

7. Prevodjenje i interpretacija Java programa, JVM

-Jezgro Jave je JVM(Java virtuelna masina).JVM je virtualni racunar koji postoji samo u memoriji i dopusta da Java programi budu izvrsavani na raznovrsnim platformama.Da bi Java program mogao da radi na odredjenoj platofrmi, JVM mora da bude implementirana na toj platformi.JVM je veoma mala kada se implementira u RAM-u(tako mala velicina JVM dopusta da se Java koristi u raznovrsnim elektronskim uredjajima, ceo jezik Java je razvijan tako da se na umu ima i kucna elektronika).JVM izvrsava bajt-kod.Pre toga, program javac(Java prevodilac) procesir .java datoteke i rezultujuci bajt-kod cuva u datoteci sa ekstenzijom .class.JVM cita tok bajt-kodova iz .class datoteke kao sekvencu masinskih kodova.Izvrsavanje instrukcija bajt-koda oslikava izvrsavanje masinskih instrukcija.Svaka od instrukcija JVM je slicna asemblerskoj instrukciji.Prilikom svog rada, JVM ima interakciju sa:bibliotekama Java klasa, bibliotekama nativnih metoda, nativnim nitima, realnim operativnim sistemom racunara na kom se izvrsava JVM.JVM sadrzi: sistem za ucitavanje klasa, podsistem za izvrsavanje, oblast za podatke prilikom izvrsavanja, sakupljac otpadaka i Java niti.Oblast za podatke prilikom izvrsavanja se sastoji od:oblasti za Java metode, hip memorije, stek memorije, registara, stek za nativne metode.Memorija kod JVM: adresna duzina koda je 32 bita, pa JVM moze da adresira do 4GB, stek i prostor koji se reciklira i oblasti za metode su smesteni u tih 4GB.JVM koristi cetiri registra za upravljanje sistemskim stekom(brojac, vrh, okvir, promenljive).Na vrh steka pokazuje registr optop, registar okvir pokazuje gde se metod izvrsava u memoriji, a registar promenljivih pokazuje na prvu promenljivu na steku.

-Prostor(heap) je deo memorije iz koga se vrsi instanciranje i alokacija objekata.Svaki put kad se alocira memorija uz pomoc new, ta memorija dolazi iz prostora.Okruzenje u kom se Java izvrsava automatski oslobadja objekte koje niko ne refefrise, a ova operacija oslobadjanja prostora se zove skupljanje otpadaka.

8.Java alati za razvoj, klase, biblioteke klasa i paketi

+ **9. Java alati za razvoj, JDK i Java API-ji**

The Java Development Kit (JDK) is a software development environment used for developing Java applications and applets. It includes the Java Runtime Environment (JRE), an interpreter/loader (Java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc) and other tools needed in Java development.

-Razvoj velike popularnosti jezika Java je, pored kvaliteta njenog dizajna, postojanje bogatog skupa alata,kao i skup biblioteka koji je organizovan u pakete.Ovi oformljeni objekti omogucavaju brz start pri radu sa Javom iz dva razloga:programer ne mora da razvija njihovu funkcionalnost, izvorni kod je svima dostupan.

-Java Application Programming Interface(Java API) je skup klasa koji je razvio Sun, za koriscenje jezika Java.Java API je dizajniran da bi se pomoglo programeru u razvoju sopstvenih klasa, apleta i aplikacija. Klase unutar Java API-ja su grupisane u pakete, pri cemu svaki paket

moze sadrzati vise klasa i interfejsa. Nadalje, svaka od klasa moze imati vise osobina, vise polja i/ili metoda. Moguce je programirati u Javi i bez API-ja, ali treba naglasiti da svaka novorazvijena klasa zavisi od bar jedne klase API-ja. Nadalje, pri razvoju sve slozenijih programa, koji rade sa niskama, soketima i grafickim interfejsom, postaje znacajno poznavanje objekata koje je Sun obezbedio. Najznacajniji paketi koji dolaze sa Javom:

*Biblioteke za GUI kao sto su prozori, dijalozi, dugmici, tekstualna polja, ...

*Klase za mreze, URL-ove, klijent-serverske sokete

*Klase za razlicite tipove ulaza i izlaza

*Klase za razlicite tipove podataka, procese i niti koji se izvrsavaju, stringove, ...

*Pomocne klase za datume, kolekcije, ...

*Klase za razvoj apleta

*Klase za manipulaciju slikama

-Paketi predstavljaju nacin za organizovanje klasa i on moze sadrzati klase, kao i druge pakete .

10. Java alati za razvoj, izdanja Java okruzenja(Standard i Enterprise)

-Java platform, Standard Edition je osnovna platforma, sa velikim brojem paketa, klasa, interfejsa i nabrojivih tipova za razvoj razlicitih aplikacija, ukljucujuci vizuelne, mrezne i distribuirane aplikacije.

-Java platform, Enterprise edition prosiruje Javu SE tehnologijama i alatima neophodnim za razvoj velikih poslovnih(pre svega mreznih) aplikacija, koje mogu koristiti milioni klijenata.

11. Java alati za razvoj, centralni Java API-ji(Core)

-Centralni API sadrzi pakete sa objektima za koje se garantuje da su dostupni bez obzira na Java implementaciju:

*java.lang: sastoji se od klasa koje su centralne za jezik java, on obezbedjuje ne samo klase-omotace za proste tipove podataka, kao sto su Character i Integer, vec i obradu gresaka uz koriscenje klasa Throwable i Error, klase System i SecurityManager omogucuju programeru kontrolu nad Java sistemom za izvrsavanje

*java.io: standardni ulazno/izlazna Java biblioteka, omogucuje rad sa tokovima, omogucava rad sa jednostavnim tipovima(String) i sa slozenim tipovima(StreamTokenizer)

*java.util: sadrzi veci broj korisnih klasa koje nisu mogle biti uklopljene u druge pakete, klase za rad sa datumima, klase za strukturisanje podataka(Stack i Vector), klase za parsiranje ulaznog toka podataka

*java.net: ovaj paket cini jezik Java mrezno zasnovanim jezikom, on obezbedjuje sposobnost komunikacije sa udaljenim cvorovima, bilo preko soketa, bilo preko URL

*java.awt: Abstract Window Toolkit, ne sadrzi samo klase, vec i nekoliko konkretnih interaktivnih alata, kao sto su Button i TextField, klasa Graphics obezbedjuje bogatsvo grafickih mogucnosti, ukljucujuci i mogucnost crtnja raznih oblika mogucnost prikaza slika

*java.awt.image: blisko povezan sa paketom java.awt, sastoji se od alata za rukovanje i mainpulaciju slikama koje dolaze kroz mrezu

*java.awt.peer: sadrzi interfejse koji sluze kao posrednici izmedju Java koda i racunara na kom se taj kod izvrsava

*javax.swing: uveden kako bi se prevazisli problemi na koje nailaze programeri koji su koristili awt za kreiranje GUI aplikacija, awt je izgledao drugacije na razl platformama, oslanja se na awt i sadrzi klase kao sto su JButton iJTextField

*java.applet: najmanji paket u API-ju, ali je cesto koriscen zbog klase Applet koja je definisana u njemu, sadrzi mnogo korisnih metoda, informacije o apletovom okruzenju se mogu dobiti kroz interfejs Appletcontext

12. Java alati za razvoj, dodatni Java API-ji(Enterprise, Server, Beans, ...)

-Java Enterprise API: odrzava vezu prema velikim bazama podataka i prema nasledjenim aplikacijama, koriscenjem ovog API-ja razvijaju se slozeni distribuisani klijent/server apleti i aplikacije u Javi

-Java RMI omogucuje pozivanje udaljenih metoda izmedju cvorova ili klijenta i servera u slucajevima kada se na oba kraja poziva nalaze aplikacije pisane u jeziku Java

-Java Server API: omogucuje i olaksava razvoj Java-zasnovanih Internet i Internet servera. Ovaj API je neophodan programerima koji razvijaju java servlete

-Java Beans definise prenosiv skup API-ja za softverske komponente koji je nezavistan od platforme. Java Beans je skup konvencija koji omogucava uniforman prenos objekata i podataka izmedju razlicitih platofrmi

13. Java kao OO jezik

-Java je OO programski jezik. Java i C++ dele mnoge od principa na kojima se zasnivaju, a najveci broj razlika se odnosi na stil i strukturu. Java je dizajnirana tako da podrzava samo jendostruko nasledjivanje. U cilju odrzavanja jednostavnosti Jave, nisu bas svi elemetni realizovani kao objekti: Logicke velicine, brojevi i drugi prosti tipovi nisu realziovani kao objekti. Java je OO jezik sa hijerarhijskom strukturom klasa. Na vrhu hijerarhije se nalazi klasa Object(java.lang). Za svaku klasu su bitne dve karakteristike(stanje i ponasanje). Klase u javi su organizovane po paketima. Sve klase iz jednog ili vise paketa ciji je bajt kod spakovan zajedno cine biblioteku klasa. Java je striktno objektno orijentisana(ne dopusta da se deklarise bilo sta sto nije enkapsulirano u objekat, nema samostalnih procedura). Na nivou hardvera promenljiva koja predstavlja objekat u Javi ustvari sadrzi referencu na deo memorijskog prostora koji zauzima dati objekat.

14., 15., 16.

Примитивни и објектни типови података и њихово представљање

Однос између објекта и класе у Јави-i ovo je u prethodnim pitanjima

Хијерархија наслеђивања у Јави, класа Object-ovo je u prethodnim pitanjima

U javi postoje primitivni i objektni tipovi podataka, primitivni su oni koji nemaju komponente.

Kreirani podatak se naziva objekat, a objektima se pristupa preko referenci tako da se objektni tip naziva i referencni tip

```
<tip> :: = <primitivni tip>|<objektni tip>
```

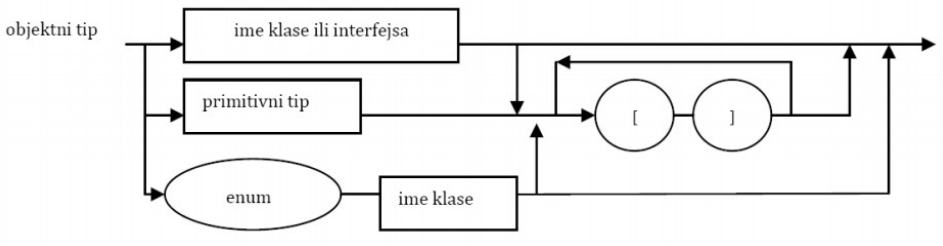
Primitivni tip je odredjen skupom vrednosti koje se formiraju iz odgovarajucih literala i skupom operacija nad tim vrednostima, to su unapred definisani tipovi u Javi(brojevi, znakovni, logicki)

```
<primitivni tip>::=<aritmetički tip> | boolean
<aritmetički tip> ::= <celobrojni tip>|<realni tip>
<celobrojni tip> ::= byte | short | int | long | char
<realni tip> ::= float | double
```

Byte masa = 5;

Ako je promenljiva primitivnog tipa ona ima lokaciju u kojoj je smestena njena vrednost(kontejnerski tip), u memoriji ce postojati lokacija kojoj je dodeljeno ime masa i ima vrednost 101(5 u binarnom)

Objekat u sebi objedinjuje skup podataka i operacija za rad sa njima, on moze biti korisnicki – definise ga korisnik preko imena klase ili interfejsa, nizovni – moze i preko korisnickog i preko primitivnog tipa da se predstavi, i nabrojivi(enum).



III Struktura Java programa

1. Jezici za opis konstrukcija jezika Java, Bekusova notacija. Primeri

-Sintaksa programskog jezika se opisuje pomocu konacnog skupa metalingvistickih formula(MLF). MLF se sastoji iz leve i desne strane razdvojene univerzalnim meta-simbolom ::=. MLF je oblika a::=b, gde je a metaligvisticka promenljiva, a b metalingvisticki izraz.

-ML promenljiva je fraza prirodnog jezika ogradjena uglastim zagradama.

-ML konstanta je znak ili kljucna rec objekt-jezika

-Univerzalni metasimbol ::= cita se "po definiciji"

-Univerzalni metasimbol | se cita "ili"

-ML izraz moze biti:

*ML promenljiva

*ML konstanta

*konacan niz ML promenljivih ili ML konstanti razdvojenim metasimbolom | ili nadovezanih jedni na druge

Primer:

-Ceo broj se uz pomoc Bekusove notacije moze def na sl nacin:

<ne nula cifra> ::=1|2|3|4|5|6|7|8|9

<cifra> ::= 0|<ne nula cifra>

<znak broja> ::= +|-

<dekadni ceo broj bez znaka> ::=<ne nula cifra>|<dekadni ceo broj bez znaka><cifra>

<dekadni ceo broj> ::=0|<dekadni ceo broj bez znaka>|<znak broja><dekadni ceo broj bez znaka>

-Postoje razne modifikacije izvorne Bekusove notacije uvodjenjem metasimbola(leva i desna velika zagrada, leva i desna srednja zagrada, leva i desna mala zagrada)

2. Jezici za opis konstrukcija jezika Java, sintaksni dijagram. Primeri

-Graf sa jednim ulazom i jednim izlazom. Predtsavlja metalingvisticki izraz:

*cvorovi sintaksnog dijagrama su ML konstante i promenljive

*Elipse predstavljaju konstante

*Pravougaonici predstavljaju promenljive koje se definisu pomocu osebnog sintaksnog dijagrama

*Grane ili potezi definisu vezu izmedju cvorova, tj. strukturu sintaksnog dijagrama

Primere pogledati na prezentacijama

3. Azbuka jezika Java. Primeri

-Svaki Java program je niz Unicode znakova. Svaki Unicode znak se predstavlja kao 16-bitna rec. Ukoliko se u programu pojavljuju ne Unicode znaci, Java prevodilac ih prevodi u Unicode znake i ceo program organizuje po redovima. Svaki Unicode znak u jeziku Java se predstavlja na jedinstven nacin preko tzv. eskejp-sekvence.
<hex cifra> ::= 0|1|2|3|4|5|6|7|8|9|A|a|B|b|C|c|D|d|E|e|F|f
<izvorni znak> ::= <bilo koji ASCII ili EBCDIC kôd>
<unicode znak> ::=
\u{u}<hex cifra><hex cifra><hex cifra><hex cifra>
|<izvorni znak>
Primeri:
A = \u0041 Š = \u0161 π = \u03C0

4. Elementarne konstrukcije jezika Java, identifikatori, literali. Primeri

U elementarne konstrukcije (tokene) ubrajamo elemente jezika koje kompajler izdvaja kao nedeljive celine prilikom prevodjenja programa. U elemetarne konstrukcije spadaju: identifikatori, literali, separatori, operatori, kljucne reci, komentari i beline.

- *Identifikatori :*

  Identifikator, kao sto i njegovo ime kazuje, sluzi za identifikovanje neke konstrukcije u Javi. Sve konstrukcije u Javi, kao sto su: promenljive, klase metodi itd. na jedinstven nacin se imenuju preko identifikatora, pa identifikatore mozemo poistovetiti sa imenima. Sintaksa identifikatora je sledeca: <identifikator> ::= <unicode slovo> |$|_{ <unicode slovo>|$|_|unicode slovo>}

  Identifikator mora poceti slovom, znakom za dolar ili crtom za podvlacenje. U preostalom delu identifikatora, pored ovih znakova, mogu da se pojave cifre. Iz sintaksne definicije ne sledi da je duzina imena ogranicena. Medjutim, iz prakticnih razloga, ona je uvek ogranicena. Prilikom definisanja sopstvenih imeta preporucuje se izbor preglednih imena, po mogucnosti u skladu sa semantikom objekata ili pojma koji predstavljaju. Imena koja su prihvatljiva: strana, Krug, a1, x11,obimKvadrata, Masa1, godPrihod, ..., dok imena : x1yz, xy1z, x1zy,... mogu lako biti pomesana medjusobno i

prouzrokovati greske u programu. Treba istaci da u Javi postoji razlika izmedju malih i veliki slova, tako da su Pera i pera dva razlicita identifikatora.

- Sledeće reči predstavljaju identifikatore u Javi:

  ImeKlase, _read, X, xy123, \u03C0, $b1, I_Ovo_Je_Identifikator, x1y21T23, jo\u0161, MojeSve

- Sledeće reči ne predstavljaju identifikatore u Javi:

  2brata - počinje cifrom

  Novi Sad - sadrži blanko

  lose-definisan - sadrži crtu (znak minus)

  x,y.c - sadrži zapetu i tačku

  a&b - sadrži nedozvoljeni znak &.

- *Literali:*

  Literali u jeziku su reci koje predstavljaju same sebe. U Javi su konstante primitivnog tipa ili konkretan primerak klase String. Pomocu Bekusove notacije to zapisujemo na sledeci nacin:

  <literal> ::= <celobrojni> | <realni> | <logicki> | <znakovni> | <stringovni>

  - *Celobrojni*

    Celi brojevi u Javi mogu biti zapisani kao: dekadni, oktalni ili heksadekadni (a od verzije 7 i kao binarni).

    <nenula cifra> ::= 1|2|3|4|5|6|7|8|9

    <cifra> ::= 0 | <nenula cifra>

    <binarna cifra> ::= 0 | 1

    <oktalna cifra> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

    <hex cifra>::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | a | B | b | C | c | D | d | E | e | F | f

    Primer:

    -korektno zapisani: 0, 125, 3567, 0564, 0XABC, 0x23a4, 56L, 04343343l, 0XE653aL

    -nekorektno zapisani:

    05693 - oktalni broj sadrži dekadnu cifru veću od 7

    123A4 - dekadni broj sadrži nedekadnu cifru

    0XaBH2 - pojavljuje se neheksadekadna cifra u zapisu broja

  - *Realni*

    Realni literali su konstante koje se zapisuju u obliku pokretne tacke. U Javi razlikujemo dva tipa literala: float i double. Razlika izmedju ovih tipova pojavljuje se samo u preciznosti zapisa literala. Realni literal mora sadrzati bar jednu cifru i bilo decimalnu tacku, bilo eksponent. Realni literali mogu da se izraze u: pozicionom zapisu ili eksponencijalnom zapisu.

    Sledeće reči su sintaksno ispravni realni literali: 23.57, 8.879f, .345d, .569, 0.569F, 4455.D, 3.14, 2e-5f, 0.003e+4d, 123E-5, 1.456575e+12F, 3.5E7

    Sledeći zapisi ne predstavljaju realne literale:

0x0.233 - ne postoje heksadekadni realni literal

5F - nije realni literal

53.4-12 - nedostaje slovo E

999E - nedostaje ceo broj iza slova E

- *Logicki*

  Postoje dva logicka literala predstavljena rezervisanim recima false (netacno) i true (tacno). Dakle:

  <logicki literal> ::= false | true

  Prilikom poredjenja nekih velicina uvek se kao vrednost dobija true ili false.

- *Znakovni*

  U izvornom Java programu mogu se koristiti izvorni znaci i neke eskejp-sekvence preuzete iz programskog jezika C. Izvorni kod se podvrgava pretprocesiranju gde se prepoznaju izvorni znaci, a potom se oni prevode u Unicod znake. Sintaksa znakovnog literala se opisuje na sledeci nacin:

  <znakovni> ::= <bilo koji znak osim apostrofa i obrnute kose crte>| <eskejp sekvenca>

  <eskejp sekvenca> ::= \b | \t | \n | \f | \r | \”| \’ | <oktalni eskejp> | <Unicode eskejp>

  <oktalni eskejp> ::= \0 | 1 | 2 | 3 [oktalna cifra][oktalna cifra]

  <Unicode eskejp> ::= \u {u}<hex cifra><hex cifra><hex cifra><hex cifra>

  U Javi se mogu koristiti sledeće eskejp-sekvence:

  ‘\‘‘ - apostrof

  ‘\"‘ - navodnik

  ‘\\‘ - obrnuta kosa crta

  ‘\r‘ - znak za povratak na početak reda

  ‘\n’ - znak za prelazak u novi red

  ‘\f‘ - znak za prelazak na novu stranu

  ‘\t‘ - znak tabulatora

  ‘\b‘ - znak za povratak za jedno mesto unazad.

- *Stringovni*

  Stringovni literal je niska znakova između navodnika. Kao znak stringovnog literala može da se pojavi bilo koji znak osim apostrofa i obrnute kose crte ili eskejp sekvenca. To preko Bekusove notacije zapisujemo na sledeći način:

  <stringovni literal>::= “{[<znak osim apostrofa i obrnute kose crte>]| [<eskejp sekvenca>]}“

  Primeri stringovnih literala:

  “” // prazan string

  “Programiranje i matematika“ // neprazan string

  “Ovo je navodnik \”, a ovo ne \u3232” //string sa eskejp sekvencama

5. Elementarne konstrukcije jezika Java, separatori, operatori. Primeri

- *Separatori:*

  U Javi postoji nekoliko znakova koji služe za razdvajanje jedne vrste elementarnih konstrukcija od drugih. Na primer, u separatore spada simbol ; koji služi za razdvajanje naredbi u Javi.

  U separatore spadaju sledeći znaci: <separator> ::= (|) | { | } |[|] | ; | : | , | .

  Separatori služe samo za razdvajanje i ne određuju operacije nad podacima.

- *Operatori:*

  Operatori omogućavaju operacije nad podacima. Podaci na koje se primenjuju operatori nazivaju se operandi. Operatore možemo da klasifikujemo na više načina. Prema poziciji u odnosu na operande možemo da razlikujemo prefiksne, infiksne i postfiksne operatore. Međutim, s obzirom na to da li se pomoću operatora dodeljuje vrednost ili ne, možemo da razlikujemo dve klase operatora: operatore dodele i ostale.

  <operator> ::= <operator dodele> |<opertor bez dodele>

  <operator bez dodele> ::= <aritmetički opertor> |<relacioni opertor> | <operator po bitovima> | <logički operator> |<uslovni operator> |<instancni operator>

  - *Aritmeticki operatori*

    Aritmetički operatori zajedno sa operandima i separatorima služe za formiranje aritmetičkih izraza. Aritmetički izrazi služe za izračunavanje vrednosti.

    <aritmetički operator>::= + | - | * | / | % | ++ |--

    Operatori + i – mogu biti prefiksni i infiksni. Pored poznatih operatora + – * i /, operator % se koristi za računanje ostatka pri deljenju. Operatori ++ i –– služe za uvećanje, odnosno umanjenje vrednosti izraza za 1.

    Primer:

    7*3 - 7 / 2 + 4 ► 21 – 7/2 +4 // Realizuje se množenje

    21 – 7/ 2 + 4 ► 21 - 3 + 4 //Realizuje se celobrojno deljenje

    21 - 3 + 4 ► 18 + 4 // Realizuje se oduzimanje

    18 + 4 ► 22 // Realizuje se sabiranje

  - *Relacioni operatori*

    Relacioni operatori se mogu još nazvati i operatorima poređenja. Oni služe za poređenje vrednosti operanada.

    <relacioni operator> ::= == | != | < | > | >= | <=

    U Javi se za ispitivanje da li su dva operanda jednaka koristi se simbol == (dvostruka jednakost). Za ispitivanje da li su dva operanda različita koristi se operator != . Rezultat primene relacionih operatora je uvek logičkog tipa (false ili true).

    Primer:

    (2*3 -10/7)!=(6-7%2) ► (6 -10/7)!=(6-7%2) //Mnozenje

    (6 -10/7)!=(6-7%2) ► (6 -1)!=(6-7%2) //Deljenje

    (6-1)!=(6-7%2) ► 5 != (6-7%2) //Oduzimanje

5!=(6-7%2) ► 5 != (6-1) //Racunanje ostatka

5!=(6-1) ► 5 != 5 //Oduzimanje

5 != 5 ► false

- *Bitovni operatori*

Operator po bitovima može biti logički ili operator pomeranja.

<operator po bitovima> ::= **&** | **|** | **~** | **^** | **<<** | **>>** | **>>>**

Redom: Bitovna konjukcija (AND), Bitovna disjunkcija (OR), Bitovna negacija (NOT), Bitovna ekskluzivna disjunkcija (XOR), Pomeranje (šiftovanje) ulevo, Pomeranje (šiftovanje) udesno, Pomeranje (šiftovanje) udesno sa nulama.

- *Logički operatori*

Postoje tri osnovna logička operatora . To su konjukcija, disjunkcija i negacija:

<logički operator> ::= && | || | !

Operator ! je unarni i prefiksni, dok su operatori && i || binarni i infiksni. Kao operandi kod logičkih operatora mogu se pojavljivati samo podaci logičkog tipa.

Primer: Izračunavanje složenog izraza (2 < 3) && (3 != 4) || false se realizuje na sledeći način:

//Realizuje se prvo poredjenje

(2 < 3) && (3 != 4) || false ► true && (3!=4) || false

//Realizuje se drugo poredjenje

true && (3!=4)|| false ► true && true||false

//Realizuje se konjunkcija

true && true || false ► true || false

//Realizuje se disjunkcija

- *Uslovni i instancni operatori*

Uslovni operator se opisuje pomoću znaka pitanja i dvotačke: <uslovni operator> ::= (? :)

Uslovni operator se najčešće koristi u formi: <logicki izraz>?<prvi izraz>:<drugi izraz>

Pomoću instancnog operatora proverava se da li konkretan primerak pripada nekoj klasi.

<instancni operator>::= instanceof

Operator instanceof generiše vrednost true ako je objekat primerak navedene klase (ili interfejsa), a u suprotnom daje vrednost false.

- *Operatori dodele*

Operatori dodele služe da dodele vrednost nekoj promenljivoj.

<operator dodele> ::= <prost operator dodele> | <sastavni operator dodele>

Osnovni oblik operatora dodele je: <prost operator dodele> ::= =

Operator dodele se najčešće upotrebljava u formi: <promenljiva> = <izraz>

Tip promenljive mora da bude kompatibilan sa tipom izraza.

Primer: Operator dodele se može upotrebiti i u tzv. lančanom obliku za višestruko dodeljivanje.

m = n = k = 5; // k dobija vrednost 5, kako je i vrednost izraza k=5 takodje 5, n dobija vrednost 5,

// a po tom principu i m dobija vrednost 5.

Sastavni operatori dodele nastaju kombinovanjem nekih prethodnih operatora i prostog operatora dodele.

<sastavni operator dodele> ::= **+=** | **-=** | ***=** | **/=** | **%=** | **&=** | **|=** | **^=** | **<<=** | **>>=** | **>>>=**

Primer:

P *= a; je istovetno sa: P = P*a;

d /= x+y*z; je kraći zapis za: d = d/(x+y*z);

I sastavni operatori dodele mogu biti ulančani:

int a=2, b=3, s=5, s1=1;

s+=s1+=a*b // s1 dobija vrednost 7, a s vrednost 12.

6. Elementarne konstrukcije jezika Java, ključne reči, komentari, beline. Primeri

- *Ključne reči:*

  Ključne reči su identifikatori koji imaju specijalnu namenu u jeziku Java i ne mogu se koristiti za imenovanje drugih entiteta (promenljivih, klasa i metoda).

  Neke od ključnih reči su: abstract, case, char, boolean, do, double, goto, if, for, while, const, assert, enum...

- *Komentari:*

  Komentari služe da se objasne pojedina mesta u programu. Komentari su, pre svega, namenjeni čoveku, ali se u Javi mogu koristiti i za automatsko generisanje dokumentacije. Konstrukcije Jave su često dovoljno jasne pa komentari ponekad mogu biti i suvišni.

  U Javi postoje 3 vrste komentara:

  1) Višelinijski (komentar u stilu jezika C)

  2) Jednolinijski (komentar u stilu jezika C++)

  3) Dokumentacioni.

  Korišćenjem Bekusove notacije, komentar definišemo na sledeći način:

  <komentar>::=<jednolinijski> | <višelinijski>|<dokumentacioni>

  Jednolinijski komentari se mogu pisati od početka reda ili u redu gde se završava neka naredba

  <jednolinijski>::= //<unicode znak različit od znaka za kraj reda>

  {<unicode znak različit od znaka za kraj reda>} <znak za kraj reda>

  Višelinijski komentar počinje simbolima '/*', a završava se simbolima '*/'.

  <višelinijski> ::= /* {{<unicode znak različit od *> } * <uncode znak različit od /> } */

  Dokumentacioni komentar je jedna vrsta višelinijskog komentara.

<dokumentacioni> ::= /** {{<unicode znak različit od * > } * <unicode znak različit od / > } */

Dokumentacioni komentar se može koristiti za automatsko generisanje dokumentacije.

Primer dokumentacionog komentara:

/** U ovom delu vrsi se korekcija nekih podataka Korekcija se vrsi na osnovu podataka dobijenih iz banke i ucitanih sa Interneta */

- *Beline:*

  Belina je znak koji nema grafički prikaz na izlaznom uređaju. Beline služe za međusobno razdvajanje elementarnih konstrukcija i za oblikovanje programa.

  <belina> ::= <razmak> |<horizontalni tab> |<znak za kraj reda> |<znak za novu stranu> | <znak za kraj
  datoteke>

7. Primitivni tip, celobrojni tipovi. Primeri

Primitivni tip je određen: skupom vrednosti koje se formiraju iz odgovarajućih literala i skupom operacija nad tim vrednostima. To su unapred definisani tipovi u Javi i odmah stoje na raspolaganju korisniku.

Kao primitivni tipovi pojavljuju se:

-brojevi (celi ili realni),

-znakovni tip

-i logički tip.

Primitivni tip moze biti aritmeticki i logicki. Logicki se opisuje rezervisanom reciju boolean :

<primitivni tip>::=<aritmetički tip> | boolean

Aritmeticki tip moze biti celobrojni ili realni.

<aritmetički tip> ::= <celobrojni tip>|<realni tip>

Postoji pet celobrojnih tipova (tu spada i znakovni tip):

<celobrojni tip> ::= byte | short | int | long | char

Realni tip moze biti jednostruke i dvostruke tacnosti , tj:

<realni tip> ::= float | double

- *Celobrojni tipovi*

  U okviru celobrojnih tipova podataka možemo razlikovati: byte, short, int, long i char.

  Za svaki od tih tipova postoji određeni interval (određen veličinom memorijske reči) iz kojeg mogu uzimati vrednosti. Na primer, podatak tipa byte se upisuju u memorijsku reč dužine osam ćelija (u potpunom komplementu) pa su vrednosti iz [-2^7, 2^7-1]. Svi celobrojni tipovi, osim znakovnog, mogu imati negativne vrednosti. Celi brojevi su u Javi reprezentovani u formatu potpunog komplementa.

tip	reprezentacija	interval
`byte`	8-bitni, oznacen broj u potpunom komplementu	-128 do 127
`short`	16-bitni, oznacen broj u potpunom komplementu	-32768 do 32767
`int`	32-bitni, oznacen broj u potpunom	-2147483648 do

	komplementu	2147483647
long	64-bitni, oznacen broj u potpunom komplementu	-9223372036854775808 do 9223372036854775807
char	16-bitni, neoznacen broj, Unicode	'\u0000' do '\uffff'

Celobrojni tip karakterišu aritmetički, relacioni i operatori po bitovima. U Javi se aritmetičke operacije izvršavaju prevođenjem svih ostalih celobrojnih tipova u int ili long.

8. Primitivni tip, realni tipovi. Primeri

- *Realni tipovi:*

U Javi postoje dva realna tipa: **float** – jednostruke tačnosti odnosno 32 bita i **double** – dvostruke tačnosti odnosno 64 bita. Brojevi se zapisuju prema standardu IEEE 754.

Za svaki od prethodna dva podtipa postoje: najmanji i najveći negativan realni broj, nula, najmanji i najveći pozitivan realni broj, pa realni tip možemo predstaviti pomoću brojne ose:

----------|----------|----|----|----------|----------

- ∞ −Ω −ω 0 ω Ω ∞

Ovde su sa Ω i ω označeni, redom, maksimalni i minimalni realan broj po apsolutnoj vrednosti u okviru odgovarajućeg realnog tipa.

-Realni brojevi iz oblasti (-∞, -Ω) ne mogu se registrovati.

-Ako je rezultat neke operacije iz tog intervala, nastupilo je prekoračenje (eng. overflow) - taj rezultat tretira se kao -∞ (-Infinity).

-Slično važi za brojeve iz oblasti (Ω, +∞) samo je (+Infinity).

Realni brojevi iz oblasti (-ω, 0)U(0, ω), takođe ne mogu biti registrovani. Ako je rezultat neke operacije iz ove oblasti, pojavljuje se potkoračenje (eng. underflow). Za razliku od prekoračenja, rezultat se tretira kao nula zato što je reč o veoma malim brojevima – bliskim nuli. Pri operisanju sa ovakvim brojevima mogu se dobiti nekorektni rezultati.

-Realni brojevi iz oblasti [-Ω, -ω]U{0}U[ω, Ω] mogu se registrovati u Javi.

U stvari, tačno se mogu registrovati samo tzv. centralni brojevi, a svi ostali samo približno. Ako je x centralni broj, tada se svi realni brojevi (u matematičkom smislu), iz dovoljno male okoline za x, zamenjuju brojem x.

Na realni tip podataka mogu da se primenjuju relacioni i aritmetički operatori.

Prilikom operisanja sa realnim brojevima može se kao rezultat pojaviti nešto što nije broj. Na primer, ako se nula deli nulom i stoga postoji posebna vrednost označena sa NaN (eng. Not a number).

9. Primitivni tip, znakovni tip. Primeri

Java koristi Unicode skup znakova za prikaz karaktera (znakova). Tip char je 16 bitni bez znaka i koristi se za smeštanje desetina hiljada karaktera iz punog internacionalog Unicode skupa karaktera. Opseg karaktera je od 0 do 65536.

Proverite:

```
char a='a', b=97;
int i=a+1, j=b+1;
System.out.println("i="+i+", j="+j+"\na="+a+", b="+b);
```

Iako se znakovi (karakteri) ne koriste kao brojevi mogu se vršiti operacije nad njima kao da su celi brojevi.

Proverite:

```
int tri=3;
char jedan='1';
```

```
char четири=(char) (tri+jedan);
int četiri=tri+jedan;
System.out.println("tri="+tri+", jedan="+jedan+"\nчетири="+четири+",
četiri="+četiri);
```

Znakovni tip služi i za unos kontrolnih sekvenci. Početak kontrolne sekvence je obrnuta kosa crta. Uobičajeni kontrolni znakovi su:
\uXXXX - Unicode znak (na mestu XXXX je heksadekadni broj)
\b - Backspace (brisanje prethodnog znaka)
\f - Form feed
\n - novi red
\r - Carriage reurn
\t - tabulator
\\ - kosa crta
\” - navodnici

10. Primitivni tip, logički tip. Primeri

Logički (boolean) tip je okarakterisan:

- skupom logičkih konstanti true i false koje ne mogu imati drugo značenje,
- skupom logičkih operatora i operatorima jednakosti i nejednakosti.

Logički tip je dobio naziv po imenu engleskog matematičara Bula (George Boole, 1815-1864) koji se smatra osnivačem matematičke logike.

Naredbe if, while, for, do-while i uslovni operator ?: zahtevaju logičke vrednosti za navođenje uslova.

*Objektni tipovi

Objekat u sebi objedinjuje skup podataka i postupaka za rad sa tim podacima.
Objektni tip u Javi može biti:
• korisnički,
• nizovni
• i nabrojivi.
Korisnički objektni tip definiše sam korisnik preko imena klase ili imena interfejsa.
Nizovni tip se može definisati bilo preko korisničkog objektnog tipa, bilo preko primitivnog tipa.
Nabrojivi tip definiše se preko ključne reči enum i imena klase.
Ovde je :
<ime klase> ::= <identifikator>
<ime klase ili interfejsa> :: = <identifikator>

- *Korisnički objektni tip:*
  Deklaracijom klase praktično se definiše novi korisnički tip u Javi. Ime klase može da se koristi za deklarisanje promenljivih, kao što se kod primitivnih tipova koriste rezervisane reči poput: int, boolean, double,...
  Primer: ako definišemo klasu Figura na sledeći način:
  class Figura { ... }
  tada ima smisla deklarisati promenljive:
  Figura a, b;
  Dakle, naredbom Figura a, b; deklarisane su dve promenljive pomoću kojih možemo pristupati konkretnim objektima klase Figura.
  Iz postojeće klase može se kreirati nova klasa tako što će imati sve osobine postojeće klase i neke dodatne. Takva klasa naziva se potklasa postojeće klase, i pošto ima sve osobine postojeće klase (natklase), za nju se kaže da je nasledila postojeću klasu. Dakle, svi kreirani objekti potklase imaju osobine postojeće klase i neke dodatne koje su navedene u potklasi. Mehanizam nasleđivanja je bitan za objektno orijentisane jezike jer omogućava kreiranje novih klasa iz postojećih.

Način zapisa podataka objektnog tipa (objekata) u memoriji se razlikuje od načina zapisa podataka primitivnog tipa. Podacima u memoriji se pristupa preko promenljivih:

• Kod primitivnih tipova promenljive sadrže podatke sa kojima se operiše,

• dok su kod objektnog tipa promenljive pokazivači na objekte.

Primer: ako deklarišemo klasu:

class Osoba{ …}

i kreiramo primerak klase korišćenjem promenljive p

Osoba p = new Osoba ();

onda je p referenca (pokazivač) na adresu u memoriji od koje počinje zapis kreiranog objekta.

Još jedna razlika između objekata i podataka primitivnog tipa je sto se objekti kreiraju dinamički, tek prilikom izvršavanja programa, tada se za njih rezerviše memorija, dok se za podatke primitivnog tipa memorijski prostor rezerviše statički još u fazi prevođenja.

Pozivanje metoda nad objektom: Nad promenljivama objektnog tipa se mogu pozivati metodi pripadajuće klase. To se rati tako što se navede ime promenljive, zatim punktualni operator (tj. tačka), a zatim naziv metoda koji se poziva praćen argumentima poziva između malih zagrada. Ako nema argumenata poziva, tada se iza naziva metoda obavezno moraju napisati otvorena i zatvorena mala zagrada.

Primer:

Pretpostavimo da promenljiva prvi klase Ucenik pokazuje na novonapravljeni objekat. Poziv metoda stampajIme koji postoji u klasi Ucenik nad objektom na koji referiše promenljiva prvi realizuje se sledećom naredbom: prvi.stampajIme();

11. Promenljive, deklaracija i inicijalizacija vrednosti, opseg važenja. Primeri

Promenljiva- lokacija u memoriji za zapis neke vrednosti ili referenca na objekat.
U Javi postoje 3 vrste promenljivih:

1. instance (promenljive primerka odnosno objekta)
2. klasne
3. i lokalne.

Svaku promenljivu karakterišu: ime, tip i vrednost:

1. ime promenljive je identifikator.
2. tip promenljive je jedan od nabrojanih tipova.
3. promenljiva ili sadrži vrednost ili je referenca na objekat. (Ako promenljiva sadrži vrednost, onda je ta vrednost literal primitivnog tipa.)

Svaka promenljiva mora biti deklarisana. Prilikom deklarisanja određuje se tip promenljive. Može se dodeliti i početna vrednost (inicijalizacija):

• Lokalne promenljive moraju dobiti početnu vrednost pre korišćenja (inače se javlja greška pri prevođenju).

• Promenljive primerka i klasne promenljive ne moraju eksplicitno dobiti početnu vrednost pre korišćenja.

Ako promenljive primerka nisu eksplicitnom naredbom dobile početnu vrednost, podrazumevane vrednosti su:

null - referenca,
0 - numerička,
‘\0’ - znakovna,
false - logička.

Primer:

```
//deklaracija
int brojGodina;
String prva, druga, tvojaNiska;
//sa inicijalizacijom
int i, k, n=32;
boolean ind = false;
String ime = “Dusan”;
float a= 3.4f, b=5.8f, y=0.2f;
```

12. Naredbe, obeležena naredba. Primeri

Iskoristićemo Bekusovu notaciju da definišemo naredbu:
<naredba> ::= <blok>|<obeležena naredba>
|<prazna naredba>|<izraz-naredba> |<osnovna naredba dodele>|<naredba if>
|<naredba switch>|<naredba ponavljanja> |<naredba break>|<naredba continue>
|<naredba return>|<naredba throw> |<naredba try>|<naredba synchronized>
Blok je sekvenca od nula, jedne ili više naredbi ili deklaracija lokalnih promenljivih ograđenih vitičastim zagradama. Uočavamo da je prethodna definicija bloka rekurzivna: blok je definisan preko naredbe, a već smo videli da naredba može biti blok. Tela klasa, metoda itd- su blokovi. Blok je strukturirana naredba koja može sadržati druge blokove i bilo koje druge naredbe koje se izvršavaju jedna za drugom dok se ne naiđe na naredbu za promenu toka upravljanja. Naredba je konstrukcija pomoću koje se kontroliše tok programa i odvijanje operacija u Javi. Naredbe se završavaju znakom ; (tačkom-zapetom).
U bloku mogu biti deklarisane promenljive. One su lokalne za blok i vidljive su (mogu se koristiti) samo od mesta deklarisanja do kraja bloka. Lokalna promenljiva ne može biti korišćena ukoliko joj nije dodeljena početna vrednost. Blok ne može sadržati više promenljivih sa istim imenom. Naredba deklarisanja promenljivih sastoji se iz navođenja tipa i spiska promenljivih razdvojenih zapetama.
Formalni argumenti tj. parametri jednog metoda dobijaju početne vrednosti prilikom poziva metoda. Prazna naredba je bez dejstva. Koristi se u onim delovima programa gde nema nikakvih akcija. Naredba u programskom jeziku može imati jedno ili više obeležja (labela) i onda se naziva obeležnom naredbom.
Obeležena se koristi u kombinaciji sa naredbama break i continue za bezuslovni prenos upravljanja na određeno mesto u programu. Dodeljivanje vrednosti promenljivoj u Javi može da se izvrši na više načina, ali glavni način naredba dodele. Ona omgućava da se promenljivama dodele vrednosti do kojih se došlo nakon određene obrade.
Primer:

```
public class PrekidSaObelezjem {
	public static void main(String[] args) {
		int[][] niz = { { 32, 87, 3, 589 }, { 12, 1076, 2000, 8 }, { 622, 127, 77, 955 } };
		int traziSe = 12;
		boolean pronadjen = false;
		obelezje1: //obelezje
		for (int i = 0; i < niz.length; i++) {
			for (int j = 0; j < niz[i].length; j++) {
				if (niz[i][j] == traziSe) {
					pronadjen = true;
					break obelezje1;
				//da je samo break, bez obelezja, onda bi se iskocilo samo iz
				unutrasnje petlje
				}
			}
		}
		System.out.println("Pronadjen "+pronadjen);
	}
}
```

13. Izrazi, operatori: arnost, asocijativnost, prioritet. Primeri

Izrazi u Javi se koriste da donesu, izračunaju i smeste neku vrednost. Prioritet operatora u izrazu određuje redosled izračunavanja vrednosti. U jedan izraz mogu biti uključeni: operandi, operatori i separatori. Operand u izrazu može da bude: konstanta, tekuća vrednost promenljive, rezultat poziva metoda i dr.

<izraz> ::= <primarni izraz>| < izraz dodele>
| <aritmetički izraz> | <relacioni izraz> | <logički izraz>| <izrazi sa operatorima po bitovima>
| <uslovni izraz>| <instancni izraz> | <izraz kastovanja>| <unarni izraz>

Primarni izraz je jedna od najkompleksnijih komponenti pojma izraz. Sintaksu primarnog izraza definišemo ovako:

<primarni izraz> ::=<primarni prefiks>{<primarni sufiks>}

<primarni prefiks> ::= <literal> | <ime> | this | super.<identifikator> | (<izraz>) | <alokacioni izraz>

<primarni sufiks> ::= .<identifikator> | .this | .class | .<alokacioni izraz> | [<izraz>] | <argumenti>

Iz navedene definicije vidimo da izraz može biti literal, ime promenljive, ključna reč this, ali i izraz između male otvorene i zatvorene zagrade. Dakle, pojam izraza se definiše preko samog sebe, što znači da je prethodna definicija izraza rekurzivna.

Primeri nekih izraza: args.length, Math.random(), System.out.print(niz).

Oslanjajući se na ranije definisane operatore, možemo definisati sledeće tipove izraza:

<izraz dodele> ::= <izraz> <operator dodele> <izraz>

<aritmetički izraz> ::= <izraz> <aritmetički operator> <izraz>

<relacioni izraz> ::= <izraz> <relacioni operator> <izraz>

<logički izraz> ::= <izraz> <logički operator> <izraz>

<izrazi sa operatorima po bitovima> ::= <izraz> <operator po bitovima > <izraz>

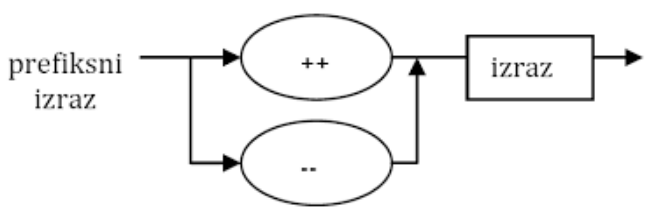
<uslovni izraz> ::= <izraz> ? <izraz> : <izraz>

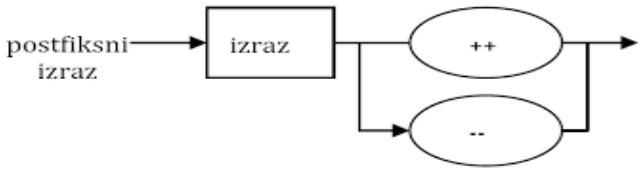
<instancni izraz> ::= <izraz> instanceOf <tip>

<izraz kastovanja> ::= (<tip>) <izraz>

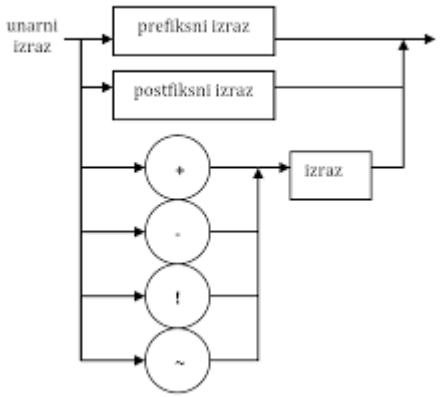
U izrazu dodele kao operator dodele može da se pojavi i =. Dakle, osnovna naredba dodele predstavlja specijalni slučaj izraza dodele.

Da bi se definisao unarni izraz, potrebno je prvo definisati prefiksni izraz i postfiksni izraz:





Sada je unarni izraz definisan sledećim sintaksnim dijagramom:



U jednom izrazu može da se pojavi veći broj operatora. Postavlja se pitanje kojim redosledom treba da se primene operatori da bi se korektno izračunala vrednost izraza. Svakom operatoru pridružen je prioritet – operatori u jednom izrazu koji imaju veći prioritet realizuju se pre operatora sa nižim prioritetom. Ako više operatora imaju isti prioritet, postavlja se pitanje kako se tada realizuje izračunavanje. Jedna od karakteristika operatora je asocijativnost. Operator može biti levo-asocijativan, desno-asocijativan i neasocijativan. Aritmeticki operatori su levo-asocijativni. Desno-asocijativni operator je operator dodele. Relacioni operatori su neasocijativni pa u Javi nema smislia izraz: a < b < c. U sledećoj tablici prikazani su operatori sa odgovarajućim prioritetima i asocijativnostima.

Prioritet	**Operator**	**Asocijativnost**
1	(), []	neasocijativni
2	new	neasocijativni
3	.	levo-asocijativni
4	++, --	neasocijativni
5	-(unarni), +(unarni), !, ~, ++, --, (*tip*)	desno-asocijativni
6	*, /, %	levo-asocijativni
7	+, -	levo-asocijativni

8	<<, >>, >>>	levo-asocijativni
9	<, >, <=, >=, instanceof	neasocijativni
10	==, !=	levo-asocijativni
11	&	levo-asocijativni
12	^	levo-asocijativni
13	|	levo-asocijativni
14	&&	levo-asocijativni
15	||	levo-asocijativni
16	?:	desno-asocijativni
17	=, *=, /=, %=, -=, <<=, >>=, >>>=, &=, ^=, |=	desno-asocijativni

***Kompilacione jedinice**

U sastav kompilacione jedinice mogu ući:

- package direktiva (paketne naredbe),
- import direktiva (naredbe uvoza) i
- definicije klasa i/ili interfejsa i/ili enumeracija .

Paketna naredba služi za određivanje da se data klasa/interfejs (koja se definiše) nalazi u datom paketu. Naredba uvoza omogućuje da se, prilikom poziva tj. slanja poruke, umesto punog imena klase/interfejsa (imena koje obuhvata ime paketa). Klasa za čitanje podataka sa ulaza je jednoznačno određena punim imenom java.util.Scanner. Naredba import omogućuje da, toj klasi više ne mora da se pristupa korišćenjem punog imena te klase, već korišćenjem njenog skraćenog imena: Scanner.

14. Naredbe grananja, naredba if. Primeri

Naredbe grananja omogućavaju da se odabere izvršavanje jednog dela programa u zavisnosti od ispunjenja određenih uslova. Koriste se još i nazivi uslovne naredbe, odnosno naredbe odabiranja.

Postoje dve naredbe grananja u jeziku Java: naredba if i naredba switch.

- *Naredba if:*

Za uslovno izvršenje naredbe ili za izbor između izvršenja dve naredbe obično se koristi naredba if.

Naredba if pojavljuje se u dva oblika:

1. if (<izraz>) <naredba> - nepotpuni oblik
2. if (<izraz>) <naredba> else <naredba> - potpuni oblik.

Naredba if u nepotpunom obliku izvršava se na sledeći način:

1. izračunava se vrednost izraza;
2. ako je vrednost izraza istinita, izvršava se naredba koja sledi iza izraza;
3. ako je vrednost izraza neistinita, izvršava se prva naredba koja sledi iza naredbe if.

Naredba if u potpunom obliku se izvršava na sledeći način:

1. izračunava se vrednost izraza;
2. ako je vrednost izraza istinita, izvršava se naredba koja sledi iza izraza;
3. u suprotnom, izvršava se naredba iza rezervisane reči else.

Primer:

```
if (a < b ) System.out.println("a je manje od b");
if (tezina>200) {
    tezina = 200;
    ind = true;
}else
    ind =false;
```

Umesto naredbe if u nekim situacijama može se upotrebiti uslovni operator ?:

(uslov) ? then-izraz : else-izraz

Primer:

```
min = (x<y) ? x : y;
```

15. Naredbe grananja, naredba switch i naredba break. Primeri

- *Naredba switch*

Naredba switch služi za izbor jedne naredbe iz skupa od nekoliko mogućih, a na osnovu vrednosti nekog izraza.

Rezervisana reč default može se pojaviti samo jedanput, što nije posebno naznačeno.

Izraz iza rezervisane reči switch naziva se selektor i mora biti tipa:

1. byte, char, short ili int;
2. Enumerisani tip (od verzije 5);
3. String, kao i tipa neke od klasa-omotača prostih tipova: Character, Byte, Short ili Integer (od verzije 7).

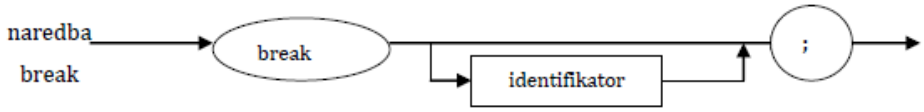
Vrednosti (oznake) koje se pojavljuju iza naredbe case moraju biti istog tipa kao i selektor i međusobno različite.

Naredba switch se izvršava na sledeći način:

1. Izračunava se vrednost selektora i ako nije tipa int, vrednost se prevodi u tip int.
2. Upoređuje se izračunata vrednost sa vrednostima (oznakama) iza naredbi case.
3. Ako se desi poklapanje izvršava se poklapajuća case naredba.
4. Ako se ne poklapa ni sa jednom case naredbom om onda se izvršava default naredba.
5. Ako nema naredbe koja ima oznaku default, naredba posle naredbe switch biće sledeća koja se izvršava.

- *Naredba break*

Sledećim dijagramom opisana je sintaksa naredbe break:



Ako naredba break ne sadrži identifikator, upravljanje se prenosi na prvu sledeću naredbu iza (bloka?) naredbe u kojoj se nalazi.

Primer:

```
switch (doba) {
        case “prolece”:
                System.out.println(“Setnja u prirodi”);
                break;
        case “leto”:
                System.out.println(“Kupanje”);
                break;
        case “jesen”:
                System.out.println(Branje voca”);
                break;
        case “zima”:
                System.out.println(“Skijanje”);
                break;
}
```

16. Naredbe ciklusa, naredba while, naredba do-while, naredba for brojački ciklus. Primeri

- *Naredba while*

Naredba while se još naziva naredba ciklusa sa preduslovom. U njoj se najpre proverava da li je ispunjen uslov i ako jeste izvršavaju se naredbe ciklusa.

Izraz u naredbi while je logičkog tipa. Efekat naredbe while je sledeći:

1. izračunava se vrednost izraza;
2. ako je vrednost izraza istinita, izvršava se naredba i ponavljaju koraci 1. i 2. ;
3. ako je vrednost izraza neistinita, završava se izvršavanje naredbe while i prelazi se na prvu sledeću naredbu iza while.

Primer:

```
while (x>0.1) {
        x=Math.random();
        System.out.println("x="+x);
}
```

- *Naredba do-while*

Naredba do-while se još naziva naredba ciklusa sa postuslovom. Sintaksa naredbe do-while je sledeća:

do-----naredba-----while-----(--izraz--)----;

Izraz je logičkog tipa.

Izvršavanje naredbe do-while realizuje se preko sledećih koraka:

1. izvršava se naredba iza rezervisane reči do;
2. izračunava se vrednost izraza u zagradama;
3. ako je vrednost izraza istinita, proces se nastavlja;
   u suprotnom, prelazi se na prvu naredbu iza naredbe do-while.

Primer:

```
do
        x=Math.random();
        System.out.println("x="+x);
while (x<0.9);
```

- *Naredba for*

Najčešće se koristi za opis ciklusa kod kojih je broj ponavljanja naredbi unapred poznat.

Sintaksa:

for----(--inicijalizacija--;--izraz--;--iteracija--)----naredba

Svaka od sekcija naredbe for može biti izostavljena, ali znak ; mora postojati.

Naredba for se izvršava na sledeći način:

1. Prvo se izvršava inicijalizacioni deo petlje for:
   - Podešava se vrednost upravljačke promenljive petlje for.
   - Inicijalizacija se izvršava samo jedanput.
2. Nakon inicijalizacije:
   - Izračunava se vrednost izraza (ako postoji) koji mora biti logičkog tipa.
   - Ako izraz ne postoji, podrazumevana vrednost je true.
3. U zavisnosti od vrednosti izraza:
   - Ako ima vrednost true, izvršava se telo petlje, a zatim se izvršava iteracija petlje i ponovo se ide na korak 2.
   - Ukoliko izraz ima vrednost false, okončava se izvršavanje petlje.

Primer:

```
for (int i=1; i<=10; i++)
        System.out.println(”Ana voli programiranje”);
```

17. Naredbe ciklusa, naredba break, naredba continue. Primeri

- *Naredba break*

Pored toga što se naredba break može koristiti:

1. za izlazak iz naredbe switch,
2. ona se može iskoristiti za izlazak iz petlje,
3. ali i za bezuslovni prenos upravljanja na obeleženu naredbu (pod određenim uslovima).

Ovo predstavlja jednu vrsta “kontrolisane” goto naredbe.

Primer1: Da bi se prekinulo izvršavanje naredbi unutrašnje petlje i vratilo se u spoljašnju, koristi se naredba break bez obeležja:

```
for(int i=0; i<100; i++) {
    for(int j=0; j<100; j++) {
        if (uslov)
                break;
    }
}
```

Primer2: Da bi se prekinulo izvršavanje naredbi obe petlje, koristi se naredba break sa obeležjem

```
izlaz: // ovo je labela
for(int i=0; i<100; i++) {
    for(int j=0; j<100; j++) {
        if (uslov)
                break izlaz;
    }
}
```

- *Naredba continue*

Može se koristiti samo u naredbama za opis ciklusa. Korisna je kada treba da se nastavi izvršavanje ciklusa, uz preskakanje određenog skupa naredbi. Pomoću ove naredbe upravljanje se prenosi na izraz za proveru uslova izlaska iz petlje.
Primer: Učitavanje niza od četiri binarne cifre i spajanje učitanih cifara u jedan string, te prikaz stringa. Nastavlja se sa sledećom iteracijom petlje obeležene obeležjem ako cifra nije binarna.

```
String s="";
System.out.println("Unesite cetiri binarne cifre ");
vrh:
do {
        d=sc.nextInt();
        if((d!=0)&&(d!=1))
                continue vrh;
        s +=d;
}
while(s.length()<4);
System.out.println("Formiran je binarni string: "+s);
```

18. Ulazna tačka programa, static metoda main. Primeri

Ulazna tačka Java programa je metod main(), tj. Izvrsavanje programa pocinje izvrsavanjem ove metode(funkcije). Da bi se program mogao izvrsiti, metod main() mora biti oznacen kljucnom reci public i kljucnom reci static. Metod main() mora vratiti void. Metod ima jedan parametar koji omogucuje da se procitaju vrednosti prosledjene preko komandne linije prilikom prokretanja programa. Zaglavlje ovog metoda ima oblik:

```
public static void main (String[] argumentiKomandneLinije)
```

Primer:

```
Public static void main (String[] args)
{
        System.out.println( 40 + 2);
}
```

19. Definicija i poziv static metoda, supstitucija parametara. Primeri

Static - ukazuje da postoji samo jedan takav metod u datoj klasi tj. da nema potrebe praviti objekat da bi se pozvao taj metod.
Staticki metod main() obicno nije jedini metod koji se nalazi u klasama. Obicno se delovi funkcionalnosti izdvajaju u posebne celine. To se moze postici funkcionalnom dekompozicijom. U tom slucaju se realizacija izdvojenih funkcionalnosti izmesta iz metoda main() u posebno izdvojeni metod, koji je takodje static i onda u main()-u ostaje samo poziv tog izdvojenog statickog metoda. Ideja je da se dugacak kod razbije na manje celine. Osnovna struktura statickog metoda je:

```
static PovratniTIp imeMetoda(arg1,...,argn)
{
        Kod metoda
}
```

Primer:

```
static double zbir(int x, double y)
{
        return x + y;  //i u mainu System.out.println(zbir(21, 2.0));
}
```

Prilikom poziva static metoda desava se sledece:

1. Evaluiraju se argumenti iz poziva metoda
2. Izracunata vrednost argumenata zamenjuje parametre metoda, prema redosledu navodjenja, a da bi to bilo moguce potrebno je da se broj parametara u definiciji metoda poklapa sa brojem argumenata i da tip svakog od argumanata bude u saglasnosti sa tipom odgovarajuceg parametra
3. Izvrsavaju se redom naredbe u telu metoda
4. Izvrsavanje tela metoda zavrsava se kada se dodje do izvrsavanja naredbe return (rezultat rada funkcije je vrednost izraza koji sledi iza te naredbe i tip tog izraza mora odgovarati povratnom tipu iz deklaracije metoda) ili do poslednje naredbe u telu metoda – tada nastupa povratak u pozivajuci metod

5. Prilikom povratka u pozivajuci metod, izvrsavanje se nastavlja od naredbe koja sledi iza naredbe poziva

Pri pozivu se vrsi supstitucija parametara po vrednosti, evaluiraju se vrednosti argumenata i parametri funkcije se zamenjuju tim vrednostima.

***Rekurzija**

Oznacava postupak ili funkciju (metod) koji u svojoj definiciji koriste sami sebe. Drugim recima, ukoliko neki postupak zahteva da delovi problema koje je razdvojio od drugih bivaju nezavisno podvrgnuti istom tom postupku, taj postupak je rekurzivan. (svodimo na problem manje dimnzije)

U svakom rekurzivnom postupku razlikujemo dve celine:

1) rekurzivni korak – svodjenje problema datog tipa i dimenzije na problem/potproblem istog tipa ali nize dimenzije

2) izlaz iz rekurzije – zavrsetak postupka kada je dimenzija problema mala

Razlikuju se samorekurzija (metod se direktno svodi na sebe) i uzajamna rekurzija (svodjenje ide preko drugih metoda).

Primer:

```
static int faktorijel (int n) {
	if(n==0 || n==1)
		return 1;
	return n*faktorijel(n-1);
}
```

IV Rad sa korisničkim objektnim tipovima isporučenim uz JDK

1. Pristup sistemu, klasa System. Primeri

Pun naziv ove klase je java.lang.System. Ona se nalazi u paketu lang unutar paketa java i na nju se može referisati korišćenjem punog imena tj. Pomoću java.lang.System. Za nju, kao i za ostale klase iz paketa java.lang (i samo za njih) važi da ne mora da se koristi puno ime, već može da se referiše na klasu korišćenjem skraćenog naziva klase tj. pomoću System.

Klasa System, pored ostalih funkcionalnosti, obezbeđuje i sledeće mogućnosti:

- standardni izlaz, objekat System.out
- tok za prikaz poruka o greškama, System.err
- merenje proteklog vremena pomoću časovnika računara
- interakcija sa sakupljačem otpadaka, System.gc();
- pristup spoljašnje definisanim osobinama i promenljivima okruženja, System.getProperty(“....”); - kao informaciju nam daje string koji predstavlja datu osobinu,
  za ostavljanje osobina - System.setProperty(“...”,”...”);
  za uklanjanje osobina - System.clearProperty(“...”);
  metod za čitanje znaka za kraj linije System.lineSeparator()
- direktan završetak rada programa, System.exit(status);
- standardni ulaz System.in
- pomoćni metod za brzo kopiranje memorijskih blokova, tj. delova niza
- mehanizam za učitavanje datoteka i biblioteka

Primer:

```
long vremePocetka = System.currentTimeMillis();	//moze i System.nanoTime();
  //for petlja ili bilo sta drugo za sta zelimo da izracunamo vreme izvrsavanja
long trajanje = System.currentTimeMillis() - vremePocetka;
```

2. Prikaz teksta na konzoli, objekat System.out. Primeri

Nad objektom out koji u klasi System mozemo pozvati metod println (ili neki drugi – print, printf) kako bismo prikazali neki tekst u konzoli.

Primer:

```
double x = zbir(40, 2.0); //zbir iz 19og zadatka
System.out.print(x); //ispise x
System.out.println(); //novi red
System.out.println(x); //alternativa da se i jedno i drugo postigne odjednom
```

System.out.printf(“%6.0f\n”, x); //sintaksa analogna onoj u C-u
System.out.println(“Suma je” + x + “!”);
System.out.printf(“Suma je %8.0f!\n”, x)

3. Pristup elementima Java okruženja, objekat System.gc, metod System.lineSeparator(). Primeri

- *System.gc*
  Sakupljac otpadaka radi kao pozadinska nit koja se probudi s vremena na vreme kada se neki spoljasnji uslovi dese. Samo vreme budjenja ne moze direktno specificirati Java programer, ali moze signalizirati da smatra da bi bio dobar trenutak za pokretanje sakupljaca otpadaka. Kad se probudi, prolazi kroz memoriju, uocava koji su delovi memorije markirani kao zauzeti, a njima niko ne moze pristupati, to su zapravo otpadci, pa ih reciklira, tj. prijavljuje da su ponovo slodobni.
  Primer:
  ```
  int n = 1000;
  double s = 0;
  for(int i = 0; i < n; i++) {
      for(int j = 0; j < n; j++) {
          Object o = new Object();
          s++;
      } //kada se izadje iz ove unutrasnje for petlje ne moze se vise referisati na taj
  objekat pa on                    //postaje otpadak
  System.out.gc(); //ovo samo znaci da mislimo da je pravi trenutak da se pokrene
garbage collector ali to                    //ne znaci da on mora da se pokrene
  }
  ```
- *System.lineSeparator()*
  lineSeparator sadrži posebne znakove kao što su '\ r' ili '\ n' koji označavaju kraj linije. Međutim, iako bi oni bili napisani u Java izvornom kodu, oni se ne pojavljuju na taj način kada se štampaju. Svrha separatora linija je odvajanje linija.

4. Objekti, primerci klase Object, kreiranje objekta. Primeri

Klasa java.lang.Object je koren drveta koji predstavlja hijerarhiju klasa. Dakle, za svaku od Java klasa, klasa Object je njihova nadklasa – ako ne direktna nadklasa, onda natklasa natklase itd. – ako se prođe kroz nadklase, na vrhu se
nalzi klasa Object. Svi objekti, uključujući i nizove i enumerisane tipove, implementiraju metode koji su definisani u klasi Object. Metodi definisani u klasi Object se mogu primeniti na bilo koji objekat (tj. primerak bilo koje Java klase).
Kreiranje objekata:
Objekte pravimo pomocu new Object();
Primer:
```
Object objekat1 = new Object();
Object objekat2 = new Object();
//mogu se porediti pomcu == ili pomocu .equails (objekat1.equals(objekat2)), u oba
slucaja dobijamo da nisu //identicni jer ne zauzimaju isti memorijski prostor
```
Posto se prave pomocu new, to su primerci klase Object i na njima se mogu primenjivati metodi primerka:

- .hashCode() - vraca ceo broj koji nam govori koji je hash kod objekta
- .getClass() - daje nam informaciju koja je klasa tog objekta
- Operator instanceof sluzi za proveru da li je izraz levo od instanceof primerak klase koja se nalazi desno

5. Objekti, primerci klase Object, poređenje objekata. Primeri

• Klasa java.lang.Object je koren drveta koji predstavlja hijerarhiju klasa.
  ◦ Dakle, za svaku od Java klasa, klasa Object je njihova nadklasa- ako ne direktna nadklasa, onda natklasa natklase itd. - ako se prodje kroz sve natklase, na vrhu se nalazi klasa Object.
  ◦ Svi objketi, uključujući i nizove i enumerisane tičove, implementiraju metode koji su definisani u klasi Object
  ◦ Metodi definisani u klasi Object se mogu primeniti na boli koji objekat

Primer:

```
class Auto{
 int brojTockova;
 int brojPutnika;
 float zapreminaMotora;
}

public class Primer{
 public static void main(String[] args){
//Kreiranje objekta tipa Auto postizemo pozivanjem operatora new (tip(u ovom slucaju Auto())

Auto auto1 = new Auto();
auto1.brojTockova=4;
auto1.brojPutnika=4;
auto1.zempreminaMotora=1500;

Auto auto2 = new Auto();
auto2.brojTockova=4;
auto2.brojPutnika=4;
auto2.zempreminaMotora=1500;


Auto auto3 =auto1;
// Sa (==) se ispituje da li dve promenljive pokazuju na isti objekat, a ne da li su ta dva objekta ista. Zato sto promenljiva sadrzi referencu na objekat, a ne sam objekat. Isto vazi I za primenu metoda equals(auto1.equals(auto2)) ukoliko ga nismo definisali u nasoj klasi drugacije, ukoliko nije definisan, poziva se drektno iz Objecta.
if(auto1 == auto2)
System.out.println("Ova dva objekta su isti tj. Identični(==)");
else
System.out.println("Ova dva objekta nisu isti tj. Identični(==)");
 }
}
```

6.Objekti, primerci klase Object, određivanje da li objekat pripada klasi. Primeri

Početni deo isti kao prethodni

Primer:

Početni deo primera isti

```
//Obe naredbe IF ce vratiti true,cak I druga, iako nije napisano "class Auto extends Object". Zato sto svaka klasa nasledjuju klasu Object cak I ako to ne pise. Binarni operator INSTANCEOF se koristi samo kod referencijalnih topova. Njime se ispituje da li je tip prvog operanda jednaka drugom operandu. Ako je tip prvog operanda moguce eksplicitnom konverzijom konvertovati u tip naveden u drugom operandu, tada je vrednost true, inace false.
if(auto1 instanceof Auto)
System.out.println(" auto1 je primerak klase Auto");
else
System.out.println(" auto1 nije primerak klase       Auto");

if(auto1 instanceof Object)
System.out.println(" auto1 je primerak klase Object");
else
System.out.println(" auto1 nije primerak klase Object");
}
}
```

7.Niske, primerci klase String, karakeristike niski, imutabilnost. Primeri

• Klasa java.lang.String predstavlja niske, koje obmotavaju(enkapsuliraju) nizove znakova. Kada u java kodu napisemo print(“abc”), kompajler ce generisati objekat tipa String I sadrzaj niske tog objekta ce podesiti na “abc”,

• Niske tj primerci klase String su IMUTABILNI(KONSTANTNI) I kada se ovi objekti jednom kreiraju njihove vrednosti se ne mogu menjati

• Niske su objekti, pa se nad njima mogu primeniti sve metode definisane u klasi Object, pri cemu se dobijaju drugaciji odgovori. Zato sto su na nivou klase String neke od realizacija metoda iz klase Object predefinisane.

• Nad primercima klase String mogu da se koriste metode kojima se realizuju sve uobičajene operacije nad niskama(nalepljivanje, pretraga, zamena, formatiranje I sl.);

Primeri:

Najveća prednost imutabilnosti je ta što više referenci mogu bezbedno da pokazuju na isti objekat , jer nema opasnosti da promena sadržaja objekta preko jedne reference utiče na sadržaj koji se očekuje kada se koriste ostale reference. Glavna mana pristupa je u utrošku memorije I efikasnosti izvršavanja koda u kome se radi veliki broj izmena stringova, jer se za svaku izmenu kreira potpuno nova intnscaa tipa String.

String str=”Stringovi”;

str+=” u Javi”;

U ovom slučaju se kreira poptuno novi objekat tipa String sa novim sadržajem “Stringovi u Javi” I referenca na taj objekat se dodeljuje promenljivoj str, a prethodna referenca se odbacuje i ceka na oslobadjanje od skupljaca otpadaka.

8.Niske, primerci klase String, kreiranje niski. Primeri.

Početni deo isti kao prethodni

Kreiranje niski:
Način 1.
```
   String niska1=new String(“Ovo je neki tekst”);
```
Način 2.
```
   String niska2=”Ovo je neki tekst”;
```
Rezultat je isti.

9.Niske, primerci klase String, poređenje niski. Primeri

Početni deo isti kao prethodni

Primer:
```
 public static void main(String[] args){
String niska1=new String(“Ovo je neki tekst”);
String niska2=”Ovo je neki tekst”;
String niska3=niska1;
/*Kada se porede niske sa ( == ) niska1 I niska2 nece biti iste, jer ne referisu na isti prostor u memoriji, dok niska1 I niska3 referisu,i bice iste.
Sa druge strane,koriscenjem metoda equals,niska1.equals(niska2) vratice true, zato sto sadrze iste znakove u istom redosledu(jednake su).*/
if(niska1==niska2)
System.out.println(“ Prve dve niske su iste”);
else
System.out.println(“Prve dve niske nisu iste”);
if(niska1.equals(niska2))
System.out.println(“ Prve dve niske su jednake”);
else
System.out.println(“Prve dve niske nisu jednake”);
```

10.Niske, korišćenje metoda klase String i StringBuilder. Primeri

Primeri metoda klase String:
Postoje razni metodi klase String, evo nekoliko primera, ostatak pogledajte ovde
https://github.com/MatfOOP/OOP/blob/master/predavanja/primeri-java/src/P050302NiskeiZ01MetodiPokretanje.java

```
String niska1 = new String("Ovo je jedna niska.");
String niska2 = " Ovo je druga niska.";
```
Ovim metodama se ne menja niska nad kojom se primenjuju, vec se pravi nova niska..
```
System.out.printf("\nniska1.length(): %d\n", niska1.length());
System.out.printf("\nniska2.trim(): %s\n", niska2.trim());
System.out.printf("niska1.toLowerCase(): %s\n", niska1.toLowerCase());
System.out.printf("niska1.toUpperCase(): %s\n", niska1.toUpperCase());
int pozicija = 2;
System.out.printf("\nniske1.charAt(%d): %c\n", pozicija, niska1.charAt(pozicija));
int pozicijaPocev = 1;
int pozicijaDo = 5;
System.out.printf("\nniska1.substring( %d, %d): %s\n", pozicijaPocev,
pozicijaDo,niska1.substring(pozicijaPocev, pozicijaDo));

System.out.printf("\nspoj niski: %s \n", niska1.concat(niska2));
System.out.printf("spoj niski: %s \n", niska1 + niska2);

niska1.indexOf('%c'); niska1.lastIndexOf('%c'); niska1.startsWith(\"%s\");
niska1.endsWith(\"%s\"); niska1.replace(slovo, novoSlovo));
```

Primeri metotoda klase StringBuilder:

Za efikasno nalepljivanje je bolje koristiti klasu StringBuilder ciji su objekti mutabilni.

Metodi: append(),insert() (za umetanje na datu poziciju u niski). Takodje postoje metodi slicni metodama u klasi String: length, indexOf, lastIndexOf, substring itd.
Postoje I novi metodi koji iskorišćavaju promenljivost sadržaja,na primer:
• deleteCharAt(int index);
• delete(int start,int end);
• reverse();
• setCharAt(int index, char ch);

11. Klase-omotači za primitivne tipove, rad sa objektima tipa Integer, Long. Primeri

INTEGER:

• Klasa java.lang.Integer omotava vrednost primitivnog tipa int u objekat
• Objekat primerka klase Integer u sebi sadrži polje tipa int u kom se čuva vrednost tog Integer objekta
• Primerci klase Integer su imutabilni – kada se ovakav objekat jednom kreira njihova vrednost ne može da se menja
• Nad primercima klase Integer mogu da se primene metodi definisani u klasi Object, sa drugacijim rezultatom.
• Dopuštena je dodela int konstante Integer objektu I obrnuto – sam prevodilac zna, zavisno od konteksta, da prilikom prevodjenja u bajt-kod umetne naredbe za konverziju (autoomotavanje – autoboxing)
• Operacije nad primitivnim tipom int su brže od operacija nad primercima klase Integer

Primer 1:
```
Integer ceoBroj1=new Integer(42);
Integer ceoBroj2= 42;
String niskaZaCeoBroj= ceoBroj1.toString();
System.out.printf("%s",ceoBroj2); // toString ce sam biti pozvan
```

```
// (==) ce vratiti true ako referisu na isti prostor, a equals ce vratiti true ako su polja
medjusobno jednaka
if (ceoBroj1 == ceoBroj2)
System.out.printf("dva cela broja su isti (==)\n");
else
System.out.printf("dva cela broja nisu isti (==)\n");
if (ceoBroj1.equals(ceoBroj2))
System.out.printf("dva cela broja su jednaki (equals)\n");
else
System.out.printf("dva cela broja nisu jednaki (equals)\n");
```

Primer 2:

```
Integer ceoBroj1 = Integer.valueOf("42");
Integer ceoBroj2 = Integer.parseInt("2600");
Integer ceoBroj3 = Integer.parseInt("2A", 16);

int poredjenje = ceoBroj1.compareTo(ceoBroj2);
if (poredjenje > 0)
System.out.printf("ceoBroj1(%s) > ceoBroj2(%s)\n", ceoBroj1, ceoBroj2);
else if (poredjenje < 0)
System.out.printf("ceoBroj1(%s) < ceoBroj2(%s)\n", ceoBroj1, ceoBroj2);
else
System.out.printf("ceoBroj1(%s) == ceoBroj2(%s)\n", ceoBroj1, ceoBroj2);

System.out.printf("\nceoBroj2.byteValue(): %d\n", ceoBroj2.byteValue());
System.out.printf("ceoBroj2.intValue(): %d\n", ceoBroj2.intValue());
System.out.printf("ceoBroj2.longValue(): %d\n", ceoBroj2.longValue());
System.out.printf("ceoBroj2.doubleValue(): %f\n", ceoBroj2.doubleValue());

int x = 42;
System.out.printf("\nInteger.toBinaryString(%d): %s\n", x, Integer.toBinaryString(x));
System.out.printf("Integer.toOctalString(%d): %s\n", x, Integer.toOctalString(x));
System.out.printf("Integer.toHexString(%d): %s\n", x, Integer.toHexString(x));
System.out.printf("Integer.toString(%d): %s\n", x, Integer.toString(x));

Integer max = Integer.max(ceoBroj1, ceoBroj2);
Integer min = Integer.min(ceoBroj1, ceoBroj2);
Integer sum = Integer.sum(ceoBroj1, ceoBroj2);
```

LONG:

• Klasa java.lang.Long omotava vrednost primitivnog tipa long u objekat
• Objekat primerka klase Long u sebi sadrži polje tipa long u kom se čuva vrednost tog Long objekta
• Primerci klase Long su imutabilni – kada se ovakav objekat jednom kreira njihova vrednost ne može da se menja
• Nad primercima klase Long mogu da se primene metodi definisani u klasi Object, sa drugacijim rezultatom.
• Dopuštena je dodela long konstante Long objektu I obrnuto – sam prevodilac zna, zavisno od konteksta, da prilikom prevodjenja u bajt-kod umetne naredbe za konverziju (autoomotavanje – autoboxing)
• Operacije nad primitivnim tipom long su brže od operacija nad primercima klase Long

Primeri za Long su identicni kao primeri za Integer, samo sto se metode pozivaju nad klasom Long.

12.Klase-omotači za primitivne tipove, rad sa objektima tipa Character. Primeri

• Klasa java.lang.Character omotava vrednost primitivnog tipa char u objekat
• Primerka klase Character u sebi sadrži polje tipa char u kom se čuva vrednost tog objekta

• Primerci klase Character su imutabilni – kada se ovakav objekat jednom kreira njihova vrednost ne može da se menja
• U klasi Character su definisani metodi kojima se realizuju uobičajne operacije nad znacima
• Nad primercima klase Character mogu da se primene metodi definisani u klasi Object, sa drugacijim rezultatom.
• Dopuštena je dodela char konstante Character objektu I obrnuto – sam prevodilac zna, zavisno od konteksta, da prilikom prevodjenja u bajt-kod umetne naredbe za konverziju (autoomotavanje – autoboxing)
• Operacije nad primitivnim tipom char su brže od operacija nad primercima klase Character

Primer:

```
Character znak1 = '*';
Character znak2 = 42;
Character znak3 = Character.forDigit(11, 16);

if (znak1.equals(znak2))
System.out.printf("prva dva znaka su jednaki (equals)\n");
else
System.out.printf("prva dva znaka nisu jednaki (equals)\n");
int poredjenje = znak1.compareTo(znak2);
if (poredjenje > 0)
System.out.printf("znak1('%s') > znak2('%s')\n", znak1, znak2);
else if (poredjenje < 0)
System.out.printf("znak1('%s') < znak2('%s')\n", znak1, znak2);
else
System.out.printf("znak1('%s') == znak2('%s')\n", znak1, znak2);
char ch1 = '7';
System.out.printf("\nCharacter.getNumericValue('%c'): %d\n", ch1,
Character.getNumericValue(ch1));
System.out.printf("Character.digit('%c', 16): %d\n", ch1, Character.digit(ch1, 16));
Character.isAlphabetic(ch1); Character.isDigit(ch1);
Character.isLetter(ch1); Character.isLetterOrDigit(ch1);
Character.isLowerCase(ch1); Character.isUpperCase(ch1); Character.isWhitespace(ch1);
Character.toLowerCase(ch1); Character.toUpperCase(ch1));
```

13. Klase-omotači za primitivne tipove, rad sa objektima tipa Float, Double. Primeri

• Klasa java.lang.Float I java.lang.Double omotavaju vrednosti primitivnih tipova float I double
• Primerci ovih klasa u sebi sadrži polje odgovarajućeg primitivnog tipa u kom se čuva vrednost objekta
• Primerci ovih klasa su imutabilni
• U ovim klasama su definisani metodi kojima se realizuju uobičajne operacije nad brojevima u pokretnom zarezu, kao I konstante koje predstavalju brojeve u pokretnom zarezu, po formatu definisanim standardom IEEE 754
• Nad primercima ovih klasa mogu da se primene metodi definisani u klasi Object
• Automatizovana je konverzija izmedju vrednosti primitivnog tipa I objekta primerka klase-omotača, I to u oba smera
Primer:
Metodi za Float I Double su isti. Samo se pise Float umesto Double.

```
Double brojPokretniZarez1 = 42.0;
Double brojPokretniZarez2 = Double.parseDouble("42");
Double brojPokretniZarez3 = Double.parseDouble("42e10");
System.out.printf("brojPokretniZarez1: %s\n", brojPokretniZarez1);

if (brojPokretniZarez1.equals(brojPokretniZarez2))
System.out.printf("prva dva broja su jednaki (equals)\n");
else
System.out.printf("prva dva boja nisu jednaki (equals)\n");
```

```
int poredjenje = brojPokretniZarez1.compareTo(brojPokretniZarez2);
if (poredjenje > 0)
System.out.printf("brojPokretniZarez1('%s') > brojPokretniZarez1('%s')\n",
brojPokretniZarez1, brojPokretniZarez2);
else if (poredjenje < 0)

System.out.printf("brojPokretniZarez1('%s') < brojPokretniZarez2('%s')\n",brojPokretniZarez1,
brojPokretniZarez2);
else
System.out.printf("brojPokretniZarez1('%s') == brojPokretniZarez2('%s')\
n",brojPokretniZarez1, brojPokretniZarez2);

System.out.printf("\n%s.intValue(): %d\n", brojPokretniZarez1, brojPokretniZarez1.intValue());
System.out.printf("%s.isInfinite(): %b\n", brojPokretniZarez1, brojPokretniZarez1.isInfinite());
System.out.printf("%s.isNaN(): %b\n", brojPokretniZarez1, brojPokretniZarez1.isNaN());
double broj1 = -5;
double min = Double.min(brojPokretniZarez1.doubleValue(), brojPokretniZarez2);
min = Double.min(min, broj1);
double max = Double.max(brojPokretniZarez1.floatValue(), brojPokretniZarez2);
max = Double.max(max, broj1);
Double sum = Double.sum(brojPokretniZarez1.floatValue(), brojPokretniZarez2);
sum = Double.sum(sum, broj1);
```

14.Klasa za matematičke funkcije, klasa Math, polja i metodi. Primeri

• Klasa java.lang.Math sadrži:
◦ matematičke konstante e i π, iskazane kao brojeve u pokretnom zarezu dvostruke tačnosti(64 bita po formatu IEEE 754)
◦ metode za realizaciju osnovnih matematičkih operacija, kao što su eksponencijalne I logaritamske funkcije., stepenovanje, trigonometrija I sl.
◦ Metode za generisanje pseudo-slučajnih brojeva iz intervala [0,1) po uniformnoj raspodeli

Primer :

```
System.out.printf("Math.E: %f\n", Math.E);
System.out.printf("Math.PI: %f\n", Math.PI);
double x = -10.7e-1;
System.out.printf("\nMath.abs(%f): %f\n", x, Math.abs(x));
System.out.printf("Math.signum(%f): %f\n", x, Math.signum(x));
System.out.printf("Math.ceil(%f): %f\n", x, Math.ceil(x));
System.out.printf("Math.floor(%f): %f\n", x, Math.floor(x));
System.out.printf("Math.round(%f): %d\n", x, Math.round(x));

double z = 6.25;
 Math.sqrt(z))
double izlozilac = 2;
 Math.pow(z, izlozilac));
Math.sin(y); Math.cos(y); Math.tan(y); Math.asin(t); Math.acos(t); Math.atan(t);
Math.exp(u); Math.log(v); Math.log10(v); Math.toDegrees(y); Math.toRadians(y);
Math.max(x, y); Math.min(x, y);
```

15.Korišćenje metoda klase Math za rad sa pseudo-slučajnim brojevima. Primeri

Primer :

```
int brojPonavljanja = 9999;
double sumaSlucajnih = 0;
for (int i = 1; i <= brojPonavljanja; i++) {
double slucajan = Math.random();
sumaSlucajnih += slucajan;
System.out.printf("%d. Math.random(): %f\n", i, slucajan);
}
```

```
System.out.printf("Prosek slucajnih: %f\n\n", sumaSlucajnih / brojPonavljanja);

int pocetak = 40;
int kraj = 44;
sumaSlucajnih = 0;
for (int i = 1; i <= brojPonavljanja; i++) {
int slucajan = slucajaCeoIzIntervala(pocetak, kraj);
sumaSlucajnih += slucajan;
System.out.printf("%d. slucajan iz [%d,%d]: %d\n", i, pocetak, kraj, slucajan);
}
System.out.printf("Prosek slucajnih: %f\n\n", sumaSlucajnih / brojPonavljanja);
}
static int slucajaCeoIzIntervala(int intervalOd, int intervalDo) {
double slucajan = Math.random();
return (int) (slucajan * (intervalDo - intervalOd) + intervalOd);
```

16.Klase za rad sa datumima i vremenima, merenje proteklog vremena. Primeri

• To su klase java.time.LocalDate, java.time.LocalTime I java.time.LocalDateTime. Koje služe za rad samo sa datumima, samo sa vremenima u danu I sa parovima datum-vreme.
◦ Klase su uvedene realtivno skoro, ranije se koristila klasa java.util.Date, sa kojom je bilo problema
◦ kako se klase ne nalaze u paketu java.lang, pa se prilikom referisanja na njih mora koristiti puno ime tih klasa
• Sadrže metode za realizaciju osnovnih operacija nad datum-vremenima kao što su poredjenja datum-vremena, odredjivanje datum-vremena na osnovu postojećeg, merenje proteklog vremena izmedju dva datuma-vremena I sl.

Primer:

```
java.time.LocalDateTime datumVreme1 = java.time.LocalDateTime.now();
java.time.LocalDateTime datumVreme2 = java.time.LocalDateTime.parse("2019-02-
03T23:15:45");
java.time.LocalDateTime datumVreme3 =
java.time.LocalDateTime.of( 2019,02,03,23,15,45);

System.out.println("datumVreme1: " + datumVreme1);
System.out.println("datumVreme2: " + datumVreme2);
System.out.println("datumVreme3: " + datumVreme3);

java.time.format.DateTimeFormatter formater =
java.time.format.DateTimeFormatter.ofPattern("dd.MM.YYYY. HH:mm:ss.n");
System.out.printf("\ndatumVreme1 (custom format): %s\n", datumVreme1.format(formater));
java.time.format.DateTimeFormatter formater2 =
java.time.format.DateTimeFormatter.ISO_DATE_TIME;
System.out.printf("\ndatumVreme1 (ISO format): %s\n", datumVreme1.format(formater2));
if (datumVreme1.equals(datumVreme2))
System.out.println("datumVreme1 i datumVreme2 su jednaki (equals)");
else
System.out.println("datumVreme1 i datumVreme2 nisu jednaki (equals)");
System.out.println("\nporedjenje datumVreme1 i datumVreme2 (compare) daje rezultat "
+ datumVreme1.compareTo(datumVreme2));

if( datumVreme1.isBefore(datumVreme2))
System.out.println("prvo datumVreme je pre drugog");
else
System.out.println("prvo datumVreme nije pre drugog");
if( datumVreme1.isAfter(datumVreme2))
System.out.println("prvo datumVreme je posle drugog");
else
System.out.println("prvo datumVreme nije posle drugog");
```

datumVreme1.minusDays(2).format(formater); datumVreme1.plusDays(2).format(formater);
datumVreme1.minusWeeks(2).format(formater);
datumVreme1.plusWeeks(2).format(formater);
datumVreme1.minusMonths(2).format(formater);

I tako sve isto I za godine, sate, sekunde minute....
Ne menja se taj konkretan objekat, nego se pravi novi koji je promenjen (imutabilni objekti)...

datumVreme1.withDayOfMonth(17).format(formater));
System.out.printf("datumVreme1 sa sto sedamnaestim danom u godini: %s\n",
datumVreme1.withDayOfYear(117).format(formater));

Ostale primere za DatumVreme pogledajte ovde:
https://github.com/MatfOOP/OOP/blob/master/predavanja/primeri-java/src/
P050703DatumVremeZ01LocalDateTimePokretanje.java

Ima I primera samo Datuma, tj samo vremena na ova dva linka:
https://github.com/MatfOOP/OOP/blob/master/predavanja/primeri-java/src/
P050701DatumVremeZ01LocalDatePokretanje.java

https://github.com/MatfOOP/OOP/blob/master/predavanja/primeri-java/src/
P050702DatumVremeZ01LocalTimePokretanje.java

17.Skeneri, klasa Scanner, skeniranje podataka iz niske. Primeri

• Klasa java.util.Scanner predstavlja jednostavni skener teksta koji može parsirati primitivne tipove I niske korišćenjem regularnih izraza
• Kako se klasa java.util.Scanner ne nalazi u paketu java.lang, pa se prilikom referisanja na nju mora koristiti njeno puno ime
• Slede važne osobine klase java.util.Scanner:
◦ Za razliku od matematičkih f-ja, pre skeniranja ulaza je neophodno kreirati primerak klase Scanner
◦ Napravljeni objekat pri skeniranju razdvaja ulaz na tokene korišćenjem obrasca za razdvajanje, podrazumevano postavljenog na beline
◦ Operacija skeniranja moze da blokira čekajući na ulaz
Primer:

```
String izvor = "paja patak\n miki maus 12 23\n dusko dugousko";
java.util.Scanner skener = new java.util.Scanner(izvor);
StringBuilder rez = new StringBuilder();
while (skener.hasNext()) {
String s = skener.next();
rez.append(s);
rez.append("|");
}
skener.close();
System.out.printf("nalepljene reci niske \n \"%s\":\n%s\n\n", izvor, rez);

izvor = "1 2 3 4 -3 -8 0";
skener = new java.util.Scanner(izvor);
int suma = 0;
while (skener.hasNext()) {
int i = skener.nextInt();
suma += i;
}
skener.close();
System.out.printf("suma celih brojeva niske\n \"%s\":\n%d\n\n", izvor, suma);
```

```
izvor = "1 2 3.5 4.3 -3e1";
skener = new java.util.Scanner(izvor);
double suma1 = 0;
while (skener.hasNext()) {
double x = skener.nextDouble();
suma1 += x;
}
skener.close();
System.out.printf("suma brojeva niske \"%s\":\n%f\n\n", izvor, suma1);
```

18.Skeneri, klasa Scanner, skeniranje podataka sa standardnog ulaza. Primeri

```
static long faktorijel(int n) {
if (n == 0 || n == 1)
return 1;
return n * faktorijel(n - 1);
}

public static void main(String[] args) {
System.out.printf("Unesite broj:\n");
java.util.Scanner skener = new java.util.Scanner(System.in);
int n = skener.nextInt();
skener.close();
for (int i = 0; i <= n; i++)
System.out.printf("%3d! = %d\n", i, faktorijel(i));
}
```

V Nizovi u Javi

Pogledati prezentaciju, ima dosta dijagrama

1.Deklaracija i inicijalizacija niza. Primeri

• Niz se definiše kao grupa promenjivih istog tipa koje se pojavljuju pod zajedničkim imenom
• Nizovni tip podataka u Javi ima sledeća svojstva:
1. sadrži linearno uredjen, uanpred poznat, broj članova,
2. svi članovi su istog tipa I imaju zajedničko ime; članovi mogu biti primitivnog ili objektnog tipa,
3. svakom članu pristupa se pomoću zajednickog imena niza I indeksa člana,
4. svi indeksi su celobrojnog tipa,
5. svi članovi niza se tretiraju kao posebne promenljive(nazivaju se I indeksnim promenljivim).
• Nizovni tip u Javi je uvek objektni tip, tj. niz je uvek objekat.
• Dodeljivanje početnih vrednosti (inicijalizacija) članovima niza vrši se u delu deklaracije nazvanom nizIniz.(Prezentacija zbog dijagrama)

- Sintaksa naredbe za alokaciju memorije:(Dijagram)

Deklarisanje nizova(dva načina):
I način
int niz[];

II način
int[] niz;
Kreiranje nizova sledi u sledeca dva primera(1. Pomoću operatora new, 2. nabrajanjem članova)

Primer 1:
int[] niz = {24, 48, 96, 192, 36, 72, 144, (int) (Math.random() * 56) * 4};

```
for (int i = 0; i < niz.length; i++)
System.out.print(niz[i] + " ");
System.out.println();
```

Primer 2:

```
 int[] niz = new int[10];
 for (int i = 0; i < niz.length; i++)
 niz[i] = i
```

2.Nizovna promenljiva i indeksna promenljiva. Primeri

- Nizovna promenljiva se deklariše u delu programa za deklaraciju.
- Pri tome se mogu zadati i vrednosti članovima niza.
- Elementima niza(indeksnim promenljivima) pristupa se pomoću indeksa
- Sa indeksnim promenljivima operiše se na isti način kao I sa promenljivima bez indeksa.
- Dakle, vrednosti indeksnih promenljivih se mogu menjati pomoću naredbe dodeljivanja I koristiti u izrazima

Stvarno nzm sta više ovde treba,niti koji primeri.

3.Brojački I kolekcijski for ciklus. Primeri

Brojački:

Brojački ciklus je posebna varijanta uslovnog ciklusa. Ovakav ciklus postoji u velikom broju programskih jezika i obično podrazumeva "brojanje" kroz neku promenljivu od početne do krajnje vrednosti. Koliko se različitih vrednosti dodeli toj promenljivoj, toliko puta se izvrši i sam ciklus.

Primer:

```
int[] niz = {24, 48, 96, 192, 36, 72, 144};
for (int i = 0; i < niz.length; i++)
System.out.print(niz[i] + " ");
System.out.println();
```

Kolekcijski:

- Kolekcijska naredba for služi za prolazak kroz kolekciju/niz
- Pritom promenljiva u naredbi for ne predstavlja brojač, već redom uzima vrednosti elemenata niza tj. Kolekcije.
- Kolekcijska naredba for ima sledecu sintaksu:(Prezentacija zbog dijagrama)

Primer kolekcijskog for ciklusa:

```
int[] niz = {24, 48, 96, 192, 36, 72, 144};
for(int a : niz)
System.out.printf("%d ",a);
```

4.Argumenti komandne linije kod ulazne tacke programa, static metoda main. Primeri

- Argumenti komandne linije omogućavaju da se prilikom pokretanja java programa, tom programu proslede parametri.
- Ulazna tačka programa, tj staticki metod main ima sledecu sinatksu:

```
public static void main(String[] argumentiKomandneLinije){

        //naredbe

    }
```

- Naredbe u telu programa mogu referisati na jednodimentionalni niz niski argumentiKomandneLinije koji će sadržavati argumente specificirane u komadnoj liniji
    - element niza sa indeksom 0 predstavlja prvi prosledjeni argument, sa indeksom 1 drugi itd.
    - Prilikom parsiranja komadne linije, razmak tj ‘ ‘ razdvaja, a ako je potrebno da parametar sadrži razmak ‘ ‘, koristi se ‘”’

Primer:

```
public static void main(String[] argumentiKomandneLinije) {

System.out.printf("Argumenti komandne linije:\n");

for (String argument : argumentiKomandneLinije)

        System.out.printf("\"%s\" \n", argument);

        System.out.printf("\n");

}
```

5.Nizovi nizova, dvodimenzionalni niz. Primeri

- Dvodimenzionalni niz čine elementi istog tipa I svakom od njih pristupa se pomoću imena niza I dva indeksa
- Dvodimenzionalni niz predstavlja jednodimenzionalni niz čiji elementi su jednodimenzionalni nizovi

Deklaracija:

```
int a[][];

//ili int [][] a;
```

- Nakon deklaracije nizovne promenljive, kreiramo nizovni objekat,tj. Vršimo rezervisanje memorijskog prostora za smeštanje članova niza

```
a=new int[50][100];
```

- A možemo I jednom naredbom izvršiti I deklaraciju I rezervisanje memorijskog prostora

```
int a[][] = new int[50][100];
```

Primer:

```
double[][] g = {{1.5, 2, 3}, {4, 5, 6}};
if (g == null) {
System.err.println("GRESKA!");
return;
}
System.out.println("Elementi matrice su");
for (int i = 0; i < g.length; i++) {
for (int j = 0; j < g[i].length; j++)
System.out.printf("%15.4f", g[i][j]);
System.out.printf("\n");
}
```

6.Argumenti funkcija promenljive dužine. Primeri

- Novije verzije programskog jezika Java dopuštaju da parametri funkcije ne budu strogo fiksirani, već da jedna ista funckija može biti pozvana I sa različitim brojem argumenata istog tipa:

- Tada je parametar funckije niz, a prilikom poziva se uzastopni argumenti istog tipa automatski konvertuju u niz
- Nalog za automatsku konverziju uzastopnih argumenata datog tipa u niz se postiže deklarisanjem pomoću "..."
- Ovako definisan parametar funckije mora da bude poslednji u spisku parametara.
- Ulazna tačka programa može da bude realizovana kao metod sa promenljivim brojem argumenata

Primeri:

```
// static int zbir(int a1, int a2) {
// return a1 + a2;
// }
//
// static int zbir(int a1, int a2, int a3) {
// return a1 + a2 + a3;
// }
```

// Umesto da pisemo funkciju za svaki moguci poziv iz maina, tj. Ako unesemo 2 ili 3 ili 4 ili 5 argumenta, možemo samo napisati donju funkciju sa "...",i radice za bilo kakav poziv sa tim tipom.

```
static int zbir(int... a) {
int ret = 0;
for (int x : a)
ret += x;
return ret;
}

public static void main(String[] args) {
System.out.println(zbir(1, 2));
System.out.println(zbir(1, 2, 3));
System.out.println(zbir(1, 2, 3, 4, 5, 6));
System.out.println(zbir(1));
System.out.println(zbir());
int[] a = {1, 2, 3, 4};
System.out.println(zbir(a));
}
```

5.7.Argumenti funkcija promenljive dužine, preporuke za korišćenje. Primeri

• Novije verzije programskog jezika Java dopuštaju da parametri funkcije ne budu strogo fiksirani, već da jedna ista funkcija može biti pozvana i sa različitim brojem argumenata istog tipa:

• Tada je parametar funkcije niz, a prilikom poziva se uzastopni argumenti istog tipa automatski konveruju u niz

• Nalog za automatsku konverziju uzastopnih argumenata datog tipa u niz se postiže deklarisanjem pomoću …

• Ovako definisan parametar funkcije mora da bude poslednji u spisku parametara

• Ulazna tačka progama može da bude realizovana kao metod sa promenljivim brojem argumenata

6. Klase, polja, metodi i objekti u Javi

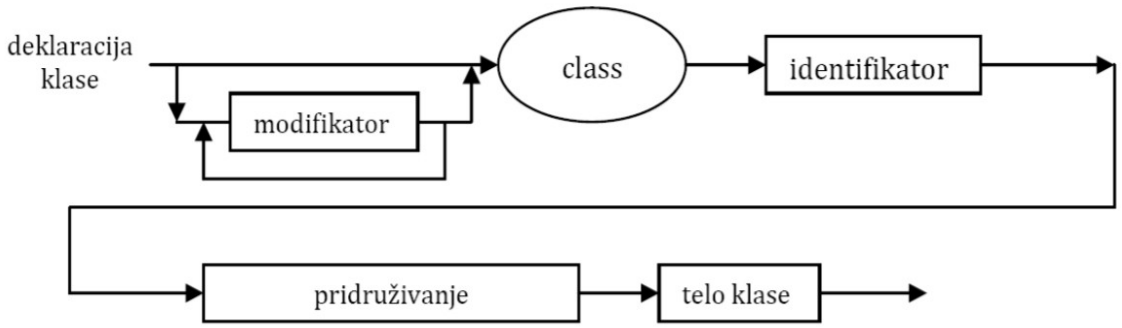
1. Klase u Javi. Kreiranje objekta - primerka date klase. Primeri

- Ceo Java kod se nalazi unutar klasa.

- Jednoj klasi odgovara jedna datoteka sa ekstenzijom .java. Naziv klase i naziv datoteke treba da budu isti.

- Klasa se definiše korišćenjem ključne reči class.

- Na primer, naredbom: class Duzina je definisana nova klasa sa nazivom Duzina, a blok (tj. kôd između vitičastih zagrada), koji sledi, opisuje kakva je struktura novonapravljene klase.

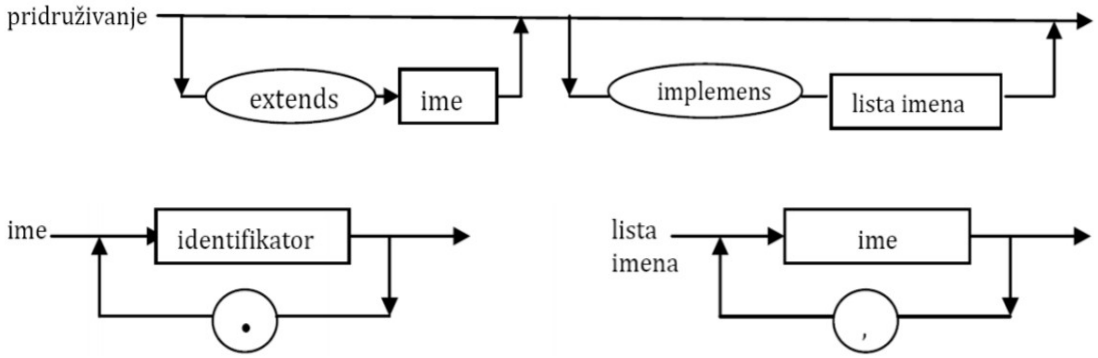


-Modifikator je rezervisana reč koja menja osnovno značenje neke konstrukcije u Javi:

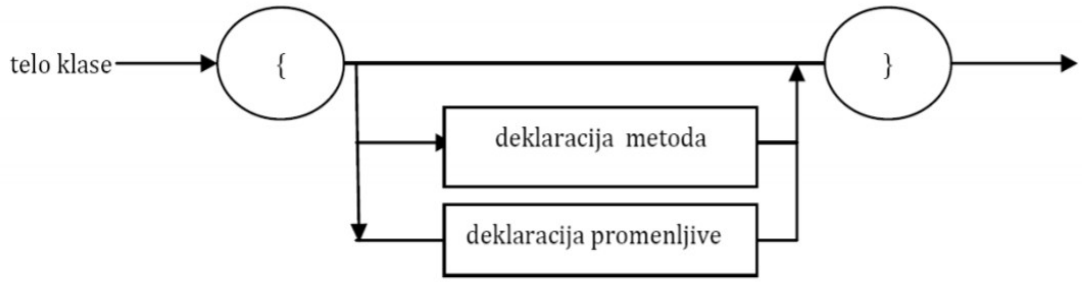
<modifikator> ::= public | private | protected | abstract | final | static | native | strictfp | synchronized | transient | volatile

-Metalingvistička promenljiva pridruživanje određuje iz koje klase je data klasa izvedena, odnosno, da li data klasa implementira neke interfejse:

-Metalingvistička promenljiva ime može biti indentifikator ili identikatori razdvojeni tačkom, dok je lista imena niz imena razdvojenih zarezima



Telo klase predstavlja blok u kom se navode razne deklaracije



-Kreiranje objekta-

-Za kreiranje primeraka (konkretnih objekta neke klase) koristi se operator new.

Primeri:

String niska = new String();

Random slicajan = new Random();

Knjiga x = new Knjiga();

Point tacka1 = new Point(20 30);

- Ako objektna proemnljiva ne referiše na objekat, tada se njoj dodeljuje vrednost null.

Primeri:

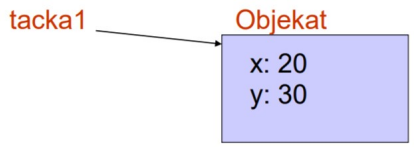
String niska = null;

Random slicajan = null;

Knjiga x = null;

Point tacka1 = null;

-Kada se kreira objekat, promenljiva kojoj se dodeljuje vrednost novokreiranog objekta predstavlja pokazivač (referencu) na taj objekat.



-Može biti i više promenljivih koje referišu na isti objekat.

- Delom memorije koji se koristi za objekte Java virtuelna mašina automatski upravlja, i tu važnu ulogu ima tzv. sakupljač otpadaka.

-Komponente objekta su: promenljive primerka (ili polja) i metodi.

-Komponentama objekta pristupa se preko tzv. tačka-notacije.

2.Klase i objekti – polja. Primeri.

Postoji nekoliko vrsta promenljivih:

• promenljive deklarisane u samoj klasi, koje predstavljaju clanove-podatke unutar klase, tj. opisuju atribute objekta koji je primerak date klase – to su **polja**

• promenljive deklarisane u telu metoda ili bloku – to su **lokalne promenljive**

• promenljive deklarisane u zaglavlju metoda – to su **parametri** (formalni argumenti)

Deklaracija polja se sastoji od tri komponente: modifikatora (koji se opciono pojavljuju), tipa polja (tip) i imena polja (identifikator). Modifikatori omogućuju da se podesi vidljivost date promenljive člana (tj. polja), da se odredi da li se radi o promenljivoj primerka (instancnoj promenljivoj) ili o klasnoj (statiˇckoj) promenljivoj, kao i da li vrednost promenljive postaje nepromenljiva neposredno po kreiranju (finalna tj. konstantna). U tom smislu, razlikuju se instancna promenljiva, klasna promenljiva i konstanta.

3.Polje primerka. Pristup polju u primerku date klase. Primeri.

To su promenljive definisane u okviru klase (polja klase tj. promenljive članovi klase) takvi da svaki od kreiranih objekata date klase sadrži sopstveni primerak te promenljive. Promenljivoj primerka se može pristupiti samo ako se referencira primerak klase koji sadrži tu promenljivu. Promena vrednosti jedne promenljive primerka nema uticaja na ostale.

```
class Point extends GeometryObject{
        int x;
        int y;
}
```

Kao što je ranije istaknuto, komponentama objekta pristupa se preko tzv. tačka-notacije i to bez obzira da li se radi o instancnim promenljivima ili o instancnim metodama.

4.Klasna polja. Primeri

Klasa sadrži samo jednu kopiju klasne promenljive i ta promenljiva je deljena među svim objektima date klase. Ona postoji čak iako se ne kreira nijedan primerak date klase. Ona pripada klasi i nju mogu svi da referenciraju, a ne samo primerci date klase. Klasna promenljiva se deklariše korišćenjem modifikatora static.

Primer:

```
class ClanPorodice{
static String prezime="Jankovic"; // klasna promenljiva
int uzrast; // instancna promenljiva.
String ime; // instancna promenljiva
ClanPorodice otac = new ClanPorodice("Nikola", 52);
ClanPorodice sin = new ClanPorodice("Petar", 21);
}
```

Za pristup klasnoj promenljivoj se koristi tačka-notacija, pri čemu se kao primalac poruke može koristiti ime klase ili ime neke instance klase. Preporučuje korišćenje se imena klase za pristup klasnoj promenljivoj. Primer:

```
class ClanPorodice {
static String prezime = "Jankovic";
String ime;
int godine;
//i kreiran primerak sin:
ClanPorodice sin = new ClanPorodice();
//klasnoj promenljivoj prezime mozemo pristupiti na sledece nacine:
System.out.println( "Porodicno prezime je: " + sin.prezime);
System.out.println( "Porodicno prezime je: " + ClanPorodice.prezime);
}
```

5.Opseg vazenja za promenjlive i polja. Primeri

Opseg važenja promenljive je deo programa u kojem ime promenljive može da se koristi. Kod lokalnih promenljivih, opseg važenja promenljive je blok u kojem je definisana od mesta gde je definisana. Dakle, lokalnoj promenljivoj se može pristupati u naredbama bloka u kom je ta promenljiva definisana, a koje slede iza naredbe definicije.

Primer:

```
{
        int a=1; //ovde se može pozivati a, ali ne i b!
{
//ovde se može pozivati a, ali ne jos b!
int b=2;
// ovde se mogu pozivati a i b!
}
// ovde se može pozivati a, ali ne i b!
}
        // ovde se ne mogu koristiti niti a, niti b!
```

Klasnim promenljivima se može pristupati u svakom trenutku rada programa. Promenljivima primerka se može pristupati svo vreme dok postoji objekat – primerak date klase.

Primer:

```
class Opseg{
int n, probna = 10;
void stampaProbnu(){
int probna = 30;// this.probna
```

```
System.out.println("Lokalna probna = "+probna);
}
        n=probna;
}
```

Ako treba da se unutar metoda pristupi sakrivenom polju probna, to se postiže pomoću this.probna.

6.Klase i objekti – metodi. Primeri

--

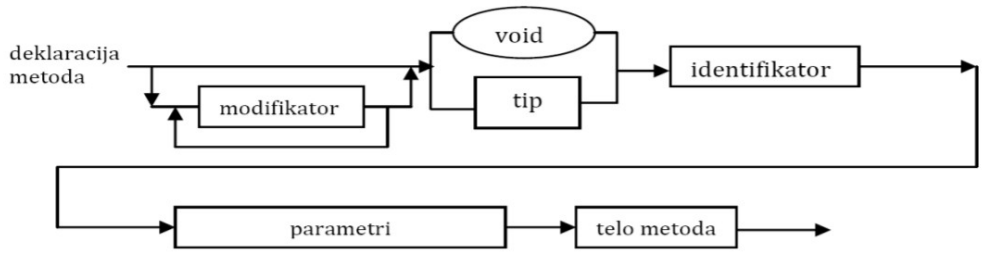
7. Definicija metoda. Parametri metoda, potpis metoda, telo metoda, povratna vrednost metoda. Primeri(ista su pitanja) -Metodi se pojavljuju u telu klase.

-Oni sadrže deklaracije lokalnih promenljivih i druge Java naredbe koje se izvršavaju pri pozivu metoda.

-Argumenti metoda se navode u zagradama iza imena metoda (broj argumenata može biti nula).

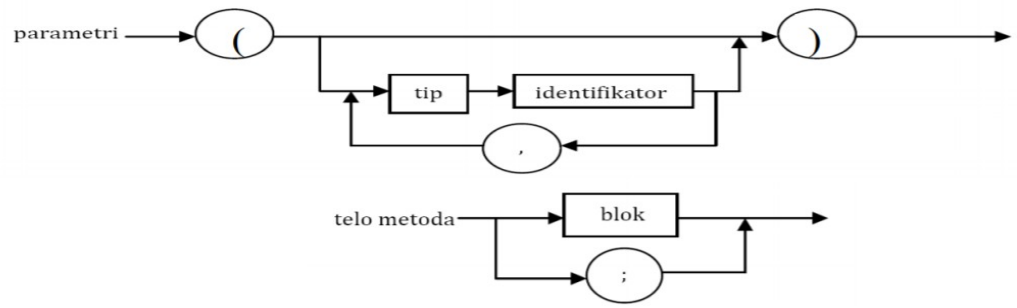
-Prilikom deklaracije metoda u klasi, navodi se tip argumenata.

-Pri pozivu metoda navode se stvarni argumenti.



-Metod je samostalni blok koda koji ima svoje ime i svojstvo višestruke upotrebljivosti.

- Parametri (ili formalni argumenti) omogućavaju da se proslede vrednosti u metod:

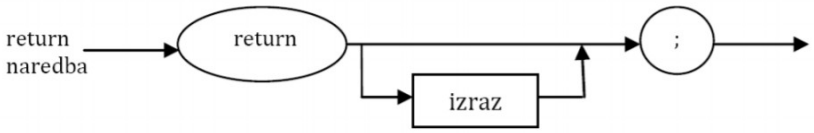


-Metalingvistička promenljiva blok je definisana u jednoj od prethodnih prezentacija.

- Metod može vratiti vrednost, i u tom slučaju je potrebno navesti tip povratne vrednosti.

- Ukoliko metod ne vraća vrednost, njegov tip je void.

- Povratna vrednost iz metoda se vraća preko return naredbe:



-Sintaksa izraza je definisana na slajdovima prethodne prezentacije.

• Metodi mogu biti:

1. instancni (metod primerka)

2. klasni (statički).

• Ključna reč static u deklaraciji ukazuje da je metod klasni

• Klasni metodi se ne odnose na instancne promenljive, tj. ne mora postojati ni jedna instanca, a metod se može koristiti!

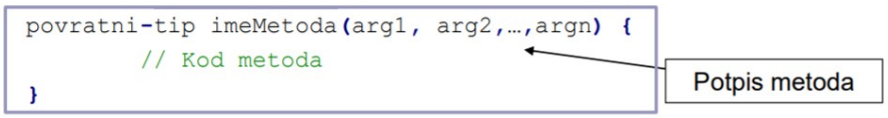
• U telu klasnog metoda se ne može referisati na promenljive primerka date klase, već samo na klasne promenljive za datu klasu.

• Primer. Metod main(String args[]), koji je neophodan u aplikacijama, mora biti klasni.

-Kod navođenja formalnih argumenata, za svaki argument mora se navesti tip argumenta.

-Ime metoda zajedno sa tipom i redosledom parametara čini potpis metoda.

-Potpis dva metoda u klasi mora biti različit, kako bi kompajler mogao da odredi koji metod se poziva.



8. Poziv metoda. Argumenti metoda, konstantni parametri, supstitucija parametara pri pozivu metoda. Primeri

-Pozivom metoda se prelazi na izvršavnje pozvanog metoda, i kada se metod bude izvršio, on vraće rezultat i izvaršavanje programa se nastavlja dalje, od mesta gde je izvršen poziv

-Prilikom poziva metoda, na mesto formalnih argumenata (parametara), navode se stvarni argumenti.

-Prilikom poziva metoda u Javi, argumenti poziva moraju da budu sa parametrima metoda po broju i tipu, inače će prevodilac prijaviti grešku.

Pri pozivu metoda:

- U slučaju argumenata koji predstavljaju objekte, prosleđuje se vrednost te promenljive, tj. vrednost reference na objekat, a ne sam objekat!

- Za argumente metoda primitivnog tipa, tu promenljiva sadrži podatak, i ta vrednost, tj. podatak se prosleđuje metodu.

Dakle, u oba slučaja se vrši supstitucija vrednosti, samo što je kod objektnih argumenata u pitanju vrednost reference.

U naredbama u telu metoda se mogu koristiti četiri potencijalna izvora podataka:

1. formalni argumenti metoda,

2. instancne i klasne promenljive,

3. lokalne promenljive, definisane u telu metoda

4. vrednosti koje vraćaju drugi metodi koji su pozvani u tekućem.

9.Sakrivanje polja parametrima, referenca this.Primeri

U okviru metoda primerka ili konstruktora, promenljiva this predstavlja referencu na sam taj objekat, tj. na objekat primalac poruke čiji se metod ili konstruktor upravo izvršava. Korišćenjem this se može referisati na ma koje polje tekućeg objekta nad kojim se poziva metod primerka ili konstruktor.

Primer:

```
public void brisi(){
this.x =0;
this.y=0;
}
```

Ovde se ključna reč this može izostaviti.

Najčešći razlog za korišćenje ove promenljive je to što polje klase bude sakriveno parametrom metoda ili parametrom konstruktora.

Primer:

```
public void pomeri(int x, int y){
this.x = x;
this.y = y;
}
```

Ovde je ključna reč this neophodna da podvuče razliku izmedu instancnih promenljivih i parametara metoda. Svaki od argumenta metoda sakriva po jedno polje objekta, tako da je unutar metoda x oznaka lokalne kopije prvog argumenta. Da bi se referisalo na polje x primerka klase Point, unutar metoda se mora koristiti this.x.

Primeri:

- U naredbama koje slede se koristi ključna reč this:

```
t = this.a; // a -instancna promenljiva tekućeg objekta
this.prvimetod(this); // poziva se metod
return this; //vraća se tekući objekt iz metoda

//U nekim od navedenih slučajeva može se izostaviti:
t = a;
prviMetod(this);

//U nekima ne moze:
return this;
```

10.Preopterćenje metoda. Pozivanje drugog preopterećenog metoda, referenca this. Primeri

Definisanje metoda sa istim imenom, ali različitim parametrima naziva se preopterećenje metoda.

Primer:

```
class Pravougaonik {
int x1=0;
int x2 = 0
int y1 = 0;
int y2 = 0;


Pravougaonik gradi(int x1, int y1, int x2, int y2) {
this.x1 = x1;
this.y1 = y1;
this.x2 = x2;
this.y2 = y2;
return this;
        }
}


Pravougaonik gradi(Point goreLevo, Point doleDesno) {
x1 = goreLevo.x;
y1= goreLevo.y;
x2 = doleDesno.x;
y2 = doleDesno.y;
return this;
}


Pravougaonik gradi(Point goreLevo, int l, int h){
x1 = goreLevo.x;
y1 = goreLevo.y;
x2 = goreLevo.x + l;
y2 = goreLevo.y + h;
return this;
```

```
}

void printPrav(){
        System.out.println("Pravougaonik :["(+x1+","+y1+ "),("+x2+","+y2+")]");
}


public static void main (String args[]){
Pravougaonik p = new Pravougaonik();
p.gradi(10,20, 30,40);
p.printPrav();
p.gradi(new Point(10,10), new Point(30,30));
p.printPrav();
p.gradi (new Point (10,10), 20, 30);
p.printPrav();
}
```

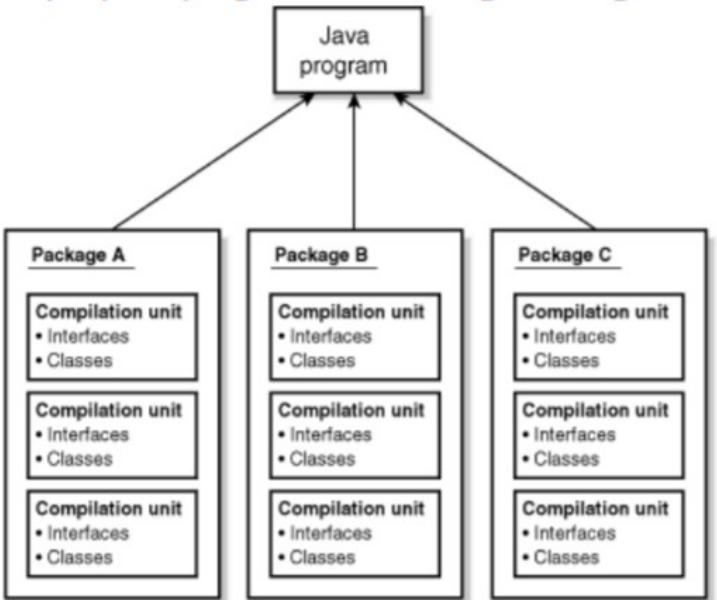
11. Klasni metodi. Primeri

Nisam našao ništa za ovo pitanje. Uglavnom se svodi na prethodno, jer su klasni metodi zapravo statički metodi. Dakle, main funkcija je jedan klasni metod jer je statički, tj. ne instancira se.

12.Organizacija klasa po paketima. Definisnje paketa, uvoz klasa iz paketa. Primeri

-Programeri grupišu slične tj. povezane tipove u pakete i na taj način izbegavaju konflikte u imenima i kontrolišu pristup.

-Paket je grupa povezanih tipova (klasa, interfejsa, enumerisanih tipova i tipova notacije) za koju je obezbeđuje zaštita pri pristupu i upravljanje prostorom imena.



Razlozi za pakovanje klasa i interfejsa u pakete su :

1. Lakše određivanje da li su tipovi povezani.

2. Lakše se mogu pronaći traženi tipovi.

3. Nema imenskih konflikta sa drugim tipovima istog naziva, jer paket kreira novi prostor imena.

4. Dopuštanje da tipovi unutar paketa imaju neograničen pristup jedan drugom.

Naime, podaci u okviru objekata se podrazumevano definišu tako damogu biti „vidljivi“ (tj. dostupni) iz svih klasa koje se nalaze u istom paketu u kom se nalazi ta klasa.

• Proces kreiranja sopstvenih paketa se može opisati u tri koraka.

• Prvi korak je izbor imena paketa.

Preporuka proizvođača: korišćenje naziva Internet domena sa elementima poređanim po obrnutom redosledu.

Na primer, ako je naziv domena: math.rs, naziv paketa bi trebalo da počne sa rs.math

Na taj način se postiže da naziv paketa bude jedinstven.

Po konvenciji, nazivi paketa počinju malim slovima.

• Drugi korak je kreiranje strukture direktorijuma (fascikli, foldera).

Ako je naziv paketa iz jednog dela (nema tačaka u nazivu), naziv direktorijuma poklapa se sa nazivom paketa.

Ako se naziv paketa sastoji iz više delova (odvojenih tačkom), tada za svaki deo treba formirati poddirektorijum.

Na primer, za rs.math.oop, glavni direktorijum treba da se zove rs, njegov poddirektorijum math i u njemu treba da postoji direktorijum math.

U svaki od ovih direktorijuma se mogu ubaciti datoteke, odnosno klase, interfejsi itd.

• Treći korak je dodavanje package naredbe

Ovo treba da bude prva naredba Java programa, dakle, pre prve naredbe import.

Na primer, ako je naziv paketa rs.math.oop, na početku svake datoteke u tom paketu mora pisati:

package rs.math.oop;

Integrisana razvojna okruženja omogućuju da se sva tri prethodno opisana koraka realizuju jednom opcijom, i sama predlažu rešenje u situacijama kada neusklađenost putanje datoteka i naziva paketa dovede do greške.

• Da bi moglo da se rukuje ugrađenim Java klasama, mora se znati gde se klase nalaze u okviru sistema.

• Mesto gde se klase nalaze određuje se preko komande operativnog sistema CLASSPATH.

• Ovom komandom se definiše putanja do direktorijuma u kom Java okruženje za izvršavanje traži klase.

• Ako CLASSPATH nije definisan, podrazumeva se direktorijum java\lib u konkretnoj instalaciji Jave.

• Kombinovanjem putanje date u CLASSPATH-u i naziva paketa, Java virtuelna mašina pronalazi klase sa kojima se operiše.

• Razvojna okruženja dopuštaju podešavanje putanje za klase kroz korisnički interfejs, u okviru definisanja tzv. projekta

13. Uvoz klasnih metoda. Primeri

```
import static java.lang.Math.*; // ovim uvozimo sve staticke metode klase Math
class Test2 {
    public static void main(String[] args)
    {
        System.out.println(sqrt(4)); // sada umesto Math.sqrt koristimo samo
sqrt
        System.out.println(pow(2, 2));
        System.out.println(abs(6.3));
    }
}
```

Ovo nije preporucljivo jer smanjuje citljivost, moze doci do mesanja metoda trenutne klase sa uvezenim

14.Klase - nasleđivanje. Primeri

-Nasleđivanjem se opisuje odnos «jeste».

-Mehanizam nasleđivanja omogućuje da primerci podklase (uz određena ograničenja) imaju pristup (kao da sadrže) poljima i metodima definisana u svim natklasama

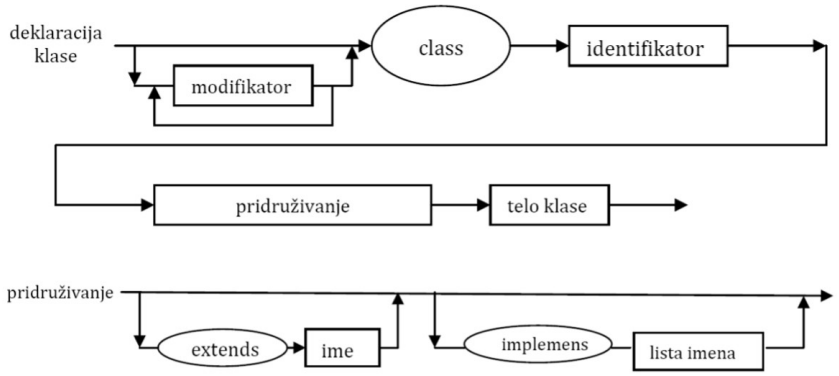
-Programski jezik Java karakteriše jednostruko nasleđivanje.

-Svaka klasa ima tačno jednu nadklasu

-Graf nasleđivanja je drvo, u čijem korenu se nalazi klasa Object (iz paketa java.lang)

-Sve Java klase (čak i one kod kojih to nije eksplicitno zapisano) su podklase klase Object

Sintaksni dijagram za deklarisanje klasa ima sledeći oblik:



15.Klase - nasleđivanje. Tip objektne promenljive u vremenu izvršavanja. Primeri

Java je strogo tipiziran programski jezika, pa svaka promenljiva, uključujući i promenljive objektnog tipa mora imati svoj tip.

Provera da li dati objekat pripada datoj klasi realizuje se pomoću operatora instanceof.

Primer.

”bilo sta” instanceof String // true

Point tacka = new Point(10,10);

tacka instanceof String // false

Ova provera se realizuje u vremenu izvršavanja programa i odnosi se na aktuelnu vrednost objekta.

Promenljiva opštijeg tipa (tj. promenljiva tipa nadklase) može referisati na objekte konkretnijeg tipa (tj. objekte primerke potklase).

Kao što je već istaknuto, operator instanceof proverava da li je objekat na koji referiše dati izraz primerak klase (u vremenu izvršavanja, a ne tokom prevođenja).

Operator instanceof vraća true ako je dati objekat primerak neke klase, a isti rezultat daje i ako je dati objekat primerak ma koje od natklasa date klase.

Primer. Neka je klasa Kamion potklasa klase Vozilo

Kamion x = new Kamion();

boolean y = x instanceof Kamion; // true

boolean z = x instanceof Vozilo; // true

boolean t = x instanceof String; // false

boolean u = x instanceof Object; // true

16. Klase - nasleđivanje. Prevazilaženje polja i metoda u podklasama. Primeri

• Proces definisanja metoda u potklasi koji ima isti potpis kao i metod u natklasi naziva se prevazilaženje metoda.

• Omogućuje promenu ponašanja ili dodavnje novog ponašanja u podklasama.

• Sposobnost podklase da prevaziđe metod omogućuje da neka klasa:

1. nasledi „dovoljno blisku“ nadklasu

2. da po potrebi modifikuje njeno ponašanje.

• Novi metod ima isto ime, isti broj i tipove parametara i vraće rezultat istog tipa kao metod nadklase koji se prevazilazi.

• Novi metod može da vrati i podtip tipa koji vraća metod nadklase, što se naziva kovarijantni tip rezultata

• Kada se prevazilazi metod poželjno je koristi anotaciju @Override.

• Na taj način se daje na znanje prevodiocu da želimo da uradimo prevazilaženje i on nas upozorava ukoliko metod sa takvim potpisom ne postoji u nadklasi.

```
public class Line {

void drawLine(){

//implementacija crtanja linije u nadklasi

}

}


public class DottedLine extends Line{

@Override

void drawLine() {

//prevazilazimo implementaciju iz nadklase

}

}
```

17.Klase - nasleđivanje. Pristup poljima nadklase u metodima. Primeri

Mehanizam nasleđivanja omogućuje da primerci podklase (uz određena ograničenja) imaju pristup (kao da sadrže) poljima i metodima definisana u svim natklasama

Primer:

super.label = “ zdravo svete”

18.Klase - nasleđivanje. Pozivanje metode nadklase. Primeri

• Pozivanje odgovarajućeg metoda definisanog u natklasi date klase, a koji je prevaziđen u potklasi, realizuje se korišćenjem ključne reči super.

Primer.

```
void mojMetod( int x, int y) {

// neke naredbe

super.mojMetod(x,y);

// jos naredbi

}

void mojMetod2( int x, int y) {
```

```
// neke naredbe
mojMetod(x,y);
// poziva se metod definisan u istoj ovoj klasi
super.mojMetod(x,y);
// poziva se metod definisan u nadklasi
// jos naredbi
}
```

• Ako u natklasi nije definisan konstruktor, tada biva pozvan podrazumevani implicitni konstruktor natklase.

• Eksplicitno pozivanje konstruktora natklase se može realizovati ključnom rečju super kao u slučaju poziva metoda.

super(arg1, arg2, …)

Primer.

```
class LabeledPoint extends Point {
String label;
LabeledPoint (int x, int y, String label) {
super(x,y);
this.label = label;
}
}
```

19.Modifikatori za kontrolu pristupa poljima. Primeri

-To su specijalne ključne reči koje menjaju ponašanje klasa, metoda, promenljivih ili naredbi uvoza.

-Modifikatori su opcioni i njihov redosled nije bitan.

-U nekim situacijama su implicitno definisani, tj.podrazumeva se njihovo korišćenje.

-Postoji veliki broj modifikatora u Javi:

1. modifikatori kontrole pristupa,
2. modifikatori sinhronizacije,
3. nativni modifikatori,
4. modifikatori - anotacije itd.

• Postoje 4 nivoa vidljivosti, tzv. “4P-zaštita”.

(public, package, protected, private)

• Opisati povezanost modifikatora sa konceptom učaurivanja?

• public - omogućava vidljivost promenljive (ili metoda u svim klasama (čak iz različitih paketa).

```
public class SvimaDostupna {
public int javnaPromenljiva;
public String niskaJavna;
public float javniMetod() {
```

……

}

}

• package - služi za sužavanje vidljivosti promenljivih i metoda na nivo paketa. Karakteriše ga nepotrebnost navođenja modifikatora. To je podrazumevani nivo zaštite.

public class PodrazumevanaZastita {

int PaketnaCela = 3;

String paketnaNiska = “Pera”;

float metodPaketa() {

…….

}

}

• protected - nivo vidljivosti po liniji nasleđivanja odnosno na nivou klase i njenih potklasa (direktnih i indirektnih).

• Saopštava da je dozvoljeno korišćenje metoda i promenljivih samo:

1. od strane klasa istog paketa,

2. od potklasa izvan paketa

(ako klasa u paketu ima potklase izvan paketa).

U praksi se zaštićena polja trebaju pažljivo koristiti, jer bi svaka izmena dovela do problema u nasleđenim klasama. Zaštićeni metodi imaju više smisla. To indicira da se podklasama može verovati da će uspešno koristiti metod, a da druge klase to ne mogu.

• private - najviši nivo zaštite. Metodi i promenljive sa ovim modifikatorom vidljivi su samo iz klase u kojoj su definisani.

public class KlasaSaPrivatnim {

private int privatnaCela;

private float privatniMetod() {

……

}

}

Табела видљивости

Видљивост	public	protected	package	private
1. из исте класе	да	да	да	да
2. из класе у пакету	да	да	да	не
3. из било које класе	да	не	не	не
4. из поткласе ван пакета	да	да	не	не

20. Modifikatori za kontrolu vidljivosti metoda

/* ovo je vec verovatno u prethodnom pitanju:

Modifikator - prefiks koji moze biti primenjen (u razlicitim kombinacijama) na metode, promenljive pa i same klase, kao i na naredbu uvoza. To su specijalne kljucne reci koje menjaju ponasanje klasa, metoda, promenljivih ili naredbi uvoza.

Modifikatori su opcioni i njihov redosled nije bitan. U nekim situacijama su implicitno definisani, tj. podrazumeva se njihovo koriscenje. Postoji veliki broj modifikatora u Javi: modifikatori kontrole pristupa, modifikatori sinhronizacije, nativni modifikatori, modifikatori - anotacije itd.

*/

Postoje 4 nivoa vidljivosti, tzv. ”4P-zastita”. (public, package, protected, private)

- **public** - omogucava vidljivost promenljive (ili metoda u svim klasama (cak iz razlicitih paketa). Primer:

```
public class SvimaDostupna{
        public int javnaPromenljiva;
        public String niskaJavna;
        public float javniMetod(){...}
}
```

- **package** - sluzi za suzavanje vidljivosti promenljivih i metoda. Karakterise ga nepotrebnost navodenja modifikatora. To je podrazumevani nivo zastite. Primer:

```
public class PodrazumevanaZastita{
        int PaketnaCela = 3;
        String paketnaNiska = "Pera";
        float metodPaketa(){ ... }
}
```

Ako se pokusa pristup ovim elementima iz drugih paketa, nastaje greska pri prevodenju.

- **protected** - sledeci nivo vidljivosti. Regulise odnos izmedu klasa i njenih potklasa (direktnih i indirektnih). Ovim se saopstava da je dozvoljeno koriscenje metoda i promenljivih samo od strane klasa istog paketa ili od potklasa izvan paketa (ako klasa u paketu ima potklase izvan paketa). U praksi se zasticena polja trebaju pazljivo koristiti. Ako je je napravljena klasa sa zasticenim poljima koju nasleduju druge klase i ako se u nasledenim klasama direktno pristupa poljima, onda se vise ne sme menjati implementacija bazne klase, jer bi svaka izmena dovela do problema u nasledenim klasama. Zasticeni metodi imaju vise smisla. Metod se u klasi deklarise kao zasticen ako se tesko koristi. To indicira da se podklasama moze verovati da ce uspesno koristiti metod, a da druge klase to ne mogu.
- **private** - najvisi nivo zastite. Metodi i promenljive sa ovim modifikatorom ne mogu biti videni ni iz jedne druge klase osim one u kojoj su definisani (cak ni iz potklasa). Primer:

```
public class KlasaSaPrivatnim{
private int privatnaCela;
private float privatniMetod(){ ... }
}
```

Primer:

U ovom primeru se moze se promeniti vrednost polja za primerke klase Krug iz ma koje klase koja se nalazi u istom paketu u kome se nalazi i klasa Krug:

```
class Krug{
        int x, y, r; ...
        Krug(int x, int y, int r){ ... }
        void crtaKrug(){ ... }
```

```
}
```

Ovde se koriste privatne promenljive primerka, pa je za rad sa njima potrebno pozvati metode za ocitavanje/postavljanje (eng. getter/setter) tj. za pristup/mutaciju (eng. accessor/mutator).

```
class Krug{
        private int x, y, r; ...
        public int getR (){ return r; }
        public int setR(int vred){
                r = vred;
                crtakrug();
                drugimetod();
                return r;
        }
}
```

Tabela vidljivosti:

vidljivost	public	protected	package	private
Ista klasa	da	da	da	da
Klasa u paketu	da	da	da	ne
Bilo koja klasa	da	ne	ne	ne
potklasa van paketa	da	da	ne	ne

21. Modifikator konstantnosti za polja. Primeri.

Modifikator static utice na promenljive, metode, naredbe uvoza i inicijalizacione blokove modifikujuci njihovo znacenje.

static - modifikator i njegovo koriscenje kod klasnih promenljivih: Ako se polje definise kao staticko, tada postoji samo jedno takvo polje za klasu. Nasuprot tome, svaki od objekata ima svoju kopiju svih polja primerka. Kreira se na nivou klase (vezana za postojanje klase). Primer koriscenja staticke promenljive – lista studenata, svi imaju zajednicki naziv fakulteta

Ovaj modifikator se moze koristiti i kod metoda (staticki metod vezan za klasu, moze se pozvati bez instanciranja nekog objekta te klase u kojoj je deklarisan), naredbi uvoza (omogucava upotrebu statickih polja i metoda iz navedenih klasa bez upotrebe pune kvalifikacije), inicijalizacinog bloka (inicijalizacija statickih polja)

Primeri:

Staticki metod:

```
static void statMetod()
   {
      System.out.println("poruka iz statickog metoda");
   }
```

Staticki blok (neko izracunavanje da bi se inicijalizovale staticke promenljive – izvrsava se jednom kada se klasa ucita):

```
class Test
{
   // staticka promenljiva
   static int a = 10;
   static int b;

   // staticki blok
```

```
static {
    System.out.println("Static block initialized.");
    b = a * 4;
}
```

Analogno kljucnoj reci const u C++ u javi se moze koristiti modifikator **final.** Za razliku od C++ gde se konstantnoj promenljivoj mora dodeliti vrednost pri inicijalizaciji, u javi se final promenljivoj vrednost moze dodeliti i kasnije, ali samo jednom, dalje je nije dozvoljeno menjati. Takodje se koristi i kod metoda kada selimo da sprecimo njihovo override-ovanje

Ovaj modifikator utiče na promenljive, metode i klase modifikujući njihovo značenje na sledeći način:

1. Klase ne mogu biti nasleđene;
2. Polja ne mogu biti promenjena nakon inicijalizacije;
3. Metode ne mogu biti redefinisane;
4. Parametri metoda ne mogu biti menjani unutar metoda.

22. Modifikator konstantnosti za argumente metoda. Primeri.

U tu svrhu koristi se modifikator final (ima u prethodnom pitanju o njemu).

Ilustrovano na primeru:

```
public void doSomething(final int i, final int j){
        // ovde se ne mogu menjati vrednosti promenljivih i i j...
        // svaka promena bi bila vidljiva samo unutar ovog bloka (unutar metoda)
}
```

Ovde koristimo final da osiguramo da se promenljive i i j nece greskom promeniti u ovom metodu.

23. Modifikator za ogranicavanje nasledjivanja i prevazilazenja. Primeri.

Ovde se opet moze koristiti **final**. Finalna klasa ne moze biti nasledjena, a to se koristi da bi se sprecilo da neko pokusa da override-uje neke metode.

Modifikator **abstract** se koristi za definisanje apstraktnih klasa i metoda. Klasa je apstraktna ako se ne mogu napraviti konkretni objekti tog tipa, vec sluzi za obezbedivanje informacija za potklase. Ovim se postize suprotan efekat od koriscenja final. Apstraktni metod ima samo zaglavlje (bez tela f-je) i on u nekoj potklasi (koja nije apstraktna) mora biti predefinisan (override), dok final bas sluzi za sprecavanje predefinisanja (prevazilazenja???).

24. Inicijalizacioni blok primeraka. Primeri.

Inicijalizacioni blok je blok koji se nalazi u okviru definicije klase, a koji omogućuje da se dati programski kod izvrši prilikom inicijalizacije. Razlikujemo inicijalizacioni blok primerka i klasni inicijalizacioni blok.

Inicijalizacioni blok primerka (instancni incijalizacioni blok) se izvršava prilikom svakog kreiranja novog objekta – primerka klase koja sadrži inicijalizacioni blok

Primer – u sledecem pitanju (zajednicki primer da bi se uocile razlike)

25. Klasni inicijalizacioni blok. Primeri.

Inicijalizacioni blok klase (statički inicijalizacioni blok) – on se izvršava samo jednom, pri pokretanju programa, odnosno kada podsistem za učitavanje klasa JVM učita datu klasu.

Primer: islustruje oba tipa inicijalizacionih blokova

```
static int staticVariable;
```

```
int nonStaticVariable;

// Staticki inicijalizacioni blok se izvrsava jednom po klasi.
        static {
                System.out.println("Static initalization.");
                staticVariable = 5;
        }
// Instancni se poziva pre konstruktora – svaki put kad se instancira novi objekat ove
klase
{
        System.out.println("Instance initialization.");
        nonStaticVariable = 7;
}
```

26. Konstruktor. Primeri.

Prilikom kreiranja konkretnog primerka neke klase (preko operatora new), uvek se poziva konstruktor te klase.

Konstruktor je metod klase koji se poziva prilikom kreiranja novog primerka date klase.

Karakteristike konstruktora:

- nikad ne vraća vrednost
- njegovo ime se poklapa sa imenom klase.
- Kao i kod ostalih tipova metoda, tipovi i brojevi formalnih parametara pri deklaraciji i pozivu se moraju poklapati.

Ukoliko programer nije definisao konstruktor za datu klasu, prevodilac poziva podrazumevani konstruktor.

Primer:

```
class Knjiga {
        String autor;
        String naslov;
        int brojStrana;

        Knjiga(String a, String n, int bs) {   // konstruktor sa argumentima
                autor = a;
                naslov = n;
                brojStrana = bs;
        }
        void stampaPod() {
                System.out.println("Autor: "+autor);
                System.out.println("Naslov: "+naslov);
                System.out.println("Bro strana: "+brojStrana);
        }
        public static void main(String args[]) {
                Knjiga jedina;
                jedina = new Knjiga("Ivo Andric", "Gospodjica", 257);  // poziv konstruktora
                jedina.stampPod();
        }
}
```

27. Podrazumevani konstruktor. Primeri.

Podrazumevani konstruktor nema argumenata; On inicijalizuje sve instancne i klasne promenljive na podrazumevane vrednosti(to su npr. za int 0, za boolean false, za objekte null...).

Ako je programer definisao bar jedan konstruktor za datu klasu, podrazumevani konstruktor više ne postoji.

Primer:

```
Class Knjiga{
        String autor;
        String naslov;
        int brojStrana;

        public static void main(String args[]){
                Knjiga jedina;
                jedina = new Knjiga(); //  konstruktor bez argumenata je podrazumevan
                                //br strana ce se postaviti na 0, a autor i naslov na null
        }
}
```

28. Supstitucija parametara pri pozivu konstruktora. Primeri.

+ 29. Vezivanje polja i argumenata. Primeri

Objekat čije unutrašnje stanje (tj. vrednost nekog polja) može da se promeni naziva se mutirajući (promenljivi) objekat. U suprotnom se radi o nemutirajućem (nepromenljivom) objektu. Stoga treba pažljivo raditi sa konstruktorima mutirajućih objekata.

Pri pozivu se vrsi supstitucija, evaluiraju se vrednosti argumenata i postavljaju za parametre konstruktora. Naime, može se dogoditi da se, pri izvršavanju konstruktora, vrednost polja novokreiranog objekta postavi tako da sadrži vrednost argumenta konstruktora. U tom slučaju stvarni argument konstruktora i polje novokreiranog objekta postaju „vezani“ i referišu na isti objekat. Zbog toga promena objekta argumenta konstruktora dovodi do promene polja novokreiranog objekta i obrnuto.

/* Kako sam ja to razumela: Prenos argumenata vrsi se po vrednosti, medjutim kada je argument neki objekat, njegova vrednost je zapravo referenca i postavljanjem polja novokreiranog objekta na vrednost objekta-argumenta se kopira taj “broj” koji je zapravo referenca (adresa). Tada izmenom polja novog objekta menjamo i taj objekat koji je poslat kao argument */

Primer:

```
class Line extends GeometryObject {
        Point a;
        Point b;
        Line( Point a, Point b ) {
                this.a = a;
                this.b = b;
        }
        public static void main( String[] argsKL ) {
                Point tackaA = new Point( 19, 20 );
                Point tackaB = new Point( 7, 52 );
                Line linijaAB = new Line( tackaA, tackaB );
                tackaA.setX(200);
        // ovde je doslo do promene vrednosti linijaAB sa [(19,20)-(7, 52)] na [(200,20)-(7, 52)]
        iako nije radjeno sa linijaAB, vec sa tackaA
```

```
        }
    }
```

U prethodnom primeru kazemo da su vezani tacka A (kao objekat argument) i polje A objekta duz AB . Da bi se izbegle takve situacije, potrebno je umesto upotrebe reference na isti objekat napraviti kopiju objekta, sto se moze uraditi na sledeci nacin:

```
    class Line extends GeometryObject {
        Point a;
        Point b;
        Line( Point a, Point b ) {
            this.a = new Point( a.x, a.y );  // ovde sada pravimo dve nove tacke sa istim
            vrednostima polja
            this.b = new Point( b.x, b.y );
        }
    }
```

Sada više nema „vezivanja“ polja i argumenta. Međutim, zapis je rogobatan, naročito u slučaju kada mutirajući objekat koji predstavlja polje novog objekta ima složenu strukturu.

U prošlom slučaju bilo bi jako dobro kada bi postojala mogućnost da se jednostavnom naredbom napravi nova kopija. Da bi se to postiglo, potrebno je da u klasi koja opisuje tip polja postoji tzv. **kopirajući konstruktor**. Kopirajuci konstruktor od prosledjenog objekta pravi novi koji je isti takav (nova referenca, ali iste vrednosti polja), dalje kada se koristi novi objekat, on nikako nije vezan za onaj od kog je nastao.

Da bi se istaklo da kopirajući konstruktor ne može da modifikuje svoje argumente koristi se modifikator final.

Primer:

```
    class Point extends GeometryObject {
        int x;
        int y;
        Point( int x, int y ) {  // klasicni konstruktor sa argumentima
            this.x = x;
            this.y = y;
        }
        Point( final Point tacka ) {  // kopirajuci konstruktor – od prosledjenog objekta pravi novi
            this.x = tacka.x;
            this.y = tacka.y;
        }
    }
```

Kada imamo ovakav kopirajuci konstruktor, u primeru 2 mozemo liniju koda this.a = new Point(a.x, a.y); zameniti sa this.a = new Point(a); sto u slucaju da ima vise argumenata dosta skracuje kod.

30. Preopterecenje konstruktora, referenca this. Primeri.

Konstruktori mogu biti preopterećeni, isto kao i ostali metodi.

Ako postoji dodatni konstruktor, koji ima neke nove osobine, u njemu se može pozvati već postojeći konstruktor upotrebom ključne reči this na sledeći način: this(arg1, arg2, ...);

Primer:

```
    class Knjiga2 { String autor; String naslov; int brojStrana;
        Knjiga2(String a, String n, int bs){
            autor = a;
            naslov = n;
            brojStrana = bs;
```

```
        }
            // konstruktori koji koriste referencu this
        Knjiga2(String a, String n){ this( a, n, 200); }
        Knjiga2(String a){ this( a, "Gorski vijenac", 140); }
        Knjiga2(){ this( "Njegos", "Gorski vijenac", 140); }
        void stampaPod() {
            System.out.println("Autor: "+autor);
            System.out.println("Naslov: "+naslov);
            System.out.println("Bro strana: "+brojStrana);
        }
        public static void main(String args[]) {
            Knjiga2 prva;
            prva = new Knjiga2("Ivo Andric", "Gospodjica", 257);  // konstruktor sa argumentima
            prva.stampPod();
            Knjiga2 druga = new Knjiga2();  // konstruktor bez argumenata
            druga.stampPod();
            prva = new Knjiga2("Dzordz Orvel");  // konstruktor koji prima za argument samo naslov
            prva.stampPod();
            druga = new Knjiga2("Borislav Pekic", "Atlantida");  // konstruktor koji prima 2 argumenta
            druga.stampPod();
        }
    }
```

Isto kao sto koristimo this da pristupamo poljima i metodama koje ima klasa u kojoj se trenutno nalazimo, tako pomocu this() pozivamo konstruktor trenutne klase.

31. Poziv konstruktora natklase, referenca super. Primeri.

Pozivanje odgovarajuceg metoda definisanog u natklasi date klase, a koji je prevazidjen u potklasi, realizuje se koriscenjem kljucne reci **super**. Isto kao sto koristimo kljucnu rec this da referisemo na polja i metode u klasi u kojoj se nalazimo, tako koristimo super za referisanje na polja i metode natklase. Konstruktor natklase je takodje samo jedan metod, pa i njega mozemo pozvati na sledeci nacin: super(arg1, arg 2, ...);

Primer: (jako je glupo, nisam bas mastovita... treba proveriti da li je tacno)

```
class Lektira extends Knjiga{
    int razred; // imamo dodatno polje koje nema u natklasi, a ostala su ista

    Lektira(String naslov, String autor, int brStrana, int razred){
        super(naslov, autor, brStrana);        // konstruktor koji koristi konstruktor natklase
        this.razred = razred;
    }
    Lektira(){ this( "Njegos", "Gorski vijenac", 140, 2); }  // ovaj konstruktor indirektno poziva super()
}
```

Glava 7

1. Objekti, klasa Object, kreiranje objekata. Primeri.

Za kreiranje primeraka (konkretnih objekta neke klase) koristi se operator **new**.

Primeri:

```
String niska = new String();
Random slicajan = new Random();
Knjiga x = new Knjiga();
Point t = new Point(20, 30);
```

Komponente objekta su instancne promenljive i metodi. Kao sto je vec istaknuto, komponentama objekta pristupa se preko tzv. tacka-notacije.

Pristup promenljivim primerka:

```
prviObjekat.prom;
prviObjekat.prom.stanje;
prviObjekat.prom.stanje = true;
```

Pristup metodima:

```
prviObjekat.stampajMe();
prviObjekat.uzmiIme();
prvi objekat.postaviVelicinu(20);
prviObjekat.prom.metod3(arg1, arg2, arg3); // pozvan preko promen.
prviObjekat.getClass().getName();
                                    // metod iz metoda
```

Kada se kreira objekat, promenljiva kojoj se dodeljuje predstavlja pokazivac (referencu) na taj objekat. Pri pozivu metoda, u slucaju argumenata koji predstavljaju objekte, prosledjuje se vrednost reference na objekat, a ne sam objekat. Obrnuto vazi za argumente metoda primitivnog tipa. Tu promenljiva sadrzi podatak i duplikat tekuce vrednosti promenljive tj. podatka se prosleduje metodu. Dakle, u oba slucaja se vrsi supstitucija vrednosti, samo sto je kod objektnih argumenata u pitanju vrednost reference.

Klasa Object je roditeljska klasa svim drugim klasama u Javi, tj. nalazi se na vrhu hijerarhije, nju nasledjuju sve druge klase. U ovoj klasi su implementirani razni metodi pomocu kojih se obezbedjuju neka osnovna ponasanja, kao sto su podrazumevano poredjenje dva objekta, kloniranje, prevodjenje u nisku itd. Sve klase kao naslednice ove klase mogu da predefinisu ova ponasanja na zeljeni nacin.

2. Objekti, klasa Object, poredjenje objekata. Primeri.

*za klasu Object i objekte uopsteno pise u prethodnom pitanju.

Moguće je primenjivati dva operatora za poređenje objekata: == i !=. U ovom slučaju, objekti se smatraju jednakim ako zauzimaju isti prostor u memoriji (porede se njihove reference). Ne postoji mogućnost preopterećenja operatora u programskom jeziku Java, pa samim tim ni operatora poredjenja.

Drugi način ne upotrebom metoda equals koji je definisan u okviru klase Object, pa svaka klasa u Javi može da ga prevaziđe. Implementacija ovog metoda u klasi Object proverava da li su objekti identički jednaki. Programer može odlučiti da u svojoj klasi prevaziđe ovaj metod, npr. tako da vraća true npr. kada su sadržaji svih polja isti.

Primeri:

```
Knjiga k1 = new Knjiga("Gorski vijenac", "P. P. Njegos", 140);
Knjiga k2 = new Knjiga("Gorski vijenac", "P. P. Njegos", 140);
Knjiga k3 = k1;
boolean prvo = k1 == k2; // ovo poredjenje ce biti false iako su sva polja ista
boolean drugo= k1 == k3; // ovo poredjenje ce vratiti true jer je referenca ista, tj. Isti deo
memorije
```

/* cak i kada se novi objekat instancira pomocu kopirajuceg konstruktora operator == vraca false !! */

Naravno ako je u klasi Knjiga predefinisan metod equals, rezultati poredjenja mogu biti drugaciji.

3. Objekti, klasa Object, zivotni ciklus objekta, metod finalizator. Primeri.

Da bi se kreirao novi objekat, potrebno je prvo da se klasa ucita u memoriju, a novi objekat se instancira koriscenjem kljucne reci new. Tada se promenljivoj dodeljuje referenca na deo memorije u kom se nalaze podaci instanciranog objekta. Programer ne mora da vodi racuna o oslobadjanju tog dela memorije, jer ga sakupljac otpadaka uklanja i oslobadja njegove resurse onda kada vise nije potreban. (za razliku od npr. C++ gde je neophodno koristiti delete() ili free())

Metod finalizator je definisan u okviru klase Object, pa svaka klasa u Javi može da ga prevaziđe. Ovaj metod se može biti pozvan pri sakupljanju otpadaka. Implementacija ovog metoda u klasi Object ne radi ništa. Programer može odlučiti da u metodu klase koji ga prevazilazi izvršava npr. oslobađanje resursa sa kojima ne upravlja Java virtuelna mašina. Metod se poziva sa finalize(), ali obično nije neophodno njegovo korišćenje. Može se prevazići u sopstvenoj klasi sa:

```
protected void finalize() {

......

}
```

4. Realizacija metoda klase Object. Primeri.

*za ovo pitanje se moze reci prakticno sve iz prethodna 3 pitanja

Kao i svaka druga klasa, ova klasa ima implemenirane razne metode. Svaka klasa naslednica (dakle svaka klasa u javi) moze da prevazidje neke od ovih metoda dajuci svoju definiciju. Ovde su implementirana neka osnovna podrazumevana ponasanja.

Odredjivanje klase objekta se moze postici koriscenjem metoda getClass, koji je definisan u klasi Object. Ovaj metod se ne moze prevazici. On vraca objekat tipa Class, koji sadrzi metode (vise od 50) pomocu kojih se mogu dobiti informacije o datoj klasi (natklasama, interfejsima...). Primer:

```
String runtimeClassName = objekat.getClass().getName();
```

Ovde su implementirane neke vec spomenute metode kao sto su equals i finalize. Takodje je implementiran metod za prevodjenje u nisku – toString() koji vraca tekstualnu reprezentaciju objekta. Ovakva reprezentacija u potpunosti zavisi od strukture objekta, pa je preporuka da se ovaj metod uvek prevazilazi u novim klasama.

```
@Override
public String toString() { .... }
```

5. Niske, klasa String, karakteristike niski, imutabilnost. Primeri.

Stringovni tip je objektni tip koji se koristi za predstavljanje teksta, koji se interno (u okviru objekta) cuva kao niz znakova. Primerci klase String ne mogu da mutiraju, sto znaci da jednom kada dobiju vrednost, ta unutrasnja vrednost vise ne moze da se menja.

Dakle, kada se izvrsava operacija nad postojecim string objektom, kao rezultat se uvek kreira novi primerak klase String. Promenom niske na koju referise promenljiva tipa String, se ne menja stari objekat, vec promenljiva referise na novonapravljeni String objekat. Kao i kod

drugih objekata, moze se koristiti literal null za odbacivanje objekta na koji trenutno referise data String promenljiva.

6. Niske, klasa String, poredjenje niski. Primeri.

*pocetak isti kao prethodno

Niske se mogu porediti na 3 nacina:

- Metod equals() klase String – poredi stringove po vrednosti, tj. Uporedjuje karakter po karakter da li su isti (postoji i odgovarajuci equalsIgnoreCase() metod koji ne pravi razliku izmedju malih i velikih slova u niskama koje poredi)
- Operator == - uporedjuje da li su nike identicki iste, tj. Poredi njihove reference, a ne vrednost
- Metod compareTo() - uporedjuje niske leksikografski, tj. Ne vraca samo bool tacno/netacno vec int kao informaciju koja je veca (pozitivan broj ako je prva veca, negativan ako je druga, a nula kada su jednake po vrednosti). Ovaj metod se koristi pri sortiranju niski.

Primer:

```
String s1 = “niska”;
String s2 = “niska”;
String s3 = new String(“niska”);
String s4 = “NISKA”;
```

U primeru se poredjenjem sa equals dobija da su prve tri niske iste, a samo poslednja se razlikuje. Poredjenjem sa == dobija se da su prve dve iste (!!! proveriti) jer su u pitanju konstantne niske, pa s1 i s2 referisu na istu nisku (prvi put se niska upise u segment podataka(?) pa se sledeci put dodeli referenca na istu nisku), dok je s3 razlicita jer je razlicita referenca. Poredjenjem metodom compareTo() su prve tri niske jednake, a poslednja je manja od njih (po ascii kodovima). Jedino poredjenjem equalsIgnoreCase metodom se dobije da je poslednja niska jednaka kao ostale.

7. Realizacija metoda klase String. Primeri.

/* treba navesti osnovno o stringovima, da su imutabilni i da se npr. konkatenacijom i drugim operacijama ne menja dati string, vec se u memoriji pravi novi, a zatim referenca preusmerava njega */

Klasa String ima implementirane razne metode za rad sa stringovima, u narednoj tabeli su prikazane najbitnije: (mrzelo me stvarno da prevodim ovo sve, ali razumljivo je)

1	char charAt(int index)	returns char value for the particular index
2	int length()	returns string length
3	static String format(String format, Object... args)	returns a formatted string.
4	static String format(Locale l, String format, Object... args)	returns formatted string with given locale.
5	String substring(int beginIndex)	returns substring for given begin index.
6	String substring(int beginIndex, int endIndex)	returns substring for given begin index and end index.
7	boolean contains(CharSequence s)	returns true or false after matching the sequence of char value.
8	static String join(CharSequence delimiter, CharSequence... elements)	returns a joined string.
9	static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)	returns a joined string.
10	boolean equals(Object another)	checks the equality of string with the given object.
11	boolean isEmpty()	checks if string is empty.
12	String concat(String str)	concatenates the specified string.
13	String replace(char old, char new)	replaces all occurrences of the specified char

		value.
14	String replace(CharSequence old, CharSequence new)	replaces all occurrences of the specified CharSequence.
15	static String equalsIgnoreCase(String another)	compares another string. It doesn't check case.
16	String[] split(String regex)	returns a split string matching regex.
17	String[] split(String regex, int limit)	returns a split string matching regex and limit.
18	String intern()	returns an interned string.
19	int indexOf(int ch)	returns the specified char value index.
20	int indexOf(int ch, int fromIndex)	returns the specified char value index starting with given index.
21	int indexOf(String substring)	returns the specified substring index.
22	int indexOf(String substring, int fromIndex)	returns the specified substring index starting with given index.
23	String toLowerCase()	returns a string in lowercase.
24	String toLowerCase(Locale l)	returns a string in lowercase using specified locale.
25	String toUpperCase()	returns a string in uppercase.
26	String toUpperCase(Locale l)	returns a string in uppercase using specified locale.
27	String trim()	removes beginning and ending spaces of this string.
28	static String valueOf(int value)	converts given type into string. It is an overloaded method.

8. Realizacija metoda klase StringBuilder. Primeri.

Ova klasa se koristi za kreiranje mutabilnog (promenjivog) stringa. Dok se klasicnim operacijama sa stringovima svaki put kreira novi string, na ovaj nacin se direktno menja dati string. Nova instanca stringBuilder-a se moze kreirati na neki od sledecih nacina:

new StringBuider(); // kreira prazan string buider sa inicijalnim kapacitetom 16

new StringBuider(String str); // inicijalizuje kreirani string builder na str

new StringBuider(int length); // kreira prazan string buider sa zadatim pocetnim kapacitetom

Neke od najvaznijih metoda koje su implementirane u ovoj klasi su date u sledecoj tabeli: (takodje me jako mrzi da prevodim, mozda neki drugi put...)

public StringBuilder append(String s)	is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.
public StringBuilder insert(int offset, String s)	is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
public StringBuilder replace(int startIndex, int endIndex, String str)	is used to replace the string from specified startIndex and endIndex.
public StringBuilder delete(int startIndex, int endIndex)	is used to delete the string from specified startIndex and endIndex.
public StringBuilder reverse()	is used to reverse the string.
public int capacity()	is used to return the current capacity.
public void ensureCapacity(int minimumCapacity)	is used to ensure the capacity at least equal to the given minimum.
public char charAt(int index)	is used to return the character at the specified position.
public int length()	is used to return the length of the string i.e. total number of characters.
public String substring(int beginIndex)	is used to return the substring from the specified beginIndex.
public String substring(int beginIndex, int endIndex)	is used to return the substring from the specified beginIndex and endIndex.

9. Prevazilaženje podrazumevanog ponašanja objekata prilikom određivanja niske koja opisuje objekat. Primeri:

- Proces definisanja metoda u potklasi koji ima isti potpis kao i metod u natklasi naziva se prevazilaženje metoda. On omogućuje promenu ponašanja ili dodavanje novog ponašanja u potklasama. Novi metod ima isto ime, isti broj i tipove parametara i vraća rezultat istog tipa kao metod natklase koji je prevazilazi. Novi metod može da vrati i podtip tipa koji vraća metod natklase, što se naziva kovarijantni tip rezultata. Prilikom prevazilaženja poželjno je koristiti anotaciju @Override.

Pozivom metoda toString(), dobija se niska-reprezentacija objekta, koja u potpunosti zavisi od strukture tog objekta, i zbog toga je preporučljivo da se prevaziđe u novonapravljenim klasama.

Primer:

```
public class Tacka{

...

@Override

        public String toString() {

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

                }

}
```

10. Prevazilaženje podrazumevanog ponašanja objekata prilikom poređenja jednakosti između objekata. Primeri:

- Ukoliko se porede sa "==", objekti će vraćati rezultat true samo ako dva objekta referišu na isti prostor u memoriji. Ukoliko npr. želimo da poredimo dva kompleksna broja, to možemo uraditi prevazilaženjem metoda equals() u klasi Complex.

Primer:

```
class Complex {

private double re, im;

...

public boolean equals(Object o) {

Complex c = (Complex) o;

 return Double.compare(re, c.re) == 0 && Double.compare(im, c.im) == 0;

}

...

}

...

public static void main(String[] args) {

      Complex c1 = new Complex(10, 15);

      Complex c2 = new Complex(10, 15);
```

```
    if (c1.equals(c2)) {
        System.out.println("Equal ");
    } else {
        System.out.println("Not Equal ");
    }
  }
```

Ovde će biti ispisano Equal, a da smo poredili sa ==, ispisalo bi se Not Equal.

11. Prevazilaženje podrazumevanog ponašanja objekata prilikom finalizacije objekata. Primeri:

- Metod finalize() je definisan u klasi Object, i njegova implementacija ne radi ništa, tako da programer može odlučiti da u nekoj klasi njega prevaziđe, za npr. oslobađanje resursa sa kojima ne upravlja Java virtuelna mašina, tako da ovaj metod može biti pozvan pri sakupljanju otpadaka.

Primer:

```
...
@Override
public void finalize() {
        try {
        reader.close();
        System.out.println("Closed BufferedReader in the finalizer");
        } catch (IOException e) {
                // ...
        }
}
```

12. Realizacija klasa omotača za primitivne tipove. Primeri:

- Klase omotači konvertuju primitivne tipove u objekte (char - Character, int - Integer...). Objekti su neophodni ako želimo da menjamo argumente prosleđene u nekom metodu(primitivni tipovi se šalju po vrednosti, a objekti po referenci). Klase u java.util paketu rade samo nad objektima, pa su i ovde korisni. Objekti su potrebni i za sinhronizaciju i multithreading. Neki metodi klase omotača:

1. valueOf(): korisit se da se napravi omotač oko objekta za dat primitivan tip ili String

2. xxxValue(): koristi se da se dobije primitivan tip od datog objekta, ovde xxx može biti byte, char, int...(byteValue(), intValue()...)

3. toString(): koristi se da se objekat ili primitivan tip konvertuje u String.

Implementacija se vrši pomoću autoboxing i unboxing mehanizma.

Primer autoboxing:

```
class Auto{
		public static void main(String args[]){
			//Konvertovanje int u objekat Integer
int number=1;  //int primitive
			Integer obj=Integer.valueOf(number); //Kreiranje klase omotača objekta
			...
  }
}
```

Primer unboxing:

```
class Un{
		public static void main(String args[]){
		Integer obj = new Integer(1); //Kreiranje klase omotača objekta
		int number = obj.intValue(); //Konvertovanje u primitivan tip
    ...
  }
}
```

13. Nizovi objekata. Primeri:

- Realizovani su kao obični nizovi, samo kao svoje elemente čuvaju reference na određene objekte. Kada se niz objekata instancira na prikazan način, individualni elementi niza se kreiraju pomoću new.

Primer:

```
class Main{
		public static void main(String args[]){
		//create array of employee object
		Employee[] obj = new Employee[2] ;

		//create actual employee object
		obj[0] = new Employee();
		obj[1] = new Employee();

		//assign data to employee objects
		obj[0].setData(100,"ABC");
		obj[1].setData(200,"XYZ");

		//display the employee object data
```

```
            System.out.println("Employee Object 1:");
            obj[0].showData();
            System.out.println("Employee Object 2:");
            obj[1].showData();
  }
}
```

VIII NAPREDNI RAD SA KLASAMA I OBJEKTIMA

1.Apstraktne klase. Definisanje apstraktne klase. Primeri:

- Apstraktna klasa je ona klasa u kojoj postoji bar jedan metod koji nema definiciju, već samo njegovu deklaraciju (Npr. u klasi Životinje, metod pomeriSe() možemo proglasiti apstraktnim, jer znamo da bi trebalo da svaka životinja može da se kreće, ali se ne kreću sve na isti način, i samim tim, i klasa Životinja postaje apstraktna). Klasa može biti apstraktna čak iako ne sadrži ni jedan apstraktni metod. Da bi klasa bila prozvana za apstraktnu, neophodno je staviti ključnu reč abstract. Apstraktna klasa se ne moze instancirati

Primer:

```
public abstract class Covek(){
...
abstract void predstaviSe();
...
}
```

2. Apstraktne klase. Nasleđivanje između apstraktnih i konkretnih klasa. Primeri:

- Potklasa apstraktne klase:

1. Može realizovati sve apstraktne metode i u tom slučaju ona postaje regularna klasa
2. Ne mora realizovati apstraktne metode, i u tom slučaju i potklasa ostaje apstraktna.

Takođe, može da koristi neapstraktne metode apstraktne natklase. Apstraktna klasa može da ima konstruktor kojim definiše sopstvena polja, a taj konstruktor se potom poziva od strane konstruktora konkretne potklase.

Primer:

```
public abstract class Osoba{
...
public Osoba(String jmbg){this.jmbg = jmbg;}
...
}
```

```
public class Student extends Osoba{
...
public Student(String jmbg, int index){
super(jmbg);
this.index = index;
}
...
public static void main(String[] args) {
Person s=new Student("xxxxxxxxxxxxx", "yyyyy");
}
}
```

3. Interfejsi. Definisanje interfejsa. Primeri:

- Interfejsi su totalno apstraktne klase, tj. u njima se mogu naći samo zaglavlja metoda i definicije konstanti. Oni opisuju neko ponašanje, a klasa koja ih implementira određuje kako se to ponašanje realizuje. Nije moguće direktno praviti primerak interfejsa, on samo može da bude implementiran od strane klase ili nasleđen od strane drugog interfejsa. U java biblioteci postoje već ranije definisani interfejsi, a moguće je i napraviti nove.

Primer:

```
public interface MojInterfejs{
...
}
```

Kreiranje objekta tipa interfejs preko klase koja ga implementira:

```
Runnable trceci = new MojaKlasa();
```

ovde se od objekta trceci očekuje da izvršava metod run() interfejsa Runnable.

4. Interfejsi. Proširivanje interfejsa. Primeri:

- Jedan interfejs može proširivati veći broj interfejsa uz pomoć ključne reči extends, i klasa koja implementira taj interfejs (koji proširuje drugi interfejs), mora implementirati sve metode iz oba interfejsa.

Primer:

```
public interface Radoznao{
void pita();
void interesujeSe();
}
```

```
...
public interface Eksperimentator extends Radoznao{
void realizujeEksperimente();
}
...
public class Istrazivac implements Eksperimentator{
// mora realizovati sva tri metoda iz interfejsa
}
```

5.Interfejsi. Implementiranje interfejsa od strane klasa. Primeri:

- Klasa implementira neki interfejs ako sadrži realizaciju svakog od metoda tog interfejsa, a naziv navodimo iza ključne reči implements.

Primer:

```
class MojaKlasa implements Runnable {
...
//implementacije svih metoda iz interfejsa Runnable
}
```

Klasa može da implementira više od jednog interfejsa, čime se nadoknađuje nemogućnost višestrukog nasleđivanja.

6. Interfejsi. Parametri tipa interfejs. Primeri:

- Jedna od mogućnosti je da se parametri deklarišu tako da budu tipa interfejsa.

Primer:

```
public interface Radoznao {
void pita(Radoznao neko);
...
}
public class Naucnik implements Radoznao {
public pita(Radoznao neko){
Naucnik pravi = (Naucnik)neko;
...
}
}
```

7. Interfejsi u JDK-u. Uređenje u kolekciji, interfejs Comparable. Primeri:

- Metod sort u klasi Arrays obećava da će sortirati niz objekata, ali samo pod uslovom da objekti pripadaju klasi koja implementira interfejs Comparable. Da bi klasa implementirala ovaj interfejs, ona mora imati realizaciju njenog jedinog metoda compareTo(Object other). Ovaj metod treba impementirati tako da vraca int i to pozitivnu vrednost ako je trenutni objekat veci od prosledjenog, negativnu ako je manji, a 0 ako su jednaki.

Primer:

```
public static void main(String[] args) {

int[] niz = {1, 3, 2, 4, 5};

Arrays.sort(niz);

}
```

8. Interfejsi u JDK-u. Uređenje u kolekciji, interfejs Comparator. Primeri:

- Interfejs Comparator (koji u sebi ima metod compare), omogućava da se elementi koji se porede urede na više načina sortiranja. Za razliku od compareTo(), ovaj metod prima dva argumenta. Primer:

```
public static void main(String[] args) {

...

Tacka[] niz = {t1, t2, t3, t4, t5, t6};

        Arrays.sort(niz, Tacka.PorediPoXPaPoY);

...

Arrays.sort(niz, Tacka.PorediPoYPaPoX);

}
```

Drugi argumenti funkcija sort su komparatori implementirani u klasi Tacka preko metoda compare, tako da vraćaju -1 ukoliko je prvi element manji od drugog, 0 ako su isti, i 1 ako je prvi element veći.

9. Metodi klase Arrays. Primeri:

Najbitniji metodi klase Arrays:

- static <T> List<T> asList(T... a)
- static int binarySearch(elementToBeSearched)
- static <T> int binarySearch(T[] a, int fromIndex, int toIndex, T key, Comparator<T> c)
- compare(array 1, array 2)
- copyOf(originalArray, newLength) // ako je nova duzina veca, ostatak se popunjava po defaultu
- copyOfRange(originalArray, fromIndex, endIndex)
- equals(array1, array2)
- fill(originalArray, fillValue)
- setAll(originalArray, functionalGenerator)
- sort(originalArray)
- sort(originalArray, fromIndex, endIndex)

- static <T> void sort(T[] a, int fromIndex, int toIndex, Comparator< super T> c)
- static <T> void sort(T[] a, Comparator< super T> c)
- toString(originalArray) // implementiran da ispisuje elemente niza u [ ] razdvojene zarezom

1.static void sort(Object[] a)

sortira elemente niza poboljšanom verzijim Merge sort-a, elementi moraju implementirati interfejs Comparable.

2. static void sort(Object a, comparator c)

druga varijanta koristi eksplicitni način poređenja definisan klasom Comparator. Primeri su prethodna dva pitanja.

10. Interfejsi u JDK. Kloniranje objekata, interfejs Cloneable. Primeri:

Klasa Object sadrži metod clone(), koji se može koristiti za pravljenje kopije datog objekta. Realizacija vrši plitko kopiranje, što znaci da dolazi do bočnog efekta pri radu sa klasama koje imaju mutabilne podatke, a ne dolazi ukoliko su svi podaci imutabilni. (tj. kopira se referenca ako je u pitanju objektni tip). Kada se ovaj metod pozove u nekoj klasi, njega pozivaju (a takođe ga i moraju implementirati) sve njegove natklase preko implementiranog interfejsa Cloneable

Primer:

```
public class PasLjubimac implements Cloneable{

...

@Override

        public Object clone() throws CloneNotSupportedException {

        return super.clone();

        }

}

...

public static void main(String[] args) {

...

PasLjubimac mojPas = new PasLjubimac("Pas", "Astor");

      PasLjubimac vasPas = null;

...

vasPas = (PasLjubimac)mojPas.clone();

...

}
```

11. Nasleđivanje i polimorfizam. Primeri:

Pomoću nasleđivanja se nasleđuju metodi i svojstva iz drugih klasa. Polimorfizam koristi te metode za različite namene. Ovo omogućava da se isti metodi koriste na različite načine (Npr. u klasi Životinje implementiramo metod zvuk. Njene potklase, npr. Pas i Mačka, takođe mogu imati svoju različitu implementaciju ovog metoda). Na ovaj način se preko polimorfizma obrazuje hijerarhija klasa od opštijih ka specifičnijim. Sve klase u Javi nasledjuju klasu Object, pa nju mozemo videti kao koren hijerarhijskog stabla nasledjivanja.

12. Preporuke za nasleđivanje. Primeri:

1. Zajedničke operacije i polja smestiti u natklase

2. Izbegavati upotrebu zaštićenih polja

3. Ne koristiti nasleđivanje sem ukoliko ono ima smisla za sve metode klase iz koje se nasleđuje

4. Prilikom prevazilaženja metoda ne menjati očekivano ponašanje, tj. poštovati princip zamene

5. Koristiti polimorfizam, a ne informacije o tipu

IX UGNJEŽDENE I UNUTRAŠNJE KLASE

1. Ugnježdene klase. Nestatičke ugnježdene klase. Primeri:

- Ugnježdena klasa je ona koja je definisana unutar neke druge klase. Ona se može kreirati pomoću ključne reči static ili bez. U prvom slučaju reč je o statičkoj ugnježdenoj, a u drugom, o unutrašnjoj klasi.

Iz unutrašnje klase se može pristupiti bilo kojem članu spoljašnje klase (top-level class), bez obzira na njene atribute, a takođe i statičkim članovima bilo koje statički ugnježdene klase unutar iste spoljašnje klase. Unutrašnje klase ne mogu imati statičke članove. Objekti primerci unutrašnje klase postoje samo u okviru primerka spoljašnje klase, što znači, da bi se instancirala unutrašnja klasa, prvo se mora instancirati spoljašnja.

Primer:

```
SpoljašnjaKlasa spoljašnjiObjekat = new SpoljašnjaKlasa();

SpoljašnjaKlasa.UnutrašnjaKlasa unutrašnjiObjekat = spoljašnjiObjekat.new UnutrašnjaKlasa();
```

2. Ugnježdene klase. Statičke ugnježdene klase. Primeri:

- Statičke ugnježdene klase mogu imati i statičke i nestatičke članove, a iz nje se može pristupiti samo statičkim članovima spoljašnje klase. One su zapravo statički članovi i mogu se koristiti u statičkim kontekstima (kao statički metodi). Instanciranje statičke ugnježdene klase ne zahteva instanciranje spoljašnje klase, a vrši se kao:

```
SpoljasnjaKlasa.StatickaUgnjKlasa unutrasnjiObjekat = new
SpoljašnjaKlasa.StatickaUgnjKlasa();
```

3. Lokalne unutrašnje klase. Primeri:

- Klasa se može definisati unutar metoda, i to je tzv. lokalna unutrašnja klasa (local inner class). Objekti lokalne unutrašnje klase se mogu kreirati samo unutar metoda u kome je i definicija klase (korisno za npr. osluškivače događaja (event listener)). Može referisati promenljive deklarisane u metodu u čijoj definiciji se pojavljuje, i to samo ako su te promenljive finalne.

Primer:

```
public class Spoljasnja {


  void mojMetod() {
    // lokalna unutrašnja klasa
    class Lokalna {
      public void print() {
        ...
      }
    } // kraj lokalne unutrašnje klase


    // pristup unutrašnjoj klasi
    Lokalna unutrasnja = new Lokalna();
    unutrasnja.print();
  }


  public static void main(String args[]) {
    Spoljasnja spoljasnja = new Spoljasnja();
    spoljasnja.mojMetod();
  }
}
```

1. Doseg lokalne unutrasnje klase je ogranicen na nivo bloka u kom je ona definisana.
2. Ova klasa ne moze biti instancirana izvan bloka u kom je kreirana
3. Sve do JDK 7, ova klasa je mogla pristupati jedino finalnim lokalnim promenljivima iz bloka, ali od JDK 8 omogucen je i pritup nefinalnim lokalnim promenljivima iz bloka u kom je definisana.
4. Ima pristup clanovima svoje okruzujuce (spoljasnje) klase.
5. Moze nasledjivati neke apstraktne klase ili implementirati neki interfejs.

4. Anonimne klase. Primeri:

- Dopuštaju da se istovremeno deklariše klasa i kreira njen primerak, pri čemu ta klasa nema ime, tj. one su kao lokalne klase, samo nemaju ime. Omogućavaju koncizan kod i koriste se

kada se lokalna klasa koristi samo jednom (Npr. kod sortiranja, koristimo anonimnu klasu koja nasleđuje Comparator jer nam treba specifican nacin sortiranja niza i to samo jedanput). Za razliku od normalnih klasa, anonimne klase mogu ili implementirati samo jedan interfejs ili nasledjivati samo jednu klasu, a takodje nemaju ni konstruktor (bas zbog toga sto i nemaju ime, niti mogu imati vise instanciranih objekata)

Primer:

```
public class Tacka{

public final static Comparator<Tacka> PorediPoYPaPoX = new Comparator<Tacka>() {

        @Override

        public int compare(Tacka t1, Tacka t2) {

            return ...

        }

}


public static void main(String[] args) {

...

Tacka[] niz = {t1, t2, t3, t4, t5, t6};

        Arrays.sort(niz, Tacka.PorediPoXPaPoY);

...

}
```

9. IZUZECI I TVRDNJE

1. Izuzeci u Javi. Tipovi izuzetaka, izuzeci tipa Error i RuntimeException. Primeri.

• Java koristi izuzetke kao nacin signaliziranja ozbiljnih problema prilikom izvrsavanja programa.
• Standardne klase (klase iz JDK-a) ih intenzivno koriste.
• Posto oni nastaju kada u Java programima stvari krenu naopako, a pre ili kasnije tako nesto se desi,
izuzetke treba uzeti u ozbiljno razmatranje kada se dizajnira aplikacija i kada se pisu programi.
• Izuzetak obicno signalizira gresku ili neki posebno neobican dogadjaj u programu koji zasluzuje
posebnu paznju.
• Ukoliko se usred koda koji rukuje normalnim operacijama programa, istovremeno rukuje mnostvom
cesto neobicnih gresaka, struktura programa uskoro postaje komplikovana i teska za razumevanje.
• Glavna korist od izuzetaka jeste sto oni razdvajaju kod koji obradjuje greske od koda koji se izvrsava
kada stvari teku glatko.

• Drugi pozitivan aspekt izuzetaka jeste sto obezbedjuju mehanizam prisile da se reaguje na odredjene
vrste gresaka.
• Vazna ideja koju treba shvatiti je da ne treba sve greske u programima signalizirati izuzecima –
samo neuobicajene ili katastrofalne. Na primer, ako korisnik ne unese ispravan ulazni podatak, za to ne treba koristiti izuzetke, jer se radi o uobicajenom dogadjaju.
• Razlog za takvu odluku je sto rukovanje izuzecima ukljucuje mnogo dodatnog procesiranja sto usporava celokupan program.
• U Javi, izuzetak je **objekat** koji se kreira kada se desi nenormalna situacija u nasem programu.
• Objekat-izuzetak sadrzi polja (atribute) u koja se smestaju informacije o prirodi problema.
• Za izuzetak se kaze da je podignut ili izbacen (thrown).
• Objekat koji identifikuje izuzetnu situaciju tj. izuzetak se podize kao argument u specificnom delu
koda koje se pise bas da bi rukovalo tom vrstom problema.
• Za kod koji prima objekat izuzetak kao parametar se kaze da ga hvata (catch).

Situacije koje uzrokuju izuzetke su prilicno raznovrsne, ali spadaju u cetiri kategorije:
1) greske koda ili podataka (neispravan pokusaj kastovanja objekta, koriscenje indeksa koji je izvan
granica za taj niz, deljenje celog broja nulom);
2) izuzeci standardnih metoda (npr. izbacivanje StringIndexOutOfBoundsException izuzetaka);
3) izbacivanje nasih sopstvenih izuzetaka;
4) Java greske (obicno posledica greske u nasem programu).

Jednostavan primer koda koji izbacuje izuzetak:

```
public class primer{
	public static void main(String[] args){
		try{
			int a[] = new int[2];
			System.out.println("Pristupam elementu: " + a[3]);
		}catch(ArrayIndexOutOfBoundsException e){l
			System.out.println("Izuzetak izbacen:  " + e );
		}
	}
}
```

Tipovi izuzetaka:
Izuzetak je uvek objekat neke potklase standardne klase Throwable. To vazi i za izuzetke koje sami
definisemo, kao i za standardne izuzetke.
Dve direktne potklase klase Throwable – klasa Error i klasa Exception – pokrivaju sve standardne
izuzetke. Obe ove klase imaju potklase za specificne izuzetke.

Izuzeci tipa Error:
Izuzeci definisani klasom Error i njenim potklasama karakterisu se cinjenicom da se od nas ne ocekuje
da preduzimamo nista, ne ocekuje se da ih hvatamo. Klasa Error ima tri direktne potklase:
• ThreadDeath – izbacuje se kada se nit koja se izvrsava namerno stopira.
• LinkageError – ozbiljni problemi sa klasama u programu, npr. nekompatibilnost medju klasama ili

pokusaj kreiranja objekta nepostojeceg klasnog tipa su neke od stvari koje mogu izazvati izbacivanje izuzetka ovog tipa.

• VirtualMachineError – sadrzi cetiri potklase izuzetaka koji se izbacuju kada se desi katastrofalni pad JVM.

Izuzeci koji odgovaraju objektima klasa izvedenih iz LinkageError i VirtualMachineError su rezultat katastrofalnih dogadjaja i uslova. U takvim situacijama obicno sve sto mi mozemo da uradimo jeste da procitamo poruku o gresci koja se generise kada se izbaci izuzetak, posebno u slucaju LinkageError izuzetka i da pokusamo da shvatimo sta je u nasem kodu moglo da izazove problem.

Izuzeci RuntimeExeption:
Za skoro sve izuzetke predstavljene potklasama klase Exception, mora se u program ukljuciti kod koji ce rukovati njima ukoliko nas kod moze izazvati njihovo izbacivanje.

Ako metod u nasem programu moze da generise izuzetak tipa koji ima Exception kao nadklasu, mora se ili rukovati izuzetkom unutar tog metoda ili registrovati da taj metod moze izbaciti takav izuzetak. U suprotnom, program se nece prevesti.

Razlog zbog kog se RuntimeException izuzeci drugacije tretiraju i prevodilac dopusta da ih ignorisemo je taj sto oni generalno nastaju zbog ozbiljnih gresaka u nasem kodu. U vecini slucajeva mozemo da uradimo malo da popravimo situaciju. Medjutim, u nekim kontekstima za neke od ovih izuzetaka, to nije uvek slucaj i mozda zelimo da ukljucimo kod za njihovo prepoznavanje.
Potklase klase RuntimeException definisane u standardnom paketu java.lang su:
• ArithmeticException
• IndexOutOfBoundsException
• NegativeArraySizeException
• NullPointerException
• ArrayStoreException
• ClassCastException
• IllegalArgumentException
• SecurityException
• IllegalMonitorStateException
• IllegalStateException
• UnsupportedOperationException

2. Rukovanje izuzecima, hvatanje izuzetaka. Primeri.

Dakle, ako dati programski kod moze izbaciti izuzetke koji nisu tipa Error ili RuntimeException (i njihovih potklasa – sto se podrazumeva), onda se mora nesto preduzeti povodom toga. Kad god se pise programski kod koji moze da izbaci izuzetak, javlja se izbor:
• moze se kreirati kod unutar metoda za rukovanje datim izbacenim izuzetkom

• moze ignorisati izuzetak u tom metodu i omogucitii da metod koji sadrzi kod koji moze izbaciti
izuzetak prosledi taj izuzetak u pozivajuci metod

Pretpostavimo da imamo metod koji moze izbaciti izuzetak koji nije tipa potklase RuntimeException
niti Error klase. Neka je izuzetak npr. tipa IOException.
Ako izuzetak nije uhvacen i nije zavrseno sa njegovom obradom unutar metoda, najmanje sto moramo
da uradimo jeste da deklarisemo da moze biti izbacen izuzetak.
Kako se to radi? Jednostavno se doda throws klasa u definiciju metoda npr.

double myMetod() throws IOException{...}
double myMetod() throws IOException,FileNotFoundException {...}

Kao sto prethodni primeri pokazuju, da deklarisemo da nas metod moze izbaciti izuzetke, samo dodamo
kljucnu rec throws nakon liste parametara metoda. Zatim se navodi lista klasa za izuzetke koji mogu
biti izbaceni, razdvojenih zapetama.

Ako neki drugi metod poziva ovaj metod, on mora da uzme u obzir izuzetke koje ovaj moze izbaciti, pa
ih ili obradjivati ili i on deklarisati da izbacuje izuzetke istog tipa.
Ukoliko se ne uradi ni jedno ni drugo, prevodilac ce to utvrditi i doci ce do grske prevodjenja, pa se
Java kod nece prevesti u bajt-kod.
Ako zelimo da rukujemo izuzecima tamo gde se oni dese, potrebno je ukljuciti tri vrste blokova koda u
metod koji rukuje izuzecima, i to su:
• try blok – obuhvata kod gde se moze javiti jedan ili vise izuzetaka. Kod koji moze da izbaci izuzetak
koji zelimo da uhvatimo mora biti u try bloku;
• catch blok – obuhvata kod koji je namenjen da rukuje izuzecima odredjenog tipa koji mogu biti
izbaceni u pridruzenom try bloku;
• finally blok – uvek se izvrsava pre nego se metod zavrsi, bez obzira da li je bilo koji izuzetak izbacen u try bloku ili nije.

Try blok, primer:
Kada treba da se uhvati izuzetak, kod metoda koji moze izbaciti izuzetak mora biti obuhvacen try-blokom.
Kod koji moze izazvati izuzetke ne mora biti u try-bloku, ali onda metod nece biti sposoban da uhvati
nijedan od izbacenih izuzetaka, pa u deklaraciji metoda treba biti istaknuto da taj metod moze izbaciti
tipove izuzetaka koji nisu uhvaceni.
try-blok cini kljucna rec **try** za kojom sledi par viticastih zagrada koje okruzuju kod koji moze izbaciti
Izuzetak.

```
try{
// kod koji moze izbaciti jedan ili vise izuzetaka
}
```

try-blokovi su neophodni i kada je potrebno da se uhvate izuzeci tipa Error ili RuntimeException (oni
se lako generisu).

Catch blok, primer:
Kod za rukovanje izuzetkom datog tipa se ogradjuje catch-blokom.
catch-blok se mora nalaziti neposredno iza try-bloka koji sadrzi kod koji moze izbaciti taj odredjeni
izuzetak. Sastoji se od kljucne reci catch pracene jednim parametrom unutar oblih zagrada kojim se
identifikuje tip izuzetka kojim blok rukuje. Ovo prati kod za rukovanje izuzetkom koji se nalazi unutar
para viticastih zagrada:

```
try{
// kod koji moze izbaciti jedan ili vise izuzetaka
}
catch(ArithmeticException e){
// kod za rukovanje izuzetkom tipa ArithmeticException
}
```

Ovaj catch blok rukuje samo izuzecima tipa ArithmeticException. To povlaci da je to jedina vrsta izuzetaka koja moze biti izbacena u try-bloku. Ako mogu biti izbaceni i drugi, ovo se nece iskompajlirati.

Generalno, parametar za catch blok mora biti tipa Throwable ili neke njene potklase. Ako klasa koju
zadamo kao parametar ima potklase, od catch bloka se ocekuje da procesira izuzetke tog tipa, ali i svih
potklasa tog tipa.

Visestruki catch blok, primer:
Ako try-moze da izbaci nekoliko razlicitih vrsta izuzetaka, tada je potrebno postaviti vise catch blokova
za rukovanje njima nakon try-bloka. Primer:

```
try{
// kod koji moze izbaciti izuzetke...
}
catch(ArithmeticException e){
// kod za rukovanje ArithmeticException izuzecima
}
catch(IndexOutOfBoundsException e){
// kod za rukovanje Index... izuzecima
}
// Izvrsavanje se nastavlja ovde ...
```

U prethodnom primeru izuzeci tipa ArithmeticException bice hvatani prvim catch blokom, a tipa IndexOutOfBoundsException drugim. Naravno, ako se izbaci izuzetak tipa ArithmeticException, bice izvrsen samo kod tog catch bloka. Kada se on zavrsi, izvrsavanje se nastavlja naredbom nakon poslednjeg catch bloka.
Redosled catch blokova moze biti od znacaja. Kada se izbaci izuzetak, on cee biti uhvacen prvim catch
blokom ciji je parametar istog tipa kao izuzetak ili je tipa neke njegove nadklase.
Ako postoji hijerarhija, redosled catch-blokova bi trebalo da bude: najizvedeniji tip prvo, najosnovniji
tip na kraju. Inace, kod se nece prevesti.

Finally blok, primer:
Priroda izuzetaka je takva da se izvrsavanje try bloka prekida po izbacivanju izuzetka bez obzira na
znacaj koda koji sledi tacku u kojoj je izuzetak izbacen. To uvodi mogucnost da izuzetak ostavi stvari u nezadovoljavajucemm stanju.
Na primer, mozemo imati otvoren fajl i posto je izbacen izuzetak, ne izvrsava se kod za zatvaranje
fajla.
finally-blok obezbedjuje sredstvo da se ”pocisti” na kraju izvrsavanja try-bloka.
Finally blok se koristi kada treba da se bude sigurno da se neki odredjeni kod izvrsava pre kraja metoda,
bez obzira koji izuzeci su izbaceni unutar pridruzenog try-bloka.
finally-blok se izvrsava uvek, bez obzira da li su ili ne izbaceni izuzeci za vreme izvrsavanja pridruzenog
try bloka.
Struktura finally bloka je:
finally{
// odgovarajuci kod ...
}

Upravo kao i catch blok, tako je i finally blok pridruzen odredjenom try bloku i mora biti smesten neposredno nakon catch blokova za taj try blok. Ako nema catch blokova, finally blok se smesta neposredno nakon try bloka. Inace se program nece uspesno prevesti. Ukoliko je koriscenjem return naredbe vracena neka vrednost unutar finally bloka, to ponistava return naredbu koja je eventualno izvrsena u try bloku.

```
try{
// Kod koji moze izbaciti izuzetke ...
}
catch(ExceptionType1 e) {
// ...
}
catch(ExceptionType2 e) {
// ...
// ako je potrebno, jos catch blokova ...
}
finally{
// kod koji se uvek izvrsava nakon try-bloka
}
```

Nije moguce da postoji samo try-blok, vec njega uvek mora da prati bar jedan od catch i finally blokova.
Izuzeci koji nisu uhvaceni mogu biti izbaceni bilo gde u telu metoda, u delu koda koji nije ogradjen

try-blokom.

3. Rukovanje izuzecima, propagiranje izuzetaka. Primeri.

U mnogim situacijama, kada se u nekom metodu pojavi izuzetak, programer moze da odluci da taj
izuzetak ne obradjuje na u tom metodu vec da ga omoguci da izuzetak propagira u pozivajuci metod.
Motivacija za takvu odluku je sto je pozivajuci metod u principu svesniji konteksta u kom je izuzetak
nastao, pa se na tom nivou lakse moze odluciti koje akcije treba preduzeti.
Propagiranje izuzetka u
metod-pozivalac se realizuje pomocu kljucne reci throws iza koje sledi lista izuzetaka kojima je dopusteno
propagiranje.

4. Rukovanje izuzecima, Izbacivanje izuzetaka. Primeri.

U mnogim situacijama, kada neki metod uhvati izuzetak, implementiranjem odgovarajuce catch klauze, pozivajuci program mozda mora da zna da se to desilo.
Ako izuzetak koji smo uhvatili treba da
prosledimo pozivajucem programu, mozemo da ga ponovo izbacimo iz unutrasnjosti bloka catch, koristeci
throw naredbu, na primer:

```
try{
// ...
}
catch(ArithmeticException e){
// obrada ovog izuzetka
throw e;
}
```

Kod throw naredbe je data kljucna rec throw za kojom sledi objekat tipa izuzetka koji se izbacuje.
Programer moze odluciti da izbaci izuzetak kad god nadje za shodno, cak i u normalnoj situaciji.
Medjutim, treba naglasiti da su izuzeci neefikasni i da ih treba iskljucivo koristiti za neregularne situacije, a ne za realizaciju delova ”normalne” poslovne logike.

5. Tvrdnje. Primeri.

Tvrdnje dopuštaju da se testira ispravnost bilo koje od pretpostavki koja je napravljena prilikom pisanja programa.
Tvrdnje se u Javi postavljaju pomoću naredbe **assert**.
Tokom izvršavanja tvrdnje, pretpostavka je da je ona tačna. Ako tvrdnja nije tačna, tada JVM izbacuje grešku tipa AssertionError.

Tvrdnje se najčešće koriste za potrebe testiranja prilikom razvoja programa i njihovo procesiranje se po pravilu isključuje kada program pređe produkciju.
Jedinični testovi su u velikoj meri potpisnuli tvrdnje.
Naredba assert se može javiti u dva oblika:

```
assert logicki-izraz;
assert logicki-izraz : niska-izraz;
```

Omogućavanje tvrdnji
Podrazumevano je postavljeno da su tvrdnje onemogućene
Pokretanje Java programa tako da budu omogućene tvrdnje zahteva uključivanje **–ea** opcije za JVM.
Na primer, pokretanje main metoda u klasi Test sadržanoj u datoteci Test.java sa omogućenim tvrdnjama postiže se sa:
java –ea Test

Onemogućavanje tvrdnji
Pokretanje Java programa tako da budu onemogućene tvrdnje zahteva uključivanje **–da** opcije za JVM.
Na primer, pokretanje main metoda u klasi Test sadržanoj u datoteci Test.java sa onemogućenim tvrdnjama postiže se sa:
java –da Test

Osnovni razlog za korišćenje tvrdnji je želja programera da proveri da li važe pretpostavke koje je uveo i na kojima se oslanja njegov programski kod.
Tvrdnjama se obezbeđuje:
-da programski kod koji izgleda nedostupan zaista i jeste nedostupan
-da pretpostavke opisane u komentarima zaista važe
-da se ne događa da u switch naredbi izvršavanje dolazi u default granu
-da se potvrdi ispravnost stanja objekta

Tvrdnje se obično postavljaju ili na početku izvršenja metoda (preduslovi) ili po završetku poziva metoda (postuslovi).

U principu, tvrdnje se koriste za proveru logički nemogućih situacija, tj. situacija koje ne bi smele da se dogode i iz kojih nema regularnog oporavka.
Stoga, za razliku od rada sa „normalnim“ izuzecima, nema smisla da se oni obrađuju tokom izvršavanja kada problem bude detektovan.
Obično se tvrdnje isključuju u produkcionom kodu.

Gde se obično koriste tvrdnje:
-Za proveru argumenata prosleđenih privatnim metodama
-Za proveru ispravnosti prolaska kroz naredbe grananja
-Za proveru važenja datih uslova na početku metoda

Gde u principu **ne** treba koristiti tvrdnje:
-Tvrdnje ne treba da zamene poruke o greškama
-Tvrdnje ne treba da se koriste za kontrolu argumenata u javnim metodama, jer tada argumente može proslediti bilo ko. Tada proveru argumenata treba realizovati kroz mehanizam izuzetaka
-Tvrdnje ne treba koristiti za proveru argumenata komandne linije, jer ne možemo znati unapred ko će i sa kojim argumentima pokretati dati program.

10. ENUMERISANI TIPOVI.

1. Enumerisani tipovi. Karakteristike. Definisanje. Primeri.

Ponekad promenljiva sluzi za čuvanje vrednosti iz ograničenog skupa. Na primer, možda se prodaje
odeca u cetiri veličine: small, medium, large i extra large. Naravno, ove velicine se mogu kodirati celim
brojevima 1, 2, 3, 4 ili znacima S, M, L, X. Ali takav pristup je sklon greskama. Veoma lako se moze
dogoditi da promenljiva dobije pogresnu vrednost.

Pocev od verzije 5.0 moguce je u Javi definisati sopstveni enumerisani tip. Takav tip ima **konacan**
broj imenovanih vrednosti. Na primer, za prethodno opisanu situaciju uvodi se enumerisani tip:
enum Velicina {SMALL, MEDIUM, LARGE, EXTRA_LARGE};

Enumerisani tip se zapravo ponasa kao klasa. Klasa iz prethodnog primera sadrzi tacno cetiri podatka
tj. tacno cetiri primerka - nije moguce konstruisati nove objekte (ne koristimo new).
Prema tome, nema potrebe koristiti metod equals() za poredenje vrednosti enumerisanog tipa, vec se podaci tog tipa porede koriscenjem
operatora ==.

Moguce je deklarisati promenljivu enumerisanog tipa. Na Primer:
Velicina v = Velicina.MEDIUM;

Promenljiva tipa Velicina moze cuvati samo jednu od vrednosti izlistanih u deklaraciji ovog tipa ili
specijalnu vrednost null. Moguce je, po potrebi, dodati konstruktore, metode i polja tipu enumeracije.
Naravno, konstruktori se pozivaju samo prilikom konstruisanja konstanti enumeracije.
Primer:

```
enum Velicina{
	SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");

	private String skracenica;

	private Velicina(String skracenica){
		this.skracenica = skracenica;
	}
	public String getSkracenica(){
		return skracenica;
	}
}
```

2. Enumerisani tipovi. Pretvaranje u nisku i dobijanje iz niske. Primeri.

Svi tipovi enumeracije su potklase klase Enum. Oni nasleduju izvestan broj metoda od te klase. Najko-
risniji medu njima je metod toString(), koji vraca ime konstante enumeracije. Na primer:
Velicina.SMALL.toString(); //vraca string SMALL.
Suprotan efekat ima staticki metod valueOf(). Na primer:
Velicina v = (Velicina) Enum.valueOf(Velicina.class, "SMALL"); //postavlja v na Velicina.SMALL.

Svaki tip enumeracije poseduje staticki metod **values()** koji vraca niz vrednosti enumeracije. Na primer:

Velicina[] vrednosti = Velicina.values();
vraca niz sa elementima Velicina.SMALL, Velicina.MEDIUM, Velicina.LARGE i Velicina.EXTRA LARGE.

Metod **ordinal()** vraca poziciju konstante enumeracije u enum deklaraciji, pri cemu brojanje pocinje
od 0. Tako, na primer, Velicina.MEDIUM.ordinal() vraca 1.

Enumeracije se mogu koristiti kod visestrukog grananja pomocu naredbe switch:

```
Velicina v = . . .;
switch (v){
        case SMALL: // nema potrebe koristiti Velicina.SMALL
        ...
        break;
        ...
}
```

U case klauzama ne navodi se tip enumeracije, vec samo ime konstante.

java.lang.Enum<E>
• static Enum valueOf(Class enumClass, String name) - vraca enumerisanu konstantu datog tipa
koja ima dato ime.
• String toString() - vraca ime enumerisane konstante.
• int ordinal() - vraca poziciju enumerisane konstante u deklaraciji enum, pri cemu odbrojavanje
pocinje od nule.
• int compareTo(E other) - vraca negativan ceo broj ako je data enumerisana kontanta pre konstane
other u nabrajanju, vraca nulu ako je this == other, a inace vraca pozitivni ceo broj.

3. Енумерисани типови. Додатни подаци придружени енумерисном типу. Примери

enum and constructor :

- enum can contain constructor and it is executed separately for each enum constant at the time of enum class loading.
- We can’t create enum objects explicitly and hence we can’t invoke enum constructor directly.

enum and methods :

- enum can contain both **concrete** methods and **abstract** methods. If an enum class has an abstract method, then each instance of the enum class must implement it. E.g.

```
// An enum (Note enum keyword inplace of class keyword)
enum Color
{
    RED, GREEN, BLUE;

    // enum constructor called separately for each constant
    private Color()
    {
        System.out.println("Constructor called for : " +
        this.toString());
    }

    public void colorInfo()
    {
        System.out.println("Universal Color");
    }
}

public class Test
{
    // Driver method
    public static void main(String[] args)
    {
        Color c1 = Color.RED;
        System.out.println(c1);
        c1.colorInfo();
    }
}
```

Output:

```
Constructor called for : RED
Constructor called for : GREEN
Constructor called for : BLUE
RED
Universal Color
```

4. Енумерисани типови. Како направити тип који има карактеристике енумерисаног

Enum with customized value. Now to use this enum in code, there are some points we have to follow:-

We have to create parameterized constructor for this enum class. Why? Because as we know that enum class's object can't be create explicitly so for initializing we use parameterized constructor. And the constructor cannot be the public or protected it must have private or default modifiers. Why? if we create public or protected, it will allow initializing more than one objects. This is totally against enum concept.

We have to create one getter method to get the value of enums. E.g.

```
// Java program to demonstrate how values can be assigned to enums.
enum TrafficSignal
{
    // This will call private enum constructor with one String argument
    RED("STOP"), GREEN("GO"), ORANGE("SLOW DOWN");

    // declaring private variable for getting values
    private String action;

    // getter method
    public String getAction()
    {
        return this.action;
    }

    // enum constructor - cannot be public or protected
    private TrafficSignal(String action)
    {
        this.action = action;
    }
}

// Driver code
public class EnumConstructorExample
{
    public static void main(String args[])
    {
        // let's print name of each enum and there action - Enum values()
examples
        TrafficSignal[] signals = TrafficSignal.values();

        for (TrafficSignal signal : signals)
        {
            // use getter method to get the value
            System.out.println("name : " + signal.name() +
                        " action: " + signal.getAction() );
        }
```

```
    }
}
```

Output:

```
name : RED action: STOP
name : GREEN action: GO
name : ORANGE action: SLOW DOWN
```

Isto mozemo uraditi i ako zelimo da vrednosti dodeljene enum konstantama budu nekog drugog tipa, npr. int. Bitan je privatni konstruktor koji dodeljuje vrednost

// nisam sigurna da se ovo trazi u pitanju, ali je korisna stvar, moze se dodati kod bilo kog pitanja za enum

(za ova pitanja nisam uspela da nadjem u prezentacijama)

11. GENERICKI TIPOVI

1. Pojam generickog tipa. Prednosti generickog tipa. Primeri.

Genericki tip omogucuje da pri definisanju klasa, interfejsa i metoda sami tipovi (tj. klase i interfejsi) budu parametri. Dakle, genericki tipovi se ponasaju slicno kao formalni parametri pri definisanju metoda. Na taj nacin se postize ponovna iskoristivost istog koda za razlicite ulaze.

Programiranje koriscenjem generickih tipova ima sledece prednosti:
• Strozija kontrola tipa prilikom prevodjenja Java programa.
• Eliminacija eksplicitne konverzije tipa (kastovanja).
• Omogucavanje da se implementiraju genericki algoritmi.

Prednosti generickog tipa, primeri

Kod starog pristupa eksplicitna konverzija je dosadna ali sustinski neophodna!

```
List stones = new LinkedList();
stones.add(new Stone(RED));
stones.add(new Stone(GREEN));
stones.add(new Stone(RED));
Stone first = (Stone) stones.get(0);

public int countStones(Color color) {
        int tally = 0;
        Iterator it = stones.iterator();
        while (it.hasNext()) {
                Stone stone = (Stone) it.next();
                if (stone.getColor() == color) {
                        tally++;
                }
        }
return tally;
}
```

U novom pristupu interfejs List je genericki interfejs kome tip elementa liste predstavlja parametar.

```
List<Stone> stones = new LinkedList<Stone>();
stones.add(new Stone(RED));
stones.add(new Stone(GREEN));
stones.add(new Stone(RED));
Stone first = /*no cast*/ stones.get(0);

public int countStones(Color color) {
	int tally = 0; /*no temporary*/
	for (Stone stone : stones) {
		/*no temporary, no cast*/
		if (stone.getColor() == color) {
			tally++;
		}
	}
return tally;
}
```

Stari nacin:

```
List stones = new LinkedList();
stones.add(\ovo nije kamen"); //nema provere
...
Stone stone = (Stone) stones.get(0);  // greska pri izvrsavanju Runetime error
```

Novi nacin:

```
List<Stone> stones = new LinkedList<Stone>();
stones.add(\ovo nije kamen"); // proverava se pri prevodjenju
...
Stone stone = stones.get(0);  //izvrsenje bez problema
```

2. Pojam generickog tipa. Definisanje generickog tipa. Genericki poziv tipa. Primeri.

Genericki tip je u stvari genericka klasa ili interfejs sa parametrizovanim tipovima tj. cija definicija

sadrzi parametre koji predstavljaju tipove. Primer:

Sledeca klasa, nazvana Box, ce biti modifikovana tako da opise koncept generickog tipa. Na pocetku se

radi o jednoj obicnoj klasi.

```
public class Box{

	private Object object;

	public void set(Object object) {
```

```
        this.object = object;
    }
    public Object get() { return object; }
}
```

Genericka klasa se definise na slede ci nacin:

class name<T1, T2, ..., Tn> { /* ... */ }

Sekcija sa parametrima koji predstavljaju tipove, ogranicena sa znacima < i > sledi neposredno iza

imena klase. U toj sekciji se specificiraju parametri koji predstavljaju tipove T1, T2, ..., i Tn.

Sada klasa Box ima sledecu strukturu:

```
public class Box<T> {
    // T je oznaka tipa
    private T t;
    public void set(T t) {
        this.t = t;
    }
    public T get() {
        return t;
    }
}
```

Konvencije za imenovanje parametara koji predstavljaju tip kod generickih tipova, interfejsa i metoda:

Po konvenciji, parametri su oznaceni jednim velikim slovom, sto se na prvi pogled razlikuje od imena

polja, parametara i metoda. Najcesce se koriste sledece oznake:

- E – Elemenat (Element)
- K – Kljuc (Key)
- N – Broj (Number)
- T – Tip (Type)
- V – Vrednost (Value)
- S, U, V - drugi, treci, cetvrti itd. Tip

Genericki poziv tipa, primer

Da bi se iz programskog koda referisalo na genericku klasu Box, potrebno je izvrsiti genericki poziv
tipa, kojim se parametar T menja sa nekom konkretnom vrednoscu, kao sto je Integer:

```
Box<Integer> integerBox;
```

Genericki poziv tipa se moze posmatrati slicno obicnom pozivu metoda, ali umesto sto se vrednost
argumenta prosledjuje metodu u ovom slucaju se prosledjuje argument koji predstavlja tip (u ovom slucaju tip Integer) samoj klasi (u ovom slucaju klasi Box).
Pozivanje generickog tipa se obicno oznacava kao parametarski tip. Za kreiranje primerka klase koristi se kljucna rec new, ali se izmedju imena klase i
zagrada postavlja <Integer>:

```
Box<Integer> integerBox = new Box<Integer>();
```

Programski jezik Java dopusta, pocev od verzije 7, da se u konstruktoru primerka za genericku klasu,
oznaka tipa kod operatora new zameni sa parom znakova <>, dok god prevodilac moze na osnovu
konteksta da odredi kog tipa treba da budu argumenti tipovi. Par znakova <> se u zargonu naziva
dijamant. Na primer, sledecom promenljivom mozemo napraviti instancu klase Box<Ineger>:

```
Box<Integer> integerBox = new Box<>();
```

Argument koji predstavlja tip se moze prilikom generickog poziva zameniti sa konkretnim tipom (npr.
sa Integer ili sa Student) ali se moze zameniti i sa parametrizovanim tipom (npr. sa Box<Double> ili sa
List<String>).

3. Pojam generickog tipa. Genericki metod. Primeri.

Genericki metodi su oni metodi sa parametrima koji predstavljaju tipove. Oni su slicni generickim
tipovima, ali je opseg parametara koji predstavljaju tipove ogranicen na metod u kom su ti parametri
deklarisani.
Sintaksa za genericki metod sadrzi (pre povratnog tipa) parametre koji predstavljaju tip izmedju znaka
< i >, a ti parametri se pisu pre tipa povratne vrednosti u deklaraciji metoda. Primer: Klasa Util sadrzi
genericki metod compare za poredenje dva primerka genericke klase Pair:

```
public class Util {
	// Generic static method
	public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
		return p1.getKey().equals(p2.getKey()) &&
	p1.getValue().equals(p2.getValue());
	}
}
```

Sama klasa Pair ima sledecu strukturu:

```
public class Pair<K, V>{
	private K key;
	private V value;

	// Generic constructor
	public Pair(K key, V value){
		this.key = key;
		this.value = value;
	}
	// Generic methods
	public void setKey(K key) {
		this.key = key;
	}
	public void setValue(V value) {
		this.value = value;
	}
	public K getKey() {
		return key;
	}
	public V getValue() {
		return value;
	}
}
```

Sintaksa za poziv metoda koji poredi primerke klase Pair je sledeca:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.<Integer, String>compare(p1, p2);
```

4. Pojam generickog tipa. Ogranicenja za tipove. Primeri.

Ponekad je potrebno da genericka klasa ili genericki metod postavi ogranicenje na promenljive tipa.
Primer:
Treba odrediti najmanji elemenat niza:

```
class ArrayAlg {
	public static <T> T min(T[] a) { // almost correct
		if (a == null || a.length == 0)
			return null;
		T smallest = a[0];
		for (int i = 1; i < a.length; i++){
			if (smallest.compareTo(a[i]) > 0)
				smallest = a[i];
		return smallest;
		}
	}
}
```

Promenljiva smallest je tipa T, sto znaci da ona moze biti primerak ma koje klase. Odmah se postavlja
pitanje: kako mozemo znati da klasa T sadrzi metod compareTo?
Resenje je da se ogranici parametar tipa T samo na one klase koje implementiraju interfejs Comparable
– standardni interfejs koji sadrzi samo jedan metod, compareTo. Na taj nacin ce biti obezbedjeno da
poziv metoda compareTo ne dovede do greske. Ovo se postize postavljanjem ogranicenja na promenljivu
tipa T:

```
public static <T extends Comparable> T min(T[] a) . . .
```

U ovom slucaju, interfejs Comparable je i sam genericki tip. Sada se genericki metod min moze pozivati
samo nad nizovima primeraka klasa koje implementiraju interfejs Comparable, kao sto su String, Date
itd. Poziv ovog metoda nad nizom objekata neke klase koja ne implementira interfejs Comparable dovodi
do greske u prevodjenju.
Postavlja se pitanje: zasto se u prethodnom slucaju koristi kljucna rec extends, a ne npr. implements –
na kraju krajeva, Comparable je intrefejs, a ne klasa. Notacija
<T extends TipKojiOgranicava>
izrazava da parametarski tip T treba da bude podtip tipa koji ogranicava. Pri tome, i parametarski tip
T i tip koji ogranicava mogu biti i klase i interfejsi.

Kljucna rec extends najbolje opisuje takav odnos medu njima, tj. koncept podtipova - uzevsi u obzir da dizajneri Jave nisu zeleli da dodaju novu kljucnu rec (npr. sub) u jezik.

Parametar tipa moze imati vise ogranicenja. Na primer:

T extends Comparable & Serializable

Tipovi koji ogranicavaju su odvojeni znakom & zato sto se znak zarez koristi za razdvajanje promenljivih
tipa. Isto kao i kod Java nasledivanja, moguce je imati vise interfejsa koji ogranicavaju parametar tipa,
ali samo jedno ogranicenje moze da se odnosi na klasu. Ako klasa predstavlja ogranicenje za tip, naziv
klase se mora navesti pre naziva interfejsa. Primer:
Metod minmax, koji odreduje minimum i maksimum niza, je napravljen tako da bude genericki:

```
import java.util.*;
public class PairTest2 {
	public static void main(String[] args){
		GregorianCalendar[] birthdays ={
			new GregorianCalendar(1906, Calendar.DECEMBER, 9), // G. Hopper
			new GregorianCalendar(1815, Calendar.DECEMBER, 10), // A.
			Lovelac
			new GregorianCalendar(1903, Calendar.DECEMBER, 3), // v.
			Neumann
			new GregorianCalendar(1910, Calendar.JUNE, 22), // K. Zuse
		};
		Pair<GregorianCalendar> mm = ArrayAlg.minmax(birthdays);
		System.out.println("min = " + mm.getFirst().getTime());
		System.out.println("max = " + mm.getSecond().getTime());
	}
}

class ArrayAlg{
	public static <T extends Comparable> Pair<T> minmax(T[] a){
		if (a == null || a.length == 0)
			return null;
		T min = a[0];
		T max = a[0];
		for (int i = 1; i < a.length; i++){
			if (min.compareTo(a[i]) > 0)
				min = a[i];
			if (max.compareTo(a[i]) < 0)
				max = a[i];
		}
	return new Pair<T>(min, max);
	}
}
```

5. Generici i virtuelna masina.

Kad god se definise genericki tip, automatski biva obezbedjen odgovoarajuci sirovi (raw) tip. Ime
sirovog tipa je isto kao ime generickog tipa, samo sto su uklonjeni parametri koji predstavljaju tipove.

Promenljive koje predstavljaju tipove su prosto zamenjene sa tipovima koji ih ogranicavaju ili sa Object
tipom (ako za te promenljive nije bilo ogranicenja). Primer:
Sirovi tip za Pair<T> ima sledeci oblik:

```
public class Pair{
	private Object first;
	private Object second;

	public Pair(Object first, Object second){
		this.first = first;
		this.second = second;
	}
	public Object getFirst() {
		return first;
	}
	public Object getSecond() {
		return second;
	}
	public void setFirst(Object newValue) {
		first = newValue;
	}
	public void setSecond(Object newValue) {
		second = newValue;
	}
}
```

U prethodnom primeru parametar T nije bio ogranicen, pa je stoga on jednostavno zamenjen sa klasom
tj. tipom Object. Mada program moze sadrzati razlicite vrste parova, kao sto su Pair<String> ili
Pair<GregorianCalendar>, tokom prevodjenja svi oni budu prevedeni u sirove Pair tipove. Primer:
Pretpostavimo da je donekle drugacije definisan tip:

```
public class Interval<T extends Comparable & Serializable> implements Serializable{
	private T lower;
	private T upper;
	public Interval(T first, T second){
		if (first.compareTo(second) <= 0) {
			lower = first;
			upper = second;
		}
		else { lower = second; upper = first; }
	}
	. . .
}
```

Tada sirovi Interval tip ima sledeci oblik:

```
public class Interval implements Serializable{
	private Comparable lower;
	private Comparable upper;

	public Interval(Comparable first, Comparable second){
```

```
            if (first.compareTo(second) <= 0) {
                lower = first;
                upper = second;
            }
            else {
                lower = second;
                upper = first;
            }
        }
        ...
}
```

Ako se promeni redosled granica u definiciji klase Interval :
class Interval<Serializable & Comparable>
tada se kod sirovog tipa T zamenjuje sa Serializable i prevodilac ubacuje konverzije u Comparable gde
god je to neophodno.

6. Generici i nasledjivanje.

Uopsteno govoreci, ne postoji veza izmedju Pair<S> i Pair<T>, bez obzira na to kako su S i T povezani.
U Javi je klasa Manager podklasa klase Employee samo ako interfejsi potklase ukljucuju sve interfejse
nadklase. Medjutim, napravljeni generici obicno imaju razlicite interfejse pa ne moze biti nasledivanja.

U primeru koji sledi, prevodilac sprecava prevodjenje koda sa operacijama koje nisu bezbedne za tip
– nije moguce sa listom niski raditi kao sa listom objekata, bez obzira na to sto je klasa String izvedena
iz klase Object.

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls;            //greska pri prevodjenju jer konverzija nije bezbedna za tip

lo.add(0, new Object());    // legalno?!
ls.get(0);                          // Nije string?!
```

Kao zakljucak, potrebno je voditi racuna o sledecimm cinjenicama kada se radi sa Java genericima:
• U radu Java virtuelne masine ne postoje generici, vec samo obicne klase i metodi.
• Svi tipovni parametri bivaju zamenjeni sa svojim granicama.
• Prevodilac generise metode-mostove i na taj nacin zadrzava polimorfno ponasanje.
• Da bi se obezbedila bezbednost tipova, prevodilac tamo gde je potrebno ubacuje naredbe za eksplicitnu konverziju tipova.

POSLEDNJE POGLAVLJE – Elementi funkcionalnog programmiranja u javi:

Ovo nista nema na prez, ali ima ispitnih pitanja ofc

Za pitanja 1 i 2: ima na prezentaciji: https://matfoop.github.io/OOP/predavanja/prezentacije/OOP18-Lambda%20izrazi.pdf

Za pitanja 3-10: https://www.baeldung.com/java-8-functional-interfaces

https://www.geeksforgeeks.org/functional-interfaces-java/

https://www.baeldung.com/java-8-collectors // ovde ima bas dosta, vrv je dovoljno samo osnovno

Za pitanja 11-13: https://www.baeldung.com/java-8-streams

prezentacija ulaz/izlaz: (to su isto tokovi, ali nije bas povezano sa pitanjem)

https://matfoop.github.io/OOP/predavanja/prezentacije/OOP17-Ulaz%20i%20izlaz,%20serijalizacija.pdf

13.REFLEKSIJA

1.Refleksija i klase za refleksiju

-Biblioteka za refleksiju obezbedjuje veoma bogat skup alata za pisanje programa koji manipulisu Java kodom na dinamican nacin.

Moze se koristiti za:

-Analiziranje mogucnosti klasa tokom izvrsavanja(refleksivni program)

-Istrazivanje objekata tokom izvrsavanja

-Implementaciju generickog koda za manipulaciju nizovima

-Koriscenje primeraka klase Method

Svaki element je ili primitivnog ili objektnog tipa, svi referentni tipovi nasledjuju klasu java.lang.Object

Класе, енумератори, низови и интерфејси су референтног типа.

Постоји фиксиран скуп примитивних типова: boolean, byte, short, int, long, char, float и double.

Примери референтних типова су java.lang.String, све класе-омотачи за примитивне типове као што су java.lang.Double, интерфејс java.io.Serializable и енумератор javax.swing.SortOrder.

Klase za refleksiju:

Java.lang.Class - Primerci klase Class predstavljaju klase i interfejse u Java aplikaciji koja se izvrsava, tj tip svakog objekta kreiranog tokom rada Java aplikacije predstavljen je sa primerkom Class

Static Class forName(String className) - vraca Class objekat koji predstavlja klasu sa imenom className

Object newInstance() - vraca novi primerak klase opisane datim Class objektom

Field[] getFields() i Field[] getDeclaredFields() - getFields vraca niz Field objekata koji sadrzi javna polja date klase i njenih nadklasa, getDeclaredField vraca niz Field objekata za sva polja deklarisana u klasi opisanoj datim Class objektom

Method[] getMethods() i Method[] getDeclaredMethods() - vraca niz Method objekata, getMethods javne metode i sadrzi nasledjene metode, a getDeclaredMethods vraca sve metode klase opisane datim Class objektom ili interfejsa, ali rezultat ne sadrzi nasledjene metode

Constructor[] getConstructors i Constructor[] getDeclaredConstructors – vraca niz Constructor objekata koji sadrzi sve javne konstruktore(getConstructors) ili sve konstruktore(getDeclaredConstructors) klase koja je predstavljena datim Class objektom.

Java.lang.reflect - paket koji sadrzi klase i interfejse za podrsku refleksiji

Java.lang.reflect.Member - Interfejs koji odslikava identifikujuce informacije o clanu klase(polju, metodu ili konstruktoru)

Java.lang.reflect.Method - Implementira interfejs Member.Ova klasa obezbedjuje informacije o metodu, kao i pristup tom metodu(bilo da se radi o metodu klase ili metodu interfejsa, bilo da se radi o statickom metodu ili metodu primerka)

Public Object invoke(Object implicitParameter, Object[] explicitParameters) - Poziva metod koji je opisan datim Method objektom, tako sto mu prosledi date parametre i kao rezultat vraca rezultat izvrsenja metoda, za staticki metod implicitni parametar je null, a ako su parametri primitivnog tipa tada se prosledjuju vrednosti objekata omotaca

Class getDeclaringClass() - vraca primerak klase Class za klasu u kojoj je definisan ovaj metod

Class getExceptionTypes() - vraca niz objekata tipa Class, koji predstavljaju tipove izuzetaka koje moze izbaciti ovaj metod

Int getModifiers() - vraca ceo broj koji opisuje modifikatore ovog metoda. Za analizu vracene vrednosti koriste se metode iz klase Modifier

- String getName()
  враће ниску која представља име метода.
- Class[] getParameterTypes()
  враће низ Class објеката који представља типове параметара за метод.
- Class getReturnType()
  враће примерак класе Class који представља тип који враће дати метод.

Java.lang.reflect.Field - klasa Field implementira interfejs member, ona obezbedjuje informacije o jednom polju(ili statickom ili polju primerka).Takodje obezbedjuje i dinamicki pristup polju tj ocitavanje i modifikaciju njegovog sadrzaja

Object get(Object obj) - vraca vrednost polja koje je opisano Field objektom obj

Void set(Object obj, Object newValue) - postavlja vr polja opisanog Field objektom obj na vr new

Class getDeclaringClass() - vraca primerak klase Class za klasu koja sadrzi ovo polje

- int getModifiers()
  враће цео број који описује модификаторе овог поља.
- String getName()
  враће ниску која представља име поља.

Java.lang.reflect.Package - Objekti primerci ove klase sadrze informacije o implementaciji i specifikaciji java paketa

Java.lang.reflect.Constructor - Implementira interfejs Member,Obezbedjuje informacije i pristup konkretnom konstruktoru klase

Object newInstance(Object[] args) - kreira novi primerak klase

Class getDeclaringClass() - vraca primerak klase Class za klasu u kojoj je definisan dati konstruktor

String getName() - vraca nisku koja predstavja ime konstruktora

- Class getDeclaringClass()
  враће примерак класе Class за класу у којој је дефинисан дати конструктор.
- Class[] getExceptionTypes()
  враће низ објеката типа Class, који представљају типове изузетака које може избацити овај конструктор.
- int getModifiers()
  враће цео број који описује модификаторе овог конструктора. За анализу враћене вредности користе се методе дефинисане у класи Modifier.

Java.lang.reflect.Modifier - ova klasa obezbedjuje staticke metode i konstante za dekodiranje modifikatora,skup modifikatora je enkodiran kao ceo broj gde razliciti bitovi ukazuju da li je u skup ukljucen dati modifikator

Static boolean isAbstract(int modifiers), isFinal,isInterface,isPrivate,isNative,isProtected,isPublic

Java.lang.reflect.AccessibleObject - nadklasa klase field, pomocu nje se definise mogucnost pristupa objektu

Void setAccessible(boolean flag)

Boolean isAccessible

- static void setAccessible(AccessibleObject[] array, boolean flag)
  погодан метод за постављање маркера доступности flag за читав низ објеката.

2,3,4,5,6

Za svaki prethodno izbrojani element, JVM formira nemutirajuci primerak klase java.lang.Class koji obezbedjuje metode za istrazivanje runtime osobina objekta, ukljucujuci informacije o poljima, metodama i tipovima. Primerak klase ‘Class’ omogucuje da se u letu kreiraju nove klase i objekti – ova klasa predstavlja ulaznu tacku za Reflection API

Referenca na objekat tipa Class se moze dobiti pozivom metoda getClass() nad primerkom date klase:

Class c = mystery.getClass(); ili Class c = mysteryClass.class; ili Class c = Class.forName(“mysteryClass”);

Referenca na objekat tipa Class koji predstavlja nadklasu datog objekta:

Class s = c.getSuperclass();

Odredjivanje imena klase:

String s = c.getName();

Испитивање модификатора

Пример. Приликом извршавања следећег кода:

```
Employee e;
e = new MonthlyEmployee();
Class c = e.getClass();
 int m = c.getModifiers();
 if (Modifier.isPublic(m))
        System.out.println("public");
 if (Modifier.isAbstract(m))
        System.out.println("abstract");
 if (Modifier.isFinal(m))
       System.out.println("final");
```

добија се следећи излаз:

```
public final
```

Odredjivanje interfejsa koji implementira data klasa:

Class[] interfaces = c.getInterfaces();

Odredjivanje polja date klase:

Field[] fields = c.getFields();

Odredjivanje metoda date klase:

Method[] methods = c.getMethods();

Jednostavni primeri:

1)

Class c = “foo”.getClass();

Class c2 = System.console().getClass();

Enum E{A, B};

Class c3 = A.getClass();

Byte[] bytes = new byte[1024];

Class c4 = bytes.getClass();

2)

```
Boolean b;
Class c = b.getClass();//neispravno daje compile time error
Class c2 = boolean.class;//ispravno
```

3)

```
Class c = Class.forName("com.duke.MyLocaleServiceProvider");
Class cDoubleArray = Class.forName("[D");//double[].class
Class cStringArray = Class.forName("[[Ljava.lang.String;");
```

Veci primeri:

Primer 1: Ilustruje kako ispitati koje klase nasledjuje i interfejse implementira data klasa

```
Public static void showType(String className) throws ClassNotFoundException {
Class thisClass = Class.forName(className);
String flavor = thisClass.isInterface() ? "interface" : "class";
        System.out.println(flavor + " " + className);
        Class parent = thisClass.getSuperclass();
        If(parent != null) {
System.out.println("extends" + parent.getName());
}
Class[] interfaces = thisClass.getInterfaces();
For(int i = 0;i < interfaces.length;++i) {
System.out.println("implements" + interfaces[i].getName());
}
}
```

Dobijaju se sledeci rezultati prilikom izvrsavanja:

```
class java.lang.Object
class java.util.HashMap
        extends java.util.AbstractMap
        implements java.util.Map
        implements java.lang.Cloneable
        implements java.io.Serializable
Class Point
extends java.lang.Object
```

Primer 2: Ilustruje kako ispitati koje metode sadrzi data klasa

```
static void showMethods(Object o) {
        Class c = o.getClass();
        Method[] theMethods = c.getMethods();
        for (int i = 0; i < theMethods.length; i++) {
                String methodString = theMethods[i].getName();
                System.out.println("Name: " + methodString);



                System.out.println(" Return Type: " +
        theMethods[i].getReturnType().getName());
                 Class[] parameterTypes = theMethods[i].getParameterTypes();
                System.out.print(" Parameter Types:");
                for (int k = 0; k < parameterTypes.length; k ++) {
                        System.out.print(" " + parameterTypes[k].getName());
                }
                System.out.println();
        }
 }
```

Dobija se izlaz oblika:

```
Name: equals
        Return Type: boolean
        Parameter Types: java.lang.Object
Name: getClass
        Return Type: java.lang.Class
        Parameter Types:
Name: intersects
        Return Type: boolean
        Parameter Types: double double double double
```

7.Citanje vrednosti za polja

```
Field fields[] = c.getFields();
for(int i = 0; i < fields.length; i++) {
        System.out.print(fields[i].getName() + "= ");
        System.out.println(fields[i].getInt(e));
}
```

На излазу се добија:
number= 111
level= 12

e је примерак класе Employee или њене подкласе

Уочава се да на излазу нису приказана поља која нису јавна.

Метод getDeclaredFields враће сва поља која су декларисана у класи, али искључује поља наслеђена из надкласа.

8.Postavljanje vrednosti za polja

Пример. Увећање вредности поља level објекта e за 1 се постиже следећом секвенцом наредби:

```
Employee e;
e = new MonthlyEmployee();
Class c = e.getClass();
Field f = c.getField("level");
f.setInt(e,f.getInt(e)+1);
```

9.Poziv metoda primerka

Математички факултет

vladaf@matf.bg.ac.rs

Позив метода примерка

Може се позвати метод објекта тј. примерка дате класе, у ком случају се при позиву морају проследити и имплицитни и експлицитни аргументи.

Пример. Позив метода print објекта e се постиже следећом секвенцом наредби:

```
Employee e = new HourlyEmployee();
Class c = e.getClass();
Method m = c.getMethod("print", null);
m.invoke(e, null);
```

Као резултат, на излазу се добија:

```
I'm a Hourly Employee
```

10.Dinamicko kreiranje objekata

Динамичко креирање објекта

За позивање конструктора са аргументима, потребно је користити класу Constructor:

```
Constructor c = ...
Object newObject = c.newInstance( Object[] initArguments )
```

Пример. Нека је класа UniversalPrinter дефинисана на следећи начин:

```
class UniversalPrinter {
  public void print(String empType) {
    Class c = Class.forName(empType);
    Employee emp = (Employee ) c.newInstance();
    emp.print();
  }
}
```

Шта је резултат извршавања следећег кода?

```
UniversalPrinter p = new UniversalPrinter();
String empType;
empType = "HourlyEmployee";
p.print(empType);
empType = "MonthlyEmployee";
p.print(empType);
```

11.Refleksija i nizovi

Рефлексија и низови

java.lang.reflect.Array

- static Object get(Object array, int index)
- static xxx getXxx(Object array, int index)
  (xxx је један од примитивних типова boolean, byte, char, double, float, int, long, short.) Ови методи враћу вредност датог низа која се налази на датој позицији index.
- static void set(Object array, int index, Object newValue)
- static setXxx(Object array, int index, xxx newValue)
  (xxx је један од примитивних типова boolean, byte, char, double, float, int, long, short.) Ови методи смештају нову вредност newValue у дати низ на дату позицију index.
- static int getLength(Object array)
  враће дужину датог низа.
- static Object newInstance(Class componentType, int length)
- static Object newInstance(Class componentType, int[] lengths)
  враће нови низ чије су компоненте датог типа componentType, а чија је димензија одређена другим аргументом.

Пример. Илуструје како се креира и манипулише низовима чије димензије нису познате до извршења програма.

```
public static void testArray()
{
   Class cls = String.class;
   int i=10;
   Object arr = Array.newInstance(cls, i);
   Array.set( arr, 5, "this is a test");
   String s = (String) Array.get(arr, 5);
   System.out.println(s);
}
```

12.Implementacija refleksije u Javi

Имплементација Јава рефлексије

Током извршавања Јава програма, JVM учитава бајт-код тј. class датотеке и креира објекте који представљају те класе.

Објекат који представља класу садржи име (поље типа String), листу поља (свако је типа Field), листу метода…

```
class Field {
   String name;
   Class  type;
   Class  clazz;
   int offset;

   Object get(Object obj) {
      if (clazz.isInstance(obj)) {
            f = ((char*)obj) + offset;
            return (type.primitive = TRUE ? wrap(f)  :  (Object)f);
      }
   }
}
```

```
class Class {
            String   name;
            Field[]  fields;
            Method[] methods;
            boolean  primitive;

            bool isInstance...
            Object newInstance..
}
```

13.Ogranicenja refleksije

- Рефлексија је искључиво самоиспитивање
  - Није могуће додати/модификовати поља (тј. мењати структуру класе)
  - Није могуће додати/модификовати методе (тј. мењати понашање)
- Преко рефлексије није доступна имплементација
  - Рефлексијом се не одсликава програмерска логика
- Велики утицај на перформансе
  - Код је много спорији него у случају када се иста операција реализује директним путем…
- Резултујући код је веома комплексан

14.Zabeleske(anotacije)

1.Zabeleske i njihov format

Anotacije obezbedjuju informacije o programu a same po sebi nisu deo programa

-one su specijalna vrsta komentara – ne uticu na ponasanje prog

-one su meta opisi – za razliku od komentara njima mogu pristupiti i koristiti drugi razvijeni programski alati

Svrhe anotacija: Dodatne informacije za prevodioc – radi detektovanja gresaka ili uskracivanja upozorenja,Procesiranje u vremenu prevodjenja i isporucivanja – softverski alati procesiraju anotacije radi generisanja koda, XML datoteka, itd, Procesiranje u vremenu izvrsavanja – neke anotacije dostupne za ispitivanje u vremenu izvrsavanja

Anotacije se mogu primeniti na deklaracije, tada se one pisu u novoj liniji

Format anotacija:

@Entity, @ je znak za anotaciju

Primer:

@Override

Void mySuperMethod() {…}

Anotacija moze da sadrzi elemente bilo imenovane ili ne, i da postave vr za elemente:

@Author(

Name=”Benjamin”

Date=”../../....”

)

Class myClass{…}

Ili

@SuppressWarnings(value = “unchecked”)

Void myMethod() {…}

Ako postoji samo jedan imenovani element, tada se ime el moze ukloniti pri dodeli vrednosti:

@SuppressWarnings(“unchecked”)

Ako nema elemenata:

@EBook

2.Tipovi anotacija

-novonapravljeni i predefinisani

-predefinisani – vec postoje u jeziku java u paketi java.lang:

Deprecated, Override, SuppressWarnings, SafeVarargs

Deprecated:ukazuje da oznaceni element vise nije potreban(zastareo je), kad god program koristi zastareo element java prevodilac generise upozorenje, takodje je potreban tag @deprecated

Primer:

//Javadoc komentar

/**

*@deprecated

*objasnjenje zasto je metod nepotreban i kako se preporucuje da radi

*/

@Deprecated

Static void deprecatedMethod() { }

Override: informise prevodilac da anotirani element treba da prevazidje element koji je deklarisan u nadklasi

Primer:

// mark method as a superclass method

@Override

Int overridenMethod() { }

Ako metod nekorektno prevazilazi element iz nadklase dobijamo gresku

SuppressWarnings: Nalaze prevodiocu da ne pokazuje konkretni tip upozorenja, za koje prevodilac obicno generise upozorenje

Primer: u kodu se koristi zastareli deprecatedMethod, za koji prevodilac obicno generise upozorenje

@SuppressWarnings(“deprecation”)//ova anotacija donjeg metoda blokira generisanje jednog tipa upozorenja

Void useDeprecatedMethod() {

ObjectOne.deprecatedMethod();

}

Postoje dve kategorije upozorenja: zastarelost i neproverenost(kad se radi sa starim kodom koji je pisan pre razvoja generickih klasa i metoda)

@SuppressWarnings({“unchecked”,”deprecation”}) //ovo uklanja obe vrste upozorenja

SafeVarargs: kada se primeni na metod ili konstruktor, obezbedjuje da se u telu metoda/konstruktora ne izvrsavaju operacije koje mogu biti nesigurne za parametre promenljivog tipa ovog metoda/konstruktora, tj za varargs parametre. Ako se koristi ovaj tip anotacije tada se kod prevodioca blokira i generisanje upozorenja neproverivosti koji bi se odnosili na koriscenje varargs parametara(@SafeVarargs)

FunctionalInterface: ukazuje da deklaracija treba da bude funkcionalni interfejs

3.Kreiranje novog tipa anotacije

Slicno je kreiranju interfejsa pri cemu deklaraciji tipa anotacije prethodi @, anotacija ne sme da sadrzi extends ali anotacije implicitno nasledjuju interfejs Annotation, telo anotacije se sastoji od deklaracije metoda(bez tela) a metodi unutar tela anotacije se ponasaju kao polja

Primer:

Пример. Тип анотације Description, којом се нпр. описује програмска датотека, може да има следећи облик:

```
@Retention( RetentionPolicy.RUNTIME )
@interface Description
{
  String author();
  String date();
}
```

Када анотација датог типа придружује декларацији, потребно је обезбедити одговарајуће вредности за чланове типа анотације.

Пример (наставак). Када је креиран тип анотације Description, тада се њиме могу анотирати класе и методе (нпр. Test и testMethod):

```
@Description( author = "Vlado", date = "22/10/2011,23/10/2011" )
public class Test {
  @Description( author = "Vlado", date = "22/10/2011" )
  public static void testMethod(){
    System.out.println( "Welcome to Java" );
    System.out.println( "This is an example of Annotations" );
  }
  public static void main( String args[] ){
    testMethod();
    showAnnotations();
  }
}
```

4.Meta anotacije

Primer: u prethodnom primeru se koristila anotacija @Retention, ona je predefinisana i sluzi za anotiranje anotacija pa se zato zove meta anotacija, one su deklarisane u paketu java.lang.annotation

To su: @Retention, @Documented, @Target, @Inherited

Retention: odredjuje kako se pamti anotacija

RetentionPolicy.SOURCE - anotacija se pamti samo na nivou izvornog koda i biva ignorisana od strane prevodioca

RetentionPolicy.CLASS - anotacija se pamti od strane prevodioca pri prevodjenju, ali je JVM ignorise

RetentionPolicy.RUNTIME - pamti se od strane JVM

Documented: oznacava da anotacija koja je markirana sa Documented treba da bude dokumentovana uz pomoc javadoc alata

Target: oznacava ogranicenje tipa java elementa na koji se moze primenjivati tako markirana anotacija, moguci ciljevi markirane anotacije su:

ElementType.ANNOTATION_TYPE primenjuje se na tip anotacije

ElementType.CONSTRUCTOR primenjuje se na konstruktor

ElementType.FIELD na polje ili na osobinu

ElementType.LOCAL_VARIABLE primenjuje se na lokalnu promenljivu

ElementType.METHOD primenjuje se na metod

ElementType.PACKAGE na deklaraciju paketa

ElementType.PARAMETER na parametar metoda

ElementType.TYPE na bilo koji element klase

Inherited: ukazuje da tip anotacije moze biti nasledjen iz nadklase, sto nije podrazumevano ponasanje

Kada korisnik ispituje tip anotacije, a klasa nema anotaciju datog tipa, tada ce se nadklasa ispitivati za dati tip anotacije

Repeatable: oznacena anotacija moze biti vise puta primenjena na istu deklaraciju

5.Ispitivanje anotacija

Reflection API sadrzi metoda za ispitivanje anotacija

Stari nacin za ispitivanje: getAnnotation(Class <T>), vraca jednu anotaciju

Noviji nacin: getAnnotationByType(Class <T>) klase AnnotatedElement, pod pretpostavkom da postoji anotacija datog tipa

U Java SE8 su ukljuceni dodatni metodi koji prolaze kroz kontejner anotacija i jednim pozivom vraca sve anotacije datog tipa: getAnnotations(Class <T>) klase AnnotatedElement

Пример. Метод који следи приказује анотације:

```
public static void showAnnotations()
{
  Test test = new Test();
  try{
    Class c = test.getClass();
    Description annotation1 = (Description) c.getAnnotation( Description.class );
    System.out.println( "Name of the class: " + c.getName() );
    System.out.println( "Author of the class: " + annotation1.author() );
    System.out.println( "Date of Writing the class: " + annotation1.date() );
    Method m = c.getMethod( "testMethod" );
    Description annotation2 = m.getAnnotation( Description.class );
    System.out.println( "Name of the method: " + m.getName() );
    System.out.println( "Author of the method: " + annotation2.author() );
    System.out.println( "Date of Writing the method: " + annotation2.date() );
  } catch (NoSuchMethodException ex){
     System.out.println( "Invalid Method..." + ex.getMessage() );
  }
}
```

POSLEDNJA OBLAST: KOLEKCIJE I KATALOZI

Kolekcije:Vector, Stack, Hashtable, BitSet, interfejs Enumeration

1.Interfejs i implementacija kod kolekcija

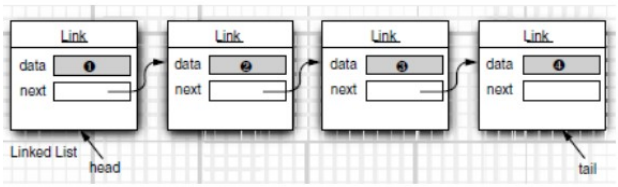
-interfejs odvojen od implementacije(nacin odvajanje posle prikazan na strukturi red)

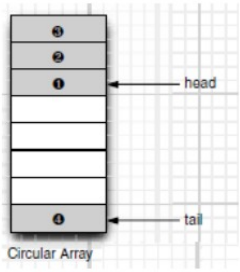
Minimalna forma interfejsa za red ima sledeci oblik:

```
// pojednostavljena verzija reda iz standardne biblioteke
interface Queue<E> {
        void add(E element);
        E remove();
        int size();
}
```

-Interfejs ne govori nista o tome na koji ce nacin red biti implementiran(npr kruzni niz, povezana lista)

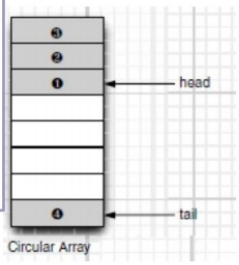
неки трећи начин).



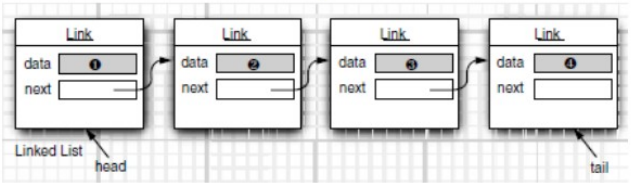


- Свака од имплеметација је одређена класом која имплементира интерфејс Queue.

```
class CircularArrayQueue<E> implements Queue<E> {

        CircularArrayQueue(int capacity) { . . . }
        public void add(E element) { . . . }
        public E remove() { . . . }
        public int size() { . . . }

        private E[] elements;
        private int head;
        private int tail;
}
```




```
class LinkedListQueue<E> implements Queue<E> {

        LinkedListQueue() { . . . }
        public void add(E element) { . . . }
        public E remove() { . . . }
        public int size() { . . . }

        private Link head;
        private Link tail;
}
```



Kada program koristi kolekciju, on ne mora znati koja je implementacija kolekcije stvarno koriscena pa ima smisla da se konkretna klasa koristi samo pri kreiranju objekta, a da se za cuvanje reference na objekat koristi tip interfejsa.

```
Queue<Customer> expressLane = new CircularArrayQueue<Customer>(100);
expressLane.add(new Customer("Harry"));
```

Ili sa LinkedListQueue:

```
Queue<Customer> expressLane = new LinkedListQueue<Customer>();
expressLane.add(new Customer("Harry"));
```

-Interfejs ne govori nista o efikasnosti implementacije

-Kruzni niz je nesto efikasniji od kruzne liste, ali je to uobicajeno pa je za njegovo koriscenje cena veca

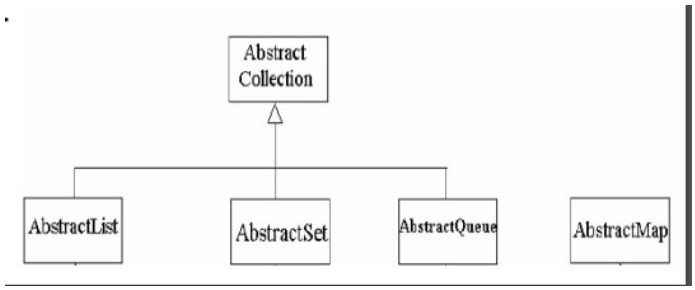
-Kruzni niz je orgranicena kolekcija i on ima konacan kapacitet

2.Apstraktne klase kao osnova za kolekcije

Postoji jos jedan skup klasa cije ime pocinje sa Abstract, kao sto je klasa AbstractQueue

Ove klase treba da koriste programeri koji implementiraju biblioteke klasa

U onim situacijama kada programer sam treba da implementira svoju klasu za red, to ce lakse realizovati ukoliko nasledi klasu AbstractQueue nego ako odluci da implementira sve metode interfejsa Queue



3.Kolekcije i iteratori. Primeri, 4.Interfejs Collection, 5.Interfejs Iterator, 6.Interfejs Iterable i kolekcijski for ciklus

Osnovni interfejs za kolekcijske klase u Javi je interfejs Collection. Ovaj interfejs sadrzi dva najvaznija metoda:

```
public interface Collection<E> {
        boolean add(E element);
        Iterator<E> iterator();
        . . .
}
```

Метод add додаје елеменат у колекцију. Овај метод враћа true ако је додавање елемента заиста променило колекцију, а враћа false ако је колекција непромењена.

На пример, ако се додаје у скуп елеменат који се већ налази у том скупу, тада захтев за додавање нема ефекта јер скуп одбија дупликате, па метод add враћа false .

-Metod iterator vraca objekat koji implementira interfejs Iterator, tako dobijeni objekat-iterator moze se koristiti da se redom, jedan po jedan, obidju elementi u kolekciji

```
public interface Iterator<E> {
        E next();
        boolean hasNext();
        void remove();
}
```

Ponovljenim pozivom next mogu se obici svi elementi u kolekciji, ali ako se stigne do kraja kolekcije i iskoristi se next bice izbacen izuzetak NoSuchElementException, pa je

zato potrebno da se hasNext poziva pre metoda next, hasNext vraca true ako objekat-iterator ima jos elemenata koje treba da poseti

```
Collection<String> c = . . .;
Iterator<String> iter = c.iterator();
while (iter.hasNext()) {
        String element = iter.next();
        //radi nesto sa elementom
}
```

- Почев од Јава 5.0, претходни циклус се може краће и елегантније записати коришћењем "for each" циклуса:

```
for (String element : c) {
        //ovo implicitno pravi iterator nad kolekcijom c
}
```

- Преводилац једноставно преводи "for each" циклус у циклус са итератором.
- Циклус "for each" добро функционише са сваким објектом који имплементира интерфејс Iterable, који садржи само један метод:

```
public interface Iterable<E> {
        Iterator<E> iterator();
}
```

-kako interfejs Collection prosiruje interfejs Iterable, for each se moze koristiti sa bilo kojom kolekcijom, red prosledjivanja elemenata zavisi od tipa kolekcije:

Ako se iterira kroz ArrayList, tada iterator pocinje od indeksa 0

Ako se iterira kroz HashSet, bice nabrajani u sustinski slucajnom redosledu

Kod Java iteratora je dohvatanje elementa cvrsto vezano sa promenom pozicija – iterator prilikom poziva next prelazi preko elementa tj napreduje za jednu poziciju(zamislis 4 kockice i nalazis se na prvoj, iterator je na kraju prve, pozivas next, element postaje druga kockica a iterator se premesta na mesto posle druge kockice, iterator vraca referencu na element preko kojeg je upravo presao)

Metod remove interfejsa iterator uklanja element vracen poslednjim pozivom next, primer:

Пример. Уклањање првог елемент из колекције ниски c:

```
Iterator<String> it = c.iterator();
it.next(); // preskoci ce prvi element
it.remove(); // sada ce ukloni
```

!Ukoliko se remove pozove pre next dolazi do IllegalStateException, primer:

- **Пример.** Ако треба да се уклоне два суседна елемента, не може се просто позвати:

```
it.remove();
it.remove(); // Greska!
```

- Уместо тога, прво се мора позвати метод next како би итератор прешао преко елемента који треба уклонити:

```
it.remove();
it.next();
it.remove(); // OK
```

7.Operacije nad kolekcijom koriscenjem iteratora, Primeri

Java.util.Collection<E>:

-Iterator<E> iterator() - vraca iterator koji se koristi za pristup elementima kolekcije

-int size() - vraca broj elemenata u kolekciji

-boolean isEmpty() - vraca true ako je kolekcija prazna

-boolean contains(Object obj) - vraca true ako kolekcija sadrzi objekat jednak objektu obj

-boolean containsAll(Collection <? Extends E> other) - vraca true ako kolekcija sadrzi sve el kolekcije other

-boolean add(Object element) - dodaje el u kolekciju, vraca true ako je to uspesno uradio

-boolean addAll(Collection <? Extends E> other) - dodaje sve el iz kolekcije other, vraca true ako je uspesno

-boolean remove(Object obj) - brise obj iz kolekcije i vraca true ako je uspelo brisanje

-boolean removeAll(Collection <? Externds E> other) - brise iz ove kolekcije sve elemente other kolekcije i vraca true ako to uspe

-void clear() - brise ceo sadrzaj kolekcije

-boolean retainAll(Collection <? Extends E> other) - brise sve elemente iz kolekcije koje ne pripadaju other

-Object[] toArray() - vraca sve elemente kolekcije kao niz objekata

8.Interfejs List, primeri

Java.util.List<E>

- **ListIterator<E> listIterator()**
  враће итератор за приступ елементима листе
- **ListIterator<E> listIterator(int index)**
  враће итератор за приступ елементима листе, код које ће први позив next() вратити елемент са који је на позицији index
- **void add(int i, E element)**
  додаје елемент на одређену позицију i
- **void addAll(int i,Collection<? Extends E> elements)**
  додаје елементе из колекције од одређене позиције
- **E remove(int i)**
  брише и враћа елемент на датој позицији
- **E set(int i, E element)**
  замењује елемент на датој позицији са новим елементом и враћа стари
- **int indexOf(Object element)**
  враћа прву позицију на којој је пронађен дати елемент или -1,уколико исти није откривен

- **int lastIndexOf(Object element)**
  враћа последњу позицију на којој је пронађен дати елемент или -1,уколико га нема
- **boolean remove(Object obj)**
  брише obj из колекције. Враће true ако је успело брисање.
- **boolean removeAll(Collection<? extends E> other)**
  брише из ове колекције све елементе other колекције. Враћа true ако је успело брисање.
- **void clear()**
  брише цео садржај колекције.
- **boolean retainAll(Collection<? extends E> other)**
  брише све елементе из колекције који не припадају колекцији other.
- **Object[] toArray()**
  враћа све елементе колекције као низ објеката
- **void add(E new Element)**
  додаје елемент пре текуће позиције.

9.Interfejs ListIterator, primeri

- **void set(E new Element)**
  замењује последњи елемент посећен са next или previous са новим елементом.
- **boolean hasPrevious()**
  враћа true ако постоји елемент ком се може приступити при итерирању уназад
- **E previous()**
  враћа претходни објекат.Ако смо на почетку листе избацује се изузетак NoSuchElementException
- **int nextIndex()**
  враће индекс елемента који ће вратити наредни позив метода next
- **int previousIndex()**
  враће индекс елемента који ће вратити наредни позив метода previous

10.Interfejs Set, primeri

- **Iterator<E> iterator()**
  враће итератор који се користи за приступ елементима скупа.
- **int size()**
  враће број елемената у скупу.
- **boolean isEmpty()**
  враће true ако је скуп празан.
- **boolean contains(Object obj)**
  враће true ако скуп садржи објекат једнак објекту obj.
- **boolean containsAll(Collection<? Extends E> other)**
  враће true ако скуп садржи све елементе колекције other.
- **boolean add(Object element)**
  додаје element у скуп, враће true ако је успело додавање.
- **boolean addAll(Collection<? Extends E> other)**
  додаје све елементе колекције other, враће true ако је успело додавање.

- **boolean remove(Object obj)**
  уклања obj из скупа. Враће true ако је успело уклањање.
- **boolean removeAll(Collection<? Extends E> other)**
  брише из скупа све елементе колекције other. Враће true ако је успело брисање.
- **void clear()**
  брише цео скуп.
- **boolean retainAll(Collection<? Extends E> other)**
  брише све елементе из скупа који не припадају колекцији other.
- **Object[] toArray()**
  враћа све елементе скупа као низ објеката

11.Uredjenje u kolekciji

-Kako se odrejuje na koji nacin ce biti sortirani el u kolekciji?

Obicno se pretpostavlja da se u kolekciju dodaju elementi koji implementiraju interfejs Comparable

```
public interfejs Comparable<T> {
        int compareTo(T other);
}
```

- Позив a.compareTo(b) враћа:

1. 0 ако су a и b једнаки,
2. негативан цео број ако a треба да се налази испред b у сортираном поретку,
3. или позитиван цео број ако a треба да се налази иза b.

Data klasa moze implementirati interfejs samo jednom

-Ali sta ako se sortira kolekcija po odredjenom kriterijumu? Sta da se ucini ako su potrebni sortirani primerci klase ciji dizajner nije implementirao interfejs Comparable?

Moze se obezbediti da kolekcija koristi razlicite metode poredjenja, tako sto se konstruktoru kolekcije prosledjuje objekat koji implementira interfejs Comparator

- Интерфејс Comparator декларише метод compare, који прихвата два параметра:

```
public interfejs Comparator<T> {
        int compare(T a, T b);
}
```

- Метод compare интерфејса Comparator се понаша исто као и и compareTo метод интерфејса Comparable.
- На пример, за сортирање елемената по њиховом опису, једноставно се дефинише класа која имплементира Comparator интерфејс:

```
class ItemComparator implements Comparator<Item> {
        public int compare (Item a , Item b) {
                String descrA = a.getDescription();
                String descrB = b.getDescription();
                return descrA.compareTo(descrB);
        }
}
```

- Примећује се да овако направљен компаратор нема променљивих. Он је само власник метода за поређење.
- Такав објекат се обично назива функцијски објекат.
- Функцијски објекти су обично заједнички дефинисани при креирању, као примерци анонимних унутрашњих класа:

```
SortedSet<Item> sortByDescription = new TreeSet<Item>( new
        Comparator<Item>() {
                public int compare(Item a, Item b) {
                        String descrA = a.getDescription();
                        String descrB = b.getDescription();
                        return descrA.compareTo(descrB);
                }
        });
```

java.lang.Comparable<T>

- **int compareTo(T other)**
  Упоређује два објекта и враће негативну вредност ако текући објекат треба да буде испред објекта other, нулу ако се сматрају идентичним у сортираном поретку, или позитивну вредност ако текући објекат треба да буде после other-а.

java.lang.Comparator<T>

- **int compare(T a, T b)**
  Упоређује два објекта и враћа негативну вредност ако је a испред b, нулу ако се сматрају идентичним у сортираном поретку, или позитивну вредност ако је a иза b.

java.util.SortedSet<E>

- **Comparator<? super E> comparator()**
  Враће компаратор коришћен за сортирање елемената, или null ако су елементи упоређивани методом compareTo интерфејса Comparable.
- **E first()**
- **E last()**
  Враће први или последњи елемент у сортираној колекцији.

12. ^ evo ga ovde

13.Red - interfejs Queue

- Редови омогућују ефикасно додавање елемената на крај и уклањање елемената са почетка.
- Ред са два краја, (енг. deque), омогућује ефикасно додавање и уклањање елемената и са почетка и са краја.
- Додавање елемената у средину није подржано.
- Java SE 6 уводи интерфејс Deque. Овај интерфејс имплементирају класе ArrayDeque и LinkedList, при чему обе обезбеђују колекције чија величина расте по потреби.

java.util.Queue<E>

- **boolean add(E element)**
- **boolean offer(E element)**
  додаје дати елемент на крај и враћа true када ред није пун. Ако је ред пун, први метод избацује IllegalStateException, а други враћа false.
- **E remove()**
- **E poll()**
  уклања и враћа елемент са почетка реда када ред није празан. Ако је ред празан, први метод избацује NoSuchElementException, а други враћа null.
- **E element()**
- **E peek()**
  враћа елемент са почетка реда не уклањајући га када ред није празан. Ако је ред празан, први елемент избацује NoSuchElementException, а други враћа null.

14.Interfejs Deque

java.util.Deque<E>

- **void addFirst(E element)**
- **void addLast(E element)**
- **boolean offerFirst(E element)**
- **boolean offerLast(E element)**
  додаје дати елемент на почетак или крај када ред није пун. Ако је ред пун, прва два метода избацују IllegalStateException, а последња два враћају false.
- **E removeFirst()**
- **E removeLast()**
- **E pollFirst()**
- **E pollLast()**
  уклања и враћа елемент са почетка или краја када ред није празан. Ако је ред празан, прва два метода избацују NoSuchElementException, а последња два враћају null.
- **E getFirst()** **E getLast()**
- **E peekFirst()** **E peekLast()**
  враћа елемент са почетка или краја реда не уклањајући га када ред није празан. Ако је ред празан, прва два метода избацују NoSuchElementException, а последња два враћају null.

15.Interfejs RandomAccess

```
public interface RandomAccess
```

Marker interface used by `List` implementations to indicate that they support fast (generally constant time) random access. The primary purpose of this interface is to allow generic algorithms to alter their behavior to provide good performance when applied to either random or sequential access lists.

The best algorithms for manipulating random access lists (such as `ArrayList`) can produce quadratic behavior when applied to sequential access lists (such as `LinkedList`). Generic list algorithms are encouraged to check whether the given list is an `instanceof` this interface before applying an algorithm that would provide poor performance if it were applied to a sequential access list, and to alter their behavior if necessary to guarantee acceptable performance.

It is recognized that the distinction between random and sequential access is often fuzzy. For example, some `List` implementations provide asymptotically linear access times if they get huge, but constant access times in practice. Such a `List` implementation should generally implement this interface. As a rule of thumb, a `List` implementation should implement this interface if, for typical instances of the class, this loop:

```
for (int i=0, n=list.size(); i < n; i++)
    list.get(i);
```

runs faster than this loop:

```
for (Iterator i=list.iterator(); i.hasNext(); )
    i.next();
```

16.Povezana lista

-kod povezane liste je vazna pozicija objekta, u praksi je cesto potrebno dodati elemente u sredinu liste, koriscenje iteratora za dodavanje elemenata ima smisla jedino kod kolekcije sa prirodnim poretkom, prema tome nema metoda add u interfejsu Iterator, ali postoji podinterfejs ListIterator, koji sadrzi metodu add:

```
interface ListIterator<E> extends Iterator<E>{
        void add(E element);
         ...
}
```

-add dodaje novi element ispred pozicije iteratora

-neposredno nakon poziva next, metod remove uklanja element levo od iteratora, medjutim ako je pozvan previous, uklanja se element desno, nije moguce pozvati remove dva puta zaredom

-metod set zamenjuje poslednji element vracen pozivom metoda next ili previous

-za razliku od metoda add interfejsa Collection, ovaj metod ne vraca boolean – pretpostavlja se da operacija add uvek menja listu

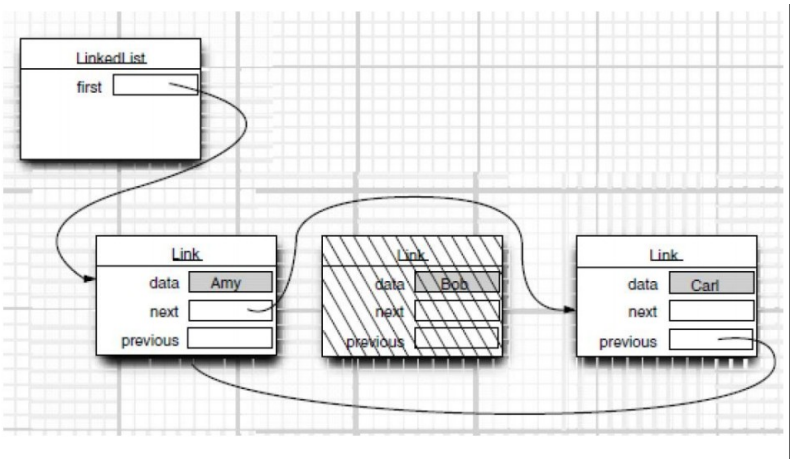
Nadalje ListIterator poseduje dva metoda koja se mogu koristiti za obilazenje liste unazad:

-E previous()

-boolean hasPrevious()

Метод listIterator класе LinkedList враћа итератор, објекат класе која имплементира интерфејс ListIterator.

```
ListIterator<String> iter = osoblje.listIterator();
```



- Многи корисни методи за оперисање повезаним листама декларисани су у интерфејсу Collection.
- Они су, већим делом, имплементирани у суперкласи AbstractCollection класе LinkedList.
- Повезана листа не подржава брз случајан приступ.
- Ако желимо да видимо n-ти елемент, морамо да пођемо од почетка и најпре пређемо преко првих n-1 елемената. Не постоји пречица.

- Дакле, програмери обично не користе повезане листе у ситуацијама у којима је потребно приступати елементима коришћењем целобројног индекса.
- Без обзира на то, класа LinkedList поседује метод get који омогућује приступ одређеном елементу:

```
LinkedList<String> lista = … ;
String obj = lista.get(n);
```

java.util.LinkedList<E>

- **LinkedList()**
  прави празну повезану листу
- **LinkedList(Collection<?extendsE> elements)**
  прави листу и додаје све елементе из колекције
- **void addFirst(E element)**
  додаје елемент на почетак листе
- **void addLast(E element)**
  додаје елемент на крај листе
- **E getFirst()**
  враће елемент са почетка листе
- **E getLast()**
  враће елемент са краја листе
- **E removeFirst()**
  брише и враће елемент са поцетка листе
- **E removeLast()**
  брише и враћа елемент са краја листе

17.Nizovna lista, ArrayList, primeri

-Interfejs liste je uredjena kolekcija u kojoj je pozicija elemenata bitna

-Postoje dva pravila za pristup elementima: preko iteratora, kao i metoda get i set za direktan pristup(drugi metod ima smisla za nizove)

-ova klasa takodje implementira interfejs List

- Методи size, isEmpty, get, set, iterator, и listIterator се извршавају за константно време.
- Додавање елемента дуже траје него што је то случај са LinkedList имплементацијом, али је зато бржи приступ елементу преко његовог индекса.
- Сваки примерак класе ArrayList има свој капацитет.
- Капацитет је величина низа који служи за смештај елемената листе.
- Он аутоматски расте током додавања елемената у ArrayList објекат.

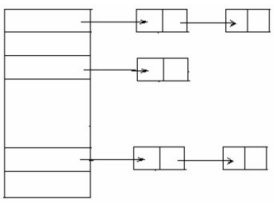
- Класе ArrayList и Vector имплементирају интерфејс RandomAccess.
- Јава програмери који су ветерани користе класу Vector кад год им је потребан динамички низ.
- Зашто би требало користити ArrayList уместо Vector?
- Сви методи класе Vector су синхронизовани што обезбеђује конзистенттност података у вишенитном програмирању, али успорава извршење метода.
- Насупрот томе, методи класе ArrayList нису синхронизовани, па их треба користиткад год није потребна синхронизација.

18.HashSet, primeri

- Повезане листе и низови дозвољавају убацивање елемената произвољно и притом воде рачуна о њиховом редоследу.
- Међутим, ако треба наћи елеменат а коме није запамћена позиција, потребно је претражити (у најгорем случају) све елементе колекције.
- Уколико је редослед елемената небитан, може се користити колекција која обезбеђује много бржи приступ елементима.
- Једино је незгодно што се тада не зна ништа о редоследу елемената, већ их та структура организује по сопственом редоследу.

- Добро позната структура за брзо проналажење елемената је хеш табела.
- За разлику од низова код којих су индекси цели бројеви, овде је позиција у структури одређена произвољним објектом.
- Ипак, имплицитно се захтева да сваком објекту буде придружен цели број тако што се рачуна његов тзв. хеш-код.
- Да би се конзистетно дефинисао хеш-код за неки објекат, програмер мора редефинисати две методе: hashCode и equals.
- Притом мора бити испуњено да ако важи a.equals(b),онда a и b морају имати исти хеш-код.

- Отворена хеш табела је обично реализована као низ повезаних листи.
- За проналажење места објекта у хеш табели, израчунава се хеш-код и дели се по модулу са димензијом низа.
- Резултат је индекс члана у низу тј. индекс повезане листе која садржи дати елеменат.



- Да би се одредила позиција објекта у табели, израчунава се његов хеш-кôд и нађе остатак при дељењу са укупним бројем листи.
- Тако добијени број је индекс листе који садржи дати елемент.
- Неизбежно је да се понекад деси да има више елемената којима одговара исти индекс листе и тада долази до тзв. колизије.
- У том случају, нови објекат се пореди са свим објектима из листе како би се видело да ли је већ присутан.
- Ако хеш-кôдови имају разумну случајну дистрибуцију и број листи довољно велик, требало би да буде потребно свега неколико поређења.

- Ако се превише објеката убаци у хеш-табелу, број колизија расте, а перформансе опадају.
- Обично се број листи се поставља на нешто између 75% и 150% очекиваног броја елемената.
- Стандардна библиотека за број листи користи степене двојке, подразумевано 16. и свака вредност која се зада за број листи аутоматски бива заокружена на следећи степен двојке.
- Ако се хеш-табела препуни, неопходно је да буде рехеширана.
- Да би се табела рехеширала, неопходно је да се креира табела са већим бројем листи, а сви елементи убаце у нову табелу.

- Хеш табела се користи за имплементацију неколико важних структура података.
- Најједноставнија међу њима је скуп. То је колекција елемената без понављања.
- ЈДК садржи класу HashSet која имплементира скуп базиран на хеш табели.
- HashSet се користи једино кад редослед елемената у колекцији није битан.
- Основне методе су:
  1. додавање елемента у скуп,
  2. провера да ли је елемент у скупу,
  3. избацивање из скупа,
  4. итерирање скупа у привидно произвољном редоследу.

java.util.HashSet<E>

- **HashSet()**
  креира празан каталог.
- **HashSet(Collection<? extends E> elements)**
  креира каталог и у њега додаје све елементе из колекције.
- **HashSet(int initialCapacity)**
  креира празан каталог одређеног капацитета.
- **HashSet(int initialCapacity,float loadFactor)**
  креира празан каталог одређеног капацитета и датог фактора пуњења (број између 0.0 1.0 који одредјује проценат при поновном хеширању).

java.lang.Object

- **int hashCode()**
  враћа хеш-код за објекат this. Хеш-код може бити цео, позитиван и негативан. Дефиниција equals и hashCode морају бити сагласни: ако x.equals(y) враћа true, онда x.hashCode() мора бити исти као y.hashCode().

19.Drvo-skup, TreeSet, Primeri

- Дрвоидни скупови су слични хеш скуповима, али уз разлику што је дрвоидни скуп је сортирана колекција.
- Тто значи да се могу додавати елементе у колекцију произвољним редоследом а да се при пролазу кроз колекцију елементи аутоматски обилазе у сортираном поретку.
- **Пример.** Претпоставимо да се додају нике у дрвоидни скуп и потом се приказују елементи који су додати у колекцију.

```
SortedSet<String> sorter = new TreeSet<String>();
sorter.add("Bob");
sorter.add("Amy");
sorter.add("Carl");
for (String s : sorter)
        System.println(s);
```

- Као што се може очекивати, приказани елементи су сортирани:

Amy Bob Carl.

- Као што име класе говори, сортирање се извршава по принципу дрволике структуре података.
- Сваки пут када се елемент дода у дрво, он се поставља на одговарајућу позицију дрвета.
- Стога, итератори увек посећују елементе у сортираном поретку.
- Подразумевано, се претпоставља да се убацују елементи класе која имплементира интерфејс Comparable!
- Ако убацујемо сопствене објекте, тада морамо сами дефинисати поредак имплементирањем интерфејса Comparable.
- Алтернативно, конструктору класе TreeSet може се проследити објекат која имплементира интерфејс Comparator.

- Додавање елемената дрвету је спорије од додавања хеш табели, али је много брже од постављања елемената на право место у низу или повезаној листи.
- Ако дрво садржи n елемената, тада је у просеку потребно $\log_2 n$ поређења да би се нашла права позиција за нови елемент.
- Ако дрво садржи 1000 елемената, додавање новог захтева око 10 поређења - много више него код додавања елемента у хеш скуп.

java.lang.Comparable<T>

- **int compareTo(T other)**
  пореди текући и објекат other и враћа негативну вредност ако текући објекат долази испред other, 0 ако су идентични у сортираном поретку, а позитивну вредност иначе.

java.util.Comparator<T>

- **int compare(T a, T b)**
  пореди два објекта и враћа негативну вредноста ако a долази испред b, 0 ако су идентични у сортираном поретку, а позитивну вредност иначе.

java.util.SortedSet<E>

- **Comparator<? super E> comparator()**
  враћа компаратор који се користи за сортирање елемената или null ако се елементи пореде методом compareTo() интерфејса Comparable.
- **E first()**
- **E last()**
  враћа најмањи или највећи елемент сортираног скупа.

java.util.NavigableSet<E>

- **E higher(E value)**
- **E lower(E value)**
  враћа најмањи елемент >value или највећи елемент <value или null ако такав елемент не постоји.
- **E ceiling(E value)**
- **E floor(E value)**
  враћа најмањи елемент >=value или највећи елемент <=value или null ако такав елемент не постоји.
- **E pollFirst()**
- **E pollLast()**
  уклања и враћа најмањи или највећи елемент скупа или null ако је скуп празан.
- **Iterator<E> descendingIterator()**
- враћа итератор који обилази скуп у опадајућем смеру.

java.util.TreeSet<E>

- **TreeSet()**
  Конструише дрво за чување Comparable објеката.
- **TreeSet(Comparator<? Super E> c)**
  Конструише дрво и користи дати компаратор за сортирање елемената.
- **TreeSet(SortedSet<? extends E> elements)**
  Конструише дрво, додаје све елементе из сортиране колекције, и користи исти компаратор као и сортирани скуп прослеђен као аргумент позива метода.

20.Nizovni red sa dva kraja, ArrayDeque

java.util.ArrayDeque<E>

- **ArrayDeque()**
- **ArrayDeque(int initialCapacity)**
  конструише неограничени deque иницијалног капацитета 16 или задатог иницијалног капацитета.

21.Metodi klase Collections

- Скупови су колекције уз помоћу којих се брзо проналазе постојећи елементи.
- Чешћа ситуација у којо се поседује нека кључна информација, и треба да се на основу ње пронађете дотични елемент.
- Структуре података у облику **каталога** служе за ту сврху.
- Каталози чувају парове кључ/вредност и код њих се лако може пронаћи вредност ако се наведе кључ.
- На пример, може се чувати табела службеника, где су кључеви ниске - службенички идентификатори, а вредности објекти типа Employee.

- Јава библиотека подржава две главне имплементације за каталоге: HashMap и TreeMap.
- Обе класе имплементирају Map интерфејс.
- Хеш каталог не сортира кључеве, за разлику од дрвоидног каталога који користи поредак кључева.
- Да ли користити хеш каталог или дрвоидни каталог?
  Као и са скуповима, хеширање је нешто брже, и то је бољи избор уколико кључеви не морају бити сортирани.

- **Пример.**

```
Map<String,Employee> staff = new HashMap<String,Employee>();
Employee harry = new Employe("Harry Hacker");
staff.put("987-98-9996",harry);
```

- Кад год се додаје објекат у каталог, мора се добро дефинисати кључ.
- У претходном примеру, кључ је ниска, а одговарајућа вредност је објекат типа Employee.
- **Пример.** Добијање објекта који се налази у каталогу на основу кључа се реализује на следећи начин:

```
String s ="987-98-9996";
e= staff.get(s);
```

- Ако нема сачуване информације у каталогу за дати кључ, тада метод get враћа null .
- Кључеви морају бити јединствени. Не могу се сачувати две вредности са истим кључем.
- Ако се позове set метод два пута за исти кључ, тада друга вредност замењујепрву.
- Метод remove брише елемент са датим кључем из каталога.
- Метод size враћа број елемената у каталогу.

- Каталог није самостална колекција у ЈДК-у, тј. састоји се из више погледа (подструктура).
- Постоје три погледа: скуп кључева, колекција (није скуп) вредности, те скуп парова кључ/вредност.
- Кључеви и парови кључ/вредност образују скуп због тога јер може бити само једна копија кључа у колекцији.
- Ова три претходно побројана погледа обезбеђују методе:

```
Set<K> keySet()
Collection<K> values()
Set<Map.Entry<K,V>> entrySet()
```

22.Interfejs Map

java.util.Map<K, V>

- **V get(K key)**
  враће вредност кључа, тј. враће објекат на који показује кључ, или null ако се кључ не налази у каталогу. Кључ може бити null.
- **V put(K key, V value)**
  убацује у каталог пар кључ/вредност. Ако кључ већ постоји, тада нови објекат замењује стари, претходни на који је показивао кључ. Овај метод враће стару вредност за дати кључ, или null ако кључ претходно није био дефинисан. Кључ може бити null, али вредност не сме бити null.
- **void putAll(Map<? extends K, ? extends V> entries)**
  убацује све елементе из спецификованог каталога entries у овај каталог.
- **boolean containsKey(Object key)**
  враће true ако је кључ већ у каталогу.
- **boolean containsValue(Object value)**
  враће true ако је вредност већ у каталогу.
- **Set<Map.Entry<K, V>> entrySet()**
  враће скуп погледа објеката типа Map.Entry, парове кључ/вредност из каталога. Могу се избрисати елементи овог скупа и они ће бити избрисани из каталога, али се не могу додавати елементи.

java.util.Map<K, V>

- **Set<K> keySet()**
  враће скуп свих кључева у каталогу. Могу се избрисати елементи овог скупа и кључеви и вредности на њима биће избрисани из каталога, али се не могу додавати елементи.
- **Collection<V> values()**
  враће вредности свих вредности у каталогу. Могу се избрисати елементи овог скупа и кључеви и вредности на њима биће избрисани из каталога, али се не могу додавати елементи.

java.util.Map.Entry<K, V>

- **K getKey()**
- **V getValue()**
  враћа кључ или вредност за ову величину.
- **V setValue(V newValue)**
  поставља вредност у елементу каталога и враћа стару вредност.

23.HashMap

java.util.HashMap<K, V>

- **HashMap()**
- **HashMap(int initialCapacity)**
- **HashMap(int initialCapacity, float loadFactor)**
  конструише празну хеш мапу са наведеним капацитетом, и фактором испуњености (број између 0.0 и 1.0 који одређује када ће хеш табела бити поново хеширана у већу). Подразумевани фактор испуњености је 0.75.

24.TreeMap

java.util.TreeMap<K, V>

- **TreeMap(Comparator<? super K> c)**
  креира дрвоидни каталог и користи наведени компаратор за сортирање кључева.
- **TreeMap(Map<? extends K, ? extends V> entries)**
  конструише дрвоидни каталог и додаје све entries из каталога-аргумента.
- **TreeMap(SortedMap<? extends K, ? extends V> entries)**
  конструише дрвоидни каталог, додаје све entries из наведеног каталога и користи наведени компаратор за сортирање кључева.

25.Generici i kolekcije + 26.Dzoker tip kod kolekcija + 27.Ogranicenja nad dzoker tipom kod kolekcija

Колекције и генерици

```
void printCollection(Collection c) {
    Iterator i = c.iterator();
    while (i.hasNext()) {
        System.out.println(i.next());
    }
}
```

Треба направити генерички метод који приказује све елементе колекције.

```
void printCollection(Collection<Object> c) {
    for (Object e: c){
        System.out.println(e);
    }
}
```

Ово је први, наивни покушај...

```
printCollection(stones);
```

Не може се превести!

- Проблем је решен увођењем генеричког типа који може бити било шта, а који се назива џокер тип и означава симболом ?

```
Collection<?>
```

"колекција непознатих" је колекција чији елементи могу одговарати било ком типу - **џокер тип**

```
void printCollection(Collection<?> c) {
   for (Object e: c){
      System.out.println(e);
   }
}
```

```
printCollection(stones);
```

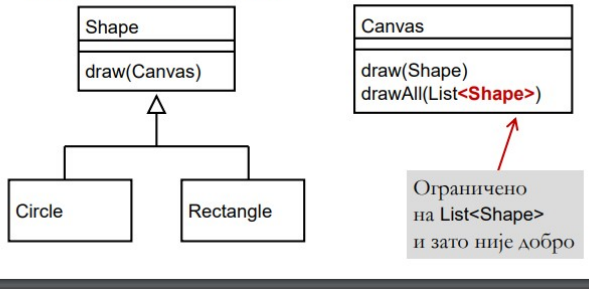
```
stone(java.awt.Color[r=255,g=0,b=0])
stone(java.awt.Color[r=0,g=255,b=0])
stone(java.awt.Color[r=0,g=255,b=0])
```

- Проблем са коришћењем колекције и џокер типа наступа зато што се не зна тип компоненте, па преводилац не може да преведе наредбу за доделу у колекцију елемената непознатог типа.

```
String myString;
Object myObject;
List<?> c = new ArrayList<String>();

// c.add("hello world");                    // compile error
// c.add(new Object());                     // compile error
((List<String>) c).add("hello world");
((List<Object>) c).add(new Object());       // no compile error!

// String myString = c.get(0);              // compile error
myString = (String) c.get(0);
myObject = c.get(0);
myString = (String) c.get(1);               // run-time error!
```


Колекције и генерици (4)

- Понекад је потребно направити ограничење над џокер-типом.
- **Пример.** Треба направити апликацију која исцртава различите облике (кругове, провоугаонике итд.). Класе у апликацији би биле:



Ограничено на List<Shape> и зато није добро

- **Пример.** Потребно је креирати метод који као аргумент прихвата колекцију са ма којом врстом облика, тј. са ма којом подкласом класе Shape.
- Истовремено, треба инструисати преводилац да спречи да елементи колекције буду примерци класе која није подкласа Shape.
- То се постиже на следећи начин:

```
public void drawAll(List<? extends Shape>)
{
    …
}
```

ограничени џокер

У овом случају, класа Shape је **горње ограничење** за џокер

```
import java.util.*;
…

    public void pushAll(Collection<? extends E> collection) {
        for (E element : collection) {
            this.push(element);
        }
    }

    public List<E> sort(Comparator<? super E> comp) {
        List<E> list = this.asList();
        Collections.sort(list, comp);
        return list;
    }
```

Сви елементи морају бити **бар** типа E (тј. типа подкласе класе E)

Метод за поређење мора захтевати да се извршава над типовима **највише** E (тј. над типовима неке од наткласа класе E)

- Генерички колекцијски интерфејси имају огромну предност - потребно је имплементирати своје алгоритме само једанпут.
- **Пример.** Размотримо једноставан алгоритам за израчунавање максималног елемента колекције. За највећи елемент низа:

```
if (a.length == 0) throw new NoSuchElementException();
T largest = a[0];
for (int i = 1; i < a.length; i++)
        if(largest.compareTo(a[i]) < 0)
                largest = a[i];
```

- Налажење максимума низовне листе је незнатно другачије:

```
if(v.size() == 0) throw new NoSuchElementException();
T largest = v.get(0);
for (int i = 1; i < v.size(); i++)
        if(largest.compareTo(v.get(i)) < 0)
                largest = v.get(i);
```

- **Пример (наставак).** У повезаној листи немамо ефикасан случајан приступ, али можемо користити итератор:

```
if (l.isEmpty()) throw new NoSuchElementException();
Iterator<T> iter = l.iterator();
T largest = iter.next();
while(iter.hasNext()) {
        T next = iter.next();
                if(largest.compareTo(next) < 0)
                        largest = next;
}
```

- Ове петље су напорне за писање и омогућују грешке. Пожељно је да се избегне понављање тестирања и имплементација мноштва метода попут:

```
static <T extends Comparable> T max(T[] a)
static <T extends Comparable> T max(ArrayList<T> v)
static <T extends Comparabla> T max(LinkedList<T> l)
```

- **Пример (наставак).** Ту на сцену ступају колекцијски интерфејси.
- Треба осмислити минимални колекцијски интерфејс који је потребан за ефикасан алгоритам.
- Израчунавање максимума може се урадити просто итерирањем кроз елементе. Имплементација метода max тако да прихвата објекат ма које класе која имплементира интерфејс Collection:

```
public static <T extends Comparable> T max(Collection<T> c){
        if (c.isEmpty()) throw new NoSuchElementException();
        Iterator<T> iter = c.iterator();
        T largest = iter.next();
        while (iter.hasNext()) {
                T next = iter.next();
                if (largest.compareTo(next) < 0)
                        largest = next;
        }
        return largest;
}
```

28.Genericke metode sa ogranicenjem dzoker tipa u JDK

- Генерици су моћан концепт, који се користи у сортирању, бинарној претрази и још неким корисним алгоритмима.
- Како метод sort сортира листу?
  Алгоритми за сортирање у књигама су презентовани за низове и користе директан приступ елементима.
- Међутим, листа је колекција и нема случајан приступ?
- Имплементација у Јави просто прекопира све елементе у низ, сортира га користећи варијанту merge sort алгоритма, а затим копира сортирану секвенцу натраг у листу.
- Merge sort алгоритам коришћен у библиотеци је за нијансу спорији од quick sort-а, али има једну предност: стабилан је, тј. не врши размену једнаких елемената.
- Зашто нам је битан поредак једнаких елемената?

- **Пример.** Следи уобичајени сценарио. Претпоставимо да имамо листу запослених која је већ сортирана по имену.
- Сада сортирамо по зради. Шта се дешава са запосленима који имају исту зараду?
- Када је сортирање стабилно, а алгоритам сортирања је merge sort, поредак по имену бива очуван. Другим речима, излаз је листа сортирана најпре по зради, а потом по имену.
- Класа Collections поседује метод shuffle који случајно пермутује редослед елемената у листи.

Пример. Мешање карата

```
ArrayList<Karta> karte = …;
Collections.shuffle(karte);
```

- Метод binarySearch класе Collections имплементира бинарну претрагу.
- Наравно, колекција мора претходно бити сортирана или ће алгоритам вратити погрешан резултат.

```
i = Collections.binarySearch(c, element);
i = Collections.binarySearch(c, element, comparator);
```

- Повратна вредност ≥0 означава индекс пронађеног елемента у складу са коришћеним поређењем. Ако је повратна вредност негативна, не постоји тражени елемент у колекцији.

- Међутим, повратна вредност се може користити за израчунавање позиције на коју треба уметнути element у колекцију како би она остала сортирана.
- Та позиција је: insertionPoint = -i - 1;
- Није просто -i јер онда би вредност 0 била двосмислена. Другим речима, операција:

```
if ( i < 0)
        c.add(-i - 1, element);
```

додаје елемент на исправну позицију.

- Како би имала смисла, бинарна претрага захтева случајан приступ тако да метод binarySearch проверава да ли задата листа имплементира интерфејс RandomAccess!
- Ако да, онда ради бинарну, а иначе линеарну претрагу.

java.util.Collections

•static <T extends Comparable<? super T>> void sort(List<T> elements)

•static <T> void sort(List<T> elements, Comparator<? super T> c)

сортира елементе листе користећи стабилни алгоритам. Временска сложеност алгоритма је O(n log n), где је n дужина листе.

•static void shuffle(List<?> elements)

•static void shuffle(List<?> elements, Random r)

случајно меша елементе листе. Временска сложеност алгоритма је O(n a(n)), где је n дужина листе, док је a(n) просечно време приступа елементу.

•static <T> Comparator<T> reverseOrder()

враћа компаратор који сортира елементе у обрнутом поретку од поретка одређеног методом compareTo() интерфејса Comparable.

•static <T> Compator<T> reverseOrder(Comparator<T> comp)

враћа компаратор који сортира елементе у обрнутом поретку од поретка одређеног компаратором comp.

java.util.Collections

•static <T> void copy(List<? super T> to, List<T> from)

копира све елементе из изворне листе на исте позиције у одредишној листи. Одредишна листа мора бити дуга бар колико изворишна.

•static <T> void fill(List<? super T> l, T value)

попуњава све позиције у листи истом вредношћу.

•static <T> boolean addAll(Collection<? super T> c, T… values)

додаје све вредности у дату колекцију и враћа true ако је тиме колекција промењена.

•static <T> boolean replaceAll(List<T> l, T oldValue, T newValue)

замењује све елементе једнаке старој вредности новом вредношћу.

•static int indexOfSubList(List<?> l, List<?> s)

•static int lastIndexOfSubList(List<?> l, List<?> s)

враћа индекс прве или последње подлисте од l једнаке са s или -1 ако нема такве подлисте у l.

•static void swap(List<?> l, int i, int j)

размењује елементе на датим позицијама.

29.Apstraktne kolekcijske klase i kreiranje novih kolekcija ???