Objektno Programiranje

Predavanja

Uvod u Metodologiju

Kroz istoriju programerske struke pojavljivale su se tri metodologije:

1. **Kompozitna metodologija** – nastaje sa pojavom FORTRAN-a i trajala je do 60-ih godina prošlog veka.
2. **Strukturirana metodologija** – je metodologija koja nastaje posle kompozitne i zasnovana je na FORTRAN-u i COBOL-u i traje do kraja 70-ih godina dvadesetog veka.
3. **Objektna metodologija** – koja nastaje krajem 70-ih i koja je i danas dominantna metodologija.

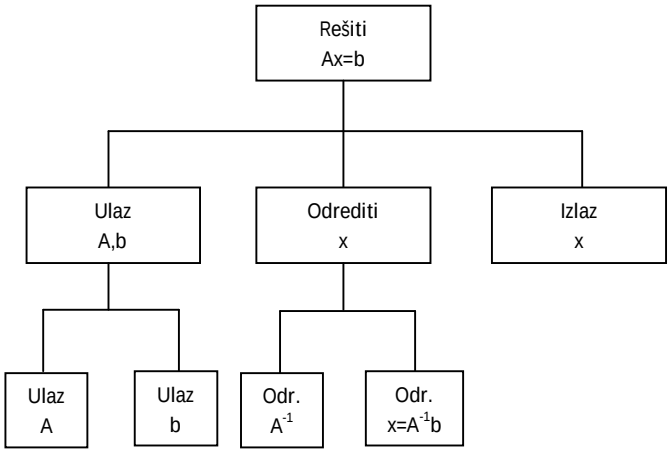
Metodologije su se međusobno smenjivale i menjale zbog porasta kompleksnosti programa, prvenstveno zbog naglog razvoja hardvera koji je pružao svoj maksimum tek kada je bio praćen odgovarajućim softverom. Kompleksnost programa nije tako lako rešiti. U rešavanju nekih problema programeri su došli do neke određene granice, programerskog „plafona“, kada ni njihovo umeće ni programski jezik nisu mogli rešiti probleme i tada se pribeglo promeni metodologije. Npr. strukturirano programiranje se nije moglo izboriti sa problemom korisničkog interfejsa i tada je objektna metodologija i objektno programiranje postalo dominantno.

Strukturirano i kompozitno programiranje su srodnije međusobno nego sa objektnim programiranjem, pa se zajedničkim imenom zovu **procedurno programiranje**.

Dekompozicija, razlike između metodologija

Jedino sredstvo sa kojim se borimo sa kompleksnošću programa jeste **dekompozicija**. Ove tri metodologije se suštinski razlikuju po vrsti dekompozicije.

Kompozitno programiranje je izvedeno tako da se dekompozicija vrši na bazi potprograma. Npr. ako imamo problem množenja matrice vektorom Ax=b, kompozitni programer odmah razmišlja šta je u ovom problemu dovoljno komplikovano da odradi preko potprograma a šta je dovoljno jednostavno da ide u glavni program. U rešavanju ovog problema se pojavljuje inverzna matrica što je očigledno dovoljno komplikovano da se inverzija matrice nađe kao potprogram, dok je drugi potprogram u stvari unos matrice jer je u to vreme on bio komplikovan zbog ograničene veličine vrste na bušenim karticama koje sutada bile u upotrebi. Ostatak je bio program. Redosled potprograma nije bio bitan ali uvek se prvo pristupalo najtežem potprogramu zbog testiranja mašine, jer ako je taj potprogram mogao da radi onda je mašina bila dovoljno jaka da pokrene ceo program.



Kompozitno programiranje je nasleđeno od strane strukturiranog programiranja koje postaje dominantno. Kod njega je uvedeno vreme kao komponenta. Npr na istom primeru množenja matrice vektorom, programer traži rešenje i polazi upravo od rešenja. On se pita šta program treba prvo da radi? Npr treba prvo da se obavi ulaz, pa traženje vektora x, pa izlaz. Ovo je sekvencijalna dekompozicija i potpuno se održava u vremenu. Ona je sekvencijalna jer se jedan deo problema rešava tek kada je prethodni u potpunosti rešen.

Problemi strukturirane metodologije

Strukturirano programiranje je imalo takoreći „ugrađenu naprslinu“. Ovakav način dekompozicije je u stvari dekompozicija alogoritma, ali se onda postavlja pitanje šta je sa strukturama podataka? Ova metodologija to ne rešava. To je u stvari osnovni problem ove metodologije, to što se bavi samo dekompozicijom alogoritama. Iz toga proističe poblem **kompatibilnosti**. To je problem koji nastaje kada neki tim radi na rešavanju problema i jedan programer pravi ulazni potprogram (npr. unos matrice i mora da je sažme da bi stala u memoriju), a neki drugi programer koji radi sa drugim potprogramom ne koristi istu strukturu podataka (npr. inverziju kompletne a ne sažete matrice) i tu nastaje problem. Ovaj problem se vidi tek kod velikih projekata kada i mali problem može da ima katastrofalne posledice. Tj u opštem slučaju nema garancije da će se za različite potprograme koristiti iste strukture podataka.

Drugi problem jeste problem **kontinuiteta (proširivosti)**. Svaki softver se vremenom proširuje i nadograđuje da bi opstao na tržištu. Strukturirano programiranje ne garantuje da će softver biti proširiv tj. ne rešava ga uopšte jer je vrlo teško raditi izmene kod alogoritamsko orijentisanih programa, za razliku od unosa izmena u strukturu podataka.

Treći problem je problem **mogućnosti višestruke upotrebe**. On nastaje kada hoćemo neke potprograme da koristimo uvek kada nam zatrebaju tj. hoćemo da oni budu univerzalni. Načini na koji su organizovane biblioteke potprograma je postao ograničenje u radu sa potprogramima. Zahtev za višekratnom upotrebom je zahtevao i određene izmene i proširenja potprograma kada hoćemo da ih koristimo. Ovo dvoje se jednim imenom naziva **modularnost**. Ona se pojavljuje na svakom koraku programiranja. Naročito se to primećuje kod korisničkog interfejsa jer sva dugmad su 99% ista po izgledu koda, tj. suštinski se razlikuje samo njihova funkcija. Zato mi samo kod dugmadi u stvari menjamo njihovu funkciju jer bi njihovo pravljenje piksel po piksel bilo veoma teško.

Objektno programiranje

Zbog predhodnih problema moralo je doći do izmena u metodologiji i to u obliku objektne metodologije i objektnog programiranja. Ono je mnogo savršenije od predhodne dve vrste i bazira se na logici ljudskog uma. Neki smatraju da je ono počelo 1967. sa člankom Dahla u knjizi „Strukturirano programiranje“. Stvarni početak je odložen do kraja 70-ih godina kada ne njegovo uvođenje bilo neminovno.

Objektni programer počinje sa softverskim modelovanjem. Prvo se ustanovi domen problema kome pripada naš problem i pravi se njegov model. Problem sata koji uvek radi je karakteristika objektnog programiranja i samo ono nudi rešenje tog problema. O programer uočava **entitete** i povezuje ih u **klase entiteta** i tek onda ih softverski modeluje. U problemu Ax=b uočavamo tri entiteta: A,b,x. Sada se modeluju ove klase entiteta (primer je dat na ad-hok jeziku):

```
Naziv: Matrica
Podaci: m,n,P
Operacije:
    Učitati
    Invertovati
    Prikazati
    Transponovati
    Pomnožiti SL ( vektorom )
    Pomnožiti SD ( vektorom )
```

```
Naziv: Vektor
Podaci: k,V
Operacije:
    Učitati
    Prikazati
    Moduo
    .
    .
    .
```

```
Matrica A;
Vektor x,b;
A.Učitati
b.Učitati
A.Invertovati
X=A.Pomnožiti SD ( b )
x.Prikazati
```

Ključni deo objektnog programiranja je da u podacima ne stoje samo osobine entiteta već i operacije nad tim entitetom. Kada se završi rad sa jednom klasom počinje rad na drugoj klasi i tek kada ovo sve završi programer se vraća na početni problem. Ova kratkoća glavnog programa (treća kolona u primeru) je u stvari karakteristična za objektno programiranje. Znači ovde osim podataka uviđamo i operacije. Objektno programiranje je podešeno tako da više nikada ove klase nećemo pisati već ih možemo uvek koristiti.

Nasleđivanje je mehanizam C++ kojim omogućava dopunjavanje i prilagođavanje klasa našim potrebama tj. možemo klase da prilagodimo našim potrebama bez da remetimo izvorni kod te klase. Kod objektnog programiranja objekti međusobno vrše interakciju. Deo **klase** naziva se **objekat** ili instanca klase. U predhodnom primeru objekti su A i x,b. Klasa i objekat stoje u istom odnosu kao i tip i promenljiva.

Temelji objektnog programiranja

Objektna metodologija kao i svaka druga metodologija stoji na određenim principima tj. ima određene osnove. Ona se zasniva na nekoliko osnovnih principa koji su bili poznati čak i pre objektnog programiranja u rudimentalnom obliku, samo što tada nisu igrali bitnu ulogu u izradi softvera. Osnovni elementi objektnog programiranja su: apstrakcija i skrivanje informacija, inkapsulacija i modularnost, polimorfizam, veze između klasa a posebno nasleđivanje.

Apstrakcija ili **apstrahovanje** je veoma opšti postupak, na kome je zasnovana naša logika, poznat i hiljadama godina pre nastanka objektne metodologije i predstavlja izdvajanje odnosno uočavanje bitnog i zanemarivanje nebitnog. Postoji i u strukturnom programiranju gde svaki obučen programer vidi potprograme kao apstraktne elemente.

Skrivanje informacija je odavno jedan od ključnih principa objektnog programiranja. Razlog skrivanja informacija je smanjivanje kompleksnosti koju programer mora da razume. Takođe ovaj princip podrazumeva da klijentu ne moraju biti poznati detalji realizacije programa i time smanjuje kolicinu informacija koju programer-klijent mora da drzi u glavi.

Inkapsulacija je tehnika kojom se pri realizaciji objedinjavaju deskriptivne i dinamičke osobine modela entiteta (u našem slučaju objekata) poštujući pri tome princip skrivanja informacija.

Modularnost je tehnika razlaganja složenog softverskog sistema u domenu implementacije na jednostavnije elemente koje imaju unutrašnju logku. Ona je bila poznata i kod procedurnog programiranja kao skup servisa namenjenih korisnicima (biblioteke). Sama modularnost predstavlja mogućnost proširivosti i višekratne upotrebe i to je ključna stvar u objektnom programiranju.

Polimorfizam (*poli* – više, *morfe* – oblik) je osobina, ili pak mogućnost, da se softverska komponenta ponaša zavisno od konteksta ili okolnosti. Tako se polimorfno mogu ponašati objekti, promenljive pa i funkcije. Tako se na primer ponaša C-ova znakovna promenljiva kada se pojavljuje u kontekstu 'a'+1 i tada se ponaša kao celobrojna promenljiva. Za ostvarenje polimorfizma uvek mora postojati određen preduslov koji je u prethodnom slučaju ugrađen u C, tj. znakovna promenljiva je podtip celobrojnog tipa.

Veze između klasa su jedan od bitnih elemenata, jer vrlo retko klase postoje kao izolovane, već su veoma česte veze sa drugim klasama. Vezama se ostvaruje uređenje apstrakcija. Najveću važnost ima **nasleđivanje** tj. izvođenje klase iz klase. Nasleđivanje predstavlja prenošenje osobina jedne klase na drugu uz mogućnost dopune i modifikacije bez izmene izvornog koda prve klase. Ono omogućuje npr. da osnovnoj funkciji tastera, koja je ista za sve, dodamo i novu, nama potrebnu, funkciju bez menjanja njegovog koda.

Objekat i klasa

Postoji mnogo definicija i klasa i objekata ali jedno što im je zajedničko jeste da kažu da su klasa i objekat u stvari pojmovi. **Pojam** je po svojoj prirodi misao o bitnim karakteristikama predmeta kojih mi u realnosti prepoznajemo. To je ono što mi u stvari modelujemo. Najbitniji pojmovi u programiranju jesu pojam za stvar, proces, osobinu i na kraju pojam za odnos. Takođe veoma bitna stvar jesu i odnosi među pojmovima. Pojam može biti individualni ili opšti (klasni) pojam.

Logika se bavi opštim, bitnim osobinama problema, ali mi sa stanovišta programiranja ih možemo posmatrati konkretnije jer ih vezujemo samo za pitanje problema i ne idemo ka univerzalijama. Od problema do problema mi posmatramo druge karakteristike koje su nam u tom trenutku relevantne. Tako se svi autori slažu da su najbitnije osobine objekta: indentitet, stanje i ponašanje.

U objektnom sistemu svaki objekat je jedinstven. **Identitet** predstavlja osobinu objekta da se izdvaja od ostalih objekata i po kome je on prepoznatljiv. Ako uzmemo problem dve matrice videcemo koliko je bitan pojam identiteta. Imamo dve matrice:

$$\begin{matrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{matrix} \quad i \quad \begin{matrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{matrix}$$

Pitanje glasi: da li imamo dve ekvivalentne matrice ili je ovo samo prva matrica prepisana? Ukoliko je prepisana ako izmenimo element prve izenićemo i element druge a ukoliko nije onda druga matrica ostaje nepromenjena. Zbog ovakvih situacija bitno je da se objekti (u ovom slučaju matrice) označe nekim imenom tj da im se dodeli neki identitet.

I koncept stanja je sasvim generalizovan jer prevazilazi granice objektnog programiranja iako ono upravo stoji na pojmu stanja. **Stanje** je deo prošlosti i sadašnjosti objekta neophodan za određivanje njegovog budućeg ponašanja. To ponašanje sadrži i promenu stanja. **Ponašanje** je u stvari reakcija na pobudu i ono je određeno operacijama.

Iz predodnih objašnjenja izvodimo definiciju objekta. **Objekat** je softverski model individualnog pojma za stvar ili proces zasnovan na stanjima i koji sadrži osobine relevantne za domen problema. Definicija klase je ekvivalentna. **Klasa** je softverski model klasnog pojma za stvari ili procese zasnovane na stanjima i koja sadrži osobine relevantne za dati domen problema.

Objekat kao sastavni deo može da ima i druge objekte tj objekti sadrže podatke o sebi kao i podobjekte. Objekat zajedno sa svojim podobjektima i njihovim podobjektima, do proizvoljne dubine, čini strukturu objekata. Ona teži da bude tipa stabla. Svi objekti iste klase imaju istu strukturu i isto ponašanje. Klasa se generalno sastoji od tri stvari:

1. podaci u najužem smislu stvari i u C++ se nazivaju **podatak – član**;
2. objekti koji se u C++ nazivaju **objekat – član**;
3. operacije ili u C++ **funkcija – član**, koja nije nezavisna jer je unutar klase i može još da se zove i **metoda**.

Objekat – član i podatak – član se jednom rečju zovu **polje**. Od deskriptivnih osobina objekti imaju samo **atribute**. On otvara problem jer svako tvrdi da je nešto drugo. Tako je atribut nekada polje, nekada podatak – član, nekada i sama metoda. Ali bitno je napomenuti da atributi nisu objekti jer nemaju sopstveni identitet, semantički su nestabilni i dobijaju smisao samo u sklopu objekta u kome egzistiraju. Aktiviranje funkcije – članice zovemo slanje poruke, ili jednostavnije **poruka**. Za korisnika klase takođe uvodimo poseban pojam, jer kada kažemo korisnik automatski nas to asocira na čoveka, a klase koristi softver, pa za takvog korisnika kažemo **klijent**.

Principi objektnog programiranja

U vezi sa programskom realizacijom objekata, a prilikom kreiranja jezika smalltalk, Alan Kej je formulisao sledećih pet principa:

1. Sve je objekat. Drugim rečima u programu nema ničeg drugog osim objekata, nema potprograma nema funkcija pa čak ni konstanata koje se interpretiraju kao konstantni objekti. Kod C++ ovo nije tako jer je on hibridni jezik tj. nije čisti objektni program, i kod njega postoje slobodne funkcije. C++ štaviše omogućava istovremenu upotrebu „običnih“ konstanti i konstantnih objekata. Java nema ni tih slobodnih funkcija kao ni funkciju main. Java ima statičke metode koje su po karakteristikama iste kao slobodne funkcije samo što imaju drugačiji naziv.
2. Program je skup objekata koji zadaju poslove jedan drugom putem slanja poruka. Ovo predstavlja potpuni prelaz sa procedurnog programiranja. Objektni način razmišljanja je komunikacija objekata, kao npr komunikacija računara koji su zajedno na mreži.

3. Svaki objekat poseduje sopstvenu memoriju sastavljenu od drugih objekata (i drugih promenljivih u hibridnim jezicima). Objekti ne smeju deliti memoriju. Pokazivači npr. utiču na deljenje memorije od strane objekata.
4. Svaki objekat pripada nekoj klasi.
5. Svi objekti iste klase primaju iste poruke. Ovaj princip potiče iz definicionog zahteva da svi objekti iste klase imaju isto ponašanje.

Programski jezik C++

Programski jezik C++ se pojavio u periodu između 1980 i 1984 godine, kao objedinjeno proširenje, ili kako neki kažu pojačanje programskog jezika C. Bjarne Stroustrup je u stvari tvorac C++. Stvaranje ovog programskog jezika imalo je i retroaktivno dejstvo na C, tako da C i C++ faktički predstavljaju jedan jezik.

Definisanje klase u C++

Kada je u pitanju objektno programiranje najvažniji pojam oko koga se sve vrti jeste pojam klase i njihova izgradnja. C++ za stvaranje klase ima posebnu naredbu **`class`** koja je slična `struct` naredbi. Ovaj primer pokazuje uopšten prikaz izgleda definicije klase:

```
class ImeKlase {
    //podaci-članovi
    //objekti-članovi
    //funkcije-članovi
};
ImeKlase::imeFunkcije {
    //telo funkcije
}
```

Definicija se deli na dva dela, sastoji se od zaglavlja i tela modula tako da se deklaracija funkcija-članova nalazi u zaglavlju, a definicija u telu. Bitno je napomenuti da po završetku deklarisanja klase mora da stoji „;“ inače nam kompajler neće prijaviti grešku vezanu za tu liniju već spisak grešaka u linijama koje slede. Takođe postoji i jedno pravilo da se imena funkcija, klasa i objekata pišu takozvanom **kamel** notacijom tj da se ime svake nove reči u nazivu piše velikim slovom pazeći pri tome da je početno slovo za sva imena malo osim za ime klase koje počinje velikim početnim slovom. Ova notacija je u Javi obavezna dok je u C++ stvar navike. Definisanje klase pokazaćemo sada na primeru dekartove tačke. Pravimo klasu Point i tokom njenog definisanja mi tražimo način da je napravimo što univerzalnijom da bi je i posle nas programeri mogli koristiti ukoliko im zatreba.

Kao prvo osobine tačke jesu njene koordinate pa njih definišemo kao prodatak-član. Ali pored ovog deskriptivnog dela objektni programer definiše i operacije nad tom tačkom. Pri pisanju metoda mi imamo potpunu slobodu ali moramo da pazimo da repertoar bude dovoljan tj ni premali ni preveliki. Jer ako je metoda premalo naićićemo na problem da će nam neka operacija u sklopu rada sa tačkom zatreba a mi je nećemo imati tada na raspolaganju. Ovo shvatanje koliko metoda nam je potrebno dolazi sa iskustvom. Prvo zadajemo vrednost za x i y tj objekat dovodimo u neko početno stanje. Objekat je sintaksno vrlo sličan promenljivoj. Kada naredba u programu Point a,b počne da se izvršava zauzimaju se memorijske lokacije za objekte a i b sa po dve double promenljive. Metoda setPoint se vrši nad objektom i postavlja određene vrednosti za promenljive x i y. Sledeća metoda getX služi da se očita abscisa. Ona vraća kao rezultat x koordinatu. Analogno tome funkcioniše i funkcija getY. Sada sledi metoda distance koja je u stvari prototip funkcije koja je definisana van klase. Takođe dodela objekta objektu kao b=a je moguća između objekata iste klase.

```
class Point {
      double x,y;
      void setPoint(double xx,
                    double yy)
      {x=xx; y=yy;}
      double getX(){return x;}
      double getY(){return y;}
      double distance();
};

double Point::distance(){
      return sqrt(x*x+y*y);}


Point a,b;
double r,t;
...
a.setPoint(1,2);
t=2*a.getX();
b=a;
b.getY();
r=a.getY();
```

Metode koje su u potpunosti definisane unutad klase nazivaju se **umetnute** ili **inline** metode. One se bitno razlikuju od običnih funkcija po pozivu i to unutar prevodioca. Na mesto umetnutih funkcija prevodilac stavlja kompletan njen tekst. One se tretiraju isto kao i makro direktive u C-u, s tim da su one ovde izgurale makro direktive. Unutar umetnutih funkcija mogu da stoje samo naredbe dodele a ne neke komplikovane naredbe poput petlji.

Detalji realizacije programa treba da budu skriveni a to znači da delovi klase moraju da budu skriveni. Glavni kandidati za skrivanje jesu podaci-članovi dok su glavni kandidati za otvaranje u stvari metode. Objekti -članovi se po potrebi ili zatvaraju ili otvaraju. Zatvoreni deo počinje labelom, tj službenom rečju **private**, dok otvoreni deo počinje rečju **public**. Otvorenost ili zatvorenost traje sve dok se ne pojavi suprotna labela ili do kraja klase. Obično se prvo definiše zatvoreni deo. Radi uvođenja nekih novih stvari daljim tokom predavanja definisaćemo klasu kompleksnog broja koja se inače ovako ne definiše ali izmene ćemo uraditi kada budemo radili nove stvari iz C++. Takođe uvedeno je prenošenje promenljivih po referenci koje je slično kao pokazivačima ali za razliku od njih ona se ne dereferencira.

```
class Complex {
private:
      double r,i;

public:
      void create (double rr, double ii){
            r=rr; i=ii }
double re() {return r;}
double im() {return i;}
void conjugate() {i=-i;}
void modarg (double &, double &);
};
```

```
void Complex::modarg (double &mod,
                      double &arg) {
      mod=sqrt(x*x+y*y);
      arg=(r==0&&i==0)?0:atan(i,r);
}
...
Complex z;
double p,q;
...
z.modarg (p,q);
```

```
class K {
private:
      L obl;
public:
      L &getObl() { return obl; }
};


K obk;
...
obk.getObl().m()
```

Referenca se čvrsto vezuje za objekat a kada se pristupa preko nje nema nekog posebnog operatora. Ona se vezuje za funkcije i to za prenos promenljivih po adresi. Npr kada u private delu klase imamo objekat klase mi njemu ne mozemo direktno pristupati već jedino preko metode, tj moramo odmah početi razmišljati o pisanju metode koja će pristupati tom zatvorenom objektu. Zato pišemo metodu getObl. Ali sada prilikom poziva te metode u glavnom programu ova metoda neće raditi jer se operacija `return obl` izvodi nad kopijom koja se nalazi na steku dok original ostaje isti. Da bi smo to razrešili stavljamo referencu. Sada metoda vraća original i radi sa originalom. Predhodno navedene upotrebe reference ne znače da je ona u potpunosti istisnula pokazivače iz

upotrebe. Štaviše objektima se rukuje preko pokazivača i od svih programskih jezika najviše se koriste u C++. U Javi objektima se radi isključivo preko adrese.

U gore napisanoj klasi Complex nedostaju operacije sa kompleksnim brojevima kao što su sabiranje, oduzimanje... Sada ćemo napisati operaciju sabiranja kompleksnih brojeva add ali je nećemo izvesti kao metodu jer to nije u duhu C++ i izaziva mnoštvo nedoslednosti sa matematičkim aparatom sabiranja kompleksnih brojeva. Evo i zašto. Kada bi mi i realizovali add kao metodu to bi kasnije otvorilo čitav niz problema kod preklapanja promenljivih, izvođenja tipa... Pretpostavimo da smo napravili metodu add i imamo da saberemo dva kompleksna broja z_1+z_2. I sada pozovemo metodu add kao z1.add(z2) ili z2.add(z1) koja će raditi zbog toga što su operandi potpuno ravnopravni i važi zakon komutacije. Ali upravo tu leži problem jer ako su oni ravnopravni mi sa ovom našom metodom jedan od njih smatramo neravnopravnim. Još veći problem nastaje kada pokušamo sabirati kompleksan broj i realan. Dok će z1.add(r) raditi već r.add(z1) neće jer je r običan tip double i nikakve metode nad njim ne dolaze u obzir. U klasi komplex sabiranje ćemo realizovati slobodnom funkcijom koja se nalazi izvan definicije klase:

I ovde takođe postoji jedna stvar bitna za napomenuti. Ova funkcija je u odnosu na klasu u stvari klijent. Ova funkcije je u stvari po logici usko vezana za klasu i kada bi pisali ovu klasu i funkciju one bi se nalazile u istoj biblioteci takođe postoji i mehanizam kojim se ova funkcija i fizički veže za klasu.

```
Complex add (Complex z1, Complex z2) {
      Complex w;
      w.create(z1.re()+z2.re(),z1.im()+z2.im());
      return w;
 }
```

Apstrakcija i skrivanje informacija

Kod predmeta kojima mi manipulišemo u programiranju nas zanimaju samo bitne stvari koje su vezane za dati domen problema. Tako su **relevantne** osobine u stvari one osobine koje su vezane za domen problema. **Apstrakcija** je radnja kojom izostavljamo pojedinačno, slučajno, sporedno, a zadržavamo opšte, bitno, nužno, važno. Shvatiti apstahovanje je jedna od najbitnijih stvari u programiranju. Kada objektno programiramo mi u stvari modelujemo stvari i upravo pri modelovanju mi se koristimo apstrakcijom. **Model** je uprošćena slika onoga što modelujemo, i ona je uprošćena upravo apstrakcijom. Od svih apstrakcija u našoj stuci nas zanimaju samo dve:

1. **Apstrakcija entiteta** (stvari). Ova vrsta apstrakcije jeste model neke konkretne stvari. Najbolji primer jeste materijalna tačka jer smo nju sveli apstrahovanjem samo na masu. Ovu vrstu apstrakcije imamo kada zadržimo samo bitne osobine nekog entiteta.
2. U ovoj apstrakciji detalji realizacije su proglašeni za nebitne. Npr u C-u sin(x) je apstrakcija jer mi ovu funkciju koristimo bez znanja kako ona radi tj kako se izvršava, i detalji realizacije nas ne zanimaju. Npr vozač automobila vozi auto i dodaje gas pritiskom na papučicu gasa bez znanja šta se u stvari događa ispod haube tim njegovim postupkom. On jednostavno zna da se brzina automobila povećava. On u stvari vidi auto kroz komande. Ova apstrakcija je karakteristična za svaku metodu jer kada je pozovemo mi u suštini ne znamo kako ona radi.

Na ovu drugu vrstu apstrakcije nastavlja se jedan od najbitnijih principa u programiranju, a to je princip **skrivanja informacija** (eng. – Principle of information hiding). Ovaj princip je postavio i formulisao Parnas još 1972. kao jedan od kriterijuma za dekompoziciju složenog sistema na module. Detalji realizacije ne samo da treba da budu razdvojeni od interfejsa nego treba da budu skriveni od korisnika. Ukoliko ovaj princip nije poštovan program koji smo napisali ništa ne valja. To je jedna od stvari kojih se strogo pridržavamo. Sve što je realizacija treba da bude nedostupno klijentima. Pri skrivanju informacija težimo da olakšamo klijentu rad bez znanja funkcionisanja onoga što upotrebljava. Klasom treba rukovati preko metoda. Takođe kao prednost skrivanja informacija dolazi da korisnik i ne oseti kada mi promenimo neku metodu tj kada nadogradimo naš

softver jer će on raditi isti posao ali sada možda na drugačiji i bolji način, ali to korisnik ne primećuje osim možda u brzini izvršenja naredbe.

Inkapsulacija i modularnost

Kvalitetan softver osim apstrakcije i skrivanja informacija mora zadovoljiti i ove pojmove. Svi objektni jezici nas primoravaju da vršimo inkapsulaciju. **Inkapsulacija** je sredstvo za objedinjavaje strukture i ponašanja u softversku celinu tako da bude ostvarena kontrola pristupa. U C++ strukturu čine polja a ponašanje čine metode. Inkapsulacija u C++ je sredstvo za objedinjavanje podataka-članova, objekata-članova i funkcija-čanova u jednu celinu. Inkapsulaciju čine naredba class, zatim ako se metode ne nalaze u klasi koristi se ::, pa labele private i public i još jedan nivo kontrole pristupa a to je protected. U programerskom žargonu mi inkapsulaciju koristimo i šire, tj za označavanje nečega što je pogodno za upakovati.

Parnas je takođe postavio prve principe modularnosti. U programskom svetu ima nesporazum oko toga šta je modul. Nekada je potprogram bio modul. Razdvajamo modularnost kao osobinu softvera i modul kao softversku komponentu. Modularnost po Mejeru podrazumeva dve stvari: proširivost i višekratna upotreba. Proširivost je mogućnost dodavanja osobina ali bez menjanja izvornog koda. Modul je svaka ona softverska komponenta koja ima dve osobine:

1. Realizuje se autonomno tj projektuje, kodira i testira se autonomno. Potprogram nema tu osobinu.
2. Predstavlja skup servisa namenjen svakom klijentu.

Besmisleno je govoriti o izvršenju modula (on se ne može izvršiti). U C-u ulogu modula igra biblioteka, i to se takođe koristi i u C++. Bertran Mejer je inaogurisao jedno pravilo **klasa=modul** i to se zove mejerova jednakost. Ona nalaže da se klasa obavezno nalazi u modulu tj u biblioteci jer mora postojati višekratna upotreba. Osim te klase nema druge klase u tom modulu tj svaka klasa se nalazi u svojoj biblioteci. U C++ to nije uvek moguće jer se ozbiljan program bazira na mnoštvu klasa. Ukoliko uključimo previše biblioteka dolazi do razmrvljenja softvera i mi tada ne znamo kako šta funkcioniše. Zbog toga je ova jednačina uprošćena na **klasa∈modul** a to znači da se u istu biblioteku smeštaju srodne klase tj ne mogu se u istom modulu naći potpuno disjunktne klase. Modul mora da ima osobinu logičke kohezije.

Biblioteka tj modul treba da se sastoji od dva dela i to interfejsa i skrivenog tj realizacionog dela. Prvi u C-u ima ekstenziju .h koja se koristi i u C++ ali se u ovom programskom jeziku vise koristi .hpp. Za drugi deo slično kao prethodno su karakteristične ekstenzije .c i .cpp. Interfejs sadrži deo modula koji je otvoren za korisnika i u njemu se obično nalaze prototipovi funkcija dok je drugi deo skriven i predstavlja u stvari skup funkcija koje su u prvom delu samo deklarisane. Realizacioni deo u stvari sadrži realizaciju klase i zato on ne treba da bude vidljiv za korisnika. Takođe ceo modul se može napraviti u interfejsu ali se onda cela datoteka mora uvek prevoditi zato što #include uključuje ceo interfejs.

Isti modul može da se pojavi na više mesta u programu ali njegova definicija može da se pojavi samo jednom. Zato je ona zaštićena sa #ifndef sp, #define sp i #endif. Zbog ovih naredbi klasa ostaje definisana samo jednom.

Klasifikacija operacija

Jedan od najtežih poslova pri izradi klase jeste izbor metoda i pitanje je koliko metoda odabrati za datu klasu i taj skup metoda ne sme biti ni premali ni preveliki. Ukoliko je premali mi nedovoljno modelujemo dati problem, a ukoliko je preveliki zbog toga što preveliki broj metoda znatno otežava rukovanje objektom. Klasifikacija metoda pripomaže u ovome problemu. Ona ima teorijsku podlogu u knjizi „Strukturirano programiranje“ . Kaže se da se sve u programu može posmatrati kao operacija. Npr int i; se do tada posmatrala kao samo deklaracija, opisna naredba. 70-ih godina se taj pogled promenio i od tada se smatra za operaciju jer se u najmanju ruku pozivanjem ove operacije zauzima memorija. U C-u imamo ograničene mogućnosti da utičemo na njeno izvršenje kao npr int i=0;. U objektnom programu ovo se shvata kao operacija, tj mora se

shvatiti kao metoda. Još tada je izvršena klasifikacija operacija. Inače klasifikacija ima mnogo ali generalno metode klasifikujemo u pet grupa koje i nisu potpuno disjunktne.

Konstruktori su metode koje imaju glavni zadatak da učestvuju u kreiranju objekata ili promenljive, i mogu ali i ne moraju da sadrže njihovu inicijalizaciju. **Inicijalizator** je operacija kojim se objekat isključivo dovodi u neko početno stanje. U C++ ovaj postupak uglavnom radi konstruktor. Npr kada realizujemo konstruktor steka inicijalizovali bi ga na prazan stek.

Destruktori su poput konstruktora specijalne metode koje ili direktno uništavaju ranije kreiran objekat ili pak učestvuju u toj operaciji. U C-u postoji operacija free koja ima ulogu destruktora. **Termnator** dovodi objekat u završno stanje. Deskriptor može da sadrži terminator koji takođe može da bude i nezavisan. Njihov odnos je isti kao i odnos konstruktra i inicijalizatora.

Akcesor (pristupnik) je operacija kojom se pristupa sadržaju. Oni se dalje dele na **selektore**, koji vrše odabir nekih podataka (tipičan selektor je „.“ u C++), i **indikatore**, a to su obične metode kojima se očitavaju obični podaci. Skup indikatora mora biti takav da pomoću njih mora biti određeno stanje objekta. Oni često imaju prefiks get pa ih zovemo još i geteri.

Modifikatori su metode kojima se menja stanje objekta. Oni često imaju prefiks set pa ih još zovemo i seteri. Modifikator može da bude u klasi a i van nje. Praktično nema klase bez modifikatora budući da je u prirodi objekta da bude aktivan.

Iteratori sačinjavaju grupu operacija koje se vezuju za složene tipove i klase. U opštem slučaju se primenjuju na **kontejnerske klase** (tj objektne realizacije struktura podataka). Oni omogućuju sistematski, sukcesivan pristup elementima. U C-u naredba for je iterator za niz. Iterator ima tri elementa i to su: startovanje, prelazak na sledeći element i provera kraja. Ove tri stvari su neizbežne jer su fundamentalni delovi svakog iteratora. Iterator npr može da se definiše i za stablo kod obilazaka stabla.

Ova klasifikacija operacija sadrži i stavku ostalo jer ne možemo sve metode uvek smestiti u prethodnih pet vrsta jer postoji mogućnost da napravimo metodu koja ne spada ni u jednu vrstu ili metodu koja je mešavina nekoliko vrsta.

Konstruktori

```
Class Complex {
private:
      double r,i;
public:
(1)   Complex () {
      r=i=0; }
(2)   Complex (double rr,
                double ii) {
      r=rr; i=ii; }
(3)   Complex (Complex *);
(4)   Complex (Complex &);
};
```

```
Complex::Complex (Complex *z) {
      r=z->r; i=z->i; }
Complex::Complex (const
                  Complex &z) {
      r=z.r; i=z.i; }
```

```
Complex t;
t=Complex(1,3);
```

Svaka klasa mora imati jedan ili više konstruktora. To je u stvari metoda za kreiranje objekta iz klase tj **instanciranje** klase. Konstruktori su specijalne metode na koje programer ima uticaj. Konstruktor će deo posla da obavi bez našeg mešanja (npr zauzimanje memorije) i u tom smislu su ovo specijalne metode. Mi najčešće radimo inicijalizaciju objekta u sklopu konstruktora. Konstruktori imaju specijalne sintaksne osobine:

1. Konstruktor nosi ime klase. Svi konstruktori se zovu identično kao i klasa u kojoj se nalaze.
2. Pri pisanju konstruktora se ne navodi njegov tip jer se podrazumeva da je on isti kao i tip klase. Zato se kaže da on nema nikakav tip. A rezultat rada konstruktora je stvaranje objekta pa zato je isti tip kao i klasa.
3. Konstruktor se najvidljivije od ostalih metoda ralikuje po pozivu. Metode se pozivaju preko objekta a konstruktori se ne pozivaju tako. Razlog je tehničke prirode jer pozivom npr Complex a,b,c; možemo stvoriti tri objekta. Ovde se u stvari vrši poziv i to tri puta. Poziv konstruktora može da se nađe i u izrazu. Takođe kada bi napisali z.Complex () mi šaljemo poruku nepostojećem objektu.

U klasi iza reči public uvek se prvo pišu konstruktori zatim destruktori a zatim i ostale metode. Svaka klasa ima bar dva

konstruktora iako ih mi ne napišemo. Zato se oni zovu **ugrađeni konstruktori**. Jedan je konstruktor objekta a drugi konstruktor kopije. Ako napišemo konstruktor ugrađeni se više ne koristi. Ugrađeni konstruktor ima ulogu samo u zauzimanju memorije.
Prvi konstruktor se poziva sa Complex z; a drugi sa Complex t(1,2); ili pod jednim pozivom Comlex z,t(1,2); Treći se poziva Complex u(&w); a četvrti sa Complex w(z);

Postoji još jedna vrsta konstruktora a to je **podrazumevani konstruktor** a on nema ni parametre. U određenim situacijama konstruktor klase se poziva implicitno. Ako u klasi ima mnogo konstruktora prilikom implicitnog poziva, poziva se podrazumevani konstruktor. C++ zabranjuje konstruktore čiji prototip izgleda Complex (Complex);

Konstruktor se može pozivati i u vidu slobodne funkcije. Pri ovakvom pozivanju konstruiše se bezimeni objekat. On se konstruiše na steku a posle toga biva kopiran u objekat t. C++ iz posebnih razloga ima još jedan način realizacije konstruktora koji se često koristi a to je **konstruktor – inicijalizator**. Na primer ukoliko uzmemo one konstruktore napisane po (1) i (2) i napišemo ih kao konstruktore realizator oni bi izgledali (1') i (2'). Posle ( ) sledi inicijalizacioni blok i vrednosti koje će biti pridružene podacima-članovima i objektima-članovima. Ovaj konstruktor se vrlo često koristi i uveden je zbog inicijalizacije objekata-članova.

Npr ako imamo klasu L i klasu K:

```
(1') Complex ():r(0),i(0){}
(2') Complex (double rr, double ii):
       r(rr),i(ii){}
```

U klasi K se napiše ime objekta-člana a u zagradu argumenti po kojima se prepoznaje koji se konstruktor iz klase L koristi kada se poziva objekat.

```
class L {
private:
      double x,y;
public:
      L (double xx, double yy) {
            x=xx; y=yy; }
};
```

```
class K {
private:
      int m;
      L obl;
public:
      K (int k, double x1, double x2):
            m(k),obl(x1,y1){}
... };
```

Podrazumevane vrednosti parametara

Podrazumevane vrednosti parametara se ne vezuju isključivo za konstruktore već i za slobodne funkcije i metode. U pitanju je novina koja se odnosi i na C. Ovaj mehanizam omogućava da funkciju pozovemo sa manje parametara nego što je navedeno argumenata u njoj. Npr funkciju f možemo pozvati na više načina. Kao npr f (p, -0.5, 10, 0); ili f (1, 3, 10) ili f (a, b);

```
void f(double x, double z, int j=0, double t=-1);
```

Vrednosti koje nisu navedene u pozivu funkcije podrazumevane su tako što su navedene uz argument. Kod ovakvog definisanja funkcija postoji samo jedno pravilo, a to je da se prilikom pisanja prototipa prvo navedu svi parametki koji nemaju podrazumevane vrednosti pa onda oni koji imaju. Na primer umesto konstruktora (1') i (2') možemo pisati novi konstruktor koji sadrži podrazumevane vrednosti. Ovaj konstruktor zamenjuje oba predhodna i bolji je od njih zajedno jer može da se poziva na tri načina: Complex z1, z2(3,5), z3(10).

```
Complex (double rr=0, double ii=0):r(rr),i(ii){}
```

Konstruktor kopije

Konstruktor kopije je vezan za C++ i on je u stvari iznuđeno rešenje pre nego neka prednost. On služi za prevljenje kopije objekta baš kako mu i ime govori. Pojavljuje se u kontekstu prenošenja argumenata po

vrednosti. Npr kada prenosimo neki objekat po vrednosti i taj prenos je jedna univerzalna stvar koja ne zavisi ni od programskog jezika ni od metodologije. Tada se jednostavno pravi kopija objekta na steku i to se u mnogome razlikuje od pravljenja kopije nekog baznog tipa na steku. Ova kopija se u stvari konstruiše na steku i mora da ima svoj konstruktor i to konstruktor kopije.

Konstruktor kopije postoji u svakoj klasi. On se prepoznaje po tipu parametara jer za parametre ima referencu na svoju klasu. Npr Complex (Complex &); se pisalo kod starijih vezija prevodioca dok je danas uobičajno da se piše Complex (const Complex &);. On se prilikom poziva slobodne funkcije automatski uključuje. Npr funkcija Complex g(a); vraća kao rezultat objekat i ostavlja ga na steku i onda ga neko preuzima. Objekat koji g prenosi konstruiše se na steku od strane konstruktora kopije. Kada se konstruktor kopije ne navodi aktiviraće se ugrađeni konstruktor kopije Taj konstruktor kopije pravi bit po bit. Pozivanje konstruktora kopije se može izbeći prenosom argumenata po adresi ali i tada će se on možda upaliti jer može da se desi da se rezultat formira na steku i adresa koja se vraća je adresa objekta na steku.

Kada imamo dinamički član moramo da pravimo konstruktor kopije gde parametak ima prirodu lokalne promenljive. Pozivanjem funkcije void f (list); uključiće se konstruktor kopije i to ugrađeni tj kopiraće se samo pokazivač na prvi član naše liste (jer se jedino on nalazi u klasi). I sada imamo dva objekta koji dele istu memorijsku lokaciju: original i kopija koja ima iste memorijske lokacije osim kopiranog pokazivača na prvi član. Kada se funkcija f završi objekat izlazi iz opsega pa se aktivira destruktor kopije i originalni ulazni podatak biva obrisan. Prema tome konstruktor kopije mora da napravi kopiju ne samo pokazivača već i cele dinamičke strukture, u ovom slučaju cele liste. Ako tako uradi destruktor kopije će obrisati samo kopiju i original ostaje netaknut.

Postoje dve vrste kopija a to su plitke (kopije statičkih struktura) i duboke (kopije dinamičkih struktura). Kada se one poklapaju ne trebaju se praviti posebni konstruktor i destruktor kopije već su dovoljni ugrađeni.

Destruktor

U C++ destruktor je takođe iznuđeno rešenje kao i konstruktor kopije. **Destruktor** ima zadatak da uništi objekat. Po njegovoj primeni objekat nije više na našem raspolaganju. Karakteriše se time što se pre njega ne navodi nikakav tip jer je podrazumevani tip void. Druga karakteristika je njegov poziv jer indentifikator ima specijalni znak ~ npr: ~ImeKlase() i nikada nema nikakve parametre stim da programer ubacuje u telo destruktora svoje naredbe koje se izvršavaju pri izvršavanju destruktora. Svaka klasa ima samo jedan destruktor. To znači da svaka klasa ima konstruktor originala, konstruktor kopije i destruktor.

Destruktor može da se pozove na dva načina. Prvi i najčešći je implicitni poziv. On se uključije prilikom izlaska objekta iz opsega. C++ nudi i drugo rešenje a to je da se pozove preko objekta kao npr z.~Comlex().

Postoji i situacija kada destruktor moramo da napravimo sami, a to je kada u klasi postoji neki dinamički član. P makar bio i jedan pokazivač na heap moramo sami napraviti destruktor da nebi došlo do curenja memorije (memory leak). Kod curenja memorije nije bitno koliko bajtova curi jer mi ne smemo dopustiti da curi niti jedan jedini bajt. Reimo da radimo sa jednostruko spregnutom listom u objektu će biti samo pokazivač na prvi element koji se nalazi na heapu. Dešava se da destruktor kada objekat izađe van opsega on uništava samo pokazivač na prvi element a lista ostaje u memoriji i tako dolazi do curenja memorije. To se dešava zbog toga što je destruktor u stanju zaključiti šta treba obrisati samo dok je to u sklopu memorije objekta, a u ovom slučaju to je samo pokazivač na prvi član liste. Zato smo prinuđeni praviti destruktor gde ćemo mi moratu da obezbedimo pražnjenje liste.

U C++ destruktor u stvari ne uništava objekte već to samo tako izgleda korisnicima. On ga u stvari ne uništava zato što je to nemoguće. Ukoliko imamo neki objekat u statičkom delu memorije on se ne može uništiti jer to je podatak koji je već unet pa iako stavmo sve nule i to je opet podatak. Npr objekat se može uništiti kada je na steku spuštanjem pokazivača vrha steka, ili kada je na heapu tako što se u tabele heapa upiše da su te obektove lokacije prazne. To npr radi operacija free. Tako što se tiče ove adnje dve situacije objekat je isti i za promenlljive i za objekte jer u principu se radi ista stvar.

Ukoliko u C++ napišemo npr obK.~K( ) dešava se samo ono što je programer predvideo u telu programa. Preporuka u C++ je da se destruktor ne poziva već da se pusti da objekat izađe iz opsega i da se destruktor uključi implicitno i to se skoro uvek i radi.

Prijateljske funkcije i klase

U programiranju se za prijateljske funkcije i klase često koristi i još dva izraza kao što su kooperativne funkcije ili **friend** funkcije. Friend funkcija je u stvari slobodna funkcija. Relacija kooperativnosti između slobodne funkcije i klase znači da funkcija ima pristup svim elementima (osim polju **this** koje je pokazivač objekta na samog sebe) bilo da su oni public ili private. Ta slobodna funkcija nije u stvari fizički član te klase već je logički tesno vezana za tu klasu i čak se nalazi u istom modulu sa klassom pa joj se zato dodeljuje i status prijateljske. Proglašavanje funkcija za prijateljske se ogleda u **povećanju brzine** izvršenja funkcije. Npr funkcija add treba da bude član klase Complex ali po svojoj prirodi ona nije metoda već je slobodna funkcija. Ona se uvek nalazi u istoj biblioteci sa klasom pa se zato deklariše kao friend funkcija. Takođe ovo proglašavanje funkcije za prijateljsku u stvari omogućuje da se logička povezanost funkcije sa klasom i **fizički ostvari**.

```
Class Complex {
private:
      double r,i;
public:
...
friend Complex add (Complex, Complex);
}
```

```
Complex add (Complex z1, Complex z2) {
      Complex w;
      w.r=z1.r+z2.r;
      w.i=z1.i+z2.i;
      return w; }
```

Prilikom ovog proglašenja funkcija za prijateljske, princip skrivanja inormacija ne sme biti narušen. Da se to ne bi desilo i da ne bi smo imali neovlašten pristup private polju klasa u stvari proglašava funkciju prijateljskom a ne funkcija samu sebe. Npr za funkciju Complex imamo prijateljsku funkciju add koju sada možemo malo drugačije da deklarišemo nego što smo mogli dok nije bila proglašena za prijateljsku. Sve što treba je napisati reč **friend** i posle napisati prototip funkcije. Kada god u C++ imamo slobodnu funkciju koja je u svakoj mogućoj vezi sa klasom ona će biti proglašena za friend metodu. Tu dakle dolazi i do povećanja brzine.

U C++ postoje određeni principi na kojima je ovaj jezik zasnovan i između ostalih tu je princip da je u klasi sve zatvoreno i ukoliko ne napišemo drugačije podrazumeva se da su svi podaci private. Zato mi moramo da sa labelom public otvaramo pojedine delove koje hoćemo da koristimo i van klase. Ukoliko imamo dve klase i ako su one u određenoj međusobnoj vezi one se ponašaju kao klijenti. Kada se dve klase (recimo K i L) nađu u istoj biblioteci postavlja se pitanje zašto bi one bile zatvorene jedna za drugu kada su zajedno? U C++ zaštita zavisi za svaku klasu. Odgovor na pitanje je u tome da se klase proglase kao friend klase jedna u drugoj. U C++ se takođe i žrtvuje brzina kako bi se one ponašale kao klijenti. Jednostavno se napiše *friend class L*; u klasi K i *friend class K;* u klasi L. Ovo se radi samo kada su klase u istoj biblioteci i kada su međusobno logički povezane i ovo je način kako da se i fizički čvršće povežu.

Takođe za prijateljske funkcije možemo proglasiti i metode jedne klase u drugoj. Tako ukoliko hoćemo neku metodu iz klase L da proglasimo za prijateljsku u klasi K mi ćemo jednostavno u klasi K napisati *friend L::met();*

Uvod u polimorfizam. Preklapanje operatora.

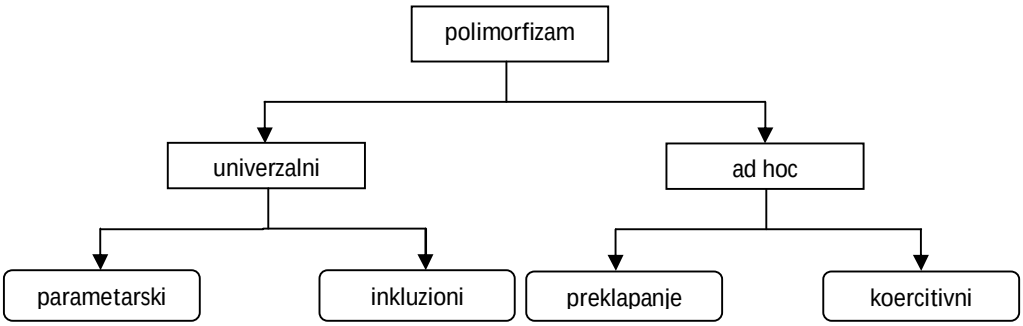
Polimorfizam je konteksno zavisno ponašanje. Konteksno zavisno se ponašaju promenljive i objekti , ali i funkcije. Polimorfizam se prepoznaje i kod čitavih klasa posoje četiri vrste polimorfizama.

Jedna vrsta se tipično susreće u Pascalu. Npr pascal ima dve funkcije *pred* i *succ*. Ove funkcije se ponašaju polimorfno. Npr u pascalu znakovni tip i celobrojni tip nemaju nikakve veze ove dve funkcije mogu da prihvate oba tipa i u zavisnosti od tipa generišu rezultat. To je kontekstno zavisno ponašanje.

Drugi vid možemo posmatrati na sledećem primeru. Ukoliko imamo char c; logika nalaže da c+1 nema nikakvog smisla za napisati. Ali pošto je char podtip od celobrojnog tipa u C-u, promenljiva c se u ovom slučaju ponaša kao promenljiva celobrojnog tipa a posle nastavlja kao char tipa.

Treći vid možemo primetiti ukoliko pogledamo operator + u pascalu. On može da znači tri stvari ukoliko ga vidimo u izrazu a+b. Može da znači sabiranje, pa ukoliko su a i b stringovi može da znači spajanje stringova, i ukoliko su a i b tipa skupa može da znači uniju skupova. Kontekst je ovde u stvari tip operatora i u zavisnosti od njega ovaj operator se ponaša različito. Npr u C-u imamo operator * koja može da znači množenje i pokazivač. Ali ovo nije pravi polimorfizam pošto je ista oznaka uvedena samo zbog nedostatka simbola na tastaturi.

Četvrta vrsta je tipična za C gde operatorom *(int) y* mi primoravamo y da se ponaša kao int promenljiva. Podela polimorfizama se najbolje očitavana sledećem dijagramu:



Univerzalni polimorfizmi su karakteristični po tome što se ista kategorija ponaša različito. To su prve dve vrste polimorfizama o kojima smo pričali malopre. Kod **ad hoc polimorfizama** ne radi se o istoj kategoriji.

Parametarski polimorfizam je najstariji polimorfizam. On se zapaža i kod funkcija i kod klasa. Parametarski polimorfizam funkcije je njena osobina da ona podešava svoje ponašanje prema tipu podatk. On je u stvari naša prva vrsta polimorfizma od malopre. Kao što potprogram ima parametre koji tek pri pozivu dobijaju vrednost tako je i funkcija u stvari parametarizovana. Takve klase i tipovi koji iskazuju osobine ovoga polimorfizma nazivaju se **generičke klase** i **generički tipovi**. Generičke klase i tipovi su u stvari parametarizovani drugim tipovima. Generički tip u C-u je niz. Ukoliko napišemo niz a mi moramo znati kog je tipa taj niz. Niz je u stvari parametarizovan tipom. Niz je generički tip.

Inkluzioni polimorfizam je malopre opisan kao druga vrsta polimorfizma u prvom delu teksta. Kada se promenljiva ili objekat ponaša kao da menaj tip. Ova vrsta polimorfizma je posebno važna za objekte. Od svih polimorfizama ovaj je najvažniji jer on u kombinaciji sa nasleđivanjem predstavlja srž objektnog programiranja.

Preklapanje je polimorfizam kada u jednoj klasi mogu da se nađu funkcije sa istim imenom i operatori sa istim znakom. To je treća vrsta. Na engleskom se preklapanje naziva **overload**.

Koercitivni polimorfizam (četvrta vrsta) ili na engleskom **coertion** ili još se naziva **prinudni polimorfizam**. Promenljiva biva prinuđena da se ponaša kao neka druga promenljiva ili kao promenljiva drugog tipa. Tj biva prinuđena da promeni svoj tip.

Preklapanje funkcija i operatora

Preklapati se mogu i slobodne funkcije kao i metode, a takođe preklapaju se i operatori. Preklapanje predstavlja zadavanje istog imena različitim rutinama. Tako operator + u paskalu znači i sabiranje i uniju i spajanje stringova. Mehanizmi preklapanja funkcija su jednostavniji nego mehanizmi preklapanja operatora, i takođe preklapanje funkcija je mnogo bitnije za programiranje.

Preklapanje funkcija. C++ dozvoljava da dve različite metode u dve klase nose isto ime jer kod poziva datih metoda ne može da dođe do zabune jer se tačno vidi na koji se objekat koja funkcija odnosi. Radi se o tome da metode, ili slobodne funkcije, koje u klasama rade isti posao treba da nose isto ime. To se radi zbog toga što kod velikih projekata i programa imamo mnoštvo klasa i kada bi smo još neke funkcije koje rade isti posao nazivali različito od klase do klase, javio bi se problem da ne bi smo bili u stanju da zapamtimo sve te

nazive. Pravilo da metode koje rade isti posao imenujemo jednako se zove **pravilo očuvanja sintakse** i glasi: funkcije koje imaju isto ime rade isti posao, i obrnuto. Ovoga pravila mi se strogo pridržavamo.

Drugi slučaj preklapanja funkcija nastaje kada se funkcije sa istim imenima nađu u istoj klasi. C++ dozvoljava da se funkcije istog imena nađu na istom mestu. Funkcije sa desne strane mogu da se nađu u istom mestu. Kada logički razmatramo ovaj skup funkcija shvatamo da jedino mesto gde može da se javi problem jeste u stvari kod poziva jer tada prevodilac možda ne bi bio u stanju da razlikuje poslednje dve funkcije kada ih mi pozivamo. To se naravno neće dogoditi jer se funkcije prepoznaju po parametrima tako da njihov tip kod prepoznavanja uopšte i nije bitan. C++ funkcije se dakle odlikuju imenom i listom parametara. Minimalna razlika između dve funkcije mora biti barem tip jednog parametra. Zabune može biti samo ukoliko npr funkcija double f ima jedan parametar sa podrazumevanom vrednošću pa prilikom poziva ne znamo da li se poziva prva ili druga funkcija. Ovaj problem se ne da tako lako rešiti pa je preporuka da se kod funkcija koje se preklapaju ne koriste parametri sa podrazumevanim vrednostima. Npr preklapanje se primećuje i kod konstruktora jer svi konstruktori u istoj klasi nose isto ime. Npr kod ovakvog pisanja funkcije abs klijent stiče utisak da postoji samo jedna funkcija abs koja prepoznaje tip.

```
void f (double);
double f (double, int);
int f (double, double);
```

```
int abs (int x) {
      return (x<0)?-x:x; }
double abs (double x) {
      return (x<0)?-x:x; }
long abs (long x) {
      return (x<0)?-x:x; }
long double abs (long double x) {
      return (x<0)?-x:x; }
```

Preklapanje operatora

U C++ različitim mehanizmima možemo pridruživati isti simbol operacijama koje imaju istu ili barem veoma sličnu semantiku. Svrha preklapanja je olakšani rad sa vrednostima i pisanje operacija koje je bliže logici. Npr kod klase Complex imamo definisane operacije sa kompleksnim brojevima. E sada ukoliko bi želeli da napišemo izraz a+b/c+(d-e)*f to bi išlo veoma teško sa ovakvim metodama [npr add(a,divide(b,c),multiple(....)] ukoliko ne bi smo razložili ovaj izraz u više linija i obavili operacije deo po deo. Preklapanje operatora nam omogućuje da programiramo dejstvo operatora. Preklapanje operatora spada po svojoj prirodi i u objektno i u procedurno programiranje. Ono je u svakom programskom jeziku ili ugrađeno ili nikako ne postoji. U C++ je ugrađeno i moguće ga je izvesti samo u objektnom okruženju i C++ nudi moćna sredstva za preklapanje operatora. Za preklapanje operatora važi **zakon o održanju semantike** koji kaže da prilikom biranja operatora koji preklapamo mi biramo operator koji je osobinama isti kao naš ili bar veoma sličan.

Ukoliko hoćemo da dodelimo ime + operatoru mi to možemo uraditi. Npr umesto z=add(a,b) mi možemo jednostavnije pisati z=+(a,b) ukoliko damo ime funkciji +. Takvo ime mogu nositi samo posebne vrste funkcija i to **operatorske funkcije**. Znači ne možemo funkciju definisati sa +(a,b) ali zato možemo upotrebom službene reči **operator** i možemo dakle napisati operator + (a,b). Ali postavlja se pitanje šta smo mi ovim dobili jer opet ovu funkciju moramo pozvati kao add a ima čak i više slova? Odgovor na to se nalazi u specifičnom pozivu ovakvih funkcija kod kojih umesto npr z=operator +(a,b) možemo mnogo jednostavnije da napišemo z=a+b što je ekvivalentno sa malopređašnjim pozivom.

Od grupe do grupe funkcija mi rešavamo problem ravnopravnosti operanada tako što su operatori definisani ili kao metode ili kao friend funkcije. Preklopiti se ne mogu sledeći operatori: „.“, „::“, „?:“, „.*“ i „sizeof“. Samo operatori iz C-a mogu da se preklope. Izvedljivo je da se uvede i novi operator ali to je veoma komplikovano zbog toga što se mora širiti i hijerarhija operacija jer se javlja problem prioriteta operacija. Pošto preklapamo već postojeće operatore zadržavaju se sve osobine osnovnog operatora. Zakon očuvanja semantike to nalaže pa se očuvava i hijerarija i smer grupisanja.

U svakoj klasi postoje operatori dodele „=“, adresni operator „&“ i vezivanje elemenata u niz je definisano sa „ , “. Kod preklapanja ovakvih operatora je najbitnije da se držimo principa o očuvanju semantike. Bez ikakvih problema sve klase možemo podeliti u dve grupe:

1. **Metodski orijentisane klase** i one nemaju preklopljenih operatora osim operatora dodele „=“ ili eventualno relacionih operatora „==“ i „!=“. One predstavljaju osnovnu vrstu klasa.
2. **Operatorski orijentisane klase** su klase koje jednostavno nemaju metoda i one se pretežno sastoje od preklopljenih operatera i friend funkcija. One su specijalna vrsta i tipičan primer je klasa Complex.

Postavlja se pitanje da li operator treba preklapati metodom ili slobodnom friend funkcijom? Opšte pravilo, više preporuka nego pravilo, je: ako operator menja stanje operanda onda ga preklapamo metodom, a ako ne menja stanje onda je kandidat friend funkcija ali opet se može preklopiti i metodom.

Svaki objekat ima polje this i ono predstavlja pokazivač objekta na samog sebe. Ovo polje se može pojavljivati samo u metodama. Unutar metode svi pristupi su kvalifikovani sa this i nekada se mora eksplicitno njemu pristupati.

Preklapanje osnovnog operatora dodele. Ovo je najvažniji oblik preklapanja. Iako ovaj operator postoji u svakoj klasi mi često moramo da ga preklapamo. Ovo preklapanje se vrši isključivo metodom i ovo je jedini slučaj kada će nam prevodilac javiti grešku ukoliko pokušamo da preklopimo slobodnom funkcijom. U klasi Complex „=“ bi smo preklopili sa funkcijom definisanom sa desne strane. Zbog principa očuvanja semantike moramo napisati liniju return *this i tako je ovaj princip očuvan jer a=b=c ne bi moglo da radi bez ovoga. Preklapanje „=“ se vrši obavezno kada imamo barem jedan dinamički element u klasi. Jer ukoliko na primer imamo listu, u objektu se nalazi samo pokazivač na listu pa će prilikom dodele samo on biti dodeljen novom objektu. Tu dolazi do deljenja memorije od strane dva objekta što se ne sme dozvoliti. To znači kada imamo dinamičke elemente u klasi moramo posebno napisati konstruktor kopije, destruktor kopije i operator dodele. U tome nam olakšava spoznaja da je kostur metode kojom se preklapa ovaj operator uvek isti. Kao što vidimo na primeru pre nego što se izvrši kopiranje liste moramo da oslobodimo memoriju objekta kome dodeljujemo naš objekat jer ukoliko je i on u sebi sadržavao neke dinamičke elemente doćiće do toga da se prilikom dodele obriše samo pokazivač na te elemente a oni ostanu na heapu i dolazi do curenja memorije. Ukolko želimo da omogućimo npr dodelu liste same sebi (npr lst=lst) moramo da dodamo ne početak funkcije jednu if naredbu. Ukoliko nje ne bi bilo objekat kome se dodeljuje bi bio obrisan a pošto je to u stvari i objekat sa desne strane i on bi bio obrisan i to bi bila velika greška. U Pascalu i Javi ne postoji preklapanje operatora već se u help klase u kojoj ne postoji preklopljeni element ubaci da se ne preporučuje korištenje takve dodele ali mi ipak možemo napraviti metodu kopiranja objekata. Ali ako smo u softverskoj mogućnosti da rešimo problem, kao što jesmo u C++, onda se nećemo obraćati klijentu da nešto ne radi već ćemo to sami obezbediti da može da radi.

```
Complex &operator = (const Comlex &z) {
      r=z.r; i=z.i;
      return *this; }
```

```
List &operator=(const List &rhs){
      if (rhs==this) return *this;
      //clear();
      //(copy);
      return *this;}
```

Preklapanje operatora +=, -=, *=, /=. Ove operacije se preklapaju isključivo metodama jer one menjaju stanje objekta. Možemo ih izraziti kao:

```
Complex &operator+=(const Complex &rhs) {
      r+=rhs.r; i+=rhs.i;
      return *this;}
```

Preklapanje relacionih operanada. Ove operande preklapamo isključivo slobodnim prijateljskim funkcijama. Uzećemo za primer operand ==. On vraća vrednost int. Ukoliko bi smo operator preklopili metodom ne bismo obezbedili ravnopravnost elemenata. Takođe u C++ postoji typecasting i ukoliko bi pokušali da poredimo element iz naše klase i neki drugi element to bi radilo samo kao z1.operator==(r); ali ne i kao r.operator==(z1). To se dešava kada mi realizujemo ovo naše preklapanje metodom čak i kada napravimo poseban typecast za našu klasu (koji je uzgred u iole ozbiljnijim klasama obavezan za napraviti). Sve u svemu nećemo koristiti metodu već slobodnu funkciju koja

```
Class Complex {
...
friend int operator==(const Complex&, const Complex&)
};
int operator==(const Complex &z1, const Complex &z2) {
      return (z1.r==z2.r)&&(z1.i==z2.i); }
```

obavezno mora biti prijateljska. One se razlikuju od metoda jer ne vraćaju this već objekat. Od relacionih operanada dobra praksa je da se preklope == i !=.

Kada napišemo sada a==b mi smo u stvari napisalo operator==(a,b) i ovo zadovoljava malopređašnje zahteve o typecastingu i ravnopravnosti elemenata.

```
Complex operator+(const Complex &z1, const Complex &z2)
      Complex w;
      w.r=z1.r+z2.r;
      w.i=z1.i+z2.i;
      return w; }
```

Preklapanje aritmetičkih operanada. Preklapanje binarnih operatora je veoma slično preklapanju relacionih i obavlja se takođe pomoću slobodne funkcije. Referenca koja bi se stavila pre reči operator da se izbegne konstruktor kopije ne sme se stavljati. Jer ova funkcija vraća objekat a ne njegovu adresu i taj objekat se nalazi na steku. Kada bi vraćali po adresi mi bismo vratili adresu nepostojećeg objekta sa steka jer kada se završi funkcija taj objekat izlazi iz opsega i on se jednostavno uništava. Za rešavanje ovog problema se koriste dve tehnike pisanja funkcije. Prva je **tehnika privremenog objekta** opisana funkcijom gore. Druga tehnika je **tehnika bezimenog objekta** gde se kompletno telo funkcije zameni naredbom `return Complex (z1.r+z2.r, z1.i+z2.i);`

```
Complex operator –(const Complex &z) {
      Complex w;
      w.r=-z.r; w.i=-z.i;
      return w; }
```

Unarni operator promene predznaka -, se preklapa pomoću slobodne funkcije jer ne menja vrednost operanda. Ta funkcija bi bila:

Preklapanje inkrementa i dekrementa. Oni su takođe unarni operatori i menjaju stanje objektima. Zato ih i preklapamo metodama. Kod ovih operanada je specifično to što imaju prefiksnii sufiksni oblik. Problem kod preklapanja ovog operanda nastaje kada prevodilac ne zna koji je koji oblik. Zato u C++ postoji zakrpa koju koristimo tako što kod sufiksnog oblika dodamo neku konstantu tipa int kao parametar koju nikada i nećemo koristiti ali sada prevodilac ovu funkciju prepoznaje kao sufiksnu.

prefiksni oblik:
```
Complex &operator++(){
      ++r; ++i;
      return *this; }
```
sufiksni oblik:
```
Complex &operator++(int k){
      Complex w(r,i);
      r++; i++;
      return w; }
```

Preklapanje operatora (). Preklapanje ovoga operatora nam omogućuje parametrizovan pristup objektu i njegovo pobuđivanje . Npr kada računamo vrednost polinoma obavezno preklapamo ovaj operator.

```
#define MAX_RED 10

class Polinom {
private:
      int n;
      double a[MAX_RED+1];
public:
...
};
double Polinom::operator () (double x) {
      int i; double s;
      for (s=a[n],i=n;i>0;i--)
            s=x+s+a[n-1];
      return s;
}

Polinom p(10);
double r,y;
r=p(y);
```

Preklapanje operacije indeksiranja. Operator indeksiranja preklapamo kada se u private delu klase nalazi skriven neki niz. Ako je niz skriven postavlja se pitanje kako očitati neki i-ti član? Problem možemo rešiti običnom metodom, npr get, ali praktičnijerešenje se ogleda u preklapanju operatora indeksiranja []. Njegovim preklapanjem mi postižemo mogućnost upisa i ispisa i-tog elementa.

Kada napišemo str[i] očitava se i-ta vrednost. Referenca se stavlja zbog upotrebe stringa u c=str[i] ili str[i]=c tj stavlja se zbog dodele.

```
#define MAX_DUZINA 256
class String {
private:
      char s[MAX_DUZINA];
...
};
char &operator[](int i) {
      return s[i]; }
char operator[](int i) const {
      return s[i]; }
```

Konstantne metode su takve metode koje su primenljive na konstantne objekte. Prevodilac ne znasemantiku naših metoda već samo sintaksu pa ne može da zna da li smo na konstantni objekat primenili modifikatore. Ako neka metoda ne menja stanje objekta iza () se stavlja službena reč const. Kod stringova nam treba konstantna metoda [] koju ćemo primenjivati na konstantan string. Kod dobrog programiranja moramo da pazimo da sve metode koje ne menjaju stanje objekta proglasimo za konstantne. Pored parametara i imena indentifikacija funkcija se vrši i pomoću parametra const.

Konverzija

Konverzija u C++ je drugi naziv za prinudni ili preciznije koercitivni polimorfizam. Shodno zakonu očuvanja semantike konverzija se vrši isto kao i u C-u. U C-u postoje dve vrste konverzije: automatska ili implicitna (npr sabiranje double i int tipa), i eksplicitna konverzija (najbolji primer je (int)y). Ove konverzije moraju da se obezbede i uvek se opisuju istim alogoritmima u procedurnim jezicima. U objektnom programiranju ne može se znati tip nekog objektapre nego što ga korisnik ne napravi. Dakle ovo implicira da se konverzija mora obezbediti.

```
class A {
...
public:
A(tip t){...}
A(klasa &b) {...}
...
};
void f(A); double r;
f(r);
```

Konverzija u klasu. Ovde spadaju konverije iz tipa u klasu i iz klase u klasu. Konverzija tipa u klasu se obavlja preko konstruktora. Ovi konstruktori se po potrebi uključuju i automatski kada npr unutar funkcije f imamo objekat kao parametar A. Sada prilikom poziva funkcije sa argumentom r automatski će biti pozvan konstruktor i konvertovaće se r u objekat.

Konverzija iz klase.Kada konvertujemo tip iz klase K u klasu A, dodeljujemo konstruktor klasi A i to oblika A(K&k){...}. Konverzija iz klase u tip se obavlja preklapanjem operatora typecast i piše se metoda `operator tip() {...}`. Sada ako hoćemo da konvertujemo objekat u tip pisaćemo (tip)K.

Može doći i do pojave kružne konverzije (da je konverzija definisana u obe klase) i tada se dešava da prevodilac ne zna šta se u šta pretvara.

Sada klasu Complex možemo tako da napravimo da se ne vidi da iza nje u stvari stoje objekti već da izgleda kao da je ona u stvari tip iz C-a. Preklobili bi smo sve operatore, očitavanje delova broja bi bile funkcije, i definisali bi smo jediničnu imaginarnu jedinicu kao konstantan objekat Complex IM(0,1).

Kod a=b+1, ako su ai b kompleksni brojevi, automatski se poziva konstruktor sa podrazumevanim vrednostima i vrši se konverzija.

Veze između klasa. Nasleđivanje

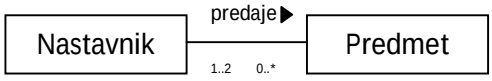
Jezgro objektnog programiranja čine nasleđivanje i inkluzioni polimorfizam. U domenu problema skoro nikada ne egzistira samo jedna klasa tj u jednom problemu klase nikada nisu u potpunosti disjunktne. Naš zadatak pri modelovanju je da izvučemo bitne veze. Zato se govori o vezama između klasa jer retko koja klasa je nezavista. Postoji više vrsta veza, a nasleđivanje je veza koja modeluje odnos subordinacije pojmova. Npr budilnik je sat i mi ga možemo posmatrati i kao budilnik i kao sat. Na engleskom se za nasleđivanje koristi i izraz **subclassing**.

UML je grafički sistem za modelovanje i omogućuje nam da razvijamo dijagrame za naš softver. Prilikom pisanja knjige koja ide uz softver u principu se samo uzmu dijagrami dobijeni sa UML-om i dodaju se uvod i

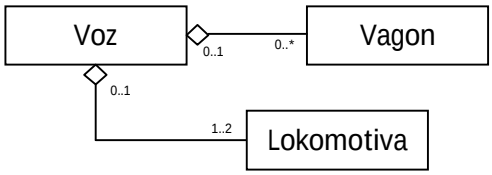
zaključak i to je to. U delu projektovanja i realizacije osnovni dijagram je **dijagram klase**. Predstavlja se kao pravougaonik i ima jedno do tri polja u kojima redom pišu ime klase, private deo i public deo. Obično se mi zaustavljamo na delu u kome je naziv jer druga dva dela znaju biti dosta opširna. Najvažnija svrha dijagrama je da prikaže veze između klasa. Veza je model odnosa između pojmova ili klasa. Veze između klasa generalno se dele na **klijentske veze** i **nasleđivanje**. U okviru klijentskih veza imamo:

1. Asocijacija
2. Agregacija
3. Kompozicija
4. Veze zavisnosti (ostale veze)

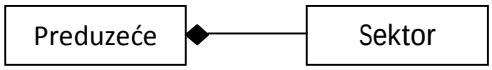
Asocijacija je relacija koja povezuje parove bez posebnih pravila. U UML-u veza se predstavlja sa linijm koja povezje dve klase:



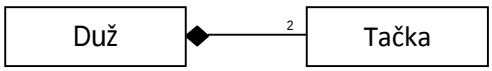
Asocijacija se obavezno dopunjava nazivomodavde sledi da svaki nastavnik prve klase može da bude u vezi sa nekim predmetom iz druge klase. Kardinalitet na strani predmet pokazuje broj predmeta koje može da predaje jedan nastavnik. **Kardinalitet** jeste skup celih nenegativnih brojeva koji pokazuju broj mogućih pojavljivanja različitih objekata iz date klase u u n-torkama sa klasom sa kojom su u vezi. Može da bude i kardinalitet na drugoj strani. Standardni kardinalitet se prikazuje kao „donja..gornja“ granica i sa takvim označavanjem moramo biti sigurni da smo obuhvatili sve slučajeve. U našem slučaju 0..* znači i nula ili više tj jedan nastavnik može da predaje 0 ili više predmeta, a sa druge starne 1..2 znači da jedan predmet mogu da predaju jedan ili dva nastavnika. Pored imena veze dodaje se i trougao u smeru veze. Ako realizator ovoga dijagrama ne zna kardinalitet on neće biti u stanje da realizuje tu vezu. Ovde bi se na primer izvršila realizacija listom.



Agregacija je formalno vrsta asocijacije gde je odnos tipa celina deo. Ovaj odnos je posebno važan i zato ga i izdvajamo iz asocijacije. Na strani celine u UML-u stoji romb. Kardinalitet može da stoji na obe strane ali na strani celine i ne mora. Ovde je karakteristično da delovi postoje i nezavisno od celine. Jednom rečju životni vek celine i objekata su nezavisni.



Kompozicija je posebna vrsta agregacije ali je takođe veoma važna pa se zato navodi kao posebna vrsta veze. Označavanje ove vrste veze u Uml-u je slična kao i agregacije samo što je ovde romb popunjen. Pored nasleđivanja kompozicija je najvažnija veza. Kompozicija je odnos celina deo ali je životni vek dela sadržan u životnom veku celine. Kompozicija je znatno čvršća veza. U sklopu kompozicije postoji i jedna znatno čvršća veza a to je kada se životni vek celine i dela poklapaju. Npr slučaj duži i tačaka koje je određuju jer sa uništavanjem duži mi uništavamo i te tačke sa kojima je ona bila određena. U C++ ova veza predstavlja biti objekat član. Posebna terminologija postoji kod kompozicije. Objekti iz prve klase zovu se **vlasnici** (owner), a iz druge klase zovu se **komponente**.



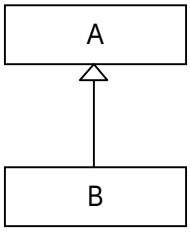
Veze zavisnosti sadrže mnogo veza koje nemaju mnogo veze međusobno. One čine šaroliku grupu veza koje bi se najtačnije definisale iskazom „nisu klijentske veze niti nasleđivanje“ ili pak „ostale veze“. Zajednički im je samo simbol u UML-u. Među njima postoji jedna veza koja se naziva veza korišćenja. Slušbena reč je stereotip i predstavlja se sa <<use>>. Može se vrlo lako definisati i podefiniciji klasa B se pojavljuje bilo kao tip parametara ili tip rezultata metoda iz klase A. Klasa A ne može da se prevede bez klase B.

Nasleđivanje (inheritiance, subclassing)

Nasleđivanje modeluje odnos između dve klase koji je sasvim drugačiji od predhodnih veza. Nasleđivanje ima dva aspekta. Sa jedne strane predstavlja vezu generalizacija-specijalizacija ili opšte-pojedinačno. Na primer veza budilnik–sat.

Po Aristotelu prvi viši pojam u hijerarhiji naziva se genus maxima i ova relacija treba pobliže da objasni taj niži pojam. Veza koja označava odnos generalizacija-specijalizacija naziva se **subordinacija**. Uvedimo vezu P2 sub P1. Ova veza podrazumeva da je sadržaj pojma višeg reda (P1) podskup sadržaja pojma nižeg reda (P2). Skup pojmova koji su subordinirani višem pojmu naziva se **opseg**. Opseg P2 je podskup opsega P1. Nasleđivanje je softverski model odnosa subordinacije među klasama pojmova.

Nasleđivanje je bitno i kao sredstvo kojim se obezbeđuje višekratna upotreba. To se drugi aspekt nasleđivanja. **Tehnička definicija**: Nasleđivanje predstavlja preuzimanje kompletnog sadržaja neke druge klase uz mogućnost dodavanja članova svih vrsta i modifikacije metoda. U UML-u se nasleđivanje prikazuje sa strelicom sa trouglstim vrhom okrenutim od naslednika ka neposrednom pretku (precima). Jedan terminski sistem kaže A je predak a B je potomak. S obzirom da je nasleđivanje tranzitivno klasu koja neposredno nasleđuje zovemo naslednik, a za predak koji je neposredan kažemo da je roditeljska klasa. Drugi terminski siste kaže A je bazna klasa a B je izvedena klasa pa se samo nasleđivanje zove i izvođenje. Treći terminski sistem kaže A je nadklasa, B je podklasa. Četvrti sistem kaže A je klasa davalac, B je klasa primalac... Samo ovo izobilje termina vezanih za nasleđivanje ukazuje na to kolikoje ova veza u stvari značajna.

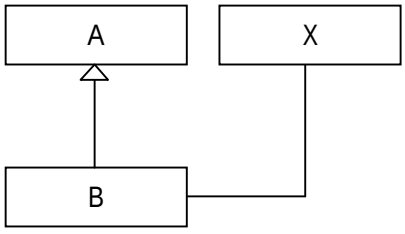


Osobine nasleđivanja, ima ih pet i one su:

1. (tehnička osobina) jedna od ključnih osobina je da izvođenje klase B iz klase A ne zahteva pristup njenom izvornom kodu.
2. (suštinska osobina) je mogućnost dodavanja sadržaja. Ona znači da klasa B preuzima sve što sadrži klasa A s tim da može još sadržaja i da se doda. U klasi B se mogu dodati nova polja ili metode.
3. (tehnička osobina) mogućnost modifikacije metoda. Ona je u stvari modifikacija alogoritma. Isti posao u klasi A i klasi B se ne izvršava istom metodom. Modifikacija metode se zove **redefinisanje** (overriding) jer nema drugog načina da metodu promenime nego da je ponovo napišemo pod istim imenom.
4. **Tranzitivnost** je obavezna osobina. Ako klasa C nasleđuje klasu B, a ona nasleđuje klasu A, onda klasa C nasleđuje klasu A.
5. **Višestruko nasleđivanje** je takvo nasleđivanje kod koga klasa nasleđuje dve ili više klasa. U domenu problema nema preterano mnogo višestrukog nasleđivanja. Npr radio-sat može da nasledi i klasu radio i klasu sat, isto tako ulazno-izlazna datoteka.

Demetrin zakon

Ovo je zakon koji povezuje nasleđivanje i inkapsulaciju. Ukoliko imamo klase A i B i klijenta X i njihove međusobne veze možemo definisati sledećim dijagramom:



Odnos između A i B je mnogo čvršći nego između B i X. Metode klase B imaju pristup public delu klijenta X. Međutim metode iz klase B mogu da pristupaju i private delu iz klase A.

Demetrin zakon: Metode klase ne smeju ni na koji način da zavise od strukture neke druge klase osim neposrednog pretka.

Ovaj zakon ne ide dalje od neposrednog pretka. Demetrin zakon kaže da ne treba pristupati tim podacima.

Nasleđivanje u C++

Sintaksni deo tj sintaksa nasleđivanja mora da bude jednostavna i izgleda: `class B:`α`A.` Alfa ovde predstavlja kontrolu pristupa i može da bude public, private, protected ili ništa. Kod nasleđivanja se na ovaj način reguliše pristup metoda iz klase potomka tj sa biranjem ovih alfa mi regulišemo da li će metode iz klase B imati pristup samo publc delu ili možda protected. Mi u stvari nemamo njen izvorni kod i ostaje pitanje šta će biti sa delom private i delom public iz nje. Postoji i treća vrsta kontrole pristupa a to je **protected** i ona je malo otvorenija od private ali i zatvorenija od public. Njemu imaju pristup sve klase koje nasleđuju tu klasu ali ostale klase mu nemaju pristup dok private delu nema ni jedna druga klasa pristup osim metoda iz same te klase. Nasleđivanje možemo predstaviti na sledećem kodu:

```
class A {
private:
...
protected:
...
public:
...
};

class B:αA {
//dodatni članovi
//redefinisane metode
};
```

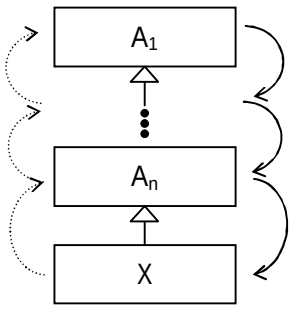
Zbog toga što private delu ni jedna klasa nema pristup dobar programerski običaj je da se umesto private piše protected uvek kada mislimo da će naša klasa poslužiti za izvođenje nekih drugih klasa. Kontrola pristupa delovima klase A, osim kontrolom pristupa u samoj klasi A gde su delovi označeni kao public, private i protected, je određena i sa alfa koje se nalazi u definiciji klase B. Ovo se sve može predstaviti tebelom pristupa:

Nasleđivanje (ispod je alfa)	Član bazne klase	
	public	protected
public	public	protected
protected	protected	protected
private	private	private

Tipična situacija kada ćemo staviti private je kada hoćemo da prosledimo samo interfejs iz klase A u klasu B. Sva ograničenja su jasno definisana. Operator dodele se ne nasleđuje jer može da se desi da je = preklopljen u osnovnoj klasi a u našoj izvedenoj klasi može da se ponaša sasvim drugačije. Zato u B imamo samo osnovni operator dodele mada mi još uvek možemo da u klasi B pozivamo operator = iz klase A. Konstruktor i destruktor se takođe ne nasleđuju, kao i friend funkcije jer to takoreći „prijateljstvo“ određuje sama klasa i u to se niko ne meša.

Redefinisanje metoda. Ukoliko i klasa A i klasa B sadrže metodu m(), prilikom poziva u klasi B metode m biće pozvana baš metoda iz te klase. Ukoliko želimo da pozovemo metodu iz klase A mi je možemo pozvati sa A::m().Ovaj drugi poziv se najverovatnije pojavljuje u samoj definiciji metode u klasi B jer ta metoda verovatno treba da radi isto kao i metoda iz A samo sa malim dodacima koji se dodaju posle poziva metode iz klase A.

Pitanje konstruisanja i destruisanja generalno. Izolovane klase (one koje ništa ne nasleđuju) su veoma retke. U javi npr postoji klasa Object koju sve novo nastale klase nasleđuju ako nije definisano da nasleđuju neku drugu klasu. U C++, u ozbiljnijim programima mi se trudimo da uvedemo 1,2,više hijerarhija nasleđivanja.



U opštem slučaju u objektu X postoji deo koji je stigao iz A_1, A_2... i najzad deo koji smo mi dodali. Princip na kojem funkcionišu konstruktori je da se deo klase nasleđen iz A_i konstruiše baš konstruktorom iz te klase. Ovaj princip je univerzalan. Ttako da prilikom izrade ma koje klase moramo obezbediti pozivanje konstruktora prethodne klasejer je taj koji je pravio tu klasu obezbedio pozivanje prethodnog konstruktora. U konstruktoru klase X se mora obezbediti pozivanje konstruktora iz A_n pa onda tek da se dodaje konstruktor onoga dela članova koji smo mi dodali u toj klasi. Na dijagramu isprekidanom strelicom je označen put pozivanja konstruktora dok je punom linijom označen put njihovog izvršavanja.

Konstruktor bazne klase moramo pozvati eksplicitno. U protivnom može se desiti da će biti pozvan podrazumevani konstruktor a pitanje je da li ta klasa uopšte ima podrazumevani konstruktor i prevodilac onda javlja grešku. U klasi može da se nađe An(){} koji ima ulogu podrazumevanog konstruktora. Veza se mora

uspostaviti samo sa neposrednim pretkom i mora se pozvati samo njegov konstruktor jer se smatra da ko god je pravio tu klasu da je obezbedio u stvari da se poziva konstruktor prethodnika.

Što se tiče destrukcije i ona se izvršava na istom principu. Destruktor date klase destruiše samo deo X koji smo mi dodali, pa onda se poziva destruktor prethodne klase i sve tako dok destruktor bazne klase ne destruiše svoj deo i time je završena destrukcija objekta. Destruisanje i njegovo izvršavanje se vrši obrnuto od konstruisanja.

Inkluzioni polimrfizam

Inkluzioni polimorfizam je vrsta polimorfizma u kojoj se neka promenljiva ponaša kao da menja tip. Inkluzioni polimorfizam i objektnom programiranju se vezuje za nasleđivanje. Objekat klase budilnik u određenim situacijama se ponaša kao objekat klase sat. Inkluzioni polimorfizam ima smisla samo u slučaju kada potomak se posmatra kao predak. Ovaj kontekst se pojavljuje kod operacije dodele, zatim zamene parametra argumentom i kod vraćanja rezultata funkcije. Ovde važi **pravilo pridruživanja**: potomak može da zameni predak a obrnuto ne.

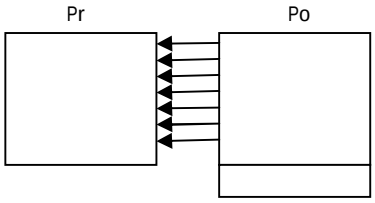
Jedan objekat zbog logike može da pripada samo jednoj klasi ali on može u operaciji da bude tipa klase ili tipa nekog od svojih predaka.

Zakon supstitucije je postavila Barbara Liskov : Ako za svaki objekat s klase S postoji objekat t klase T takav da se proizvoljni program definisan nad T ponaša isto kada se t zameni sa s, tada je S izvedena klasa iz T.

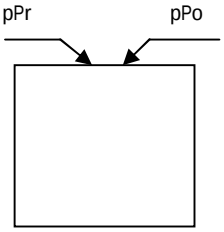
Zamenu pretka potomkom demonstrirаćemo na operaciji dodele i na prenosu argumenata.

```
predak  pr,*pPr;
potomak po,*pPo;

1. pr=po;
   *pPr=*pPo;
2. pPr=pPo;
   (pPr=(predak*)pPo;
   pPr->m();
```

U slučaju pod 1. je direktna dodela. Prvi slučaj podrazumeva dva različita memorijska prostora. Posle izvršene operacije dodele oba objekta se ponašaju kao i pre. Potomak se ponaša polimorfno. Slika sa desne strane.



U slučaju pod 2. (slika desno dole) posle dodele pPr pokazuje na isti objekat. Sada po izvršenju dodele pozivamo metordu m() iz pPr i postavlja se pitanje koja će metoda biti izvršena. U C++ biće pozvana verzija iz pretka.

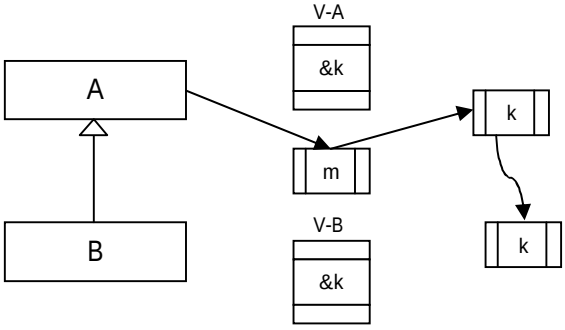



```
void f (Predak);
      f(po);
void g (Predak &);
      g(po);
Predak h(....) {
      Potomak w;
      .....
      return w;
}
```

Inkluzioni polimorfizam pri zameni parametara argumentima.

U slučaju funkcije g parametar će biti prenet po adresi, dok u slučaju prve funkcije ona radi nad objektom konstruisanim na steku. U svim okolnostima predak zadržava ponašanje dok je potomak taj koji ponašanje menja.

Virtuelne funkcije



Kada su one u pitanju imamo sličnu situaciju kao sa prototipom funkcije. To je posebna vrsta metode i njihova prava vrednost dolazi do izražaja u izrazu pPr->m(); Na slici ispod se nalazi geneza virtuelnih metoda.

kada se metoda m() prevede u kod je ugrađen skok. I sada ako neko pravi klasu B i zaključi da mu m odgovara ali hoće da promeni k. I sada ako neko treći piše program i na jednom mestu napiše b.m() (gde je b objekat klase B) on očekuje da se pozove redefinisano k (donje). Desiće se da

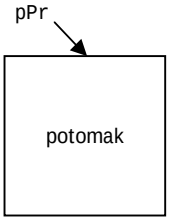
metoda m nije dirana pa se poziva gornje k. Odavde proizilazi da nasleđivanje postaje nebezbedno, a kada je ono nebezbedno objektno programiranje postaje beskorisno.

Rešenje ovog problema može se predstaviti na sledeći način.

Svaka klasa se snabdeva tabelom. Te tabele se zovu V-tabele (VMT – Virtual metod table). U njoj se nalaze adresa virtualne metode k. Te adrese nisu iste za obe klase. Jedna sadrži adresu prve metode k a druga tabela druge (redefinisane). Sada prilikom prevođenja m prvo se pristupa V tabeli klase i nađe se adresa metode k i tek se onda stavlja njena adresa u naredbu skoka. Uvek je obezbeđeno da se pristupi onoj metodi koja je vezana za datu klasu. U tabeli se nalazi i veličina objekta u bajtovima.

Mora se poštovati princip da je verzija metode koja se poziva određena je objektom preko koga se poziva. To ima uticaj na rešavanje problema.

```
pPr->m();
```



Sada metoda m je redefinisana i ona je virtualna. Ovaj poziv metode m na slici sa leve strane, automatski pretražuje V tabelu i daće pravi objekat na koji m pokazuje jer je određena objektom a ne pokazivačem. Sve što treba uraditi da se metoda proglasi virtuelnom je da se napiše:

```
virtual tip ime (parametri);
```

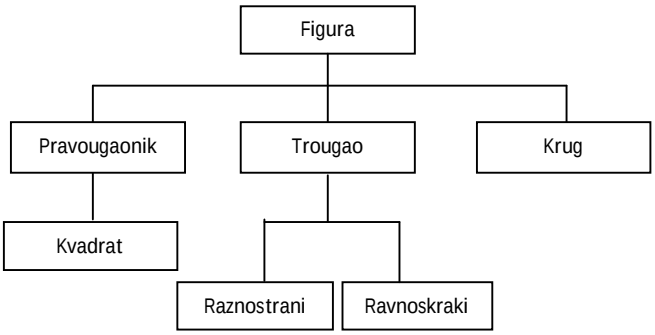
Na primer k mora biti proglašena virtualnom u klasi A. Zato programer mora biti malo vidovit i znati koju metodu proglasti virtuelnom. Ništa ne smeta da i da sve metode proglasimo virtuelnom jer to ništa ne menja. Cena za to je sporost rada. U javi su sve metode virtualne jer tamo brzina nije baš neki presudan faktor pošto je ona sama po sebi spora.

Svaki objekat u svom memorijskom prostoru sadrži adresu V tabele ili barem način kako pristupiti toj adresi. Uglavnom ako se redefiniše metoda ona je virtuelna.

Operacija delete u destruktoru konsultuje destruktor. Na destruktoru je da obezbedi veličinu objekta. Šta ako pokazivač na potomak je tipa pretka pa onda destruktor ne zna šta da uradi. **Ukoliko imamo virtualnih metoda destruktor mora biti virtuelan.**

U trenutku prevođenja ako imamo poziv pPr->m(); mi ne znamo (prevodilac ne zna) na šta pokazuje metoda. Verzija metode se zna u trenutku izvršavanja programa. Ovaj mehanizam ima posebno ime i zove se **late building** ili **dynamic linkage**, **dynamic dispatch**.

Apstraktne metode i klase



E sada ukoliko hoćemo da napravimo metodu za računanju poluobima koja će da prihvati objekte svih klasa. Da bi smo postigli da metoda radi za ma koji objekat napravićemo neku klasu koju će sve klase da nasleđuju. Sve metode poluobim će biti virtuelne.

```
double poluobim (Figura &f) {
      return 0.5*f.obim();
}
```

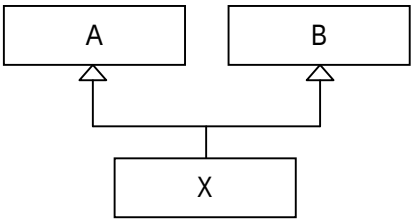
Formula za računanje obima bilo koje figure ne postoji . Šta onda da radimo? C++ ima apstraktne metode. To su metode koje nemaju realizaciju a to se obezbeđuje tako što se posle metode napiše =0. One moraju biti virtuelne. Po definiciji klasa koja ima bar jednu apstraktnu metodu nosi naziv apstraktna klasa. Ona ne može da se instancira. Meyer apstraktne klase naziva nepotpuna klasa. Redefinisanje apstraktne metode se naziva **operacionalizacija** zato što nešto što nije definisano ne možemo ni da redefinišemo. Treba održavati tendenciju da se klase uređuju u hijerarhiju. Time mi smanjujemo entropiju sistema. Klasa u kojoj su sve bitne osobine osobine izvučene iz ostalih klasa po pravilu je apstraktna klasa.

```
Virtual double obim ()=0;
```

Višestruko nasleđivanje

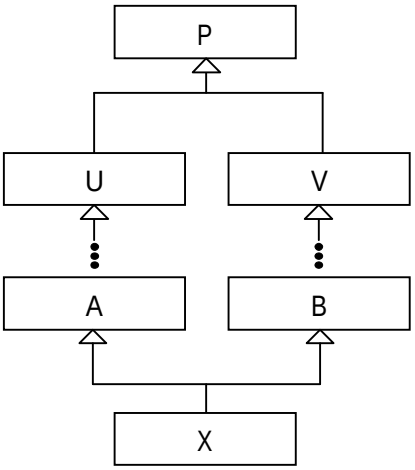
Višestruko nasleđivanje je nasleđivanje od više predaka. Ono podrazumeva da se nasleđuje iz više klasa. Klasa X nasleđuje sve od A i sve od B. Sintaksno se to može zapisati kao:

```
class X:αA,βB {
...
};
```



Najopštije pravilo je da ako X nasleđuje A i B tada se objekat klase X u svakom trenutku ponaša kao instanca A i B. Primer za to je primer radio-sata. On se u svakom trenutku ponaša i kao radio i kao sat. Drugi primer je primer ulazno izlazne datoteke, pa dalje imamo amfibiju... Višestruko nasleđivanje od svih objektnih jezika koji su u upotrebi postoji samo u C++. Java ga nema i tamo mi to simuliramo kompozicijom, ali time mi ne odražavamo inkluzioni polimorfizam.

Suštinski problem je da u domenu problema ne postoji mnogo situacija gde možemo koristiti višestruko nasleđivanje. Pojavljuje se problem i pogrešnog izvođenja (npr pegaz ne možemo izvesti od konja i ptice). Druga vrsta poteškoća su tehničke prirode. Recimo da klasa A ima metodu m i klasa B ima metodu m. C++ prepušta programeru da reši ovu situaciju. On prvo uoči koja mu metoda treba pa on redefiniše metodu m u X pozivajući tu koja mu treba iz prethodnih klasa.



Sledeći problem je **ponovljeno nasleđivanje.** Tu se dešava da klasu P npr nasleđuju dve klase i njih mogu da nasleđuju druge klase ali u glavnom postoje dve grane nasleđivanja koje vode od P. Sada ako se pojavi neka nova klasa X koja nasleđuje po jednog predstavnika iz obe grane dobićemo problem jer X maltene dva puta nasleđuje osobine P. C++ ima neka sredstva i u opštem slučaju rešava ovo virtuelnim nasleđivanjem.

```
class U:virtual public P {
...
};
class V:virtual public P {
...
};
```

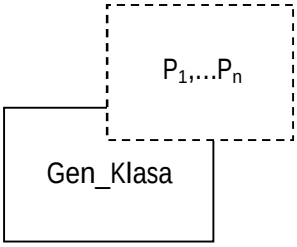
Ovde dolazi do problema da se ovakvo nasleđivanje mora odraditi još na klasama U i V koje mogu biti dosta daleko od klase X koju mi pravimo. I zato ponekad moramo biti pravi proroci što se tiče takve vrste nasleđivanja. Ovo se rešava projektovanjem sistema.

Generičke klase

Osobina generičnosti odnosno generička klasa je klasa koja je parametrizovana drugim klasama. To je parametarski polimorfizam. Određene klase unutar naše klase su označene nekim parametrima. Situacija je slična parametrima u funkcijama. Generička klasa ima parametre za koje se u trenutku definisanja klase ne zna šta su ali se na tom mestu pojavljuju korektne klase prilikom instanciranja. Ti parametri nazivaju se generički parametri a argumenti generički elementi (konkretizovani parametri).

Ovim se domen klase širi. Najčešće se javlja u kontejnerskim klasama (strukture podataka napisane u objektnim jezicima). Kod pravljenja liste javlja se problem tipa elementa liste. Tip može biti slog ili objekat. Informacioni sadržaj je problem tj ne zna se šta se nalazi u elementu liste. To se menja od slučaja do slučaja. U C++ nema intervencija na izvornom kodu kada na primer hoćemo da pretvorimo int listu u double. Generička klasa omogućava da rešimo problem različitog sadržaja elemenata klase. Tip elemenata biće T pa od slučaja do slučaja on će se menjati.

Što se tiče generičnosti, sredstva u C++ su najsavršenija. U C++ sredstvo koje se koristi za ostvarivanje generičnosti naziva se **template** ili **šablon** (posebna vrsta klase). Generička klasa se u UML-u prikazuje:



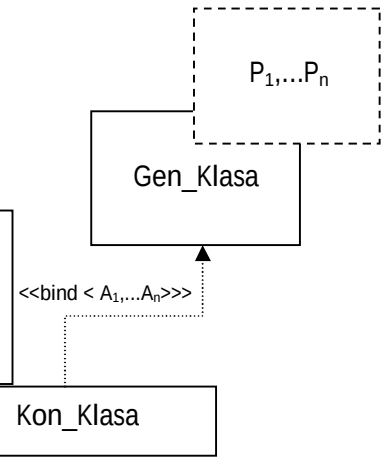
U isprekidanom bloku se piše lista parametara koji se unutar generičke klase koriste kao da su poznati. Klase koje se realizuju na bazi ovog templejta se mogu prikazati na dva načina:

Gen_Klasa <A1,...An>

U prvom načinu navodimo konkretne tipove $A_1,...A_n$. Drugi način je da konkretnu klasu za Gen_Klasu vežemo vezom koja ima stereotip u kome se kaže šta su u stvari argumenti u klasi Kon_Klasa.

Mogućnosti koje pruža C++ za pravljenje generičnih klasa su široke. Ali ima cena koja se plaća.

U zaglavlju klase klasa više ne počinje rečju class već rečju template pa se zatim navode nazivi generičkih parametara pa zatim pišemo reč class.




```
template <class T1,...,class Tn>
class A {
// mozemo koristimo T1...Tn;
};
```

Umesto class možemo da pišemo typename ukoliko želimo da podvučemo da će tu da stoji tip. Ovde je karakteristično da se u klasi A klase T1...Tn slobodno koriste kao da su poznate. C++ omogućuje da imamo dva objekta tipa npr T_1 a,b, i da slobodno napišemo a+b, ali bez garancije da će to biti izvršeno jer to u stvari zavisi od toga da li je nad tim objektima definisana operacija dodele. Cena koja se plaća za ovo je da templejt mora biti prevođen zajedno sa klijentom.

Kada navodimo neku generičku funkciju f čiji se prototip nalazi u klasi A, sintaksa je sledeća:

```
template <class T1,...,class Tn>
tip A< T1...Tn>::f(parametri) {
...
}
```

Indentifikacija generične klase je specifična zato što je ona praćena u stvari listom parametara i to je tek potpuna indentifikacija. Generičke klase mogu biti nasleđene (od strane konkretne tako i generičke klase). Generičnost je takođe retroaktivno preneta u C tako da možemo praviti i generičke slobodne funkcije.

Generička klasa steka

Problem koji se javlja je taj da mi hoćemo uopšten niz mi u [] ne smemo staviti određenu dužinu niza. To znači da moramo obaviti intervenciju u kodu ukoliko mi želimo da naš stek ima od slučaja do slučaja različitu količinu elemenata. Da se to ne bi radilo mi samo definišemo tip konstante i prilikom instanciranja steka mi definišemo njegovu veličinu. Objekti koju su instancirani u mainu sa različitim parametrima nisu međusobno kompaktibilni jer nisu istogtipa. Tako iS i iS1 ne mogu da se kopiraju jedan u drugog jer imaju različit broj elemenata. Da bi se mogućnost da pišemo uopštene tipove i da ga koristimo bez znanja koji je mi u kompletnu generičnu klasu pišemo u zaglavlju zato što se prevodi zajedno sa klijentom.

```
template <class T, int capacity>
class Stack {
private:
    int t;
    T s[capacity];
public:
    Stack() {t=-1;}
    int empty () const { return t<0; }
    int full () const { return t==capacity-1; }
    T top () const { return s[t]; }
    void pop () {t--;}
    void push (T el) {S[++t]=el;}
};
```

```
Stack <int,500> iS;
Stack <char,1000> cS;
Stack <int,2000> iS1;
```

Korektnost i prevencija otkaza. Obrada izuzetaka

Modularnost i pouzdanost su dve glavne osobine softvera. Modularnost po Meyeru predstavlja skup proširivosti i višekratne upotrebe. Pouzdanost se sastoji od dve komponente. Jedna je korektnost a druga je robusnost. Ukoliko se realizacija poklapa sa specifikacijom onda je softver korektan. To znači da softver sam po sebi nije ni korektan ni nekorektan i to je relativan pojam. Tek sa specifikacijom mi možemo videti njegovu korektnost. Druga komponenta je robusnost, odnosno ponašanje softvera u eksploatacionim uslovima. Robustan program se za nekorektne ulazne podatke ponaša izdržljivo tj ne puca.

U klijentu može da dođe do greške (npr kada se korektna metoda pozove na nekorektan način) iako je klasa dobro napisana. Pop praznog steka, push u pun stek, top praznog steka, sve su to greške koje se mogu uvideti prilikom rada sa stekom. Ovakve situacije mogu da se rešavaju na tri načina: havarijskim prekidom programa, korišćenjem reuzltata potprograma kao koda uspešnosti i mehanizmom izuzetaka.

Havarijski prekid programa je takav prekid programa kada više nemamo mogućnosti za oporavak i svaki sledeći korak u izvršavanju donosi velike probleme. U C++ postoje funkcije **exit (broj)**, s tim da je broj različit od nule jer je exit (0) validan kraj programa ekvivalent return 0, kao i **abort** koja takođe poziva exit. Postoji još jedna makro direktiva za havarijski prekid programa a to je **assert (logički izraz);**. Ako je logički izraz netačan (=0) dolazi do havarijskog prekida programa. Na primer možemo naredbu top sada realizovati pomoću ovog izraza:

```
T top () {
    assert (t>-1);
    return s[t];
}
```

Ova funkcija je sada zaštićena u smislu da neće doći do nepravilnog očitavanja steka. Nedostatak ovoga je da akcija koja se preduzima prilikom incidenta se ne nalazi na mestu incidenta tj ne nalazi se u klijentu već u klasi. Npr kod interaktivnog programa nam ne dopušta da se rati pređašnje stanje (quo ante).

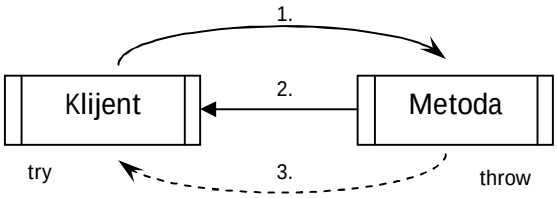
Korišćenje rezultata potprograma kao koda uspešnosti. Za izvođenje mogućnosti da klijent unosi ponovo neku vrednost koristi se rezultat koji vraća potprogram. Ideja je da se vraćeni rezultat metode koristi kao indeks uspešnosti. Kada vrati neku karakterističnu vrednost klijent zna da je sve u redu.

```
enum ErrorCode {ok,underflow,overflow};

ErrorCode pop () {
    if (t<0) return underflow;
    t--;
    return ok;
}
ErrorCode push (T el) {
    if (t==capacity-1) return overflow;
    s[++t]=el;
    return ok;
}
ErrorCode top (T &rez) {
    if (t<0) return underflow;
    rez=s[t];
    return ok;
}
```

Kod enumeracije možemo samo tag koristiti kao da je definsan sa typedef. Ovako se intenzivno programira obrada greške i sasvim lepo radi. Kakva je reakcija na grešku to zavisi od klijenta jer on sada vidi kakva je greška.

Sve bi bilo lepo da nema funkcija koje vraćaju rezultat. Ovde moramo vraćati vrednost preko liste parametara. Ovo ni malo ne valja jer moramo top pozivati na sasvim drugi način. Tako se top pre pozivao sa s.top() dok ga sada moramo pozivati na primer sa s.top(x) gde je x vrednost preko koje vraćamo rezultat. Ovo nam uvodi ograničenja jer se ovakve funkcije ponašaju kao void funkcije. U nekim slučajevima kod nekih funkcija (koje vraćaju vrednost) izlaz iz funkcije mora da zadovolji određene uslove. Onda odaberemo neku „nelegalnu“ vrednost za vrednost greške. Na primer kada se vraća pokazivač, funkcije vraćaju NULL pokazivač za grešku. fopen kada vrati NULL znači da otvaranje datoteke nije uspelo.

Mehanizam rukovanja izuzecima. Zbog prethodnih problema uvodi se mehanizam rukovanja izuzecima. **Izuzetak** (exception) je namerno izazvan događaj čija je svrha predupređivanje otkaza i koji ostavlja program u zatečenom stanju (status quo ante). Obavezno je praćen prijavom nastanka. Incident je vidljiva manifestacija otkaza. Sistem za rukovanje izuzecima (exception handling) je upravo zasnovan na vidljivoj manifestaciji incidenta. Izuzetak je namerno izazvan incident s namerom sprečavanja pojave incidenta jer je izuzetak u stvari kontrolisan. Čak ako klijent ne želi da reaguje na izuzetak dolazi do havarijskog prekida programa, tj jednom rečju preko izuzetka se jednostavno ne može preći ili drugim rečima on ne može ostati neprimećen. Izuzetak u sebi objedinjava prethodne dve vrste rukovanja greškama. Izuzeci se ne mogu simulirati zbog postojanja trećeg kanala za razmenu podataka između metode i klijenta. Prvi kanal služi za slanje podataka u metodu, preko drugog kanala se vraća rezultat dok treći kana l služi za slanje poruka o izuzecima. Taj treći kanal se ne vidi i prva dva ostaju slobodna za standardne prenose pa se tim rešava problem vraćanja rezultata.



C++ ima dva načina za rukovanje izuzecima, u zavisnosti kako se modeluju izuzeci, tj izuzeci se mogu softverski realizovati kao objekti ili kao vrednosti. Izuzeci mogu da se koriste kao vrednosti kada npr radimo procedurne programe.

```
try {
    //rizični poziv
} catch (kvaziparametar) {
    //reakcija
} catch (kvaziparametar) {
    //reakcija
} catch (kvaziparametar) {
    //reakcija
}
```

Generisanje izuzetka i njegovo prosleđivanje klijentu se vrši naredbom **throw**. Postoje dva tipa naredbe, za svaki od tipova izuzetaka po jedan, i to su: throw izraz (najčešće tipa enumeracije), i drugi throw poziv_konstruktora (jer je izuzetak objekat koji je deo klase). Prihvatanje i obrada izuzetaka se vrši posebnom naredbom **try**. U okviru try se nalaze naredbe C++ među kojima moraju da se nađu oni rizični pozivi koji mogu da izazovu prekid. U sklopu try bloka nalaze se rukovaoci pozivima (exception handlers). Svi nose naziv **catch** a u zagradi se nalaze kvaziparametri jer tu može da stoji pravi parametar ili samo tip parametra. Potom slede naredbe programskog jezika C++. Tu treba da se prepozna reakcija na kvaziparametar. Može biti više catch naredbi i stavljaju se jedna iza druge. Deo koda u try se normalno izvršava. Bilo gde da se naiđe na izuzetak izvršavanje koda klijenta se

prekida i prelazi se na deo sa rukovaocima izuzetaka. Ispituje se koji od rukovaoca može da prepozna tip izuzetka i pristupa se prvom na koji se naiđe a da može da primi taj izuzetak.

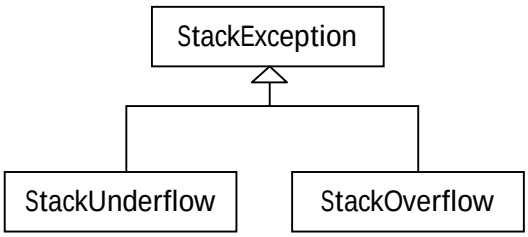
Rukovanje izuzecima kao vrednostima. U ovom slučaju izuzetak je obično enumeracija jer sa tim obezbeđujemo da znamo u čemu je greška. Kada se generiše izuzetak funkcija ne mora da vraća vrednost. Kod koji može da sadrži rizične izuzetke se nalazi u try bloku pa kada se naiđe na izuzetak pozove se rukovaoc izuzecima. Među svim rukovaocima postoji jedan poseban koji se zove **univerzalni rukovaoc izuzecima**. On je posebnavrsta rukovaoca koji prihvata sve izuzetke. On se stavlja na kraj tj posle svih ostalih rukovaoca.

```
enum ErrorCode {underflow,overflow};

T top () {
    if (t<0) throw underflow;
    return s[t];
}

void push (T el) {
    if (t==capacity-1) throw overflow;
    s[++t]=el;
}
```

U klijentu se nalazi sledeće:

```
try {
    .....
} catch (ErrorCode err) {
    switch (err) {
        case underflow: cout<<"Stack
underflow"<<endl; break;
        case overflow: cout<<"Stack
overflow"<<endl;
    }
}
```

Izuzetak kao objekat. Programer formira sopstvene klase sa izuzecima. Npr mogli bi da napravimo jednu klasu za underflow a drugu za overflow. Ovde je zgodno to što reakciju možemo da ugradimo u klasu. Kada su neke klase izvedene, kod rukovaoca, rukovaoc potomkom mora da se nađe pre rukovaoca pretka. Ovde ćemo ove dve klase izvesti iz abstraktne klase StackExeption. Sve će biti isto osim što ćemo umesto dve imati samo jednu catch naredbu.

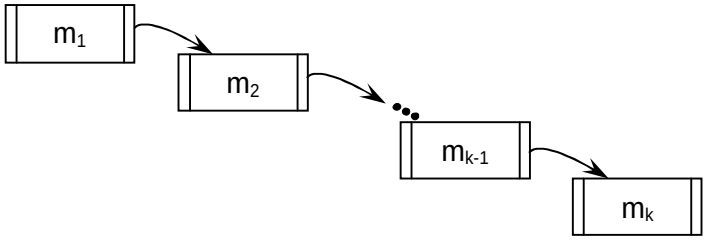


```
class StackException {
public:
    virtual void action()=0;
};
class StackUnderflow:public StackException {
public:
    void action () {
        cout<<"Stack underflow"<<endl;
    }
};
class StackOverflow:public StackException {
public:
    void action () {
        cout<<"Stack Overflow"<<endl;
    }
};
```

U klijentu se nalazi sledeće:

```
try {
    //Rizične naredbe
} catch (StackException &ex) {
    ex.action ();
}
```

Propagacija izuzetaka



Ukoliko dođe do izuzetka u m_k njen rad je prekinut. Izuzetak se pojavljuje i u m_{k-1}. m_{k-1} (i svaka prethodna) može da postupi na tri načina.

Kao prvi načinona može da ignoriše izuzetak. Princip je da izuzetak ne sme da prođe neprimećen. Ovde on ne prolazi neprimećen jer bi sva odgovornost bila samo na klijentu, već samo m_k prosledi izuzetak dalje. Drugi način je da se propagacija zaustavlja kod naredbe try gde se izuzetak obrađuje i onda se rad zaustavlja. Treći način je kombinacija prva dva načina. Neka neka funkcija ima try obradu izuzetka ali samo ga delimično obradi i prosledi ga dalje.

Programski jezik Java

Java je napravljena prvi put 1992. Godine pod imenom OAK. 19953 se pojavljuje pod sadašnjim imenom. Z arazliku od C++ nju je radio čitav tim. Ima i svojih mana ali i prednosti jer kada je C++ pravljen metodologija objektnog programiranja je bila u povoju dok je za vreme pravljenja jave ova metodologija bila već razvijena.

Java je dobila ime po vrsti kafe koji su pili autori. Da bi neki programski jezik uspeo na tržištu on mora ponuditi nešto što drrugi programski jezici ne nude. Na primer pascal je ponudio razumljivost koda, C je nudio male i razumljive programe, C++ je opšteprihvaćen zato što je proširenje uspešnog C-a, dok java nudi veoma elegantno rešenje portabilnosti ili prenosivosti programa. Ovaj problem je star koliko i samo programiranje i javlja se prilikom prenošenja programa sa računara na računar. Viši programi su obezbedili prenos programa ali u izvornom kodu a ne na nivou mašinskog koda. Na drugom računaru taj kod se morao prevesti. Problem se javlja kada razni računari imaju razne mašinske jezike i tada se ne može preneti program. Rešenje je napraviti virtualnu mašinu koja glumi računar. Taj program se pojavljuje kao interfejs između našeg programa i računara.

Ta mašina se zove Java Virtual Machine i nema potrebe da simulira računar već samo neke procese. Ova portabilnost nosi svoju cenu u brzini programa. I to ima najveći uticaj na razmišljanje u javi. Jedan od glavnih zaokreta je taj da se memory leak zanemaruje tj nema destruktora. U toku izvršavanja programa u javi može se izvući deo mašinskog koda. Java je dobila na značaju pojavom interneta.

Počinjemo sa konstatovanjem čega to u javi ima a u C++ nema i obrnuto. Najvidljivija promena je način rukovanja objektima. Java je čist objektni jezik dok je C++ hibridni jezik. U C++ se promenljivim i objektima može rukovati preko pokazivača dok se u javi isključivo rukuje preko adrese. Objektima se u javi rukuje preko reference koja nema mnogo dodira sa referencom u C++ već je bliža njegovim pokazivačima. Ta ideja je preneta iz Deflija i takođe još mnogo stvari je u javi rehabilitovano iz objektnog paskala. Od operacije nad referencama ostala je operacija new, a delete više ne postoji. Pošto je java čist objektni jezik izvedenih tipova više nema već susvi oni klase (u suštini specijalne klase). Takođe ne postoji naredba typedef, kao ni mogućnost preklapanja operatora jer je ono više usmereno procedurnom jeziku negoli objektnom. Nema predprocesora ni predprocesorskih direktiva, nema automatske konverzije tipa u manje složeni tip, ali takođe nema ni slobodnih funkcija. Formalno nema slobodnih potprograma, ali u suštini oni postoje samo su realizovani na malo drugačiji način. Višestruko nasleđivanje ne postoji i može se simulirati kompozicijom ali ne u potpunosti.

Ne postoje destruktori u javi već se jednostavno sa objekta skine referenca i on postaje nedostupan. Takođe ne postoje ni generične klase. Kod definisanja baznih tipova rehabilitovan je paskalski način i tipovi ovde imaju tačno ime pez mnoštva raznih modifikatora. To ne znači da modifikatori ne postoje, naprotiv sistem modifikatora u javi je veoma razvijen. Naredba goto je u potpunosti izbačena, i to po prvi put u istoriji. Ona je civilizovana time što je naredba break pojačana. Način modularizacije je takođe promenjen i u javi je on veoma razvijen. Svaka klasa je za sebe modul i uvedena su dva nivoa modularizacije: klase i paketi.

Uveden je logički tip pa se selekcije i ciklusi realizuju pomoću ovoga tipa. Znakovni tip je 16-bitni. Metode sada mogu da vraćaju reference. U javi postoji konkurentno programiranje. Pitanje gotovog sofvera je izraženo u javi i ne postoje biblioteke kao što je to slučaj u C++ već su sve klase, mada java ima biblioteke gotovih klasa. Sve je uređeno u jedinstvenu hijerarhiju. Osnovna klasa u toj hijerarhiji je klasa Object i iz nje se sve klase izvode ako nie definisana neka druga klasa, ali i tada preko te druge klase se vežu za klasu Object.

Virtuelna memorija

Ova memorija se ne ponaša kao stvarna memorija. Ona je svedena na svega dve vrste memorija i to po ugledu na objektni paskal. S obzirom da u našem programu nema ničega drugog osim objekata i promenljivih, virtuelna memorija je podeljena na stek i hip. Sve skalarne promenljive nalaze se na steku pa shodno tome i sve reference su na steku. Svi objekti se nalaze na hipu. Sam objekat nema ime već kada mu ga dodelimo mi u stvari napravimo referencu na taj objekat. Ako referenca na objekat ne postoji objekat nije dostupan. Ako objekat ima podobjekat, unutar našeg objekta se nalazi referenca na taj objekat član.

Elementi programskog jezika Java

Kao i svaki drugi alfabet, alfabet Jave je podeljen na tri vrste simbola:

1. mala, velika slova i _ (underscore)
2. cifre 0..9
3. specijalni znaci + - = * >>> itd.

Java ima slobodan format što nalaže da postoji rečnik. S teorijske strane ovde nema ništa novo. S praktične strane uvedene su novosti kod indentifikatora. Samo slova i cifre ulaze u ime i prvo mora biti slovo. Novost jestu ona neformalna pravila (camell notacija, i da ime klase počinje velikim slovom) kod C++ koja su ovde obavezna. Osim ove dve novosti postoji i treća a to je da se imena paketa obavezno pišu malim slovima.

Prefiksi pokazuju šta neke metode rade (get, set, is koji se koristi za realizaciju bulovih funkcija koje proveravaju karakteristične osobine nekih objekata).

U javi imaju tri vrste komentara. Standardni /* */ i // ali i još jedan komentar koji nam olakšava kodiranje i vezan je uz izradu dokumentacije. Dokumentacija je najslabija tačka u softveru jer je najteži deo za uraditi. Ovde se dokumentacija klase pravi paralelno sa klasom i ona je deo koda, a sredstvo za njeno takvo definisanje je treća vrsta komentara /** */. Ovi komentari su programabilni. Unose se direktive u komentar radi lakšeg kreiranja dokumentacije. Počinju sa @ npr @param ime opis. Kada se završi izvorni kod on se prevodi prevodiocem a ovi komentari se prevode posebnim prevodiocem koji se zove javadoc. Ti helpovi se mogu objediniti i može se od njih napraviti dokumentacija.

Bazni ili primitivni tipovi

Ovo su jedini tipovi koji postoje u javi. Java se ubraja u jako tipizirane jezike. Ovde je bulov tip rehabilitizovan što utiče na tipiziranost. Bazni tipovi su:

1. **boolean** – i on je nekompaktibilan sa ostalim tipovima
2. **celobrojni tipovi** – int, long, short, byte i ovde nema modifikatora kao što su signed, unsigned...
3. **realni tipovi** – double, float
4. **znakovni tip** – char i on u javi se oslanja na unicode i zauzima 2 bajta a ne 1

Reference na objekte se ponašaju kao skalarne promenljive ali su zato tipa klase. String je u javi klasa koja je već ugrađena. null je konstanta tipa reference i označava da referenca ne pokazuje ni na jedan objekat. Logički tip je boolean. Dužina mu je jedan bit a konstante su uzete iz paskala i glase true i false. Operacije se u javi prepoznaju prema operandima a ne prema simbolu operacije.

Celobrojni tipovi su sledeći: byte koji ima veličinu 8 bita, short 16 bita, int je 32-bitni a long 64 bita. Svi tipovi su signed. Postoje i sufiksi kada hoćemo da podvučemo da je konstanta nekog tipa. Npr 142L.

Realni tipovi su float i double i oni su u javi strogo standardizovani i strogo definisani. Float ima 32 bita i raspon mu je od $3.4*10^{-38}$ do $3.4*10^{38}$, a double je 64-bitni i raspon mu je od $1.7*10^{-308}$ do $1.7*10^{308}$.

Neke novosti su vezane za tip char. U javi je njegova veličina 16 bita i on je kompaktibilan sa ostalim tipovima. Za upravljačke simbole postoje neki koji ne mogu da se predtave npr: \ \“ \’ i postoje eskejp sekvence \n \t \r \f \b ... Svi unicode mogu da se predstave kao \ooo gde je o –oktalna cifra, ili kao \uxxxx.

Konstantan string zove se **literal**. Referenca u javi je adresa. Promenljive se pojavljuju u dva slučaja: ili kao polje u klasi ili kao lokalna promenljiva. Promenljiva se deklariše naredbom koja je u potpunosti ista kao i u C++ i ona se može opciono inicijalizovati: tip ime[=vrednost] ili na primeru int i,j=0,k. Promenljiva se može deklarisati bilo gde. Na sve što utiče mesto deklarisanja jeste domet te promenljive. Promenljiva se može deklarisati i u izrazima:

```
double h=Math.sqrt(k+k1);
```

Java nema biblioteke slobodnih funkcija jer je sve u klasama (tj slobodne funkcije ne postoje). Ovde je Math u stvari klasa a sqrt() je statička metoda koja se poziva preko klase.

Domet promenljive

U slučaju kada je promenljiva u klasi domen je isti kao i u C++. U slučaju metode sopstvene klase ovo polje je u potpunosti dostupno. U slučaju metoda drugih klasa i ovde postoji mogućnost da polje bude tipa public, private ili protected ali postoji i četvrti nivo a to je default ili friendly koji razrešava pitanje klasa u istom modulu. Ovaj nivo nema službenu reč.

U slučaju kada je promenljiva lokalna, u osnovi pravilo je isto kao i u C++. Promenljiva je upotrebljiva od mesta njenog deklarisanja pa do kraja tog segmenta u kome se nalazi. Lokalne promenljive se pojavljuju u bloku. Blok je u stvari sekvenca snabdevena deklaracijama. U C-u sekvenca je {naredbe+deklaracije}, dok je ovo u javi blok.

Primer programa na Javi

```
Krug.java
public class Krug {
    private double r;
    public Krug ( double r ) {
        this.r=r;
    }
    public double getR () {
        return r;
    }
    public double obim () {
        return 2*r*Math.PI;
    }
    public double povrsina () {
        return r*r*Math.PI;
    }
}

TestKrug.java
public class TestKrug {
    public static void main (string args[]) {
        Krug kr=new Krug(1);
        System.out.println("Poluprečnik: "+kr.poluprečnik());
        System.out.println("Obim: "+kr.obim());
        System.out.println("Površina: "+kr.povrsina());
    }
}
```

Ovde klasa ima osobinu modularnosti pa je moramo smestiti u istoimenu datoteku. Takođe i paket ima osobinu modularnosti. Datoteka se zove kao i klasa. Kompletan sadržaj klase se nalazi u klasi. Metode se takođe definišu unutar klase jer je ovde klasa drugačije definisana. Nema segmenta public i private već se pre svake metode i svakog polja navodi njen nivo zaštite. Ovde se te službene reči pojavljuju u ulozi modifikatora čija je uloga u javi mnogo pojačana. U javi se polje i parameti mogu isto zvati pa čak i metoda se može zvati isto npr svo troje se može zvati r (poluprečnik, argument u metodi, pa i sama metoda r()). Pošto java nema slobodnih funkcija pa nema ni glavni program, pa je on ovde u suštini klasa u kojoj se nalazi statička metoda. Takođe ova statička metoda može da se nalazi u bilo kojoj klasi pa i u našoj čime se obezbeđuje da klasa bude izvršiva. Ako pozovemo klasu ona će biti aktivirana.

Procedurni deo Jave

U velikoj meri ovaj deo je izvučen iz C-a. Najveća izmena je postojanje bulovog tipa.

Aritmetički operatori. Imamo unarne: +, -, ++, --, pa onda ostali +, -, *, /, %, =, +=, -=, *=, /=, %=.

Bit operatori. Ovim operatorima se rukuje kao u paskalu. Ovde je primenjen princip da operatori prepoznaju tip operanda. Ovi operatori su: ~, &, |, ^, >>, >>>, <<, &=, |=, ^=, >>=, >>>=, <<=.

Relacioni operatori. ==, <, >, !=, >=, <=. Pošto je uveden boolean tip više se ne može pisati 2*(x==y).

Logički operatori. !, &, |, ^, &=, |=, ^=, ==, !=. Postoje i optimizovani logički operatori konjukcije i disjunkcije && i ||, da bi oslikali ponašanje ovih operatora u C-u.

Osnovna dodela. Lvrednost=izraz.

Uslovni izraz. Je kontrolisan logičkim tipom log_izraz ? izraz1 : izraz2.

Konverzija. Logički tip je boolean i on je nekonvertibilan. Konverzija može biti implicitna ili eksplicitna. Typecast je primenljiv na reference i pojavljuje se kod nasleđivanja. Prvo se radi promocija (byte i short se odmah konvertuju u int) zatim se i sa ostalima postupa na isti način. **char->int->long->float->double**. Takođe je bitno da se zna i char+char->int. U javi je zabranjeno dodeljivanje složenijeg manje složenom tipu npr long->int. Typecast je isti kao u C++ npr (int)q.

Pošto je u javi sve klasa tako su i nizove klase. Specijalna je po tome što ima ugrađenu operaciju indeksiranja. Sintaksno se malo drugačije definiše nego ostali objekti jer mora da se zada dužina. Osim indeksiranja i sintakse ne razlikuje se od ostalih klasa. Niz se nalazi na hipu.

Jednodimenzionalni nizovi

Definišu se na sledeći način: `tip ime []`, ali može na još jedan način kao `tip [] ime`. Prvi oblik je uveden po tradiciji a drugi je deo novijeg stila i logičniji je. Npr int [] x po logici gledajući uveli smo promenljivu x koja je niz integera. Ono je takođe uvedeno jer metode kao rezultat mogu da vraćaju niz jer je x skalar, referenca na int niz koji još ne postoji.da bi se došlo do tog niza kao i kod svakog drugog objekta treba ga napraviti na hipu operacijom new. To moramo izvesti ime=new tip [dužina]. U našem slučaju npr x=new int [5]. Time je napravljen int niz dužine 5. Raspon indeksa je kao u C-u od 0 do dužina – 1. Za razliku od C-a java proverava proboj indeksa.

U javi se prvi put niz posmatra kao zaokružena struktura podataka. Mi u C++ posmatramo niz kao zauzeti memorijski prostor npr od 100 elemenata a da li ćemo mi taj celi prostor iskoristiti to nam nije poznato. U javi su kod niza svi elementi iskorišteni. Ili nema ni jedan vrednost ili svi imaju vrednost. Niz se posmatra kao niz elemenata u pravom smislu reči. Uzimajući u obzir da je niz objekat u svakom smislu reči postoji polje lenght koje pokazuje broj elemenata. x.lenght==5. Niz se može inicijalizovati i kao y[]={1.5,-3.4,0.8}. Ovde je new pozivano automatski.

Višedimenzionalni nizovi

```
int dvaD = new int [2][];
dvaD[0] = new int [3];
dvaD[1] = new int [4];
```

Mogu se deklarisati kao `tip ime [][]` ili kao `tip [][] ime`. Ovde se novosti mogu uočiti kod rukovanja matricom jer se rukuje sa referencom. Na hipu se posebno pojavljuju redovi matrice kao zasebni nizovi. Pa se matrica može definisati kao na slici desno.

String

String se u C-u realizuje kao specijalni niz. Kod nizova je bitno indeksiranje a kod stringova je još bitna i operacija dodavanja stringova ili konkatenacija. U C-u znakovni niz je loša realizacija stringa.

String je u svakom jeziku univerzalan nosač podataka. Npr to se može videti kod argumetana funkcije main. String kao univerzalan nosač je veoma zastupljen i kao nosač teksta. U javi je string klasa.

Postoje sting konstante kao na primer String s="ovo je konstantan string".

Najvažnije svojstvo stringova u javi je njihova konkatenacija i označava se kao operator +. Ukoliko imamo dva stringa i napišemo s1+=s2 prvi string će biti napušten otvoriće se novi objekat u koga će biti kopiran s1 i dodat s2 i sada će s1 pokazivati na njega dok će stari objekat biti napušten odnosno nedostupan. StringBuffer klasa nam dozvoljava da imamo string koji je promenljive dužine. Konkatenacija je polimorfna operacija. Objekat koji se dodaje, da bi se dodao, mora imati funkciju toString().

Objekat kada je parametar on se prenosi po adresi tj on se nužno prenosi preko adrese. Kada objekat prenesemo po adresi tada ga mi možemo menjati. Ako string prenesemo po adresi i menjamo ga on će opet zadržati svoje početno stanje na kraju funkcije.

String ima mnogo konstruktora i mnogo metoda. Ima metodu length() koja vraća dužinu stringa, ima toString() koja nije u okviru klase, pa u okviru klase ima još i charAt(int i).

Naredbe

Naredbe se dele na proste naredbe i upravljačke strukture.

Prosta naredba je izraz praćen sa ; npr izraz;. Taj izraz mogu biti konstantne metode, pa čakmože biti i prazna naredba mada se ona u javi veoma često koristi.

Upravljačke strukture. Među njima se pojavljuju standardne blok strukture (sekvenca), ciklusi, selekcije i naredbe skoka.

Sekvence. U ulozi sekvence u javi se pojavljuje blok. Blok su naredbe poređane jedna pored druge zajedno sa deklaracijom {naredbe+deklaracije}. Blok se može labelirati mada se to retko čini.

Selekcije. Nema novosti u odnosu na C++ osim toga da su one sada kontolisane pravim logičkim izrazima, zato što je uveden bulov tip. Switch i if se isto pišu kao i u C++ osim malopre navedene razlike.

```
while (logički_izraz) naredba
do naredba while (logički_izraz)
```

Ciklusi. Ciklusi su takođe kontrolisani logičkim izrazima. Takoće naredba for se izmenila pošto je njen deo kojim se proverava uslov sada takođe smatran logičkim izrazom. Još jedna razlika jeste inicijalizacija. Pošto nema niza izraza ne mogu se osim u foru inicijalizovati više od jedne promenljive. Java dozvoljava da se kontrolna promenljiva definiše unutar fora.

Skokovi. Naredbe goto nema nikako. Break postoji i najviše se koristi u switch ili unutar nekog ciklusa i izaziva njegov momentalni prekid. Pošto nema goto kada želimo da iskočimo iz nekog ugneždenog ciklusa, sada koristimo break koji je ojačan i sada može da sadrži labelu na koju skok treba da se izvrši. Postoji i naredba continue i ona prelazi na sledeću iteraciju ciklusa. Dodatna mogućnost je da se i ona labelira. Return se koristi isto kao i u C++.

Klase u Javi

U javi se klasa definiše na sledeći način:

```
class ImeKlase {
    //polja
    //metode
}
```

U javi se sve nalazi u klasi. U C++ i Pascalu to nije tako jer je tamo struktura modula dvodelna (interfejs tj zaglavlje i telo modula). Realizacija funkcija, metoda je nedostupna pa se u C++ one nalaze u drugom fajlu. Javin modul zove se paket i njegova struktura je drugačija. U javi ne govorimo o objektima članovima i podacima članovima jer nema neke razlike pa ih jednostavno zovemo jednim imenom polja. Ona se definišu kao `[modifikatori] tip ime [=inicijalizator];` Sistem modifikatora je veoma zastupljen u javi. Npr uz svako polje ide kontrola pristupa. Inicijalizacija je mehanizam za dodelu vrednosti i kod objekata članova mora da stoji new.

Kada su u pitanju polja tada će svako polje biti inicijalizovano. Ako je polje bulovog tipa biće inicijalizovano na false, ako je znakovnog tipa na null, a za ostale tipove na 0. U javi jednom rečju ne postoji polje koje nema stanje (početnu vrednost). Ponekad je najbolje inicijalizaciju ostaviti za konstruktor.

Metode imaju opšti oblik `[modifikatori] tip ime (parametri) { ... }`. Java dozvoljava da se metode zovu kao i polja nad kojim radi (npr malopre smo imali r i r() umesto getR()). U C-u to ne može jer ako se pojavi r to može značiti polje ali i pokazivač na početak funkcije. Metoda u javi ima sve osobine potprograma. Parametri se prenose ili po vrednosti ili po adresi. U javi nema mnogo izbora. Bazni tipovi se isključivo prenose po vrednosti a objekti isključivo po adresi. To znači da metoda radi isključivo sa

```
double [] zbirNizova (double x[], final double y[]) {
    double [] zbir=new double [x.length];
    for (int i=0;i<x.length;i++) zbir[i]=x[i]+y[i];
    return zbir;
}
```

originalom. Još jedna od novina je pitanje rezultata. U javi rezultat metode može da bude niz jer je niz objekat pa metoda vraća referencu na niz. Metoda koja sabira nizove:

Ukoliko u funkciji promenimo x promeniće se x i na hipu. Ako hoćemo da se odbranimo od toga umesto const što smo stavljali u C++ ovde stavljamo final. Ovo ne znači ništa jer java nema konstantnih metoda pa onda ako napišemo x.m() ne zna prevodilac da li je metoda promenila stanje objekta ili ne.

Konstruktori

Kada su u pitanju konstruktori sve je isto kao i u C++. Svi nose ime klase, moraju se razlikovati bar po tipu parametara. Postoje ugrađeni i podrazumevani konstruktor. Polje this postoji i u javi ali je ovde ono adresa pa nema dereferenciranja.

Destruktori

U javi ih nema jer se curenje memorije ovde dozvoljava. Ovde su svi objekti na hipu. U javi postoju background proces koji se zove garbage collector (GC). On se uključuje i sve objekte na koje ne postoji referenca uništi. Tako objekt koji nećemo koristiti samo njegovu referencu postavimo na null i rešen problem.

Za ozbiljnije aplikacije gde je potrebna brzina i gde se zauzima veliki deo memorije GC nije baš najbolje rešenje. Jer GC može da se uključi u kritičnom trenutku i prekine nam rad programa. Metoda finalize() uključuje GC i ona se preklapa u klasama ali opet ne znamo kada će i da li će GC biti uključen.

Smeštanje klase

Svaka klasa se smešta u posebnu datoteku. Može da se desi da ih ima više u jednoj datoteci s tim da je samo glavna public. Datoteka se zove isto kao i klasa uključujući i mala i velika slova. S ovim je postignuto da je klasa u stvari moduo. Neupravljivost velikim brojem datoteka ovde je rešena jer se klase objedinjavaju u nove celine.

Instanciranje klase. Svaka klasa se instancira preko new. Uglavnom treba odraditi 3 posla. Treba napraviti referencu, novi objekat na hipu i povezati ih. Sve ovo može da se uradi u jednoj ili dve naredbe. X objX=new X(..). Ovakvo instanciranje vraća kao rezultat referencu na objekat pa možemo ovo smestiti i kao argument neke funkcije.

Dodela i upoređivanje

Ukoliko imamo dva objekta recimo iste klase obj1 i obj2 i napišemo obj1=obj2 mi smo u stvari napravili izjednačavanje referenci (kao dodela pokazivača u C++). Sada će obj1 da pokazuje na objekat koji je pokazivala referenca obj2 tj neće biti stvoren novi objekat. Stvaranje novog objekata se zove kloniranje. Isti slučaj je i za obj1==obj2. Ovde se upoređuju reference a ne sadržaj objekata.

Statički članovi klasa

Statički elementi klase (postoje i u C++) se ne odnose na objekat nego i na klasu. U javi se intenzivno koriste. Oni mogu biti polja, metode (naročito važni) i mogu čak da budu i klase. Oni se odlikuju time da za njihovo korištenje ne treba imati objekat klase. Statička polja se definišu sa `static tip ime [=inicijalizator]`. Ova polja se mogu inicijalizovati na dva načina:

```
static double a=5,b,c;
static {
      b=a*Math.log(a);
      c=a*b;
}
```

Za polja objektnog tipa inicijalizacija se obavlja sa new. Statičke metode se definišu na isti način stavljanjem modifikatora static ispred imena metode. Statička metoda ne može da koristi nestatička polja jer se ne zna kojem se objektu pristupa. Takođe ne sme da se koristi nestatičke metode.

Prividni izuzetak su konstruktori. Njih mogu da pozivaju jer je u suštini metoda nivoa klase pa može da se koristi. Statičke metode se pozvaju preko klase ali mogu da se pozivaju i preko objekata ali onda gube svoj smisao. Redosled inicijalizacije je prvo statički elementi pa nestatički. Inicijalizacija će biti izvršena prilikom prvog obraćanja metodi ili pri instanciranju klase. Statičke metode u javi jesu način realizacije slobodnih funkcija.

Imenovane konstante. U javi nema globalnih promenljivih već je sve u sklopu klase. Konstante se definišu nas ledeći način `final tip ime=vrednost`. Ali ovo ređe koristimo većinom je bitnije da su one statička polja pa se piše `static final tip ime=vrednost`. Konstante bi trebalo da se pišu velikim slovima. System.out.print () ovo out je takoće statički član tipa objekta.

Klasa Object

Java se intenzivno oslanja na hijerarhiju klasa. Svaka klasa automatski se uključuje u tu hijerarhiju i njen koren se zove object. U toj klasi postoji i desetak metoda. Zbog inkluzionog polimorfizma. Ako je parametar tipa objekt argument može biti bilo koji objekat i to je jedan problem ovakvog uređenja. Neke od metodi u klasi Object su:

- Object clone () koristimo je za operaciju kloniranja. To je primena operacije dodele
- boolean equals (Object obj) služi kao i clone samo u proveri jednakosti
- void finalize ()
- String toString () veoma važna metoda. Ako pravimo svoju klasu i ako želimo da naš objekat bude polimorfan i da može da se spaja sa stringom moramo preklopiti ovu metodu.

Ove metode uvek postoje i naravno da ih možemo redefinisati po potrebi.

Problem dodele. Ukoliko imamo dva objekta i napišemo r1=r2 izjednačiće se reference tj r1 će pokazivati na objekat na koji pokazuje r2. Mi želimo u stvari da kloniramo taj objekat. Postoje dva načina.

```
Object clone () {
      Rectangle cln=new Rectangle (a,b);
      return cln;
}
```

Kao prvo možemo da redefinišemo metodu clone() u klasi za koju nam treba ova operacija. Slika desno. Sada napišemo r1=r2.clone(). Ovo se ne može raditi jer se ovde dodeljuje predak potomku a to nije dozvoljeno pa se mora raditi konverzija a samim tim i pisati r1=(Rectangle)r2.clone().

Kao drugo možemo da ne diramo metodu clear() već da kloniranje vršimo direktnom primenom konstruktora, naravno pod uslovom da smo ga već napravili. I sada se može napisati r1=new Rectangle (r1);

```
Rectangle (Rectangle rec) {
      a=rec.a; b=rec.b;
}
```

Ova druga primena ne traži typecasting i konstruktor bi već sam po sebi bio napisan. Ali ono prvo rešenje nudi dinamičko povezivanje. Jer ako ne napravimo clone() klijent može da je pozove i da ne dobije validan rezultat. Zato se obično rade oba načina.

Problem relacije jednakosti. Ako napišemo r1==r2 porede se samo reference. Ako hoćemo da poredimo objekte moramo koristiti običnu metodu jer u javi nema preklapanja operatora. Jednop rešenje je da redefinišemo metodu equals (). Kada je virtuelna metoda redefinisana ona mora zadržati ime i parametre (sve metode u javi su virtuelne pa i ova).

```
boolean equals (Object rec) {
      return ((a==((Rectangle)rec).a)&((b==((Rectangle)rec).b);
}
```

Postoji još jedna mogućnost. U načinu pod jedan nailazimo na problem. Pošto je parametar object može biti prosleđen bilo koji objekat. Drugi način je:

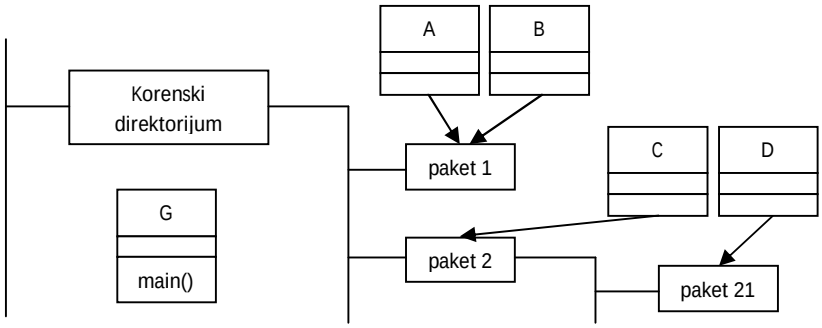
Ova metoda bi radila bez problema. Ali sada titanje je ako imamo Circle c; Rectangle r; Square s; i napišemo r.equals(s) to će raditi bez problema jer je pretpostavka da je Square nasledio Rectangle. r.equals(c) neće raditi za prvi način ali hoće ako ovu metodu equals definišemo na drugi način jer ako redefinišemo metodu equals menjajući joj parametre onda će postojati obe verzije te metode i jednostavno će se naći prvi equals u prethodnoj klasi koji odgovara ovim parametrima. Ovde će se pozvati equals iz klase Object i vratiće se false. Ako u ma kojoj situaciji ne sme da se desi da se prosledi c kao argument onda koristimo prvi način. U suprotnom se koristi drugi način.

```
boolean equals (Rectangle rec) {
      return (a==rec.a)&(b==rec.b);
}
```

Modularnost i inkapsulacija

Ovde važe iste osobine kao i u C++ ali su drugačija sredstva. Način na koji se realizuju moduli u javi je dobra stvar. To su **paketi**. Paketi su skupovi klasa. Problem u C++ je bio kod mejerove jednakosti ali se javljao i problem razmrvljenosti softvera. Ono što se traži je da klasa ima osobinu modularnosti. Da bi klasa bila moduo, svaka klasa mora biti poseban fajl. Java ima takvu koncepciju modula i klasa ima osobinu modularnosti i to otvara problem razmrvljenosti.

Java to rešava na jedinstven način sa dva nivoa modularnosti. Prvi nivo omogućava da se svaka klasa nalazi u sopstvenoj datoteci, a datoteke se organizuju u pakete čime se uvodi kopaktnost tj nema razmrvljenosti. Cena koja se plaća za ovo je ta da struktura paketa tj paket se vezuje za folder i oni moraju da nose isto ime. Naziv paketa ne sadrži velika slova. U ovakvoj organizaciji postoji korenski direktorijum. Naredba



za definisanje pripadnosti klase direktorijumu stoji ispred klase i glase `package ime_paketa;` npr ispred klase A bi stajalo package paket1. Pošto sistem mora da prati strukturu paketa moramo navoditi i pune putanje tako ćemo za klasu D da imamo package paket1.paket21; Sa komandne linije kompilacija se vrši kao javac paket1\A.java.

Puno ime klase A više nije A, već je ime koje sadrži i paket u kome se nalazi paket1.A. Sada se klase koriste kao i svake druge samo sa punim imenom npr paket1.A obA=new paket1.A(arg);

Da bi se ovo ublažilo za skraćivanje ovih poziva u klijentu možemo napisati import i posle čega sledi ime klase koju ubacujemo npr import paket1.A; i posle ovoga klasu A pišemo samo sa imenom. Ova naredba se ne nalazi u klasi. Ako hoćemo da koristimo sve klase iz nekog paketa možemo napisati import paket1.*; i time će biti uključene sve klase.

U javi se polazi od postulata da svaka klasa pripada nekom paketu. Ono što se dešava u javi jeste da u jednom direktorijumu ima mesta za dva paketa. Prvi paket je imenovan npr paket1 a drugi je neimenovan i on može da se nađe u istom direktorijumu kao i prvi paket. Ova dva paketa nemaju veze međusobnoa klase se odnose kao klijenti. Klase u imenovanom paketu su naravno bliske i imaju pravo pristupa. Isto važi i za klase u neimenovanom direktorijumu.

Pošto u javi klijent bi trebao da bude u korisničkom direktorijumu i to bi dovelo do neupotrebljivosti jave kao jezika. To se rešava na poseban način. Uvedena je posebna varijabla classpath. Tu se ubacuju svi direktorijumi u kojima se mogu tražiti klase koje pripadaju drugom projektu. Classpath treba dovesti do

direktorijuma koji se nalazi iznad našeg direktorijuma. Ovo se radi zbog portabilnosti. U nekim razvojnim okruženjima ovo se radi automatski na primer u eclipse. Ako želimo da uključimo sve direktorijume koji se nalaze u direktorijumu iz koga pozivamo setclasspath pišemo setclasspath=*;

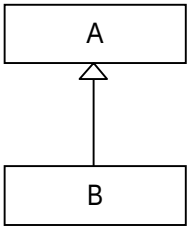
Kontrola pristupa članovima klase

Ona je u javi sprovedena na 4 nivoa. **Private** deo, njemu pristupaju metode matične klase. Sledeći nivo nema posebnu reč i zovemo ga **default** ili **friendly**. Onome što je kontrolisano ovim nivoom mogu pristupiti metode iz sopstvene klase kao i metode iz klasa istoga paketa. **Protected** nivou imaju pristup klase iz toga paketa kao i izvedene klase. **Public** delu pristupaju sve klase. Default je jači kandidat od private kada treba nešto zatvoriti.

Kao i podaci i klasa može da ima modifikatore koji kontrolišu pristup samoj klasi. Klasa može da se kontroliše nivoima **public**, što je i očekivan pristup i on dozvoljava da se klasa upotrebljava na svim mestima tj da se njeno ime nađe u svim drugim klasama, kao i **default** što znači da je klasa u potpunosti nevidljiva izvan paketa. Obična klasa je public i to je slučaj sa većinom klasa jer je public kandidat za kontrolu pristupa, a default stavljamo, odnosno ne pišemo ništa ispred imena klase, kada želimo da klasa bude korištena samo u paketu kao neka klasa za unutrašnje potrebe. Test program je poželjno da se nalazi izvan paketa u kome se nalaze klase jer u pisanju klase može da se desi da zaboravimo da napišemo public i to se vrlo lako otklanja testiranjem izvan paketa jer u tom slučaju ta klasa ne može da se koristi. Iako se u javi svaka klasa nalazi u posebnoj datoteci java dozvoljava da se više klasa nađu u jednoj datoteci s tim da samo jedna klasa onda ima public kontrolu pristupa, ona koja i nosi ime datoteke, a ostale su po pravilu sa default kontrolom pristupa.

Nasleđivanje

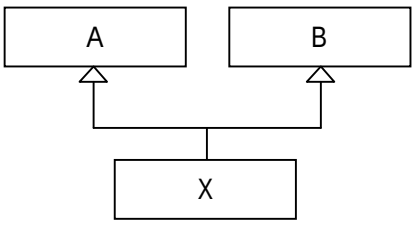
Na nivou metodologije nasleđivanje je isto kao i kod C++ jer metodologija ne zavisi od programskog jezika, ona je univerzalna. Što se tiče jave sintaksa nasleđivanja je sledeća:




```
modifikator class B extends A {
...
}
```

Sama po sebi klasa se B se izvodi iz klase A pod istim uslovima kao i u C++. U javi postoji mogućnost proširivanja kao i u C++, dodavanja polja i metoda. Postoji i mogućnost redefinisanja metoda i redefinisana metoda ne sme da se razlikuje od originalne ni po imenu ni po argumentima. Ako redefinišemo metodu metodom koja ima samo isto ime onda će se u klsi B pojaviti obe metode. Nasleđivanje u javi takođe je tranzitivno. Višestrukog nasleđivanja nema i to se rešava upotrebom interfejsa. U C++ u ulozi interfejsa bi se pojavila čisto apstraktna klasa.

Višestruko nasleđivanje možemo da izvedemo ali ne potpuno, tj možemo da ga zamenimo kompozicijom. Jedino što ne možemo da postignemo na ovaj način jeste inkluzioni polimorfizam.




```
modifikator class X {
public A a;
public B b;
}
```

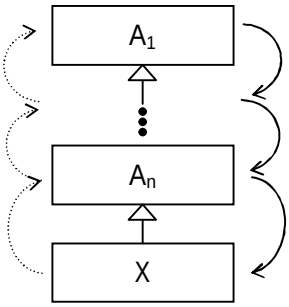
Ukoliko želimo da što više sakrijemo ovakav način implementacije višestrukog nasleđivanja mi možemo da redefinišemo sve metode iz klasa A i B.

Što se tiče nasleđivanja u pretku mogu da se nađu polja sa modifikatorom protected i on dozvoljava da to polje vide sve klase paketa kao i klasa koja nasleđuje datu klasu.

U javi je moguće i sprečavanje nasleđivanja ali nema mnogo situacija u kojima se koristi. Sve što treba da se uradi je da se u ime klase stavi reč final posle modifikatora za pristup. Takođe ovaj modifikator (final) može da se koristi i za metode i on u tom slučaju govori da ta metoda ne može da se redefiniše.

Što se tiče konstruisanja i odnosa između dve klase (u nasleđivanju) java u potpunosti podstiče poštovanje Demetrinog zakona, tako što svaka klasa pored polja **this** ili reference na samu sebe, ima i polje

super i to je polje preko koga se može pristupiti verzijama iz bazne klase. Ukoliko u klasi B imamo metodu m, koja postoji i u klasi A, mi njoj možemo pristupiti tako što ćemo je pozvati preko polja super: super.m(). Može se pristupiti metodama m iz predaka klase a kao super.super.m() ali ovo nije preporučljivo. Konstruktor iz bazne klase poziva se na poseban način kao super (parametri). Kada se konstruktor klase A poziva iz konstruktora klase B ova naredba mora biti na prvom mestu.



Konstruisanje se vrši na istom principu kao i u C++. Objekat klase X ima delove nasleđene iz klase A_1, klase A_2 ... Princip je da se deo koji potiče od pretka A_i konstruiše konstruktorom te klase. Ovo se obezbeđuje tako što se u svakoj klasi poziva konstruktor prethodne klase. Zato super (param) mora biti na prvom mestu. Ova veza može da se ostvari na dva načina. Može da se stavi super (param) ili da se ništa ne stavlja. U drugom slučaju biće pozvan podrazumevani konstruktor, ali postoji verovatnoća da data klasa nema podrazumevani konstruktor.

Zbog jedinstvene hijerarhije klasa, svaka nova klasa nasleđuje klasu Object, bilo direktno ili indirektno (zbog tranzitivnosti), tj svaka klasa osim Object je izvedena klasa.

Kada se metoda redefiniše nivo kontrole pristupa se može proveniti ali redefinisanoj metodi se ne sme dodeliti oštrija kontrola pristupa. To je zbog inkluzionog polimorfizma jer npr ako dodelimo potomak pretku i pozovemo metodu m koja je redefinisan i kontrola pristupa je promenjena sa public na protecte, doćiće do pucanja programa.

Java podleže stalnim promenama i stalno se menja. Pogotovo je to primetljivo na brojnim gotovim klasama koje java u sebi sadrži. Te klase se menjaju, ali se menjaju i metode tih klasa... Tako je nekada klasa Vector bila final, tj nije se mogla naslediti da bi to kasnije bilo ukinuto. Reč **deprecate** (nije više dobro kao što je bilo) se koristi za metode koje se menjaju uvođenjem novih metoda.

Polimorfizam

Kao i u C++ prisutna je ista podela polimorfizama. **Parametrizovani polimorfizam** se pojavljuje kao generičnost. U osnovi u javi se generičnost izvodi kao u Delfi Paskalu, a ne kao u C++, i zasnovana je na činjenici da je klasa Object opšti predak, i gde treba da ide objekat bilo koje klase stavlja se Object. Do poslednje verzije jave generičke klase u pravom smislu te reči nisu postojale. **Inkluzioni polimorfizam** je isti kao i u C++ jer je on vezan za metodologiju. **Preklapanje operatora** u javi ne postoji, ali preklapanje funkcija naravno da postoji. **Koercitivni polimorfizam** postoji. Typecast postoji i u javi i najčešće se pojavljuje nad referencom (kod nasleđivanja). Koristi se i implicitno i eksplicitno.

Inkluzioni polimorfizam

On se odnoosi na operaciju pridruživanja, bilo direktno, bilo preko zamene parametara argumentima, bilo da se vraća rezultat funkcije koji je tipa pretka a vraća se potomak.

Potomak može da se dodeli pretku a obrnuto nije moguće. U javi nema čistih objekata kao u C++ već ovde radimo sa referencama pa kada pišemo pr=po ovde dodeljujemo referencu tipa Potomak referenci tipa Predak. Ovo je analogno dodeli pokazivača u C++. Sada kada pozovemo pr.m() (u C++ zbog ovakve situacije su uvedene virtuelne metode) biće pozvana metoda m() iz potomka. U Javi nema problema kao u C++ jer se u javi sve metode ponašaju kao virtuelne ako mi ne kažemo drugačije. Da bi neku metodu proglasili za nevirtuelnu treba da dodamo final i tada ona ne može da se redefiniše što čini postupak bezbednim.

Koercitivni polimorfizam (typecast)

Typecast se intenzivno koristi. Moguć je i nad baznim tipovima ali nam je mnogo bitniji nad referencama. po=pr nije moguće bez typecasta, i treba da se piše po=(Potomak)pr ali ovo je veoma rizično. Ovo može da proradi jedino ako pr pokazuje na objekat koji je u stvari tipa Potomak. Da bi se proverilo na šta referenca pokazuje, tj da li pokazuje na objekat koji je instanca neke klase postoji naredba **instanceof**. Sada je bezbedno pisati:

```
if (ob1 instanceof K2) ob2 = (K2) ob1;
```

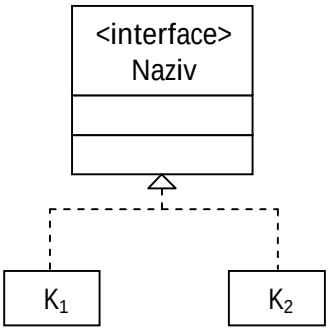
Apstraktne klase i interfejs

U C++ apstraktne klase su nepotpune. U javi apstraktna klasa je klasa koja se ne može instancirati. Ona se opisuje modifikatorom **abstract** tj neće moći da se instancira. Ovde se klasa eksplicitno proglašava za apstraktnu. U C++ je bilo dovoljno da sadrži jednu apstraktnu metodu da bi klasa bila apstraktna, dok u javi apstraktna klasa može a i ne mora da sadrži apstraktne metode, i ina se eksplicitno proglašava za apstraktnu. Naravno ako bude imala apstraktne metode ona će biti proglašena za apstraktnu. Apstraktna klasa se koristi za pravljenje mini hijerarhija (npr figure).

Interfejsi nisu klase, a u javi se koriste intenzivno. Po ponašanju oni bi se skoro pa poklopili sa apstraktnom klasom u C++. Formalne osobine su sledeće:

1. Nemaju ni jednu operacionalizovanu metodu.
2. Najčešće nemaju polja i ako se definiše polje podrazumeva se da je public static final.
3. Interfejsi mogu biti public ili default.
4. Svi članovi interfejsa su public.
5. Ne može se instancirati.
6. *(najvažnije)* Interfejsi omogućuju inkluzioni polimorfizam.

Interfejs se u javi prikazuje kao klasa, a prepoznaje se po interface stereotipu pored koga ima i naziv.



U odnosu sa klasama interfejs stoji u vezi koja se zove veza implementacije ili operacionalizacije. Veza ima vrlo toga zajedničkog sa nasleđivanjem ali ova veza se predstavlja isprekidanom linijom. Interfejs se može implementirati u više klasa a ne samo u jednoj, a takođe i interfejs može da bude nasleđen ali samo od strane drugog interfejsa. Takođe kao i klasa može biti sa modifikatorima kontrole pristupa public i default.

Veoma je važno napomenuti da za interfejs važi inkluzioni polimorfizam. Interfejs ne može da se instancira ali može da se definiše referenca na interfejs:

```
Naziv inter = new K ();
```

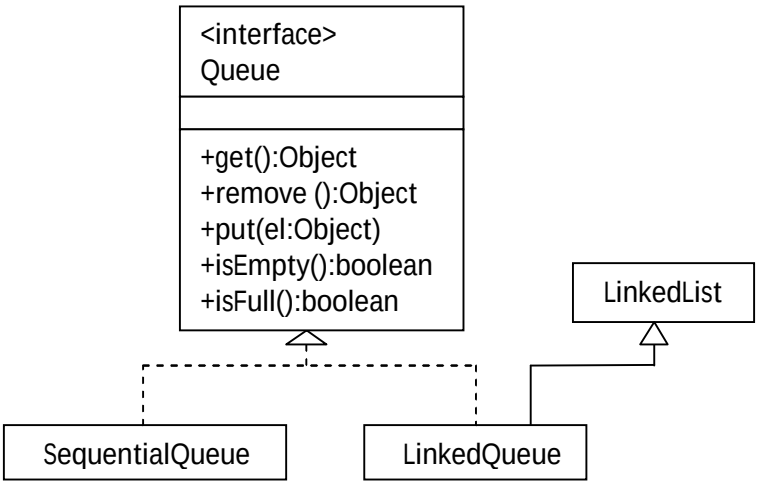
```
modif  interface Naziv {
...
}
modif class K implements Naziv {
...
}
```

I ovo se ponaša tako što se klasa vidi kroz metode koje se nalaze u interfejsu. Relacija instanceof važi i za vezu operacionalizacije kao što je slučaj i kod klasa.

U opštem slučaju možemo da implementiramo beskonačno interfejsa i posle reči implements samo se redom navode sa zapetom kao separatorom. Implementacija interfejsa samo vizuelno liči na višestruko nasleđivanje. U javi se interfejsi intenzivno koriste iz nekolicine razloga kao što su nedostatak višestrukog nasleđivanja ali i psihološki aspekt jer je interfejs karakterističan samo za javu a u ostalim programskim jezicima u ovom obliku ne postoji. Postoji više primena interfejsa ali neke od tipičnih ćemo sada navesti.

Omogućuje višestruke realizacije klase. Interfejs po svojoj prirodi nije ništa drugo do interfejs klase u pravom smislu te reči. Kod struktura podataka smo videli da one mogu imati jednu logičku realizaciju strukture ali više fizičkih realizacija, a to važi i ako se strukture podataka realizuju u objektnom programskom jeziku. Interfejs je taj koji nam omogućuje da sa stanovišta logike, klasa koja ima više fizičkih realizacija, održi isto

ponašanje, Na primer red može da se realizuje sekvencijalo i spregnuto, i od nas se zahteva da ponašanje reda ne zavisi od tipa fizičke realizacije koji odabere klijent. To se može obezbediti tako što dve klase sa različitim realizacijama reda nasleđuju isti interfejs.




```
Queue q1 = new SequentialQueue ();
Queue q2 = new LinkedQueue ();
```

Kada je uklanjanje u pitanju već je odavno u programskim jezicima konvencija da se vraća uklonjeni objekat, a to da li ćemo ga koristiti ili ne to ostaje na nama. Takođe je konvencija da se funkcije koje ispituju neke boolove promanljive započinju rečju is kao npr isEmpty(). Što se tiče funkcije one će biti realizovane i u jednoj i u drugoj klasi s tim da se funkcija isFull() u LinkedQueue klasi definiše tako da uvek vraća false vrednost, jer ovde ne može doći do stanja prepunjenosti.

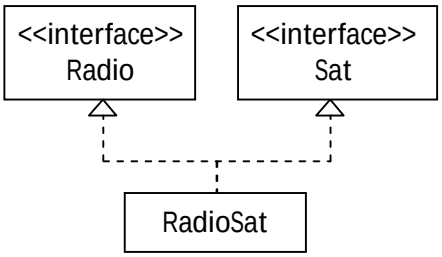
Sada klijent može izabrati jednu od dve realizacije reda tako žto će definisati referencu na Queue. Sada se za klijenta q1 i q2 ponašaju isto jer ih on u stvari vidi preko interfejsa.

```
public interface Queue {
    Object get();
    Object remove();
    void put (Object el);
    boolean isEmpty();
    boolean isFull();
}

java.util.LinkedList;
public class LinkedQueue extends
            LinkedList implements Queue {
    public Object get() {
        return getFirst();
    }
    public Object remove() {
        return removeFirst ();
    }
    public void putObject (Object el) {
        addLast(el);
    }
    public boolean isFull () {
        return false;
    }
}

Queue q = new LinkedQueue();
```

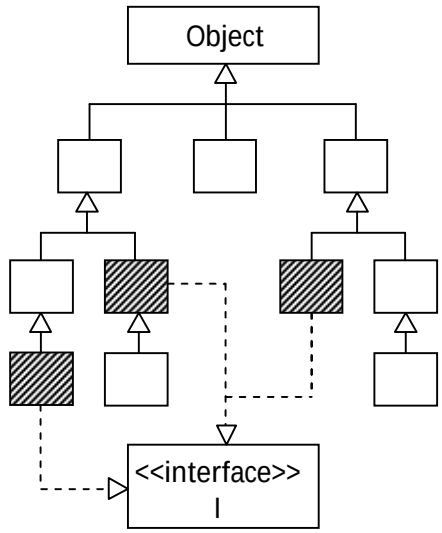
Pomoću interfejsa može da se obezbedi nasleđivanje implementacije. Java ima već urađenu jednostruko spregnutu listu koju može da nasledi naša klasa LinkedQueue. Ova jednostruko spregnuta lista je u stvari realizovana kao dvostuko spregnuta lista iz nekog razloga (možda da se ubrza neka metoda...). U C++ bi bilo potrebno višestruko nasleđivanje za ovo jer umesto interfejsa mi imamo apstraktne klase.

Sada kada hoćemo da pravimo spregnutu realizaciju reda nasledićemo LinkedList. To je odlika objektne metodologije da sve što imamo ne pravimo ponovo. Metode getFirst() i removeFirst() već imamo u LinkedList pa ih ovde samo pozivamo. Takođe metoda isEmpty() takođe postoji u toj klasi pa nema potrebe da je operacionalizujemo.

U C++ ovakvu situaciju, možda čak i nešto bolju postižemo private nasleđivanjem. Ovakva situacija je u skladu sa svim postulatima jave osim jednog. LinkedQueue ima metode koje joj trebaju ali sadrži i sve metode iz LinkedList koje joj u stvari ne trebaju jer će biti pozivana preko interfejsa. To je jedina primedba koja može da se navede kod ovakve realizacije programa.



Implementacija nam omogućava više pogleda na klasu. Taj mehanizam je korišten kod C-ovih biblioteka na primer. Klasa RadioSat se može posmatrati i kroz radio i kroz Sat. Ovo liči na višestruko nasleđivanje ali kod njega bi bilo karakteristično da postoji gotov kod za Radio i Sat i on se naredbom stavlja u RadioSat. Ovde je sav kod baš u toj klasi. Takođe ovde važi i inkluzioni polimorfizam. Višestruko nasleđivanje u javi se može uraditi pomoću kompozicije, ali mana je nedostatak inkluzionog polimorfizma.



Inkluzioni polimorfizam bez nasleđivanja. Uzmimo za primer da se treba napraviti metoda koja prihvata objekte samo iz pojedinih klasa određene hijerarhije. Na primer samo objekte klasa koje sadrže serijalizaciju (postupak pretvaranja objekta u niz izlaznih podataka). Tipičan primer je memorisanje programa jer tu treba memorisati i njegovo stanje. Ovo radimo tako što napravimo interfejs bez metodi, bez ičega.

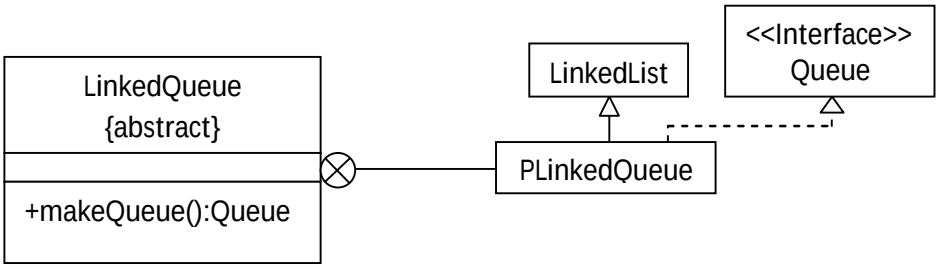
Enumeracija. Java sve do poslednje verzije nije imala enumeraciju, a potreba je bila velika. U poslednjoj verziji pojavila se posebna klasa Enum koja je omogućila enumeraciju. Pre toga (što ne znači da se danas ne može koristiti) se koristio interfejs bez metoda za realizaciju enumeracije.

```
public interface Boja {
      int Crvena=0;
      int Zuta=1;
      int Plava=2;
}
```

Unutrašnje klase

Praktično na isti način postoje i u C++ ali se mnogo više koriste u javi. U C++ unutrašnji mogu da budu i strukture i klase..

Unutrašnje klase su članovi klase. Java to širi pa se unutrašnja klasa može definisati i unutar metode. Ova klasa može da bude private, protected, public ili default, a može da bute i static. Opšta namena ovik klasa je za unutrašnje potrebe klase u kojoj se nalaze. Na primer kod klase lista element te klase nije slog već je klasa.



Element liste je pokazivač i njega ćemo definisati u unutrašnjoj klasi. Pomoću unutrašnje klase metode iz LinkedList mogu da se sakriju i time reše problem od malopre.

Da bi sadržaj LinkedListe bio nedostupan mi ćemo sadržaj LinkedListe da nasledimo u PlinkedQueue i proglasimo je za private i implementiramo interfejs Queue. LinkedQueue čemo dodavanjem reči abstract onemogućiti instanciranje. Dodavanjem metode makeQueue() omogućujemo stvaranje objekta.

Ovim kodom mi postižemo da je LinkedQueue ponaša isključivo kao LinkedQueue i spolja se ne može pristupiti ni jednom preuzetom članu iz LinkedList. Unutrašnja klasa (njen objekat) se ne može instancirati durektno spolja, već se instancira objekat spoljašnje klase pa preko njega objekat unutrašnje klase.

```
import java.util.LinkedList

public abstract class LinkedQueue {
    public static Queue makeQueue () {
        return new PLinkedList ();
    }

    private static class PLinkedQueue extends
                    LinkedList implements Queue {
        //kod kao kod LinkedQueue
    }

Queue q = LinkedQueue.makeQueue();
```

Omotačke klase

Generičke klase, kao što je na primer generički red, mogu da kao elemente da imaju objekte bilo keoje klase jer je njihov parametar u stvari Object. Javlja se pitanje šta ako su ti elementi bazni tipovi. Direktan unos baznih tipova nije onemogućen već njih moramo pretvoriti u objekte. Taj posao upravo rade omotači tj vrše konverziju u objekat baznih tipova. Na primer Integer, Long, Float, Double, Character, Boolean su sve omotačke klase. Prve četiri nasleđuju klasu Number. Sada ukoliko imamo neki red q možemo napisati q.add(new Character ('a')). Nove verzije jave dozvoljavaju da se napiše samo q.add('a'). Neke od metoda omotačkih klasa:

```
//konstruktori
Integer (int i) //konstruiše objekat na osnovu baznog tipa
Integer (String s)
//metode
boolean equals (Object ob);
int intValue () //očitavanje vrednosti koja se nalazi u objektu
long longValue ()
float floatValue ()
double doubleValue ()
String toString () // Ukoliko želimo da vratimo string možemo da izvršimo
                    konkatenaciju sa nekim drugim stringom, i ovde se, za
                    razliku od baznih tipova koji se automatski konvertuju,
                    Poziva metoda toString pošto je ovo objekat.
//statičke metode
static String toString (int i)
static String toBinaryString (int i)
static String toOctalString (int i)
static String toHexString (int i)
```

Kontejnerske klase

One predstavljaju objektnu realizaciju struktura podataka. Java ih ima priličan broj kao npr red, stek, dek, listu... Intenzivno se koriste Vector, Hashtable, LinkedList (jednostuko spregnuta lista koja je ovde izvedena kao dvostruko spregnuta lista)... U novijim verzijama jave sve ove klase su generičke ali mogu i da se koriste u svom izvornom obliku ali nam prevodilac daje upozorenje da koristimo „sirove“ (raw) tipove.

Vector. U pitanju je dinamički niz (dinamička struktura kod koje je dozvoljeno dodavanje i učitavanje svakog elementa i gde se pristup vrši prema indeksu). Kod uklanjanja dolazi do pomeranja elemenata ka početku. Kod dinamičkog niza kada dođe do prepunjenosti, a to je česta situacija, prostor koji zauzima dinamički

niz će se proširiti (zauzme se novi prostor, kopira se sadržaj..). Kada se ovaj prostor širi širi se za blok koji možemo odrediti koliki je. Neke metode:

```
void add (Object el) //dodaje element
void add (int index, Object el) //dodaje element na određeno mesto
Object elementAt (int i) //očitavanje sa i-tog mesta
Object get (int i)
Object firstElement ()
Object lastElement ()
int size ()
boolean isEmpty ()
Object remove (int i)
void removeAll ()
```

Elementi ovih klasa se definišu stavljanjem tipa klase u <Object>. Ako tako ne deklarišemo tip elemenata klase koristi se sirova klasa pa kada hoćemo da koristimo naš tip moramo da vršimo typecast. Primer ovakvog deklarisanja tipa elemenata: Vector <K> v; i označava vektor sa elementima klase k.

Hashtable. Ovakve strukture zovemo heš tabele. Heš tabele služe za što je moguće brži direktan pristup preko ključa. Ideja je jednostavna. Imamo ključ K i hoćemo da sa zadanim ključem što brže dobijemo element koji mu odgovara. Jedna ideja je da transformišemo ključ tabele u neku adresu. Osnovni način je da se ključ pretvori u numerički oblik, zatim na bazi numeričkog elementa ključa generiše pseudo slučajan broj. Kada se dve int vrednosti pomnože i pređe se gornja granica ne javlja se greška jer dobijeni broj, koji nastaje odsecanjem ostatka, više manje liči na slučajan broj i otprilike generatori pseudo slučajnih brojeva rade na ovome principu.

Hashtable ima određenu veličinu, ako se pređe tabela se kompletna prepisuje. Takođe može da se desi da za isti ključ dobijemo istu adresu. Ovo se zove kolizija ključeva.

Iteratori

Iterator je složena operacija koja omogućuje sistematizovan obilazak strukture podataka. Iterator za niz u C-u je for petlja. Iteratori su u Javi interfejsi i dosta se koriste. Uvedeni su da bi iteriranje kroz strukture podataka bilo univerzalno. Bitne operacije su startovanje, očitavanje sledećeg, provera da li postoji naredni element. Iterator je sam po sebi objekat. Postoji više interfejsa kojim su iteratori implementirani. Jedan od njih je Enumeration. Enumeration ima metode boolean hasMoreElements (); i Object nextElement (); Sada možemo to primeniti na sledeći način:

```
Vector<E> v=new Vector<E>();
Enumeration iter;
iter=v.elements();

while (iter.hasMoreElements)
      obraditi iter.nextElement();
```

Metoda elements konstruiše iterator. Za svako korištenje iteratora ova metoda se ponovo koristi. Ukoliko je vektor sirov tip u while petlji je potrebno izvršiti typecasting posle pozivanja neke vrste obrade podataka. U novoj javi je uvedena i druga mogućnost za kreiranje iteratora. Napravljen je novi interfejs Iterator u kome postoje metode boolean hasNext (); Object next() void remove();

Sada se takođe više i ne koristi toliko npr v.elements() već v.iterator (). Iteratori mogu da se izvedu i preko funkcija kao na primer for (E el:v) obraditi el; el je element u koji se učitava sledeći element iz kontejnerske klase. Prelazak na sledeći i provera kraja su automatizovane. Elementi hash tabele moraju imati hashCode() i nema potrebe daje pravimo jer podrazumevana radi sasvim solidan posao.

Anonimne klase

Ovo su bezimene klase. Instanciraju se direktno iz metode, a u metodi se i opisuju. Primer je u stvari iterator jer su iteratori anonimne klase.

```
public class EnhancedVector extends Vector {
...
public Enumeration reverseElements() {
      return new Enumeration () {
            int current=size()-1;
            public boolean hasMoreElements() {
                  return current>-1;
            }
            Public Object nextElement () {
                  if (current<0) return null;
                  else return get (current--);
            }
      }
}
```

Realizacija anonimne mape počinje sa return new Enumeration (). Ovo znači da se kao rezultat vraća objekat klase koja implementira ovaj interfejs ili nasleđenu klasu. Ova anonimna klasa se i definiše istovremeno sa instanciranjem. U UML-u anonimna klasa se predstavlja stereotipom <<Anonimous>>.

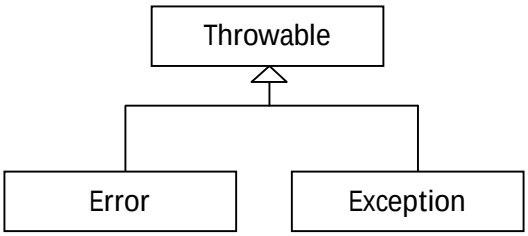
Prevencija otkaza

U javi sredstva za prevenciju otkaza su veoma slična kao u C++. Postoje tri načina: havariski prekid programa, korištenje rezultata metode za indikaciju uspešnosti, obrada izuzetaka.

Havarijski prekid programa. Java ima dva sredstva. Jedno je standardno u obliku posebne metode koja je statička metoda u klasi System u paketu java.lang. To se System.exit(n). U javi havarijski prekid programa postižemo najbolje pomoću izuzetaka jer postoje posebne klase izuzetaka.

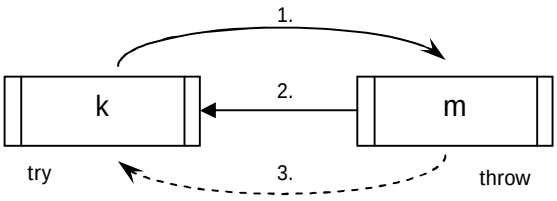
Korištenje rezultata metode za indikaciju uspešnosti. Ovde nema ničeg novog. Takođe je ovde glavni problem jer zauzimamo kanal za razmenu podataka pa moramo da vraćamo vrednost preko parametara. U javi radije koristimo izuzetke. U javi kada metoda vraća kao rezultat objekat i tu imamo na raspolaganju vrednost null. Ovaj pristup je češći nego u C++.

Rukovanje izuzecima. Izuzeci su u javi isključivo objekti. Svi izuzeci objekti su članovi klasa i te klase su uređene u jednu hijerarhiju. Na vrhu je throwable. Error i ne koristimo jer je to za interne potrebe.



RuntimeException je posebna klasa sa izuzecima koja ima specijalne osobine. U javi generisanje izuzetaka se vrši naredbom throw a prihvatanje sa try. RuntimeException generiše takve izuzetke koje možemo da pustimo neobrađene. U javi izuzetak koji se koristi u metodi mora se u njoj i deklarisati dok izvedeni izuzeci iz RuntimeException se ne moraju deklarisati. Ima veliki broj izuzetaka. U klasama izuzetaka postoje metode:

```
K () //konstruktor
K (String poruka) //konstruktor izuzetka koji ispisuje poruku
String getMessage ()
String toString ()
void printStackTrace () //ispisuje put izuzetka
```



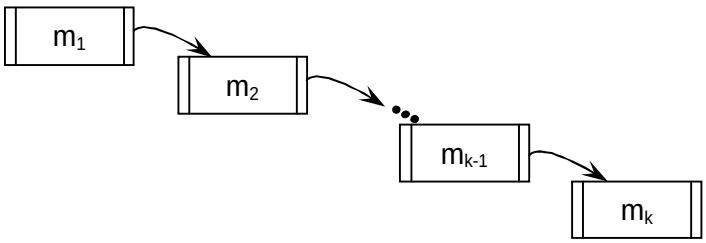
Metoda m baca izuzetak dok k prima izuzetak sa try blokom. S obzirom na činjenicu da je svaki izuzetak objekat u throw se dodaje new k(param). Metoda koja generiše izuzetak mora taj izuzetak da deklariše inače prevodilac neće da prevede metodu. To izgleda kao :

```
tip m(param) throws Ex1,...,Exn {
      throw new ...
}
```

U metodi m se deklarišu samo klase izuzetaka koji se koriste. Tu važe dva pravila:

1. Važi inkluzioni polimorfizam. Ako neka metoda deklariše izuzetak onda se unutar klase može generisati izuzetak koji pripada toj deklarisanoj klasi ili nekoj od izvedenih klasa. Tako možemo da stavimo Exception pa metoda može da generiše bilo koji izuzetak, Ove i sistemske izuzetke omogućava deklarisanje klase Throwable.
2. Iz ovakve situacije je izuzeta klasa RuntimeException. Ona ne mora da se deklariše. S obzirom da se izuzetak iz te klase propagira bez ikakvog dodatnog koda. Izuzetak se propagira do maina i tu završava funkciju. Ovo je bolje od exit jer ovde dobijamo invormaciju gde je pukao program.

Pitanje prihvatanja i obrade izuzetaka



mk mora deklarisati izuzetak osim ako on nije RuntimeException. I taj izuzetak može da ide u prethodnu metodu i ona može na tri načina da reaguje na taj izuzetak.

Prvi način. Traži se prvi catch koji može da obradi taj izuzetak. Treba obratiti pažnju na inkluzioni polimorfizam. Prvo se stavlja handler za potomak pa tek onda za predaka. Ulogu univerzalnog handlera igra Catch (Exception e) ili catch (Throwable e) i on ide na kraj. Kod ovog načina izuzetak je prihvaćen i ne propagira se dalje.

```
try {
    //rizični poziv
} catch (Ex1 e) {
    //reakcija
} catch (Ex2 e) {
    //reakcija
} catch (Exn e) {
    //reakcija
}
```

Drugi način. metoda prosledi dalje izuzetak i ništa ne radi. U javi naravno da važi pravilo da izuzetak ne prolazi nedetektovan. Ovde važi da ako metoda m_{k-1} ne obrađuje izuzetak već ga prosleđuje dalje ona mora da ga deklariše sa throws Ex1 e...

Treći način je u stvari kombinacija prva dva načina. Tj izuzetak se prihvati, delimično obradi i prosledi dalje. Sve što treba uraditi je da kada se izuzetak obradi ubaci se throw e.

U okviru handlera obično stavljamo e.printStackTrace(). Postoji još jedan deo, tj posle catch naredbi možemo da stavimo finaly. Taj blok ima definicionu metodu koja će sigurno biti izvršena. To se koristi kada na primer treba da zatvorimo datoteku. Sve što mora da se izvrši stoji u finaly.

Klasa Class

Ova klasa je vrlo različita od ostalih klasa. U ostalim programskim jezicima ova klasa ne postoji. U javinom bytecode-u nalaze se instrukcije koje ne mogu da se nađu u mašinskom kodu. Te instrukcije su određene metodama virtuelne mašine. Ovu klasu u nekoj literaturi zovu metaclass. Ona nam omogućuje da pokupimo podatke o drugim klasama. Ti podaci se mogu dohvatiti na dva različita načina. Direktno i preko objekta klase.

Direktno. Direktno tj bez posredovanja objekata podaci se mogu dobiti na dva načina

```
(1)   Class c = Class.forName („My Class“);
(2)   Class c = MyClass.class;
```

Prvi način generiše izuzetak. Metoda c nam omogućuje pristupanje delovima naše klase. Drugi način ne generiše izuzetak i ovo se zove klasni literal (neka vrsta konstante). Sada o klasi MyClass možemo prikupiti podatke preko c. Mogu se koristiti i klasni literali vezani za bazne tipove kao npr boolean.class;... Iz nekog razloga može da se napiše i boolean.type;...

Dohvatanje podataka o klasi preko objekata klase. U toku izvršavanja programa mi možemo da vidimo kojoj klasi pristupa ovaj objekat. To se radi preko metode getClass(). Ovo se može raditi i na drugi način preko equals. Za razliku od instanceof ovaj način daje true samo ako objekat pripada ovoj klasi (a ne i potomcima). Postoji i još jedan način korištenja. I ovaj treći način radi isto kao i instanceof.

```
MyClass ob = new MyClass();
Class c = ob.getClass();
-----------------------------------
ob.getClass().equals(MyClass.class);
-----------------------------------
MyClass.isInstance (ob);
```

U klasi Class postoji mnogo metoda a samo neke od njih su:

```
Class getClass();
String getName ();
Package getPackage ();
Class getSuperClass ();
```

Generičke klase

Generičnost u javi je pre rešavana upotrebom klase Object. Ako želimo da suzimo to samo na omotače brojnih tipova stavljamo Number. Problem je bio stalni typecast maltene za svaku operaciju.

Po načinu upotrebe radi se kao i u C++. Mogućnosti templatea su mnogo šire nego mogućnosti generičkih klasa.

```
modif Gime <P1,...,Pn> {
...
}
Gime <A1,...,An> ob=new Gime <A1,...,An> (param);
```

Nema korištenja metoda +,- za ove P... Samo metode iz Object mogu da se koriste.

Sve kontejnerske klase su pretvorene u generičke klase. Java pruža mogućnosti da se generički parametri mogu ograničiti odozgo u smislu nasleđivanja. Ovde mogu da se pojavljuju klase samo nasleđuju datu klasu. Recimo da treba da napravimo neku statičku metodu i hoćemo da parametar metode bude iz generičke klase. Public static void met (GenClass <P>,...); Prevodilac ne bi znao ako stavimo <P> prevodilac ne bi znao da li je P parametar ili stvarna klasa zato se umesto P stavlja ? i sada prevodilac zna dali je konkretizovana generička klasa.

Sve nestatičke metode slobodno raspolažu parametrima generičkih klasa, dok za statičke je uveden poseban mehanizam. Svaka metoda može da ima sopstvene statičke parametre.

static<T,Vextends T> int metoda (Tx,V[]y)

Sirovi tipovi. Nije ništa drugo nego povezivanje generičkih klasa iz stare jave i nove. To predstavlja ništa drugo nego korištenje generičkih klasa po starom načinu (typecast).

Brisanje (erasing). Generičke klase nisu mogle biti drugačije napravljene nego na osnovu već postojećeg mehanizma koji je već postojao. Kada napišemo class K1 <P> { ...P... } ovo neće stići do prevodioca već će to neka vrsta pretprocesora da zameni P sa Object i onda će da se ubaci typecast kod svih poziva i obraćanja.