Programski jezici – usmeni deo ispita

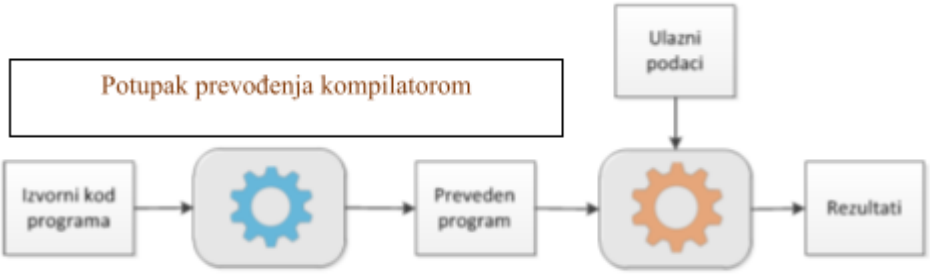
1. Podela jezika prema stepenu zavisnosti od arhitekture računara

- Prema stepenu zavisnosti od arhitekture računara, programske jezike delimo na:
  - Mašinki zavisne:
    - Mašinski kod
    - Mašinski orijentiani jezici (jezici niskog nivoa):
      - Simbolički/asemblerski
      - Makro asemblerski
  - Mašinski nezavisne:
    - Jezici visokog nivoa
      - Proceduri orijentisani
    - Jezici ceoma visokog nivoa
      - Problemu orijentisani

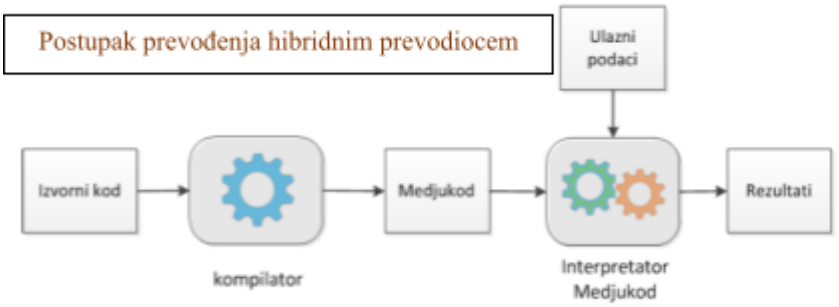
2. Načini prevođenja viših programskih jezika

- Viši programski jezici se mogu prevoditi pomoću:

Kompilatora – prevode ceo kod i kreiraju izvršnu verziju, koja se nakon toga može izvršavati neograničen broj puta, bez ikakvog ponovnog prevođenja.



Interpretatora – prevode naredbu po naredbu i svaku prevedenu naredbu odmah izvršavaju.



Hibridnih prevodilaca – najpre se kompilatorom vrši prevođenje do nivoa međukoda, koji je jako sličan asemblerkom jeziku, ali nema nikakvu zavisnost od arhitekture uređaja na kom se izvršava. Dalje se međukod interpretatorom prevodi i izvršava. Jedini deo koji je zvistan od platforme na kojojće se program izvršavati je taj interpretator međukoda.

3. Podela programskih jezika po aplikacionom domenu. Navesti po jednog predstavnika za svaki.

- Inženjerski (naučni) domen
  - Proste strukture podataka (nizovi i matrice), ali se zato zahteva veliki broj preciznih numeričkih računanja
  - FORTRAN . Prvi jezik projektovan za precizna numerička računanja
- Poslovni domen
  - Akcenat je na dobrom opisu i velikoj količini podataka koji se obrađuju
  - COBOL – prvi jezik projektovan za poslovni domen
  - Danas se u te svrhe koriste sistemi za upravljanje bazama podataka
- Veštačka inteligencija
  - Funkcionalni jezici – LISP
  - Logički jezici – Prolog
- Sistemsko programiranje

    - Zahtevajuveliku preciznost i brzinu rada, mogućnost upravljanja hardverskim resursima
    - C – prvi jezik projektovan za sistemsko programiranje – nastao i primenje za razvoj OS UNIX
- Internet i Web
    - Jezivi za distribuirano programiranje
    - Jezici za formatiranje teksta na Web stranici
        - Markup jezici – HTML, XML
    - Jezici za definisanje sadržaja Web stranica
        - Skript jezici – JavaScript, Ruby, PERL, PHP, TypeScript...

4. Podela programskih paradigmi i predstavnik svake.

- Programska paradigma definiše stil programiranja
- Najgrublja podela:
    - Proceduralne paradigme – cilj je da programer tačno opiše algoritam (način) rešavanja problema
    - Konceptualne paradigme – zadatak programera je da precizno opiše problem, koji se rešava, a mehanizmi programkog jezika treba da obezbede način njegovor rešavanja
- Osnovne programske paradigme:
    - Imperativna programska paradigma [Fortran, Cobol, Algol, PL/1, Baic, Pascal, C...]
        - Program se sastoji od skupa podataka, koji se obrađuju i postupaka (algoritama), kojima se oni obrađuju
        - Koncepti koje imperativni jezici uvode:
            - Tipove podataka – za predstavljanje podataka koji se obrađuju
            - Programske strukture – za predstavljanje algoritama
    - Objektno-orijentisana programska paradigma [C#, Java, C++, Simula 67, SmallTalk, Eiffel...]
        - Softver se posmatra kao mreža objekata koji međusobno komuniciraju razmenom poruka
        - Objekti se modeluju korisničkim tipovima podataka - klasama
    - Funkcionalna programska paradigma [Lisp, Scheme, Haskell, ML, Scala, Ocaml...]
        - Ceo program se svodi na definisanje i evaluaciju matematičkih funkcija
        - Nema bočnih efekata (izlaz funkcije zavisi iključivo od ulaza funkcije)
        - Nema programskih struktura – petelje bivaju zamenjene rekurzivnim funkcijama
    - Logička programska paradigma [Prolog, Datalog, CLP, ILOG, Solver, ParLog, LIFE]
        - Bazirana na principu matematičke logike
        - U programu se definišu:
            - Činjenice
            - Pravila zaključivanja
            - Upiti
        - Izvršenje programa podrazumeva traženje odgovora na upite korišćenjem datih činjenica i pravila zaključivanja

5. Komponente u opisu programskih jezika

- Programski jezik definišu:
    - Azbuka – neprazan skup simbola od kojih se prave reči jezika

        - Velika i mala slova engleske azbuke
        - Dekadne cifre
        - Specijalni znaci
    - Leksika – nauka o rečima:
        - U programskim jezicima definiše vrste reči, koje se u programskom jeziku koriste, i kako se one grade
        - Definiše se formalno, pomoću regularnih izraza
    - Sintaksa – nauka o rečenicama
        - U programskim jezicima definiše strukturu programa
        - Definiše se formalno pomoću beksonteksnih gramatika
    - Semantika – nauka o značenju rečenica
        - U programkim jezicima definiše značenje trukturnih elemenata koje sintaksa definiše
        - Definiše se neformalno, pomoću govornog jezika

6. Vrste reči u programskim jezicima (naziv i primena)

- Ključne reči
- Simbolička imena (identifikatori)
    - Služe za imenovanje različitih entiteta (simboličkih konstanti, promenljivih, funkcija, tipova...) koje programeri definišu u kodu
    - Niz karaktera sastavljen od velikih i malih slova, cifara i simbola '_' i '$' u kojem prvi znak ne može biti cifra
        [A-Z, a-z, _, $][A-Z, a-z, 0-9, _, $]*
- Konstante (literali)
    - Vrednosti različitih tipova – definisane za svaki elemntarni tip podataka
- Operatori
    - Specijalni znaci, ili kombinacije specijalnih znakova, koje služe za formiranje izraza
- Separatori
    - Specijalni znaci, koji razdvajaju logičke celine u kodu, tj. Različite sintaksne elemente
- Beli simboli i komentari
    - Delovi koda koje prevodilac ignoriše. Nemaju nikakvo intaksno, ni semantičko zančenje – nisu tokeni. Korite e da obezbede čitljivost i razumevanje koda

7. Šta su regularni izrazi?

- Regularni izrazi definišu moguće oblike (zapise) određene vrste reči
- Sastavljeni su od:
    - Simbola koji čine azbuku programskog jezika
    - Metasimbola – simbola sa specijalnim značenjem u regularnom izrazu

8. Šta su metasimboli?

- Simboli sa specijalnim značenjem u regularnom izrazu:
- () – kreiranje grupe
- [] – alternativa (izbor jednog iz skupa simbola navedenih unutar zagrada)
- - - koristi se za definisanje intervala simbola unutar alternative
- + - označava da se prethodni imbol, ili grupa može pojaviti jednom, ili više puta u zapisu
- * - označava da se prethodni simbol, ili grupa može pojaviti nijednom, ili više puta u zapisu
- ? – označava da se prethodni simbol ili grupa može, ali ne mora pojaviti u zapisu
- . – zamenjuje bilo koji simbol azbuke
- ~ - zamenjuje bilo koji niz karaktera do znaka ili grupe koja sledi
- | - izbor jedne od dve alternative

- \ - poništava specijalno značenje metakaraktera koji sledi (takođe, unutar alternative, ili između dvostrukih navodnika metakarakteri nemaju specijalno značenje)

9. Definicija sintakse, beskontekstna gramatika

- Sintaksa – nauka o rečeniacama
  - U programskim jezicima definiše strukturu programa
  - Definiše se formalno pomoću beskonteksne gramatike
- Beskonteksne gramatike predstavljaju uređenu četvorku, čiji su elementi :
  - Terminali (terminalni simboli) – Reči jezika, odnosno njihovi tokeni.
    - Primer: if, then, else, ID, CONST, +…
  - Neterminali (neterminalni simboli) – Pomoćne sintaksne promenljive, kojima se označavaju skupovi reči. Neterminali se uvode da bi se lakše definisale ispravne jezičke konstrukcije, kao i da bi se lakše definisala hierarhija programskog jezika.
    - Primer: Expression, Statement, IfStatement, WhileStatement…
  - Startni simboli – Neterminal iz kojeg se izvode ispravne jezičke konstrukcije.
  - Produkciono pravilo . definiše način na koji se kombinuju terminalni i neterminalni simboli da bi se kreirale složenije jezičke konstrukcije. U opštem slučaju, pravila su oblika: X Y, gde je X neterminalni simbol, a Y niz sastavljen od terminalnih i neterminalnih simbola.
  - Primer: WhileStatement **while** (Expression) Statement
  - Načini predstavljanja beskonteksnih gramatika:
    - Bekusova normalna forma
    - BS notacija
    - Sintaksni dijagrami

10. Šta određuje tip podatka?

- Tip podatka određuje:
  - Skup vrednosti koje mogu biti predstavljene
  - Format registrovanja podataka
  - Skup operacija koje se nad podacima tog tipa mogu izvršavati

11. Podela tipova podataka prema složenosti

- Po strukturi, tipovi podataka se dele na:
  - Elementarne (primitivne)
    - Celi brojevi
    - Realni brojevi
    - Znakovi
    - Logički
  - Strukturne (složene) – sastavljene od više podataka elementarnog tipa
    - Polja
    - Enumeracije
    - Strukture
    - Klase
    - Interfejsi

12. Statička i dinamička tipizacija

- *Statička tipizacija*

- Podrazumeva da tipovi promenljivih ostaju konstantni za sve vreme izvršavanja programa.
- Svaka promenljiva mora biti deklarisana pre njenog korišćenja.
- Provera tipova se vrši u vreme prevođenja programa (compile-time type checking)
- Jezici sa statičkom tipizacijom:
  - C, C++, C#, Java

• *Dinamička tipizacija*

- Podrazumeva da su tipovi promenljivih poznati tek u vreme izvršenja programa I mogu se menjati u toku izvršenja programa.
- Promenljive se ne deklarišu u programu.
- Provera tipova se vrši u vreme izvršenja programa (run-time type checking)
- Jezici sa dinamičkom tipizacijom:
- Perl, PHP, Python, JavaScript

13. Slaba i jaka tipizacija

- Slaba tipizacija (weakly typed languages)
  - Informacija o tipu se koristi samo na mašinskom nivou, da bi se odredio format i veličina memorijskog prostora potrebnog da se podatak registruje.
  - Na nivou izvornog koda dozvoljava mešovite izraze pri čemu se vrši implicitna konverzija tipova podataka.
  - Jezici sa slabom tipizacijom: C, C++
  - Primer:
    - double x;
    - int i, j, k;
    - i = x; // moguć gubitak informacije, ali je ovo implicitna konverzija i kompajler neće prijaviti grešku
    - k = x * j; // ovo već zahteva eksplicitnu konverziju, ili izraza u int, ili x u int
- Jaka tipizacija (strongly typed langueages)
  - Tip podataka određuju sledeći elementi:
    - Skup vrednosti
    - Format registrovanja podataka
    - Skup operacija koje se nad podacima mogu izvršavati
    - Skup funkcija za konverziju – uspostavljanje veze sa drugim tipovima podataka
  - Sve definicije tipa moraju da budu javne i eksplicitne. Nisu dozvoljene implicitne definicije tipova.
  - Promenljivoj se definiše samo jedan tip.
  - Dozvoljeno je dodeljivanje vrednosti samo odgovarajućeg tipa.
  - Nad podacima su dozvoljene samo operacije obuhvaćene tipom kojem pripadaju.
  - Tip je zatvoren u odnosu na skup operacija koji obuhvata. Te operacije mogu se primenjivati sao nad oprandima tog tipa podataka. Mešoviti izrazi nisu dozvoljeni.
  - Dodeljivanje vrednosti raznorodnih tipova i operacije nad raznorodnim operandima moguće su samo uz eksplicitnu konverziju tipova.

- Programski jezici sa jakom tipizacijom:
  - C#, Java, Python...

14. Polja

- Kolekcija elemenata istog tipa
- Elementima polja se pristupa korišćenjem njihove pozicije (indeksa) u polju, pa se zato često nazivaju i indeksne strukture
- Po definiciji, to je statička struktura podataka (broj elemenata polja se ne menja u toku njegovog životnog ciklusa)

15. Polja u C

- Polja mogu biti i u tatičkoj i u dinamičkoj zoni memorije
- Polja u dinamičkoj zoni memorije:
  - Broj elemenata polja je poznat u fazi izvršenja programa i tada se rezerviše prostor za njegove elemente u dinamičkoj zoni memorije
  - Za pristup podacima u dinamičkoj zoni memorije u programskom jeziku C se koriste pokazivači:
    - Specijalni tipovi podataka koji pamte adrese drugih podataka u memoriji
    - Najbitnija primena im je da pamte adrese podataka koji e kreiraju u toku izvršenja programa u dinamičkoj zoni memorije.
- Primer:
  - int *niz, velicina;
  - scanf(″%d″, &velicina);
  - niz = (int*) malloc(velicina * sizeof(int));
- Brisanje polja iz dinamičke memorije:
  - free(niz);
- Moguća je promena veličine niza, ukoliko se on nalazi u dinamičkoj zoni memorije:
  - Realloc(niz, velicina2 * sizeof(int));

16. Polja u C++

- Mogu biti smeštena i u statičkoj i u dinamičkoj zoni memorije
- Ne može im se menjeti veličina tokom životnog ciklusa
- Primer:

```
int niz[50];
int *niz=new int [50];
```

- Višedimenziona polja:
  - U statičkoj zoni memorije

    ```
    int matrica[n][m];
    ```
  - U dinamičkoj zoni memorije:
    - Svaka vrsta poseban niz (ne moraju biti iste dužine)
    - Potreban niz pokazivača na vrste
    - Potreban i pokazivač na pokazivače na vrste
    - Kriranje:

      ```
      int n,m;
      int ** matrica;
      matrica =new int*[n];
      for( int i = 0; i < n; i++ )
              matrica[i] = new int[m];
      ```

- Brisanje:

```
for( int i = 0; i < n; i++ )
        delete[] matrica[i];
delete[] matrica;
```

17. Polja u programskom jeziku C#

- Referentni tipovi podataka
  - Smešteni uvek u dinamičkoj zoni memorije
  - Pristupa im se korišćenjem reference
  - To su statičke strukture podataka (ne može se menjati veličina tokom životnog ciklusa)
- Deklaracija promenljive tipa polja:
  <tip>[] <imePolja>
- Kreiranje polja:
  <imePolja> = new <tip>[<veličina>];
- Deklaracija promenljive niza objekta:
  <ImeKlase>[] <imePolja>;
- Kreiranje polja:
  <imePolja> = new <ImeKlase>[<veličina>];
- Kreiranje objekata:
  <imePolja>[<pozicija>]=new <ImeKlase>();
- Atribut polja:
  - Lenght – predstavlja broj elemenata polja
- Višedimenziona polja (nizovi nizova):
  - <tip> [,] <imePolja> int [] [] matrica;
  - Kreiranje pravougaonog dvodimenzionog polja
    <imePolja> = new <tip> [<brojVrsta>, <brojKolona>];
  - Kreiranje nazubljenog polja je identično kao i u Javi

18. Polja u Javi

- Referentni tipovi podataka
  - Smešteni uvek u dinamičkoj zoni memorije
  - Pristupa im se korišćenjem reference
  - To su statičke strukture podataka (ne može se menjati veličina tokom životnog ciklusa)
- Deklaracija promenljive tipa polja:
  <tip>[] <imePolja> ili <tip> <imePolja>[]
- Kreiranje polja:
  <imePolja> = new <tip>[<veličina>];
- Deklaracija promenljive niza objekta:
  <ImeKlase>[] <imePolja>; ili <ImeKlase> <imePolja>[];
- Kreiranje polja:
  <imePolja> = new <ImeKlase>[<veličina>];
- Kreiranje objekata:
  - <imePolja>[<pozicija>]=new <ImeKlase>()
- Atribut polja:
  - Lenght – predstavlja broj elemenata polja
- Višedimenziona polja (nizovi nizova):
  - <tip> [] [] <imePolja> int [] [] matrica;
  - Kreiranje pravougaonog dvodimenzionog polja
    <imePolja> = new <tip> [<brojVrsta>] [<brojKolona>];
  - Kreiranje nazubljenog polja:

```
trougaonaMatrica = new int[3][];
for( int i = 0; i < 3; i++ )
        trougaonaMatrica[i] = new int [i+1];
```

19. Enumeracije

- Enumeracije su nabrojivi tipovi podataka
- Promenljiva tipa enumeracije može uzti jednu od vrednosti navedenih u definiciji tipa.
- Enumeracija se može shvatiti i kao skup celobrojnih simboličkih konstanti

20. Enumeracije u Javi

- Referentni tip podatka
- Tip vrednosti može biti proizvoljan (isti za sve članove)
- Definicija enumeracije:

```
[<pravo pristupa>] enum <ImeEnumeracije>
{
        Name_1 [(<values_1>)],
    Name_2 [(<values_2>)],
    …
    Name_n [(<values_n>)];
    [<definicije_metoda]
            };
```

21. Enumeracije u C i C++

- Definicija enumeracije:

```
enum{
        <name1>[=<value1>],
        <name2>[=<value2>],
        ...
};
```

- Ukoliko inicijalizacija konstante nije navedena, podrazumeva se da prva ima vrednost 0, a vrednost svake sledeće je za 1 veća od vrednosti prethodne navedene konstante.

22. Strukture

- Složeni tipovi podataka sastavljeni od elemenata različitog tipa.
- Svaki član strukture ima svoje ime – služi za pristup članu strukture

23. Strukture u C

- Definicija strukture :

```
struct <imeStrukture>
{
    <tip1> <ime1>;
    <tip2> <ime2>;
    ...
};
```

- Definicija promenljive tipa strukture:
- Pristup članovima strukture:

24. Strukture u C#

- Definicija strukture:

```
[<pravoPristupa>] struct <imeStrukture>
{
    // definicije tipova
    // definicije atributa
    // definicije metoda
    // definicije svojstava
};
```

- Razlike u odnosu na klase:
  - Struktura je VREDNOSNI tip
  - Strukture se ne nasleđuju
  - U strukturi se vrednosti atributa ne inicijalizuju (osim ako nisu konstantni ili statički)

25. Strukture u C++

  - Kao i klase, služe za predstavljanje objekata u programu.
  - Razlika je samo u podrazumevanom pravu pristupa, za članove struktra je to PUBLIC.

26. Strukture u Javi

- NE POSTOJE!

27. Imperativna paradigma

- Program predstavlja skup naredbi koje menjaju njegovo stanje
- Naredbama se opisuje algoritam za rešavanje problema
- Poreklo imena:
  - Imperativ – glagolski oblik kojim se iskazuje naredba, poznat kao "zapovedni način"
- Naredbe u imperativnim jezicima:
  - Naredbe obrade (obično izrazi dodele)
  - Upravljačke strukture

28. Podela programske strukture i pojam strukturnog programiranja

- Programska (upravljačka) struktura je instrukcija programa koja određuje redosled izvršavanja drugih instrukcija programa.
- Osnovne programske(upravljačke) strukture su:
  - Sekvenca – predstavlja niz intrukcija u programu koje se izvršavaju onim redosledom, kako su zapisane
  - Selekcija – na osnovu ispunjenosti ili neispunjenosti određenog uslova određuje koji će deo koda biti izvršen
  - Petlja – definiše ponavljanje izvršenja određenog skupa instrukcija
- Strukturno programiranje predstavlja način programiranja u kom programske strukture se mogu ugnježdavati, ali ne i preklapati.

29. Opseg važenja imena

- Blokovi se često koriste i za ograničavanje opsega važenja simboličkih imena
  - Simboličko ime definisano u okviru jednog bloka je lokalna za taj blok, a globlana za unutrašnje blokove (blokove definisane unutar njega)
- Problem nastaje kada se isto ime deklariše više puta

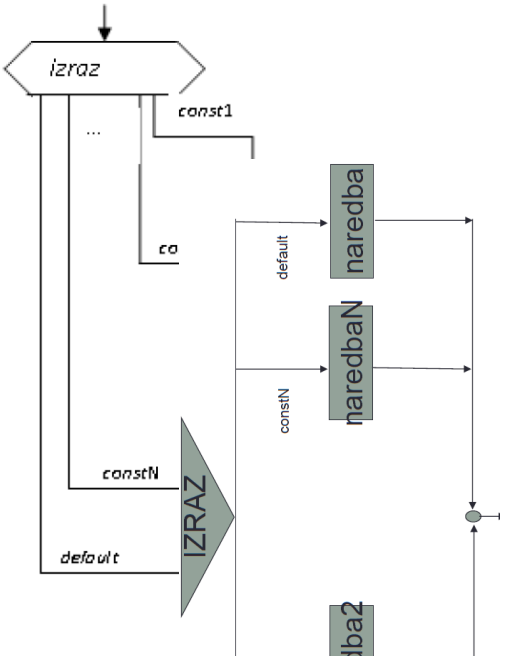
```
A: {
    int A; B: {float A; }
    }
```

- Razrešavanje:
  - C, C++: Važi posldenja definicija imena (u bloku najbližem tački korišćenja)
  - C#, Java: Kompajler prijavljuje grešku

30. Strukture grananja u Javi i C#

- Naredba selekcije (if naredba) omogućava da se izvrši jedan ili drugi deo programa zavisno od toga da li je neki uslov ispunjen ili nije.
- Probelm ugnježdavanja struktura grananja
  - If (B1) if (B2) Sa else Sb
  - U ovoj naredbi nije potpuno jasno da li se Sb izvršava kada je B1= false, ili kada je B1=true I B2=false (što je uvom slučaju tačno).
- Višestruko grananje u C:

```
switch (<izraz>)
{
    case <const1>: <naredba1>
    case <const2>: <naredba2>
            ...
    case <constN>: <naredbaN>
    [default: <naredba>]
}
```



- Struktura češlja u C:

```
switch (<izraz>)
{
    case <const1>: <naredba1> break;
    case <const2>: <naredba2> break;
            ...
```

```
    case <constN>: <naredbaN> break;
    [default: <naredba>]
}
```

- U Javi
  - Switch struktura ima potpuno identičnu semantiku kao I u C-u.
- U C#
  - Switch struktura je namenjena kreiranju strukture tipa češlja
    - Break naredba je obavezna nakon svake case klauzule, pa čak I nakon klauzule default

31. Programske petlje u Javi i C#

- Omogućavaju višestruko izvršavanje istog bloka naredbi programa više puta.
- Vrste programskih petlji:
  - Brojačke petlje – petlje sa unapred poznatim brojem ponavljanja
  - Petlje sa unapred nepoznatim brojem prolaza (ponavljaju se dok važi određeni uslov):
    - Petlje sa uslovom na početku (while-do)
    - Petlje sa uslovom na kraju (do-while/repeat-until)
- Brojačke petlje
  - Koriste specijalnu promenljivu, nazvanu brojačom petlje, koja pri svakom izvršenju naredbi u petlji, menja svoju vrednost, počevši od startne vrednosti, do konačne vrednosti za određeni inkrement.
  - Često se koristi i za obilazak kolekcija (foreach):
    - Java:
      - For (<variable> : <collection>) <statement>
      - Ograničenje:
        - Kolekcija mora da bude:
          - Polje, ili
          - Klasa kojaimplementira interfejs Collection
    - C#:
      - Foreach (<variable> in <collection>) <statement>
      - Ograničenje:
        - Kolekcija može da bude:
          - Polje, ili
          - Klasa koja implementira jedan od intefejsa:
            - System.Collections.IEnumerable
            - System.Collections.IEnumerator
            - System.Collections.Generic.IEnumerable<T>
            - System.Collections.Generic.IEumerator<T>
- Ptelje sa nepoznatim brojem prolaza
  - Uslov ponavljanja petlje se može naći na početku (telo petlje je moguće da se nikad ne izvrši), ili na kraju petlje (telo petlje se izvršava barem jednom).

32. Petlje za obilazak kolekcija

- Za obilazak kolekcija može da se koristi brojačka petlja, međutim petlja namenjena za obilazak kolekcija jeste foreach petlja
  - U javi:
    - For (<variable> : <collection>) <statement>
  - U C#:

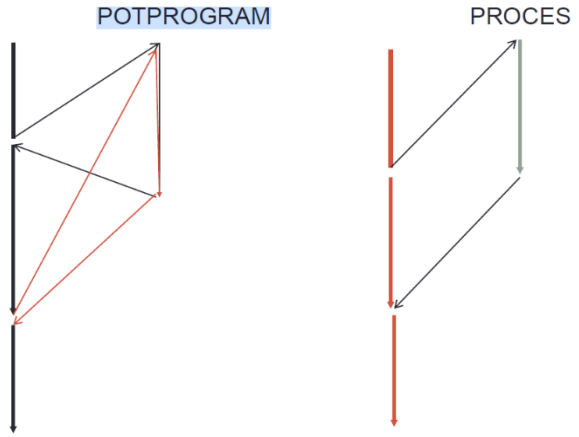
- Foreach (<variable> in <collection>) <statement>

33. Naredbe prekida u Javi i C#

- CONTINUE
  - Za prekidanje tekuće iteracije petlje
  - Java ima i mogućnot prekidanja izvršenja tekuće iteracije obeležene petlje:
    continue labela;
- BREAK
  - Za prekidanje tekuće programske strukture
  - Java ima i mogućnot preidanja izvršenja tekuće iteracije obeležene petlje
    break labela;
- RETURN [izraz]
  - Za prekidanje izvršenja tekućeg programa
- THROW [izraz]
  - Za prekidanje tekuće trukture, niza ugnježdenih struktura, tekuće funkcije ili niza pozvanih funkcija

34. Pojam procesa. Razlika između procesa i potprograma.

- Proces, nit (thread), ili task je deo programa koji se može izvršavati konkurentno sa drugim procesima.
- Razlika između potprograma i procesa:
  - Proces se mora eksplicitno startovati.
  - Kada neki programski modul pokrene novi task, njegovo izvršenje se ne prekida – nastavljaju da se izvršavaju zajedno.
  - Kada se izvršenje procesa završi, upravljanje ne mora da se vrati modulu, koji ga je kreirao i pokrenuo.



35. Vrste parametara potprograma

- Semantički posmatrano prenos parametara u ptprogram može da bude po jednom od tri semantička modela
- Program može da:
  - Od glavnog programa primi vrednost parametara (ulazni)
  - Da mu preda vrednost (izlazni)
  - Da primi vrednost i preda rezultat glavnom programu (ulazno-izlazni)

36. Prenos parametara

- U teoriji programskih jezika najčešće korišćene su dve metode prenosa parametara:

37. Penos po vrednosti (Call by value)

- U trenutku poziva se rezerviše novi prostor u memoriji za fiktivni parametar i u taj prostor upisuje vrednost stvarnog parametra.
- Ovaj način prenosa se obično sreće kod funkcijskih potprograma. Sastoji se u tome što se pri pozivu funkcije vrednosti stvarnih argumenata kopiraju u pomoćne memorijske lokacije fiktivnih argumenata, koje su sastavni deo potprograma. Sve promene argumenata odvijaju se u lokacijama potprograma i nedostupne su glavnom programu. Zbog toga se ova tehnika prenosa može koristiti samo za prenos ulaznih parametara potprograma, a ne i za vraćanje rezultata glavnom programu.
- Osnovni problem prenosa po vrednosti je u tome što se za formalne parametre potpgrorama rezerviše memorijski prostor. Uz to i sama operacija kopiranja vrednosti može da utiče na efikasnost programa. Ovi nedostaci dolaze do izražaja, ukoliko se prenose strukture podataka, kao na primer vektori i matrice.

38. Prenos pro referenci (Call by reference)

- Fiktivni parametar je drugo ime za stvarni parametar, koji se navodi u pozivu

- Kod ovog načina prenosa nema kopiranja vrednosti parametara iz glavnog programa u potprogram, ili obrnuto. Potprogramu se samo prenosi referenca(obično samo adresa) memorijske lokacije u kojoj je vrendost stvarnog parametra. Efekat je kao da se reference stvarnih argumenata preslikavaju u reference fizičkih. Potprogram pristupa istim memorijskim lokacijama kojima i glavnim program, nema rezervisanja memorijskih lokacija za fiktivne argumente u okviru potprograma. Sve promene nad argumentima koje se izvrše u okviru potprograma vidljive su i glavnom programu. Zbog toga je ovo način prenosa koji se koristi za prenos ulazno-izlaznih argumenata. U mnogim programskim jezicima to je osnovni način prenosa argumenata i podrazumeva se implicitno.
- Prednost su efektivnost, kako u pogledu vremena, tako i u pogledu memorijskog prostora.
- Neodstaci su ti da se za adresiaranje koristi indirektno adresiranje, kao i neki bočni efekti, kao na primer kolizija.

39. Prenos po rezultatu (Call by result)

- Prenos po rezultatu je tehnika koja se primenjuje za prenos izlaznih(out) parametara. Kada se argumenti prenose po rezultatu, njihove rednosti se izračunavaju u lokalnim meorijskim lokacijama formalnih argumenata potprograma (nema prenosa vrednosti iz glavnog programa u potprogram). Međutim, pre vraćanja upravljanja glavnom programu, ove vrednosti se kopiraju u memorijske lokacije stvarnih argumenata. Stvarni argumenti u ovom slučaju moraju da budu promenljive. Kao i kod prenosa po vrednosti, za fiktivne argumente potprograma rezervišu se lokalne memorijske lokacije.
- Kod ovog načina prenosa mogući su međutim i neki negativni efekti kao što je na primer kolizija stvarnih argumenata, koja nastaje u slučajevima kada se potprogram poziva tako da isti stvarni argument zamenjuje više fiktivnih argumenata. Neće se znati odakle treba kopirati vrednosti na memorijsku lokaciju stvarnog argumenta.

40. Prenos po vrednosti I rezultatu (Call by Value – Result / Call by Copy)

- Prenos po vrednosti I rezulatu je način prenosa ulazno-izlaznih argumenata sa kopiranjem vrednosti argumenata. To je u suštini kombinacija prenosa po vrednosti I prenosa po rezultatu. Vrednosti stvarnih argumenata kopiraju se u memorijske lokacije fiktivnih argumenata potprograma. To su lokalne memorijske lokacije fiktivnih argumenata potprograma i sve izmene nad argumentima pamte se u ovim lokacijama. Pre vraćanja upravljanja glavnom programu, rezultati potprograma se iz ovih lokalnih memorijskih lokacija ponovo kopiraju u memorijske lokacije stvarnih argumenata. Ovaj način prenosa se često naziva i prenos kopiranjem (Call by Copy), jer se stvarni argumenti najpre kopiraju u potprogram, a zatim ponovnim kopiranje rezultati vraćaju u glavni program.
- Nedostaci i negativni efekti su isti kao i za Call by value I Call by result.

41. Prenos po imenu

- Prenos po imenu je način prenosa ulazno-izlaznih argumenata potprograma koji se razlikuje od prethodnih po tome što nema ni kopiranja vrednosti ni prenosa reference, već se pri pozivu potprograma imena stvarnih argumenata ubacuju u tekst potprograma svuda umesto imena fiktivnih parametara. Potprogram se izvršava kao da se radi o kodu koji je napisan sa imenima stvarnih argumenata. Ovde se odvija jedno zakasnelo povezivanje u trenutku poziva potprograma. Implementacija prenosa po imenu zavisi od toga šta se prenosi kao stvarni argument. Ako je stvarni argument skalarna pormenljiva, onda je prenos po imenu isti kao i prenos po referenci. Ako je stvarni argument konstanta, onda se prenos vrši kao u slučaju prenosa po vrednosti. Ako se kao stvarni argument prenosi element niza, onda se ovaj prenos razliku od svih prethodnih i može imati specifične efekte.

42. Prenos parametara u Javi

- Postoji samo prenos parametara po vrednosti!
- Tipovi podataka u Javi:
  - Vrednosni
    - Svi primitivni tipovi podataka

    - Referentni
        - Polja, Enumeracije, Klase, Interfejsi
- Prenos parametra vrednosnog tipa
    - Pri prenosu se rezerviše novi prostor za vrednost parametra I tu upisuje vrednost stvarnog parametra.
    - Izmena vrednosti parametra se ne odražava na stvarni argument.
- Prenos parametra referentnog tipa
    - Prilikom poziva funkcije se kreira nova referenca, ali ne I novi objekat.
    - Promena objekta u metodi, znači promenu originalnog objekta.
    - Ukoliko se referentnom parametru dodeli drugi objekat u metodi, promena tog objekta se neće videti u pozivajućem modulu.

43. Prenos parametara po referenci vrednosnih i referentnih tipova u C#

- Prenos parametara vrednosnog tipa po referenci:
    - Izlazni i ulazno-izlazni
    - Promene parametara unutar metode se vide u pozivajućem modulu
    - Štedi se memorijski prostor
    - Ulazni parametar se ne može koristiti u metodi pre nego što mu se ne dodeli vrednost
    - Stvarni parametar u pozivu može biti samo promenljiva
- Prenos parametara referentnog tipa po referenci:
    - Izlazni i ulazno-izlazni parametar
    - Koristi se ista referenca kao i u pozivajućem modulu
    - Metoda može parametru da dodeli sasvim drugi objekat
    - Ukoliko je parametar označen kao OUT, stvarni argument može imati i vrednost NULL pre poziva

44. Vrste parametara u C#. Načini na koji se prenose te vrste.

- Parametri u programskom jeziku C# mogu biti:
    - Ulazni – podrazumevani, navode se bez eksplicitnog navođenja vrste prenosa
    - Izlazni – ispred definicije paramtera stoji ključna reč *out*
    - Ulazno–izlazni – ispred definicije parametra stoji ključna reč *ref*
- Parametri vrednosnog tipa:
    - Ulazni parametri se prenose po vrednosti
        - Pravi se kopija podataka u memoriji
        - Promene parametra unutar metode se ne vide van nje
        - Kao stvarni parametri se mogu koristiti promenljive, konstante ili izrazi odgovarajućeg tipa
    - Izlazni i ulazno–izlazni se prenose po referenci
        - Promene parametara unutar metode se vide u pozivajućem modulu
        - Štedi se memorijski prostor
        - Ulazni parametar se ne može koristiti u metodi pre nego što mu se dodeli vrednost
        - Stvarni parametra u pozivu može biti samo promenljiva
- Parametri referentnog tipa:
    - Ulazni parametar – prenos po vrednosti

        - Kreira se kopija reference, ali ne i podatka
        - Metoda može da menja i objekat (ta promena će biti vidljiva spolja), ali ne može parametru da dodeli neki novi objekat
    - Izlazni i ulazno – izlazni parametri – prenos po referenci
        - Koristi se ista referenca kao i u pozivajućem modulu
        - Mreoda može parametru da dodeli sasvim drugi objekat
        - Ukoliko je parametra označen sa out, stvarni argument može imati i ulaznu vrednost null.

45. Prenos parametara u C++

- Po vrednosti
    - U trenutku poziva rezerviše se novi prostor u memoriji za fiktivni parametar, i u taj prostor upisuje vrednost stvarnog parametra.
- Po referenci
    - Fiktivni parametar je drugo ime za stvarni parametar koji se navodi u pozivu
- Programer odlučuje da li se prenos parametara vrši po vrednosti ili po referenci

```
void f ( int i, int &j ) // i po vrednosti, j po referenci
```

46. Funkcije sa promenljivim brojem parametara u Javi i C#

- C# i Java omogućuju da se definišu funkcije sa proizvoljnim brojem parametara ISTOG TIPA.
- U C#:
    - Ključna reč params omogućava da se funkcija poziva sa različitim brojem parametara istog tipa.
    - U listi parametara samo poslednji može da ima ispred sebe ovu ključnu reč.
    - Nakon ključne reči params obavezno stoji niz
    - Funkcija takav parametar vidi kao niz, a u pozivu se navode nazavisni stvarni parametri
- U Javi:
    - Umesto ključne reči params, ispred niza tipa, u Javi stoji … iza tipa. Ponašanje je identično, smatra se nizom.

47. Podrazumevane vrednosti

- U nekim jezicima (C++, C#,…) se mogu navesti podrazumevane vrednosti argumenata.
- Definicija funckije sa podrazumevanim vrednostima parametara:

```
void f(float a = 0.5f, int k = 0)
```

- Podrazumevane vrednosti paarametara se koriste ukoliko u pozivu funkcije odgovarajući stvarni parametar nije naveden.
    - Dzovoljeni pozivi funkcije f:

```
f(15.5f, 2);
f(15.5f);
f(2);
f();
```

- Podrazumevane vrednosti mogu imati samo poslednji parametri u listi.
    - Dozvoljena definicija funkcije sa podrazumevanim vrednostima parametara:

```
void f(float a, int k = 0)
```

    - Nedozovljena definicija funckije sa podrazumevanim vrednostima parametara:

```
void f(float a = 0.5f, int k)
```

48. Rekurzivni potprogrami. Prednosti i mane.

- Rekurentni potprogrami su potprogrami koji u svom telu dozvoljavaju poziv sebe samog.
- Oni su sredstvo za rešavanje problema, koji su matematički rekurzivno definisani, poput Fibonačijevog niza, faktoriela, itd...
- Prednosti su te što su vrlo kratki i laki za ljudsko razumevanje, međutim, problem je u tome što zahtevaju veliki broj poziva, a to može biti vrlo zahtevno.

49. Fibonacci (rekurzivno i nerekurzivno)

```
long fibonacciReccursion(int n)
```

```
{
        if (n < 2)
                return 1;
        else
                return fibonacciReccursion(n - 1) + fibonacciReccursion(n - 2);
}
long fibonacciReccurenceRelation(int n)
{
        double c1 = (5 + sqrt(5)) / 10, c2 = (5 - sqrt(5)) / 10;
        double t1 = (1 + sqrt(5)) / 2, t2 = (1 - sqrt(5)) / 2;

        return c1 * pow(t1, n) + c2 * pow(t2, n);
}
long fibonacciLoop(int n)
{
        long f0, f1, fn;
        f0 = f1 = 1;
        for (int i = 1; i < n; i++)
        {
                fn = f0 + f1;
                f0 = f1;
                f1 = fn;
        }
        return fn;
}
```

50. Pojam aktivacionog sloga i sadržaj

1. Aktivacioni slog je deo memorije koji biva rezervisan pri pozivu potprograma i on sadrži:
   - Rezultate koje vraća potprogram – preko polja
   - Stvarne parametre
   - Adresu povratka
   - Upravljačke linkove – opcioni linkovi do aktivacionog sloga programa iz kojeg je pozvan potprogram
   - Linkove za pristup podacima iz okoline – opcioni linkovi do nelokalnih podataka koji su sadržani u drugim aktivacionim slogovima
   - Kontekst procesora – polje gde se pamti status procesora prilikom prenošenja upravljanja na potprogram
   - Lokalne podatke (potprograma)
   - Privremene promenljive – Promenljive koje generiše kompilator prilikom evaluacije aritmetičkih izraza

51. Strategije za alokaciju memorije

- Statička alokacija svih objekata u vreme kompilacije
- Alokacija pomoću stack-a
- Dinamička alokacija pomoću heap-a
- Opširnije:
  - Prilikom prevođenja programa, kompilator vrši planiranje memorijskog prostora, koji će biti dodeljen programu (Storage allocation). Pored toga što treba da se rezerviše memorijski prostor za smeštaj koda programa i podataka koji se obrađuju u njemu, za svaki od potprograma koji se poziva iz programa generiše se i jedan aktivacioni slog u koji se smeštaju podaci koji se preuzimaju od glavnog programa, lokalni podaci potprograma i takozvane privremene promenljive koje generiše sam kompilator prilikom generisanja mašinskog koda. Struktura jednog aktivacionog koda: (pitanje 50)
  - Za smeštaj aktivacionih slogova se koristi jedna od tri strategije:
    1. Statička alokacija
    2. Alokacija uz pomoć STACK-a

3. Alokacija uz pomoć HEAP-a (dinamička)
   - U zavisnosti od toga koja je strategija primenjena, zavisi da li će ili ne biti moguć rekurzivni poziv potprograma.

52. Objasniti statičku alokaciju

- U toku kompiliranja programa generiše se po jedan aktivacioni slog za svaki potprogram tako da se isti aktivacioni slog koristi kod svakog poziva potprograma.
- Nema mogućnosti za rekurzivne pozive procedura
- Strukture podataka se ne mogu kreirati dinamički
- Primenjuje se kod implementacije jezika Fortran 77
- Opširnije :
  - Kod statičkog smeštaja aktivacionih slogova kompilator unapred rezerviše fiksan memorijski prostor, čiji se sadržaj obnavlja kod svakog poziva potprograma. To znači da kod ovakog smeštaja nisu mogući rekurzivni pozivi, ker se lod svakog novog poziva potprograma gubi informacija o prethodnom pozivu. Ovakva tehnika primenjuje se na primer kod realizacije kompilatora za programski jezik FORTRAN, pa se u nejmu zvog toha ne mogu koristiti rekurzije

53. Alokacija pomoću stack-a

- Memorija predviđena za pamćenje aktivacionih slogova je organizovana kao stack
- Prilikom svakog poziva potprograma u stek se ubacuje njegov aktivacioni slog.
- Slog se izbacuje iz steka kada se potprogram završi
- Ova tehnika omogućava rekurzivne pozive potprograma. U tom slučaju u steku se nalazi više aktivacionih slogova istog potprograma i svaki sadrži odgovarajuće podatke.
- U vreme kompiliranja programa zna se samo veličina aktivacionog sloga, ali ne i dubina rekurzije (koliko aktivacionih slogova jednog potprograma će biti generisano)
- Opširnije:
  - Kod strategije koja je zasnovana na primeni steka, za smeštaj aktivacionih slogova koristi se stek u koji se za svaki poziv potprograma smešta jedan aktivacioni slog. Ova tehnika dozvoljava rekurziju jer se kod svakog poziva potprograma generipe novi slog i smešta u stek. Na kraju potprograma, slog se izbacuje iz steka. To znači da je u toku izvršavanja programa ovaj stek dinamički zauzet aktivacionim slogovima onih potprograma koji su trenutno aktivni. Od veličine memorijskog prostora koji je dodeljen steku zavisi I dubina rekurzivnih poziva potprograma.

54. Alokacija pomoći HEAP-a

- Dinamička strategija sastoji se u tome da se za svaki poziv potprograma generiše aktivacioni slog koji se smešta u poseban deo memorije nazvan Heap. U ovom slučaju aktivacioni slogovi potprograma povezuju se u dinamičku strukturu podataka koja odgovara stablu poziva potprograma. Ova tehnika dozvoljava rekurziju.

55. Objasniti razliku između alokacije memorije pomoću stack-a i heap-a

- Aktivacioni slogovi se smeštaju u dinamičku memoriju i ne izbacuju se kao kod stek alokacije po završetku potprograma.

56. Objektno-orijentisana paradigma

- Svi podaci koji se u programu obrađuju predstavljaju objekte.
- Objekat je definisan podacima koji opisuju njegovo stanje i skupom metoda koje opisuju njegovo ponašanje
- Objekti komuniciraju međusobno slanjem poruka (pozivanjem metoda)
- Svaki objekat pripada svojoj klasi (ima svoj tip).

57. Pojam apstrakcije podataka

- Konceptualno pojednostavljanje prikaza različitih entiteta pri čemu se namerno ignorišu detalji, a ističu globalna svojstva
- Koncept apstrakcije je fundamentalan u programiranju
  - Apstrakcija procesa
    - Korišćenjem potprograma – potprogram definiše neki proces, a njegov poziv je apstrakcija tog procesa, korisnik potprograma nije upoznat sa detaljima implementacije

- Apstrakcija podataka
  - Apstraktni tipovi podataka su korisnički definisani tipovi, koji zadovoljavaju sledeće uslove:
    - Reprezentacija podataka tog tipa je sakrivena od delova programa koji te podatke koriste. Nad podacima top tipa mogu da se primenjuju samo operacije definisane u tipu.
    - Deklaracija podataka i operacija koje su nad njima primenljive se definišu u jednistvenom modulu, drugi programski moduli mogu samo da kreiraju promenljive definisanog tipa. (Enkapsulacija)
  - Sredstva za apstrakciju podataka:
    - U proceduralnim programskim jezicima:
      - Moduli (podržavaju module: Pascal, Modula, ADA (Objektno-orijentisani jezik, ali podržava module)
        - Sadrže:
          - Interfejs – definicije tipova podataka koji se u umodulu obrađuju i deklaracije potprograma koji te podatke obrađuju.
          - Implementaciju – sadrži implementaciju potprograma definisanih u interfejsu.
    - U objektno-orijentisanim programskim jezicima
      - Klase

58. Klase kao apstrakciono sredstvo

- Enkapsulacija:
  - Klasa sadrži skup podataka koji opisuju stanje objekta i metode koje se nad objektom mogu izvršavati
- Sakrivanje podataka
  - Za sve članove klase je definisano pravo pristupa:
    - Public
    - Protected
    - Private
  - Nepisano pravilo je da svi podaci treba da budu privatni.
- Definicija klase je odvojena od implementacije
- Pravima pristupa se postiže sakrivanje podataka

59. Načini pristupa u Javi i C#

- Pored osnovnih prava pristupa :
  - Public – dostupnost globalno
  - Protected – dostupnost u matičnoj klasi i u nasleđenim klasa
  - Private – dostupnost samo u matičnoj klasi
- Postoje i specifična prava pristupa
- Java:
  - Default – pravo pristupa na nivou paketa
    - Sve definisano u okviru jednog paketa se smatra “prijatljeskim”
    - Sve definisano u okviru jednog paketa čemu nije eksplicitno deodeljeno pravo pristupa se može koristiti u svim funkcijama članicama svih klasa definisanih u okviru istog paketa
- C#:
  - Internal – članovi kalsa vidljivi u okviru istog asemblija (assembly), ovo je podrazumevano pravo pristupa

- Protected internal – članovi vidljivi u okviru istog asemblija I u izvedenim klasama

60. Nasleđivanje u C++, Javi i C#

- Nasleđivanje omogućava da se definišu nove klase na osnovu postojećih
- Nova klasa se definiše kao specijalizacija, ili proširenje neke klase koja već postoji
- Nova klasa inicijalno ima sve atribute i metode klase koju nasleđuje
- Novoj klasi mogu da budu pridodati novi atributi i nove metode
- Metode mogu biti izmenjene sa ciljem da se nova klasa prilagodi specifičnim potrebama
- Definicija izvedene klase:
  - C++:
    class <ClassName> : <Parent1> [, <Parent2>]...
    - Gde je definicija roditeljske klase (<ParentK>):
      $\text{access-specifier}_{\text{optional}}$ $\text{virtual}_{\text{optional}}$ <ParentClassName>
  - Java:
    class <ClassName> extends <ParentClassName>
  - C#:
    class <ClassName> : <ParentClassName>
- Načini izvođenja:
  - C++:
    - Public:
      - Objekat izvedene klase JE I objekat roditeljske klase (IS A)
    - Protected
    - Private
      - Objekat roditeljske klase JE DEO objekat roditeljske klase (PART OF)
  - Java / C#
    - Samo jedan način izvođenja, koji odgovara javnom izvođenju u C++
    - Kada je potrebno da objekat jedne klase bude deo druge, definiše se kao atribut.

61. Polimorfizam

- Termin označava "više formi" – "many formas", I u kontekstu objektnih jezika ukazuje na mogućnost da se različiti metodi pozivaju preko istog interfejsa
- Sposobnost da različiti objekti odgovore na iste poruke na sebi svojstven način.
- Obeležavanje polimorfnih funkcija:

Programski jezik	C++	Java	C#
U osnovnoj klasi	virtual	-	Virtual
U izvedenoj klasi	-	-	Override

- Aktiviranje polimorfizma pri pozivu funkcije

Programski jezik	C++	Java	C#
Objekat	-	-	-
Pokazivač	+	-	-
Referenca	+	+	+

62. Apstraktne metode i klase. Definicije u C++, C# i Javi

2. Apstraktna metoda nema implementaciju.
3. Apstraktne klase sadrže bar jednu apstraktnu metodu
4. Ne mogu se kreirati objekti apstraktnih klasa
5. Služe kao kalup za pravljenje novih klasa

Programski jezik	C++	Java	C#
Apstraktne klase	-	abstract	abstract
Apstraktne metode	virtual ... =0	abstract	abstract

Metode koje predefinišu apstraktne	-	-	override

63. Interfejsi u Javi

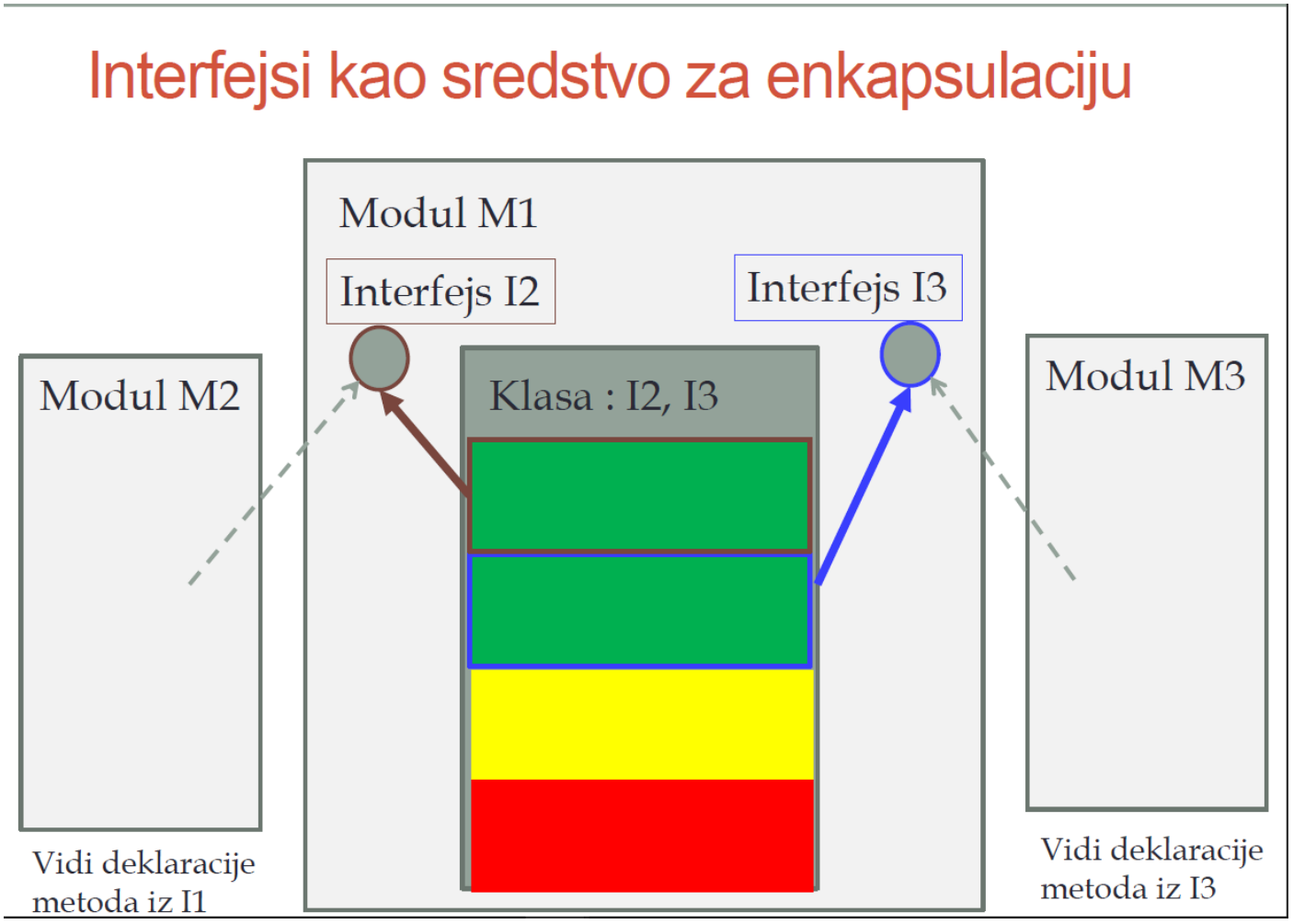
- Interfejs – definiše skup apstraktnih metoda koje izvedene klase treba da implementiraju
- Metode članice interfejsa su uvek aptraktne i javne
- U javi:
  - Članovi interfejsa mogu da budu atributi koji u obavezno javni, statički i konstantni
- Definicija interfejsa:
  interface <ImeInterfejsa> {...}
- Navođenje modifikatora PUBLIC i ABSTRACT ispred metoda u JAVI nije neophodno
- Definicija klase koja implementira interfejs u Javi:
  class <ImeKlase> implements <ImeInterfejsa> {...}

64. Interfejsi u C#

- Definišu skup apstraktnih metoda koje izvedene klase treba da implementiraju
- Metode članice interfjesa su uvek apstraktne i javne
- U C#:
  - Članovi interfejsa mogu biti i:
    - Svojstva (properties)
    - Delegati (delegates)
    - Događaji (events)
  - Definicija interfejsa:
    interface <ImeInterfejsa> {...}
  - Navođenje modifikatora public i abstract ispred metoda se u C# NE PIŠE!
  - Definicija klase koja implementira interfejs
    class <ImeKlase> : <ImeInterfejsa> {...}

65. Enkapsulacija preko interfejsa

- Enkapsulacija:
  - Predstavlja deklarisanje podataka i operacija (koje se nad njima mogu primenjivati) u jedinstvenom modulu, dok drugi programski moduli mogu samo da kreiraju promenljive definisanog tipa.
  - Klasa sadrži skup podataka koji opisuju stanje objekta i metode koje se nad objektom mogu izvršavati.
- Modul sadrži:
  - Interfejs – definicije tipova podataka koji se u moduu obrađuju i deklaracije potprograma koji podatke tih tipova obrađuju
  - Implementaciju – definicija potprograma deklarisanih u interfejsu modula
- Interfejs
  - Definiše skup apstraktnih metoda, koje izvedene klase treba da implementiraju
  - Metode članice interfejsa su uvek apstraktne i javne



66. Generičke metode i klase u C#

- Funkcije ili klase koje realizuju često korišćene algoritme ili strukture podataka sa nepoznatim tipovima podataka (a u C++-u i sa nepoznatim vrednostima nekih konstanti)
- Parametar templejta može biti samo tip
- Na mesto stvarnog parametra templejta se mogu koristiti primitivni tipovi
- U biblioteci klasa za predstavljanje kolekcija paralelno definisane generičke i negeneričke klase (klase, čiji su elementi itpa Object)
- Definicija generičke klase u C#-u:

```
class Ime<argumentiTemplejta> [<ograničenjaArgumenata>]
{
        ...
};
```

  - Definicija ograničenja:

    where <argumentTemplejta> : <ograničenje>

    - U ograničenjima parametara se pišu uslovi koje treba da zadovolje tipovi koji se koriste kao stvarni argumenti templejta.
    - Vrste ograničenja:
      - Class – argument templejta je referentnog tipa
      - Struct – argument templejta je vrednosnog tipa
      - New() – argument templejta je klasa koja sadrži javni konstruktor bez argumenata
      - <imeKlase> - argument templejta je tipa date klase ili klase direktno ili indirektno izvedene iz date klase
      - <imeInterfejsa> - argument templejta može biti samo klasa koj implementira dati intefejs
- Generičke metode u C#-u:
  - Mogu se definisati i u klasama koje nisu generičke

- Definicija generičke metode:
  tip imeFunkcije<argumentTemplejta> (...) {...}
- Poziv generičke metode:
  imeFunkcije<listaTipova> (...)

67. Generičke metode i klase u Javi

- Prvi put uvedene u verziji Java 5.0
  - Smatralo se da tip Object može da zameni nepoznati tip
    - Problem: stalno kastovanje
- Parametar templejta može biti samo tip
- Na mesto stvarnog paramtera templejta se ne mogu koristiti primitivni tipovi:
  - Koriste se WRAPPER klase
- Sve bibliotičke klase za predstavljanje kolekcija su zamenjene generičkim klasa:
  - ArrayList, Vector, LinkeList, PriorityQueue, HashSet. LinkedHashSet, TreeSet...
- Definicija generičke funkcije:

```
TIP Ime<argumentTemplejta>(argumenti_funkcije)
{
    ...
}
```

- Definicija generičke klase

```
class Ime<argumenti_templejta>
```

- Upotreba parametara templejta
  - Parametar templejta se može koristiti kao:
    - Tip atributa
    - Tip parametra metode
    - Tip lokalne promenljive metode
    - Povratni tip metode
  - Parametra templejta se ne može koristiti kao:
    - Tip statičkog atributa
    - Tip parametara statičke metode
    - Tip lokalne promenljive u statičkoj metodi
    - Povratni tip statičke metode
    - Tip niza ili objekta koji se kreira
- Dodavanje ograničenja parametrima templejta:
  - U definiciji templejta mogu se dodavati ograničenja koja treba da zadovolje tipovi da bi se mogli upotrebiti kao argument templejta pri instanciranju objekta
  - Definicija ograničenja:
    - T extends B1 & I2 & I2 & … & In
- Upotreba džokera na mesto stvarnih parametara generičkih klasa
  - Na mesto stvarnog argumenta templejta se može upotrebiti:
    - Neograničeni džoker: ?
    - Ograničeni džoker: ? extends B

68. Programska struktura za obradu izuzetaka u Javi u C#

```
try{
    // Kôd koji može da prijavi izuzetke
```

```
}
catch (ExceptionType1 ex){
        // Obrada izuzetka tipa ExceptionType1
}
...
catch (ExceptionTypeN ex){
        // Obrada izuzetka tipa ExceptionTypeN
}
finally {
        // Deo koji se izvršava svejedno da li je
        // izuzetak prijavljen ili ne
}
```

Finally blok služi da oslobodi resurse koji su eventualno zauzeti pre nego što je izuzetak prijavljen.

69. Propagiranje izuzetaka u C++, Java i C#

- Ukoliko u funkciji postoji throw naredba, van try-catch strukture, koja taj tip izuzetaka obrađuje, kaže se da funkcija prijavljuje izuzetak.
- U deklaraciji ili definiciji funkcije **može**, ali **ne mora**, da se navede spisak tipova izuzetaka, koje funkcija prijavljuje.
- Spisak tipova izuzetaka koje funkcija prijavljuje se navodi iza zaglavja funkcije u opisu:
    - Throw (NizTipova)
- Ako ovakav opis stoji u definiciji funkcije, a funkcija prijavi izuzetak tipa koji nije u nizu identifikatora, kompajler će prijaviti grešku.
- Kompajler će prijaviti grešku i ako se ovakva funkcija pozove, a izuzeci navedenih tipova se nigde ne obrade.
- Ako iza zaglavlja funkcije ne stoji lista tipova izuzetaka, koje funkcija može da prijavi, funkcija sme da prijavi izuzetak proizvoljnog tipa.
- Funkciji se može zabraniti da prijavljuje izuzetke na dva načina:
    - Void radi(...) thorw();
    - Void radi(...) noexcept;
- Poneka se u bloku za obradu izuzetaka može javiti potreba da se izuzetak prosledi dalje, tada se navodi throw naredba bez argumenata:

```
Catch(ExcType ex)
{
        //  Obrada izuzeztka
        throw;
}
```

- Izuzeci u programskom jeziku Java
    - Obrada izuzetaka u programskom jeziku Java

```
try{
        //  Kod koji može da prijavi izuzetke
}
catch(ExceptionType1 ex){
        //  Obrada izuzetaka tipa ExceptionType1
}
...
catch(ExceptionTypeN ex){
        //  Obrada izuzetaka tipa ExceptionTypeN
}
finally{
        //  Deo koji se izvršava svejedno da li je izuzetak prijavljen ili ne
        //  Služi da oslobodi resurse koji su eventualno zauzeti pre nego što je
        //  izuzetak prijavljen
}
```

    - Checked & Unchecked exceptions

        - Unchekced – U fazi kompajliranja se ne proverava da li su izuzeci ovog tipa obrađeni
            - Izuzeci izvedeni iz klasa Error i RuntimeException
        - Checked – U fazi kompajliranja se proverava da li su izuzeci ovog tipa obrađeni
            - Svi izuzeci izvedeni iz klase Exception, osim onih izvedenih iz klase RuntimeException.
    - Ukoliko funckija prijavljuje izuzetak iz skupa Checked, u zaglavlju funkcije se obavezno navodi skup tipova itutetaka koje ona prijavljuje, kako bi kompajler mogao da proveri da li su obrađeni.
    - Skup checked tipova izuzetaka, koje funkcija prijavljuje se navode nakon zaglavlja u klauzuli throws:

      throws NizTipova
- Sličnosti i razlike u odnosu na C++:
    - Sličnosti:
        - Format naredbe za prijavu izuzetaka isti
        - Struktura za obradu izuzetaka sadrži try-catch delove identične onima u Javi
    - Razlike:
        - Izuzetak (u C++) može biti proizvoljnog tipa
        - Struktura za obradu izuzetaka ne sadrži finally deo
        - U zaglavlju funkcija koje prijavljuju greške, throws klauzula nije obavezna
- Sličnosti i razlike u odnosu na C#:
    - Sličnosti:
        - Format naredbe za prijavu izuzetaka je isti
        - Struktura za obradu izuzetaka try-catch-finally je identična
        - Objekti koji se prijavljuju kao izuzeci pripadaju standardnim klasam za predstavljanje izuzetaka, ili klasam izvedenih iz njih
    - Razlike:
        - Kod propagiranja izuzetaka, u zaglavlju funkcije koja prijavljuje izuzetke, ne navodi se lista tipova izuzetaka koje funkcija može da prijavi.

70. Razlika između Checked i Unchecked tipa izuzetaka u Javi

- Checked – U fazi kompajliranja se proverava da li su izuzeci ovog tipa obrađeni
    - Svi izuzeci su izvedeni iz klase Exception, osim onih izvedenih iz klase RuntimeException
- Unchecked – U fazi kompajliranja se ne proverava da li su izuzeci ovog tipa obrađeni
    - Izuzeci izvedeni iz klasa Error i RuntimeException

71. Pojam konkurentnog programa. Vrste konkurentnog programiranja

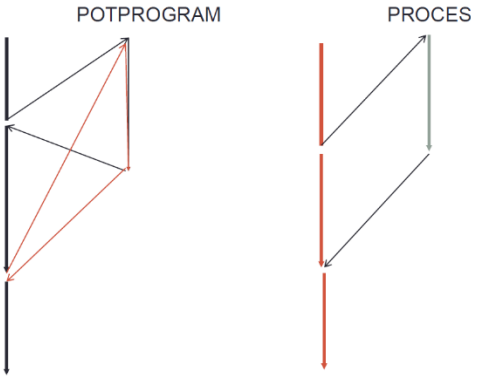
- Sadrže više izvršnih delova koji se mogu istovremeno izvršavati (biti aktivni)
- Konkurentnost može biti:
    - Fizička – izvršni delovi se izvršavaju paralelno na različitim procesorima
    - Logička – svi aktivni izvršni delovi se izvršavaju na istom procesoru u time sharing-u
- Nivoi konkurentnosti:
    - Nivo instrukcija – različite mašinske instrukcije se izvršavaju istovremeno
    - Nivo naredbi – naredbe višeg programskog jezika se izvršavaju istovremeno
    - Nivo potprograma – različiti potprogrami se izvršavaju istovremeno (korutine, procesi)
    - Nivo programa – više programa se izvršava istovremeno

72. Niti

- Proces(nit/thread) ili task je deo programa koji se može izvršavati konkurentno sa drugim procesima.

73. Razlika između proces i potprograma

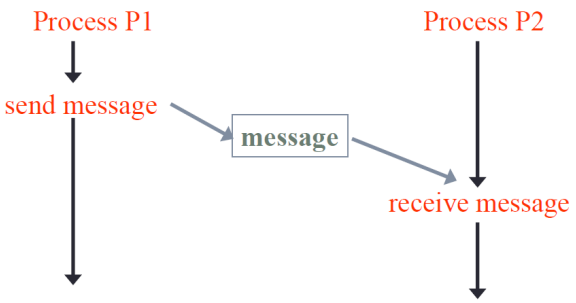
- Proces (nit/thread) ili task je deo programa koji se može izvršavati konkurentno sa drugim procesima.
- Razlika između procesa i potprograma:
  - Proces se mora eksplicitno startovati
  - Kada neki programski modul pokrene novi task, njegovo izvršenje se ne prekida – nastavljaju da se izvršavaju „zajedno“
  - Kada se izvršenje procesa završi, upravljanje ne mora da se vrati moduu koji ga je kreirao i pokrenuo



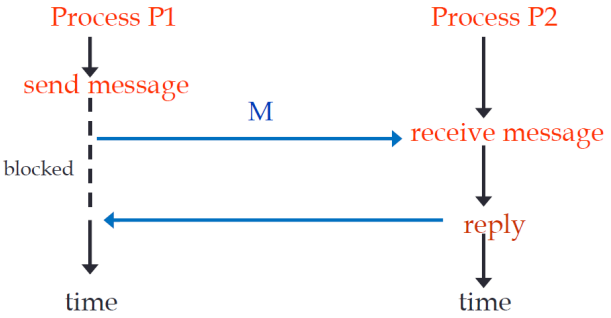
74. Načini sinhronizacije procesa

- Sinhronizacija pristupa zajedničkim resursima – međusobno isključivanje procesa kod pristupa zajedničkim resursima.
  - Načini sinhronizacije:
    - Semafori
    - Monitori
- Sinhronizacija procesa (komunikacije)
  - Način za sinhornizaciju:
    - Razmena poruka
- Razmena poruka (Message Passing) može biti:
  - Asinhrona:
    - Postoji prihvatni buffer, koji privremeno čuva pristigle poruke, dok primala ne bude spreman da ih primi.
  - Sinhrona (randevu):
    - Randevu je koncept sinhrone simetrične komunikacije.
    - Procesi se izvršavaju asinhrono, dok ne dođu do tačke komunikacije. Ko pre stigne do navedene tačke komunikacije mora da čeka da i drugi proces dođe do te tačke.
    - Ovaj način sinhronizacije je podržan u Adi.

Asinhrona razmena poruka



Sinhrona razmena poruka



75. Semafori

- Definisao ih je Dijkstra 1965. godine
- Služe za sinhronizaciju pristupa zajedničkim reursima
- Semafor – truktura podataka koja sadrži:

        - Brojač – sadrži broj procesa koji mogu da pristupe zajedničkom resursu
        - Red deskriptora procea koji čekaju:
            - Deskriptor adrži sve podatke potrebne da se suspendovani proces nastavi
- Semafori su podržani u programskim jezicima: Ada, Java, C#...

76. Načini rada semafora

- Semafori su jednostavno sredstvo za zaštitu deljivih resursa i međusobno isključivanje.
- To je obično binarna(logička) promenljiva koja se logički vezuje za resurs koji treba zaštiti. Resurs je dostupan kada je vrednost semafora jednaka jedinici. Za vrednost nula resurs semafora je nedostupan i proces koji se obraća njemu mora da čeka. U tom slučaju proces se prekida i svrstava se u red čekanja pridružen semaforu. Upravljanje semaforom se postiže pomoću procedura zauzeti i osloboditi semafor.

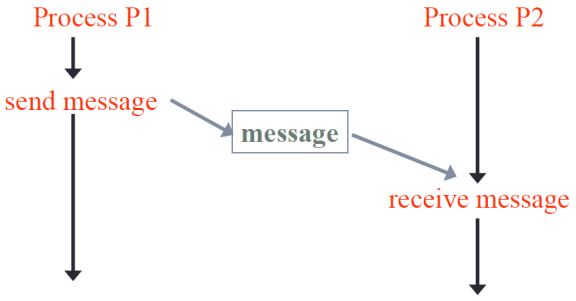
77. Producer/Consumer problem

- Sistem se sastoji od dva procesa od kojih jedan (proizvođač) generiše podatke, a drugi (korisnik) koristi te podatke. Pretpostavimo da je brzina generisanja i korišćenja podataka različita (promenljiva), pa ćemo za komunikaciju između ovih procesa koristiti bafer, u koji jedan proces upisuje vrednosti, a drugi ih čita. Ovaj primer ilustruje dva osnovna problema koji se javljaju u konkurentnim programima.
- Prvi problem je problem zaštite zajedničkih resursa, odnosno potreba za međusobnim isključivanjem kod pristupa zajedničkim podacima. U ovom konkretnom primeru, to znači da oba procesa ne mogu istovremeno da upisuju vrednosti u zajednički bafer, dok jedan proces pristupa podacima, drugi, ako treba da koristi iste podatke, mora da čeka
- Drugi problem je problem sinhronizacije između procesa. U opisanom primeru sinhronizacija se sastoji u tome da korisnik ne može da pročita ništa iz bafera, dok se u njega ne upiše bar jedan podatak, a proizvođač ne može da upiše ništa u bafer dok je pun.
- Prvo rešenje:
    - 2 semafora – jedan reguliše dodavanje elementa u bafer, a drugi čitanje
- Drugo rešenje:
    - 3 semafora – jedan reguliše dodavanje, jedan čitanje, a treći pristup baferu – u jednom trenutku samo jedan proces može da pristupi baferu

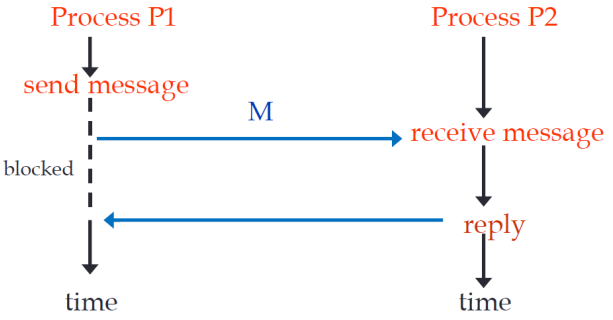
78. Sinhronizacija procesa razmenom poruka

- Razmena poruka može biti:
    - Asinhrona:
        - Postoji prihvatni buffer, koji privremeno čuva pristigle poruke, dok primalac ne bude spreman da ih primi.
    - Sinhrona (randevu):
        - Randevu je koncept sinhrone simetrične komunikacije.
        - Procesi se izvršavaju asinhrono dok ne dođu do tačke komunikacije. Ko pre stigne do navedene tačke komunikacije, mora da čeka da i drugi proces dođe do te tačke.
        - Ovaj način sinhronizacije je podržan u Adi.

Asinhrona razmena poruka



Sinhrona razmena poruka



79. Princip rada monitora

- Monitor je po strukturi modul koji objedinjuje podatke koji predstavjaju zajednički resurs za više procesa sa procedurama (funkcijama) kojima se realizuje pristup tim podacima. Po strukturi, monitor je isto što i svaki drugi modul, razlika je u posebnom koneptu upravljanja monitorom, kojim se prostiže da u jednom trenutku može koristiti samo jedna od procedura za pristup podacima monitora.
- Princip funkcionisanja monitora sastoji se u sledećem:
  - Ako se neko od aktiviranih procesa obrati monitoru, njemu se ne dozvoljava pristup podacima monitora već se prekida i svrstava u red čekanja pridružen monitoru.
  - Uvek kada jedan proces napušta monitor, proverava se red čekanja i ukoliko u njemu ima procesa koji čekaju, prvi od njih se aktivira i dozvoljava mu se pristup podacima.
  - Na taj način se postiže međusobno isključujući pristup zajedničkim podacima više procesa.

80. Niti u Javi

- Procesi se u Java aplikacijam definišu pomoću niti (threads)
- Nit je jedan tok u izvršenju programa
- Kada se kreira java aplikacija, pokreće se jedna nit – osnovna nit aplikacije
- Ako želimo da se izvesni delovi aplikacije izvršavaju istovremeno, kreiraju se nove niti.
- Kreiranje niti:
  - Definisanje klase koja će predstaviti novu nit u programu
    - Klasa za predstavljanje niti treba da:
      - Bude klasa izvedena iz klase Threads
      - Implementira interface Runnable
  - Kreiranje objekta tipa navedene klase
  - Pokretanje izvršenja niti
- Pokretanje izvršenja niti:
  - Ukoliko je klasa koja predstavlja nit u programu izvedena iz Thread:
    - Pozivom metode start() nad objektom kreirane klase
  - Ukoliko je klasa koja predstavlja nit implementirana sa Runnable:
    - Kreira se objekat klase Thread (čijem konstruktoru se prosleđuje objekat kreirane klase)
    - Nad tako kreiranim objektom olase Thread se poziva metoda start()

81. Sinhronizovani metodi u Javi

- Pomoću semafora
- Pomoću monitora, tj. Sinhronizovanih metoda i blokova

82. Semafori u Javi

- Semafor služi za sinhronizaciju pristupa zajedničkim resursima.
- Struktura podataka koja sadrži:
  - Brojač – sadrži broj procesa koji mogu da pristupe zajedničkom resursu

- Red deskriptora procesa koji čekaju pristup
  - Deskriptor sadrži sve podatke potrebne da se suspendovani proces nastavi
- Semafori u programskom jeziku Java
  - Klasa Semaphore je definisana u paketu:
    - Java.util.concurent
  - Konstruktor klase:
    - Semaphore(int permits)
  - Bitnije metode:
    - Void aquire() // (int permits)
    - Bool tryAquire() // (int permits)
    - Void release() // (int permits)
    - Int availablePermits()

83. Sinhronizovani metodi u Javi

- Definicija sinhronizovanih metoda:
  - Ključna reč synchronized ispred definicije
- Samo jedan (sinhronizovani) metod se može izvršiti u datom trenutku nad zadatim objektom
- Proces sinhronizacije koristi interni „lock“ koji je pridružen uz svaki objekat. To je neka vrsta flega koji proces postavlja (kada sinhronizovani metod započinje izvršavanje)
- Svaki sinhronizovani metod za objekat proverava da li je neki drugi metod postavio lock, pa ako jeste, ne započinje se izvršavanje dok se lock ne ukloni.
- Ne postoji uslov da se ne mogu simultano izvršavati sinhronizovani metodi nad različitim objektima klase
- Kontroliše se samo konkurentni pristup jednom objektu
- Metodi koji nisu deklarisani kao sinhronizovani mogu da se izvršavaju uvek, bez obzira da li je u toku izvršenje sinhronizovanog metoda nad objektom, ili ne.

84. Sinhronizovani blokovi

- Moguće je postaviti „lock“ na proizvoljan objekat za dati blok naredbi
- Kada se izvršava blok koji je sinhronizovan, za dati objekat, ne može se izvršavati nijedan drug blok, koji je sinhronizovan za taj objekat
- Definicija sinhronizovanog bloka:

```
synchronized(theObject)
        statement;
```

85. Producer/Consumer u Javi

- Brate, ovo su definicije klasa, bmk ovo da učim

86. Niti u C#

- Nit je jedan tok u izvršenju programa
- Prostor imena (namespace) System.Threading
  - Sadrži skup klasa i interfejsa potrebnih za rad sa više niti (multithread programming)
  - Glavna klasa u ovom prostoru je Thread
  - Sadrži i skup klasa i interfejsa neophodnih za sinhronizaciju niti
- Metoda koju nit izvršava:
  - Obavezno void
  - Bez parametara, ili sa parametrom tipa Object
- Sinhronizacija niti:
  - Zaključavanjem resursa
  - Klasom Monitor
  - Klasom Semaphore
  - Klasom Mutex

87. Sinhronizacija niti pomoću zaključavanja I monitora u C#

- Sinhronizacija niti zaključavanjem:
  - Koristi se kada neki deo koda u jednom trenutku može da izvrši samo jedna nit:
    - Tipičan primer je deo koda koji radi sa nekim deljivim resursom
  - Za zaključavanje se koristi sinhronizacioni objekat (objekat koji je vidljiv svim nitima)
    - **SINHRONIZACIONI OBJEKAT MORA BIT REFERENTNOG TIPA**
  - Sintaksa :

```
lock (<lockObj>)
{
    // kod koji izvršava samo nit koja je zaključala lockObj
}
```

- Sinhronizacija niti korišćenjem klase Monitor
  - Statička klasa, koja koristi sinhronizacioni objekat
  - Pre početka koda, koji se zaključava, poziva se metoda:
    monitor.Enter(<lockObj>)
  - Nakon koda koje se zaključava, poziva se metoda:
    monitor.Exit(<lockObj)

Zaključavanje

```
lock(buffer) {
  //critical section
}
```

Korišćenje klase Monitor

```
Monitor.Enter(buffer);
try
{
    //critical section
}
finally
{
    Monitor.Exit(buffer);
}
```

88. Sinhronizacija niti pomoću semafora i muteksa u C#

- Mutex omogućava da određenu kritičnu sekciju u kodu u jednom trenutku izvršava samo jedna nit.
- Semafori omogućavaju da određenu kritičnu sekciju u kodu u jednom trenutku izvršava najviše N procesa.
- Način rada sa muteksima/semaforima
  1. Kreirati statički objekat klase Mutex/Semaphore
  2. U svakoj niti, kritičan deo koda (deo koda za pristup zajedničkom resursu) omeđiti pozivima metoda WaitOne() i ReleaseMutex()/Release() nad tim objektom

89. Osnovni principi funkcionalne paradigme

- Ceo program se svodi na definisanje i evaluaciju matematičkih funkcija
- Nema bočnih efekata: izlazna vrednost funkcije zavisi isključivo od ulaznih parametara
- Nema programskih struktura – petlje se zamenjuju rekurzivnim funkcijama
- Predstavnici funkcionalnih jezika:
  - Lisp, Scheme, Haskell, ML, Scala, Ocaml...
- Lisp:
  - Jedina struktura podataka koja se koristi su liste
  - Podaci su nepromenljivi
- Program je niz definicija i poziva funkcija
- Osobine funkcionalnog programiranja:
  - Ne postoji eksplicitno zadavanje redosleda izračunavanja
    - Program definiše samo izraz koji je rešenje nekog problema
  - Nepostajanje naredbi
    - Umesto njih postoje izrazi
      - Uslovni izrazi se koriste umesto naredbi grananja
      - Rekurzivni pozivi se koriste umesto programskih petlji
  - Nepostojanje sporednih efekata
    - Vrednost izraza za iste vrednosti ulaznih parametara je uvek ista, bez obzira u kojoj se okolini izračunava
  - Nepromenljivost podataka

- Promenljive u programu nisu neophodne
  - Potpuno funkcionalni jezici ne podržavaju rad sa promenljivama kao memorijskim lokacijama gde se čuva neka vrednost.
- Funkcije višeg reda
  - Funkcije čiji su parametri i/ili rezultati druge funkcije
- Slaganje (kompozicija) funkcija
  - U matematici kompozicija funkcija je definisana na sledeći način:
    - $(f \circ g)(x) = f(g(x))$
  - U funckionalnom programiranju se programi grade kao kompozicije funkcija.

90. Elementi funkcionalog jezika

- Strukture podataka
  - Atomi
  - Liste
- Skup elementarnih (ugrađenih) funkcija
- Format za definisanje novih funkcija
- Format za primenu (poziv) funkcije

91. Elementi funkcionalnog jezika koji odgovaraju programskim strukturama imperativne paradigme

- Programske strukture:
  - Sekvenca – kompozicija funkcija
  - Selekcija – uslovni izrazi u funkcionalnom jeziku
  - Petlja – Rekurzivni pozivi u funkcionalnom jeziku

92. Neke strukture iz imperativne paradigme koje su iste/slične sa funkcionalnom paradigmom

- Programske strukture:
  - Sekvenca – kompozicija funkcija
  - Selekcija – uslovni izrazi u funkcionalnom jeziku
  - Petlja – Rekurzivni pozivi u funkcionalnom jeziku

93. LISP strukture podataka

- Atomi – elementarni podaci mogu biti:
  - Numerički – number
  - Stringovi (nizovi karaktera između dvostrukih navodnika)
  - Logički (t- true, NIL – false)
- Strukturni tipovi:
  - Par – dva podatka elementarnog ili strukturnog tipa napisana između malih zagrada
  - Lista – tri ili više podataka elementarnog ili strukturnog tipa napisana između malih zagrada
    - Liste se smatraju glavnim tipom podatka, otuda i ime jezika LISt Processing
    - Primeri listi:
      - (5 3 4)
      - (1 (1 2) 2)
      - (A „A“ (A „A“))

94. Definisanje i poziv funkcije u Lisp-u

- Poziv funkcije je lista u kojoj je prvi element ime funkcije, a ostali parametri
- Primer poziva funkcije

  (funkcija a b c)

95. Definisanje i poziv funkcije u JavaScript-u

- Definicija:

  function name (parameters) { body }
- Primer:

function Product (a, b) { return a*b; }

- Funkcija može biti dodeljena promenljivoj:
  - ES5:

    var x = function (a, b) { return a*b; }
  - ES6:

    const x = (a, b) => a * b;
- Funkcija može biti parametar druge funkcije
- Funkcija može biti rezultat druge funkcije

96. Funkcije višeg reda u Lisp-u i JavaScript-u

- Funkcije višeg reda su funkcije čiji su parametri i/ili rezultati druge funkcije
- Lisp:
  - Mapping funkcije:
    - (mapcar fn arg1 arg2 ... argn)
      - Primenjuje zadatu funkciju nad odgovarajućim elementima listi arg1-argn i vraća listu rezultata (ukoliko liste arg1-argn nisu iste dužine, rezultat će biti lista dužine min(arg1.lenght, arg2.lenght,...,argn.lenght)
      - Primer:

        (mapcar + (1 2 3) (4 5 6) (7 8))) (12, 15)
    - (maplist fn arg1 arg2 ... argn)
      - primenjuje zadatu funkciju nad odgovarajućim elementima listi arg1 – argn, a zatim nad njihovim podlistama cdr(arg1)-cdr(argn) pa nad njihovim podlistama...
      - Rezultata je lista listi
  - Reduce funkcija:
    - (reduce fn list)
      - Primenjuje zadatu funkciju nad prva dva elementa liste, zatim nad rezultatom i sledećim elementom itd
      - Primer:
        - (reduce + (1 2 3)) 6
- JavaScript
  - Array.map(fn)
    - Zadatu funkciju primenjuje nad svakim elementom niza
    - Parametri funckije fn:
      - Tekući član – obavezno
      - Tekući indeks – opciono
      - Niz – opciono
    - Primer:
      - ES5:

        var niz = [1, 2, 3];
        var noviNiz = niz.map (function(clan) { return 2*clan; }};
      - ES6:

        var niz = [1, 2, 3];
        var noviNiz = niz.map(x => 2*x);
  - Arra.filter(fn)
    - Kreira novi niz od elemenata polaznog niza, koji zadovoljavaju zadatu test funkciju
    - Paramteri funkcije fn su isti kao kod map funkcije

        - Primer:
            - ES6:
                - Var niz = [1, 2, 3];
                - Var noviNiz = niz.filter(x => x%2 == 1);
    - Array.reduce(fn)
        - Zadatu funkciju primenjuje nad svakim elementom niza i kreira jednu rezultujuću vrednost.
        - Parametri funkcije fn:
            - Akumulator . inicijalna vrednost rezultata – obavezno
            - Tekući element – obavezno
            - Tekući ideks – opciono
            - Niz – opciono
        - Primer:
            - ES5:

```
var niz = [1, 2, 3];
var sum = niz.reduce( (acc,x) => acc + x );
```

97. Pojam i elementi predikatske logike

- Iskaz je svaka rečenica koja može imati istinitosnu vrednost: tačno/netačno
- Složeni iskazi se grade korišćenjem operatora
- Iskazna logika se bavi proverom da li je vrednost nekog iskaza tačna, ili ne.
- Predikat je iskaz, čija istinitosna vrednost zavisi od vrednosti nekog parametra
    - Npr:
        - „3 je paran broj“ – iskaz (netačan)
        - „x je paran broj“ – predikat, jer tačnost iskaza zavisi od x
- U predikatskoj logici se osim logičkih operatora u građenju formule koriste i kvantifikatori:
    - Univerzalni kvantifikator: ∀
    - Egzistencijalni kvantifikator: ∃
- Uvođenjem kvantifikatora istinitosna vrednost predikatske formule prestaje da zavisi od parametara.
- Predikatske formule sa kvantifikatorima:
    - (∀x) (P(x)) [npr: (∀x) („x je paran broj“) - netačno]
    - $(\exists x)$(P(x)) [npr: $(\exists x)$ („x je paran broj“) - tačno]
- Formalna definicija predikatske logike prvog reda:
    - Termovi – objekti nekog domena:
        - Konstante
        - Promenljive
        - Funkcije
    - Atomične formule (mogu imati istinitosnu vrednost)
        - Logičke konstante T i ⊥ su atomične formule.
        - Ako je ρ n-arni relacijski simbol i t1, t2, ... , tn termovi, onda je ρ(t1, t2, ... , tn) atomična formula.
    - Formule (mogu imati istinitosnu vrednost)
        - Atomične formule su fomrule
        - Ako je A formula, onda je i ¬(A) formula.

- Ako su A i B formule, onda su i AvB, A^B, A B, A ⇔ B formule.
- Ako je A formula, a x promenljiva, onda su i (∀x)(A) i (∃x)(A) formule.

98. Naredbe u Prolog-u

- Postoje tri osnovne vrste naredbi u Prolog-u:
  - Činjenice – predikatske formule, čija je instinitosna vrednost uvek T
  - Pravila – definišu način za izvođenje novih činjenica (zaključaka)
  - Upiti – definišu proble koji program treba da reši
- Činjenice i pravila se čuvaju na disku i predstavljaju bazu znanja
- Upiti – naredbe programa – program korišćenjem baza znanja kreira odgovor na upit

99. Činjenice u Prologu

- Činjenica je jedna od tri osnovne vrste naredbi u prologu.
- To su predikatske formule, čija je istinitosna vrednost uvek T (tačna).
- Čuvaju se na disku (zajedno sa pravilima) i predstavljaju bazu znanja.
- Sadrže neke informacije o objektima i relacijama između njih.
- To je naredba napisana u jednom redu, koja počinje predikatom i završava se tačkom
- Primer:
  - Zena(Ana).
  - Roditelj(Ana, Petar).
  - Roditelj(Pavle, Ana).

100. Definicija pravila u Prolog-u

- Pravila – definišu način za ozvođenje novih činjenica (zaključaka)
- Pravilo sadrži dva dela:
  - Zaključak (posledica) – leva strana pravila
  - Uslov (telo) – desna strana pravila
- Format pravila:
  <zaključak> :- <uslov>
- Primer pravila:
  predak(A, B) :- roditelj(A, B).
  predak(A, B) :- roditelj(A, C), predak(C, B).

101. Definicija upita u Prolog-u

- Upiti – definišu problem koji program treba da reši
- Upiti – naredbe programa – program korišćenjem baza znanja kreira odgovor na upit
- Format upita:
  - ?- <predikat>.
- Postoje dva osnovna tipa upita u programskom jeziku Prolog (zavisno od toga kakav odgovor traži):
  - YES/NO query – upit u čijem predikatu učestvuju samo konstante.
    - Odgovr na ovakav upi može iti samo T ili NT (YES/NO)
  - Unification/No query – upit u čijem predikatu učestvuju i promenljive
    - Odgovor na ovakav upit su k-torke objekata koje zadovoljavaju upit (gde k predstavlja broj promenljivih u predikatu) ili NT (ne postoji katorka objekata koji zadovoljavaju upit)
- Upit – primeri
  - ?- predak(Pavle, Petar)
    - YES
  - ?- predak(X, Petar)
    - X = Ana
    - X = Pavle