

UVOD U VEB I INTERNET TEHNOLOGIJE

2021/2022

Jelena Maksimovid

Jovana Đurovid

Lucija Miličid

Marija Čulid

Natalija Asanovid

dopunila: Julijana Jevtid

UVOD U VEB I INTERNET TEHNOLOGIJE.....	1
1. Uvod u racunarske mreze	2
2. Internet, istorijat, opis, usluge i protokoli.....	9
3. Mrezni protokoli	19
4. Jezici za obeležavanje.....	36
5. Programski jezi JavaScript	72
6. Struktura JavaScript programa.....	77
7. Funkcije i zatvorenja	95
8. Objekti i nizovi	107
9. Funkcije viseg reda	114
10. Napredni objekti	123
11. Asinhroni JavaScript	133
12. Rukovanje greskama	143
13. Moduli.....	152
14. JavaScript programiranje koriscenjem okruzenja node	164
15. JavaScript serversko programiranje koriscenjem okruzenja node.....	173
16. JavaScript veb programiranje na klijentskoj strani	186

1. Uvod u racunarske mreze

1. Uloga i nacin rada racunarskih mreza

Osnovne uloge racunarskih mreza:

- 1) **Komunikacija** - elektronska posta, drustvene mreze, Skype itd.
- 2) **Deljenje informacija i podataka** - raspored casova, red voznje itd.
- 3) **Deljenje softvera** - korisnici povezani u mrezu mogu da koriste mnoge usluge koje im pruza softver koji radi na racunarima u okviru mreze (putem Veba je moguće kupovati, rezervisati karte, realizovati bankarske transakcije)
- 4) **Deljenje hardverskih resursa** - zajednicko koriscenje hardvera u mreznom okruzenju (stampaca, skenera)

Nacini rada racunarskih mreza:

- 1) **Centralizovana obrada** - svi poslovi se izvrsavaju na jednom centralizovanom racunaru, dok se ostali cvorovi koriste samo kao terminali za unos podataka i prikaz teksta; koristilo se u proslosti
- 2) **Klijent - server okruzenje** - jedan racunar (obicno mocniji) ima ulogu servera na kome se nalaze podaci i aplikativni softver, koji se stavljaju na raspolaganje klijentima na njihov zahtev
 - najcesce se vise klijenata obraca jednom serveru
 - cesto se trazene usluge distribuiraju vecem broju servera
 - proksi - serveri: kesiraju veb strane kojima se pristupa, omogucavaju drugim racunarima posredan pristup sadrzajima na netu (serveri koji klijentima ubrazavaju pristup Internetu)
- 3) **Mreza ravnopravnih racunara** (peer-to-peer, **P2P**) - racunari direktno komuniciraju jedan sa drugim, dele podatke i opremu; koriste se za masovnu razmenu velikih kolicina podataka (Bittorent)

2. Komponente racunarskih mreza – Mrezni hardver

Racunarska mreza je sistem koji se sastoji od **skupa hardverskih uredjaja** medjusobno povezanih **komunikacionom opremom** i snabdeven odgovarajucim **kontrolnim softverom**, kojim se ostvaruje kontrola funkcionisanja sistema tako da je omogucen prenos podataka izmedju povezanih uredjaja. Komponente mreze: mrezni hardver, komunikacioni kanali, mrezni softver

- Da bi uredjaj mogao biti umrezen, neophodno je da sadrzi specijalizovan deo hardvera namenjen umrezavanju koji se smatra delom komunikacione opreme. Obicno je to **mrezna kartica** (mrezni adapter) - **NIC** (network interface card) koja omogucava uredjaju da fizicki pristupi mrezi. Svaku mreznu karticu karakterise jedinstvena **fizicka (MAC) adresa**, kojom se uredjaj jedinstveno identifikuje prilikom komunikacije.

- Skoro svi desktop racunari imaju ugradjenu NIC karticu, dok prenosni imaju ugradjenu karticu za bezicno povezivanje (**WNIC**)

- Osim mreznih kartica, koriste se i **modemi** (telefonski, kablovski). Modem (modulator-demulator) je uredjaj koji konvertuje digitalni signal u analogni, koji se prenosi, a zatim obrnuto konvertuje preneti signal u digitalni; koristi se kablovski ili ADSL pristup internetu

- Modem se obično zakupljuje od dobavljača interneta i priključuje se na parice fiksne tehnologije, koaksijalne kablove kablovske televizije ili bezicne mreze mobilnih operatera.
- Hab, most, svc i ruter su tipovi mreznog hardvera i sluze za prosledjivanje komunikacije izmedju racunara u mrezi i povezivanju mreze sa internetom

3. Komponente racunarskih mreza – Komunikacioni kanali

- Komunikacioni kanali su kablovi ili bezicni medijumi, koji prenose podatke elektromagnetnim talasima (radio-talasima, opticki talasima, mikrotalasima)
- Osnovna mera kvaliteta komunikacionog kanala je **brzina prenosa/protok** (throughput, bandwidth), koja se meri brojem bitova koji se mogu preneti u sekundi; najcesce Mbps ili Gbps
- Druga vazna mera kvaliteta je **kasnjenje** (latency) - vreme potrebno da se komponenta pripremi za pristup podacima (meri se u mikrosekundama u lokalnim mrežama i milisekundama u okviru vecih mreza)
- **Brzina prenosa** je fizicka karakteristika komunikacionog kanala i zavisi od frekvencijalnog opsega (bandwidth) koji se moze propustiti kroz kanal bez gubitka signala
- Neke mrezne kartice obezbedjuju pristup zicanim, a neke bezicnim komunikacionim kanalima

4. Komponente racunarskih mreza – Komunikacioni kanali, ozicene komunikacije

1) Parice (twisted-pair wire)

- najkorisceniji nacin komunikacije
- uredjaji se povezuju koriscenjem uvijenih, uparenih, izolovanih, bakarnih zica, slicno povezivanju obicnih telefona
- zice se uparuju i uvijaju kako bi se smanjile smetnje u komunikaciji
- brzina prenosa: 2Mbps - 100Mbps
- UTP kablovi (unshielded twisted pair) kategorije 3 koriste se u fiksnoj telefoniji, a kategorije 5 ili 6 u lokalnim mrežama; protok oko 100 Mbps(brzi Ethernet), pa i 1 Gbps(gigabitniEthernet)

2) Koaksijalni kablovi

- obicno se koriste za televizijske sisteme, a koriste se i u LAN mrežama itd.
- sastoje se od centralne bakarne ili aluminijske zice obmotane savitljivim izolatorskim slojem, oko kojeg je opet obmotan provodni sloj tankih zica; sve je to obmotano spoljasnjom izolacijom
- brzina prenosa: do 200Mbps (cak i do 500Mbps); moze doci do manje osetljivosti na elektromagnetne smetnje

3) Opticki kablovi

- prave se od velikog broja (stotina, hiljada) veoma tankih, staklenih vlakana umotanih u zastitni sloj
- podaci se prenose svetlosnim talasima, koje emituje mali laserski uredjaj
- na ove kablove ne uticu smetnje prouzrokovane elektromagnetnim zracenjem
- skupi su i komplikovani za instalaciju, pa se uglavnom koriste za osovinski deo mreze (mrežnu kicmu), na koji se onda koaksijalnim kablovima ili ukrstenim paricama povezuju pojedinačni uredjaji
- brzina prenosa: od 10Gbps pa navise

5. Komponente racunarskih mreza – Komunikacioni kanali, bezicne komunikacije

- ne koriste kablove za prenos podataka, vec se koriste radio talasi, mikro talasi i infracrveni zraci
- podaci se prenose moduliranjem amplitude, frekvencije ili faze talasa (moduliranje je proces u kojem signal informacije menja drugi signal vise frekvencije, takozvani nosilac, da bi se omogucio prenos)
- prakticno kod prenosivih racunara, mobilnih uredjaja ili relativno udaljenih lokacija za koje bi uspostavljanje kablovske mreze bilo nedopustivo skupo

Najcesce bezicne tehnologije:

1) Bluetooth

- za komunikaciju na veoma malim razdaljinama
- brzina prenosa: do 3Mbps
- koristi radio talase, moze da prodje i kroz cvrste prepreke

2) Bezicni LAN (WLAN, WiFi)

- koristi radio talase za bezicnu komunikaciju vise uredjaja na ogranicenom rastojanju (nekoliko desetina ili stotina metara)
- brzina prenosa: od 10Mbps do 50Mbps (u novije vreme i do 600Mbps)
- najrasireniji standard za bezicnu LAN komunikaciju je IEEE 802.11
- mrezi se pristupa preko **pristupnih tacaka** (access point)
- oblast prostora u kojoj je mreza dostupna se naziva **hot spot** (ili sto bi Vlado rekao --> vruca tacka, lol)

3) Bezicne gradske mreze (WiMAX)

- pokrivaju sira podrucja i daju protok do 40Mbps

4) Celijski sistemi

- za komunikaciju se koriste radio talasi i sistemi antena koje pokrivaju odredjenu geografsku oblast, pri cemu se signal od odredista do cilja prenosi preko niza antena

5) Zemaljski mikrotalasi

- koriste antensku mrezu na Zemlji
- za komunikaciju koriste mikrotalase niske frekvencije, koji zahtevaju da antene budu opticki vidljive, tako da se one obicno smestaju na visoke vrhovne tacke (Avala be like)

6) Komunikacioni sateliti

- koriste mikrotalase za komunikaciju tako sto se prenos izmedju dve tacke koje nemaju opticku vidljivost ostvaruje poprecnom komunikacijom preko komunikacionih satelita
- ovako se pored racunarske komunikacije, obicno prenose televizijski i telefonski signali
- brzina prenosa: relativno mala, oko 100Mbps

6. Komponente racunarskih mreza – Mrezni softver

- neophodan za funkcionisanje racunarskih mreza
- zbog kompleksnosti racunarskih mreza, softver se organizuje hijerarhijski
- obuhvata razne slojeve: od sistemskog softvera niskog nivoa do aplikativnog softvera
- slojevitost olaksava programiranje mreznog softvera (autori aplikativnog softvera ne moraju da brinu o detaljima mrezne komunikacije)
- Moze da se podeli na dva nivoa:

1. **Mrezni softver niskog nivoa** - omogućuje koriscenje razlicitih mreznih uredjaja (mreznih kartica ili modema); nalazi se obicno u jezgru operativnog sistema, uglavnom u obliku upravljacka perifernim uredjajima (drajvera); korisnik nikada ne koristi ovaj softver direktno, u opstem slucaju on nije ni svestan da taj softver postoji

2. **Mrezni softver visokog nivoa** - zadatak mu je da pruzi usluge mreznim aplikacijama koje korisnici koriste; ove aplikacije pruzaju razlicite usluge korisnicima na mrezi (e-mail, pregledanje veba itd.)

Danas operativni sistemi sadrze sve nivoe mreznog softvera, osim aplikativnog

7. Raspon racunarskih mreza

- jedan od kriterijuma za klasifikovanje mreza je i njihova fizicka velicina, tj. geografski prostor koji mreza pokriva (od mreze dva racunara do Interneta)
- hijerarhijsko umrezavanje: mreze velikog raspona povezuju manje mreze
- razlicite tehnologije se koriste za razlicite raspone

Prema rasponu, mreze se klasifikuju na:

1) *Personal area network (PAN)*

- mreze koje su namenjene za jednog coveka (npr. bezicna mreza kojom su spojeni racunar, mis i stampac)
- obicno pokrivaju raspon od nekoliko metara i koriste bilo zicanu bilo bezicnu komunikaciju

2) *Local area network (LAN)*

- povezuje uredjaje na relativno malim udaljenostima (npr. nekoliko kancelarija u okviru jedne poslovne zgrade)
- ovakve mreze se tradicionalno vezuju na zicanu komunikaciju kroz mrezne kablove, iako nove tehnologije daju mogucnost koriscenja postojećih kucnih instalacija (koaksijalnih kablova, telefonskih linija i elektricnih linija) za komunikaciju, kao i mogucnost koriscenja bezicne komunikacije

3) *Campus area network (CAN)*

- povezuju vise lokalnih mreza u okviru ogranicenog geografskog prostora (npr. u okviru jednog univerziteta, kompanije itd.)
- tehnologija za povezivanje je obicno ista kao i kod LAN

4) *Metropolitan area network (MAN)*

- povezuju vece geografske prostore (npr. jedan grad)

- obicno povezuje vise lokalnih mreza (LAN) koriscenjem veoma brze kicme komunikacije (backbone), koja je najcesce izgradjena od optickih veza

5) Wide area network (WAN)

- povezuju izrazito velike geografske prostore, cesto sire od granica jednog grada, oblasti, a i drzave
- obicno su u sastavu Interneta
- WAN infrastrukturu obicno odrzavaju komercijalne kompanije i one iznajmljuju usluge koriscenja
- za povezivanje u okviru kicme se koriste brze veze, najcesce opticke i satelitske

8. Zajednicki komunikacioni kanal. Nacini deljenja zajednickog kanala

- ove mreze se sastoje od zajednickog komunikacionog kanala preko kojeg komuniciraju svi racunari povezani u mrezu
- ovaj nacin povezivanja se obicno koristi za komunikaciju u okviru manjih, lokalnih mreza
- racunari salju kratke poruke (pakete) na mrezu, postavljajuci ih na komunikacioni kanal, pri cemu svaka poruka sadrzi i identifikaciju zeljenog primaoca
- poruku svi primaju, pri cemu je onaj kome je namenjena jedini prihvata, dok je ostali odbacuju
- pristup uredjaja kanalu se moze odredjivati *staticki*, kada svaki uredjaj ima unapred odredjena pravila kako i u kom delu kanala sme da vrsi komunikaciju ili *dinamicki* kada se pristup uredjaja kanalu odredjuje na osnovu trenutnog stanja i dostupnosti kanala

Neki od osnovnih nacina deljenja zajednickog kanala:

1) Deljenje vremena (time division multiplexing - TDM)

- deljenje komunikacionog kanala koriscenjem deljenja vremena je jedan od nacina staticke alokacije kanala (svaki uredjaj komunicira u tacno odredjenom vremenskom trenutku, pri cemu se uredjaji naizmenicno smenjuju)

2) Deljenje frekvencije (frequency division multiplexing - FDM)

- drugi nacin staticke alokacije kanala (svaki uredjaj komunicira u okviru odredjenog frekvencijskog opsega)

3) Deljenje talasne duzine (wave division multiplexing - WDM)

- deljenje frekvencije u slucaju kada se radi o optickoj komunikaciji

4) Deljenje kodiranjem (code division multiple access - CDMA)

- jedan od novijih nacina statickog deljenja kanala, u okviru kog se koristi teorija kodiranja kako bi se iz primljenog paketa izdvojile informacije relevantne za odredjeni cvor

5) CSMA/CD (carrier sense multiple access with collision detection)

- najkoriscenija tehnika dinamickog deljenja kanala
- koristi se u okviru Ethernet LAN mreza
- u ovom slucaju se postuje protokol da svaki uredjaj osluskuje da li kanalom vec tece neka komunikacija pre nego sto pocne da salje podatke; ukoliko se primeti da neko drugi istovremeno pokusava da posalje podatak, uredjaj prekida svoje slanje, ceka odredjeno vreme i pokusava ponovo

9. Topologija mreza sa zajednickim komunikacionim kanalom

- Topologija mreze predstavlja nacin na koji su povezane medju sobom razlicite komponente mreze, kao i nacin na koji intereaguju

- Dva nivoa topologije mreze:

1. *fizicka topologija* - odredjena rasporedom kablova i bezicnih veza
2. *logicka topologija* - odredjena tokom podataka

- Razlicite topologije se razlikuju prema:

1. *osnovnoj ceni* (koliko se ulaze u bas taj oblik povezivanja)
2. *ceni komunikacije* (koliko je vremena potrebno za prenos poruke)
3. *pouzdanosti* (mogucnosti prenosa podataka u slucaju otkaza nekog cvora ili veze)

- Postoje dva nacina povezivanja topologija:

- 1) Zajednicki komunikacioni kanal (broadcast)
- 2) Direktna veza od cvora do cvora (point-to-point)

Razlikuju se cetiri tipa topologije mreze:

1. **Magistrala** - mreza povezuje svoje komponente jednim istim kablom i informacija se istovremeno raznosi svim primaocima; koristi se kaoksjalni kabal; saobracaj se odvija u dva smeru (pri vecem opterecenju moze doci do sudaranja paketa ili zagusenja kanala)

2. **Zvezda** - u zvezdastoj mrezi, svi ucesnici su povezani na jednu istu centralnu tacku (cvor-racunar), a informacija putuje od posiljaoca prema primaocu iskljucivo preko centralne tacke; cena uspostavljanja mreze i cena komunikacije su niske, ali zagusenje u centralnom cvoru je cesto (postavlja se switch)

3. **Prsten** - mreza ima sve svoje komponente na istom kabl, ali taj kabl nema krajeve; informacija se krece samo u jednom, strogo odredjenom pravcu; ukoliko neki od cvorova otkaze, to nece uticati na funkcionisanje ostatka mreze

4. **Potpuna povezanost** - svaki cvor poseduje posebnu vezu sa svakim od preostalih cvorova; koristi se samo kod sasvim malih mreza i to zbog pouzdanosti (mala mogucnost padova u mrezi)

10. Direktne cvor-cvor veze. Komutiranje (switching)

- mreze se sastoje od velikog broja direktnih veza izmedju individualnih parova racunara
- kako bi informacija stigla od jednog do drugog cvora, obicno je potrebno da prodje kroz niz posrednih
- koristi se u okviru velikih mreza
- informacije mogu da putuju razlicitim putanjama, izbor utice na efikasnost komunikacije
- komutiranje (switching) - odredjivanje putanje pre ili tokom same komunikacije

Tipovi komunikacije u zavisnosti od nacina odredjivanja putanja i nacina slanja informacije:

1) Komutiranje kanala (circuit switching)

- pre zapocinjavanja komunikacije ostvaruje se trajna fiksirana putanja (kanal) izmedju cvorova i sva informacija se prosledjuje kroz uspostavljenu putanju
- na ovaj nacin funkcionisu mreze fiksne telefonije

- kanal (vod) je rezervisan sve dok se eksplicitno ne raskine, pa je ovaj nacin prilicno skup

2) Komutiranje poruka (message switching)

- svaka poruka koja se salje putuje zasebnom putanjom

3) Komutiranje paketa (packet switching)

- poruke se pre slanja dele na zasebne manje pakete i svaki paket putuje svojom zasebnom putanjom, da bi se na odredistu paketi ponovo sklopili u jedinstvenu poruku

- prednost: paketi paralelno putuju kroz mrezu, povecava se brzina

- komutiranje je lakse ako je duzina sadrzaja koji se prenosi manja

11. Topologija velikih mreza.

Prethodna podela (9.) se odnosi na male mreze, a velika mreza se obicno sastoji od velikog broja malih mreza, od kojih svaka ima sopstvenu topologiju.

Velika mreza ce, dakle, imati razlicite komponente sa razlicitim topologijama, ali ce takodje imati i jednu opstu (generalnu) topologiju koja ce biti ili zvezda, ili magistrala, ili prsten.

12. Slojevi kod mreza. OSI model.

Danasnje mreze su izuzetno kompleksne, te ih je potrebno kreirati hijerarhijski (pravimo analogiju sa hijerarhijom racunarskog sistema: sloj hardvera, sistemskog i aplikativnog softvera; komunikacija na visim slojevima se ostvaruje dostavom poruka na nizim slojevima pri cemu visi sloj ne poznaje detalje komunikacije na nizim slojevima i obrnuto).

Broj slojeva se razlikuje od mreze do mreze i na svakom sloju se sprovodi odgovarajuci protokol komunikacije. Protokol je dogovor dve strane o nacinu komunikacije (narusavanje protokola cini komunikaciju nemogucom).

Istorijski se mreze posmatraju u okviru dva referentna modela:

1) **OSI** (Open System Interconnection) - model sa 7 slojeva standardizovan od strane ISO (medjunarodne organizacije za standardizaciju). Iako se konkretni protokoli koje definise vise ne koriste, predstavlja znacajan apstraktni opis mreza.

2) **TCP/IP** - model sa 4 sloja, prisutan u okviru Interneta. Za razliku od OSI modela, njegova apstraktna svojstva se ne koriste znacajno, dok se konkretni protokoli intenzivno koriste i predstavljaju osnovu interneta.

Slojevi OSI modela (u zagradi su analogni slojevi TCP/IP modela):

1) application - aplikativni (**isto aplikativni**) = definise protokole

2) presentation (ne postoji) = salje podatke od aplikativnog ka sesiji, moze da kodira/kompresuje/enkriptuje

3) session (ne postoji) = uspostavljanje sesije izmedju trenutno pokrenutih programa koji komuniciraju

4) Transport (**isto transportni**) = deli prihvacene podatke sa visih slojeva na manje pakete, salje ih

5) Network - mrezni (**Internet**) = povezivanje razunara na mrezu

6) Data link - (**Host-to-network**) = visim slojevima obezbedjuje postojanje pouzdanog kanala komunikacije

7) Physical - fizicki (**Host-to-network**) = najnizi

2. Internet, istorijat, opis, usluge i protokoli

1. Strukturni opis interneta.

Sa strukturnog stanovista, internet se definise preko hardverskih, komunikacionih i softverskih komponenti koje ga cine. Internet je jedna WAN mreza koja povezuje mnoštvo manjih privatnih ili javnih mreza i omogućava medjusobnu komunikaciju racunara koji su povezani na ove mreze. Komunikacioni kanali su izgradjeni od raznovrsnih komunikacionih tehnologija (kablova, bezicnih i satelitskih veza). Krajnji racunari se nazivaju HOST i oni nisu medjusobno neposredno povezani, vec preko rutera.

Struktura je hijerarhijska: host racunari -> mreza lokalnih internet dobavljacka (Internet Service Provider - ISP)-> regionalne mreze -> nacionalne i internacionalne mreze.

I host racunari i ruteri postuju IP protokol komunikacije koji, izmedu ostalog, svakom od njih dodeljuje jedinstvenu logicku adresu - **IP adresa**. IP protokol definise mogucnost slanja paketa informacija izmedju hostova i rutera. Paketi od hosta do hosta putuju preko niza rutera, putanja se automatski određuje i hostovi nemaju kontrolu nad putanjom paketa (koristi se paketno komutiranje). Softver koji je instaliran na host racunarima korisnicima pruza razlicite usluge.

2. Funkcionalni opis interneta.

Sa funkcionalnog stanovista, internet se definise preko **usluga** koje nudi svojim korisnicima. Internet je mrezna infrastruktura koja omogućava rad distribuiranim aplikacijama koje korisnici koriste. Ove aplikacije ukljucuju:

- Veb (World Wide Web) - omogućava korisnicima pregled hipertekstualnih dokumenata
- elektronsku postu (e-mail)
- prenos datoteka (ftp, scp) izmedju racunara
- upravljanje racunarima na daljinu preko prijavljivanja na udaljene racunare (telnet, ssh)
- slanje instant poruka (im)...

Vremenom se gradi sve veci i veci broj novih aplikacija. One medusobno komuniciraju preko svojih specificnih aplikacionih protokola (npr. HTTP, SMTP, POP3, ...). Svi aplikacioni protokoli komuniciraju koriscenjem dva transportna protokola:

TCP - (engl. Transmission Control Protocol) protokol sa uspostavljanjem konekcije, garantuje da ce podaci biti dostavljeni ispravno, u potpunosti i u redosledu u kome su poslani

UDP - (engl. User Datagram Protocol) protokol bez uspostavljanja konekcije i ne daje nikakve garancije o dostavljanju.

3. Istorijat Interneta. Prve ideje. ARPANET

-kasnih 50-ih godina americko ministarstvo odbrane je zeledo da uspostavi mrezu komunikacije projektovanu tako da moze da prezivi nuklearni rat.

-U to vreme vojne komunikacije su koristile javnu telefonsku mrezu, sto se smatralo veoma ranjivim (ukoliko dodje do kvara u malom broju cvorova, vecina komunikacije je presecena - organizacija mreze je predstavljena kao graf sa malim brojem unutrasnjih cvorova i mnogo listova).

-oko 1960. godine ministarstvo odbrane angazuje RAND korporaciju ciji radnik, Pol Baran (eng. Paul Baran) predlaze resenje koje podrazumeva vise putanja izmedju dva cvora (graf bez listova, svaki cvor je povezan bar sa jos 2, ima mnostvo ciklusa, pa uklanjanje neke grane ne prekida komunikaciju). Podaci od cvora do cvora putuju bilo kojom od dostupnih putanja. Posto su u tom slucaju neke putanje predugacke i analogni signal nije mogao da se salje tako daleko, predloženo je da se koristi digitalno paketno komutiranje (packet-switching). U Pentagonu je ovaj koncept prihvacen, medjutim, nakon konsultacija sa vodecom telefonskom kompanijom u SAD (AT&T), odbacen je (konzervativni ljudi u AT&T nisu zeledli da ih neki novajlija uci kako treba praviti komunikacioni sistem!)

-U oktobru 1957. kao odgovor na rusko lansiranje satelita Sputnjik, predsednik SAD Ajsenhauer (eng. Eisenhower) osniva ARPA (eng. Advanced Research Project Agency - agencija ciji je zadatak bio da subvencionise istrazivanja pri univerzitetima i kompanijama cije se ideje cine obecavajuće).

-1967. godine, direktor ARPA Lari Roberts (eng. Larry Roberts), odlucuje da jedan od zadataka ARPA treba da bude i ulaganje u komunikacije. Nailazi se na ranije odbacen Baranov rad, ciji je minijaturni prototip vec bio implementiran u Velikoj Britaniji. Nakon uvida u ova pozitivna iskustva, Roberts odlucuje da sagradi mrezu koja ce biti poznata pod imenom ARPANET. Svaki cvor mreze se sastojao od mini racunara (hosta) na koji je nadogradjen uredaj pod imenom IMP (eng. Interface Message Processor). Kako bi se povecala pouzdanost, svaki IMP je bio povezan bar sa jos dva udaljena IMP-a. Udaljeni IMP-ovi su medjusobno bili povezani zicanim komunikacionim linijama (brzine 56Kbps — najbrzim koji su u to vreme mogli da se zamisle). Poruke koje su slane izmedu hostova su se delile na pakete fiksirane duzine i svaki paket je mogao da putuje alternativnim putanjama. Svaki paket je morao u potpunosti da bude primljen u jedan IMP pre nego sto se prosledi sledecem. Dakle, ARPANET je bila prva **store-and-forward packet-switching** mreza.

Tender za izgradnju dobila je americka kompanija BBN. U pisanju softvera ucestovao je i odredjen broj postdiplomaca, bez ucesca velikih eksperata i kompanija. Mreza je prvi put javno prikazana u decembru 1969. godine sa cetiri povezana cvora: UCLA (University of California at Los Angeles), UCSB (University of California at Santa Barbara), SRI (Stanford Research Institute) i UU (University of Utah).

-Mreza je izrazito brzo rasla i do kraja 1972. godine bilo je povezano cetrdesetak velikih cvorova u SAD. Kako bi se pomoglo rastu ARPANET-a, ARPA je takode finansirala i istrazivanja na polju satelitskih komunikacija i pokretnih radio mreza. Uskoro se uvidelo da je za dalji rast mreze uz mogucnost koriscenja razlicitih komunikacionih tehnologija potrebno ustanoviti i kvalitetne komunikacione protokole.

-1974. godine dizajniran je TCP/IP model i protokol. Kako bi se ubrzalo sirenje ovog protokola, ARPA daje zadatak kompaniji BBN i univerzitetu Berkley da ugrade softversku podrsku ovih protokola u Berkley Unix operativni sistem, sto je i uradjeno kroz uvodjenje programskog interfejsa za mrežno programiranje (tzv. sockets) i izgradnju niza aplikacija za rad u mreznom okruzenju.

-Tokom 1980-tih veliki broj dodatnih mreza, narocito LAN, je povezan na ARPANET. Povecanjem dimenzije pronalazenje odgovarajućeg hosta postaje problematichno i uvodi se DNS (Domain Name System).

4. Istorijat Interneta. NSFNET. Mreza svih mreza

-Kasnih 1970-tih, americka nacionalna naucna fondacija (U.S. National Science Foundation - NSF) uvidja ogroman pozitivan uticaj ARPANET-a na razvoj nauke kroz omogucavanje udaljenim istrazivacima da dele podatke i ucestvuju u zajednickim istrazivanjima. Ipak, kako bi neki univerzitet mogao da koristi ARPANET, neophodno je bilo da ima ugovor sa ministarstvom odbrane sto mnogi univerziteti nisu imali. NSF odlucuje da se izgradi naslednik ARPANET mreze koja bi omogucila slobodan pristup svim univerzitetskim istrazivackim grupama.

-Projekat je zapoceo izgradnjom kicme mreze (eng. backbone) koja je povezivala sest velikih racunarskih centara u SAD. Superracunarima su prikljuceni komunikacioni uredaji koji su nazivani fuzzball (poput IMP u slucaju ARPANET). Hardverska tehnologija je bila identicna tehnologiji koriscenoj za ARPANET, medjutim, softver se razlikovao — mreza je odmah bila zasnovana na TCP/IP protokolu.

-Pored kicme, NSF je izgradio i dvadesetak regionalnih mreza koje su povezane na kicmu cime je zvanicno izgradjena mreza poznata kao NSFNET. Ova mreza je prikljucena na ARPANET povezivanjem fuzzball i IMP na univerzitetu CMU (Carnegie-Mellon University). NSFNET je bio veliki uspeh i komunikaciona tehnologija u kicmi mreze je kroz nekoliko faza prosirivana i unapredivana do brzina od 1.5Mbps pocetkom 1990-tih.

-Vremenom se shvatilo da Vlada SAD nema mogucnost samostalnog finansiranja odrzavanja i prosirivanja NSFNET mreze. Odluceno je da se mreza preda komercijalnim kompanijama koje bi, uz ostvarivanje sopstvenog profita, izvršile znacajne investicije u razvoj. Ovo se pokazuje kao dobar potez i 1990-tih godina, ukljucivanjem komercijalnih kompanija, brzina komunikacije u okviru NSFNET kicme, povecana je sa 1.5Mbps na 45Mbps.

-Paralelno sa razvojem ARPANET-a i NSFNET-a, i na ostalim kontinentima nastaju mreze pravljene po uzoru njih (npr. u Evropi su izgradjene EuropaNET i EBONE). Sve ove postepeno bivaju povezane u jedinstvenu svetsku mrezu. Sredinom 1980-tih godina ljudi pocinju da ovu kolekciju razlicitih spojenih mreza posmatraju kao medjumrezu (eng. internet), a kasnije i kao jedinstveni svetski entitet - **Internet**. Danas se moze se smatrati da je uredaj prikljucen na Internet ukoliko koristi softver koji komunicira TCP/IP protokolima, ima IP adresu i moze da salje IP pakete ostalim masinama na Internetu. Klijent se povezuje, nekom od pristupnih tehnologija, u ovom slucaju modemskim pristupom sa racunarom njegovog dobavljacka Interneta (eng. Internet Service Provider - ISP). ISP odrzava regionalnu mrezu svojih rutera i povezan je na neku od kicmi Interneta. Razlicite kicme su povezane u okviru NAP - stanice rutera koji pripadaju razlicitim kicmama, a u okviru NAP su povezani brzom LAN vezom.

5. Tehnologije pristupa Internetu. POTS.

Tehnologije pristupa (access network) internetu su deo internet infrastrukture izmedju host racunara i prvog rutera, nekad se naziva i lokalna petlja (local loop).

Iako predstavlja jako mali procenat geografske razdaljine koju podaci prelaze, cesto predstavlja usko grlo u komunikaciji.

Komunikacija u ovom delu se obicno vrši koriscenjem zastarele postojece infrastrukture fiksne tehnologije i vrši se na analogan nacin. Promene u ovom polju i napredaj tehnologije je danas svuda vidljiv.

Modemski pristup: koriscenje vec postojece infrastrukture fiksne telefonije (plain old telephone system - POTS). Kako bi se uspostavila veza potrebno je nazvati broj telefona, tako da ovo spada u grupu pozivnog povezivanja (dial up). Podrazumeva postojanje parica koje povezuju udaljene tacke prenosom analognog signala. Racunar se prikljucuje na telefonsku infrastrukturu pomocu uredjaja koji se naziva modem i ima

zadatak da konvertuje analogni signal u digitalni i obrnuto. Na drugom kraju u okviru provajdera se nalazi sličan modem koji je povezan na ruter uključen u internet mrežu. Fizičke karakteristike telefonske mreže ograničavaju brzinu (standardna brzina modema je 56Kbps). (problem nastaje jer se filtriraju frekvencije van pojasa ljudskog glasa, jer je mreža bila prvobitno namenjena samo za telefoniranje, pa je prenos podataka usko ograničen)

6. Tehnologije pristupa Internetu. DSL.

DSL (Digital subscriber line) - digitalna pretplatna linija: tehnologija za istovremeni prenos glasovnog signala i digitalnih podataka velikim brzinama preko parica fiksne telefonske mreže. Ovo znači da korisnici istovremeno mogu i da telefoniraju i da prenose podatke, što ranije nije bilo moguće. DSL ostvaruje stalnu vezu i nema potrebe za okretanjem broja prilikom uspostavljanja veze (nije dial up). Kako bi se rešio problem prethodnog pristupa, filteri se modifikuju i ne vrši se odsecanje frekvencija, pa opseg zavisi samo od dužine kabla. Problem ove tehnologije nemogućnost instalacije na mestima koja su previše udaljena od telefonske centrale (obično do 4km). Prosireni frekvencijski raspon (obično preko 1MHz) se deli na pojaseve širine 4KHz, i svaki nezavisno koristi (multipleksovanje deljenjem frekvencija). Pojasevi se raspoređuju najčešće 1 za telefon - prenos glasa, 2 za kontrolu prenosa podataka i svi ostali (njih oko 250) za prenos podataka. Obično se više njih alokira za preuzimanje (download) nego slanje podataka (upload), pa se takva mreža naziva asimetrična - ADSL. Brzina prenosa podataka, obično je do 16Mbps u dolaznom i 1Mbps u odlaznom saobraćaju.

Na korisnikovom kraju se instalira spliter (splitter - razdelnik) koji usmerava prvi pojas ka telefonskom uređaju, a ostale ka računaru. Između računara i splitera se nalazi ADSL modem koji ima zadatak da deli i objedinjuje podatke na velikom broju komunikacionih kanala. Sličan uređaj se nalazi na drugom kraju, prihvata podatke od velikog broja korisnika i šalje ka provajderu.

7. Tehnologije pristupa Internetu. ISDN.

ISDN (Integrated Services Digital Network): Slično DSL tehnologiji, uvodi direktne digitalne veze zasnovane na zicama javne telefonije kojima se istovremeno prenosi glasovni signal i digitalni podaci (na zasebnim kanalima), što korisnicima omogućava da istovremeno razgovaraju telefonom i koriste mrežu. Za razliku od DSL, ISDN zahteva uspostavljanje veze pozivom broja tako da spada u grupu dial up pristupa. Brzina prenosa podataka je obično 128Kbps. ISDN je danas u velikoj meri potisnut od strane DSL tehnologije jer zahteva kompleksnije promene u postojećoj telefonskoj infrastrukturi, a ne donosi značajno povećanje brzine prenosa podataka u odnosu na dial up.

8. Tehnologije pristupa Internetu. HFC.

HFC (Hybrid fibre-coaxial): Opticko-kabloske mreže su mreže koje se zasnivaju na kombinovanom prenosu podataka kroz optička vlakna i koaksijalne kablove koje se koriste za istovremeni prenos televizijskog signala, radio signala, i digitalnih podataka. Ruter u centrali provajdera se povezuje optičkim kablovima sa cvorovima, koji su dalje povezani sa korisnicima koaksijalnim kablovima (obično već postojećih kablova kablovske televizije). Signal iz koaksijalnih kablova se zatim razdeljuje na radio i TV signal i na digitalne podatke. Veza sa računarnom se ostvaruje preko tzv. kablovskog modema. Na jedan cvor, obično se povezuje oko 500 korisnika. Signal u kablovima se obično prostire radio talasima frekvencije između 5MHz i 1GHz (obično se

pocetni pojas sirine nekoliko desetina MHz koristi za odlazni saobracaj, a ostatak frekvencijskog pojasa se koristi za dolazni saobracaj). Slicno kao kod DSL, frekvencijski opseg se deli na pojaseve koji se alociraju za prenos razlicitih vrsta signala i podataka. Jako je vazno napomenuti da svi korisnici povezani na lokalni cvor dele komunikacioni kanal i svi dolazni paketi bivaju dostavljeni istovremeno svim kablovskim modemima prikljucenim na isti cvor. Zbog ovoga, brzina prenosa moze da varira u zavisnosti od aktivnosti korisnika prikljucenih na lokalni cvor (obicno je brzina dolaznog 100Mbps, a odlaznog 40Mbps).

9. Tehnologije pristupa Internetu. Mreze mobilne telefonije.

Razvoj mobilne telefonije karakterise se generacijama. U prvoj generaciji vrsen je analogni prenos glasa.

U drugoj digitalni prenos glasa, dok se u okviru trece generacije vrsi digitalni prenos glasa i podataka. Starija General Packet Radio Service (GPRS) tehnologija je omogucavala brzine od samo 56 do 114Kbps i koriscena je za prenos podataka u okviru ove generacije (tzv. 2.5G).

U okviru trece generacije mobilne telefonije vrsi se digitalni prenos glasa i podataka. Tehnologije koje se u okviru trece generacije najvise koriste su High Speed Packet Access (HSPA) koje omogucavaju brzine prenosa i do 14Mbps u dolaznom i 6Mbps u odlaznom saobracaju. HSPA je unapredjenje Wideband Code Division Multiple Access (W-CDMA) tehnologije.

U okviru cetvrte generacije se koristi tehnologija Long Term Evolution (LTE) sa brzinom preuzimanja do 105Mbps i postavljanja sadrzaja do 30Mbps, omogucuje HD i 4K (gledanje sadrzaja u visokoj rezoluciji).

Trenutno je aktivna peta generacija i jos je u razvoju. Tehnologije pristupa internetu koje koriste postojece mreze mobilne telefonije u novije vreme postaju sve naprednije i sve sire koriscene.

10. Internet servisi. Elektronska posta.

Broj razlicitih servisa koje nudi Internet vremenom raste.

Osnovni servisi prisutni jos iz doba ARPANET-a su: elektronska posta, diskusione grupe, upravljanje racunarima na daljinu i prenos datoteka. Naravno, najpopularniji servis interneta je VEB (kasnije vise o njemu).

Elektronska posta (eng. e-mail) predstavlja jedan od najstarijih servisa Interneta. Elektronska posta funkcioniše tako sto svaki korisnik poseduje svoje „postansko sanduce” (eng. mailbox) na nekom serveru. Sanduce jedinstveno identifikuje elektronska adresa koja obavezno sadrzi znak @ koji razdvaja ime korisnika, od domena servera elektronske poste.

Poruke koje se salju su u tekstualnom formatu ali mogu da obuhvate i priloge u proizvoljnom formatu. Slanje i primanje poste korisnik obicno obavlja preko klijenta instaliranog na svom racunaru. Najpoznatiji klijenti za elektronsku postu danas su Microsoft Office Outlook, Microsoft Outlook Express, Apple Mail, Mozilla Thunderbird...

Znacajan obim elektronske poste se odvija preko javnih servisa elektronske poste vezanih za Veb koji ne zahtevaju koriscenje posebnog klijenta elektronske poste, vec se rad sa elektronskom postom obavlja koriscenjem Veb aplikacija. Servisi ovog tipa su Yahoo! Mail, Microsoft Hotmail, Google Gmail...

U slanje jedne elektronske poruke obično su uključena četiri računara. To su: računar osobe koja šalje poruku, server e-pošte osobe koja šalje poruku (kod provajdera), server e-pošte osobe koja prima poruku i računar osobe koja prima poruku (ovim redosledom se prenose podaci).

Za slanje elektronske pošte koristi se protokol SMTP (Simple Mail Transfer Protocol).

Za primanje elektronske pošte koriste se protokoli: POP3 (Post Office Protocol) i IMAP (Internet Message Access Protocol).

Svi oni (SMTP, POP3 i IMAP) koriste TCP (Transmission Control Protocol) protokol na transportnom nivou.

Diskusione grupe (engl. usenet) - predstavljaju distribuirani Internet sistem za diskusije koji datira još od 1980. godine.

Korisnici mogu da čitaju i šalju javne poruke. Poruke se smeštaju na specijalizovane servere (news server). Diskusije su podeljene u grupe (newsgroups) po određenim temama, koje se imenuju hijerarhijski. Na primer, *sci.math* označava grupu za diskusije na temu matematičke nauke.

Pristup diskusionim grupama se vrši korišćenjem specijalizovanog softvera (newsreader). Obično su klijenti elektronske pošte istovremeno i klijenti za korišćenje diskusionih grupa. Iako u današnje vreme veb forumi predstavljaju alternativni način diskusija, diskusione grupe se i dalje koriste u značajnoj meri.

11. Internet servisi. Udaljen pristup.

Prijavljivanje na udaljene računare (remote login) je jedan od najstarijih servisa Interneta.

Ovaj servis omogućava korisnicima (tj. klijentima) da se korišćenjem Interneta prijave na udaljeni računar (server) i da nakon uspešnog prijavljivanja rade na računaru kao da je u pitanju lokalni računar.

Kako to funkcioniše?

-Korisnik dobija terminal kojim upravlja udaljenim računarom izdajući komande. Udaljeni računar prima komande i izvršava ih korišćenjem svojih resursa, a rezultate šalje nazad klijentu koji ih korisniku prikazuje u okviru terminala.

Prijavljivanje na udaljeni računar se obično vrši preko Telnet protokola i SSH (Secure Shell) protokola.

Pored toga, korisnicima je omogućen i udaljen pristup u kome je korisniku na raspolaganju kompletan GUI (Graphical User Interface) udaljenog računara (remote desktop).

12. Internet servisi. Prenos datoteka.

Prenos datoteka (eng. file transfer) predstavlja jedan od klasičnih servisa Interneta i datira još od ranih 1970-tih. Prenos datoteka se vrši između klijentskog računara i serverskog računara u oba smera (mogu se preuzimati i postavljati datoteke na server). Ovaj servis se danas obično koristi za postavljanje datoteka na veb servere, kao i za preuzimanje velikih binarnih datoteka.

Serveri koji čuvaju kolekcije datoteka obično se identifikuju adresom koja počinje sa **ftp** (slično kao što se veb serveri identifikuju adresom koja počinje sa **www**).

Za prenos datoteka koristi se FTP (File Transfer Protocol) protokol, kao i SCP (Secure Copy Protocol) i SFTP (SSH File Transfer Protocol) protokoli bazirani na SSH (Secure Shell) koji nude enkripciju (sifrovanje - izmena podataka) pri prenosu datoteka.

Za prenos datoteka, na klijentskim racunarima se obicno koriste programi poput **ftp** (komandni program koji direktno implementira FTP protokol), **scp** (komandni program koji kopira datoteke uz koriscenje enkripcije), zatim **Veb pregledaci** (koji omogucavaju preuzimanje datoteka sa FTP servera), **klijenti** poput GnuFTP, Windows Commander i slicno.

13. Internet servisi. Veb. Istorijat, nacin funkcionisanja

World Wide Web, tj. Veb je Internet servis nastao tek ranih 1990-tih godina, medjutim veoma brzo je stekao ogromnu popularnost i postao je najznacajniji Internet servis danasnjice. To je sistem medjusobno povezanih dokumenata poznatih kao Veb stranice koje mogu da sadrze tekst, slike, video snimke i druge multimedijalne materijale. Veb stranice su povezane koriscenjem veza (linkova), tj. predstavljaju hipertekst.

Stranice se cuvaju na specijalizovanim Veb serverima a onda se na zahtev klijenata prenose na klijenske racunare, gde ih specijalizovani programi prikazuju. Ovi programi nazivaju se **pregledaci Veba** (eng. Web browsers). Najpoznatiji pregledaci danas su Microsoft Internet Explorer, Mozilla Firefox, Google Chrome, Safari, Opera..., dok je prvi bio Mosaic. Dostava Veb sadrzaja je zasnovana na HTTP protokolu i HTTPS protokolu (pruza dodatnu sigurnost jer se podaci salju u sifrovanom obliku).

Neprofitna organizacija W3C ima kao svoj zadatak kanalisnje daljeg razvoja veba i koordinacije industrijskih poizvodjaca softvera.

14. Internet servisi. Veb. Tipovi strana

Veb sajt (web site) je kolekcija veb stranica povezanog sadrzaja.

Veb stranicama su pridružene **URI adrese** (Uniform Resource Identifier) koje imaju sledeci izgled: oznaka protokola + ime domena ili IP adresa servera + putanja do resursa na internetu, npr.

<http://www.server.com/data/grafik.pdf> .

o Komunikacija između pregledaca i servera:

1. Odredjuje se IP adresa servera
2. Serveru se šalje HTTP zahtev s nazivom i lokacijom zahtevane strane
3. Server proverava da li postoji strana i ako postoji salje je u vidu HTTP odgovora
4. Klijent analizira HTML opis i ako se u njemu referise na sliku, audio ili video zapis, salje novi HTTP zahtev za resursima na koje se referise
5. Ako veb-server ne moze da pruzi zahtevanu stranu, HTTP odgovor sadrzi informaciju o tome (npr. kod greske 404 oznacava da resurs nije pronadjen)

o Tipovi veb stranica:

1. staticke veb stranice - prikazuju unapred pripremljen sadrzaj
2. veb stranice sa procesiranjem na strani servera
3. veb stranice sa procesiranjem na strani klijenta

15. Internet servisi. Veb. Veb pretraživaci

Veb sadrži enormnu količinu informacija i ne bi funkcionisao bez veb pretraživaca. Najpopularniji veb pretraživač je Google.

Veb pretraživaci sadrže komponentu pod nazivom pauk ili kroler (crawler) koja s vremena na vreme preuzima stranice i skladišti ih.

Algoritmi rangiranja stranica: Page Rank (Google) – brojanje veza koje vode ka stranici

SEO (search engine optimization) – razne taktike za obezbeđivanje da veb sajt bude prikazan među prvim rezultatima pretrage, postoje razne tehnike za poboljšanja pretrage

Veb portali pružaju relevantne informacije za određenu temu, prikupljene iz različitih izvora.

16. Internet servisi. Veb. Veb dizajn

Broj veb sajtova je ogroman i velika je međusobna konkurencija, pa je veb dizajn veoma važan.

Veb sajt treba da bude funkcionalan, bogat sadržajem, vizuelno dopadljiv.

Savremeni veb dizajn uključuje i internet marketing i SEO i vestine kreiranja ugodnog korisničkog interfejsa. Prilikom izrade veb sajta potrebno je osmisliti njegovu logičku organizaciju, a kasnije se posvetiti pitanjima estetskog dizajna.

Mnogi principi su nepromenljivi: boje teksta i pozadine treba da budu kontrastne, najvažnije stvari istaknute.. Trendovi se menjaju: danas popularan minimalistički dizajn.

17. Internet servisi. Skladista datoteka.

Kompanije nude usluge skladištenja podataka u „oblaku“, tj. u skladištima, tzv. repozitorijumima na serverima tih kompanija.

Sa različitih računara i uređaja korisnik ima pristup svim svojim podacima.

Sinhronizacija podataka sa serverima vrši se automatski. Sadržajima u skladistu je moguće pristupiti preko veba, bilo koriscenjem pregledaca, bilo aplikacija za pametne telefone. Najpopularnija skladišta datoteka su: Dropbox i Google Drive.

18. Internet servisi. Časkanje.

Časkanje (chat) - korisnicima Interneta omogućava da „pričaju“ kucanjem uživo (on-line) :

- 1) pristupom sobama za daskanje (chat room) - grupna ili privatna komunikacija
- 2) Daskanje je u današnje vreme zasnovano ili na specifičnim protokolima (npr. IRC) i aplikacijama (npr. Xchat, mIRC) ili se koriste veb-zasnovane sobe za daskanje

Instant poruke (instant messaging) - osnovna razlika u odnosu na daskanje je da se instant poruke uglavnom razmenjuju „oči-u-oči“ između poznanika, dok daskanje obično podrazumeva grupnu komunikaciju u sobi za daskanje. (Google Talk, Microsoft MSN, .. ili preko veba npr. Facebook chat)

Pojedini servisi: Skype, Viber, WhatsApp, Slack nude obe mogućnosti.

19. Internet servisi. VoIP.

VoIP servisi i programi omogućuju glasovnu i video-komunikaciju između udaljenih osoba preko Interneta.

Moguće je pozivanje onih poznanika koji su tog trenutka priključeni na ovaj servis, a moguće je i povezivanje ovih servisa i sa klasičnom telefonijom.

Najpopularniji servisi ovog tipa su Skype, Viber, WhatsApp, Google Talk, Telegram itd.

20. Internet servisi. P2P.

Peer-to-peer (P2P) servisi:

- Popularizacija P2P servisa desila se 1999. kada je servis pod imenom Napster iskorišten za razmenu velike količine muzičkih MP3 datoteka između velikog broja korisnika širom sveta.
- S obzirom na kršenje autorskih prava Napster je već 2001. zabranjen, ali je nastao veliki broj P2P protokola i aplikacija.
- Za razliku od većine Internet servisa koji funkcionišu po klijent-server modelu komunikacije, P2P servisi se zasnivaju na direktnoj razmeni podataka između različitih klijenata, pri čemu serveri samo služe za koordinaciju komunikacije, bez direktnog kontakta sa samim podacima koji se razmenjuju.
- P2P servisi se obično koriste za razmenu velikih datoteka (obično video i audio sadržaja).
- P2P aplikacije čine ogroman deo Internet saobraćaja.
- Najkorišteniji P2P servisi i protokoli danas su Bittorrent, DC++, Gnutella, G2, E-mule, KaZaA (FastTrack), itd.
- Postoji veliki broj aplikacija koje korisnicima omogućuju korišćenje ovih protokola

21. Internet servisi. Forumi, blogovi i društvene mreže.

Forum (internet forum) - korisnicima omogućavaju diskusiju, koja je organizovana u nitima (Primeri: EliteSecurity, MyCity, itd.)

Blogovi (weB LOG) - korisnici objavljuju svoja razmišljanja o nekoj temi.

Društvene mreže - omogućavaju povezivanje sa nalogima prijatelja i poznanika ili sa nalogima ličnosti iz javne sfere

- Iako su sastavni deo veba, u poslednje vreme društvene mreže doživljavaju izrazitu ekspanziju i imaju sve veći i veći društveni značaj.
- Najkorištenije socijalne mreže današnjice su Facebook, Twitter, Google+, MySpace itd.
- Postoje i socijalne mreže sa specijalizacijom, npr. LinkedIn, Foursquare, itd.
- Izuzetno dinamična dešavanja i promene – primeri YouTube, Instagram itd.

22. Internet servisi. Geografski informacioni sistemi

Geografski informacioni sistemi (GIS) - sistemi koji sadrže geografske informacije: mape, satelitske snimke, baze podataka sa interesantnim geografskim tačkama (imena ulica, pozicija stajališta)

Danas na internetu postoje svima dostupni sajtovi koji nude funkcionalnosti GIS sistema: Google Maps, Google Earth, Bing Maps, (u Srbiji PlanPlus, B92 mape) .

Pametni telefoni opremljeni sistemom globalnog pozicioniranja (GPS) doprinose korišćenju mapa za određivanje trenutne pozicije i davanje instrukcija kako stidi do željene destinacije.

23. Internet servisi. Elektronska trgovina i bankarstvo.

Elektronska trgovina sve više zamenjuje klasične oblike trgovine.

Tri vrste poslovanja:

- B2C** (business to customer) - kompanije prodaju svoju robu/usluge pojedinačnim kupcima
- B2B** (business to business) - kompanije prodaju svoje usluge drugim kompanijama
- C2C** (customer to customer) - prodavci pojedinačno prodaju svoju robu/usluge pojedinačnim kupcima

Banke danas pružaju usluge elektronskog bankarstva:

- provera stanja na računu, uplata, isplata, prenos sredstava sa računa na račun
- usluga elektronskog pladanja računa
- važno pitanje sigurnosti – obično dvofaktorska autentifikacija.

24. Internet servisi. Elektronsko učenje.

Elektronsko učenje (e-learning) podrazumeva korišćenje informacionih tehnologija, veba i interneta u oblasti obrazovanja.

Sistemi za upravljanje učenjem (learning management systems, LMS) omogućavaju da nastavnici ostave elektronski nastavni materijal, organizuju testiranje - primer Moodle.

Masovni slobodno dostupni kursevi na internetu (massive open online courses, MOOC) organizuju se u potpunosti elektronski – primer sajtovi Coursera, UDACITY, edX, MIT OpenCourseWare, itd.

3. Mrežni protokoli

1. Slojevi kod mreža. Referentni modeli OSI i TCP/IP

Internet protokol je protokol koji se koristi za komunikaciju u okviru mrežnog sloja Interneta. Dve osnovne verzije ovog protokola su IPv4 i IPv6. Fizički sloj, dakle, od sloja veze podataka dobija zadatak da preko komunikacionog medijuma prenese sekvencu bitova. To je najniži nivo komunikacije - proučava se mehanizam slanja pojedinačnih bitova od jednog do drugog uređaja kroz komunikacioni medijum.

Broj slojeva se razlikuje od mreže do mreže i na svakom sloju se sprovodi odgovarajući protokol komunikacije. Protokol je dogovor dve strane o načinu komunikacije (narusavanje protokola čini komunikaciju nemogućom).

Istorijski se mreže posmatraju u okviru dva referentna modela:

1) *OSI* (Open System Interconnection) - model sa 7 slojeva standardizovan od strane ISO (medjunarodne organizacije za standardizaciju). Iako se konkretni protokoli koje definiše više ne koriste, predstavlja značajan apstraktni opis mreža.

2) *TCP/IP* - model sa 4 sloja, prisutan u okviru Interneta. Za razliku od OSI modela, njegova apstraktna svojstva se ne koriste značajno, dok se konkretni protokoli intenzivno koriste i predstavljaju osnovu interneta.

Slojevi OSI modela (u zagradi su analogni slojevi TCP/IP modela):

- 1) Application - aplikativni (isto aplikativni) = definiše protokole
- 2) Presentation (ne postoji) = šalje podatke od aplikativnog ka sesiji, može da kodira/kompresuje/enkriptuje
- 3) Session (ne postoji) = uspostavljanje sesije između trenutno pokrenutih programa koji komuniciraju
- 4) Transport (isto transportni) = deli prihvaćene podatke sa viših slojeva na manje pakete, šalje ih
- 5) Network - mrežni (Internet) = povezivanje razunara na mrežu
- 6) Data link - (Host-to-network) = visim signalima obezbeđuje postojanje pouzdanog kanala komunikacije
- 7) Physical - fizički (Host-to-network) = najniži

2. TCP/IP komunikacija. Sloj pristupa mreži (host-to-network)

Sloj „Host-to-network“ obezbeđuje kanal komunikacije.

Na najnižem nivou, obezbeđuje postojanje komunikacionog kanala i mogućnost slanja i primanja pojedinačnih bitova kroz komunikacioni kanal.

Na najnižem nivou u okviru ovog sloja nema kontrole grešaka.

Na višem nivou se međumrežnom sloju obezbeđuje postojanje pouzdanog kanala komunikacije u kome se:

- greške automatski detektuju i ispravljaju (error control)
- automatski se vodi računa o brzini slanja podataka kako se ne bi desilo da brzi uređaji zagušuju sporije (flow control)

Ukoliko se koristi zajednički kanal komunikacije, na ovom sloju se vrši kontrola pristupa uređaja komunikacionom kanalu (medium access control)

Ovde se, gledano na najnižem nivou, dobija zadatak da se preko komunikacionog medijuma prenese sekvenca bitova, na tom najnižem nivou komunikacije se proučava mehanizam slanja pojedinačnih bitova od jednog do drugog uređaja kroz komunikacioni medijum

Najniži nivo komunikacije karakteriše potreba za velikom efikasnošću

Način komunikacije na tom najnižem nivou zavisi od tipa komunikacionog medijuma - žičana ili bežična veza, koja vrsta kablova je u pitanju i sl.

U okviru lokalne mreže komunikacija se zasniva na tehnologijama:

- Ethernet (žičano povezivanje)
- Wi-Fi (bežično povezivanje)

Brzina prenosa podataka u ovakvim mrežama veda od 1Gbps.

Ovaj sloj od uređaja koji rade na međumrežnom sloju dobija zadatak da se paket (u IP terminologiji, taj paket se naziva IP datagram) prenese:

- sa jednog rutera na drugi
- sa jednog uređaja na drugi u okviru lokalne mreže

Taj zadatak se realizuje tako što se IP datagram se obmotava dodatnim podacima i kreiraju se okviri (frame)

3. TCP/IP komunikacija. Struktura okvira (frame). Okviri i IP datagrami

Zadatak da se paket (u IP terminologiji, taj paket se naziva IP datagram) prenese:

- sa jednog rutera na drugi
- sa jednog uređaja na drugi u okviru lokalne mreže

se realizuje tako što se IP datagram obmotava dodatnim podacima i kreiraju se okviri (frame)

Potrebno je sprečiti izmenu podataka prilikom mrežnog prenosa (preskakanje bitova, izmena bitova, ponavljanje, ...)

Na kraj okvira dodaje se sekvenca za proveru okvira:

- omogućava primaocu da proveru da li je došlo do greške
- neke greške se mogu ispraviti

Mogude je detektovati i ispraviti složenije greške korišćenjem sekvenci od više bitova, kodiranih kodovima za otkrivanje i ispravljanje grešaka

Na ovom sloju koriste se MAC adrese

- Predstavljaju se pomoću 48 bita
- Zapisuju se u obliku 6 dvocifrenih heksadekadnih brojeva (primer: 2c:d4:44:a8:be:3b)

Na početak okvira dodaju se MAC adresa primaoca i pošiljaoca

Ako se u okviru nalaze IP datagrami, tada okvir sadrži i IP adrese primaoca i pošiljaoca, ali one se na ovom nivou ne analiziraju

4. TCP/IP komunikacija. Medjumrežni sloj (internet). IPv4 i IPv6 protokoli

Medjumrežni sloj (internet layer) bavi se povezivanjem više računara u mrežu. Osnovni zadatak u okviru ovog sloja je rutiranje (routing), tj. određivanja putanja paketa koji putuju kroz mrežu kako bi se odredio efikasan način da stignu na svoje odredište

Kako bi se odredila putanja, neophodno je uvođenje sistema adresiranja

Ukoliko se povezuju heterogene mreže (sa različitim shemama adresiranja), na ovom sloju se vrši prevođenje adresa (Na primer, na nižim slojevima se obično koriste fizičke MAC adrese, a na višim IP adrese)

Svaki čvor u mreži uključen u komunikaciju mora da implementira mrežni protokol, da razume odredišnu adresu i da na osnovu ovoga odluči kome da prosledi primljenu poruku

Najpoznatiji protokol ovog sloja je koji se koristi u okviru Interneta je Internet Protocol (IP)

Dve osnovne verzije ovog protokola su IPv4 i IPv6

Iz istorijskih razloga i vede preglednosti u nastavku de detaljnije biti opisana IPv4 verzija IP protokola

Osnovni zadatak ovog protokola je da pokuša da dopremi (tj. rutira) paket od izvora do odredišta u okviru mreže sa paketnim komutiranjem, isključivo na osnovu navedene adrese, bez obzira da li su izvor i odredište u okviru iste mreže ili između njih postoji jedna ili više drugih mreža

Protokol ne daje nikakve garancije da de paketi zaista i biti dopremljeni, ne daje garancije o ispravnosti dopremljenih paketa, ne garantuje da de paketi biti dopremljeni u istom redosledu u kojem su poslani i slično

- Garancije ovog tipa obezbeđuju se na višim slojevima komunikacije

Pri prosleđivanju paketa sa transportnog sloja na ovaj sloj dodaju se:

- adresa pošiljaoca,
- adresa primaoca, ...

5. TCP/IP komunikacija. IPv4. Struktura IP datagrama. IP datagrami i segmenti

IP datagram (paket) - ide od pošiljaoca do primaoca, preko serije rutera

IP adrese su strukturirane hijerarhijski: adresa se deli na bitove koji adresiraju mrežu (vodeći) i bitove koji adresiraju uređaj u okviru mreže

Paket se dostavlja:

- korišćenjem lokalnog mrežnog saobraćaja
- šalje se van mreže "u svet" - preko određenog rutera koji se naziva izlazna kapija (gateway)

Svi uređaji iz iste mreže dele zajednički početak IP adrese

- Primer: od 147.91.67.0 do 147.91.67.255 - ista prva 24 bita, razlikuju se poslednjih 8

6. TCP/IP komunikacija. IPv4. Adrese kod IPv4 protokola

IP protokol uvodi sistem adresa poznatih kao IP adrese. U okviru IPv4, adrese su 32-bitni neoznačeni brojevi, koji se obično predstavljaju kao 4 dekadno zapisana broja između 0 i 255. Postoji ukupno 2^{32} adresa IPv4, što se pokazuje kao nedovoljno. IPv6 donosi 128-bitne adrese, što rešava ovaj problem.

Za dodelu IP adresa, zadužena je agencija Internet Assigned Numbers Authority (IANA), kao i pomodni regionalni registri (Regional Internet Registries - RIRs). Svaki uređaj priključen na Internet ima jedinstvenu IP adresu. Neki uređaji imaju uvek istu IP adresu (tzv. statički dodeljenu), dok se nekim uređajima dodeljuje različita adresa prilikom svakog povezivanja na mrežu (tzv. dinamička dodela)

- Na primer, studentski server Matematičkog fakulteta ima statički dodeljenu adresu 147.91.64.2 ili binarno zapisano 10010011 1011011 01000000 00000010

Statičke adrese su pogodnije za servere, inače su pogodnije dinamičke.

Ranije su IP adrese bile deljene na klase (A, B, C, D, E) i svaka klasa je definisala broj bita za prvi i broj bita za drugi deo deo IP adrese.

Adrese klase A (prvi bit u zapisu je 0 - između 0.0.0.0 i 27.255.255.255) su bile dodeljivane jako velikim mrežama (8+24 bita - 128 mreža sa mogućih preko 16.7 miliona korisnika)

Adrese klase B (počinje sa 10 - između 128.0.0.0 i 191.255.255.255) su bile dodeljivane srednjim mrežama (16+16 bita - preko 16 hiljada mreža sa mogućih 65536 korisnika)

Adrese klase C (počinje sa 110 - između 192.0.0.0 i 223.255.255.255) su bile dodeljivane malim mrežama (24+8 bita - preko dva miliona mreža sa mogućih 256 korisnika).

Vremenom se pokazalo da ovakva organizacija nije skalabilna

Obično su mreže kompanija imale potrebu za više od 256 uređaja, tako su uzimale adrese klase B, pa je veliki broj adresa ostajao nedodeljen

Dva načina zapisa skalabilnog zapisa IP adresa:

- CIDR notacija - adresa 147.91.67.138/24
- Maska podmreže (subnet mask) - uz adresu 147.91.67.138 navodi se maska podmreže 255.255.255.0 (24 jedinice i 8 nula)

7. IPv4. Povezivanje uređaja u lokalnoj mreži. Specijalne adrese.

U okviru svake mreže postoje dve adrese sa specijalnom namenom:

-prva adresa (250.150.100.0) smatra se adresom mreže

-poslednja adresa (250.150.100.255) - adresa za javno emitovanje (broadcast address) - svaka poruka poslata na tu adresu dostavlja se svim uređajima u mreži

8. IPv4. Povezivanje uređaja u lokalnoj mreži. Mrežni hardver.

Elementi mrežnog hardvera koji se koriste:

Ruter (router) - kompleksniji uređaj namenjen povezivanju raznorodnih mreža i povezivanju mreža sa Internetom

Obično ima javnu IP adresu koju deli cela mreža

Koristi IP adrese za prosleđivanje paketa, što dopušta mrežnu komunikaciju po različitim protokolima

Prosleđuje pakete na osnovu softvera, dok svič radi hardverski

Podržava različite WAN tehnologije

Radi na međumrežnom sloju

Svič (switch) - povezuje dve ili više nezavisnih mreža

Postavljanjem sviča između povezanih uređaja poruka se prosleđuje samo uređaju kome je namenjena (efikasnija komunikacija)

Svič čuva tabelu koja preslikava MAC adrese priključenih uređaja na redne brojeve priključaka

Tabela se gradi i održava automatski tokom komunikacije

Podržava vodi broj ulaznih i izlaznih portova

Vrši kontrolu greške pre prosleđivanja paketa

U zavisnosti od tipa, realizuju prosleđivanje na nivou „host-prema-mreži“ (zasnovano na MAC adresama) i na međumrežnom nivou (zasnovano na IP adresama)

Pakete prosleđuje samo mreži u kojoj se nalazi primalac

Kod velikih mreža se svičevi koriste umesto habova za povezivanje računara u podmrežama

Hab (hub) - dobijene poruke prosleđuje svim priključenim uređajima

Postavljanje haba između povezanih uređaja - primljeni paketi se prosleđuju svim uređajima povezanim na njega (jednostavno, ali je verovatnoda sudara velika)

Ne može kontrolisati propuštanje paketa koje šalje povezanim uređajima

Ne može odrediti najbolji put za slanje paketa

Nisu efikasni

Koriste se u malim mrežama, sa niskim nivoom komunikacije

Radi na nivou sloja „host-prema-mreži“ – nisko, najbliže fizičkom sloju

Most (bridge) - povezuje lokalnu mrežu sa drugom lokalnim mrežom koja koristi isti protokol

Ima jedinstveni ulazni i jedinstveni izlazni port

Kontroliše propuštanje paketa na mreži na osnovu MAC adrese odredišta – ne šalje sve pakete bez kontrole

Pakete prosleđuje samo mreži u kojoj se nalazi primalac

Radi na nivou sloja „host-prema-mreži“

9. IPv4. Povezivanje uređaja u lokalnoj mreži. Protokol razrešavanja adresa.

Kako uređaj koji zna IP adresu primaoca određuje MAC adresu na koju prosleđuje IP datagram?

-na osnovu mrežne maske utvrđuje da li je primalac u istoj mreži; ako jeste šalje njemu, ako nije šalje izlaznoj kapiji

-u oba slučaja zna IP adresu uređaja u lokalnoj mreži

-za dobijanje adrese koristi se protokol razrešavanja adresa (address resolution protocol, ARP)

-javno se emituje ARP zahtev sa IP adresom uređaj sa tom IP adresom šalje ARP odgovor sa svojom MAC adresom

10. IPv4. Povezivanje uređaja u lokalnoj mreži. IP adrese i DHCP.

Dinamičke IP adrese se dodeljuju korišćenjem specijalizovanog protokola za dinamičku konfiguraciju (Dynamic Host Configuration Protocol - DHCP)

Specijalizovani server (tzv. DHCP server) je zadužen za skup IP adresa koje određuje administrator mreže i na zahtev uređaja koji se priključuje na mrežu dodeljuje mu neku u tom trenutku slobodnu adresu

Server se može konfigurisati tako da dodeljuje bilo koju slobodnu IP adresu, ili uvek istu adresu koja se određuje na osnovu MAC adrese uređaja koji zahteva IP adresu, i slično

11. IPv4. Povezivanje uređaja u lokalnoj mreži. Javne i privatne IP adrese.

Da ne bi došlo do nestašice IPv4 adresa uvode se privatne adrese:

- 10.0.0.0/8 (od 10.0.0.0 do 10.255.255.255) - 16.7 miliona adresa
- 172.16.0.0/12 (od 172.16.0.0 do 172.31.255.255) - milion adresa
- 192.168.0.0/16 (od 192.168.0.0 do 192.168.255.255) - 65536 adresa

Privatne adrese se koriste samo za lokalnu mrežnu komunikaciju

Prilikom pristupa Internetu:

- ruter (izlazna kapija) menja lokalnu adresu svojom (javnom) adresom
- primalac odgovor šalje nazad ruteru, a on menja adresu privatnom adresom uređaja koji je poslao zahtev i prosleđuje odgovor

Ovaj proces se naziva preslikavanja mrežnih adresa (network address translation - NAT)

Korišćenje NAT-a prilikom slanja paketa:

- U slučaju da ruter detektuje određenu adresu iz opsega adresa privatne mreže sa kojom je povezan, jasno je da je paket namenjen za lokalnu komunikaciju i šalje se jedinstvenom uređaju sa navedenom lokalnom adresom
- Ako je određena адреса javna, ruter adresu pošiljaoca zamenjuje svojom adresom (globalno jedinstvenom) i paket prosleđuje na određeno mesto.
- Korišćenje NAT-a prilikom prijema paketa:
 - U slučaju dolaznog paketa, nije odmah jasno na koju privatnu adresu je potrebno poslati paket koji je pristigao
 - Kako bi se ovo razrešilo, lokalna адреса se pakuje i postaje sastavni deo paketa koji se šalje
 - Ruter, pre prosleđivanja dolaznog paketa, vrši njegovo raspakivanje i određivanje lokalne adrese

Sve ovo narušava osnovne principe i koncepte IP protokola, pa se zato NAT smatra prelaznim rešenjem problema nestašice IP adresa, dok ne zaživi IPv6

12. IPv4. Rutiranje. Tabele rutiranja.

U vedim mrežama postoji veliki broj povezanih rutera

Uloga rutera: na osnovu IP adrese primaoca i na osnovu tabela koje su zapisane u njihovoj memoriji (tabela rutiranja) odrediti kome od povezanih čvorova treba proslediti paket da bi efikasno stigao do cilja

Tabele rutiranja sadrže spisak mrežnih adresa različitog nivoa hijerarhije i za svaku od njih kom uređaju treba dostaviti paket

Primer: Neka je u tabeli rutiranja rutera

- `0.0.0.0/0 via 200.170.10.10`
`200.0.0.0/8 via 200.100.5.20`
`200.160.0.0/16 is directly connected, Serial0/1`

- Ako ruter primi paket namenjen adresi 200.150.100.23, on se dostavlja preko rutera 200.100.5.20
- Šablonom 0.0.0.0/0 zadaje se gde proslediti paket ako адреса nije prepoznata na neki drugi način
- Traži se najpreciznije poklapanje sa šablonom - poklapanje sa najvedim brojem bitova

Kvalitet rutiranja zavisi od tabele rutiranja, one se mogu graditi statički i dinamički

13. TCP/IP komunikacija. Tansportni sloj.

Transportni sloj (transport layer) ima zadatak da prihvata podatke sa visih slojeva, deli ih na manje jedinice (pakete), salje te pakete na odrediste koriscenjem nizih slojeva.

Obično se na ovom sloju razlikuju dve vrste protokola: protokoli sa uspostavljanjem konekcije i protokoli bez uspostavljanja konekcije

- Protokoli koji zahtevaju uspostavljanje konekcije garantuju da de poslati podaci zaista i stidi na odredište u istom redosledu u kojem su i poslati
- Protokoli bez uspostavljanja konekcije ne daju ovakve garancije, ali je prenos podataka obično brži

Za razliku od protokola mrežnog sloja koji moraju da budu implementirani u svakom čvoru lanca komunikacije, protokoli transportnog sloja moraju biti implementirani jedino na krajnjim čvorovima komunikacije (host čvorovima)

Ruteri (uredaji koji posredno učestvuju u komunikaciji prenošenjem paketa) obično nisu svesni detalja transportnih protokola

Transportni protokoli se, dakle, mogu smatrati protokolima kojim komuniciraju dva host računara

S obzirom da na istom host računaru obično postoji više različitih programa koji imaju potrebu za komunikacijom (svaki korišćenjem zasebnog aplikacionog protokola, ali zajedničkim korišćenjem transportnog protokola), zadatak transportnih protokola je i da vrše tzv. multipleksovanje

- Multipleksovanje se obično ostvaruje kroz koncept portova koji predstavljaju brojeve na osnovu kojih se određuje kom programu pokrenutom na host računaru pripada paket primljen na transportnom sloju

Najkorišćeniji protokoli ovog sloja (koji se koriste u okviru Interneta) su Transfer Control Protocol (TCP) i User Datagram Protocol (UDP)

14. Protokoli transportnog sloja. TCP protokol.

TCP (Transmission Control Protocol) je protokol transportnog sloja u okviru Interneta koji pre komunikacije vrši uspostavljanje pouzdane konekcije između dva hosta

Kanal komunikacije je dvosmeran (eng. full duplex)

Konekcija se uspostavlja tako što klijent i server razmene tri poruke (three way handshake):

-Klijent traži uspostavljanje konekcije, server potvrđuje da prihvata konekciju i konačno klijent potvrđuje da je konekcija uspostavljena

Prava komunikacija može da započne tek nakon što je konekcija uspostavljena, što može da traje neko vreme

15. TCP. Mehanizam komunikacije. Struktura TCP segmenta.

Kanal komunikacije je dvosmeran (eng. full duplex)

Konekcija se uspostavlja tako što klijent i server razmene tri poruke (three way handshake):

-Klijent traži uspostavljanje konekcije, server potvrđuje da prihvata konekciju i konačno klijent potvrđuje da je konekcija uspostavljena

Poruka se deli na pakete koji se nezavisno šalju (komutiranje paketa). Više delova iste poruke može paralelno da putuje kroz mrežu.

Svaki paket se dopunjuje informacijama potrebnim za njegovu dostavu. Na transportnom sloju paketi se nazivaju **segmenti**. Komunikacija se organizuje kao komunikacija između dva programa koji se na njima izvršavaju (a ne samo kao komunikacija između dva uređaja).

Paket mora da sadrži informaciju o:

1. uređaju
2. softveru koji paket prima i koji paket šalje

Na transportnom nivou se paketima dodaju identifikatori softvera - **portovi**, a adrese uređaja tek na mrežnom sloju.

16. TCP. Garancije, kontrole i korekcije.

TCP garantuje pouzdanost prenosa podataka (reliable transfer) čime se garantuje da de paketi koji su poslati biti primljeni (i to u istom redosledu u kojem su poslali). S obzirom da niži mrežni slojevi ne garantuju dostavu paketa:

- TCP protokol mora da se stara o tome da paketi koji zalutaju automatski budu ponovno poslani, kao i da na prihvatnoj strani automatski permutuje primljene pakete tako da odgovaraju redosledu slanja
- Da bi ovo moglo da bude realizovano, uvodi se potvrda prijema paketa (acknowledgment), tj. nakon prijema jednog ili više paketa, vrši se slanje poruke pošaljocu koja govori da su ti paketi zaista primljeni
- Pošaljioc, na osnovu ovoga, može da odluči da ponovno pošalje paket koji je ranije ved bio poslat, u slučaju da u određenom vremenskom periodu ne dobije potvrdu prijema

TCP uvodi kontrolu i korekciju grešaka (error correction)

- Ovo je dodatna slaba provera (vrši se samo kontrola parnosti), jer se pretpostavlja da se jača provera (obično CRC) vrši na nižim slojevima
- Ipak, u praksi se pokazuje da ova provera ima smisla i uspeva da uoči i ispravi veliki broj grešaka koje promaknu ostalim kontrolama
- TCP uvodi i kontrolu brzine protoka (flow control)
- Njom se kontroliše brzina slanja kako se ne bi desilo da brzi uređaji šalju pakete brzinom vedom od one kojom spori uređaji mogu da ih prime (npr. brz računar koji šalje podatke na spor mobilni telefon)

Važna odlika TCP protokola je da vrši kontrolu zagušenja (congestion control)

- Pojava zagušenja se javlja kada više čvorova pokušava da pošalje podatke kroz mrežu koja je ved na granicama svoje propusne modi
- U takvim situacijama, dešava se da brzina komunikacije u celoj mreži opada za nekoliko redova veličina
- Naime, broj izgubljenih paketa se višestruko povedava jer unutrašnji čvorovi mreže (ruteri) ne mogu da prihvate nove pakete zato što su im prihvatni baferi prepuni
- TCP pokušava da detektuje ovakve situacije i da u tim slučajevima uspori sa slanjem paketa dok se mreža ne rastereti
- Jedna od tehnika koje se koriste u cilju smanjenja zagušenja je da se pri početku komunikacije paketi šalju sporije (slow-start), a da se brzina slanja postepeno povedava kada se utvrdi da paketi zaista i stižu na odredište

Činjenica da TCP protokol da vrši kontrolu zagušenja je jedan od razloga zbog čega TCP spada u grupu sporijih protokola

Stoga se TCP ne koristi se za aplikacije kod kojih je brzina prenosa presudna

17. Protokoli transportnog sloja. UDP protokol.

UDP (User Datagram protocol) je protokol transportnog sloja u okviru Interneta koji ne vrši uspostavljanje konekcije pre zapocinjania komunikacije.

Prilikom koriscenja UDP protokola ne vrši se potvrda prijema poslatih paketa, tako da se komunikacija može smatrati nepouzdanom.

Osnovni razlozi koriscenja UDP protokola su, pre svega, njegova brzina i zbog toga se uglavnom koristi od strane aplikacija koje imaju potrebu za komunikacijom u realnom vremenu (eng. real time), kao što su npr. audio-video prenosi, internet telefonija, igrice i slicno.

Takodje, UDP se koristi za aplikacione protokole koji daju elementarne mrezne usluge i vrše kontrolu mreže (npr. DHCP (Dynamic Host Configuration Protocol), DNS (Domain Name System), SNMP (Simple Network Management Protocol)).

18. Protokoli transportnog sloja. Sistem imena domena DNS

IP adrese su pogodne za koriscenje od strane racunara, ali nisu pogodne za ljudsku upotrebu. Kako bi se ljudima olaksalo pamcenje imena racunara, uveden je **sistem imena domena** (eng. domain name system — **DNS**).

DNS se smatra „telefonskim imenikom” Interneta koji imenima domena dodeljuje razne informacije (najcesce IP adrese). Primer: studentski server naseg fakulteta ima domen `alas.matf.bg.ac.rs`.

Domeni su hijerarhijski organizovani i citaju se s desna na levo.

Tako, npr. domen `rs` oznacava Republiku Srbiju, `ac.rs` oznacava akademsku mrežu u Srbiji, `bg.ac.rs` njen cvor u Beogradu, `matf.bg.ac.rs` je Matematicki fakultet, dok `alas.matf.bg.ac.rs` oznacava konkretan studentski server.

Domeni najviseg nivoa mogu biti bilo **nacionalni** (kao u navedenom primeru), bilo **genericki** (npr. `.com`, `.org`, `.net`).

Domeni se koriste u okviru jedinstvenih lokatora resursa na Vebu (URL), u okviru adresa elektronske poste, itd. Prilikom preslikavanja domena u adrese, koriste se usluge distribuirane DNS baze podataka.

Specijalizovani DNS serveri cuvaju delove ove baze. Ovi serveri su hijerarhijski organizovani i ova hijerarhija uglavnom prati hijerarhiju domena.

19. Protokoli transportnog sloja. TCP port.

TCP port: Port predstavlja jednoznacni identifikator (broj izmedju 0 i 65535), logicku vezu izmedju konkretnog softvera i hardvera na kom se taj softver izvrsava. To je broj koji oznacava pristupnu tacku (a ne mesto) za komunikaciju sa serverom i bez njega server ne bi znao sa kojom klijentskom aplikacijom komunicira.

Na primer HTTP server podrazumevano koristi port 80, Telnet 23, FTP 21, POP3 110, SMTP 25, server za vreme 37... TCP portovi izmedju 0 i 1023 su rezervisani za poznate servise i ne preporucuje se da se koriste za nove serverske programe. Ako se izvrsavaju serverski programi na racunarskoj mrezi kompanije, tada sa administratorom sistema treba proveriti koji portovi su slobodni.

20. Protokoli transportnog sloja. Programski interfejs.

Vecina savremenih operativnih sistema i programskih jezika daje direktnu podrsku za koriscenje Internet (TCP/IP familije) protokola. Podrska za koriscenje ovih protokola u okviru programa se realizuje kroz koncept **soketa** (eng. socket).

Socket je apstrakcija kojom se programeru predstavlja kanal komunikacije (zasnovan bilo na TCP bilo na UDP protokolu). Programer pise podatke u soket ili cita podatke iz soketa, obicno na slican nacin kao da je u pitanju obicna datoteka, dok se operativni sistem brine o svim aspektima stvarne mrezne komunikacije.

Primer: U recenici koju korisnik unese se sva mala slova izmenjuju velikim. Klijent prima recenicu od korisnika i salje je serveru vrseci upis u odgovarajuci soket. Nakon prijema odgovora od servera (izmenjene recenice) klijent ga prikazuje korisniku. Server ceka konekcije od klijenata koriscenjem za to specijalizovanog soketa (ServerSocket). Kada klijent uspostavi komunikaciju otvara se obican soket za komunikaciju, prihvata se recenica od klijenta, menjaju se sva mala slova u velika i izmenjena recenica se salje nazad klijentu, nakon cega se nastavlja iscekivanje nove konekcije.

21. Protokoli aplikativnog sloja.

Aplikativni sloj (application layer) definise protokole koje direktno koriste korisnicke aplikacije u okviru svoje komunikacije.

Aplikacioni protokoli u protokoli kojima dva programa tj. dve aplikacije komuniciraju

Najkorišćeniji protokoli ovoga sloja u okviru Interneta su

- HyperText Transfer Protocol (HTTP) koji se koristi za prenos veb stranica
- SMTP, POP3, IMAP koji se koriste u za prenos elektronske pošte
- File Transfer Protocol (FTP) koji se koristi za prenos datoteka, itd.

Veb se sastoji od ogromnog broja klijenata sa pregledačima (kao što su Chrome, Mozilla, Yandex, Safari itd.) i od servera (koji koriste veb servere kao što su Apache, JBoss, Tomcat, Microsoft IIS itd.) koji su međusobno povezani kroz žičane i bežične mreže. Ovi protokoli su prilagođeni specifičnim zahtevima aplikacija

Server: termin server oznacava i sam racunar (hardver) i serverski program koji se na njemu izvrsava. Korisnik preko veb pregledaca salje zahtev za resursom (HTML strana, slika, PDF dokument...), klijent zahteva taj resurs, a server taj zahtev prihvata, pronalazi resurs i salje klijentu. U slucaju da nema zahtevanog resursa generise se "404 Not found" greska.

Klijent: podrazumeva i coveka (korisnika) i aplikaciju – veb pregledac preko kog se salje zahtev (npr. Netscape, Chrome, Mozilla, Yandex, Safari, Edge, Opera). Osim poslova komunikacije sa veb serverom, pregledac treba i da interpretira HTML kod i iscrta veb stranice.

22. HTTP protokol. Karakteristike HTTP protokola.

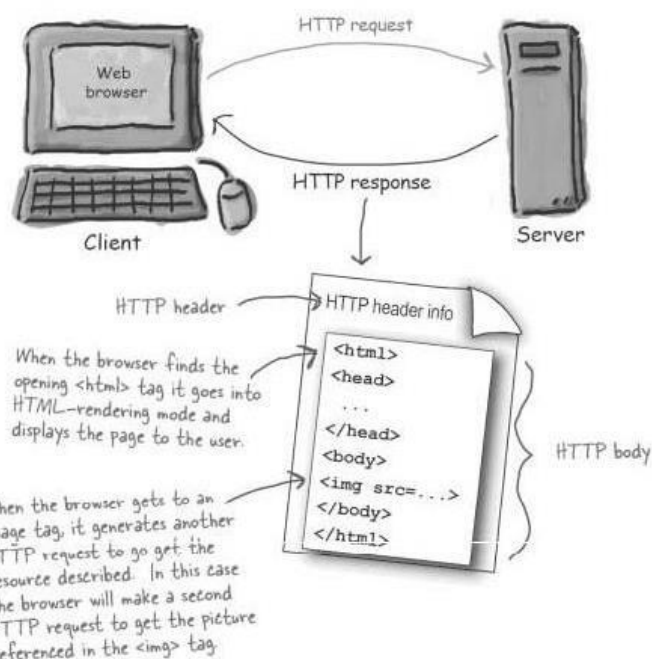
HTTP je protokol aplikativnog sloja koji predstavlja osnovu veba. To je mrezni protokol sa karakteristikama koje se odnose na veb, ali se oslanja na TCP/IP protokol radi obezbedjivanja potpunog prenosa zahteva i slanja odgovora sa jednog mesta na drugo.

Implementiran je u okviru dve vrste programa – klijentskim i serverskim programima. Sustina je komunikacija klijenta i servera: klijent uspostavlja TCP konekciju (obicno na portu 80) i salje HTTP zahtev, a server kroz uspostavljenu konekciju odgovara HTTP odgovorom.

Sam HTTP protokol je opisan IETF (Internet Engineering Task Force) dokumentom RFC 2616.

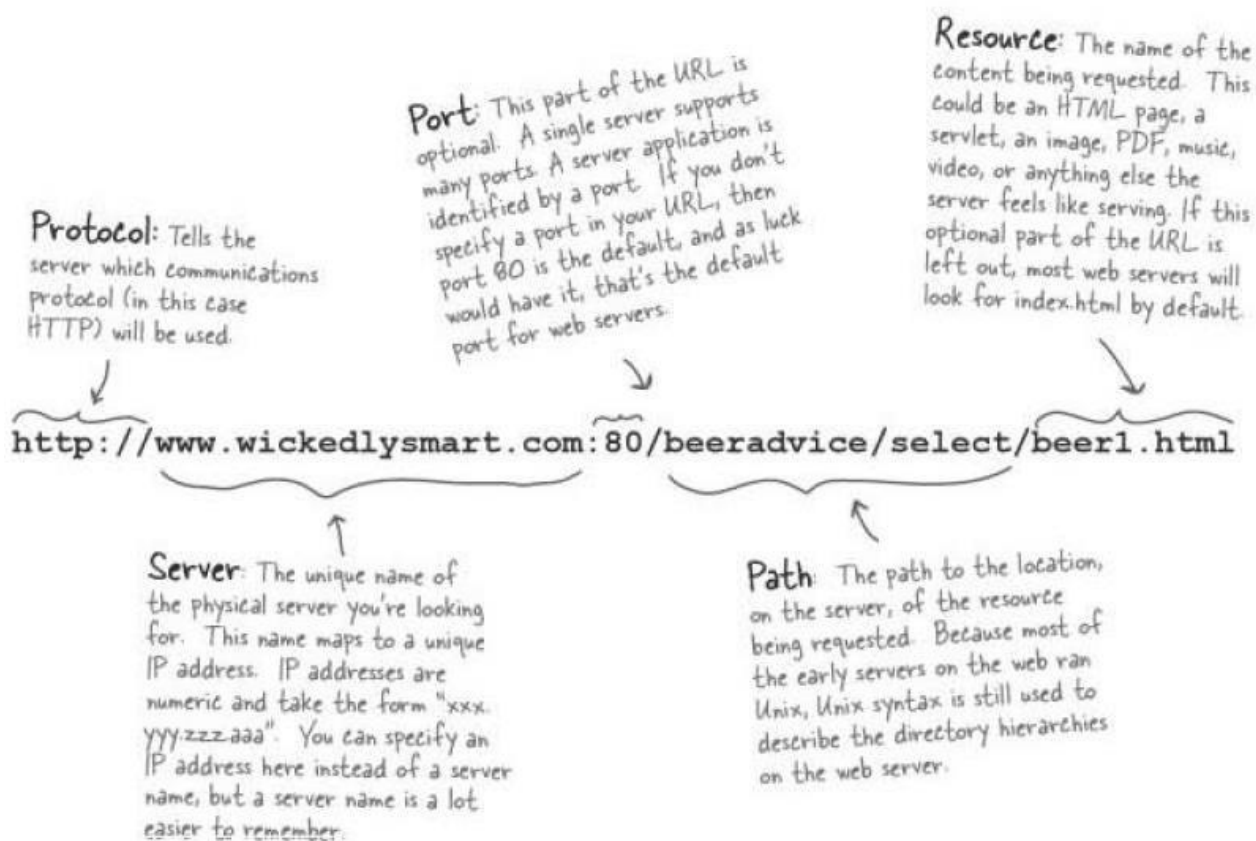
Karakteristike HTTP protokola:

- Ne održava konekciju (connectionless)
- Nezavisan je od medijuma (media independent)
- Ne održava stanja (ne održava/ne koristi nikakve informacije o klijentu) (stateless)



23. HTTP protokol. Mehanizam komunikacije. URL.

URL (Uniform Resource Locator) - format kojim se predstavlja jedinstvena adresa svakog resursa na webu.



Optional Query String: za GET extra info (parametri) je dodat na kraju URLa i pocinje sa ? i odvojen je sa &

24. HTTP protokol. Mehanizam komunikacije. HTTP metodi.

HTTP zahtev u zaglavlju sadrži naziv metoda koji govori serveru o kakvoj se vrsti zahteva radi i kako će biti formatiran ostatak poruke. HTTP protokol podržava sledeće metode:

- GET - koristi se za preuzimanje informacija sa datog servera na osnovu date adrese. Zahtevi koji koriste metod GET treba samo da pribavljaju podatke (HTML strana, slika, PDF...), a nikako ne treba da ih menjaju. To je najjednostavniji HTTP metod.
- HEAD - je vrlo sličan GET metodi, sa tim što se telo poruke ne vraća klijentu (vraća se samo statusna linija i zaglavlje). Metod se može koristiti radi utvrđivanja da li je link izmenjen u odnosu na prethodno stanje - izmenjeno stanje se testira upoređivanjem informacija poslatih u zaglavlju zahteva sa informacijama iz zaglavlja generisanog odgovora
- POST - se koristi za zahtev da se pošalju podaci HTTP serveru korišćenjem HTML forme. Mocniji je od metoda GET jer omogućava istovremeno zahtevanje i slanje podataka serveru.
- PUT - koristi se za zahtev HTTP serveru da se podaci poslati u okviru zahteva smeste na mestu navedenog resursa
- DELETE - koristi se za zahtev serveru da se ukloni navedeni resurs
- TRACE – izvršava testiranje povratne poruke duž putanje kojom se zahtev krede prema ciljnom resursu
- OPTIONS – opisuje opcije komunikacije za ciljni resurs
- CONNECT – obezbeđuje tunelsku komunikaciju prema serveru određenim sa datom adresom

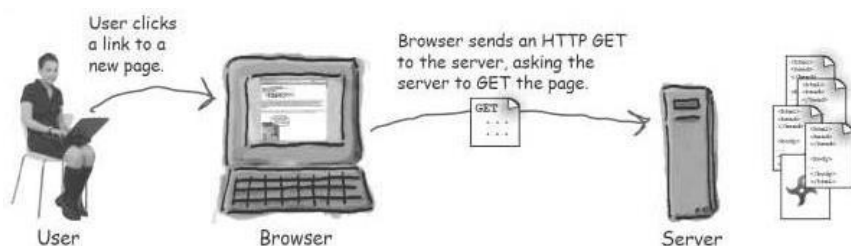
Podaci se mogu slati na server pomoću metoda GET i POST. Razlika je u tome što se u prvom slučaju može poslati manji broj karaktera i podaci su vidljivi korisniku (podložni manipulaciji) jer se direktno nalepljuju na adresu u adresnoj liniji pregledača. Korisnik ne može postaviti marker na stranicu gde je sadržaj forme prosledjen metodom POST, a može ako je GET.

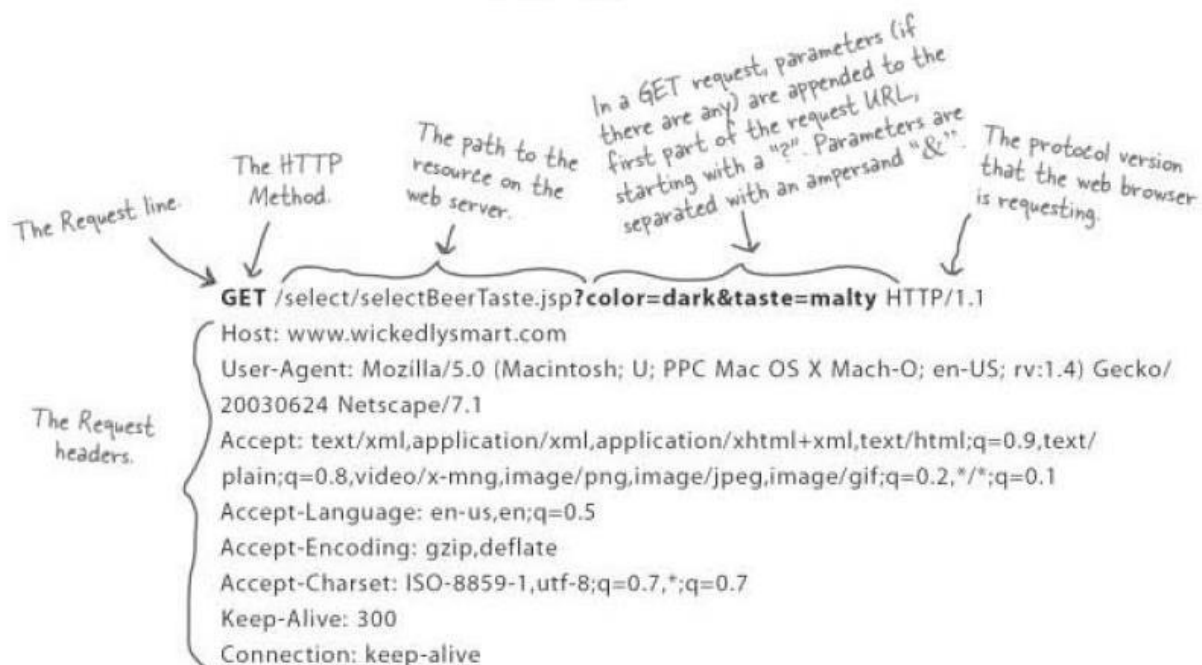
HTTP zahtev sadrži i niz polja i njihovih vrednosti kojima klijent serveru saopštava neke informacije:

- Host: obavezno polje u HTTP/1.1 i sadrži ime hosta na koji se šalje zahtev
- User-Agent: ovim se identifikuje klijentski softver koji šalje zahtev
- Accept: ovim klijent navodi vrstu sadržaja (MIME tip) koju priželjkuje
- Accept-Language: ovim klijent navodi jezik koji priželjkuje
- Accept-Charset: ovim klijent navodi kodnu stranu koju priželjkuje
- Connection: ovim se navodi da li se želi perzistentna (keep-alive) ili jednokratna (close) TCP konekcija.
- If-modified-since: ovim klijent serveru naglašava da mu objekat pošalje samo ako je bio modifikovan od datuma navedenog u ovom polju (ukoliko objekat nije modifikovan, on se ne šalje ponovo već klijent prikazuje verziju koja mu je prethodno bila dostavljena i koja je sacuvana u kesu.

25. HTTP protokol. Metod GET. Struktura zahteva.

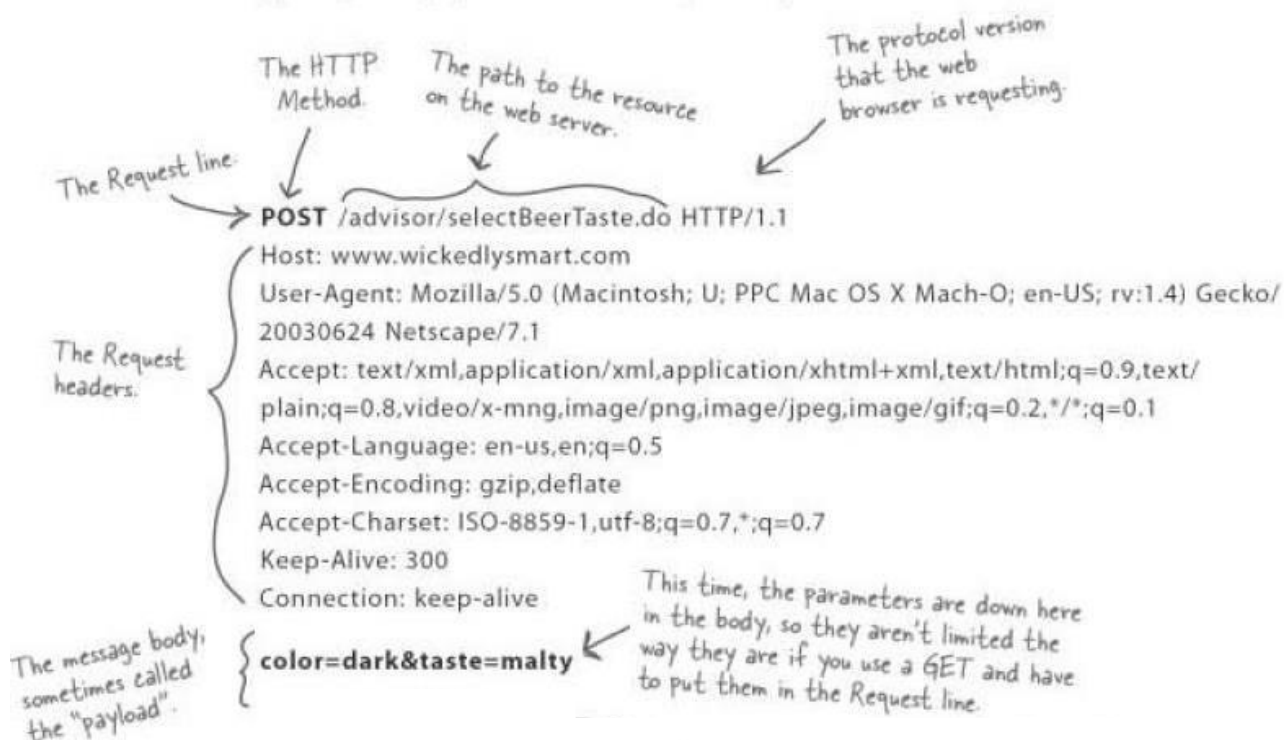
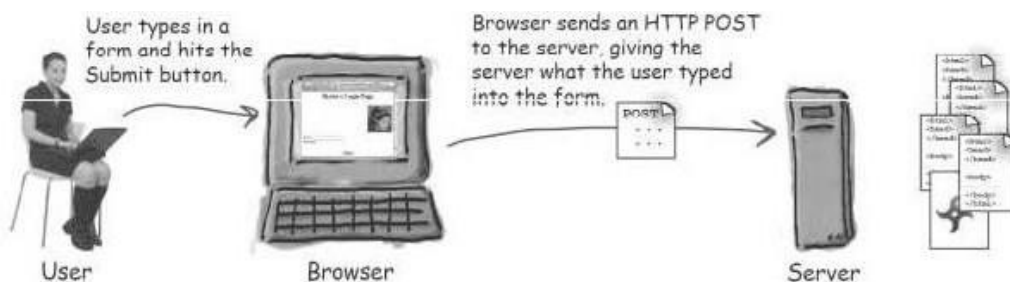
Metod GET je najjednostavniji HTTP metod. On od servera traži da pribavi resurs i da ga vrati pozivaocu. Resurs može biti HTML strana, PDF dokument, JPG slika. Svrha metoda GET je da se dobije resurs od servera.





26. HTTP protokol. Metod POST. Struktura zahteva.

Metod POST je mocniji od metoda GET. Koriscenjem ovog metoda moze se zahtevati nesto od i istovremeno slati podatke na server (pa server moze procesirati prispele podatke).



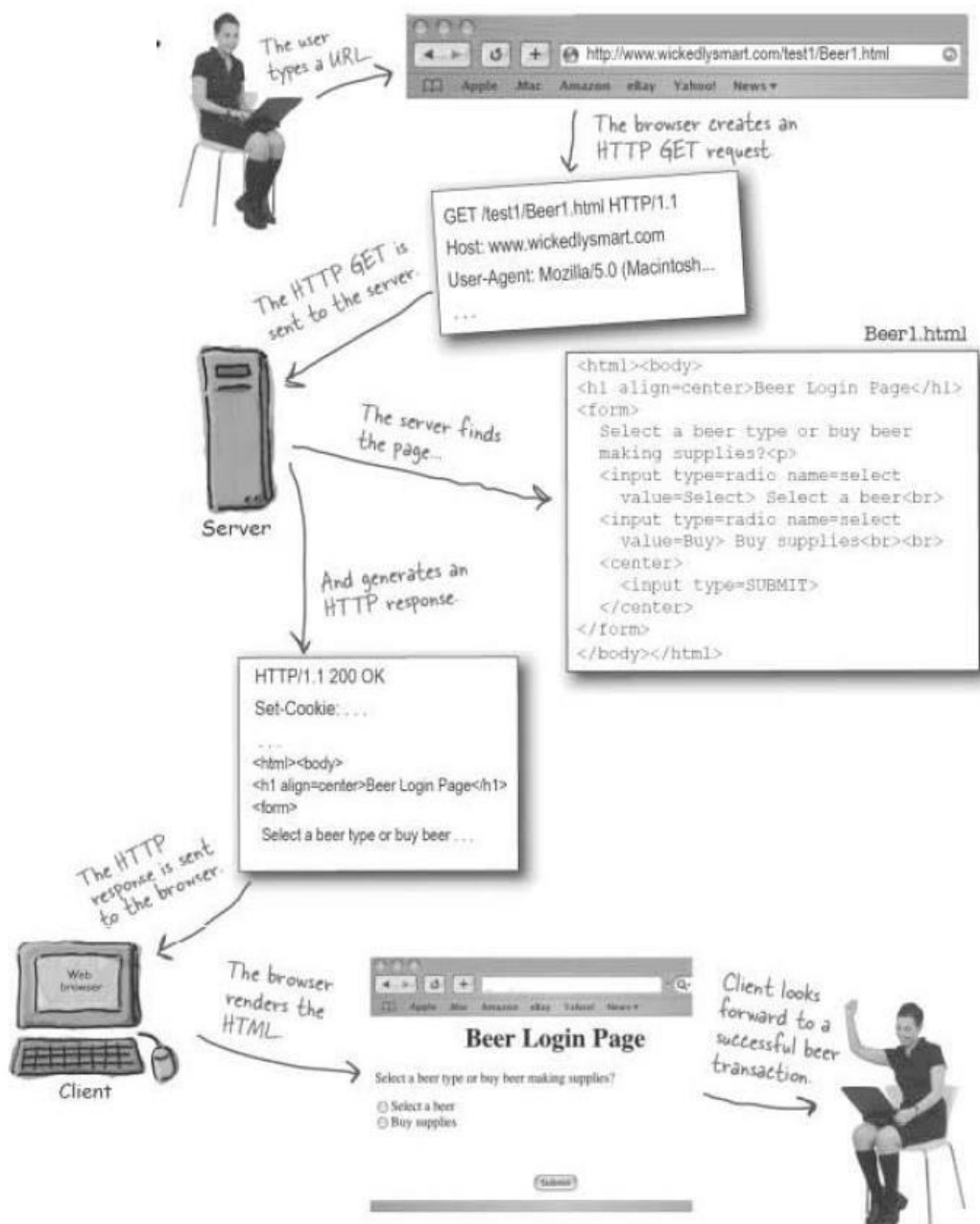
27. HTTP protokol. Struktura odgovora.

Informacije u zaglavlju HTTP odgovora govore pregledaču koji je protokol korišten (koja verzija), da li je zahtev bio uspešan (statusni kod) i koja vrsta sadržaja se nalazi u telu zahteva, a telo sadrži sam sadržaj koji pregledač prikazuje. HTTP odgovor može sadržati i HTML. HTTP dodaje informacije o zaglavlju na početak svakog odgovora, kakav god da je sadržaj. Pregledač koristi informacije iz zaglavlja pri procesiranju HTML sadržaja (HTML se može posmatrati kao sadržaj umetnut u HTTP odgovor).

Neka od najčešće navedenih polja u HTTP odgovorima su:

- Date - tačno vreme kada je odgovor poslat
- Server - identifikacija veb server programa koji je poslao odgovor
- Content-Type - vrsta sadržaja (MIME tip) poslata u okviru odgovora
- Content-Length - dužina sadržaja u bajtovima

Last-Modified - vreme kada je sadržaj poslednji put modifikovan na serveru



28. HTTP protokol. Statusni kodovi odgovora.

U odgovoru se nalazi statusni kod (trocifreni broj). Kodovi koji počinju sa 1 govore da je zahtev primljen i da se proces nastavlja, kodovi koji počinju sa 2 govore o tome da je sve proteklo kako treba, kodovi koji počinju sa 3 obavestavaju korisnika o nekoj redirekciji, kodovi koji počinju sa 4 govore o nekoj greški u zahtevu (koju je napravio klijent), a kodovi koji počinju sa 5 govore o nekoj greški na strani servera. Najcesci su:

- 200 OK - Zahtev je uspešan i informacija se vraća u okviru odgovora
- 301 Moved Permanently - Zahtevani objekat je premešten na lokaciju koja je navedena u polju Location: i klijentski program može automatski da pošalje novi zahtev na dobijenu lokaciju
- 304 Not Modified - Zahtevani objekat nije promenjen od datuma navedenog u zahtevu i nema ga potrebe ponovo slati
- 400 Bad Request - Server nije uspeo da razume zahtev
- 404 Not Found - Zahtevani objekat nije nadjen na serveru
- 500 Internal Server Error - Došlo je do neke interne greške u radu serverskog programa
- 505 HTTP Version Not Supported - Server ne podržava verziju HTTP protokola

29. HTTP protokol. Statičke i dinamičke veb strane.

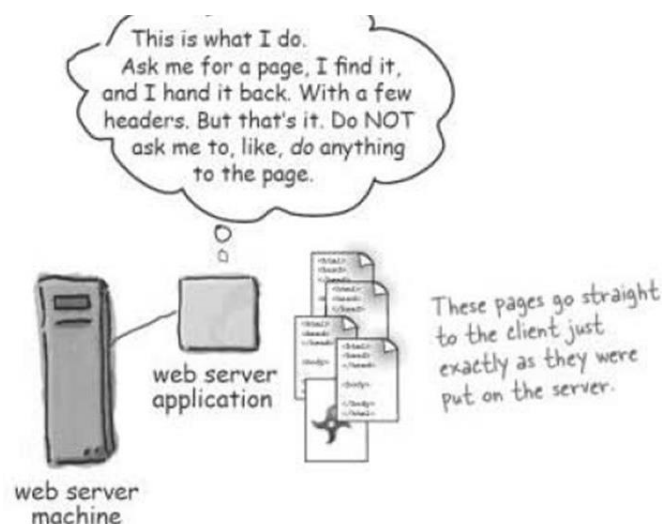
Statička veb strana se nalazi u direktorijumu na veb serveru.

Veb server takvu stranu samo pronade i prosledi klijentu, baš onakvu kakva je na serveru.

Svaki od klijenata dobija potpuno isti sadržaj kao odgovor.

Sam veb server opslužuje samo statičke strane, ali se može koristiti posebna pomodna aplikacija, sa kojom komunicira veb server, a koja kreira **dinamički** sadržaj.

Dinamički sadržaj može biti bilo šta: datum i vreme sa servera, spisak datoteka u direktorijumu, slučajno izabrana slika itd. Dinamički sadržaj ne postoji sve dok ne stigne zahtev.



Po prispedu zahteva, pomodna aplikacija „kreira“ HTML a onda veb server taj HTML „spakuje“ u odgovor.

Kada korisnik prosledi serveru podatke sa forme, tada je za procesiranje prosleđenih podataka (čuvanje podataka u bazi, radi generisanje odgovora na osnovu podataka prosleđenih uz zahtev itd.) neophodno korišćenje pomodne aplikacije.

Kada server prepozna da se zahtev odnosi na pomodnu aplikaciju, tada i prosleđene parametre prosledi pomodnoj aplikaciji, pa ta aplikacija generiše odgovor za koji se potom prosledi klijentu.

30. Protokoli aplikativnog sloja. SMTP, POP3 i IMAP.

Za slanje elektronskih poruka sa računara pošiljaoca na računar primaoca, potrebno je da u komunikaciju budu uključeni i server elektronske pošte pošiljaoca, kao i server elektronske pošte primaoca.

1. Pošiljaoc sa svog računara dostavlja poruku svom serveru, od kog se zahteva da poruku dostavi serveru primaoca i smesti je u poštansko sanduče primaoca (U ovom delu komunikacije koristi se protokol za slanje pošte SMTP)
2. Server pošiljaoca nastavlja da brine o dostavljanju poruke tj. vrši komunikaciju sa serverom primaoca i pokušava da dostavi poruku sve dok ili ne uspe ili dok ne ustanovi da dostavljanje poruke nije moguće (dobije informaciju da postoji greška u adresi ili prođe određeno vreme, a ne uspe da poruku dostavi, i slično)
3. U slučaju da dostavljanje poruke nije uspelo, server obično obaveštava pošiljaoca da dostavljanje nije uspelo
4. Kada se poruka uspešno dostavi na server primaoca, ona se smešta u njegovo poštansko sanduče gde je smeštena sve dok primaoc ne proveri svoju poštu i ne poželi da pročita dobijenu poruku
5. U tom trenutku potrebno je dostaviti poruku sa servera primaoca do njegovog ličnog računara za šta se koriste protokoli za primanje pošte kao što su POP i IMAP.

★ **Simple Mail Transfer Protocol (SMTP)** je standardni protokol za slanje pošte.

★ **Post Office Protocol (POP)** je jednostavni protokol za preuzimanje poruka sa servera, pri čemu se prilikom preuzimanja poruke obično brišu sa servera. Preuzete poruke se čuvaju na klijentskom računaru, koji nakon preuzimanja poruka više ne mora da ima pristup Internetu. Osnovne komande koje klijentski softver izdaje u POP3 protokolu su:

- **APOP** - ovim se vrši autorizacija klijenta navodjenjem njegovog korisničkog imena i kriptovane lozinke.
- **STAT** - statistika o stanju poštanskog sandučeta
- **LIST** – lista poruka
- **RETR** - primanje poruke sa navedenim rednim brojem
- **DELE** - brisanje poruke sa navedenim rednim brojem
- **QUIT** - prekidanje sesije

★ **Internet Message Access Protocol (IMAP)** je znatno napredniji protokol za primanje pošte. On je prvenstveno namenjen korisnicima koji su mobilni tj. koji svojoj pošti pristupaju sa različitih računara. Kako bi ovakvi korisnici imali mogućnost pristupa svim svojim porukama, nije poželjno brisati ih sa servera (iz poštanskog sandučeta) prilikom preuzimanja. Klijenti za elektronsku poštu na lokalnim računarima obično omogućavaju korisnicima sortiranje poruka, organizovanje u fascikle, pretragu i slično. IMAP protokol je projektovan tako da se ovakva funkcionalnost obezbedi tako što se korisnicima omogući da ove funkcije izvedu direktno u svom poštanskom sandučetu na serveru. Mana ovog pristupa je što se zahteva da korisnici imaju pristup Internetu sve vreme dok rade sa svojom elektronskom poštom. Određeni broj veb aplikacija za rad sa elektronskom poštom je zasnovan na IMAP protokolu.

31. Protokoli aplikativnog sloja. FTP.

File Transfer Protocol (FTP) je protokol za prenos datoteka između računara. Protokol datira još od 1970-tih i doba ranog Interneta, ali se i danas koristi. U okviru tipične FTP sesije, korisnik sedi za jednim host računarom i želi da prenosi datoteke na ili sa drugog host računara. FTP koristi TCP kao protokol za komunikaciju nižeg nivoa. FTP protokol ostvaruje dve TCP konekcije za prenos datoteka. Jedna konekcija (obično na portu 21) se koristi za prenos kontrolnih informacija, a druga (obično na portu 20) za prenos samih podataka. Za svaku datoteku, otvara se nova konekcija za prenos podataka, koja se automatski zatvara kada se završi prenos datoteka. Za to vreme kontrolna konekcija sve vreme ostaje otvorena. Za razliku od HTTP protokola, tokom FTP sesije server mora da čuva određene podatke o korisniku (na primer, tekudi direktorijum) tj. FTP je protokol koji čuva stanje (statefull protocol).

Kontrole koje se izdaju serveru putem kontrolne konekcije se zapisuju u čitljivom ASCII obliku:

- **USER username** - koristi se za slanje identifikacije korisnika serveru
- **PASS password** - koristi se za slanje lozinke korisnika serveru
- **LIST** - koristi se kako bi se serveru poslala poruka da pošalje listu datoteka u tekudem direktorijumu
- **RETR filename** - koristi se kako bi se sa servera (iz tekudeg direktorijuma) prenela datoteka sa datim imenom na klijent (u tekudi direktorijum)
- **STOR filename** - koristi se kako bi se sa klijenta (iz tekudeg direktorijuma) prenela datoteka sa datim imenom na server (u tekudi direktorijum)

Server obično na zahteve klijenta otvara konekciju za prenos podataka, a istovremeno preko kontrolne konekcije šalje statusne poruke ili poruke o greškama, kao što su:

- 331 Username OK, password required
- 25 Data connection already open; transfer starting
- 425 Can't open data connection
- 452 Error writing file

4. Jezici za obeležavanje

1. Standardni opšti jezik za obeležavanje. SGML.

U današnjem dobu računara, izdvajaju se dva paradigmatična pristupa za kreiranje tekstualnih dokumenata:

- **WYSIWYG** (What You See Is What You Get) pristup
- **Korišćenje jezika za obeležavanje**

Bide opisana oba pristupa, ali de (zbog mnogobrojnih prednosti koje ovaj pristup donosi) naglasak biti stavljen na eksplicitno obeležavanje teksta korišćenjem jezika za obeležavanje.

- **WYSIWYG pristup:**

Alati zasnovani na WYSIWYG pristupu zahtevaju od korisnika da tekst uređuju u obliku koji je spreman za konačno prikazivanje na ciljnom medijumu (npr. štampanje na papiru). Tekst se uređuje oslanjajući se direktno na njegovu grafičku prezentaciju, najčešće korišćenjem miša i elemenata grafičkog korisničkog okruženja. Tipični primeri ovakvih alata su alati za kancelarijsko poslovanje (npr. *Microsoft Office*, *OpenOffice.org*).

- **Pristup eksplicitnim obeležavanjem teksta**

Tehnika eksplicitnog obeležavanja strukture dokumenata olakšava njihovu automatsku obradu. Obeleženi dokumenti postaju uskladištene informacije koje je moguće automatski obrađivati korišćenjem raznovrsnih računarskih aplikacija, ali i prikazivati u obliku pogodnom za čitanje od strane čoveka. Ovaj pristup dobija na značaju kada se izvrši jasno i eksplicitno razdvajanje obeležavanja logičke strukture i obeležavanja vizuelne prezentacije dokumenta. **Logička struktura** dokumenta podrazumeva njegovu organizaciju na manje jedinice (npr. poglavlja, sekcije, pasuse), kao i označavanje njegovih istaknutih delova (npr. primeri, citati, definicije i teoreme). **Vizuelna prezentacija** određuje izgled dokumenta u trenutku prikazivanja ili štampanja.

Razdvajanje logičke strukture dokumenata od njihove grafičke prezentacije daje mogućnost da se uz minimalan trud istim podacima pridruže sasvim različiti vizuelni prikazi. Prilikom eksplicitnog obeležavanja teksta, koriste se jezici za obeležavanje teksta (*markup languages*). To su veštački jezici u kojima se korišćenjem posebnih oznaka opisuje logička struktura teksta ili njegov grafički izgled. Najpoznatiji jezici za obeležavanje su *HTML*, *TeX*, *LaTeX*, *PostScript*, *RTF*, itd. Svaki od ovih jezika odlikuje se konkretnom sintaksom označavanja i koristi se za označavanje jednog tipa dokumenta (npr. *HTML* se koristi za označavanje hipertekstualnih dokumenata). U praksi se često javlja potreba za označavanjem velikog broja različitih tipova dokumenata (npr. označavanje pisama, tehničkih izveštaja, zbirki pesama, itd.)

Jasno je da svaki pojedinačni tip dokumenata zahteva svoj način označavanja i skup oznaka pogodnih za njegovo označavanje. Ovo dalje omogućava izradu specifičnih softverskih alata pogodnih za određenu vrstu obrade specifičnih tipova dokumenata. Kako bi se na precizan i uniforman način omogućilo definisanje konkretnih jezika za označavanje različitih tipova dokumenata, razvijeni su i **meta jezici**. Najpoznatiji meta jezici za obeležavanje su SGML i XML, u čijem okviru su definisani jezici *HTML*, *XHTML*, *MathML*, *SVG* itd.

- **SGML** (karakteristike i istorijat bide napisani u sledećem pitanju)
- **XML**

2. Karakteristike SGML.

- *Standardni opšti jezik za obeležavanje (Standard Generalized Markup Language)* je meta jezik za obeležavanje standardizovan od strane međunarodne organizacije za standarde (pod oznakom „ISO 8879:1986 SGML”)
- Jezik je razvijen za potrebe kreiranja mašinski čitljivih dokumenata u velikim projektima industrije, državne uprave, vojske itd.
- Osnovna motivacija prilikom standardizovanja ovog jezika je bila da se obezbedi trajnost dokumentima i njihova nezavisnost od aplikacija kojima su kreirani. Informacije skladištene u okviru SGML dokumenta su nezavisne od platforme tj. od softvera i hardvera.
- Pretečom jezika SGML smatra se jezik GML (Generalized Markup Language) koji je nastao u kompaniji IBM 1960-tih
- Jedna od značajnijih primena jezika SGML je bila izrada drugog, elektronskog, izdanja Oksfordskog rečnika engleskog jezika (**OED**)
- Može se reći da je najznačajnija primena jezika SGML došla kroz jezik *HTML*, čije su prve verzije definisane upravo u okviru jezika SGML. Jezik HTML služi za obeležavanje hipertekstualnih dokumenata i postao je standardni jezik za obeležavanje dokumenata na webu.
- Svaki jezik za obeležavanje koji je definisan u SGML-u naziva se i SGML aplikacija, pa se i jezik HTML smatra SGML aplikacijom. SGML se koristi da bi se obeležila struktura dokumenata određenog tipa.

3. Struktura SGML.

Dokumenti se sastoje od međusobno ugnjeđenih elemenata. Za obeležavanje elemenata se koriste etikete(tagovi) oblika **<ime-elementa>** i **</ime-elementa>** (na primer **<strofa>** i **</strofa>** ili **<body>** i **</body>**). Elementi sadrže tekst, druge elemente ili kombinaciju i jednog i drugog. Elementi mogu biti dodatno okarakterisani atributima. Atributi su oblika **ime-atributa=“vrednost atributa”** (na primer naslov=“Žaba čita novine”). U okviru teksta mogu se pojaviti i znakovni entiteti. Oni su oblika **&ime-entiteta;** (na primer **©**) koji označavaju određene znakove.

Sadržaj i značenje elemenata nije propisano meta jezikom, već svaki jezik definisan u okviru SGML-a definiše sopstveni skup etiketa koje koristi za obeležavanje i definiše njihovo značenje i moguće međusobne odnose. Svakom dokumentu, pridružen je njegov tip. Tip dokumenta određuje sintaksu dokumenta tj. određuje koji elementi, atributi i entiteti se mogu javiti u okviru dokumenta i kakav je njihov međusobni odnos. Posebni programi koji se nazivaju **SGML parseri** ili **SGML validatori** mogu da ispituju da li je dokument u skladu sa svojim tipom tj. da li zadovoljava sva sintaksna pravila propisana odgovarajućim tipom.

Pripadnost određenom tipu dokumenta, izražava se deklaracijom **<!DOCTYPE>** koja se navodi na početku samog dokumenta.

- U okviru ove deklaracije se nalaze informacije o imenu tipa dokumenta, organizaciji koja ga je kreirala i sl.
- Obično se u okviru ove deklaracije nalazi uputnica na definiciju tipa dokumenta (Document type definition - DTD).
- Ove datoteke definišu elemente od kojih se grade konkretni dokumenti.
- Tip dokumenta može biti definisan datotekom npr. zbirka-pesama.dtd ili npr. datotekom <http://www.w3.org/TR/html4/strict.dtd>.

- Ukoliko se u nekom od primera pojavi oznaka **PUBLIC** ona ukazuje na to da je tip dokumenta javan i dostupan.
- Korišćenje SGML-a podrazumeva kreiranje sopstvenih ili korišćenje javnih tipova dokumenata i obeležavanje dokumenata u skladu sa njihovim željenim tipom

Proces kreiranja novih tipova dokumenata podrazumeva izradu:

- **SGML deklaracije** - formalnog opisa leksike samih dokumenata koja prvenstveno određuje koji znaci se koriste prilikom kreiranja dokumenata
- **Definicije tipa dokumenta** - formalnog opisa sintakse samih dokumenata koja određuje od kojih elemenata, etiketa, atributa i entiteta se dokument sastoji i kakav je njihov međusobni odnos
- **Semantičke specifikacije** - neformalnog opisa semantike elemenata, etiketa i atributa koji se koriste u okviru dokumenata. Ovakva specifikacija može u sebi da sadrži i neka dodatna ograničenja koja se ne mogu izraziti u okviru formalne definicije tipa dokumenta

4. SGML elementi i atributi. Primeri.

Osnovni konstruktori:

- 1) Elementi i etikete
- 2) Atributi
- 3) Entiteti
- 4) Komentari
- 5) Oznacene sekcije
- 6) Instrukcije procesiranja

1. Elementi i etikete

Osnovna gradivna jedinica SGML dokumenta su elementi. Elementi su obično označeni etiketama (tag). Razlikuju se otvarajuće etikete (<ime-elementa>) i zatvarajuće etikete (</ime-elementa>). Elementi nisu isto što i etikete. Element se sastoji od početne etikete, završne etikete i svog sadržaja između njih (tekst i drugi elementi). Ime elementa se nalazi u početnoj i u završnoj etiketi i dozvoljena je upotreba i malih i velikih slova (==).

Kod nekih SGML elemenata moguće je izostaviti završne etikete (u HTML-u je to element p, <p>), dok je kod nekih čak moguće izostaviti i početne etikete.

Neki SGML elementi nemaju svoj sadržaj (HTML element koji označava prelazak u novi red,
). Kod praznih elemenata je zabranjeno navoditi završnu etiketu.

2. Atributi

Atributi sadrže dodatne informacije o SGML elementima. Imaju svoj naziv i vrednost. Naziv atributa je razdvojen od vrednosti znakom jednakosti. Vrednost atributa treba biti navedena u okviru navodnika (" ") ili apostrofa (' '), ali ponekad mogu biti i izostavljeni. U okviru navodnika je moguće korišćenje apostrofa i obratno.

Atributi elementa se navode u okviru njegove početne etikete (npr. U HTML-u atribut href elementa a određuje određite hiperveze (npr. Link za google). Imena atributa su nezavisna od veličine slova, dok vrednosti nekada zavise, a nekada ne zavise od veličine slova.

5. SGML entiteti. Primeri.

Entiteti

Entiteti u SGML-u se mogu približno poistovetiti sa makro zamenama (mogucnost imenovanja delova sadrzaja na portabilan nacin). Zamena entiteta se vrsi kada se dokumenti analiziraju odgovarajucim parserom npr. moguće je deklarirati entitet pod imenom *uvit*, koji se zamenjuje tekstem *Uvod u Veb i Internet tehnologije*, i zatim se u okviru ovog dokumenta na ime predmeta pozivati koriscenjem reference na entitet.

Postoji nekoliko vrsta entiteta i referenci na entitete:

1. Obicni entiteti (regular entities)
2. Parametarski entiteti (parameter entities)
3. Znakovni entiteti (character entities)

Obicni entiteti

Reference na obicne entitete pocinju znakom & i završavaju se sa ; . Moguce ih je navoditi u okviru teksta dokumenta (ne u okviru DTD). Npr, ako se negde u okviru dokumenta javi sadrzaj: "Nastava iz predmeta "&uvit;" se odvija utorkom", ovim je u stvari kodiran tekst: "Nastava iz predmeta "Uvod u Veb i Internet tehnologije" se odvija utorkom. "

Parametarski entiteti

Reference na parametarske entitete pocinju znakom % i završavaju se sa ; . Moguce ih je navoditi samo u okviru DTD dokumenta (ne u okviru objektnih dokumenata). `<!ENTITY % Charset „CDATA“>`

Znakovni entiteti

Njima se uvode imena koja oznacavaju odredjene znakove. Koriste se da bi se naveli znakovi koji imaju specijalno znacenje, zatim neki retko korisnici znakovi, znakovi koji nisu podržani tekucim kodiranjem ili znakovi koje je nemoguće uneti u okviru softvera za kreiranje dokumenata (npr., u HTML-u "<" oznacava <).

Za predstavljanje znakova u dokumentima moguće je koristiti i numericke znakovne reference (navode se kao dekadni ili heksadekadni brojevi, zapisani izmedju &# i ;).

6. SGML komentari, oznacene sekcije i instrukcije procesiranja. Primeri.

Komentari

U okviru SGML dokumenta moguće je navoditi komentare na sledeci nacin

`<!-- Ovo je jedan komentar -->`

Instrukcije procesiranja (processing instructions)

To su lokalne instrukcije aplikaciji koja obradjuje dokument. One su napisane na nacin specifičan za aplikaciju i navode se izmedju `<? i ?>`.

`<?php echo date(„h:i:s“); ?>`

Oznacene sekcije (marked sections)

Koriste se da bi se oznacili delovi dokumenta koji zahtevaju posebnu vrstu procesiranja. One su oblika:

`<![kljucna rec [...oznacena sekcija....]]>`

Najcesce koriscene kljucne reci:

- CDATA - oznacava doslovan sadrzaj koji se ne parsira
- IGNORE – oznacava da se sekcija ignorise tokom parsiranja
- INCLUDE – oznacava da se sekcija ukljucuje tokom parsiranja
- TEMP – oznacava da je sekcija privremeni deo dokumenta

7. SGML definicije tipa dokumenta DTD. Primeri

Sadržaj i značenje elemenata nije propisano meta jezikom, već svaki jezik definisan u okviru SGML-a definiše sopstveni skup etiketa koje koristi za obeležavanje i definiše njihovo značenje i mogude međusobne odnose.

To je realizovano na sledeći način:

- Svakom dokumentu, pridružen je njegov tip.
- Tip dokumenta određuje sintaksu dokumenta tj. određuje koji elementi, atributi i entiteti se mogu javiti u okviru dokumenta i kakav je njihov međusobni odnos.
- Posebni programi koji se nazivaju SGML parseri ili SGML validatori mogu da ispitaju da li je dokument u skladu sa svojim tipom tj. da li zadovoljava sva sintakсна pravila propisana odgovarajućim tipom.
- Tip dokumenta se kod SGML određuje pomoću tzv. deklaracija tipa dokumenta.

Opis DTD-a

Definicija tipa dokumenta - DTD (eng. Document Type Definition) je skup deklaracija za obeležavanje koje definišu tip dokumenta kod jezika za obeležavanje iz SGML porodice jezika za obeležavanje (SGML, XML, HTML,...).

DTD koristi tercijalnu formalnu sintaksu, kojom deklarise koji elementi i reference mogu da se pojave, i na kom mestu u dokumentu određenog tipa. DTD deklaracijom se određuje i sadržaj i atributi elemenata koji mogu da se pojave u okviru SGML dokumenta. DTD takođe može da deklarise entitete koji mogu biti iskorišćeni u instanci dokumenta.

Svrha DTD-a je da definiše dozvoljene gradivne elemente jednog SGML dokumenta. Sa DTD-om, svaka SGML datoteka može nositi opis svog formata. Sa DTD-om, nezavisne grupe ljudi mogu se saglasiti da koriste standardni DTD za razmenu podataka. U takvom slučaju, razvijena aplikacija može upotrebljavati standardni DTD kako bi proverila da su podaci, primljeni iz spoljašnjeg sveta validni. Takođe, ona može koristiti DTD za verifikaciju sopstvenih podataka.

Na osnovu deklaracije u SGML dokumentu, može se reći da postoje dva tipa DTD: unutrašnji DTD i spoljašnji DTD. Kada je DTD deklarisan unutar SGML datoteke, onda se radi o unutrašnjem DTD-u, a ako je deklarisan u posebnoj tj. odvojenoj datoteci, tada je to spoljašnji DTD.

DTD-om se opisuju sledeće važne karakteristike strukture SGML dokumenta:

- elementi koji se mogu pojaviti u SGML dokumentu.
- redosled kojim se mogu pojaviti.
- fakultativni i obavezni elementi.
- atributi elemenata i da li su oni opcionalni ili obavezni.
- mogu li atributi imati podrazumevane vrednosti.

Korišćenje DTD-a obezbeđuje sledeće prednosti:

- Dokumentacija - može se definisati sopstveni format za SGML datoteke, pa proučavanjem DTD dokumenta, korisnik/programer može da razume strukturu podataka koja se obrađuje.
- Provera valjanosti - obezbeđuje se način za proveru valjanosti SGML datoteka, tako što se proverava da li se elementi pojavljuju u pravom redosledu, da li su obavezni elementi i atributi na mestu, elementi i atributi nisu umetnuti na netačan način itd.

Međutim, odmah se mogu uočiti i neki nedostaci kod DTD-a:

- Podržava samo nisku kao tip podataka.
- Nije objektno orijentisan, pa se koncept nasleđivanja ne može primeniti na DTD-ove.
- Ograničene su mogućnosti izražavanja kardinalnosti elemenata.

Sintaksa DTD-a

DTD može biti naveden unutar SGML dokumenta, ili se, alternativno, može čuvati u odvojenom dokumentu, a zatim se u okviru SGML dokumenta postavi veza prema DTD dokumentu.

Osnovna sintaksa DTD-a je sledeća:

```
<!DOCTYPE element DTD_identifikator
[
  deklaracija1
  deklaracija2
...]>
```

Značenje elemenata prethodno opisane sintakse:

- DTD počinje sa konstrukcijom **<!DOCTYPE**.
- Deo **element** saopštava parseru da parsiranje dokumenta započne baš od tog elementa kao korena.
- Deo **DTD_identifikator** predstavlja identifikator za DTD, što može biti putanja prema datoteci koja se nalazi na sistemu na kom se nalazi taj SGML dokument, ili može biti URL datoteke postavljene na Internetu. U slučaju kada ovaj identifikator referiše na spoljašnju putanju, on se naziva spoljašnji podskup.
- Srednje zagrade, tj. znaci **[i]** ograđuju listu deklaracija elemenata (tzv. unutrašnji podskup). Unutrašnji podskup se može opcionalno pojaviti, zavisno do toga kakva je struktura DTD-a.

Sam DTD se postavlja na početku SGML dokumenta čiju strukturu opisuje.

8. Unutrašnji i spoljašnji DTD. Primeri.

DTD se naziva unutrašnji DTD ako su elementi deklarirani unutar SGML datoteke. U ovom slučaju, deklaracija funkcioniše nezavisno od spoljnih izvora.

Sintaksa unutrašnjeg DTD-a je sledeća:

```
<!DOCTYPE koreni_element [deklaracije_elementa]>
```

gde je koreni_element ime elementa koji predstavlja koren, a iza njega slede deklaracije elemenata.

Deklaracije elemenata opisuju strukturu elemenata koji obarzuju SGML (naziv, način zapisa, redosled pojave, obaveznost/opcionost, atributi i sl.).

Primer. Ilustruje jednostavan unutrašnji DTD:

```
<!DOCTYPE adresa [
  <!ELEMENT adresa (ime,kompanija,telefon)>
```

```

<!ELEMENT ime (#PCDATA)>
<!ELEMENT kompanija (#PCDATA)>
<!ELEMENT telefon (#PCDATA)>
]>
<adresa>
  <ime>Vladimir Filipovic</ime>
  <kompanija>Matematički fakultet</kompanija>
  <telefon>(011) 123-4567</telefon>
</adresa>

```

Unutrašnji DTD sadrži sledeće strukturne elemente:

- Početna deklaracija. Ona sadrži deklaraciju tipa dokumenta, koja se obično naziva DOCTYPE:
Primer. Početna deklaracija kod prethodnog unutrašnjeg DTD-a:

```
<!DOCTYPE adresa [
```

Uočavamo da kljunoj reči DOCTYPE prethodi uskličnik (znak !).

Deklaracija DOCTYPE obaveštava parser da je DTD čiji opis sledi povezan sa SGML dokumentom u kom se deklaracija nalazi. Izjava o tipu dokumenta DOCTYPE mora se pojaviti na početku dokumenta - ona nije dozvoljena nigde drugde u dokumentu. Otvorena srednja zagrada (tj. *) posle reči adresa ukazuje da se radi o DTD-u smeštenom unutar datoteke.

- Telo DTD-a. Deklaraciju DOCTYPE prati telo DTD-a, gde se deklarišu elementi, atributi, entiteti i anotacije:

Primer. Telo DTD-a za unutrašnji DTD iz prethodnog primera:

```

<!ELEMENT adresa (ime,kompanija,telefon)>
<!ELEMENT ime (#PCDATA)>
<!ELEMENT kompanija (#PCDATA)>
<!ELEMENT telefon (#PCDATA)>

```

Ovde je deklarirano nekoliko elemenata koji čine rečnik dokumenta. Tako <!ELEMENT kompanija (#PCDATA)> definiše element kompanija da bude tipa #PCDATA - tekstualni podatak koji se može parsirati.

Dalje, slično deklaraciji DOCTYPE, deklaracije elemenata takođe moraju početi s uskličnikom, tj uskličnik mora da prethodi reči ELEMENT.

- Završna deklaracija. Odeljak deklaracije DTD-a se zatvara pomoću zatvorene srednje zagrade i znaka "vede" (tj pomoću znakova +>. Ovim se efektivno završava definicija tipa dokumenta, nakon koje (kod unutrašnjih DTD) neposredno sledi sadržaj SGML dokumenta.
Što se sadržaja SGML dokumenta posle DTD-a tiče, tip korenskog elementa mora da odgovara imenu datom u DOCTYPE deklaraciji.

Spoljašnji DTD

U spoljašnjim DTD-ovima elementi su deklarirani izvan SGML datoteke. Definicijama elemenata se pristupa specifikiranjem sistemskih atributa, koji mogu biti validna .dtd datoteka ili važeći URL.

Spoljašnji DTD-ovi imaju sledeću sintaksu:

```
<!DOCTYPE koreni_element SYSTEM "ime_datoteke">
```

gde je koreni_element ime elementa koji predstavlja koren, a "ime_datoteke" referiše na datoteku sa nastavkom .dtd.

Primer. Sledede označavanje prikazuje spoljnu upotrebu DTD-a:

```
<!DOCTYPE adresa SYSTEM "adresa.dtd">
<adresa>
  <ime>Vladimir Filipovic</ime>
  <kompanija>Matematički fakultet</kompanija>
  <telefon>(011) 123-4567</telefon>
</adresa>
```

Sadržaj DTD datoteke adresa.dtd je slededi:

```
<!ELEMENT adresa (ime, kompanija, telefon)>
<!ELEMENT ime (#PCDATA)>
<!ELEMENT kompanija (#PCDATA)>
<!ELEMENT telefon (#PCDATA)>
```

Referisanje na spoljašnji DTD može da se realizuje korišćenjem sistemskih identifikatora ili javnih identifikatora.

Sistemski identifikator omogućava da se odredi lokacija spoljne datoteke koja sadrži DTD deklaracije. U tom slučaju, sintaksa je slededa:

```
<!DOCTYPE koreni_element SYSTEM "adresa.dtd" [...]>
```

Kao što se može videti, sistemski identifikator sadrži ključnu reč SYSTEM iza koje sledi URI referenca koja upućuje na lokaciju DTD dokumenta.

Javni identifikatori pružaju mehanizam za pronalaženje DTD resursa. Oni su zapisani na slededi način:

```
<!DOCTYPE koreni_element PUBLIC "-//Beginning XML//DTD Address Example//EN">
```

U ovom slučaju identifikator započinje ključnom rečju PUBLIC, nakon čega sledi specijalna niska, koja označava ključ za elementat javno dostupnog kataloga. Javni identifikatori se koriste za identifikaciju elementa u katalogu. Oni mogu biti u bilo kom formatu, ali uobičajeni format za njih je FPI - formalni javni identifikatori (engl. Formal Public Identifiers).

9. DTD elementi. Primeri.

U ovoj sekciji se razmatraju SGML komponente iz perspektive DTD. DTD u osnovi sadrži deklaracije slededih SGML komponenti: elementi, atributi i entiteti.

SGML elementi se mogu definisati kao gradivni blokovi SGML dokumenta. Elementi se mogu posmatrati kao kontejner koji sadrži tekst, druge elemente, attribute, medijske objekte ili kombinaciju svega prethodno navedenog. Svaki SGML dokument sadrži jedan ili više elemenata čije su granice ili ograničene početnim i/ili završnim oznakama (etiketama, tagovima), odnosno praznim elementima.

Primer. Sledede označavanje prikazuje jednostavan primer SGML:

```
<ime> Marko Markovic </ime>
```

DTD element se deklarira deklaracijom ELEMENT. Kada SGML datoteka bude validirana preko DTD-a, raščlanjivač u početku proverava koreni element, a zatim redom sve njegove potomke.

Deklaracije DTD elemenata imaju ovaj opšti oblik: <!ELEMENT ime_elementa (sadržaj)>

Uočavaju se slededi delovi:

- Deklaracija ELEMENT koristi se za označavanje parseru da sledi definicija elementa.
- ime_elementa je naziv elementa (takođe se zove i generički identifikator) koji se definiše.
- sadržaj definiše koji sadržaj (ako postoji) može da uđe u sastav elementa.

Sadržaj deklaracije elemenata u DTD-u može se kategorisati na sledeći način:

- Prazan sadržaj
- Sadržaj elementa
- Mešoviti sadržaj
- Bilo koji sadržaj

10. DTD elementi. Prazan sadržaj. Primeri.

Ovo je poseban slučaj deklaracije elemenata, kada element ne sadrži nikakav sadržaj. Prazni sadržaji su deklarirani ključnom reči EMPTY.

Deklaracija praznog elementa ima sledeću sintaksu:

```
<!ELEMENT ime_elementa EMPTY>
```

U gornjoj sintaksi ELEMENT je deklaracija elementa kategorije EMPTY, a ime_elementa je ime praznog elementa.

Primer. Sledeći jednostavan primer pokazuje deklaraciju praznog elementa:

```
<!DOCTYPE hr[  
  <!ELEMENT hr EMPTY>  
>  
<hr />
```

U gornjem primeru adresa (element hr) je deklarirana kao prazan element. Oznaka za element adrese pojaviti će se kao niska <hr />.

11. DTD elementi. Sadržaj elemenat/elementi. Primeri.

U deklaraciji elementa sa sadržajem elementa, sadržaj bi bili dozvoljeni elementi u zagradama. Tu se, po potrebi, može uključiti više elemenata.

Sledi sintaksa deklaracije elementa sa sadržajem elementa:

```
<!ELEMENT ime_elementa (potomak1, potomak2...)>
```

U ovom slučaju, ELEMENT je oznaka deklaracije elementa, ime_elementa je ime elementa, a potomak1, potomak2 ... su elementi i svaki od tih elemenata mora imati svoju definiciju unutar DTD-a.

Primer. Jednostavan primer za deklaraciju elementa sa sadržajem elementa:

```
<!DOCTYPE adresa [  
  <!ELEMENT adresa (ime,kompanija,telefon)>  
  <!ELEMENT ime (#PCDATA)>  
  <!ELEMENT kompanija (#PCDATA)>  
  <!ELEMENT telefon (#PCDATA)>  
>  
<adresa>  
  <ime>Vladimir Filipovic</ime>  
  <kompanija>Matematički fakultet</kompanija>  
  <telefon>(011) 123-4567</telefon>      </adresa>
```

U ovom primeru, адреса је елемент-родитељ, а име, компанија и телефон су његови потомци.

U tabeli koja sledi prikazana je lista operatora i pravila sintakse koja se mogu primeniti u definisanju elemenata - potomaka:

Оператор	Синтакса	Опис	Пример
+	<!ELEMENT ime_elementa (potomak1+)>	Указује да се елементау - пупмак може појавити један или више пута унутар родитељског елемента	<!ELEMENT адреса (име+)> Елементау име може да се појави једном или више пута унутар елемента адреса.
*	<!ELEMENT ime_elementa (potomak1*)>	Указује да се елементау - пупмак може појавити нула, један или више пута унутар родитељског елемента	<!ELEMENT адреса (име*)> Елементау име може да се појави нула, једном или више пута унутар родитељског елементу адреса
?	<!ELEMENT ime_elementa (potomak1?)>	Указује да се елементау - пупмак може појавити нула пута или једном унутар родитељског елемента	<!ELEMENT адреса (име?)> Елементау име може поципнално да се појави унутар родитељског елементу адреса
,	<!ELEMENT ime_elementa (potomak1, potomak2)>	Даје секвенцу елементау-пупмака који морају бити укључени у родитељски елементау, по редоследу набрајања	<!ELEMENT адреса (име, компанија)> Секвенца елементау име, компанија се морају појавити у искуп редоследу унутар елемента адреса
	<!ELEMENT ime_elementa (potomak1 potomak2)>	Омогућава дондешоо избора по елементу-пупмку	<!ELEMENT адреса (име другоиме)> jedno od ta dva mora da se pojavi unutar adrese

12. DTD elementi. Mešoviti sadržaj. Primeri.

Ovo je kombinacija (#PCDATA) i elemenata-potomaka. PCDATA označava tekstualne podatke koji se mogu parsirati, odnosno tekst koji nije obeležen. U modelima mešovitog sadržaja tekst se može pojaviti izolovan ili se može umešati između elemenata. Pravila za modele mešovitog sadržaja slična su pravilima o sadržaju elemenata, što je objašnjeno u prethodnom odeljku.

Sledi generička sintaksa za sadržaj mešovitih elemenata:

```
<!ELEMENT ime_elementa (#PCDATA|potomak1|potomak2)*>
```

Ovde je ELEMENT je oznaka deklaracije elementa, ime_elementa je naziv elementa, PCDATA je tekst koji ne sadrži obeležavanja, a potomak1, potomak1 ... su elementi (svaki od ovih elemenata mora imati svoju definiciju unutar DTD-a).

U deklaraciji mešovitog sadržaja:

- #PCDATA mora biti na prvom mestu.
- Operator * mora slediti deklaraciju mešovitog sadržaja ako su uključeni elementi - potomci.
- Deklaracije #PCDATA i elementi - potomci moraju biti odvojene operatorom |.

Primer. Jednostavan primer koji demonstrira deklaraciju mešovitog sadržaja u DTD-u:

```
<!DOCTYPE adresa [  
  <!ELEMENT adresa (#PCDATA|ime)*>  
  <!ELEMENT ime (#PCDATA)>  

```

```
<adresa>
```

Ovo je tekst koji de biti pomešan sa elementom.potomkom.

```
<ime>
```

Paja Patak

```
</ime>
```

```
</adresa>
```

13. DTD elementi. Bilo koji sadržaj. Primeri.

Može da se deklarise element koristeći ključnu reč ANY u sadržaju. Najčešće se naziva elementom mešovite kategorije. Ovaj scenario je koristan kada tek treba da se odluči o dozvoljenom sadržaju elementa.

Sintaksa deklarisanja elemenata sa bilo kojim sadržajem je slededa:

```
<!ELEMENT ime_elementa ANY>
```

Ovde ključna reč ANY označava da se tekst PCDATA i / ili bilo koji elementi deklarirani u DTD-u mogu koristiti u sadržaju elementa ime_elementa. Oni se mogu koristiti u bilo kom redosledu bilo koji broj puta. Međutim, ključna reč ANY ne dozvoljava da se uključe elementi koji nisu deklarirani u DTD-u.

Primer. Jednostavan primer koji demonstrira deklaraciju elementa sa bilo kojim sadržajem:

```
<!DOCTYPE adresa [  
  <!ELEMENT adresa ANY>  

```

```
<adresa>
```

ovde ime nešto malo nekakvog teksta

```
</adresa>
```

14. DTD atributi. Primeri.

Atributi su deo SGML elemenata. Element može imati bilo koji broj jedinstvenih atributa. Atributi daju više informacija o SGML elementu, odnosno definišu određeno svojstvo elementa. SGML atribut je uvek par ime-vrednost.

Primer. Sledede označavanje prikazuje jednostavan primer SGML atributa:

```
<img src = "cvet.jpg"/>
```

Deklaracije atributa

U ovom poglavlju demo razmotriti DTD attribute. Atribut daje više informacija o elementu ili tačnije definiše svojstvo elementa. SGML atribut je uvek u obliku para ime-vrednost. Element može imati bilo koji broj jedinstvenih atributa.

Deklaracija atributa je po mnogo čemu slična deklaracijama elemenata, osim što se, umesto deklarisanja dozvoljenog sadržaja za elemente, deklarirše lista dozvoljenih atributa za svaki element. Ove liste se nazivaju ATTLIST deklaracija.

Osnovna sintaksa deklaracije DTD atributa je slededa:

```
<!ATTLIST ime_elementa ime_atributa tip_atributa vrednost-atributa>
```

U prethodnoj sintaksi DTD atributi počinju sa ključnom reči <!ATTLIST ako element sadrži atribut, ime_elementa specificira ime elementa na koji se atribut odnosi, ime_atributa određuje ime atributa koji je uključen u ime_elementa, a tip_atributa definiše vrstu atributa, a vrednost-atributa uzima fiksnu vrednost koju atributi moraju definisati.

Primer. Jednostavan primer koji demonstrira deklaraciju atributa u DTD:

```
<!DOCTYPE adresa [  
  <!ELEMENT adresa ( ime )>  
  <!ELEMENT ime ( #PCDATA )>  
  <!ATTLIST ime id CDATA #REQUIRED>  
<adresa>  
  <ime id = "123">Marko Kraljevid</ime>  
</adresa>
```

Svi atributi koji se koriste u SGML dokumentu moraju biti deklarirani u definiciji tipa dokumenta (DTD) pomodu izjave o listi atributa.

Atributi se mogu pojaviti samo u početnim etiketama ili praznim etiketama.

Ključna reč ATTLIST mora biti napisana svim velikim slovima.

Unutar liste atributa za dati element nisu dozvoljena ponovljena imena atributa.

Vrste atributa

Prilikom deklarisanja atributa treba odrediti kako procesor treba da obrađuje podatke koji se pojavljuju u vrednosti. Vrste atributa možemo kategorisati u tri glavne kategorije:

- Niska tipovi
- Tokenizovani tipovi
- Enumerisani tipovi

Sledi rezime različitih tipova atributa:

- CDATA - predstavlja tekstualne podatke (bez obeležavanja). Spada u kategoriju niska tipova.
- ID - jedinstveni identifikator atributa. Ne bi trebalo da se pojavi više od jednom. ID je tokenizovani tip atributa.
- IDREF - koristi se za referisanje na identifikator drugog elementa. Na taj način se uspostavlja veza između elemenata. IDREF je tokenizovani tip atributa.
- IDREFS - koristi se za referisanje na više identifikatora. IDREFS je tokenizovani tip atributa.
- ENTITY - predstavlja spoljni entitet u dokumentu. ENTITY je tokenizovani tip atributa.
- ENTITIES - predstavlja listu spoljnih entiteta u dokumentu. ENTITIES je tokenizovani tip atributa.
- NMTOKEN - sličan je CDATA tipu atributa, pri čemu se vrednost atributa sastoji se od važećeg SGML imena. NMTOKEN je tokenizovani tip atributa.
- NMTOKENS - sličan je CDATA tipu atributa, pri čemu atributa sadrži listu važećih SGML imena. NMTOKENS je tokenizovani tip atributa.
- NOTATION - element de biti upućen na notaciju deklarisanu u DTD dokumentu. NOTATION je enumerisani tip atributa.
- Enumeration - omogućava definisanje određene liste vrednosti gde se vrednost atributa mora poklopiti sa jednom od nabrojanih vrednosti. To je je enumerisani tip atributa.

Vrednosti atributa

Unutar svake deklaracije atributa mora se navesti kako de se vrednost pojaviti u dokumentu. Dakle, treba odrediti da li atribut:

- može imati podrazumevanu vrednost
- može imati fiksnu vrednost
- je zahtevan
- je implicitan

Podrazumevane vrednosti

Sadrži podrazumevanu vrednost. Vrednosti se mogu zatvoriti u pojedinačne navodnike (') ili dvostruke navodnike (").

Sledi sintaksa deklaracije atributa :

```
<!ATTLIST ime_elementa ime_atributa tip_atributa "podrazumevana-vrednost">
```

Ovde je podrazumevana-vrednost definisana vrednost atributa.

Primer. Sledi jednostavan primer deklaracije atributa sa podrazumevanom vrednošću:

```
<!DOCTYPE adresa [  
  <!ELEMENT adresa ( ime )>  
  <!ELEMENT ime ( #PCDATA )>  
  <!ATTLIST ime id CDATA "0">  
<adresa>  
  <ime id = "123">  
    Tanmay Patil  
  </ime>  
</adresa>
```


Fiksne vrednosti

Ključna reč #FIXED pradena fiksnom vrednošdu koristi se kada treba da se odredi da je vrednost atributa konstantna i da se ne može promeniti. Uobičajena upotreba fiksnih atributa je određivanje brojeva verzija.

Sledi sintaksa fiksnih vrednosti:

```
<!ATTLIST ime_elementa ime_atributa tip_atributa #FIXED "vrednost" >
```

gde je fiksirana vrednost definisane vrednosti atributa.

Primer. Sledi jednostavan primer deklaracije atributa sa fiksnom vrednošdu:

```
<!DOCTYPE adresa [  
  <!ELEMENT adresa (kompanija)*>  
  <!ELEMENT kompanija (#PCDATA)>  
  <!ATTLIST kompanija ime NMTOKEN #FIXED "matf">  
>  
<adresa>  
  <kompanija ime = "matf">mi smo mnogo jaki!</kompanija>  
</adresa>
```

Ako se pokuša sa promenom vrednosti atributa, dobide se greška, odnosno dokument više nede biti validan.

Sledi sadržaj dokumenta zasnovanog na istom DTD-u koji nije validan:

```
<!DOCTYPE adresa [  
  <!ELEMENT adresa (kompanija)*>  
  <!ELEMENT kompanija (#PCDATA)>  
  <!ATTLIST kompanija ime NMTOKEN #FIXED "matf">  
>  
<adresa>  
  <kompanija ime = "abc">ovo smo mi!</kompanija>  
</adresa>
```

Zahtevane vrednosti

Kad god treba da se navede da je atribut neophodan, koristi se ključna reč #REQUIRED.

Sledi sintaksa deklaracije:

```
<!ATTLIST ime_elementa ime_atributa tip_atributa #REQUIRED>
```

Primer. Jednostavan primer deklaracije DTD atributa sa ključnom reči #REQUIRED:

```
<!DOCTYPE adresa [  
  <!ELEMENT adresa ( ime )>  
  <!ELEMENT ime ( #PCDATA )>  
  <!ATTLIST ime id CDATA #REQUIRED>  
>  
<adresa>  
  <ime id = "123">  
    Tanmay Patil  
  </ime>  
</adresa>
```

Implicitne vrednosti

Prilikom deklarisanja atributa uvek se mora navesti deklaraciju vrednosti. Ako atribut koji se deklarise nema podrazumevanu vrednost, nema fiksnu vrednost i nije potreban, tada atribut morate proglasiti implicitnim. Ključna reč #IMPLIED koristi se za određivanje da je atribut implicitan.

Sledi sintaksa takve deklaracije:

```
<!ATTLIST ime_elementa ime_atributa tip_atributa #IMPLIED>
```

Primer. Sledi jednostavan primer deklaracije atributa sa implicitnom vrednošću:

```
<!DOCTYPE adresa [  
  <!ELEMENT adresa ( ime )>  
  <!ELEMENT ime ( #PCDATA )>  
  <!ATTLIST ime id CDATA #IMPLIED>  
<adresa>  
  <ime />  
</adresa>
```

U ovom primeru je korišćena ključnu reč #IMPLIED jer nismo želeli da navedemo bilo kakve attribute koji de biti uključeni u ime elementa.

15. DTD entiteti. Primeri.

Entiteti su rezervisana mesta u SGML-u. Oni se mogu prijaviti u prologu dokumenta ili u DTD-u. Entiteti se prvenstveno mogu kategorizovati kao:

- Ugrađeni entiteti
- Znakovni entiteti
- Opšti entiteti
- Parametarski entiteti

Pet ugrađenih entiteta se često javlja u dobro oblikovanom SGML-u. To su:

- znak za "trgovačko i" (&), zapisuje se kao &
- apostrof ili jednostuki navodnik ('), zapisuje se kao '
- znak veće od (>), zapisuje se kao >
- znak manje od (<), zapisuje se kao <
- znak za dvostruki navodnik: ("), zapisuje se kao "

Deklaracije entiteta

Generalno, entiteti se mogu deklarirati interno ili eksterno, pa se u skladu sa tim nazivaju unutrašnji ili spoljašnji. Ako je entitet deklarisan u DTD-u, on se naziva **unutrašnjim entitetom**.

Sledi sintaksa za internu deklaraciju entiteta:

```
<!ENTITY ime_entiteta "vrednost_entiteta">
```

U ovoj sintaksi ime_entiteta je ime entiteta pradenom njegovom vrednošću u dvostrukim navodnicima ili pojedinačnim navodnicima, dok vrednost_entiteta sadrži vrednost za ime entiteta.

Na vrednost entiteta kod unutrašnjeg entiteta uklanja se referenca dodavanjem prefiksa & imenu entiteta.

Primer. Jednostavan primer za internu deklaraciju entiteta:

```
<!DOCTYPE adresa [  
  <!ELEMENT adresa (#PCDATA)>  
  <!ENTITY ime "Vuk Mandušid">  
  <!ENTITY kompanija "Epika">  
  <!ENTITY phone_no "(011) 123-4567">  
>  
<adresa>  
  &ime;  
  &kompanija;  
  &phone_no;  
</adresa>
```

U gornjem primeru, odgovarajuda imena entiteta ime, kompanija i phone_no zamenjeni su njihovim vrednostima u SGML dokumentu. Vrednosti entiteta se dobijaju dodavanjem prefiksa & na ime entiteta.

Ako je entitet deklarisan izvan DTD-a, on se naziva **spoljašnjim entitetom**. Može se referisati na spoljni entitet korišćenjem sistemskih identifikatora ili javnih identifikatora.

Sledi sintaksa za eksternu deklaraciju entiteta:

```
<!ENTITY ime_entiteta SYSTEM "URI/URL">
```

U gornjem opisu ime_entiteta je naziv entiteta, SYSTEM je ključna reč, "URI/URL" je adresa spoljnog izvora zatvorena unutar dvostrukih ili pojedinačnih navodnika.

U slučaju parametarskih entiteta, koristi se oznaka %. Vec deklarisan entitet moze ucestvovati u deklaraciji drugih entiteta.

16. DTD. Identifikatori i strani ključevi. Primeri.

Referisanje na spoljašnji DTD može da se realizuje korišćenjem sistemskih identifikatora ili javnih identifikatora.

Sistemski identifikator omogućava da se odredi lokacija spoljne datoteke koja sadrži DTD deklaracije. U tom slučaju, sintaksa je slededa:

```
<!DOCTYPE koreni_element SYSTEM "adresa.dtd" [...]>
```

Kao što se može videti, sistemski identifikator sadrži ključnu reč SYSTEM iza koje sledi URI referenca koja upućuje na lokaciju DTD dokumenta.

Javni identifikatori pružaju mehanizam za pronalaženje DTD resursa. Oni su zapisani na slededi način:

```
<!DOCTYPE koreni_element PUBLIC "-//Beginning XML//DTD Address Example//EN">
```

U ovom slučaju identifikator započinje ključnom rečju PUBLIC, nakon čega sledi specijalna niska, koja označava **ključ** za elemenat javno dostupnog kataloga. Javni identifikatori se koriste za identifikaciju elementa u katalogu. Oni mogu biti u bilo kom formatu, ali uobičajeni format za njih je FPI - formalni javni identifikatori (engl. Formal Public Identifiers).

17.XML. Značaj i karakteristike XML.

Hijerarhijski format čitljiv za čoveka, jezik "potomak" HTML-a, koji se uvek može parsirati.
"Lingua franca" za podatke: služi za čuvanje dokumenata strukturisanih podataka.
Smešani su podaci i struktura.

Jezgro je šireg ekosistema:

- Podaci – XML
- Shema – DTD i XML Shema
- Programerski pristup – DOM i SAX
- Upiti – XPath, XSLT, Xquery
- Distribuisano programiranje – veb servisi

eXtensible Markup Language (skr. XML) je meta jezik za obeležavanje koji je nastao sredinom 1990-tih kao rezultat potrebe za postojanjem jezika za obeležavanje veoma sličnog jeziku SGML, a koji bi bio jednostavniji za parsiranje (raslanjavanje) i obradu. Ovaj jezik je iz SGML-a izbacio proizvoljnosti tako da se pisanjem dokumenata u ovom jeziku moraju postovati mnogo striktnija pravila. Svaki ispravan XML dokument je ujedno i ispravan SGML dokument.

Koristi se jer olakšava teznju da se "sadržaj" odvoji od "prezentacije". Prezentacija obezbeđuje lepotu pri posmatranju, a sadržaj se može interpretirati od strane racunara, a za racunare prezentacija predstavlja hendikep.

XML je "polu-struktuiran", tj. polu-uniformisan, polu-organizovan.

Informacija u XML – u je hronoloski:

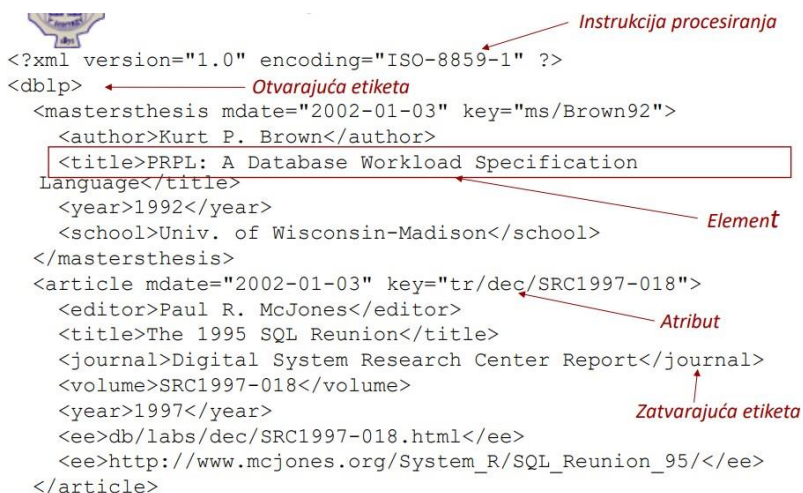
1. Razdvojena od prezentacije
2. Isecena u male delove
3. Oznacena sa semantickim znacenjem

Informacija se u ovom formatu lako može procesirati racunarima.

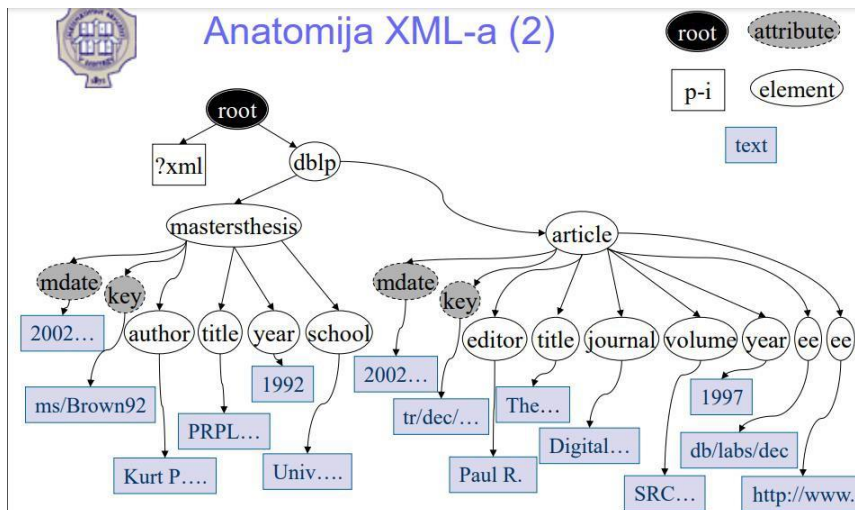
XML opisuje samo sintaksu, a ne apstraktni logicki model podataka.

Ključni pojmovi XML-a su: (svi ovi pojmovi su nasleđeni iz SGML-a)
dokumenti, elementi, atributi, deklaracije prostora imena, tekst, komentari i instrukcije procesiranja

18.XML. Anatomija XML-a. Primeri.



```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<dblp>
  <mastersthesis mdate="2002-01-03" key="ms/Brown92">
    <author>Kurt P. Brown</author>
    <title>PRPL: A Database Workload Specification
Language</title>
    <year>1992</year>
    <school>Univ. of Wisconsin-Madison</school>
  </mastersthesis>
  <article mdate="2002-01-03" key="tr/dec/SRC1997-018">
    <editor>Paul R. McJones</editor>
    <title>The 1995 SQL Reunion</title>
    <journal>Digital System Research Center Report</journal>
    <volume>SRC1997-018</volume>
    <year>1997</year>
    <ee>db/labs/dec/SRC1997-018.html</ee>
    <ee>http://www.mcjones.org/System_R/SQL_Reunion_95/</ee>
  </article>
```



- XML lako čuva relacije
Primer: Relacija student-course-grade

```
<student-course-grade>
  <tuple><sid>1</sid><cid>570103</cid>
    <exp-grade>B</exp-grade>
  </tuple>
  <tuple><sid>23</sid><cid>550103</cid>
    <exp-grade>A</exp-grade>
  </tuple>
</student-course-grade>
```

sid	cid	exp-grade
1	570103	B
23	550103	A

```
ili
<student-course-grade>
  <tuple sid="1" cid="570103" exp-grade="B"/>
  <tuple sid="23" cid="550103" exp-grade="A"/>
</student-course-grade>
```

19.XML. Elementi kod XML-a. Primeri.

```
<book year="1967">
  <title>Politics of experience</title>
  <author>
    <firstname>Ronald</firstname>
    <lastname>Laing</lastname>
  </author>
</book>
```

Elemente karakteriše:

- Ugnježdjena struktura
- Drvoidna struktura
- Redosled je važan
- Sadrži samo znakove, a ne cele brojeve, itd.

Obuhvađeni su etiketama

☐ Otvarajuća etiketa: npr. <bibliography>

☐ Zatvarajuća etiketa: npr. </bibliography>

☐ Elementi bez sadržaja (prazni): npr. <bibliography /> je skraćenica za <bibliography> </bibliography>

Elementi mogu biti ugnježdjeni <bib> <book> Wilde Wutz </book> </bib>

Elementi koji su ugnježdjeni mogu biti višestruki <bib> <book> ... </book> <book> ... </book> </bib>

U tim slučajevima, redosled je veoma važan!

Dokumenti moraju biti dobro formirani:

`<a> ` nije dopušteno!

`<a> ` nije dopušteno!

20.XML. Atributi kod XML-a. Primeri.

Atributi su pridruženi elementima

Primer:

```
<book price = "55" year = "1967" >
  <title> ... </title>
  <author> ... </author>
</book>
```

Elementi mogu sadržavati samo attribute (unutar otvarajuće etikete)

Primer: `<person name = "Wutz" age = 33"/>`

Imena atributa moraju biti jedinstvena!

Primer: Nelegalna je sledeća konstrukcija `<person name = "Wilde" name = "Wutz"/>`

Tekst se može javiti unutar sadržaja elementa

Primer: `<title>The politics of experience</title>`

Tekst može biti izmešan sa ostalim elementima ugnježenim u dati element

Primer: `<title>The politics of experience</title>`

Karakteristike izmešanog sadržaja:

☐ Veoma je koristan za podatke u obliku dokumenata, tj. rečenica

☐ Nema potreba za mešanim sadržajem u scenarijima „procesiranja podataka“, jer se tada obično obrađuju entiteti i relacije

☐ Ljudi komuniciraju rečenicama, a ne entitetima i relacijama. XML omogućuje da se sačuva struktura prirodnog jezika, uz dodavanje semantičkih oznaka koje mogu računarski interpretirane

Prostori imena

Omogućavaju integraciju podataka iz različitih izvora

Omogućavaju integraciju različitih XML rečnika (tj. prostora imena)

Svaki „rečnik“ ima jedinstven ključ, identifikovan URI-jem

Isto lokalno ime iz različitih rečnika može imati

☐ Različita značenja

☐ Različite pridružene strukture

Kvalifikovana imena (Qualified Names - QName) služe za prigrubljivanje imena „rečniku“

Kvalifikovana imena se odnose na sve čvorove XML dokumenta koji imaju imena (attribute, elemente, Instrukcije za procesiranje)

Način korišćenja

☐ Povezivanje (tj. prefiks i URI) se uvode u otvarajućoj etiketi elementa

☐ Kasnije se za opis koristi prefiks, a ne URI

☐ Postoje podrazumevani prostori imena, pa je prefiks opcionalan

☐ Prefiks se od lokalnog imena razdvaja dvotačkom, tj. znakom :

21. XML. Strukturisanje podataka kod XML-a. Primeri.

1. Prirodni jezik: Dana said that the book entitled "The politics of experience" is really excellent !
2. Polu-strukturisani podaci (tekst):
`<citation author="Dana"> The book entitled "The politics of experience" is really excellent ! </citation>`
3. Polu-strukturisani podaci (mešani sadržaj): `<citation author="Dana"> The book entitled <title> The politics of experience</title> is really excellent ! </citation>`
4. Strukturisani podaci: `<citation>
<author>Dana</author>
<aboutTitle>The politics of experience</aboutTitle>
<rating> excellent</rating>
</citation>`

Za razliku od drugih formata podataka, XML je veoma fleksibilan i elementi se mogu umetati na različite načine

Može se početi sa pisanjem XML-a koji predstavlja podatke i bez prethodnog dizajniranja strukture

☐ Tako se radi kod relacionih baza podataka ili kod Java klasa

Međutim, strukturisanje ima veliki značaj:

- ☐ Podspešuje pisanje aplikacija koje procesiraju podatke
- ☐ Ograničava podatke na one koje su korektni za datu aplikaciju
- ☐ Definiše „a priori“ protokol o podacima koji se razmenjuju između učesnika u komunikaciji

Struktura XML-a modelira podatke i sadrži:

- ☐ Definicije struktura
- ☐ Definicije tipova
- ☐ Podrazumevane vrednosti

Karakteristike struktuiranja:

Nije formalizovano na način kako je to urađeno kod relacionih baza podataka

☐ Ono je obično zasnovano na strukturi koja se već nalazi u podacima, npr relacionoj SUBP ili raširenoj elektronskoj tabeli

Pri struktuiranju se XML drvo orjentiše prema „centralnim“ objektima

Velika dilema: element ili atribut

- ☐ Element se koristi za osobinu koja sadrži svoje osobine ili kada se može očekivati da će biti više takvih unutar elementa koji ih sadrži
- ☐ Atribut se koristi kada se radi o jednoj osobini – mada je OK da se i tada koristi element!

22. XML. Dobro formirani i validni XML. Primeri.

1. Dobro formirani dokumenti

Kod njih se verifikuju samo osnovna XML ograničenja, npr. `<a>`

2. Validni dokumenti

Kod njih se verifikuju dodatna ograničenja opisana u nekom od jezika za opis strukture (npr. DTD, XML shema)

XML dokumenti koji nisu dobro formirani ne mogu biti procesirani

XML dokumenti koji nisu validni ipak mogu biti procesirani (mogu se upitima izvlačiti podaci, mogu biti transformisani, itd.)

23. XML. DTD kod XML. Primeri.

DTD (Document Type Definition) je deo originalne XML 1.0 specifikacije.

DTD opisuje "gramatiku" za XML datoteku:

- Deklaracije elemenata: pravila i ograničenja koja opisuju dopuštene načine ugnježdavanja elemenata
- Attributes lists: opisuje koji su atributi dopušteni nad kojim elementima
- Dodatna ograničenja na vrednosti elemenata i atributa
- Koji je element koreni čvor XML strukture

Provera strukturnih ograničenja pomodu DTD se naziva DTD validacija (određuje se da li je XML dokument validan ili invalidan).

Zbog svojih ograničenja, DTD se sada relativno retko koristi za validaciju XML dokumenata.

Razlike u odnosu na SGML:

- Velicina slova nije bitna u okviru SGML DTD, dok u XML DTD postaje bitna.
- Nije dozvoljena minimalizacija elemenata tako da DTD ne može da sadrži pravila minimalizacije.
- Mnogi skraćeni zapisi u okviru zapisa DTD nisu više podržani. Na primer, nije moguće koristiti komentare u okviru deklaracija, nije moguće koristiti istovremenu deklaraciju više elemenata i slično.
- Mnogi tipovi atributa su ukinuti.
- Dodatno uključivanje i isključivanje sadržaja (+(S) i -(S)) nije dozvoljeno.
- Zabranjeno je korišćenje veznika &

Referisanje na DTD u okviru XML-a:

1. Nema DTD-a (Radi se o dobro formatiranom XML dokumentu)
2. DTD je unutar dokumenta:
<!DOCTYPE name [definition]>
3. Spoljašnji DTD, specificiran URI-jem (URI - Uniform Resource Identifier):
<!DOCTYPE name SYSTEM "demo.dtd">
4. Spoljašnji DTD, dato je ime i (opcionally) URI:
<!DOCTYPE name PUBLIC "Demo">
<!DOCTYPE name PUBLIC "Demo" "demo.dtd">
5. DTD je unutrašnji + spoljašnji:
<!DOCTYPE name1 SYSTEM "demo.dtd">

Primer DTD-a koji opisuje strukturu dblp sloga:

```
<!ELEMENT dblp((mastersthesis | article)*)>
<!ELEMENT mastersthesis(author,title,year,school,committeemember*)>
<!ATTLIST mastersthesis(mdate CDATA #REQUIRED key ID #REQUIRED advisor CDATA #IMPLIED)>
<!ELEMENT author(#PCDATA)>
```

...

Primer referisanja na DTD u okviru XML datoteke:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE dblp SYSTEM "my.dtd">
<dblp>...
```


24.XML. Odnos između DTD i XML sheme.

Ograničenja DTD-ova

DTD opisuje samo „gramatiku“XML datoteke, a ne detaljnu strukturu niti tipove

Tako, na primer, preko DTD se ne može iskazati da:

- ☐ elementat “length” mora sadržavati nenegativan ceo broj (*ograničenje koje se odnosi a tip vrednosti elementa ili atributa*)
- ☐ elementat “unit” treba da bude dopušten samo onda kada je prisutan elementat “amount” (*ograničenje koje se odnosi na zajedničko pojavljivanje*)
- ☐ elementat “comment” može da se pojavi na bilo kom mestu (*fleksibilnost sheme*)
- ☐ DTD-ov ID nije preterano dobra implementacija za vrednost ključa
- ☐ Ne postoji podrška za nasleđivanje kao kod objektno-orjentisanih jezika
- ☐ Sintaksa koja je bliska XML-u, ali nije XML nije pogodna da se na toj osnovi razvijaju alati

Osnove XML sheme

Kreirana tako da prevaziđe probleme sa DTD-ovima, ima XML sintaksu (detaljnije u nastavku)

25.XML. XML sheme.

Principi dizajna za sheme:

Jezik XML shema treba da bude:

1. Izražajniiji od XML DTD-ova
2. Izražen pomodu XML-a
3. Samo-opisiv
4. Pogodan za korišćenje za širok opseg aplikacija koje koriste XML
5. Direktno pogodan za korišćenje na Internetu
6. Optimizovan za interoperabilnost (sposobnost za zajednicki rad razlicitih sistema)
7. Dovoljno jednostavan da se može implementirati na skromnim resursima
8. Usaglašen sa relevantnim W3C specifikacijama (W3C – WWW Consortium, medjunarodna organizacija zaduzena za izradu standarda interneta)

Osnove XML sheme:

- Kreirana tako da moze da prevazidje probleme za DTD-ovima
 - Ima XML sintaksu
 - Moze definisati kljuceve koriscenjem XPath konstrukcije
 - Tip korenog elementa za dokument je globalno naniže kompatibilan sa DTD
 - Prostori imena su deo XML shema
 - Podržano je nasleđivanje tipova, koje uključuje i ograničavanje opsega
 - Nasleđivanje proširivanjem (by extension) kojim se dodaju novi podaci
 - Nasleđivanje ograničavanjem (by restriction) kojim se dodaju nova ograničenja
 - Podržani su domen i predefinisani tipovi podataka
- Prosti tipovi predstavljaju način restrikcije domena na skalarne vrednosti
 - Tako se može definisati prosti tip zasnovan na celobrojnom tipu, pri čemu su vrednosti tog prostog tipa unutar zdatog opsega

- Složeni tipovi su način definisanja struktura element/atribut
 - U osnovi ekvivalentni sa !ELEMENT kod DTD-a
 - Specificira bilo sekvencu, bilo izbor među elementima – potomcima
 - Specificira minimalni i maksimalni broj pojavljivanja (minOccurs i maxOccurs), pri čemu je podrazumevana vrednost 1
- Elementima se može pridružiti složeni tip ili prosti tip
- Atributima se može pridružiti samo prosti tip
- Tipovi mogu biti predefinisani ili korisnički (složeni tipovi mogu biti samo korisnički, ne mogu biti predefinisani)

Napomene:

- Shema se obično čuva u odvojenom XML dokumentu
- Rečnik za shemu se definiše u specijalnom prostoru imena, a kao prefiks se obično koristi xsd
- Postoji shema koja opisuje XML sheme
- Element schema je uvek koren za XML shemu

26.XML sheme. Prolog XML sheme. Primeri.

Primer prologa sheme:

```
<?xml version="1.0" ?>
<xsd:schema xmlns:xsd="http://w3.org/2001/XMLSchema">
...
</xsd:schema>
```

Napomene:

- ☐ Shema se obično čuva u odvojenom XML dokumentu
- ☐ Rečnik za shemu se definiše u specijalnom prostoru imena, a kao prefiks se obično koristi xsd
- ☐ Postoji shema koja opisuje XML sheme
- ☐ Element schema je uvek koren za XML shemu

27.XML sheme. Tipovi i elementi. Primeri.

- Prosti tipovi predstavljaju način restrikcije domena na skalarne vrednosti
 - Tako se može definisati prosti tip zasnovan na celobrojnem tipu, pri čemu su vrednosti tog prostog tipa unutar zadatog opsega
- Složeni tipovi su način definisanja struktura element/atribut
 - U osnovi ekvivalentni sa !ELEMENT kod DTD-a
 - Specificira bilo sekvencu, bilo izbor među elementima – potomcima
 - Specificira minimalni i maksimalni broj pojavljivanja (minOccurs i maxOccurs), pri čemu je podrazumevana vrednost 1
- Elementima se može pridružiti složeni tip ili prosti tip
- Atributima se može pridružiti samo prosti tip
- Tipovi mogu biti predefinisani ili korisnički (složeni tipovi mogu biti samo korisnički, ne mogu biti predefinisani)

28.XML sheme. Globalni i lokalni elementi i tipovi. Primeri.

Globalne deklaracije

Instance (tj. primerci) globalne deklaracije elementa predstavljaju potencijalne korene elemente za date XML dokumente.

Globalne deklaracije se mogu referencirati na sledeći način:

```
<xsd:schema xmlns:xsd="...">
  <xsd:element name="book" type="BookType"/>
  <xsd:element name="comment" type="xsd:string"/>
  <xsd:ComplexType name="BookType">
    ... <xsd:element ref="comment" minOccurs="0"/> ...
```

Ogranicenja – ni ref, ni minOccurs, ni maxOccurs se ne mogu koristiti u globalnoj deklaraciji.

Deklaracija globalnog elementa

Primer definisanja elementa u XML shemi:

```
<xsd:element name="book" type="BookType"/>
```

Napomene:

- Etiketa *element* služi za deklarisanje elemenata
- Atribut *name* služi za imenovanje elemenata
- Atribut *type* definiše tip elementa (npr. U ovom gore primeru tip elementa book je BookType)
- Deklaracije direktno unutar elemenata *schema* su **globalne** (samo oni elementi čije su deklaracije globalne mogu doći u obzir da budu koreni za XML shemu, npr. u onom istom primeru jedini globalni element je book, pa koren za validni XML dokument opisan ovom shemom mora biti book)

Deklaracija globalnog tipa

Primer definisanja složenog tipa u XML shemi:

```
<xsd:complexType name="BookType">
  <xsd:sequence> ... </sequence>
</xsd:complexType>
```

Napomene:

- Ovaj složen tip je definisan kao sekvenca elemenata
- Atribut *name* određuje ime tipa
- Ova definicija tipa je **globalna** (jer se nalazi direktno unutar elementa shema), pa se ovako definisan tip može koristiti u bilo kojoj drugoj definiciji

Lokalni element

Primer definisanje lokalnog elementa u XML shemi:

```
<xsd:sequence>
  <xsd:element name="title" type="xsd:string"/>
</xsd:sequence>
```

Napomene:

- Ovo je lokalni element jer se ne nalazi direktno u elementu schema, već unutar kompleksnog tipa (znači da element title ne može biti koreni element dokumenta)
- Atribut *name* služi za imenovanje elementa

- Atribut *type* definiše tip elementa
- Tip *xsd:string* je predefinisani tip za XML shemu

Deklaracija lokalnog elementa

Primer definisanja lokalnog elementa *author*:

```
<xsd:element name="author" type="PersonType" minOccurs="1" maxOccurs="unbounded"/>
```

Napomene:

- Radi se o deklaraciji lokalnog elementa (ako se posmatra cela XML shema)
- Tip *PersonType* je korisnicki definisan tip
- Atributi *minOccurs* i *maxOccurs* odredjuju kardinalnost elementa *author* u tipu *BookType*
- Ovakva vrsta definisanja atributa se u žargonu naziva „brušenje“ („facet“)
- Postoji 15 različitih vrsta brišenja, npr. *minExclusive*, *totalDigits*, itd.
- Podrazumevane vrednosti za ove attribute su *minOccurs=1*, *maxOccurs=1*

Definicija lokalnog tipa

Primer definisanja lokalnog tipa *PersonType*:

```
<xsd:complexType name="PersonType">
  <xsd:sequence>
    <xsd:element name="first" type="xsd:string"/>
    <xsd:element name="last" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
```

Napomene:

- Tip *PersonType* je lokalni jer je definisan u okviru definicije tipa *BookType* i može se koristiti samo unutar opsega definicije tipa *BookType*
- Sintaksa ovog tipa je slicna sintaksi tipa *BookType*

Definicija lokalnog elementa

Primer definisanja lokalnog elementa *publisher*:

```
<xsd:element name="publisher" type="xsd:anyType"/>
```

Napomene:

- U okviru definicije tipa *BookType* => lokalan
- Svaka knjiga ima tacno jedan element *publisher*
- Brusenja *minOccurs* i *maxOccurs* imaju podrazumevanu vrednost 1
- Tip *anyType* je prefedinisan tip koji dozvoljava bilo kakav sadrzaj
- Tip *anyType* je podrazumevan (ako se nista ne napise, radi se o tom tipu), tj. Definicija koja je ekvivalentna definiciji iz primera: `<xsd:element name="publisher"/>`

29.XML sheme. Prosti tipovi (predefinisani i izvedeni). Primeri.

Predefinisani prosti tipovi

- *Numeticki tipovi*: Integer, Short, Double, Float, Decimal, HexBinary,...

- *Tipovi za datume i periode:* Duration, DateTime, Time, Date,...
- *Tipovi za niske:* String, NMTOKEN, NMTOKENS, NormalizedString
- *Ostali tipovi:* QName, AnyURI, ID, IDREFS, Language, Entity,...
- *Kao zaključak, postoje 44 predefinisana prosta tipa*

Izvedeni prosti tipovi

Primer dela sheme sa tipom gde je izvršeno ograničavanje domena:

```
<xsd:simpleType name="MyInteger">
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="10000"/>
    <xsd:maxInclusive value="99999"/>
  </xsd:restriction>
</xsd:simpleType>
```

Primer definicije tipa gde je izvršeno ograničavanje domena preko regularnih izraza, tako da valute može biti zapisana samo pomodu tri velika slova:

```
<xsd:simpleType name="Currency">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="*A-Z+,3-"/>
  </xsd:restriction>
</xsd:simpleType>
```

Primer definicije tipa gde je izvršeno ograničavanje domena preko enumeracije:

```
<xsd:simpleType name="Currency">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="ATS"/>
    <xsd:enumeration value="EUR"/>
    <xsd:enumeration value="GBP"/>
    <xsd:enumeration value="USD"/>
  </xsd:restriction>
</xsd:simpleType>
```

Napomene:

- Najvedi broj predefinisanih tipova je izveden restrikcijom iz drugih predefinisanih tipova, npr tip Integer je izveden iz tipa Decimal
- Od 44 predefinisana tipa, samo njih 19 su osnovni tipovi

30. XML sheme. Složeni tipovi, elemenat sequence. Primeri.

Složeni tipovi se konstruišu od prostih i drugih složenih tipova korišćenjem konstruktora:

– Sequence – def. uređenu grupu elemenata koji moraju da se javljaju u definisanom redosledom.

Podrazumevano, svaki element je obavezan i jednoznačan ali se to može promeniti indikatorima minOccurs i maxOccurs.

```

<?xml version="1.0" encoding="UTF-8"?>
- <xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">
  - <xsd:element name="Knjige">
    - <xsd:complexType>
      - <xsd:sequence maxOccurs="unbounded">
        <xsd:element name="Knjiga" type="TipKnjiga"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  - <xsd:complexType name="TipKnjiga">
    - <xsd:sequence>
      <xsd:element name="Naslov" type="xsd:string"/>
      <xsd:element name="Autor" maxOccurs="unbounded" type="xsd:string"/>
      <xsd:element name="Godina" type="xsd:string"/>
      <xsd:element name="ISBN" type="xsd:string"/>
      <xsd:element name="Izdavac" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>

```

31.XML sheme. Složeni tipovi, elemenat all. Primeri.

Složeni tipovi se konstruišu od prostih i drugih složenih tipova korišćenjem konstruktora:

– All – def. grupu u kojoj se svi elementi mogu pojaviti u proizvoljnom redosledu ali tačno jedanput.

32.XML sheme. Složeni tipovi, elemenat choice. Primeri.

Unutar <xsd:choice> i </xsd:choice> navode se elementi – neki od njih imace datu shemu.

Primer sheme za knjigu koja ima ili element **author** ili element **editor**:

```

<xsd:complexType name="Book"><xsd:sequence>
  <xsd:choice>
    <xsd:element name="author" type="Person" maxOccurs="unbounded"/>
    <xsd:element name="editor" type="Person"/>
  </xsd:choice>
</xsd:sequence></xsd:complexType>

```

33.XML sheme. Deklaracija atributa. Primeri.

Atributi mogu biti samo prostog tipa (npr. string)

Deklaracije atributa mogu biti globalne

☐ Takve deklaracije se mogu ponovo iskoristiti pomodu ref

Deklaracije atributa su kompatibilne sa listom atributa u DTD

☐ Mogude je korišćenje podrazumevanih vrednosti

☐ Mogude je attribute proglasiti zahtevanim ili opcionalnim

☐ Postojanje fiksnih atributa

☐ Kao dodatna osobina, postoje i „zabranjeni“ atributi

Primer dela sheme koji opisuje strukturu knjige i indeksa:

```

<xsd:complexType name="BookType">

```

```

<xsd:sequence> ... </xsd:sequence>
<xsd:attribute name="isbn" type="xsd:string" use="required"/>
<xsd:attribute name="price" type="xsd:decimal" use="optional" />
<xsd:attribute name="curr" type="xsd:string"
    fixed="EUR" />
<xsd:attribute name="index" type="xsd:idrefs"
    default="" />
</xsd:complexType>

```

Anonimni tipovi

Primer dela sheme koji opisuje strukturu knjige, gde se tip koji opisuje osobu ne imenuje:

```

<xsd:complexType name="BookType">
    ...
    <xsd:element name="author">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="first" type="xsd:string"/>
                <xsd:element name="last" type="xsd:string"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    ...

```

34. XML sheme. Predefinisani tipovi, proširenje i restrikcije. Primeri.

Predefinisani prosti tipovi

- *Numeticki tipovi*: Integer, Short, Double, Float, Decimal, HexBinary,...
- *Tipovi za datume i periode*: Duration, DateTime, Time, Date,...
- *Tipovi za niske*: String, NMTOKEN, NMTOKENS, NormalizedString
- *Ostali tipovi*: QName, AnyURI, ID, IDREFS, Language, Entity,...
- *Kao zakljucak, postoje 44 predefinisana prosta tipa*

Prosti tipovi predstavljaju način restrikcije domena na skalarne vrednosti

☐ Tako se može definisati prosti tip zanosvan na celobrojnomo tipu, pri čemu su vrednosti tog prostog tipa unutar zdatog opsega

Primer sheme za korisnički definisan tip liste sa restrikcijom:

```

<xsd:simpleType name = "Participants" >
    <xsd:list itemType = "xsd:string" />
</xsd:simpleType>
<xsd:simpleType name = "Medalists" >
    <xsd:restriction base = "Participants" >
        <xsd:length value = "3" />
    </xsd:restriction>
</xsd:simpleType>

```

35.XML scheme. Prosti tipovi liste i unije. Primeri.

Prosti tip listi

Postoji vise vrsta prostih tipova za liste:

- Predefinisani tipovi listi: IDREFS, NMTOKENS
- Korisnicki definisani tipovi listi

Primer sheme za korisnicki definisan tip liste

```
<xsd:simpleType name="intList">  
  <xsd:list itemType="xsd:integer"/>  
</xsd:simpleType>
```

Karakteristike:

- Elementi u primerku takve liste su razdvojeni belinama "5 -10 7 -20"
- Brušenja za restrikcije kod ovih listi su: length, minLength, maxLength, enumeration

Primer sheme za korisnički definisan tip liste sa restrikcijom:

```
<xsd:simpleType name = "Participants" >  
  <xsd:list itemType = "xsd:string" />  
</xsd:simpleType>  
  
<xsd:simpleType name = "Medalists" >  
  <xsd:restriction base = "Participants" >  
    <xsd:length value = "3" />  
  </xsd:restriction>  
</xsd:simpleType>
```

Prosti tip unije

- Odgovara znaku | kod DTD
- Ima isto značenje kao slogovi sa promenljivim delom u Pascal-u ili kao unije u C-u
- Instance su validne ako su validne za jedan od pobrojanih tipova; Primer sheme sa prostim tipom unije:

```
<xsd:simpleType name="Potpurri">  
  <xsd:union memberTypes="xsd:string intList"/>  
</xsd:simpleType>
```
- Za prosti tip unije su podržana brušenja pattern i enumeration

36.XML scheme. Složeni tipovi, elemenat group i attributeGroup. Primeri.

Grupe elemenata

Opis sheme kada se treba postidi da ako element **book** sadrži element **editor**, tada **book** takođe sadrži i element **sponsor**:

```
<xsd:complexType name = „Book“ > <xsd:sequence>  
  <xsd:choice>  
    <xsd:element name = „Author“ type = „Person“ .../>  
    <xsd:group ref = „EditorSponsor“ />  
</xsd:choice> </xsd:sequence> </xsd:complexType>
```



```

<xsd:group name = „EditorSponsor“ >
    <xsd:sequence> <xsd:element name = „Editor“ type=„Person“ />
    <xsd:element name = „Sponsor“ type = „Org“ />
</xsd:sequence> </xsd:group>

```

Grupe atributa

Opis sheme sa grupom atributa:

```

<xsd:attributeGroup name = „PriceInfo“ >
    <xsd:attribute name = „curr“ type = „xsd:string“ />
    <xsd:attribute name = „val“ type = „xsd:decimal“ />
</xsd:attributeGroup>
<xsd:complexType name = „Book“ >
    ...
    <xsd:attributeGroup ref = „PriceInfo“ />
</xsd:complexType>

```

37.XML sheme. Identifikatori i strani ključevi. Primeri.

Definicija kljuceva

- Ključevi jednoznačno identifikuju element i definisani su kao deo elementa
- Uveden je specijalni element koji se ugnježdava, nazvan key , u okviru kog su uvedeni:
 - selector: opisuje kontekst na koji se odnosi ključ
 - field: opisuje koje je polje ključ u kontekstu opisanim selektorom (Ako ima više elemenata field u okviru ključa, tada se radi o tzv. kompozitnom ključu)
- Vrednosti za selector i za field su XPath izrazi
- Validacija ključa u XML-u se realizuje na slededi način:
 - a. Evaluira se selector i dobije sekvenca čvorova
 - b. Evaliraju se vrednosti za field na sekvenci čvorova i dobije se skup uređenih n-torki vrednosti
 - c. Proverava se da li ima duplikata u skupu uređenih n-torki vrednosti

Primer sheme u kojoj je isbn definisano kao ključ za books u bib:

```

<element name = „bib“> <complexType> <sequence>
    <element book maxOccurs = „unbounded“ <complexType> <sequence> ... </sequence>
    <attribute name = „isbn“ type = „string“ />
</complexType> </element> </sequence>
<key name = „constraintX“ >
    <selector xpath = „book“ /> <field xpath = „@isbn“ />
</key>
</complexType> </element>

```

Reference (Strani kljucevi)

- Strani kljucevi predstavljaju deo definicije elemenata
- Uveden je specijalni element koji se ugnježdava, nazvan keyref (sa atributom refer) i u okviru njega elementi selector i field (sa atributom xpath)
 - Selector: određuje kontekst stranih kljuceva
 - Field(s): specificira strani ključ
 - Refer: daje opseg za reference (ograničenja za ključ)

Primer sheme za knjige koje referisu prema drugim knjigama:

```
<keyref name="constraintY" refer="constraintX">
  <selector xpath="book/references"/>
  <field xpath="@isbn"/>
</keyref>
```

38. XML sheme. Prostori imena. Primeri.

Prostori imena se zapisuju slično atributima

☐ Identifikuju se bilo sa "xmlns:prefix", ili sa "xmlns" (ako se radi o podrazumevanom prostoru imena)

☐ Dati prefiks se, korišćenjem prostora imena, pozezuje sa URI-jem

Opseg prostora imena je ceo elemenat u kome je taj prostor imena deklarisan – uključuje sam elemenat, njegove attribute i sve elemente koji su ugnježeni u njega

Primer:

```
<ns:a xmlns:ns="someURI" ns:b="foo">
  <ns:b>content</ns:b>
</ns:a>
```

39. XML i OOP.

Poredjenje XML-a i OO XML-a (Office Open XML):

Enkapsulacija:

-OO sakriva podatke

-XML čini da podaci budu eksplicitni

Hijerarhija tipova:

-OO definiše relacije podskup/nadskup

-XML deli strukturu, pa skupovne relacije nemaju smisla

Podaci i ponašanje:

-OO ih pakuje zajedno u jednu celinu

-XML razdvaja podatke od njihove interpretacije

40. XML i relacione baze podataka.

Poredjenje XML-a i relacione baze podataka:

Strukturne razlike

1. Drvo naspram tabele
2. Heterogene naspram homogenih
3. Opcionalni tipovi naspram striktnog tipiziranja
4. Nenormalizovani podaci naspram normalizovanih

Neke od sličnosti

1. Logička i fizička nezavisnost podataka

2. Deklarativna semantika
3. Generički model podataka

Vrste XML baza podataka

- XML-proširene baze podataka.
- Izvorne XML baze podataka.

41. Programerski modeli procesiranja XML.

Programerski modeli procesiranja XML-a

Ogromna korist od XML-a su standardni parseri i standardni API-ji (nezavisni od jezika) za njihovo procesiranje.

- DOM (**D**ocument **O**bject **M**odel) je objektno-orijentisana reprezentacija XML drveta parsiranja. DOM objekti sadrže metode kao što su `getFirstChild`, `getNextSibling` koje predstavljaju uobičajen način prolaska kroz drvo. Metode takođe mogu da modifikuju samo DOM drvo, tj. da i izmene XML, korišćenjem metoda `insertAfter` i dr.
- SAX (**S**imple **A**PI for **X**ML) se koristi u situacijama kada nisu potrebni svi podaci. Interfejs za parser je ključan u ovom pristupu. On poziva funkciju svaki put kada parsira instrukciju procesiranja, element itd. Razvijeni kod može odrediti šta treba raditi u datom slučaju, npr. modifikovati strukturu podataka ili ukloniti dete delove podataka.

42. XML i XPath. Primeri.

XML upiti

Upitni jezik predstavlja alternativni pristup procesiranju XML podataka.

Definiše se neka vrsta **šablona** koji opisuje prolaske (tj. putanje) od korenog čvora usmerenog grafa koji predstavlja XML.

Potencijalna korist ovakvog pristupa ogleda se u eksploataciji paralelizma, pogleda, mapiranja shema itd.

Veliki broj jezika je kreiran za postavljanje upita nad XML dokumentima a najznačajniji su XPath i XQuery.

XPath je W3C preporuka a sa pojavom XQuery postaje još popularniji. Koriste se za dobijanje i manipulisanje podacima iz XML baza podataka.

- Veliki broj ugrađenih funkcija.
- Korisnik ima mogućnost definisanja sopstvenih funkcija.
- Indeksi su potrebni radi efikasnijeg izvršavanja upita nad velikim kolekcijama dokumenata.

XPath u svom najprostijem obliku liči na opis putanje u sistemu datoteka:

`/mypath/subpath/*/morepath`

Međutim, XPath vrada **skup čvorova** koji predstavljaju XML čvorove (i njihova poddrveta) koji se nalaze na kraju zadate putanje. XPath na samom kraju putanje može sadržavati **testove za čvorove** i tako kreirati filter po tipu čvora metodama `text()`, `processing-instruction()`, `comment()`, `element()`, `attribute()`. XPath vodi računa o uređenju, može se postaviti upit tako da se vodi računa o uređenju i dobiti odgovor koji poštuje dato uređenje.

Xpath koristi iskaze putanja za kretanje kroz logičku, hijerarhijsku strukturu XML dokumenta.

- Dizajniran je da radi sa jednim XML dokumentom

Navigacija kroz stablo dokumenta

- dete čvor – čvor koji se nalazi na prvom nižem hijerarhijskom nivou u odnosu na drugi čvor
-- dete čvor ima tačno jednog roditelja
- čvorovi rođaci – čvorovi koji imaju istog roditelja
- čvor predak – bilo koji čvor na višem hijerarhijskom nivou » do koga vodi put preko grana stabla
- čvor potomak – bilo koji čvor na nižem hijerarhijskom nivou » do koga vodi put preko grana stabla

– osa kretanja predstavlja se punim ili skraćenim nazivom

- self:: ili .
- parent:: ili ..
- descendant:: ili //
- attribute ili @
- child:: podrazumeva se, ako se ništa ne napiše

– zamenski karakteri

- eng. wildcards
- * - zamenjuje bilo koji element
- @* - zamenjuje bilo koji atribut
- node() – zamenjuje bilo koji čvor bilo kog tipa

– poseduje operatore koji se mogu koristiti u XPath izrazima

– poseduje ugrađene funkcije

- Primer: Za sve putanje koji počinju od elementa knjiga, ispitati da li se u okviru njih nalazi element odeljak i vratiti kao rezultat element naslov koji je dete elementa odeljak. – knjiga//odeljak/naslov

– knjiga//odeljak/naslov



- Selektovati sve elemente starost u dokumentu: //starost
- Selektovati sve elemente koji su deca korenog elementa student: /student/*
- Selektovati sve studbr attribute elemenata student u dokumentu: /student*@studbr+
- Selektovati sve elemente starost: /**name()='starost'+
- Selektovati sve pretke od svih elemenata starost koji su deca od elementa student.
/student/starost/ancestor::*

43. XML i XQuery. Primeri.

XQuery je jezik koji je projektovan da bude mali, da se lako implementira i da bude lako razumljiv jezik.

On je nastao sa idejom da obezbedi upitni jezik koji ima istu širinu funkcionalnosti kao SQL nad relacionim bazama podataka.

Izrazi u XQuery-u upadaju u 6 širokih tipova:

- Izrazi putanje.
- Konstruktori elemenata.
- FLWR izrazi.
- Uslovni izrazi.
- Kvantifikovani izrazi.
- Izrazi koji u sebi uključuju korisnički definisane funkcije.

XQuery – izrazi putanje

- XQuery obezbeđuje izraze putanja koje su nadskup od onih u XPath-u.
 - Iz dokumenta koji sadrži zaposlene i njihovu mesečnu zaradu, izdvojiti godišnju zaradu za zaposlenog sa imenom Marko.
`//zaposleni[ime="Marko"]/zarada * 12`
 - U dokumentu "zoo.xml" pronadi sve slike u poglavljima od 2 do 5
`document("zoo.xml")//poglavlje[2 TO 5]//slika`

XQuery – konstruktori elemenata

- Ponekad je neophodno za upit da kreira ili generiše elemente. Takvi elementi se mogu generisati direktno u upitu u okviru iskaza nazvanog konstruktori elemenata.
 - Generisati elemente <zaposleni> koji imaju zapid atributa. Vrednost atributa i sadržaj elementa su specifikovani promenljivom \$id koja je dodeljena u nekom drugom delu upita.
`<zaposleni zapid = {$id}> {$ime} {$posao} </zaposleni>`

Xquery – FLWR iskazi

- FLWR se izgovara kao "flower".
- Ovaj izraz je upit koji se sastoji od FOR, LET, WHERE I RETURN klauze.
 - Izlistati sve izdavače koji su izdali više od 100 knjiga.

```
<veliki_izdavaci>
{ FOR $p IN distinct(document("bib.xml")//izdavac)
  LET $b := document("bib.xml")//knjiga[izdavac = $p]
  WHERE count($b) > 100
  RETURN $p
}
</ veliki_izdavaci >
```

Xquery – uslovni izrazi

- Uslovni izrazi ocenjuju test izraze i onda vrađaju jedan od dva rezultujućeg izraza. Ako je vrednost test izraza tačno onda se vrađa kao rezultat vrednost prvog rezultujućeg izraza, u suprotnom, vrađa se vrednost drugog.
- Napraviti listu svih knjiga uređenih po naslovu. Za beletristiku, uključiti izdavača a za sve ostale autora.

```
FOR $k IN //knjiga RETURN
  <knjiga>
    {$k/naslov, IF ($k[@zanr = "Beletristika"]) THEN
      $k/izdavac ELSE $k/autor}
  </knjiga>
  SORTBY (naslov)
```

Xquery – kvantifikovani izrazi

- SOME klauza i EVERY klauza - ekvivalentne kvantifikatorima koji se koriste u matematici i logici. – Pronaladi naslove svih knjiga u kojima su "jedrenje" i "surfovanje" pomenuti u nekom paragrafu.

```

FOR $k IN //knjiga
WHERE SOME $p IN $b//paragraf SATISFIES
  (contains($p, "jedrenje") AND contains($p,
"surfovanje"))
RETURN $k/naslov

```

Xquery – izrazi koji u sebi uključuju korisnički definisane funkcije

- Osim toga što je podržana centralna biblioteka funkcija sličnih onima u XPath-u, XQuery takođe daje mogućnost korisnicima da definišu funkcije koje de proširiti ovu biblioteku.
- Spisak korisnički definisanih funkcija može se nadi na adresi: http://www.w3schools.com/xpath/xpath_functions.asp

44. Preporuke za definisanje XML shema.

Svi elementi i atributi treba da koriste kamillju notaciju veliko slovo za pisanje složenica, na primer adresaPosta, i treba da izbegavaju crtice, razmake ili drugu sintaksu.

Čitljivost je važnija od dužine oznake, bar do određene tačke. Uvek treba obezbediti balans između veličine dokumenta i čitljivosti, pri čemu treba favorizirati čitljivost gde god je to moguće.

Izbegavati skradenice kod naziva elemenata, atributa i tipova. Izuzeci bi trebali da budu samo one skradenice koje su dobro poznate u datom poslovnom domenu, na primer ID (označava identifikator) i sl.

Naziv svakog tipa treba da se završava sa Type, na primer AdresaPostaType.

Enumeracije bi trebalo da koriste imena, a ne brojeve, i enumerisane vrednosti bi takođe trebale da budu pisane u skladu sa kamiljom notacijom.

Imena ne bi trebalo da uključuju ime strukture u kojoj se sadrže, na primer element koji predstavlja ime unutar nadređenog elementa customer treba da bude nazvan jednostavno name, a ne customerName.

Imenovati jednostavne i kompleksne tipove samo za one tipove kod kojih je verovatno da de se ponovo koristiti. Ako struktura postoji samo na jednom mestu, nju treba definisati ma licu mesta korišćenjem anonimnog tipa.

Izbegavati upotrebu mešovitog sadržaja.

Element definisati na globalnom nivou samo ukoliko taj element može da bude koreni element u XML dokumentu.

Koristiti konzistentna imena za prostore imena i izbegavati upotrebu standardno definisanog prefiksa :

- xml (definisano u XML standardu)
- xmlns (definisano u XML standardu, u delu koji se odnosi na prostore imena)
- xs ili xsd (definisano u <http://www.w3.org/2001/XMLSchema>)
- xsi (definisano u <http://www.w3.org/2001/XMLSchema-instance>)

Pokušajte da razmotrite pitanje verzionisanja ved na početku dizajniranja shema. Neke preporuke koje se odnose na to:

- Ako je važno da nove verzije sheme budu kompatibilne unazad, tada svi dodaci šemi trebaju biti opcionalni.

- Ako je važno da postojeće aplikacije budu u stanju da čitaju novije verzije datog dokumenta, razmotriti dodavanje elemenata `any` i `anyAttribute` na kraju svih definicija.

Preporučuje se definisanje `targetNamespace` u okviru svoje sheme, čime se bolje identifikuje data shema i olakšava se njena modularizacija i ponovna upotreba.

Preporučuje se da se okviru elementa `schema` postavi `elementFormDefault="qualified"`. Na taj način de biti olakšano čitanje kvalifikovanih prostora imena u rezultujućem XML-u.

5. Programski jezi JavaScript

1. Karakteristike jezika JavaScript.

Programski jezik JavaScript je jedan od najpopularnijih programskih jezika na svetu. Kreiran je pre 20-ak godina. To je prvi i jedini skript-jezik koji direktno podržavaju pregledači veba (engl. web browsers). U današnje vreme se JavaScript često i intenzivno koristi i van pregledača veba. Razvoj platforme **Node.js**, koja je naročito postala popularna u poslednjih nekoliko godina, omogućio je da se ovaj jezik koristi i za razvoj na serverskoj strani (što je donedavno bio ekskluzivni domen tradicionalnih serverskih programskih jezika, kao što su: Java, Ruby, C#, PHP, itd).

Programski jezik JavaScript ima sledeće karakteristike:

Jezik visokog nivoa: Obezbeđuje apstrakcije koje programeru dozvoljavaju da ignoriše detalje računara na kome se skript izvršava. Dalje, upravljanje memorijom je automatizovano korišćenjem sakupljača otpadaka (engl. garbage collector), pa se programer može koncentrisati na sam programski kod, umesto da se brine o zauzetosti memorijskih lokacija. Takođe jezik obezbeđuje brojne konstrukcije koje programeru omogućavaju da rukuje sa modnim promenljivima, objektima i funkcijama.

Dinamičan: Kod programa napisanih na dinamičkim programskim jezicima de se prilikom izvršavanja realizovati mnoge aktivnosti koje se kod programa napisanih na statičkim programskim jezicima realizuju tokom prevođenja. Takav pristup ima i prednosti i mane. Što se prednosti tiče, ovakvim pristupom se obezbeđuju moderne karakteristike kao što su: dinamička tipizacija (engl. dynamic typing), kasno povezivanje (engl. late binding), refleksija, mogućnost korišćenja funkcionalne programske paradigme, mogućnost promene strukture objekta tokom izvršavanja, mogućnost korišćenja zatvorenja za funkcije itd.

Dinamički tipiziran: Kod dinamički tipiziranih jezika, promenljiva ne utiče na tip. Dakle, programer može dodeliti promenljivoj vrednost ma kog tipa, npr. promenljivoj koja je prethodno sadržavala celobrojnu vrednost se može dodeliti niska znakova (engl. string) i sl.

Slabo tipiziran: Za razliku od jako tipiziranih jezika, kod slabo tipiziranih jezika tip ne utiče na objekat. Dakle, ovde objekti nisu strogo tipizirani, čime je programeru dopuštena veća fleksibilnost. Sa druge strane, dinamička tipiziranost znači da nema provere sigurnosti tipa (što se pokušava popraviti u jezicima TypeScript i Flow).

Interpretiran: Ovo znači da je nepotrebna faza prevođenja programa pre njegovog izvršavanja, kao što je to npr. slučaj sa programskim jezikom C. Iako se može reći da je JavaScript interpretiran jezik, u praksi pregledači veba obično (zbog boljih performansi) prevedu JavaScript kod u međukod pre njegovog izvršenja. Međutim, ovo prevođenje je transparentno i ne zahteva dodatne korake, tj. dodatne akcije korisnika.

Podržava više paradigmi: Sam jezik ne forsira jednu fiksiranu paradigmu. Programer može pisati JavaScript i koristiti objektno-orijentisanu paradigmu - i to na dva načina: preko prototipa i preko klasa (počev od verzije ES2015 tj. ES6). Dalje, programer može pisati JavaScript programe korišćenjem funkcionalne paradigme (gde su funkcije "građani prvog reda"), ili korišćenjem imperativne paradigme (kao što je npr. slučaj u programskom jeziku C).

Stil kodiranja za JavaScript u stvari predstavlja **skup konvencija** koje se koriste pri pisanju JavaScript programa. Stil kodiranja se obično opisuje kao sporazum koji sklapaju članovi programerskog tima, kako bi

održali konzistentnost programskog koda u projektu. Postojanje fiksnih pravila za formatiranje programskog koda umnogome pomaže da kod bude **čitljiviji i lakši za održavanje**.

Pored mnogobrojnih postojećih, u ovom trenutku su najpopularniji stilovi kodiranja opisani sa sledeća dva dokumenta:

1. Google JavaScript Style Guide.
2. Airbnb JavaScript Style Guide

Konvencije kodiranja koje de bi ti korišćene u okviru ovog kursa:

- U primerima i u opisu konstrukcija bide korišćena aktuelna verzija JavaSkripta. `t="_blank"` -.
- Uvlačenje: koristite se razmaci, a ne tabovi, pri čemu de novi nivo uvlačenja biti markiran sa tri razmaka.
- Tačka-zapeta: usvojena je praksa da se znak tačka-zapeta koristi na kraju svake naredbe.
- Dužina linije: maksimalna dužina jedne linije 80 znakova.
- Linijski komentari: u programskom kodu koristiti isključivo linijke komentare a blokvske komentare koristiti isključivo u dokumentacione svrhe.
- Bez "mrtvog" programskog koda: ne ostavljati iskomentaran stari programski kod. U okviru programskog koda treba da ostane samo sadržaj koji je potreban.
- Komentarisati samo kada je korisno: ne dodavati komentare ako oni ne doprinose razumevanju programskog koda.
- Imenovanje funkcija, promenljivih i metoda: nazivi funkcija, promenljivih i metoda uvek počinju malim slovom tj. pri zapisu se koristi tzv. **kamiljaNotacija**. Jedino u slučaju kada se radi o nazivu označenom sa **private**, tada naziv počinje podvlatkom (tj. znakom `_`), a nastavak naziva je u skladu sa kamiljomNotacijom.
- Imenovanje konstruktorskih funkcija i klasa: nazivi konstruktorskih funkcije i klasa koriste tzv. **PaskalNotaciju**. I tu se, slično kao kod kamilje notacije, velikom slovom signalizira početak nove reči u nazivu - samo što naziv počinje velikim, a ne malim slovom.
- Imenovanje datoteka: nazivi datoteka treba da budu napisani malim slovima, pri čemu de reči u nazivu biti razdvojene crticom (ili minusom, tj. znakom `-`).
- Deklaracije promenljivih: uvek treba deklaristi promenljive, kako bi se na taj način izbeglo "zagađivanje" stvaranjem globalnih objekata. Pri deklarisanju promenljivih ne koristiti `var`. Preporučuje se korišćenje `const` za deklaraciju promenljive, a let treba koristiti samo u slučaju kada de promenljivoj biti menjana vrednost posle inicijalne dodele.
- Beline: beline treba pametno koristiti radi povedanja čitljivosti programskog koda. Na primer, postaviti razmak iza ključne reči iza koje slede otvorena zagrada; postaviti razmak pre i posle binarnih operacija (`+`, `-`, `*`, `/`); posle svake od sekcija unutar naredbe `for`; posle svakog znaka tačka-zarez (tj. znaka `;`); posle svakog zareza (tj. znaka `,`) itd.
- Znaci za kraj linije: ubaciti prazne linije kako bi se razdvojile sekvence logički povezanih operacija.
- Apostrofi i navodnici: apostrof (tj. znak `'`) treba da ima prioritet u odnosu na navodnik (tj. znak `"`). Naime, navodnici su uobičajeni kod HTML atributa, pa korišćenje apostofa pomaže da se olakšaju problemi pri radu sa niskama koje sadrže HTML opise.

Jedan od popularnih alata koji olakšava sređivanje i formatiranje programskog koda je **Prettier**.

2. Razvoj jezika JavaScript.

Godine **1993**, Nacionalni centar za superračunarske aplikacije (engl. National Center for Supercomputing Applications - **NCSA**), deo Univerziteta u Ilinoisu, objavio je Mosaic, prvi popularni grafički pregledač veba, koji je odigrao značajnu ulogu u razvoju veba u to vreme.

Godine **1994.** godine u gradu Mauntin Vju, u Kaliforniji, osnovana je kompanija pod nazivom **Mosaic Communcations**, koja je zaposlila vedinu autora originalnog Mosaic veb pregledača, kako bi razvili novi pregledač, koji de zameniti Mosaic. Prva verzija tog novog pregledača, **Mosaic Netscape 0.9**, objavljena je krajem 1994. godine. Za samo četiri meseca ovaj veb pregledač je ved zauzeo tri četvrtine tržišta i postao je najpopularniji veb pregledač 90-tih godina 20. veka. Kako bi izbegli probleme sa autorskim pravima sa NCSA, veb pregledač je iste godine dobio nov naziv, **Netscape Navigator**, a kompanija se nazvala **Netscape Communications**.

U kompaniji Netscape su na vreme shvatili da bi Veb trebalo da postane dinamičniji. Osnivač kompanije, Mark Andresen, tvrdio je da HTML jeziku za označavanje treba pratedi programski jezik koji veb dizajneri i programeri mogu lako da koriste za sklapanje komponenti kao što su slike i dodaci, čiji bi se kod pisao direktno u HTML kodu veb stranice. Kako bi uopšte započeli sa radom, kompanija Netscape Communications je morala da sarađuje sa kompanijom **Sun Microsystems** da bi u ovaj pregledač ugradili njihov statični programski jezik Javu i time se borili sa konkurentskom kompanijom Microsoft za vedu naklonost korisnika i za usvajanje veb tehnologija i platformi.

Odlučeno je da se kreira programski jezik komplementaran Javi, sa sličnom sintaksom, što je u startu značilo odbacivanje podrške za druge programske jezike kao što su Perl, Pajton, TCL ili Scheme. Kako bi odbranili ideju JavaSkripta u odnosu na ponudu konkurenata kompanije bio je potreban **prototip**.

Programer Brendon Ajk je napisao prototip bududeg programskog jezika za 10 dana, u maju 1995. godine.

Iako je razvijan pod kodnim nazivom **Moka**, jezik je prilikom prvog objavljivanja beta verzije Netscape Navigator-a, septembra 1995. godine zvanično nazvan **Lajvskript**. Međutim, jezik je ubrzo, u decembru iste godine, prilikom objavljivanja Netscape Navigator 2.0 beta 3 verzije, preimenovan u **JavaSkript**.

3. JavaScript okruženje za izvršavanje.

JavaSkript je skriptni jezik, stoga je neophodno da se “smesti” u neki kontejner koji bi mu omogućio rad, taj kontejner se zove JavaSkript okruženje za izvršavanje (engl. JS Runtime Environment). Postoje dva tipa okruženja:

1. ugrađeno u pregledač (za rad na klijentu)
 2. node.js (za rad na serveru)
- ali je način rada ova dva okruženja konceptualno isti.

JavaSkript okruženje za izvršavanje se sastoji od slededih komponenti:

1. **JavaSkript mašina** (engl. JS Engine) koja sadrži:
 - Hip (engl. Heap)
 - Stek (engl. Stack)
2. **Spoljašnji API-ji**
3. **Red povratnih poziva** (engl. Callback Queue)
4. **Petlja za događaje** (engl. Event loop)

JavaSkript je programski jezik koji se izvršava u jednoj niti i on sam ne može da izvršava više istovremenih paralelnih radnji. Ipak, ovako organizovano okruženje za izvršavanje omogućava više paralelnih radnji (tzv. asihrono izvršavanje).

4. JavaScript okruženje za izvršavanje. JavaScript mašina.

JavaScript (JS Engine) mašina je aplikacija pisana u C++ koja izvršava JavaScript kod. Prilikom izvršavanja, radi obezbeđenja najboljih performansi, pri svakom pokretanju JavaScript programa vrši se “prevođenje u pravom trenutku” (engl. “**just-in-time compilation**”). Najpoznatija JavaScript mašina je Google-ov “**V8**” koji se inače nalazi u pregledaču Chrome ali i u node.js-u. Ostale mašine su: Mozilla-in “Rhino”, Firefox-ov “SpiderMonkey”, Microsoft-ov “Chakra”. JavaScript mašina za rad koristi dva tipa memorije: Hip i Stek.

Stek: Komponenta stek je dobila ime zbog načina pristupa ovom delu memorije (engl. Last In First Out - **LIFO**). Stek služi za privremeno čuvanje podataka (funkcija, argumenata, lokalnih promenljivih) pri izvršavanju koda. Stek karakterišu i velika brzina rada i unapred ograničen kapacitet.

Hip: Hip predstavlja nestruktuiranu memoriju u kojoj se smeštaju i čuvaju dinamički podaci (objekti). Ovaj tip memorije je sporiji od Steka. Za razliku od Steka, Hip je ograničen samo veličinom raspoložive memorije. Podaci čuvani u Hipu nemaju međusobnih zavisnosti i može im se pristupiti nezavisno - svaki podatak ima svoju adresu.

5. JavaScript okruženje za izvršavanje. Spoljasnji API.

Ved je istaknuto da je JavaScript skriptni jezik, pa je prilikom izvršavanja neophodno da se “smesti” u neki kontejner koji bi mu omogućio rad. Pod **Spoljašnji API-jem** se najčešće podrazumevaju **objekti i metode** koje omogućavaju komunikaciju JavaScripta sa spoljnim svetom. Spoljašnji API zavisi od okruženja, pa se tako razlikuje API pregledača od API-ja node.js.

API pregledač

o Objekt model dokumenta (engl. Document Object Model - DOM) - povezuje veb stranicu sa skriptom tj. programskim jezikom, jer rad sa HTML, SVG, ili XML dokumentima kao objektima nije deo jezika JavaScript.

- *window* objekat API

- setTimeout()
 - setInterval()
 - ...

- *document* objekat API

- event API

- web worker API

- ...

- o *navigator* objekat

- o *XMLHttpRequest* objekat za interakciju sa serverom

- o animation API

- o HTML5 API-ji

- video i audio API
 - canvas API
 - fullscreen API
 - geolocation API
 - local storage API
 - notifications API
 - ...

API node.js

- *Process* objekat
- *Buffer* objekat
- *require()*
- *setTimeout()*
- ...**API trede strane**

Pored API-ja okruženja (pregledača ili node.js), postoje i API-ji od spoljnih resursa (tzv. API trede strane), koji su neophodni ako treba da se koriste usluge te trede strane. Najpoznatiji takvi API-ji su: Google Maps, Twitter, Facebook, PayPal...

6. JavaScript okruženje za izvršavanje. Red povratnih poziva.

U red povratnih poziva se smešta deo koda koji je spreman da se pošalje na izvršavanje JavaScript mašini. Ova memorija se zauzima i oslobađa na način karakterističan za redove (engl. First In First Out - **FIFO**). Deo koda koji se šalje na izvršavanje JavaScript mašini se iz Reda povratnih poziva prebacuje na Stek. Odluku o tome kada de se izvršiti prebacivanje koda iz Reda povratnih poziva na Stek donosi komponenta **Petlja za događaje**. S obzirom da Red povratnih poziva sadrži zadatke koje treba da izvrši JavaScript okruženje, ponegde u literaturi se ovaj red još naziva i Red zadataka (engl. Task Queue).

7. JavaScript okruženje za izvršavanje. Petlja za događaje.

Ova komponenta JavaScript okruženja za izvršavanje je u stvari proces koji **stalno proverava** da li je Stek prazan i da li postoje neke funkcije u Redu povratnih poziva da se izvrše. Ukoliko se to desi, bira se prva na redu funkcija iz reda (ona koja najduže čeka - tj. "najstarija"), prebacuje se na Stek i počinje njeno izvršavanje.

6. Struktura JavaScript programa

1. Struktura JavaScript programa. Osnovni elementi jezika JavaScript.

Gradivni blokovi koji se koriste za izgradnju konstrukcija JavaScripta: Unicode znaci, tačke-zarezi, beline, razlikovanje velikih i malih slova, komentari, literali, identifikatori i rezervisane reči.

Unicode znaci

JavaScript programi se zapisuju korišćenjem Unicode znakova. To znači da dirilične i latinične reči mogu predstavljati imena promenljivih. Isto tako, reči na kineskom, japanskom ili arapskom jeziku mogu označavati promenljive. Naravno, ovakvo imenovanje, iako je mogude, nije baš uobičajeno - postoje različite **konvencije** za pisanje JavaScript programskog koda (pravila o imenima, načinu prelamanja teksta, način nazublivanja koda, postavljanje belina itd.), a ponekad i programerski tim definiše svoja pravila za pisanje programkog koda, koja su obično bazirana na nekoj od postojedih konvencija.

Tačka-zarez

Sintaksa JavaScripta je u velikoj meri podseda na programski jezik C, pa postoji mnogo primera koda u kojima se tačka-zarez nalazi na kraju svake naredbe. Međutim, za razliku od jezika C, tačke-zarezi nisu obavezni na kraju naredbe, ved sam unos znaka za prelazak u slededi red (osim u retkim specijalnim slučajevima) označava da je završena naredba u datom redu.

Beline

Beline su, isto kao u C-u, znaci koji ne ostavljaju prikaz na izlazu: razmak (ili blanko), znak za tabuliranje (ili tab), znak za novu liniju, itd. U jeziku JavaScript, beline nemaju značenje.

U jeziku JavaScript se **razlikuju velika i mala slova**, pa npr. promenljiva osoba se razlikuje od Osoba. Ovo pravilo takođe važi i za sve identifikatore.

Komentari

Postoje dve vrste komentara:

- blokovski - tekst, koji se može pružati kroz više linija, između `/*` i `*/`
- linijski - tekst od znakovne sekvence `//` pa do kraja linije

Literali

Literal je niz znakova koji predstavlja vrednost upisanu u samom izvornom kodu npr. broj, niska, logička vrednost ili neka vrednost sa složenom strukutrom, kao što su objektni literali ili nizovni literali. Npr. literali: `5`, `'jedna niska'`, `true`, `['a', 'b']`, `{boja: 'crvena', oblik: 'paravougaonik'}`.

Identifikatori

Identifikator (tj. ime) je sekvenca znakova koja se može koristiti za identifikaciju (tj. imenovanje) promenljive, funkcije ili objekta. Identifikator počinje sa slovom, sa znakom za dolar `$` ili sa podvlakom `_`. Identifikator može sadržavati i cifre.

Rezervisane reči

Kao što samo ime kaže, rezervisane reči (ključne) su rezervisane za konstrukcije JavaScript jezika i ne mogu se koristiti u svrhu koja je drugačija od propisane. To su npr. `break`, `do`, `typeof`, `case`, `else...`. Reči **NaN**, **Infinity** i **undefined** nisu u spisku rezervisnih reči, ali se ne preporučuje njihovo korišćenje kao identifikatora. Ove reči predstavljaju nemutirajude osobine globalnog objekta i njihova vrednost može samo da se čita.

2. Tipovi i vrednosti. Primitivni tipovi.

Jednoj promenljivoj se mogu dodeliti vrednosti različitih tipova, ali vrednosti koje se dodeljuju imaju svoj tip. Tipovi u JavaScriptu se dele u dve osnovne grupe:

1. Primitivni tipovi
2. Objektni tipovi

Primitivni tipovi

Kao što im samo ime kaže, primitivni tipovi su **najprostiji**. Vrednosti primitivnog tipa su nepromenljive, tj. nemutiraju. To znači da nakon stvaranja i čuvanja u memoriji ta vrednost na datom mestu se više ne može menjati. Nepromenljiva priroda podataka primitivnog tipa i jeste razlog što je za poređenje promenljivih koje sadrže primitivan tip vrednosti, dovoljno da porede samo vrednosti. Primitivni tipovi su: (5. i 6. su dva specijalna tipa)

1. Brojevni tipovi (**number**)
2. Niska tip (**string**)
3. Logički tip (**boolean**)
4. Simbol (**symbol** - uveden od verzije ES6)
5. Null
6. Undefined

Za razliku od programskog jezika C, gde je bila dopuštena izmena sadržaja niske, u jeziku JavaScript **niske su nemutiraju**.

Vrednosti primitivnog tipa se u literaturi često značavaju pojmom "primitivne vrednosti". Sada demo se detaljnije pozabaviti primitivnim vrednostima različitih tipova.

Brojevi

Ako se posmatra interna reprezentacija brojeva u jeziku JavaScript, moglo bi se reći da postoji jedan tip za brojeve, tj. da je svaki broj u stvari broj u pokretnom zarezu.

Brojevni literal je broj zapisan u programskom tj. izvornom kodu - u zavisnosti od toga kako je zapisan, to može biti celobrojni literal ili literal koji predstavlja broj u pokretnom zarezu.

Primeri:

zapis broja u fiksnom zarezu: 42, 0x42, 5354576767

zapis broja u pokretnom zarezu: 9,81 , .1234 (0.1234) ili npr. 2.998e8 (2.998 * 10⁸)

Niske

Niska tip predstavlja sekvencu znakova. U izvornom kodu ona se definiše pomoću niska-literal, koji je ograničen apostrofima ili navodnicima.

Niska može sadržavati i tzv. eskejp-sekvencu koja se interpretira prilikom prikaza niske - npr. eskejp-sekvencu \n predstavlja znak za kraj linije. Isto kao i u programskom jeziku C, skejp-sekvence počinju obrnutim slešom, tj. znakom \. Ove sekvence su korisne npr. kada navodnik treba da se nađe u okviru niske koju ograničavaju navodnici, kada apostrof treba da se nađe u okviru niske koju ograničavaju apostrofi i sl.

Primeri:

" Život je lep " , ' Život je lep ' ...

Logički tip

Podatak logičkog tipa može imati tačno jednu od sledeće dve vrednosti: tačno (zapisuje se sa *true*) i netačno (tj. *false*). Veliki broj operatora poređenja: ==, ===, <> itd. kao rezultat vraćaju logičku vrednost.

Naredbe grananja (npr. if naredba) i naredbe ciklusa (kao što je while naredba) koriste logičke vrednosti da bi

odredile dalji tok izvršavanja programa.

Kontrolne strukture za grananje i za cikluse nisu previše stroge: one pored čisto logičkih podataka (tj. vrednosti true ili false), prihvataju i podatke drugih tipova koji se tumače (interpretiraju) kao tačni ili kao netačni.

Vrednosti koje se interpretiraju kao netačne su: 0, -0, NaN, undefined, null, "" i "". Sve ostale vrednosti se tumače kao tačne.

Specijalni tip null

Vrednost null predstavlja specijanu vrednost koja ukazuje na nepostojanje (odsustvo) vrednosti. Sličan koncept postoji i u drugim programskim jezicima, npr. null u jeziku C#, kao i nil ili None u jeziku Pajton.

Specijalni tip undefined

Vrednost undefined ukazuje da promenljiva još nije inicijalizovana ili da nedostaje njena vrednost.

Uobičajeno je da undefined bude povratna vrednost funkcije koja ne vrada vrednost. Pored toga, ako je prilikom poziva funkcije broj argumenata u pozivu manji od broja parametara, tada de na početku izvršavanja funkcije vrednost parametara kojima nije prosleđen argument biti undefined.

Provera da li je data promenljiva undefined se vrši tako što se tip promenljive (dobijen primenom typeof operatora) uporedi bez konverzije tipova (tj. primenom operatora ===) sa tipom undefined.

3. Tipovi i vrednosti. Objektni tipovi.

U jeziku JS sve što nije primitivni tip predstavlja objektni tip. Funkcije, nizovi i objekti su primeri objektnih tipova. Svako od njih ima svoje specifične karakteristike, ali svi oni nasledjuju svojstva objektnog tipa – imaju osobine i metode.

Vrednosti objektnog (referentnog) tipa se čuvaju u delu memorije koji se naziva hip (*heap*). Za razliku od vrednosti primitivnog tipa, vrednost referentnog tipa može da se menja tokom vremena.

Dodeljivanje referentne vrednosti promenljivoj izvršava istu akciju kao i kod primitivnih: vrednost zauzima mesto u memoriji, a promenljive se postavlja tako da ukazuje na to mesto u memoriji. Međutim, dodeljivanje postojeće promenljive novoj promenljivoj ne kopira tu vrednost na novo mesto u memoriji, već samo podesi da “nova” promenljiva bude preusmerena tako da referiše na to isto “staro” mesto. Stoga de “stara” i “nova” promenljiva postati “vezane”: nakon promene vrednosti jedne (bilo “stare”, bilo “nove” promenljive), dolazi do promene obe promenljive, jer obe referišu na isto mesto u memoriji. Dakle, dodelom se ne vrsi kopiranje samog sadržaja, već kopiranje reference.

Pored primitivnih tipova za niske, brojeve i logičke vrednosti, u jezgri JavaScripta postoje predefinisani tj. “ugrađeni” objekti čija su imena ista kao imena prostih primitiva (mada nazivi počinju sa velikim slovom!): String, Number i Boolean. Nadalje, podaci primitivnih tipova tokom izvršavanja programa bivaju po potrebi “obavijeni” tj. u letu se napravi nemutirajući objekat koji sadrži istu primitivnu vrednost, pa se operacija umesto nad primitivnom vrednošću izvrši nad vrednošću objekta-omotača. Automatsko omotavanje se najčešće dešava kada programer nad primitivnom vrednošću pozove neku od metoda dostupnih iz objekta-omotača. Dakle, od primitivnog tipa se napravi objektni koji ima istu vrednost, ali i dodatne funkcionalnosti/metode.

4. Promenljive. Opseg definisanosti i konteksti.

Promenljive su imenovane lokacije u memoriji u kojima mogu da se čuvaju podaci. Vrednost promenljive može da se menja za vreme izvršavanja programa, pa otuda i naziv promenljiva. Imena promenljivih u JavaScriptu se mogu sastojati od proizvoljne kombinacije slova i brojeva, ali postoje neka pravila:

- Prvi znak mora biti slovo alfabeta ili podvlaka (engl. underscore) tj. znak `_`.
- Rezervisane reči ne mogu se koristiti kao ime promenljive.
- Velika i mala slova se razlikuju (key sensitive), uobičajeno je da se promenljive pišu malim slovima.

Deklarisanje i definisanje promenljive su dva različita postupka. Deklaracija promenljive je postupak stvaranja promenljive, tada se definiše ime promenljive i obezbeđuje prostor za njeno skladištenje, dok je definisanje promenljive postupak kada se promenljivoj dodeljuje neka vrednost koja se smešta u prostor za skladištenje. Primer rada sa promenljivima:

- Definicija promenljive: `let prom = 5;`
- Prikaz vrednosti promenljive: `console.log(prom);`
- Promenljiva može da menja vrednost: `prom = 10;`
- Uvecavanje, smanjivanje vrednosti promenljive: `prom+= 2; prom-=4;`
- Definisanje više promenljivih jednom naredbom: `let prvi = 1, drugi = 2;`
- Promenljive su nezavisne: `let a = 5, b = 2; b = a; a = 7; // sada je a = 7 i b = 5.`

Pojam opsega definisanosti promenljive predstavlja deo programskog koda u kom je neka promenljiva dostupna. Promenljivoj se može pristupiti samo iz njenog opsega definisanosti.

Prema **vremenu** definisanja promenljive opsezi se mogu podeliti na:

- Statički (još se naziva i leksički) opseg definisanosti - definiše se u trenutku prevođenja, tako da funkcija tada zapamti reference na koje ukazuju promenljive u lancu leksičkih opsega. Ovo se u žargonu naziva “rano vezivanje” (engl. early binding). Leksički opseg direktno zavisi od mesta gde je deklarirana promenljiva u samom programskom kodu, pa se može zaključiti da se ovaj tip opsega promenljive definiše u trenutku pisanja koda. Shodno tome, ovaj opseg se ne može menjati tokom vremena (osim samim menjanjem koda), pa se zato još zove i statički opseg definisanosti.
- Dinamički opseg definisanosti - definiše se u vreme izvršavanja koda, pa se vezuje za reference na koje ukazuju promenljive u tom trenutku kada se izvršava kod. Ova vrsta vezivanja se u žargonu naziva “kasno vezivanje” (engl. late binding). Kod programskih jezika koje koriste dinamički opseg, nakon potrage za promenljivom u okviru matične funkcije, krede potraga za promenljivom u funkciji koja je pozvala matičnu funkciju, itd.

Prema **veličini**, opsezi definisanosti se mogu podeliti na:

- 1) Lokalni opseg definisanosti - sve do verzije ES6 u JavaScriptu, promenljivu lokalnog opsega je bilo moguće napraviti isključivo korišćenjem funkcije (što je smatrano jednom od najvedih mana samog jezika). Novom revizijom jezika “ES6” je omogućeno definisanje lokalnog tzv. blokovskog opsega definisanosti uz pomoć rezervisane reči `let`.
 - Funkcijski opseg definisanosti - kada se promenljiva deklarira unutar funkcije, za nju se definiše lokalni opseg definisanosti koji se još zove “funkcijski opseg”. Lokalna promenljiva funkcije i njen funkcijski opseg definisanosti postoje samo tokom izvršavanja te funkcije.
 - Blokovski opseg definisanosti - uvedena u ES6 je, gde nova ključna reč `let` omogućuje deklaraciju promenljive unutar bloka koda i samim tim definiše lokalni blokovski opseg definisanosti (blok je deo koda unutar vitičastih zagrada `{ }` - to može biti npr. unutar neke petlje ili grananja). Jedna od bitnih

karakteristika kreiranja blokovskog opsega definisanosti sa ključnom reči `let` je da se pri tome ne vrši pomeranje deklaracija promenljivih na vrh tj. na početak bloka u kome su one deklarirane.

- 2) **Globalni opseg** - svaka promenljiva koja nije definisana u okviru funkcije pripada globalnom opsegu, a to znači da je promenljiva dostupna svim delovima programa. Ukoliko se promenljiva ne deklarise, ona de postati globalna pa čak i ako je definisana (tj. dodeljena joj je vrednost) unutar funkcije. Najčešće se globalna promenljiva kreira greškom, pa je preporuka da se koristi striktni mod (aktivira se sa `use strict`) koji pri kreiranju globalne promenjive prijavljuje grešku.

Pre nego što je uvedena rezervisana reč `let`, rezervisana reč `var` se koristila za deklarisanje promenljive. Promenljiva deklarirana pomoću `var` nije imala lokalni opseg definisanosti. Dakle, ako na taj način deklariramo promenljivu, to ime je zauzeto i ne smemo ga ponavljati, a česce su nam potrebne neke pomocne promenljive u manjem delu koda/bloku, pa sa `let` ne moramo da razmišljamo da li smo ranije iskoristili taj naziv. (takodje, mislim da u jednoj funkciji može biti više promenljivih istog imena, ako je jedna u spoljasmem, a druga u unutrašnjem bloku, tada u unutrašnjem vazi samo ta druga, a kad se izadje iz njega, onda je prva opet vidljiva)

Ako treba raditi sa promenljivom čija se vrednost nede menjati tokom njenog životnog ciklusa, preporučuje se da se takva promenljiva deklarise korišćenjem rezervisane reči `const`.

Kontekst - Treba napomenuti da opseg definisanosti nije isto što i okruženje tj. kontekst. Ova dva pojma se poistovecuju jer imaju neke sličnosti ali treba znati da je pojam opseg definisanosti je osobina promenjive, dok je kontekst osobina programskog kôda.

Kada se program pokrene, okruženje nije prazno - ono ved sadrži promenljive koje su deo jezičkog standarda, kao i promenljive koje obezbeđuju način interakcije programa sa sistemom koji ga okružuje. Tokom izvršavanja, program prolazi (ulazi i izlazi) kroz različite opsege definisanosti promenjive, a promenjive su tada ili u kontekstu ili van njega. Kontekst je sličan opsegu definisanosti jer takođe zavisi od pozicije u programu (leksički kontekst) ili vremena izvršavanja koda (kontekst izvršavanja).

5. Izrazi. Primarni izrazi.

Izrazi su delovi programskog kôda koji mogu biti evaluirani i čija se vrednost može izračunati.

Iako svaki sintaksno korektan izraz prilikom evaluacije dobije neku vrednost, u principu se razliku dva tipa izraza:

- izrazi **sa bočnim efektima** (npr. izraz kojim se promenljivoj dodeljuje vrednost, `x=3+4;`)
- izrazi **bez bočnog efekta** (npr. izraz `3+4;` izracunava se vrednost, ali se rezultat ne dodeljuje)

Vrste izraza su primarni, aritmeticki, logicki, niska-izrazi i izrazi leve strane.

Primarni izrazi su osnovne ključne reci i najjednostavniji izrazi u JavaScriptu, predstavljaju osnovne elemente za gradjenje svih izraza.

Tu spadaju literali, konstante i promenljive, kao i sledece ključne reci: `function`, `class`, `function*` (funkcija generator), `yield` (pauzira/nastavlja rad generatora), `yield*` (delegat drugog generatora), `async function*` (asinhroni funkcionalni izraz), `await` (pauzira/ nastavlja/ ceka zavrsetak za asinhronu funkciju), `/pattern/i` (regularni izraz (niska koja opisuje ili sparuje skup niski, u skladu s određenim sintaksnim pravilima)), `()` (zagrada, grupisanje).

6. Izrazi. Aritmetički izrazi.

U ovu kategoriju spadaju svi izrazi koji se evaluiraju u brojeve. Pri njihovom gradjenju koriste se aritmetički operatori i funkcije koje vraćaju brojeve. Aritmetički izrazi označavaju aritmetičke operacije i imaju veći prioritet od operatora poredjenja, koji imaju veći prioritet od logičkih operatora (primer za prioritete: $a + b > 0 \ \&\& \ c < 0$; prvo se izračunava zbir $a+b$, zatim se vrši poredjenje rezultata sa 0 i na kraju konjunkcija).

Spisak aritmetičkih operatora sa opisom:

+	unarni plus i sabiranje
-	unarni minus i oduzimanje
*	množenje
/	deljenje
%	modulo
++	inkrementiranje
--	dekrementiranje
**	stepen

Brojeve možemo posmatrati kao Number objekte (spomenuto ranije) i tada se mogu primeniti i sledeće metode:

- `isFinite()` - proverava da li je vrednost konačan broj (oznaka za beskonacno je **+Infinity**)
- `isInteger()` - proverava da li je vrednost ceo broj
- `isNaN()` - proverava da li je vrednost `Number.NaN`
- `isSafeInteger()` - proverava da li je vrednost siguran ceo broj
- `toExponential(x)` - pretvara broj u eksponencijalnu notaciju
- `toFixed(x)` - formatira u broj sa x cifara iza decimalne tačke
- `toPrecision(x)` - formatira broj na preciznost x
- `toString()` - vrada niska-reprezentaciju obuhvadene vrednosti
- `valueOf()` - vrada vrednost kao primitivnu

$47/0 \rightarrow +\text{beskonacno}$, $(47-50)/(100-100) \rightarrow -\text{Infinity}$, $(47/0) - (500/0) \rightarrow \text{NaN}$, $(1-1)/(2-2) \rightarrow \text{NaN}$

Pored ugrađenih metoda za rad sa brojevima, postoje i metodi za realizaciju matematičkih funkcija, za rad sa niskama, za rad sa nizovima i sl. koji vraćaju brojčane vrednosti - pozivi takvih metoda se takođe mogu nadi u okviru aritmetičkih izraza.

Bitovski izrazi su takodje aritmetički, dobijaju se primenom bitovskih operatora na celobrojne vrednosti.

Bitovski operatori su:

- `~` - bitovska negacija - invertuje svaki bit argumenta (unarni operator)
- `&` - bitovska konjunkcija - vrši konjunkciju pojedinačnih bitova dva argumenta
- `|` - bitovska disjunkcija - vrši disjunkciju pojedinačnih bitova dva argumenta
- `^` - bitovska ekskluzivna disjunkcija - vrši ekskluzivnu disjunkciju pojedinačnih bitova dva argumenta
- `<<` - levo pomeranje (šiftovanje) - vrši pomeranje bitova prvog argumenta ulevo za broj pozicija koji je naveden kao drugi argument
- `>>` - desno pomeranje (šiftovanje) - vrši pomeranje bitova prvog argumenta udesno za broj pozicija koji je naveden kao drugi argument

Unarna negacija ima najveći prioritet i desno je asocijativna, a zatim operatori siftovanja, pa redom konjunkcija, ekskluzivna disjunkcija i na kraju disjunkcija. Binarni bitovski operatori su levo asocijativni.

7. Izrazi. Logički izrazi.

Logički izrazi prilikom evaluacije vraćaju vrednosti logičkog tipa – **true** ili **false**. Nad logičkim vrednostima se mogu primeniti logički operatori:

&& - logička konjunkcija, vraća vrednost true ako su oba izraza true

|| - logička disjunkcija vraća vrednost true ako je barem jedan izraz true

! - logička negacija vraća vrednost true ako je izraz false, odnosno false ako je izraz true.

U logičkim izrazima mogu učestvovati operatori koji vraćaju logičku vrednost. Takvi operatori su npr. operatori poređenja, koji upoređuju vrednosti dva izraza, tj. vrednost izraza levo od operatora poređenja poredi se vrednošću izraza desno od operatora poređenja. To su:

== vraća vrednost true ako su vrednosti jednake, tj. iste (npr. $x == y$)

=== vraća vrednost true ako su vrednosti identične, tj. jednake i ako su istog tipa (npr. $x === y$)

!= vraća vrednost true ako vrednosti nisu jednake (npr. $x != y$)

!== vraća vrednost true ako vrednosti nisu identične (npr. $x !== y$)

> vraća vrednost true ako je vrednost s leve strane veća od vrednosti s desne (npr. $x > y$)

< vraća vrednost true ako je vrednost s leve strane manja od vrednosti s desne (npr. $x < y$)

>= vraća vrednost true ako je vrednost s leve strane veća ili jednaka od vrednosti s desne (npr. $x >= y$)

<= vraća vrednost true ako je vrednost s leve strane manja ili jednaka od vrednosti s desne (npr. $x <= y$)

Napomena: operatorima poredjenja se mogu porediti i specijalni tipovi (NaN, Infinity...) i niske.

Sa logičkim izrazima je usko vezan i operator uslovnog izraza, koji ispituje da li je uslov ispunjen (pri evaluaciji uslova dobija se vrednost true), pa ako jeste rezultat je vrednost izraza između upitnika i dvotačke, a ako nije rezultat je vrednost nakon dvotačke (npr. izraz $x = x > 0 ? 1 : -1$; predstavlja izraz koji vraća vrednost $\text{sgn}(x)$).

8. Izrazi. Niska-izrazi. Šabloni za niske. Primeri.

Niska-izrazi prilikom evaluacije vraćaju vrednosti znakovnih niski (niske karaktera). Prilikom kreiranja novih niski, često se koristi operator spajanja +. Kao što je već istaknuto pri opisu omotavanja, niska se može posmatrati kao String objekat i tada su nam na raspolaganju metode:

- `charAt()` - vraća znak/karakter na datoj poziciji
- `charCodeAt()` - vraća Unicode vrednost za znak na datoj poziciji
- `concat()` - vraća spoj dve niske
- `fromCharCode(code)` - pretvara Unicode vrednost code u odgovarajući znak
- `indexOf()` - vraća poziciju prve pronađene pojave date vrednosti u niski
- `lastIndexOf()` - vraća poziciju poslednje pronađene pojave date vrednosti u niski
- `localeCompare()` - poredi dve niske, uzimajući u obzir aktuelnu lokaciju
- `match()` - pretražuje nisku radi uklapanja regularnog izraza i vraća nađeno uklapanje
- `replace()` - pretražuje nisku radi nalaženja vrednosti ili uklapanja regularnog izraza i vraća nisku u kojoj su izvršene zamene
- `search()` - pretražuje nisku radi nalaženja vrednosti ili uklapanja regularnog izraza i vraća poziciju nađenog uklapanja
- `slice()` - vraća izdvojeni deo niske
- `split()` - vraća niz podniski originalne niske, zavisno od prosleđenog separatora
- `substr()` - vraća podnisku niske date dužine, počev od početne pozicije

- `substring()` - vraća podnisku niske između dva data indeksa
- `toLocaleLowerCase()` - vraća nisku pretvorenu u mala slova, uzimajući u obzir aktuelnu lokaciju
- `toLocaleUpperCase()` - vraća nisku pretvorenu u velika slova, uzimajući u obzir aktuelnu lokaciju
- `toLowerCase()` - vraća nisku pretvorenu u mala slova
- `toString()` - vraća vrednost odgovarajućeg `String` objekta omotača
- `toUpperCase()` - vraća nisku pretvorenu u velika slova
- `trim()` - vraća nisku u kojoj su uklonjene beline sa početka i sa kraja
- `valueOf()` - vraća niska-vrednost, tj. vrednost primitive

Primeri: `console.log(„string“.charAt(2))` , `.charCodeAt(3)` , `.concat(„nesto“, „josNesto“)`,
`console.log(„Ana voli Milovana“.split(„“), .split(„a“), substr(1, 5)`
`console.log(String.fromCharCode(106))`

Šabloni za niske: uvedeni su u verziji JavaSkripta ES6 (ta verzija se označava i sa ES2015). To su niska-literali koji dopuštaju modniji način za definisanje niski.

Šabloni za niske su, umesto apostrofa ili navodanika, ograničeni obrnutim apostofom (tj. znakom ```). U okviru šablona za niske se mogu nalaziti mesta za zamenu, koja su označena tako što su obuhvaćena vitičastim zagradama ispred kojih se nalazi znak za dolar, npr. `${izraz}`. U prethodnom primeru, izraz se nalazi unutar mesta za zamenu i on se prosleđuje funkciji, koja vrši spajanje delova u jednu nisku. Funkcija de pre spajanja u nisku izvršiti evaluaciju izraza u okviru mesta za zamenu i dobijene rezultate spojiti u rezultujuću nisku.

Kod šablona za niske, beline (znaci za kraj linije, razmaci, tabovi i sl.) ne treba da se predstavljaju pomoću eskejp-znakova, već se mogu direktno uneti u šablon. Način formiranja niski preko šablona je elegantniji od "klasičnog" spajanja niski operatorom `+` i produkuje čitljiviji programski kod.

Primer: sledeće niske su ekvivalentne: `'Tri plus šest je ' + (3 + 6) + '.'` i ``Tri plus šest je ${3 + 6}.'`

Napomena: Ako se pojavi potreba da se u okviru šablona za nisku pojavi obrnuti apostof, to se može postići pomoću odgovarajuće eskejp-sekvence, tj. znaka ```.

9. Izrazi leve strane.

Ovi izrazi predstavljaju određene naredbe dodele (bilo direktno, bilo implicitno). To mogu biti:

- Promenljive
- Izrazi za pristupanje osobinama objekta (uglaste zagrade `[]`, npr. član niza)
- Operator `new`, kojim se kreira objekat
- Operator `super`, kojim se poziva roditelj
- Liste argumenata

Vrednosti se izrazima leve strane dodeljuju korišćenjem operatora dodele. Najčešći operator dodeljivanja je znak jednakost `=`. Operator dodele može se pisati zajedno sa binarnim aritmetičkim operatorima sa sledećim značenjem:

<code>=</code>	dodeljuje vrednost promenljive ili izraza s desne strane promenljivoj ili izrazu leve (npr. <code>x=y</code>)
<code>+=</code>	sabira izraze sa leve i desne strane i dodeljuje zbir promenljivoj ili izrazu leve (npr. <code>x = x + y</code>)
<code>-=</code>	oduzima izraze sa leve i desne strane i dodeljuje razliku promenljivoj ili izrazu leve (npr. <code>x = x - y</code>)
<code>*=</code>	množi izraze sa leve i desne strane i dodeljuje proizvod promenljivoj ili izrazu leve (npr. <code>x = x * y</code>)

- /= deli izraze sa leve strane izrazom sa desne strane i dodeljuje količnik promenljivoj ili izrazu leve strane (npr. $x = x / y$)
- %= računa ostatak pri deljenju izraza sa leve strane izrazom sa desne strane i dodeljuje ostatak izrazu leve strane (npr. $x = x \% y$)

10. Prioritet operatora.

Pod operatorima se najčešće podrazumevaju simboli koji se koriste kada treba izvršiti neku standardnu često korišćenu operaciju kao npr. dodela vrednosti, poredjenje vrednosti, aritmetičke operacije i sl.

Prioritet operatora određuje kojim de se redom operatori primenjivati, a asocijativnost operatora određuje sa kog kraja se odvija grupisanje operanada (počev od levog ili od desnog kraja). Svaki operator ima svoj nivo prioriteta i svoje pravilo asocijativnosti.

U narednoj tabeli su prikazani operatori i njihova asocijativnost. Redosled operatora u tabeli je dat po njihovom prioritetu - od onih sa najvišim prioritetom izvršavanja naniže:

nivo	simbol	Znacenje	asocijativnost
20	(...)	Zagrade za grupisanje	n/d
19	Pristup osobinama objekta tacka-notacijom	sleva udesno
	... * ... +	Pristup osobini objekta notacijom uglastih zagrada	sleva udesno
	new ... (...)	Operator new (sa argumentima)	n/d
18	... (...)	Poziv funkcije	sleva udesno
	new ...	Operator new (bez argumenata)	sdesna ulevo
17	... ++	Postfiksno uvecavanje	n/d
	... —	Postfiksno umanjivanje	n/d
16	! ...	Logicka negacija	sdesna ulevo
	~ ...	Bitovska negacija	sdesna ulevo
	+ ...	Unarni plus	sdesna ulevo
	— ...	Unarni minus	sdesna ulevo
	++ ...	Prefiksno uvecavanje	sdesna ulevo
	— ...	Prefiksno umanjivanje	sdesna ulevo
	typeof ...	operator typeof	sdesna ulevo
	void ...	operator void (vraca undefined)	sdesna ulevo
	delete ...	operator delete	sdesna ulevo
15	... ** ...	stepenovanje	sdesna ulevo
14	... * ...	mnozenje	sleva udesno
	... / ...	deljenje	sleva udesno
	... % ...	ostatak pri deljenju	sleva udesno
13	... + ...	sabiranje	sleva udesno
	... — ...	oduzimanje	sleva udesno
12	... << ...	Bitovsko siftovanje ulevo	sleva udesno
	... >> ...	Bitovsko siftovanje udesno	sleva udesno
	... >>> ...	Bitovsko neoznaceno siftovanje udesno	sleva udesno

11	... < ...	manje	sleva udesno
	... <= ...	Manje ili jednako	sleva udesno
	... > ...	vece	sleva udesno
	... >= ...	Vece ili jednako	sleva udesno
	... in ...	operator in	sleva udesno
	... instanceof ...	operator instanceof	sleva udesno
10	... == ...	jednakost	sleva udesno
	... != ...	razlicitost	sleva udesno
	... === ...	identicnost	sleva udesno
	... !== ...	neidenticnost	sleva udesno
9	... & ...	Bitovska konjunkcija	sleva udesno
8	... ^ ...	Bitovska ekskluzivna disjunkcija	sleva udesno
7	Bitovska disjunkcija	sleva udesno
6	... && ...	Logicka konjunkcija	sleva udesno
5	Logicka disjunkcija	sleva udesno
4	... ? ... : ...	Uslovni izraz	sdesna ulevo
3	... = ...	dodela	sdesna ulevo
	... += ...		
	... -= ...		
	... *= ...		
	... /= ...		
	... /= ...		
	... /= ...		
	... <<= ...		
	... >>= ...		
	... >>>= ...		
	... &= ...		
	... ^= ...		
	... = ...		
	...		
2	yield ...	operator yield	sdesna ulevo
	yield* ...	operator yield*	sdesna ulevo
1	prosirenje	n/d
0	... , ...	zarez/sekvenca	sleva udesno

11. Konverzija tipova i evaluacija izraza.

JavaScript je slabo tipiziran jezik, što znači da ne zahteva da se jasno deklarirše tip pri deklarisanju promenljive. Kod JavaScripta se tip promenljive implicitno deklarirše sa pridruživanjem određene vrednosti datoj promenljivoj.

JavaScript je takođe jezik dinamičkih tipova jer je moguda promena tipa u toku izvršenja programa. Rezultat konverzije je uvek neka primitivna vrednost, tj. ne postoji konverzija čiji je rezultat neka složena vrednost (kao što je objekat ili funkcija).

Primeri evaluacije izraza pri konverziji:

- automatska konverzija tipova pri izvršenju aritmetičkih operacija

```
console.log(8 * null); // prikazade 0
console.log("5" - 1); // prikazade 4
console.log("5" + 1); // prikazade 51
console.log("pet" * 2); // prikazade NaN
```

- automatska konverzija tipova pri izvršenju operacija poređenja

```
console.log(false == 0); // prikazade true
```

- poređenje jednakosti za vrednosti null i/ili undefined se realizuje na pomalo specifičan način

```
console.log(null == undefined); // prikazade true , medjutim poredjenje sa === daje false
console.log(null == 0); // prikazade false
```

- logički operatori se "skraćeno" izvršavaju

```
console.log(undefined || "Karlo"); // prikazade Karlo
console.log("Karlo" || "Korisnik"); // prikazade Karlo
```

12. Eksplicitna konverzija tipa. Primeri.

Kada treba promeniti tip podataka neke promenljive to se zove eksplicitna konverzija tipova i vrši se koristeći neku od ugrađenih funkcija za konverziju:

- Konvertovanje u nisku: `String(...)` ili `toString()`
- Konvertovanje u broj: `Number(...)` ili `parseFloat()` ili `parseInt()`
- Konvertovanje u logičku vrednost: `Boolean(...)`

Dakle, tokom pisanja koda naglasimo da želimo da promenimo tip neke promenljive.

```
result = Number('324');      console.log(result); // 324
result = Number(true);      console.log(result); // 1
result = Number(null);      console.log(result); // 0
result = Number(' ');       console.log(result); // 0
result = parseInt('20.01');  console.log(result); // 20
result = parseFloat('20.01'); console.log(result); // 20.01
```

```
result = String(2 + 4);      console.log(result); // "6"
result = String(undefined);  console.log(result); // "undefined"
result = (324).toString();   console.log(result); // "324"
result = true.toString();    console.log(result); // "true"
```

```
result = Boolean("");        console.log(result); // false
result = Boolean(0);          console.log(result); // false / undefined null NaN / ostalo true
```

13. Implicitna konverzija tipa. Primeri.

Implicitna konverzija tipova se odnosi na konverzije koje nisu eksplicitne, a izvršava ih JavaScript izvršno okruženje u hodu kao sporedni efekat nekih drugih radnji. Implicitna konverzija se najčešće javlja kada se vrednost nekog tipa koristi na način koji automatski prouzrokuje njenu konverziju.

Najčešće su razlozi za implicitnu konverziju poređenje ili neka druga matematička operacija između različitih tipova promenljivih. Pored ovoga implicitna konverzija se dešava i usled potrebe za izračunavanjem nekog uslova.

Jedan od najpoznatijih primera implicitne konverzije je “poređenje prema jednakosti”, nakon čega se dobija logička vrednost. Algoritam za određivanje logičke vrednosti koja predstavlja rezultat poređenja tj. $x == y$ je slededi:

- 1) Ako su x i y istog tipa, tada se izvršava striktno poređenje $===$
- 2) Ako je promenjiva x null ili undefined, tada se vrada true
- 3) Ako je promenjiva x undefined a promenjiva y null, tada se vrada true
- 4) Ako je promenjiva x Number a promenjiva y String, vrada rezultat poređenja $x === \text{ToNumber}(y)$
- 5) Ako je promenjiva x String a promenjiva y Number, vrada rezultat poređenja $\text{ToNumber}(x) === y$
- 6) Ako je promenjiva x Boolean a promenjiva y Number, vrada rezultat poređenja $\text{ToNumber}(x) === y$
- 7) Ako je promenjiva x Number a promenjiva y Boolean, vrada rezultat poređenja $x === \text{ToNumber}(y)$
- 8) Ako je promenjiva x Number ili String ili Symbol a promenjiva y Object, onda vrada rezultat poređenja $x == \text{ToPrimitive}(y)$
- 9) Ako je promenjiva x Object a promenjiva y Number ili String ili Symbol, onda vrada rezultat poređenja $\text{ToPrimitive}(x) == y$
- 10) Ako nije ništa od prethodno navedenog, tada se vrada False

Tabela implicitnih konverzija:

Originalna vrednost	Konverzija u Number	Konverzija u String	Konverzija u Boolean
false	0	"false"	false
true	1	"true"	true
0	0	"0"	false
1	1	"1"	true
"0"	0	"0"	true
"000"	0	"000"	true
"1"	1	"1"	true
NaN	NaN	"NaN"	false
Infinity	Infinity	"Infinity"	true
-Infinity	-Infinity	"-Infinity"	true
""	0	""	false
"20"	20	"20"	true
"twenty"	NaN	"twenty"	true
[]	0	""	true
[20]	20	"20"	true
[10,20]	NaN	"10,20"	true
["twenty"]	NaN	"twenty"	true
["ten","twenty"]	NaN	"ten,twenty"	true
function(){}	NaN	"function(){}"	true
{}	NaN	"[object Object]"	true
null	0	"null"	false
undefined	NaN	"undefined"	false

14. Naredbe dodele vrednosti. Primeri.

Svi prethodno opisani izrazi ved sami po sebi predstavljaju naredbe programskog jezika JavaScript. Naravno, programi napisani na ovakav način bi imali malo smisla.

Izvršavanje JavaScript programa tj. skripte počinje tako što se izvrši prva naredba u skripti, zatim druga, i tako redom... Sekvenijalno izvršavanje naredbi se može izmeniti korišćenjem naredbi grananja i ciklusa, kao što je to slučaj i u programskom jeziku C.

Za razliku od programskog jezika C, gde se naredba obavezno morala završiti sa znakom tačka-zarez (eng. semicolon tj. ;), JavaScript naredba se može završiti ili znakom tačka-zarez ili znakom za kraj reda. Drugim rečima, kod jezika JavaScript u novom redu počinje nova naredba, osim ako se tekudi red na završava obratnom kosom crtom (engl. backslash tj. \) ili ako se ne radi o šablonu za niske koji se prostire kroz više redova.

→ Naredbe dodele:

U ovoj naredbi figuriše operator dodele =. Naredba dodele je istovremeno i izraz, sa bočnim efektom. Evaluacija naredbe dodele se vrši tako što se izračunava vrednost izraza koji se nalazi desno od znaka =, ta izračunata vrednost se dodeljuje levom izrazu (to može biti promenljiva, osobina objekta, element niza i sl.) koji se nalazi levo od znaka = i konačan rezultat evaluacije je izračunata vrednost izraza na desnoj strani.

15. Kombinovane naredbe dodele.

Kombinovane naredbe dodele predstavljaju kombinaciju operacije i naredbe dodele.

Operatori kod kombinovanih naredbi dodele su: +=, -=, *=, /=, %=, <<=, >>=, >>>=, &=, ^=, /= i **=. (>>>= je bitovsko neoznaceno siftovanje udesno, ostalo isto kao i u C-u).

16. Naredbe inkrementiranja i dekrementiranja.

Naredbe inkrementiranja i dekrementiranja sačinjavaju najkorisnije kombinovane naredbe dodele.

Inkrementiranje => operator uvećanja za jedan ++

Dekrementiranje => operator umanjenja za jedan --

Ovi operatori mogu biti prefiksni ili postfiksni. Ako je operator inkrementiranja/dekrementiranja prefiksni, onda se prilikom evaluacije izraza u kojem učestvuje ovaj operator, inkrementiranje/dekrementiranje (operacija) izvršava na početku, a ako je operator postfiksni onda se inkrementiranje/dekrementiranje izvršava na kraju evaluiranja izraza.

17. Pozivi predefinisanih funkcija.

U JavaScript-u izrazi u naredbama mogu sadržati pozive funkcija. Neke od često pozivanih funkcija su predefinisane, tj. sadržane u okviru JavaScript jezika i odmah su na raspolaganju programeru.

U sklopu JavaScripta postoje ugrađeni objekti sa svojim svojstvima i metodama. Programer može odmah da koristi te objekte, njihove osobine i metode – tj. predefinisane objekte i predefinisane funkcije.

JavaScript sadrži sledeće predefinisane objekte:

- Array
- Math
- Date
- Function

Pored ovih, postoje i objekti cija su imena ista kao i imena prostih primitiva (primitivnih tipova), samo sa velikim pocetnim slovom:

- String
- Number

Ovi omotaci su uvedeni kako bi se JavaScript programeru olaksao rad sa primitivnim podacima.

Objekat Math

Sadrzi osobine cijim se referisanjem dobijaju neke od poznatih matematickih konstanti:

<i>Osobina objekta</i>	<i>Opis</i>
E	Vraca Ojlerovu konstantu (priblizno 2.718)
LN2	Vraca log od 2 za osnovu e (priblizno 0.693)
LN10	Vraca log od 10 za osnovu e (priblizno 2.302)
LOG2E	Vraca log od e za osnovu 2 (priblizno 1.442)
LOG10E	Vraca log od 10 za osnovu e (priblizno 0.434)
PI	Vraca PI (priblizno 3.14)
SQRT1_2	Vraca kvadratni koren od $\frac{1}{2}$ (priblizno 0.707)
SQRT2	Vraca kvadratni koren od 2 (priblizno 1,414)

Metode Math objekta uzimaju broj/brojeve kao argument a vracaju rezultat matematicke funkcije izvršene nad tim brojem/brojevima:

<i>Metoda objekta</i>	<i>Opis</i>
abs(x)	Vraca apsolutnu vrednost x
acos(x)	Vraca arkus kosinus x, u radijanima
asin(x)	Vraca arkus sinus x, u radijanima
atan(x)	Vraca arkus tangens x kao broj izmedju $-\pi/2$ i $\pi/2$ radijana
atan2(y,x)	Vraca arkus tangens kolicnika argumenata, tj. y/x
ceil(x)	Vraca x, zaokruzen na veci ceo broj
cos(x)	Vraca kosinus x, x u radijanima
exp(x)	Vraca vrednost eksponencijalne funkcije, tj. e na x
floor(x)	Vraca x, zaokruzen na manji ceo broj
log(x)	Vraca logaritam od x za osnovu e
max(x,y,z,...,n)	Vraca maksimalnu vrednost argumenata
min(x,y,z,...,n)	Vraca minimalnu vrednost argumenata
pow(x,y)	Vraca vrednost stepene funkcije, tj. x na y
random()	Vraca slucajan broj izmedju 0 i 1
round(x)	Vraca x zaokruzen na najblizi ceo broj
sin(x)	Vraca sinus x, x je u radijanima
sqrt(x)	Vraca kvadratni koren x
tan(x)	Vraca tangens x, x je u radijanima

Primer: let miki = -5; console.log(Math.min(2, 4, 7, 6, miki, Math.PI));

Objekat Console

Ovaj objekat obezbedjuje pristup bilo konzoli za debugiranje kod pregledaca, bilo konzoli za prikaz kod node.js. Konkretni nacin rada konzole se razlikuje od jedne do druge platforme, ali uvek postoji skup funkcionalnosti koje su obezbedjene.

Objektu Console se moze pristupiti iz bilo kog globalnog objekta (kod pregledaca je izložena kao Window.console i na nju se moze jednostavno referisati sa console).

```

console.log("vrednost poromenljive x je", x);
// poziva se funkcija console.log, uz spajanje niski
console.log("vrednost poromenljive x je " + x);
// poziva se funkcija console.log, uz sablon za nisku
console.log(`vrednost poromenljive x je ${x}`);

```

18. Grananja. Naredba if.

Naredbama grananja se donosi odluka kojom ce se od vise alternativnih tokova nastaviti izvršavanje. Odluka se donosi na osnovu vrednosti datog izraza.

Naredba if

Isto kao i u C-u.

```

let br = 10 * Math.random() - 5;
console.log(`Псеудп-случајни брпј има вреднпсу $,br-`);

if (br <= 0)
    console.log(`Брпј је негауиван`);
else if (br == 0)
    console.log(`Брпј је уачнп 0`);
else if (br < 2)
    console.log(`Брпј је ппзиуиван, маои пд 2`);
else
    console.log(`Брпј је већи или једнак пд 2`);

```

19. Grananja. Naredba switch.

Naredba višestrukog grananja switch ima istu strukturu i način izvršavanja kao u C-u.

```

let godisnjeDoba = Math.floor( (Math.random() * 40) % 5 );

switch (godisnjeDoba) {
    case 0:
        console.log("Ponesite kišobran.");
        break;
    case 1:
        console.log("Ponesite naočari za sunce.");
    case 2:
        console.log(`Ponesite rukavice.`);
    case 3:
        console.log("Obucite patike za šetnju.");
        break;
    default:
        console.log("Ovo je neko nemogude doba...");
        break;
}
console.log(`Vrednost indikatora je ${godisnjeDoba}`);

```

20. Ciklusi. Opsezi važenja promenljivih u ciklusima.

Cesto se javlja potreba da se unutar programa neke naredbe izvršavaju više puta. Za to se koriste ciklusi, koji mogu biti *brojacki* ili *kolekcijski*. Brojacki ciklusi se javljaju u manje-vise istom obliku u skoro svim programskim jezicima, a kolekcijski se oslanjaju na prolazak, tj. iteriranje kroz kolekciju (npr. kroz elemente niza ili kroz osobine objekata).

Opsezi vazenja promenljivih u ciklusima

Ako je promenljiva deklarirana koriscenjem **let** opseg vazenja promenljive je **blokovski**. Ovo vazi i za brojacke promenljive koje se deklariraju u inicijalnoj sekciji for ciklusa (npr. `let i=0; i<n; i++...`). tako deklariranim promenljivama se moze pristupiti samo u kodu unutar ciklusa (izvršavanje naredbi van ciklusa kojima se pokusava pristupiti takvim promenljivama dovodi do greske).

S druge strane, ako je promenljiva u telu ciklusa ili u inicijalizacionoj sekciji for ciklusa deklarirana koriscenjem naredbe **var** onda joj se moze pristupati i u naredbama koje se nalaze van ciklusa.

21. Ciklusi. Ciklus while.

Naredba ciklusa sa preduslovom `while` ima istu strukturu i nacin izvršavanja kao u programskom jeziku C.

```
let res = 1;
let brojac = 0;
while (brojac < 10) {
    res *= 2;
    brojac++;
}
console.log(res);
```

22. Ciklusi. Ciklus do - while.

Isto kao C.

Primer generise slucajan broj sve dok generisani broj nije nenegativan:

```
let pokusaja = 0;
do {
    var slucajan = Math.random() - 0.1;
    pokusaja = pokusaja + 1;
} while (slucajan >= 0);
console.log(`Извучен је негати́ван случа́јан број $,slucajan- из \ ппкүшаја бр. $,pokusaja-`);
```

23. Ciklusi. Ciklus for.

Isto kao C. Pored brojackog for ciklusa, jezik JavaScript zadrzi i kolekcijske cikluse **for – in** (iterira kroz indekse kojima se pristupa elementima kolekcije) i **for – of** (iterira kroz same elemente kolekcije).

Primer: Skripta koja, korišćenjem naredbe `for` (ciklus u ciklusu) 20 puta ponavlja seriju izvlačenja pseudoslučajnih brojeva dok izvučeni broj ne bude između 0 i 0.1, prikazuje izvučeni pseudoslučajan broj, broj elementata u seriji, kao i prosečnu dužinu serije:

```
const n = 20;
```

```

let prosecnoPokusaja = 0;
for (let i = 0; i < n; i++) {
    var slucajan = Math.random();
    for (var pokusaja = 1; slucajan >= 0.1; pokusaja++) {
        slucajan = Math.random();
    }
    console.log(`Извучен је случајан број $,slucajan- маои пд 0.1 из \ ппкyшaja бр. $,pokusaja-`);
    prosecnoPokusaja += pokusaja
}
console.log(`Прпсечан број ппкyшaja је $,prosecnoPokusaja/n`);

```

24. Ciklusi. Iskakanje iz ciklusa i preskakanje iteracije

Iskakanje iz ciklusa

Iskakanje iz ciklusa znači da u jeziku JavaScript postoji mehanizam koji dopušta da se izvršenjem naredbe u telu ciklusa, ne konsultujući uslov za ciklus, izvrši iskakanje i prelazak na slededu naredbu van tog ciklusa, ili na slededu naredbu van obeleženog ciklusa koji obuhvata taj ciklus.

Isto kao i u C-u to se postize naredbom **break**, koja može biti bez obeležja ili sa obelezjem.

Primer: beskonacni for ciklus sa iskakanjem (bez obelezja):

```

let res = 2;
for (let brojac = 1; true; brojac++) {
    if (brojac % 10 == 0)
        break;
    res *= 2;
}
console.log(res);

```

Preskakanje iteracije ciklusa

Preskakanje iteracije ciklusa znači da u jeziku JavaScript postoji mehanizam koji dopušta da se izvršenjem naredbe u telu ciklusa izvrši preskakanje preostalih naredbi u telu ciklusa i prelazak na slededu iteraciju ciklusa, ili prelazak na slededu iteraciju obeleženog ciklusa koji obuhvata taj ciklus.

Isto kao i u C-u to se postize naredbom **continue**, koja može biti sa ili bez obelezaja.

Primer: Skripta koja, korišćenjem neobeleženog preskakanje iteracije, prebrojava koliko se puta slučajan broj našao između 0 i 0.1 (što predstavlja uspeh eksperimenta). Pri tome, eksperimenti se ponavljaju po 100 puta u seriji, broj serija je 20, a bez obzira na neuspeh/uspeh eksperimenta, eksperimenti de biti nastavljani u tekudoj seriji:

```

const brojSerija = 20
const brojPonavljanjaUSeriji = 100;
let uspesnihPokusaja = 0;
centralna:
for (let i = 0; i < brojSerija; i++) {
    for(let j=0; j<brojPonavljanjaUSeriji; j++){
        let slucajan = Math.random();
        if( slucajan >= 0.1)
            continue;
        uspesnihPokusaja++;
    }
}

```

```

    }
}
console.log(`Укупан број серија је $,brojSerija-`);
console.log(`Број понављања у серији је $,brojPonavljanjaUSeriji-`);
console.log(`Број успешних покушаја је ${uspesnihPokusaja}`);

```

Obeleženo iskakanje iz ciklusa i obeleženo preskakanje iteracije

Isto kao u programskom jeziku C i Javi, postoji mogućnost da se određenom ciklusu dodeli **obeležje** (engl. label), pa da se potom iskakanje ili preskakanje iteracije odnose na taj obeležni ciklus, a ne na najunutrašnjiji blok.

Primer za break: Skripta koja, korišćenjem obeleženog iskakanja iz ciklusa, prebrojava koliko je ukupno bilo pokušaja pre uspeha, tj. pre nego što se slučajan broj našao između 0 i 0.1 u 20 serija ponavljanja eksperimenta. Pri tome, u svakoj seriji eksperiment se ponavljao sve dok ne dođe do uspeha, koji dovodi do prekida svog daljeg rada:

```

const brojSerija = 20;
let ukupnoPokusaja = 0;
centralna:
for (let i = 0; i < brojSerija; i++) {
    for (; ;) {
        let slucajan = Math.random();
        ukupnoPokusaja++;
        if (slucajan < 0.1)
            break centralna;
    }
}
console.log(`Укупан број покушаја је $,ukupnoPokusaja-`);

```

Primer za continue: Skripta koja, korišćenjem obeleženog preskakanja iteracija, prebrojava koliko je ukupno bilo pokušaja pre uspeha, tj. pre nego što se slučajan broj našao između 0 i 0.1 u 20 serija ponavljanja eksperimenta. Pri tome, u svakoj seriji eksperiment se ponavljao sve dok ne dođe do uspeha, koji dovodi do prekida tekude i početka nove serije:

```

const brojSerija = 20;
let ukupnoPokusaja = 0;
centralna:
for (let i = 0; i < brojSerija; i++) {
    for (let j = 0; j < 100; j++) {
        let slucajan = Math.random();
        ukupnoPokusaja++;
        if (slucajan < 0.1)
            continue centralna;
    }
}
console.log(`Укупан број покушаја је $,ukupnoPokusaja-`);

```

7. Funkcije i zatvorenja

Kao i kod drugih viših programskih jezika, **funkcije** služe za bolju organizaciju programskog koda – sekvence naredbi koje predstavljaju logičku celinu se organizuju u funkcije.

U jeziku JavaScript funkcije karakteriše činjenica da predstavljaju tzv. “građane prvog reda” (engl. first class citizen). Ako programski entitet (objekat ili element) jeste “građanin prvog reda”, to onda znači da takav entitet podržava sve operacije dostupne drugim tipovima i može da se ponaša potpuno isto kao bilo koja druga vrsta entiteta koja je građanin prvog reda. Konkretno kod JavaScripta, pošto su funkcije građani prvog reda, kod njih ne postoji restrikcija (ograničenje) kako se kreiraju ili koriste i oni imaju osobine svih primitiva (osnovnih tipova) kao i sve osobine objekta.

Dakle, funkcija u jeziku JavaScript je „građanin prvog reda” što znači da ima sve osobine drugih primitiva, pa se može proslediti kao argument, može se vratiti kao rezultat neke druge funkcije ili se može dodeliti nekoj promenljivoj. Pored nabrojanih osobina, funkcije imaju i sve osobine objekata. Pored svih osobina objekata i primitiva, funkcije sadrže i dodatnu semantiku za pozivanje.

Kao što smo već videli, JavaScript se isporučuje sa skupom “ugrađenih” tj. predefinisanih funkcija koje su odmah na raspolaganju programeru. Naravno, jezik dopušta JavaScript programeru i da sam kreira sopstvene funkcije.

1. Deklaracija i poziv funkcije. Primeri.

Jedan način rada sa funkcijama u jeziku JavaScript je da se funkcija kreira *deklarisanjem*. Sintaksa deklarisanja i poziva tako napravljene funkcije veoma podseda na deklarisanje i poziv funkcije u drugim “mejnstrim” programskim jezicima, kao što su Java i C. Deklaracija funkcije često sadrži **parametre**. Parametri funkcije su način da funkcija preuzme podatke iz spoljašnjosti. Parametri i promenljive deklarirane unutar funkcije su lokalni za tu funkciju, oni se ponovo kreiraju pri svakom pozivu funkcije i oni nisu vidljivi iz spoljašnjosti.

Poziv funkcije se realizuje slično kao u jezicima Java i C, navođenjem naziva funkcije iza koga, u zagradama, sledi lista argumenata međusobno razdvojenih zarezom. Izvršenje poziva dovodi do toga da izvršavanje se nastavlja od prve naredbe funkcije, pri čemu su parametri funkcije zamenjeni prosleđenim argumentima. Po završetku izvršavanja pozvane funkcije, izvršavanje programa se nastavlja od naredbe koje sledi iza naredbe poziva.

Funkcije se definišu ključnom rečju **function**, a za vraćanje izračunate vrednosti (rezultata) funkcije pozivaocu, koristi se naredba **return**. Izvršenjem ove naredbe se završava rad funkcije i izvršavanje vrata u pozivajuću funkciju, na naredbu koja sledi iza naredbe poziva. Izvršavanje funkcije se takođe završava završetkom izvršenja poslednje naredbe funkcije.

Funkcije mogu da budu i **anonimne** (kada im se ne navede ime već samo ključna reč function, lista parametara i telo).

Primer. Deklaracija i pozivi funkcije za kvadriranje broja:

```
function kvadrat(x) {  
    return x * x;  
};  
console.log(`Квадрат броја 12 је ${kvadrat(12)}`);  
let y = kvadrat(13);  
console.log(`Квадрат броја 13 је ${y}`);  
y = 14;  
console.log(`Квадрат броја ${y} је ${kvadrat(y)}`);
```

2. Funkcijski izraz i poziv funkcije. Primeri.

Jedan način rada sa funkcijama u jeziku JavaScript je da se funkcija odredi **funkcijskim izrazom**. U tom načinu rada sa funkcijom, funkcija zaista postaje "građanin prvog reda". (I zbog toga demo, u primerima koji slede, obično koristiti funkcijske izraze)

Kod funkcijskih izraza se poziv funkcije realizuje navođenjem naziva funkcije iza koga, u zagradama, sledi lista argumenata međusobno razdvojenih zarezom.

Primer. Funkcijski izraz za kvadriranje broja i pozivi tako napravljene funkcije:

```
let kvadrat = function(x) {  
    return x * x;  
};  
console.log(kvadrat(12));  
console.log(`Квадрат броја 12 је ${kvadrat(12)}`);  
let y = kvadrat(13);  
console.log(`Квадрат броја 13 је ${y}`);  
y = 14;  
console.log(`Квадрат броја ${y} је ${kvadrat(y)}`);
```

U prethodnom primeru kvadrat je promenljiva koja referiše na funkciju - grubo rečeno, sadrži adresu "nečega" što za prosleđeni argument vrada proizvod tog argumenta sa samim sobom.

Prilikom poziva funkcije, vrši se zamena tj. supstitucija parametara funkcije argumentima poziva. U jeziku JavaScript, vrši se tzv. **supstitucija po vrednosti**, izračunavaju se vrednosti argumenata prilikom poziva i izračunate vrednosti redom zamenjuju parametre funkcije.

3. Parametri i argumenti funkcija. Primeri.

Za razliku od najvedeg broja popularnih programskih jezika (kao što su Java i C), gde prilikom poziva parametri funkcije i argumenti poziva moraju biti saglasni po broju i tipu, to kod JavaScripta uopšte nije slučaj. Naravno, jasno je da se zbog slabe tipiziranosti jezika, JavaScript ne može proveravati saglasnost tipova parametara i argumenata, ali takođe nije obavezno ni da se poklopi broj argumenata u pozivu funkcije sa brojem parametara funkcije.

Ako je broj argumenata poziva veći od broja parametara funkcije, tada se izvrši supstitucija onoliko parametara koliko ih ima u definiciji funkcije, a preostali tj. višak jednostavno bude ignorisan.

Primer. Poziv funkcije, gde je broj argumenata u pozivu veći od broja parametara funkcije:

```
let buka = function() {  
    console.log(" Tras !");  
};  
buka();  
buka();  
// broj argumenata može biti veći  
// od broja parametara funkcije  
buka("Petar");           // Tras ! Tras ! Tras !
```


U slučaju poziva funkcije kada je broj argumenata u pozivu manji od broja parametara funkcije, parametri funkcije će dobiti vrednost **undefined**. (U prethodnim poglavljima je opisano kako se evaluira (proverava) izraz koji ima undefined operande).

Primer. Poziv funkcije, gde je broj argumenata u pozivu manji od broja parametara funkcije:

```
const stepen = function(osnova, izlozilac) {
  let ret = 1;
  for (let i = 0; i < izlozilac; i++)
    ret *= osnova;
  return ret;
};
console.log(stepen(3, 4)); // 81
console.log(stepen(4, 3)); // 64
console.log(stepen(4));    // 1
console.log(stepen());     // 1
```

Ako je potrebno, može se odlučiti da se u samom telu funkcije proverava da li je izvršena supstitucija parametra sa argumentom ili ne.

Takođe, treba istadi i da prilikom svakog poziva funkcije, postoji mogućnost da se pristupi argumentima poziva (bilo da ih je više, manje ili tačno koliko treba) korišćenjem **arguments** (o čemu će biti reči u kasnijim poglavljima).

4. Opcioni parametri funkcija. Primeri.

Elegantniji način za proveru da li je izvršena supstitucija parametra sa argumentom jeste korišćenje takozvanih **opcionih parametara funkcije**. Parametar postaje opcioni ako se u definiciji funkcije, prilikom navođenja parametra specificira (definiše, odredi) njegova podrazumevana vrednost. U tom slučaju, ako prilikom poziva funkcije za opcioni parametar ne bude dat argument, parametar će dobiti podrazumevanu vrednost koja mu je data u definiciji funkcije.

Primer. Opcioni parametri funkcije:

```
const stepen = function (osnova = 10, izlozilac = 2) {
  let ret = 1;
  for (let i = 0; i < izlozilac; i++)
    ret *= osnova;
  return ret;
};
console.log(stepen(3, 4)); // 81
console.log(stepen(4, 3)); // 64
console.log(stepen(4));    // 16
console.log(stepen());     // 100
```

Primer. Provera u telu funkcije da li je pravilno izvršena supstitucija parametara argumentima:

```
let stepen = function (osnova, izlozilac) {
  if (osnova == undefined)
    osnova = 10;
  if (izlozilac == undefined)
    izlozilac = 2;
  let ret = 1;
```

```

    for (let i = 0; i < izlozilac; i++)
        ret *= osnova;
    return ret;
};
console.log(stepen(3, 4)); // 81
console.log(stepen(4, 3)); // 64
console.log(stepen(4));    // 16
console.log(stepen());    // 100

```

Primer. Ilustruje šta se dešava kada je u pozivu funkcije više argumenata, kao i kada je manje argumenata i kako se ponašanje razlikuje u zavisnosti od toga da li su u definiciji funkcije dati podrazumevani parametri:

```

function bezArgumenata() {
}
// Ovo je OK
bezArgumenata(1, 2, 3);

```

```

function triArgumenta(a, b, c) {
    console.log("---\n" + a)
    console.log(b)
    console.log(c)
}
// I ovo je OK
triArgumenta(1, 2, "tri");
triArgumenta(1, 2); // 1 2 undefined
triArgumenta(1);    // 1 undefined undefined
triArgumenta();     // undefined undefined undefined

```

```

function triArgumentaPodrazumenvano(a = 'a', b = 'b', c = null) {
    console.log("---\n" + a)
    console.log(b)
    console.log(c)
}
triArgumentaPodrazumenvano(1, 2, "tri");
triArgumentaPodrazumenvano(1, 2); // 1 2 null
triArgumentaPodrazumenvano(1);    // 1 b null
triArgumentaPodrazumenvano();     // a b null

```

5. Opsezi važenja za promenljive i funkcije. Primeri.

Ako je promenljiva deklarisan korišćenjem **let** ili **const** opseg važenja promenljive je *blokovski*. Ako je promenljiva deklarisan korišćenjem **var** opseg važenja promenljive je *funkcijski*, što znači da se može pristupiti promenljivoj u opsegu funkcije u kojoj je ta promenljiva definisana.

Naravno, u telu funkcije se može pristupiti svim parametrima funkcije i svim promenljivima koje su globalne za tu funkciju, tj. definisane u opsezima koji sadrže definiciju date funkcije.

Primer. Ilustruje kako se modifikuje promenljiva koja je globalna za datu funkciju:

```

let test = "globalna vrednost";

```

```
function testirajOpsegDefinisanosti() {
    test = "lokalna vrednost";
    console.log(test);
}
console.log(test);
testirajOpsegDefinisanosti();
console.log(test);
Ispis:
globalna vrednost
lokalna vrednost
lokalna vrednost
```

Važno je istadi da funkcije deklarisanе unutar opsega neke druge funkcije mogu pristupati lokalnom opsegu funkcije unutar koje su deklarisanе.

6. Stek poziva za funkcije. Primeri.

Funkcija u svom telu može sadržati jedan ili više poziva drugih funkcija, ili čak može da poziva samu sebe. Poziv funkcije i povratak iz funkcije se postiže korišćenjem strukture podataka **stek**. JavaScript izvršno okruženje prilikom svakog poziva funkcije na stek smešta tzv. stek-okvir (engl. stack frame) za funkciju, koji sadrži adresu povratka, tj. opisuje od koje de se lokacije u JavaScriptu nastaviti izvršavanje. Prilikom završetka rada funkcije, sa steka se skida prethodno postavljeni stek-okvir i JavaScript izvršno okruženje nastavlja izvršavanje od lokacije koja je čuvana u tom stek-okviru.

Dakle, možemo smatrati da u JavaScriptu pozivi funkcija tokom izvršenja programskog koda obrazuju stek, koji se naziva **stek poziva za funkcije**.

Primer. Ilustracija kako nepravilnim korišćenjem može biti prepunjen stek poziva za funkcije - program treba da odredi da li je starija kokoška ili jaje :) :

```
function kokoska() {
    return jaje();
};
function jaje() {
    return kokoska();
}
console.log("Starija je ", kokoska());
```

7. Rekurzivne funkcije. Primeri.

Rekurzija označava postupak ili funkciju koji u svojoj definiciji koriste sami sebe. Drugim rečima, rekurzivni postupak za rešavanje problema zahteva podelu originalnog problema na delove (manje dimenzije ali iste prirode), koji bivaju nezavisno podvrgnuti istom tom postupku, sve dok dimenzija problema ne postane toliko mala da je rešenje problema trivijalno.

Primer. Ilustracija rekurzije - rekurzivna funkcija za izračunavanje stepena osnove pozitivnim celobrojnim izložiocem.

```
function stepen(osnova, izlozilac) {
    if (izlozilac == 0)
        return 1;
    return osnova * stepen(osnova, izlozilac - 1);
}
console.log(stepen(3,4))
```

Primer. Ilustracija rekurzije - proverava se da je dati broj paran ili neparan. (Skripta se oslanja na činjenicu da dati broj ima istu parnost kao taj broj umanjen za 2)

```
function paran(broj) {
    if (broj == 0)
        return true;
    if (broj == 1)
        return false;
    return paran(broj - 2);
}
function neparan(broj) {
    return !paran(broj);
}
console.log(paran(75));
console.log(neparan(75));
```

Primer. Ilustracija uzajamne rekurzije - proverava se da li je dati broj paran ili neparan.

(U ovom primeru funkcija koja proverava da li je broj paran poziva funkciju koja proverava da li je neparan dekrement (tj. taj broj umanjen za jedan), onda opet poziva funkciju koja proverava da li je paran dekrement dekrementa itd, sve dok se broj koji se proverava ne svede na trivijalan slučaj - 1 ili 0. Onda se ide unazad kroz stek poziva dok se ne dođe do rezultata):

```
function paran(broj) {
    if (broj == 0)
        return true;
    if (broj == 1)
        return false;
    return neparan(broj - 1);
}
function neparan(broj) {
    if (broj == 0)
        return false;
    if (broj == 1)
        return true;
    return paran(broj-1);
}
console.log(paran(6));
console.log(neparan(50));
```

Primer. Rekurzivna funkcija koja proverava da li se ciljni (konačni) broj može napraviti od broja 1, uzastopnim ponavljanjem množenja sa 3 i/ili dodavanja 5 u bilo kom redosledu.

(Kroz programski kod ovog primera koristi se definisanje funkcijskog izraza unutar funkcijskog izraza, pristup promenljivoj globalnoj za funkciju, kao i rekurzija)

```
const izgradilzraz = function (cilj) {  
  const izgradi = function (start, istorija) {  
    if (start == cilj)  
      return istorija;  
    if (start > cilj)  
      return null;  
    return izgradi(start + 5, "(" + istorija + " + 5) ") || izgradi(start * 3, "(" + istorija + " * 3) ");  
  }  
  return izgradi(1, "1");  
}  
for (let i = 80; i <= 100; i++)  
  console.log(i + " = " + izgradilzraz(i));
```

```
83 = ((((((1 + 5) + 5) + 5) + 5) + 5) * 3) + 5)  
84 = (((((1 + 5) * 3) + 5) + 5) * 3)  
85 = null  
86 = ((((((((((((((1 + 5) + 5) + 5) + 5) + 5) + 5) + 5) + 5) + 5) + 5) + 5) + 5) + 5) + 5) + 5)  
87 = ((((((1 * 3) + 5) * 3) + 5) * 3)  
88 = (((((((1 + 5) + 5) + 5) + 5) + 5) * 3) + 5) + 5)  
89 = ((((((1 + 5) * 3) + 5) + 5) * 3) + 5)  
90 = null
```

8. Lambda izrazi i funkcije. Primeri.

U poslednje vreme popularan je nacin definisanja funkcija pomocu lambda izraza. Umesto rezervisane reci *function* koristi sekvence znakova jednako i vece ("debela strelica", =>). Debela strelica => se pise posle liste parametara, a pre tela funkcije. Ideja je da se na taj nacin izrazi znacenje "ulaz sa datim parametrima proizvodi izlaz na nacin opisan telom funkcije".

Ako funkcija, tj. lambda izraz sadrzi tacno jedan parametar, onda nema potrebe da se taj jedan parametar obuhvata zagradama. Ako telo funkcije sadrzi tacnu jednu *return* naredbu, onda nema potrebe da se koriste viticaste zagrade za oznacavanje tela funkcije, niti da se koristi rezervisana rec *return* – dovoljno je samo napisati izraz koji vraca ta funkciju.

Prva funkcija prikazuje fiksiran tekst, a druga prikazuje sadrzaj koji bude prosledjen kao argument.

```
console.log('---');  
const sreca = () => console.log('Sto sam srecan!');  
for (let i = 0; i < 3; i++)  
  sreca();  
  
console.log('---');  
const poruka = x => console.log(`Poruka: '${x}'`);
```

```
for (let i = 0; i < 3; i++)  
    poruka('vazna poruka broj ' + i);
```

Rezultat rada:

```
--- Sto sam srecan!      Sto sam srecan!      Sto sam srecan!  
--- Poruka: 'vazna poruka broj 0'  Poruka: 'vazna poruka broj 1'  Poruka: 'vazna poruka broj 2'
```

Definisanje funkcije pomocu lambda izraza je uvedeno u JS 2015. godine, kako bi se omogucilo da se lakse i brze zapisuju mali funkcijski izrazi.

9. Zatvorenja za funkcije. Primeri.

U JS-u kod funkcijskih izraza, funkcije se mogu tretirati kao vrednosti, dok se lokalne veze ponovo kreiraju pri svakom pozivu funkcije. Sta se dogadja sa lokalnom vezom kada funkcija, koja je kreirala tu vezu, vise nije aktivna, tj. kad je zavrсила sa radom?

Primer. Funkcija kojom se odmotava data vrednost:

```
function omotajVrednost(n) {  
    let lokalnaPromenljiva = n;  
    return () => lokalnaPromenljiva;  
}  
let omotacZa1 = omotajVrednost(1); // funkcija  
let omotacZa2 = omotajVrednost(2);  
console.log(omotacZa1());           // 1  
console.log(omotacZa2());
```

Ovaj primer radi prema ocekivanjima – u stanju smo da pristupimo obema vrednostima koje su omotane u funkciju. Ovde vidimo da se lokalne veze kreiraju prilikom svakog poziva, i da razliciti pozivi mogu pristupati samo svojim vezama i da ne uticu na lokalne veze nastale usled drugih poziva. Nakon koriscenja naredbe *return*, prekida se izvršenje funkcije i postojanje njenih promenljivih u privremenoj memoriji. Medjutim, ovde se pamte promenjive iz spoljne funkcije *omotajVrednost()*, cak i ako je vratila neku vrednost sa *return*.

Ova karakteristika – mogucnost da se referise na konkretno lokalno vezivanje u okruzujucem opsegu, naziva se **zatvorenje** (closure). Funkcija koja referise na veze iz obuhvatajucih lokalanih okvira se naziva funkcija zatvorenja, ili krace zatvorenje. Drugim recima, zatvorenja su funkcije koje imaju pristup promenljivima koje se nalaze u opsegu definisanosti druge funkcije – sto se najcesce postize ugradjivanjem funkcije unutar druge funkcije, nakon cega zatvorenje dobija sposobnost da zapamti referencu na promenjive iz opsega definisanosti funkcije koja je obuhvata.

Promenjive definisane u obuhvatajucoj funkciji se ne brisu po izvršavanju same funkcije, vec se cuvaju u memoriji da bi bile dostupne funkciji zatvorenja. Tek nakon izvršenja funkcije zatvorenja, zatvara se i spoljna funkcija. Dok god se ne izvrše funkcije zatvorenja, JS ce cuvati i potrebne promenjive iz domena obuhvatajucih funkcija, stoga funkcije zatvorenja zauzimaju vise memorije nego obicne funkcije (preterano koriscenje dovodi do vece potrosnje memorije).

Kada se kaze da funkcija zatvorenja ima pristup promenljivoj, pod tim se misli da ima pristup odredjenom mestu u memoriji na koju ukazuje ta promenjiva, tj. njenoj referenci u trenutku povezivanja funkcije zatvorenja. Ukoliko se na tom mestu u memoriji, tj. referenci menja vrednost, zatvorenje ce uvek referisati na najnoviju trenutnu vrednost.

Ako se nakon definisanja zatvorenja, promenljivoj koju koristi zatvorenje dodeli neka druga referenca, zatvorenje ce da koristi referencu koja je bila aktuelna u trenutku zatvorenja.

```
let mojIme = 'Dragoljub';
const pozdrav = (ime) =>
  () => console.log('Zdravo, ' + ime + '!');

let pozdravSalmenom = pozdrav(mojIme);
mojIme = 'Marko';
pozdravSalmenom();
// Zdravo, Dragoljub!
```

```
-----
let uvecaj = () => {
  let brojac = 0;
  return () => brojac++;
}
const izbroj2 = uvecaj();
console.log(izbroj2());
console.log(izbroj2());
console.log(izbroj2());
// 0
// 1
// 2
```

Funkcije kao generatori funkcija:

```
function umnozilac(faktor) {
  return function(broj) { return broj * faktor; };
}
var dupliraj = umnozilac(2);
console.log(dupliraj(4.5));
console.log(dupliraj(5.5));

var utrostruci = umnozilac(3);
console.log(utrostruci(4.5));
console.log(utrostruci(5.5));

var pomnoziSa2_25 = umnozilac(2.25);
console.log(pomnoziSa2_25(4.5));
console.log(pomnoziSa2_25(5.5));
```

10. Dizanje promenljivih. Primeri.

Princip dizanja promenljivih i funkcija (hoisting) se odnosi na situaciju da deklarisanje promenljive i funkcije bilo gde u kodu ima isti efekat kao da je ta promenljiva deklarisana na pocetku programskog koda.

Za promenljive sa blokovskim opsegom definisanosti, tj. promenljivima definisanim pomocu *let* i *const*, ne vazi koncept dizanja promenljivih.

Primer. Ilustruje nemogućnost dizanja promenljive sa blokovskim opsegom definisanosti.

```
console.log(x);  
let x = 5;           // greska
```

Kod promenljivih koje su deklarisanе pomoću rezervisane reči *var*, situacija je drugačija. Tu se deklaracija promenljive desava pre izvršavanja bilo kog dela koda, pa deklarisanje promenljive bilo gde u kodu ima isti efekat kao da je ta promenljiva deklarisanа na početku koda. Prema tome, u slučaju kada se referise na neku takvu promenljivu u delu koda koji prethodi njenoj deklaraciji, tada neće biti izbacena greska, ali vrednost te promenljive neće biti definisana.

Primer. Ilustruje dizanje promenljive deklarisanе pomoću *var*

```
console.log(x);  
var x = 5;
```

Iako je promenljiva *x* definisana tek posle poziva funkcije za njen prikaz, deklaracija promenljive *x* je “dignuta”, tako da u trenutku poziva funkcije promenljiva postoji. Posto se dodela vrednosti promenljive realizuje na mestu gde je napisana u programskom kodu, posle poziva funkcije za prikaz, tako da ovde, prilikom poziva funkcije za prikaz, dignuta promenljiva još uvek nema vrednost i biće prikazano *undefined* na konzoli.

Ukoliko se uopšte ne deklarise promenljiva u kodu, promenljiva će biti deklarisanа “u letu”, u globalnom opsegu definisanosti (biće dostupna iz celog programskog koda). Međutim, ova radnja se izvodi tek u vreme izvršavanja koda, što nastupa nakon trenutka kada se izvršava “dizanje” promenljivih na vrh koda. Posto je prosao trenutak kada se vrsi dizanje, takva promenljiva neće biti dignuta na početak koda (ukoliko se referise na takvu promenljivu pre njene pojave u kodu, biće izbacena greska).

Primer. Ilustruje kako globalne promenljive neće biti dignute

```
console.log(x);  
x = 5;           //greska
```

11. Dizanje funkcija. Primeri.

Deklaracija funkcije se takodje odvija pre izvršenja koda, stoga se, isto kao kod promenljivih, može smatrati da su funkcije pisane na početku koda. Postupak dizanja deklarisanih funkcija (cela funkcija sa telom) na početak programskog koda naziva se *dizanje funkcije*.

Primer. Dizanje funkcije:

```
console.log("buducnost vraca:", buducnost());  
function buducnost() {  
    return "Još uvek ne postoje letedi automobili";  
};
```

Kao što vidimo u prethodnom primeru, funkciju možemo da deklariseмо na kraju koda i da je pozivamo sa početka, a da ne dodje do greske.

Treba naglasiti da se dizanje funkcija odvija **pre** dizanja promenljivih.

Primer.

```
test();           // iz deklarisanе  
var test = function () {
```



```

    console.log("prikaz iz funkcijskog izraza");
  }
  test();                // iz funkcijskog
  function test(){
    console.log("prikaz iz deklarisanе funkcije");
  }
  test();                // iz funkcijskog

```

Efekat izvršenja sledece skripte se bolje vidi ako se eksplicitno prikaze kako izgleda JS kod kada se izvrši dizanje promenljivih i funkcija:

```

// prvo se diže cela funkcija
function test(){
  console.log("iz deklarisanе funkcije");
}
// zatim se diže deklaracija promenljive
// promenljiva je "pregazila" deklarisanu funkciju
var test;
// dodela funkcijskog izraza promenljivoj
// je ostala na istom mestu u kodu
var test = function (){
  console.log("iz funkcijskog izraza ");
}

```

12. Funkcije i bočni efekti. Primeri.

Gruba podela funkcija je podela na dve klase: one koje se pozivaju **zbog svojih bocnih efekata** i one koje se pozivaju **zbog svojih povratnih vrednosti** (moguće je napraviti funkciju koja vraća vrednosti i sadrži bocni efekat).

Primer. Funkcije za prikaz na konzoli su dobar primer funkcija koje se koriste zbog bocnih efekata, a funkcije za stepenovanje su dobar primer funkcija koje se koriste zbog povratnih vrednosti.

Funkcije koje kreiraju vrednosti se lakše kombinuju od funkcija koje imaju bocne efekte.

Funkcija koja vraća vrednost, takva da ne samo što ona nema bocni efekat, već se prilikom svog rada ne oslanja na bocne efekte drugih funkcija koje poziva, naziva se *čista funkcija*. Čiste funkcije imaju veoma dobru osobinu da kad god budu pozvane sa istim argumentima, uvek produkuju isti rezultat. Poziv takve funkcije se može zameniti sa vrednošću koju ona vraća, a da istovremeno programski kod u kom ona učestvuje neće promeniti značenje.

Čiste funkcije se mnogo lakše testiraju – ako takva funkcija dobro radi u jednom kontekstu, ona će dobro raditi u svim kontekstima.

U JS – u nema potrebe da se po svaku cenu koriste čiste funkcije (nekada je neefikasno/nemoguće), pa tu korišćenje bocnih efekata ima svoj smisao.

```

const ukupnoEksperimenata = 10;
const ukupnoBacanjaUEksperimentu = 5000;
for (let eksp = 0; eksp < ukupnoEksperimenata; eksp++) {
  var brojPojavaBroja1 = 0;
  var brojPojavaBroja2 = 0;

```

```

var brojPojaveBroja3 = 0;
for (let i = 0; i < ukupnoBacanjaUEksperimentu; i++) {
    let kockaJePalaNa = Math.ceil(Math.random() * 3); //console.log(kockaJePalaNa);
    sracunajStatistikuBocniEfekat(kockaJePalaNa);
}
console.log(`Statistika eksperimenta broj ${eksp+1}, sa ${ukupnoBacanjaUEksperimentu} bacanja u
eksperimentu`);
console.log(brojPojaveBroja1, brojPojaveBroja2, brojPojaveBroja3);
}
function sracunajStatistikuBocniEfekat(rezultat) {
    switch (rezultat) {
        case 1: brojPojaveBroja1++; break;
        case 2: brojPojaveBroja2++; break;
        case 3: brojPojaveBroja3++; break;
        default: console.log('OVO NE BI SMELO DA SE DESI !!!');
    }
}

```

8. Objekti i nizovi

1. Objekti.

Objektno orijentisano programiranje je programska paradigma koja koristi apstrakciju za kreiranje modela zasnovanih na stvarnom svetu. Danas mnogi programski jezici podržavaju objektno orijentisano programiranje. Kod objektno orijentisanog programiranja program se može posmatrati kao kolekcija kooperativnih objekata, za razliku od tradicionalnog pogleda u kojem se program može posmatrati kao kolekcija funkcija, ili kao sekvenca instrukcija za računar. Svaki objekat se može posmatrati kao nezavisna celina. Prednosti ovog pristupa su veća fleksibilnost održavanja i proširivanja programa, jednostavniji je za razvoj i kod je citljiviji. JavaScript je jezik baziran na objektima.

Podela objekata u jeziku JavaScript:

- 1) Objekti koje definiše sam programer.
- 2) Objekti ugrađeni u JavaScript. Tu spadaju objekti koji omotavaju primitivne tipove podataka (String, Number, Boolean), objekti koji omogućavaju kreiranje korisnički definisanih objekata i složenih tipova (Object, Array) i objekti koji pojednostavljaju uobičajene zadatke, kao što su Date, Math.
- 3) Objekti definisani u okviru API pregledača veba. Ovi objekti nisu deo JavaScript jezika, ali ih podržava ogromna većina veb pregledača. Primeri takvih objekata su objekti Window, Navigator, Document, o kojima će biti reči kasnije.
- 4) Objekti koji su deo DOM stabla, definisani W3C standardom. Ovi objekti omogućavaju JavaScriptu manipulaciju nad CSS-om, i lakšu realizaciju dinamičkog HTML-a (DHTML).

Objekti su kompozitni tipovi podataka, objedinjuju više vrednosti. Drugim rečima, objekat je neuređen skup svojstava (osobina, atributa) od kojih svako ima ime i vrednost. Imenovane vrednosti mogu biti primitivnog tipa (brojevi, niske, logičke vrednosti) ili i same mogu biti objekti. Objekti se prave pomoću objektnog literala, koji se sastoji od spiska svojstava, gde se dvotačka (tj. znak :) postavlja između imena svojstva i njegove vrednosti. Svojstva u spisku se međusobno razdvajaju zarezima (znakom ,), a spisak se navodi unutar vitičastih zagrada (para znakova , -).

Primer.

```
let object = {  
    name1 : value1,  
    name2 : value2,  
    name3 : value3  
};
```

Operatori poređenja == i === se mogu primenjivati i na promenljive koje referišu na objekte. Operatori jednakosti i identičnosti, kada se primene na objekte, vraćaju true samo u slučaju kada se radi o referencama koje referišu na isti prostor u memoriji. Dakle, ako napravimo dva objekta sa potpuno istim svojstvima i njihovim vrednostima, poredjenje će vratiti false jer se upoređuju njihove reference, a ne sadržaj.

Primer.

```
let object1= {  
    value : 10  
};
```

```

let object3 = object1; // BITNO! Ne dodeljuje se sadržaj, nego referenca
                        // promenis jedno od ta dva -> menja se i drugo
let object2= {
    value : 10
};

console.log(object1 == object2); // ispisuje false
console.log(object1 == object3); // ispisuje true

```

2. Osobine objekata. Primeri.

Osobine (svojstva) objekta su elementi koji se nalaze u okviru objekta. Svaka osobina u okviru objekta ima svoje ime i vrednost. Svojstva objekta se ponašaju kao promenljive, moguće je upisivati vrednosti u njih i čitati vrednosti iz njih. Svojstva mogu da sadrže bilo koji tip podataka, uključujući i nizove, funkcije i druge objekte.

Vrednosti svojstva objekta se može pristupiti tako što se navede objekat, potom tačka i naziv svojstva (npr. student.ime). Alternativno, vrednosti svojstva datog objekta može da se pristupi i korišćenjem uglastih zagrada (npr. student*"ime"+).

Elementi null i undefined ne sadrže svojstva i pokušaj pristupa njihovim svojstvima dovodi do greške koja (ako ne bude uhvaćena) uslovljava završetak izvršavanja, dok pokušaj čitanja vrednosti nepostojedeg svojstva iz datog objekta ne prekida izvršavanje skripte, već daje vrednost undefined. Dakle, pre pristupa nekom objektu/svojstvu potrebno je proveriti da li je različit od null/undefined.

Svojstvo objekta može se obrisati pomoću operatora **delete**. Provera da li dati objekat poseduje neko svojstvo ili ne može da se ispita pomoću operatora **in**.

Primer.

```

let object= {
    prvi : 1,
    drugi : 2
};
console.log( "prvi" in objekat ); // true
delete objekat.prvi;              // brisemo svojstvo // objekat*„pr“+“vi“+ // mozemo i da dodajemo osobine
console.log( "prvi" in objekat ); // false

```

Operator **in** se javlja i u for-in ciklusu, koji omogućava da se iterativno pristupa svojstvima objekta, tj. da se svojstva nabrajaju.

Primer.

```

for (let osobina in obj)
    console.log(`${osobina} - ${obj[osobina]}`);

```

3. Metodi kod objekata. Primeri.

Metodi se mogu posmatrati kao osobine koji referišu na vrednosti-funkcije. Metodi kod objekata imaju dinamičku prirodu.

Primer.

```

let rabbit = {};

```

```

rabbit.name = "Dusko Dugousko";
rabbit.speak = function(tekst) {
    console.log("Zeka kaze: " + tekst);
};
console.log(rabbit.name);
rabbit.speak("Sefer, koji ti je vrug?");

```

Takodje, funkcija moze biti definisana van objekta i tada jedan metod moze da se pridruzi vecem broju objekata.

Primer.

```

let izgovara = function (tekst) {
    console.log(`$${this.tip} zec kaze '${tekst}'`);
}
let beliZec = { tip: "beli", govor: izgovara };
let debeliZec = { tip: "debeli", govor: izgovara };
beliZec.govor("Kasnim, kasnim, kraljica ce biti ljuta!");
debeliZec.govor("Al' sam gladan!");

```

Kljucna rec **this**, koju smo upotreбили u prethodnom primeru u funkciji izgovara, služi da se referise na objekat iz kog se ta funkcija poziva kao metod.

Za prolazak kroz sva svojstva i metode koristimo **for-in** petlju.

4. Nizovi.

Niz je uređen skup vrednosti. Te vrednosti se nazivaju elementi niza. Svaki element niza ima svoju numerički opisanu poziciju - indeks. Niz u JavaScriptu može da se posmatra kao objekat čija su svojstva pozicije u nizu tj. indeksi, a vrednost koja odgovara tom svojstvu je elemenat niza na datoj poziciji. S obzirom na to da je JavaScript slabo tipizirani jezik to (za razliku od jezika Java i C), ni nizovi nisu jako tipizirani pa se u niz mogu smestati vrednosti različitih tipova. Iz istog razloga, ne moraju svi elementi jednog niza biti istog tipa.

Najjednostavnije, niz se može deklarirati tako što se eksplicitno nabroje elementi niza razdvojeni zarezima, unutar uglastih zagrada.

Za rad sa elementima niza koriste se petlje (ciklusi), a pored standardnih while, do-while i for koje se koriste i u jeziku C, JavaScript podržava još dva ciklusa: **for-in** i **for-of**.

Kod ciklusa for-in, promenljiva definisana u okviru naredbe uzima redom vrednosti **indeksa** niza (kao što kod objekata uzima imena svojstava), dok kod for-of promenljiva uzima redom vrednosti **elemenata** niza.

Primer.

```

let niz = [2, 3, 5, 7, 11];
for (let i in niz)                // iteriramo kroz indekse
    console.log(niz[i]);

for (let x of niz)                // iteriramo kroz vrednosti elemenata
    console.log(x);

```

→ Ova dva ciklusa imaju potpuno isti rezultat, a to je ispis svih elemenata niza redom.

5. Metodi nad nizovima. Primeri.

Slično kao kod objekata, metodi se kod nizova mogu posmatrati kao osobine koji referišu na vrednosti-funkcije.

Deo metoda (one jednostavnije) koje se mogu primeniti na niz:

- `concat()` - Vrađa spoj dva ili više nizova
- `copyWithin()` - Vrađa niz sa kopijama elemenata iz niza između zadatih pozicija - argumenti `to` i `from`
- `fill()` - Popunjava sve elemente niza datom vrednošću
- `indexOf()` - Traži elementa u nizu i vraća njegovu poziciju
- `isArray()` - Proverava da li argument zaista jeste niz
- `join()` - Vraća string nastao spajanjem svih elemenata niza
- `lastIndexOf()` - Traži elementa u nizu od kraja prema početku i vraća njegovu poziciju
- `pop()` - Uklanja poslednji elemenat niza, vraća upravo uklonjeni elemenat
- `push()` - Dodaje novi elemenat na kraj niza, vraća novu dužinu niza
- `reverse()` - Premešta (u mestu) elemente datog niza da bi se postigao obrnuti redosled
- `shift()` - Uklanja prvi elemenat niza, vraća upravo uklonjeni elemenat
- `slice()` - Bira deo, tj. "isečak" niza, vraća izabrano kao novi niz
- `sort()` - Sortira (u mestu) elemente datog niza
- `splice()` - Dodaje/uklanja elemente iz niza
- `toString()` - Vraća nisku koja predstavlja elemente niza
- `unshift()` - Dodaje novi elemenat na početak niza, vraća novu dužinu niza
- `valueOf()` - Vraća primitivnu vrednost niza

Primer:

```
let poruka = [];  
poruka.push("nema");  
poruka.push("povlačenja", "nema");  
poruka.push("predaje", 2);  
console.log(poruka);           // (5) *'nema', 'povlačenja', 'nema', 'predaje', 2+  
console.log(poruka.join(" ")); // nema povlačenja nema predaje 2  
console.log(poruka.join("+")); // nema+povlačenja+nema+predaje+2  
console.log(poruka.pop());     // 2  
poruka.pop();  
console.log(poruka);           // (3) *'nema', 'povlačenja', 'nema'+
```

```
let podsetnik = [];  
const podsetiMe = function(zadatak) {  
  podsetnik.push(zadatak);  
};  
const stajSledece = function() {  
  return podsetnik.shift();  
};  
const hitnoMePodseti = function(zadatak) {  
  podsetnik.unshift(zadatak);  
};  
podsetiMe("priprema slajdova za predavanja");  
podsetiMe("priprema zadataka");
```

```

hitnoMePodseti("odgovoriti na pisma");
podsetiMe("kupovina knjige");
podsetiMe(() => console.log("Odmor"));
while (podsetnik.length != 0) {
    console.log(staJeSledece());}

console.log([1, 2, 3, 2, 1].indexOf(2));
// prikazuje 1
console.log([1, 2, 3, 2, 1].lastIndexOf(2));
// prikazuje 3
console.log([0, 1, 2, 3, 4].slice(2, 4));
// prikazuje [2, 3]
console.log([0, 1, 2, 3, 4].slice(2));
// prikazuje [2, 3, 4]
const ukloni = function (niz, indeks) {
    return niz.slice(0, indeks)
        .concat(niz.slice(indeks + 1));
}
console.log(ukloni(['a', 'b', 'c', 'd', 'e'], 2));
// prikazuje ['a', 'b', 'd', 'e']

```

6. Nizovi i objekti. Primeri.

Nizovi, koji su takodje objekti, imaju svoje osobine (vec smo rekli da indeksi u nizu mogu da se posmatraju kao svojstva, a elementi na tim pozicijama kao njihove vrednosti). Njima se pristupa pomocu uglastih zagrada [] i naziva osobine pod navodnicima. Npr. `console.log(nizBrojeva["length"]);` pristup sa tackom je pokusaj pristupa osobini koja ne postoji: `console.log(nizBrojeva.length);`

Bitno je da funkcija **typeof()** kada se primeni na niz daje rezultat **Object**. Ako je potrebno proveriti da li dati objekat zaista predstavlja niz, treba koristiti funkciju **isArray()**.

Primer niza objekata koji sadrže nizove:

```

let dnevnik = [
    { karantin: true,
      aktivnosti: ["netflix", "chill", "pica", "kafa", "serija"]
    },
    { karantin: true,
      aktivnosti: ["spavanje", "puding", "nacosi", "film", "lazanja"]
    },
    { "karantin": false,
      "aktivnosti": ["faks", "ucenje", "citaonica", "rad"]
    }
];

```

```

let prviDan = dnevnik[0];
console.log(prviDan.karantin);
console.log(prviDan.aktivnosti);

```

```

console.log("=== Ceo dnevnik ==="); // moze pomocu brojace for petlje ili for-in i for-of petlji
for (let dan of dnevnik) {
    console.log(dan.karantin);
    console.log(dan.aktivnosti);
}

```

7. Niz argumenata pri pozivu funkcije. Primeri.

Iz tela funkcije se moze pristupiti argumentima poziva bez obzira na to da li ih ima vise ili manje od parametara. Ilustrovano na primeru:

```

function brojacArgumenata () {
    console.log("---");
    let s = "";
    for(let i in arguments)
        s += arguments[i];
    console.log(s);
    console.log ("Prilikom poziva su prosleđena " , arguments.length , " argumenta.");
}

```

```

brojacArgumenata ("Ako kaniš " , "pobijediti" , " ne smiješ " , "izgubiti"); // ispisuje spojenu recenicu i broj 4
brojacArgumenata ("Ako kaniš pobijediti" , " ne smiješ izgubiti"); // ispisuje istu spojenu recenicu, ali broj 2

```

Primer. funkcija koja racuna sumu prosledjenih brojeva (nije poznato unapred koliko ih ima)

```

let sumaBrojeva = function () {
    let res = 0.0;
    for (let i in arguments) {
        let elem = Number(arguments[i]);
        if (!isNaN(elem))
            res += elem;
    }
    return res;
}

```

→ Napomena: funkcija proverava za svaki prosledjeni element da li je broj (ili neka vrednost koja se moze konvertovati u broj), pa tek ako jeste dodaje u rezultat.

8. Niske. Primeri.

Niska, kao objekat, poseduje tacno odredjen skup osobina koji se ne moze menjati. Npr. moguće je procitati duzinu niske pristupanjem odgovarajucem svojstvu: niska.length, medjutim nije moguće dodavanje novih osobina, npr. niska.novaOsobina = "vrednost";

```

let niska1 = 'Fido';
console.log(niska1.length); // prikazuje 4
niska1.novaOsobina = 'vrednost';
console.log ( niska1.novaOsobina ); // prikazuje undefined

console.log('coconuts'.slice(4, 7)); // prikazuje nut

```



```

console.log('coconuts'.indexOf('u')); // prikazuje 5
console.log('one two three'.indexOf('ee')); // prikazuje 11
console.log('okay \n '.trim()); // prikazuje okay
let niska = '+abc ';
console.log(niska.charAt(0)); // prikazuje +
console.log(niska[1]); // prikazuje a

```

9. Metodi nad niskama. Primeri.

Slično kao kod objekata, metodi se kod niski mogu posmatrati kao osobine koji referišu na vrednosti-funkcije.

Neke metode koje koriste nizovi mogu koristiti i niske, dok su neki jedinstveni. Neki primeri su:

- slice() - Bira deo, tj. "isečak" niza, vraća izabrano kao novi niz
- indexOf() - Traži elementa u nizu i vraća njegovu poziciju
- trim() - uklanja beline sa kraja niske
- charAt() - vraća karakter na zadatoj poziciji // može i pristup uglastim zagradama []
- toUpperCase() - pretvara sva mala slova u velika (toLowerCase() radi obrnuto)

```

console.log('Markovid'.toUpperCase);           // toUpperCase() { [native code] }
console.log('Markovid'.toUpperCase());          // MARKOVID

```

10. JSON. Primeri.

JSON, odnosno JavaSkript Objektna Notacija, je otvoreni standard dizajniran za predstavljanje i razmenu podataka, tako da je razumljiv ljudima. JSON koristi tekstualno (a ne binarno) predstavljanje podataka.

Kao što samo ime kaže, JSON je izveden iz JavaSkripta i koristi njegove konstrukcije za predstavljanje jednostavnih struktura podataka (objekata) i nizova. Uprkos vezi sa JavaSkriptom, to je jezički nezavistan stanard, podržan od strane mnogih programskih jezika.

JSON format je prvobitno napravio Daglas Krokford, a opisan je u RFC 4627. Zvanična vrsta internet medija (engl. MIME type) za JSON je application/json. Datoteke koje sadrže JSON imaju ekstenziju .json.

JSON format se često koristi za serijalizaciju i prenos strukturiranih podataka preko mreže, kao i za razmenu podataka između servera i veb aplikacije, umesto XML-a.

Primer.

```

let person = {
  name: "Miki Maus",
  born: 1980,
  father: "Volt Dizni"
};
let niska = JSON.stringify(person);
console.log(niska); // dobijamo nisku: ' {"name": "Miki Maus", "born": 1980, "father": "Volt Dizni"} '
let object= JSON.parse(niska);
console.log(object); // od JSON niske pravimo objekat, dobijamo novi objekat isti kao pocetni objekat person
// {name: 'Miki Maus', born: 1980, father: 'Volt Dizni'}

```

9. Funkcije viseg reda

1. Funkcije kao argumenti funkcija. Primeri.

Povratni poziv (callback) je mehanizam, poznat i u drugim jezicima, koji omogućava da se funkcija prosledi kao parametar, da bi kasnije bila pozvana po potrebi.

Primer: Prikaz svih elemenata niza; to se može ostvariti na više načina:

- **for** (**let** i = 0; i < niz.length; i++)
- **for** (**let** i **in** niz) //kolekcijski ciklus for-in
- **for** (**let** tekuci **of** niz) //kolekcijski ciklus for-of

U svakom od ovih načina mora se proći kroz ceo niz, i pri svakoj iteraciji se ispisuje po jedan član niza; odaberemo jedan način i izvojimo u zasebnu funkciju:

```
let nizBrojeva = [1, 2, 3, "mika", "zika"];
```

```
function prikaziSvaki(niz) {  
    for (let i = 0; i < niz.length; i++)  
        console.log(niz[i]);  
}
```

```
prikaziSvaki(nizBrojeva);
```

U ovom primeru je prolazak kroz niz izdvojen u zasebnu funkciju, a aktivnost koja se vrši nad svakim članom niza (prikaz na konzolu) je ubacena u okviru te funkcije. Poželjno je da se prikaz svih elemenata realizuje tako da u posebnoj funkciji bude realizovan prolazak kroz niz i da argument te funkcije bude funkcija koja opisuje šta treba uraditi sa elementom niza.

Primer.

```
let nizBrojeva = [1, 2, 3, 4, "mika", "zika"];
```

```
function zaSvaki(niz, akcija) {  
    for (let x of niz)  
        akcija(x);  
}
```

```
prikazNaKonzolu = function (x) {  
    console.log(x);  
};
```

```
zaSvaki(nizBrojeva, prikazNaKonzolu);
```

```
prikazNaKonzolu2 = (x) => console.log(x);  
zaSvaki(nizBrojeva, prikazNaKonzolu2);    // ista stvar
```

```
zaSvaki(nizBrojeva, function (x) {  
    console.log(x)    // ista stvar  
});
```

```
zaSvaki(nizBrojeva, (x) => console.log(x));    // ista stvar
```

Funkcija zaSvaki kao drugi argument ima funkciju koja se izvršava nad svakim elementom niza. Ta funkcija može biti zadata funkcijskim izrazom ili lambda izrazom, pri čemu taj izraz može biti prethodno dodeljen promenljivoj ili se direktno naci u pozivu funkcije. Funkcija koja funkcionise na sličan način već postoji u prototipu za Array.

2. Korišćenje povratnog poziva nad svakim članom niza. Primeri.

Primer.

```
let nizBrojeva = [1, 2, 3, 4, "mika", "zika"];
function zaSvaki(niz, akcija) {
    for (let x of niz)
        akcija(x);
}
prikazNaKonzolu = function (x) {
    console.log(x);
};
zaSvaki(nizBrojeva, prikazNaKonzolu);

prikazNaKonzolu2 = (x) => console.log(x);
zaSvaki(nizBrojeva, prikazNaKonzolu2);    // ista stvar

zaSvaki(nizBrojeva, function (x) {
    console.log(x)
});
zaSvaki(nizBrojeva, (x) => console.log(x));    // ista stvar
```

3. Metodi objekta Array sa povratnim pozivima. Primeri.

Jedan od najcescijh primera koriscenja funkcija povratnih poziva su metode kod nizova. Na taj nacin se pruza mogucnost prolaska kroz elemente niza, njihovog ispitivanja ili sortiranja, uz mogucnost da sam programer odredi sta ce se desavati sa pojedinacnim elementima.

Postoje ugradjeni metodi u objektu Array koji podrzavaju ovakav rad:

Metoda	Opis
filter()	Kreira novi niz sa elementima koji su uspesno prosli test definisan povratnim pozivom
every()	Proverava da li je svaki element niza uspesno prosao test definisan povratnim pozivom
find()	Vraca vrednost prvog elementa niza koji je prosao test dat povratnim pozivom
findIndex()	Vraca indeks prvog elementa koji je uspesno prosao test dat povratnim pozivom
forEach()	Poziva funkciju povratnog poziva za svaki element niza
map()	Kreira novi niz koji sadrzi rezultate izvršavanja povratnog poziva nad svakim elementom originalnog niza
reduce()	Vrsi redukciju vrednosti elemenata niza na jednu vrednost (redukcija se izvršava s leva u desno)
reduceRight()	Vrsi redukciju vrednosti elemenata niza na jednu vrednost (redukcija se izvršava s desna u levo)
some()	Proverava da li postoji bar neki element niza koji je uspesno prosao test dat povratnim pozivom

Primer. Prikaz svih elemenata koriscenjem metoda forEach() kod nizova:

```
let nizBrojeva = [1, 2, 3, 4, "mika", "zika"];
nizBrojeva.forEach( (x) => console.log(x) );
```

4. Filtriranje, metod filter objekta Array. Primeri.

filter()	Kreira novi niz sa elementima koji su uspesno prosli test definisan povratnim pozivom
----------	---

Primer. Filtriranje podataka u kolekciji:

```
function filter(array, test) {
    let rez = [];
    for (let i = 0; i < array.length; i++) {
        if (test(array[i]))
            rez.push(array[i]);
    }
    return rez;
}

var opis = '{"name":"Emma de Milliano","sex":"f","born":1876,"died":1956,"father":"Petrus de Milliano","mother":"Sophia van Damme"}';
+ *niz objekata, svaki ima name, sex, born, father i mather*;
let family = JSON.parse(opis);

// prikaz muskaraca rođenih između 1900 i 1925
// prvi način, implementirana
console.log(filter(family, function (person) {
    return person.sex == 'm' && (person.born > 1900 && person.born < 1925);
}));

//drugi način, ugrađena
console.log(family.filter( person => person.sex == 'm' && person.born > 1900 && person.born < 1925));

//treći način, ugrađena
console.log(family .filter(person => person.sex == 'm').filter(person.born > 1900 && person.born < 1925));
```

Samo u poslednja dva slučaja je korišćena ugrađena funkcija, tj. metod filter() objekta Array. Kao što se vidi iz poslednjeg primera, korišćenje ugrađenog metoda omogućuje prirodno i lako ulančavanje poziva, što dovodi do boljeg krajnjeg rezultata.

5. Mapiranje, metod map objekta Array. Primeri.

map()	Kreira novi niz koji sadrži rezultate izvršavanja povratnog poziva nad svakim elementom originalnog niza
-------	--

Korišćenje ugrađenih metoda filter() i map() omogućuje prirodno i lako ulančavanje poziva, što dovodi do preglednijeg programskog koda, lakšeg za održavanje.

Primer. Filtriranje, transformacija i sumiranje podataka:

```
let opis = `{"name":"Emma de Milliano","sex":"f", "born":1876,"died":1956, "father":"Petrus de Milliano", "mother":"Sophia van Damme"}," /* *** */
+ {"name":"Carolus Haverbeke", "sex":"m", "born":1832,"died":1905, "father":"Carel Haverbeke", "mother":"Maria van Brussel"}`;
let preci = JSON.parse(opis);
function prosek(niz) {
    function plus(a, b) { return a + b; }
    return niz.reduce(plus) / niz.length;
}
function uzrast(p) { return p.died - p.born; }
```

```
function jeMusko(p) { return p.sex == "m"; }
function jeZensko(p) { return p.sex == "f"; }
console.log(prosek(preci.filter(jeMusko).map(uzrast)));
console.log(prosek(preci.filter(jeZensko).map(uzrast)));
```

Primer 2.

```
const filter = function(niz, uslov) {
  let ispuniliUslov = [];
  for (let i = 0; i < niz.length; i++) {
    if (uslov(niz[i]))
      ispuniliUslov.push(niz[i]);
  }
  return ispuniliUslov;
};

const map = function(niz, transformacija) {
  let mapirano = [];
  for (let i = 0; i < niz.length; i++)
    mapirano.push(transformacija(niz[i]));
  return mapirano;
};

let opis = `[{"name":"Emma de Milliano","sex":"f","born":1876,"died":1956,"father":"Petrus de Milliano","mother":"Sophia van Damme"},` /* *** */
+ `[{"name":"Carolus Haverbeke","sex":"m","born":1832,"died":1905,"father":"Carel Haverbeke","mother":"Maria van Brussel"}]`;

let family = JSON.parse(opis);
console.log('---'); // transformisanje svih pomodu map
console.log(map(family, x => x.name + " " + (x.died - x.born)));
console.log('---'); // filtriranje tako da se zadrže samo stariji od 90
let starijiOd90 = filter(family, function(person) {
  return person.died - person.born > 90;
});
console.log('---'); // transformisanje starijih od 90 pomodu map
console.log(map(starijiOd90, function(person) {
  return person.name + " " + (person.died - person.born);
}));
console.log('---'); // filtriranje i transformisanje pomodu metoda niza
console.log(family.filter(x => x.died - x.born > 70).filter(x => x.sex == 'm')
  .map(x => x.name + " " + (x.died - x.born)));
```

6. Redukcija, metod reduce objekta Array. Primeri.

reduce()	Vrsi redukciju vrednosti elemenata niza na jednu vrednost (redukcija se izvrsava s leva u desno)
----------	--

Primer 1.

```
const reduce = function (niz, kombinuj, pocetak) {
  let tekuci = pocetak;
  for (let i = 0; i < niz.length; i++)
    tekuci = kombinuj(tekuci, niz[i]);
  return tekuci;
}
```

```

let opis =
  `[{"name":"Emma de Milliano","sex":"f", "born":1876,"died":1956, "father":"Petrus de Milliano", "
  mother":"Sophia van Damme"},` /* *** */
  + `[{"name":"Carolus Haverbeke","sex":"m", "born":1832,"died":1905, "father":"Carel Haverbeke",
  "mother":"Maria van Brussel"}]`;
let family = JSON.parse(opis);
console.log(family.reduce(function (min, cur) {
  if (cur.born < min.born)
    return cur;
  else
    return min;
}));

```

Primer 2.

```

const reduce = function(niz, kombinuj, pocetnaVrednost) {
  let akumulator = pocetnaVrednost;
  for (let x of niz)
    akumulator = kombinuj(akumulator, x);
  return akumulator;
};
let brojevi = [2, 4, 3, 1, -5, 12, 7];
console.log('--- Clanovi niza ---'); // prikaz svih clanova niza
console.log(brojevi);
console.log('--- Suma ---'); // odredjivanje sume svih clanova niza
console.log(reduce(brojevi, function(a, b) {
  return a + b;
}, 0));
console.log('--- Suma ---'); // odredjivanje sume svih clanova niza
console.log(reduce(brojevi, (a, b) => a + b, 0));
console.log('--- Suma ---'); // odredjivanje sume svih clanova niza pomocu metoda niza
console.log(brojevi.reduce((a, b) => a + b, 0));
console.log('--- Suma pozitivnih ---'); // odredjivanje sume svih pozitivnih clanova niza
console.log(brojevi.filter(a => a >= 0).reduce((a, b) => a + b, 0));
console.log('--- Proizvod ---'); // odredjivanje proizvoda svih clanova niza
console.log(reduce(brojevi, (a, b) => a * b, 1));
console.log('--- Minimum ---'); // odredjivanje minimuma svih clanova niza
console.log(reduce(brojevi, function(a, b) {
  if (a < b) return a;
  return b;
}, Infinity));
console.log('--- Minimum ---'); // odredjivanje minimuma svih clanova niza
console.log(reduce(brojevi, (a, b) => (a < b) ? a : b, Infinity));
console.log('--- Minimum ---'); // odredjivanje minimuma svih clanova niza
console.log(brojevi.reduce((a, b) => (a < b) ? a : b, Infinity));

```

7. Metod from objekta Array. Primeri.

Pretvaranje iterabilnog objekta u niz:

Korišćenjem metode from() iz Array može se od objekta koji ima strukturu koja podseda na niz, tj. od objekta kroz koji se može iterirati, napraviti niz.

Primer ilustruje korišćenje metode Array.from.

```
let objekat = "Miki Maus";
console.log(objekat); // Miki Maus
let rezultat = Array.from(objekat);
console.log(rezultat); // ['M', 'i', 'k', 'i', ' ', 'M', 'a', 'u', 's']
objekat = { 0: 11, 1: "aaa", 2: 42 };
console.log(objekat); // {0: 11, 1: 'aaa', 2: 42}
rezultat = Array.from(objekat);
console.log(rezultat); // []
objekat = { 0: 11, 1: "aaa", 2: 42, length: 3 };
console.log(objekat); // {0: 11, 1: 'aaa', 2: 42, length: 3}
rezultat = Array.from(objekat);
console.log(rezultat); // (3) [11, 'aaa', 42]
console.log("----");
objekat = { 0: 11, 1: "aaa", 2: 42, 4: false, length: 6 };
console.log(objekat); // {0: 11, 1: 'aaa', 2: 42, 4: false, length: 6}
rezultat = Array.from(objekat);
console.log(rezultat); // (6) [11, 'aaa', 42, undefined, false, undefined]
```

Primer 2:

```
let objekat = "Miki Maus";
let rezultat = Array.from(objekat, (x, i) => x + i);
console.log(rezultat); // (9) ['M0', 'i1', 'k2', 'i3', ' 4', 'M5', 'a6', 'u7', 's8']
objekat = { 0: 11, 1: false, 2: 42, length: 3 };
rezultat = Array.from(objekat, (e, i) => " [" + i + "]:" + e);
console.log(rezultat); // (3) [' [0]:11', ' [1]:false', ' [2]:42']
```

Kreiranje sekvence:

```
const sekvenca = (n) => Array.from({ length: n }, (e, i) => i);
rezultat = sekvenca(25);
console.log(rezultat); // (25) [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]
```

Kreiranje opsega:

```
const opseg = (pocetak, kraj, korak) => Array.from(
  { length: (kraj - pocetak) / korak + 1 }, (_, i) => pocetak + (i * korak));
rezultat = opseg(1, 10, 2);
console.log(rezultat); // (5) [1, 3, 5, 7, 9]
rezultat = opseg('A'.charCodeAt(0), 'Z'.charCodeAt(0), 1).map(x => String.fromCharCode(x));
console.log(rezultat); // (26) ['A', 'B', 'C', ..., 'V', 'W', 'X', 'Y', 'Z']
```

8. Funkcije kao generatori funkcija. Primeri.

Primer. Generisanje funkcija za kvadriranje, dizanje na kub i dizanje na deseti stepen, korišćenjem zatvorenja, na klasičan način i pomodu lambda-izraza:

```
const stepenovanje = function (izlozilac = 2) {  
  return function (osnova) {  
    let ret = 1;  
    for (let i = 0; i < izlozilac; i++)  
      ret *= osnova;  
    return ret;  
  };  
}
```

```
const kvadriranje = stepenovanje(2);  
console.log(kvadriranje(4.5));  
console.log(kvadriranje(10));  
const naKub = stepenovanje(3);  
console.log(naKub(4));  
const naDeseti = stepenovanje(10);  
console.log(naDeseti(2));
```

```
const stepenovanje2 = (izlozilac = 2) =>  
  osnova => {  
    let ret = 1;  
    for (let i = 0; i < izlozilac; i++)  
      ret *= osnova;  
    return ret;  
  }
```

```
const kvadriranje2 = stepenovanje2(2);  
console.log(kvadriranje2(4.5));  
console.log(kvadriranje2(10));  
const naKub2 = stepenovanje2(3);  
console.log(naKub2(4));  
const naDeseti2 = stepenovanje2(10);  
console.log(naDeseti2(2));
```

9. Povezivanje funkcija pri pozivu. Primeri.

Funkcije se mogu tretirati kao objekti. Nad funkcijama se mogu primenjivati sledece ugradjene metode:

- **call()** - Ova metoda odredjuje u izabranoj funkciji zeljeno znacenje kljucne reci this i odmah poziva funkciju. Prvi argument definise na koji objekat ce ukazivati kljucna rec this, dok su ostali argumenti koji se prosledjuju kao lista argumenata, podaci potrebni pocetnoj funkciji.
- **apply()** - Odredjuje u funkciji na koju se primenjuje kakvo de biti znacenje kljucne reci this prilikom njenog izvršenja i potom odmah poziva tu funkciju (engl. invoke). Prvi argument definiše na koji objekat de ukazivati ključna reč this prilikom izvršenja, dok su ostali argumenti (koji se prosleđuju kroz niz) u stvari argumenti poziva ove funkcije.
- **bind()** - Odredjuje u funkciji na koju se primenjuje kakvo de biti znacenje kljucne reci this prilikom njenog izvršenja. Prvi argument je objekat na koji de ukazivati ključna reč this, dok su ostali argumenti (koji se prosleđuju kao spisak) argumenti poziva funkcije na koju se primenjuje bind().

Primer.

```
prikaz = function (visina, tezina) {  
    console.log(`tip: ${this.tip}, naziv: ${this.ime}, visina: ${visina}, tezina: ${tezina}`);  
};  
prikaz(1.82, 87); // tip: undefined, naziv: undefined, visina: 1.82, tezina: 87  
let duskoDugousko = {  
    tip: 'zec',  
    ime: 'Dusko Dugousko',  
    prikazNaKonzolu: prikaz  
};  
duskoDugousko.prikazNaKonzolu(0.3, 5.2); // tip: zec, naziv: Dusko Dugousko, visina: 0.3, tezina: 5.2  
  
let tarzan = {  
    tip: 'covek',  
    ime: 'Tarzan',  
};  
prikaz.call(tarzan, 1.9, 80); // tip: covek, naziv: Tarzan, visina: 1.9, tezina: 80  
duskoDugousko.prikazNaKonzolu.call(tarzan, 1.9, 80); // tip: covek, naziv: Tarzan, visina: 1.9, tezina: 80  
prikaz.apply(tarzan, [1.9, 80]); // tip: covek, naziv: Tarzan, visina: 1.9, tezina: 80  
duskoDugousko.prikazNaKonzolu.apply(tarzan, [1.9, 80]); // tip: covek, naziv: Tarzan, visina: 1.9, tezina: 80  
prikaz.bind(tarzan)(1.9, 80); // tip: covek, naziv: Tarzan, visina: 1.9, tezina: 80  
duskoDugousko.prikazNaKonzolu.bind(tarzan)(1.9, 80); // tip: covek, naziv: Tarzan, visina: 1.9, tezina: 80
```

Ceo postupak određivanja značenja **this** se zasniva na prepoznavanju “ko” i na koji način poziva funkciju koja sadrži this. Redosled radnji je slededi:

- Prvo se proverava da li je to konstruktorska funkcija pozvana sa operatorom new(). Ukoliko jeste, onda vrednost this unutar konstruktorske funkcije ukazuje na primerak (instancu) objekta koji se kreira.
- Potom se proverava da li je to funkcija pozvana uz pomod metoda call(), apply() (ili je pre poziva funkcije izvršeno vezivanje sa bind()). Ako jeste, onda vrednost this referiše na odgovarajući tj. prvi argument koji su prilikom poziva primile funkcije call(), apply(), bind().
- Potom se proverava da li se radi o metodu objekta. Ako jeste, onda se this odnosi na objekat iz kog je pozvana ta metoda (drugim rečima na objekat koji je “vlasnik” metode).
- Ako nijedna od prethodnih provera nije dala pozitivan rezultat, tj. ako funkcija čije telo koja sadrži this nije konstruktor, nije vezivana pomodu call(), apply(), bind() i nije metod objekta, tada ključna reč this označava globalni objekat (on zavisi od okruženja za izvršavanje, kad se programski kod izvršava u pregledaču tada je Window globalni objekat).

10. Funkcija za dekorisanje druge funkcije. Primeri.

U ovom slučaju, funkcija na koju referiše promenljiva bucna dekoriše funkciju-parametar f, tako što pre poziva funkcije na konzolu prikaže naziv funkcije f i parametre poziva, potom pozove funkciju sa tim argumentima i na kraju prikaže dobijeni rezultat. Ovde se dekoriše funkcija sa jednim parametrom.

```
const bucna = function (f) {  
    return function (arg) {  
        console.log(`poziv  '${f.name}' sa argumentom , ${arg}`);  
        let val = f(arg);
```

```

    console.log(`pozvana '${f.name}' sa argumentom`, arg, " - rezultat ", val);
    return val;
  };
};
bucna(Boolean)(0);           // poziv 'Boolean' sa argumentom , 0
                             pozvana 'Boolean' sa argumentom 0 - rezultat false

bucna(Boolean)(2);
bucna(Math.sin)(Math.PI / 2); // poziv 'sin' sa argumentom , 1.5707963267948966
                             pozvana 'sin' sa argumentom 1.5707963267948966 - rezultat 1

```

Primer. Dekorisanje funkcije pomodu apply.

Funkcija na koji referiše promenljiva `bucna2` dekoriše funkciju-parametar `f`, tako što pre poziva funkcije na konzolu prikaže naziv funkcije `f` i parametre poziva, potom pozove funkciju sa tim argumentima i na kraju prikaže dobijeni rezultat.

```

const bucna2 = function (f) {
  return function () {
    console.log(`poziv '${f.name}' sa argumentima `, arguments);
    let rezultat = f.apply(null, arguments);
    console.log(`rezultat koji vraca '${f.name}' je `, rezultat);
    return rezultat;
  };
}

console.log(bucna2(Boolean)(0));           // poziv 'Boolean' sa argumentima Arguments [0]
                                           rezultat koji vraca 'Boolean' je false
                                           false

console.log(bucna2(Math.max)(0, -3, 2, 3));
// poziv 'max' sa argumentima Arguments(4) [0, -3, 2, 3, callee: f, Symbol(Symbol.iterator): f+
// rezultat koji vraca 'max' je 3
// 3

```

10. Napredni objekti

1. Prototipovi. Primeri.

Objekti u JavaSkriptu imaju ugrađeni mehanizam nasleđivanja kroz takozvano prototipsko nasleđivanje. Prototipsko nasleđivanje je vrsta nasleđivanja gde objekat nasleđuje svojstva direktno od drugog objekta (ne od nekog šablona tj. klase). Prototipsko nasleđivanje se zasniva na delegiranju. Svaki objekat ima svoja svojstva i metode, ali i posebnu vezu ka roditeljskom objektu koji onda predstavlja njegov prototip od koga nasleđuje svojstva i metode. Dakle, svaki objekat ima svoj prototipski objekat koji je u prvi mah prazan, ali je povezan preko lanca nasleđivanja sa roditeljskim objektom, njegovim roditeljskim objektom i tako sve do objekta **Object.prototype**. Drugim rečima, lanac nasleđivanja je skup povezanih prototipskih objekata koji se završava sa **Object.prototype**.

Kada JavaScript okruženje za izvršavanje traži neku osobinu ili metodu, prvo se proverava da li je ona definisana u okviru tog objekta, a potom u okviru svog prototipskog objekta. Ukoliko nije, proverava da li je u roditeljskom prototipskom objektu i sve tako do **Object.prototype**. A ako ni tu nije, vrada **undefined**.

Da bi neki objekat mogao da prosledi svom nasledniku metodu, potrebno je da definiše tu metodu u svom prototipskom objektu i tako je učini dostupnom ostalim objektima koji de da nastanu na osnovu tog prototipa.

Korišćenje objekta prototipa ima nekoliko prednosti:

- Kada se koriste prototipovi, smanjuje se količina memorije koju zauzima svaki novi objekat, jer novi objekat može da nasledi mnoga svojstva prototipa.
- Objekat nasleđuje svojstva čak i kada se dodaju njegovom prototipu nakon što se objekat napravi.

Svi objekti u JavaSkriptu imaju zajedničke osobine i metode, definisane u okviru **Object** i navedene u tabeli koja sledi. Vedina tih osobina je korisna samo kada se radi sa objektima koje programer sam definiše.

<i>Naziv osobine</i>	<i>Opis</i>
prototype	Referenca na objekat od koga se nasledjuju svojstva, tj. na svoj prototip
constructor	Referenca na funkciju konstruktor
toString()	Konvertuje objekat u string
toLocaleString()	Konvertuje objekat u lokalizovani string
Valueof()	Pretvara objekat u odgovarajući primitivni tip, najcesce broj
HasOwnProperty(prop)	Vraca true ako objekat ima svojstvo prop, ili false u suprotnom
IsPrototypeOf(obj)	Vraca true ako objekat služi kao prototip objekta obj, u suprotnom false
PropertyIsEnumerable(prop)	Vraca true ako ce svojstvo prop biti navedeno u for-in ciklusu

Primer. Ilustruje kako i "prazan" obekat, tj. objekat koji ne sadrži nijedno polje niti metod, ipak nije potpuno "prazan":

```
let prazanObjekat = {};  
console.log(prazanObjekat.toString());           // [object Object]  
console.log(prazanObjekat.toString());           // function toString(),...-
```

2. Kreiranje objekta i prototipovi. Primeri.

```
console.log(Object.getPrototypeOf({}) == Object.prototype); //ispisuje: true  
console.log(Object.getPrototypeOf(Object.prototype));        //ispisuje: null
```

=> Ne postoji prototip za prototip od Object.

Nizovi su jedna "specijalizovana" vrsta objekata, pa stoga i **nizovi** imaju prototipove. **Funkcije** u JavaSkriptu su građani prvog reda, pa i za njih postoje prototipovi.

Primer. Ilustruje prototipove za nizove i funkcije:

```
console.log(Object.getPrototypeOf(isNaN) == Function.prototype); //true
console.log(Object.getPrototypeOf([]) == Array.prototype); //true
U okviru prototipa figuriše ključna reč this, koja referiše na objekat koji se kreira pomodu datog prototipa.
```

Primer: Ilustruje kreiranje objekata zečeva na osnovu prototipova:

```
let prototipZeca = {
  tip: "nepoznat",
  govori: function (tekst) {
    console.log("Ovaj zec " + this.tip + " kaže '" + tekst + "'");
  }
};
let zec = Object.create(prototipZeca);
zec.govori("Ko sam ja?"); // Ovaj zec nepoznat kaže 'Ko sam ja?'
let zecUbica = Object.create(prototipZeca);
zecUbica.tip = "ubica";
zecUbica.govori("Gotov si!"); // Ovaj zec ubica kaže 'Gotov si!'
```

Uočavamo da se objekat kreira na osnovu prototipa pozivom (statičkog) metoda **Object.create()**. Kako kod prvog napravljenog objekta nije postavljena osobina tip, to de njena vrednost biti preuzeta iz prototipa i bide nepoznat. Postavljanje osobine na željenu vrednost se potom realizovalo odvojeno za svaki od objekata.

Za svaki objekat u jeziku JavaSkript, mogude je pristupiti i njegovom prototipu. Ako se modifikuje prototip nekog objekta, modifikacija de uticati kako na taj objekat, tako i na sve druge objekte koji su kreirani pomodu tog prototipa. Drugim rečima, svaki objekat (što uključuje i funkcije kao građane prvog reda) sadrži vezu prema svom prototipu.

Primer: Ilustruje kreiranje objekata koji predstavljaju tačke na osnovu prototipova, kao i modifikovanje prototipova:

```
let prototipTacke = {
  x: 0, y: 0,
  prikazi: function () { console.log(`(${this.x},${this.y})`); },
  translacija: function (xV, yV) {
    this.x += xV; this.y += yV;
  }
};
let tackaA = Object.create(prototipTacke);
tackaA.prikazi(); // (0,0)
tackaA.x = 7; tackaA.y = 9;
tackaA.prikazi(); // (7,9)
tackaA.translacija(-3, -4);
tackaA.prikazi(); // (4,5)
Object.getPrototypeOf(tackaA).centralnaSimetrija = function (xC, yC) {
  this.translacija(2 * (xC - this.x), 2 * (yC - this.y));
}
```

```

let tackaB = Object.create(prototipTacke);
tackaB.prikazi(); // (0,0)
tackaB.centralnaSimetrija(tackaA.x, tackaA.y);
tackaB.prikazi(); // (8, 10)

```

Ranije smo mogli uočiti da, ako se modifikuje objekat, onda se modifikacija odnosi samo na njega, ne na ostale objekte. Međutim, u ovom primeru se jasno vidi da, ako se modifikuje prototip nekog objekta, onda se ta modifikacija odnosi na sve objekte napravljene pomoću tog prototipa.

Primer:

```

let rabbit = {};
rabbit.name = "Душкп Дугпушкп";
rabbit.speak = function(tekst) {
  console.log("Зека каже: '" + tekst + "'");
};
console.log(rabbit.name); // Душкп Дугпушкп
rabbit.speak("Кпји уи је враг, шефе?"); // Зека каже: 'Кпји уи је враг, шефе?'
let x = rabbit.name;
rabbit.name = rabbit.speak;
rabbit.speak = x;
rabbit.name("Прпба! 1,2,3..."); // Зека каже: 'Прпба! 1,2,3...'

```

3. Konstruktori i prototipovi. Primeri.

Ako se funkcija pozove koriscenjem rezervisane reci *new*, tada se taj poziv kreira kao poziv konstruktora, tj. tako pozvana funkcija predstavlja funkciju – **konstruktor**.

Kao i kod prototipova, u okviru konstruktora figurise (predstavlja se) promenljiva *this*, koja je povezana sa objektom koji se kreira, pa ce taj novi objekat biti povratna vrednost konstruktora – osim kada ova funkcija eksplicitno vraca nesto drugo (ovo ne treba raditi).

Ako se objekat kreira koriscenjem *new*, tada se kaze da je taj objekat primerak svog konstruktora. Da bi se konstruktori lako razlikovali od drugih funkcija, uobicajeno je da nazivi konstruktora pocinju velikim slovom, a da nazivi onih funkcija koje nisu konstruktori pocinju malim slovom.

Primer. Ilustruje kreiranje objekata koji predstavljaju zeceve pomocu konstruktora.

```

function Zec(tip = "непзнау") ,
  this.tip = tip;
  this.govori = function (tekst) {
    console.log("Овај зец " + this.tip + " каже '" + tekst + "'");
  }
}
let zec = new Zec();
zec.govori("Кп сам ја?"); // Овај зец непзнау каже 'Кп сам ја?'
let zecUbica = new Zec("убица");
zecUbica.govori("Гпупв си!"); // Овај зец убица каже 'Гпупв си!'

```

Kada se objektu dodaje osobina, ona ce se dodati samo tom objektu – bez obzira na to da li je ta osobina prisutna u prototipu objekta ili ne. Ako je naziv dodate osobine isti kao naziv u prototipu, na taj objekat se

nadalje nece odnositi osobina iz prototipa, vec ce biti "pregazena" novom dodelom, a sam prototip nece biti promenjen.

Cesto je jako korisno da se prevazidju osobine koje postoje u prototipovima (dobija se mogucnost da se izraze osobine koje predstavljaju izuzetke u odnosu na opstije klase objekata, a objekti koji nisu izuzetni dobijaju standardne vrednosti od svojih prototipova).

Primer. Ilustruje kreiranje objekta pomocu konstruktora i prevazilazenje osobina na primeru objekata koji predstavljaju zeceve:

```
function Zec(tip) {
    this.tip = tip;
}
let zecUbica = new Zec("ubica");
let crniZec = new Zec("crni");
Zec.prototype.zubi = "mali";
console.log(zecUbica.zubi);           // mali
zecUbica.zubi = "dugi, ostri i krvavi";
console.log(zecUbica.zubi);           // dugi, ostri i krvavi
console.log(crniZec.zubi);             // mali
console.log(Zec.prototype.zubi);       // mali
```

Konstruktori (sve funkcije) automatski sadrze i osobinu *prototype* koja ce podrazumevano sadrzati prazan objekat izveden iz *Object.prototype* (svaki primerak kreiran sa konstruktorom, kao svoj prototip ima bas ovaj objekat).

Primer. Ilustruje kreiranje objekta pomocu konstruktora i koriscenje osobine *prototype* konstruktora na primeru objekata koji predstavljaju zeceve:

```
function Zec(tip = "непзнау") ,
    this.tip = tip;
}
Zec.prototype.govori = function (tekst) {
    console.log("Овај зец " + this.tip + " каже '" + tekst + "'");
};
let zec = new Zec();
zec.govori("Кп сам ја?");              // Овај зец непзнау каже 'Кп сам ја?'
let zecUbica = new Zec("убица");
zecUbica.govori("Гпупв си!");           // Овај зец убица каже 'Гпупв си!'
Zec.prototype.predstaviSe = function () {
    console.log("Зец: " + this.tip);
}
let zecDebeljuca = new Zec("дебелјуца");
zecDebeljuca.predstaviSe();              // Зец: дебељуца
zecDebeljuca.govori("Баш сам гладан!"); // Овај зец дебељуца каже 'Баш сам гладан!'
zec.predstaviSe = () => { console.log("-----") };
zecUbica.predstaviSe();                  // Зец: убица
zec.predstaviSe();                       // -----
```

Funkcije i metodi u JS-u se cesto dizajniraju tako da umesto da modifikuju argument poziva, proizvode rezultat kao novi objekat, a samo izvršavanje funkcije ne menja argumente (ovakav kod je citljiviji i lakši za održavanje).

Primer. Ilustruje, na primeru objekta koji predstavljaju tacke, kako kreirati metode koje kao rezultat vraćaju druge objekte.

```
let Tacka = function (x = 0, y = 0) {
    this.x = x;
    this.y = y;
    this.prikazi = function () { console.log(`${this.x},${this.y}`); };
}
let tackaA = new Tacka(7, 9);
tackaA.prikazi(); // (7,9)
Tacka.prototype.translacija = function (xV, yV) {
    return new Tacka( this.x + xV, this.y + yV);
};
let tackaB = new Tacka(1);
tackaB.prikazi(); // (1,0)
const tackaC = tackaB.translacija(tackaA.x, tackaA.y);
tackaC.prikazi(); // (8,9)
Tacka.prototype.centralnaSimetrija = function (xC, yC) {
    return this.translacija(2 * (xC - this.x), 2 * (yC - this.y));
}
```

4. Prototipovi za predefinisane tipove. Primeri.

Primer. Ilustruje, kako se može menjati prototip predefinisanih klasa i kako se prilikom iteriranja kroz osobine objekta koriscenje *for – in* ciklusa, izlistavaju i osobine koje su definisane u prototipu:

```
let mapa = {};
const smesti = function(kljuc, vrednost) {
    mapa[kljuc] = vrednost;
}
smesti("olovka", 0.069);
smesti("sveska", -0.081);
Object.prototype.nesto = "bez veze!";
for (let kljuc in mapa)
    console.log(kljuc); // olovka sveska nesto
console.log("nesto" in mapa); // true
console.log("toString" in mapa); // true
delete Object.prototype.nesto;
```

Iz prethodnog primera se jasno vidi da se prototipovima može manipulirati na isti način kao što se manipuliše sa “normalnim” objektima, kao to da svaki objekat može pristupiti i svim elementima koji su definisani u okviru njegovog prototipa.

Potreban nam je neki mehanizam koji može da utvrdi da li neka osobina pripada objektu preko koga se referise, ili je definisana u odgovarajućem prototipu. Zato je u okviru tipa *Object* definisan metod *hasOwnProperty()*, koji proverava da li objekat sadrži datu osobinu, ali bez konsultovanja prototipova.

Primer. Ilustruje, kako se prilikom iteriranja kroz osobine objekta koriscenjem *for-in* ciklusa, izlistavaju samo one osobine koje su definisane u objektu:

```
console.log(mapa.hasOwnProperty("nesto"));           // false
console.log("toString" in mapa);                   // true
console.log(mapa.hasOwnProperty("toString"));       // false
for (let kljuc in mapa)
    if (mapa.hasOwnProperty(kljuc))
        console.log(kljuc);                         // olovka sveska
delete Object.prototype.nesto;
```

Kao sto se u prethodnim primerima moglo videti, predefinisana funkcija *Object.create()* omogudava kreiranje objekata sa datim prototipom (prosledjenim kao argument funkcije). Prilikom kreiranja objekta na ovaj nacin, moguće je proslediti i vrednost *null* za prototip i na taj nacin kreirati novi objekat bez prototipa.

Primer. Ilustruje kako se objekat koji nema prototip moze koristiti za cuvanje parova *kljuc – vrednost*:

```
let mapa = Object.create(null);
const smesti = function (kljuc, vrednost) {
    mapa[kljuc] = vrednost;
}
smesti("olovka", 0.069);
smesti("sveska", -0.081);
Object.prototype.nesto = "bez veze!";
console.log("nesto" in mapa);                       // false
console.log("toString" in mapa);                    // false
console.log("sveska" in mapa);                      // true
for (let kljuc in mapa)
    console.log(kljuc);                             // olovka sveska
delete Object.prototype.nesto;
```

U gornjem kodu nema vise potrebe da se koristi metod *hasOwnProperty()*, jer su sve osobine osobine bas tog objekta, pa se moze koristiti *for-in* ciklus, bez obzira sta se u medjuvremenu radilo sa *Object.prototype*.

5. Prototipsko nasleđivanje. Primeri.

Primer. Ilustruje prototipsko nasledjivanje, na primeru objekata koji predstavljaju zeceve:

```
let zecPrototip = {
    tip: "неппзнау",
    boja: "неппзнауа",
    predstaviSe: function () {
        console.log("Зеџ: " + this.tip + " бпја: " + this.boja + "." + "\n");
    },
    govori: function (tekst) {
        console.log("Овај зеџ " + this.tip + " бпје " + this.boja + " каже ' " + tekst + " ' " + "\n");
    }
};
let zec = Object.create(zecPrototip);
```



```

zec.predstaviSe(); // Зеџ: неппзнау бпја: неппзнауа.
zec.govori("Кп сам ја?"); // Овај зеџ неппзнау бпје неппзнауа каже 'Кп сам ја?'
let zecIzFikcijePrototip = Object.create(zecPrototip);
zecIzFikcijePrototip.tip = "неппзнау";
zecIzFikcijePrototip.boja = "неппзнауа";
zecIzFikcijePrototip.predstaviSe = function () {
    console.log("Зеџ: " + this.tip + ", бпја: " + this.boja + ", име: " + this.ime + "\n" + "креаупр: "
        + this.kreator.ime + " " + this.kreator.prezime + "\n" + "делп: " +
        this.delo + "\n" + "узречица: " + this.uzrecica + "\n");
}
let duskoDugousko = Object.create(zecIzFikcijePrototip);
duskoDugousko.tip = "памеуан";
duskoDugousko.boja = "сива";
duskoDugousko.ime = "Душкп Дугпушкп";
duskoDugousko.kreator = { "ime": "Tex", "prezime": "Avery" };
duskoDugousko.delo = "A Wild Hare";
duskoDugousko.uzrecica = "Шефе, кпји уи је враг?";
duskoDugousko.predstaviSe(); // Зеџ: памеуан, бпја: сива, име: Душкп Дугпушкп
                             креаупр: Tex Avery         делп: A Wild Hare
                             узречица: 'Шефе, кпји уи је враг?'

//prototipsko nasledjivanje SA KONSTRUKTORIMA:

ZecIzFikcije.prototype.skoci = function() {
    console.log("Скпк, скпк, скпк \n");
};
let duskoDugousko = new ZecIzFikcije("памеуан", "сива", "Душкп Дугпушкп",
    "Tex", "Avery", "A Wild Hare", "Шефе, кпји уи је враг?");
duskoDugousko.skoci();
// >>> Скпк, скпк, скпк

```

6. Prototipsko nasleđivanje za predefinisane tipove. Primeri.

Primer. Ilustruje da je prototipsko nasledjivanje primenjeno u predefinisanim tipovima, čime je postignuto da standardna ugradjena funkcija *toString()* ima drugacije ponasanje kada se primeni na objekat, u odnosu na njeno ponasanje kada se primeni na niz:

```

console.log(Array.prototype.toString == Object.prototype.toString); // false
console.log([1, 2].toString()); // 1, 2
console.log(Object.prototype.toString.call([1, 2])); // [object Array]

```

7. Klase. Primeri.

Za **klase** u JavaScriptu se može smatrati da predstavljaju samo dodatnu sintaksnu olakšicu na prethodno opisano nasleđivanje preko prototipa. Na taj način, programeri koji su navikli da rade sa jezicima koji podržavaju klasno nasleđivanje imaju lakši prelaz, odnosno manji nagib na krivoj učenja - JavaScript kod je naizgled sličniji kodu sa kojim su oni radili u drugim programskim jezicima.

Primer. Ilustruje rad sa klasama, na primeru objekata koji predstavljaju zečeve:

```
class Zec {
  constructor(tip = "непзнaй", boja = "непзнaя") {
    this.tip = tip;
    this.boja = boja;
  }
  govori(tekst) {
    console.log("Oвaј зeц " + this.tip + " бнје " + this.boja + " кaжe " + tekst + " ' " + "\n");
  }
  представиSe() {
    console.log("Зeц: " + this.tip + " бнја: " + this.boja + " ." + "\n");
  }
}
let zec = new Zec();
zec.predstaviSe();
zec.govori("Ko cаm ja?");
let duskoDugousko = new Zec("пaмeчaн", "cив");
duskoDugousko.predstaviSe();
duskoDugousko.govori("Кпји yи је вpaг, шeфe?");
```

Veliki broj popularnih programskih jezika direktno podržava mogućnost da se neka osobina objekta učini nedostupnom van metoda klase. Jezik JavaSkript ne podržava direktno tu mogućnost, ali se ona može imitirati korišćenjem **zatvorenja za funkcije**.

Primer. Ilustruje rad sa privatnim osobinama, na primeru klase koja predstavlja tačku.

Klasa *Tacka* u primeru koji sledi je kreirana tako da se ne mogu iz spoljašnjosti (van metoda klase) menjati njene koordinate.

```
class Tacka {
  constructor(x = 0, y = 0) {
    this.getX = function() { return x; }
    this.getY = function() { return y; }
  }
  prikazi() {
    console.log(`(${this.getX()}, ${this.getY()})`);
  }
  translacija(xV, yV) {
    return new Tacka(this.getX() + xV, this.getY() + yV);
  }
}
```

Ovde su u okviru klase obezbeđeni metodi za čitanje koordinata tačke, ali nisu izloženi metodi za postavljanje koordinata. Koordinate Tačke se mogu postaviti samo prilikom kreiranja tačke, a posle toga ne. Takođe, moguće je postaviti i odnos nasleđivanja između klasa.

Primer. Ilustruje nasleđivanje klasa, na primeru objekata koji predstavljaju zečeve:

```
class Zec {
```

```

    constructor(tip = "непзнай", boja = "непзнаја") {
        this.tip = tip;
        this.boja = boja;
    }
    govori(tekst) {
        console.log("Овај зец " + this.tip + " бпје " + this.boja + " каже '" + tekst + "' " + "\n");
    }
    predstaviSe() {
        console.log("Зец: " + this.tip + " бпја: " + this.boja + "." + "\n");
    }
}

class ZeclzFikcije extends Zec {
    constructor(tip, boja, ime, imeKreatora, prezimeKreatora, delo, uzrecica) {
        super(tip, boja);
        this.ime = ime;
        this.kreator = { "ime": imeKreatora, "prezime": prezimeKreatora };
        this.delo = delo;
        this.uzrecica = uzrecica;
    }
    predstaviSe() {
        console.log("Зец: " + this.tip + ", бпја: " + this.boja + ", име: " + this.ime + "\n" + "креаупр: "
            + this.kreator.ime + " " + this.kreator.prezime + "\n" + "делп: " + this.delo + "\n"
            + "узречица: " + this.uzrecica + "\n");
    }
    skoci() {
        console.log("Скпк, скпк, скпк \n");
    }
}

let plaviZec = new ZeclzFikcije("вепма памеуан", "плава", "Плави зец", "Душкп", "Радпвић", "Плави
зец", "Плави, зец, чудни зец, једини на свеуу.");
plaviZec.predstaviSe();
plaviZec.govori(plaviZec.uzrecica);

```

8. Metodi za postavljanje i čitanje osobina. Primeri.

U jeziku JavaSkript se mogu podesiti elementi, tako da, kada se na objekat referiše iz spoljašnjosti, izgleda kao da se radi sa "normalnim" osobinama, a da istovremeno budu primenjeni metodi objekta koji su pridruženi datoj osobini. Drugim rečima, **metodi za očitavanje** (engl. **getter**) i **metodi za postavljanje** (engl. **setter**) osobina su funkcije koje služe za napredno definisanje osobina (svojstva). One praktično predstavljaju „prvu liniju odbrane“ osobine, tačnije sprečavaju programere da pristupaju svojstvima kao „golim“ podacima.

Metod za čitanje, kako mu ime i kaže jeste funkcija koja obezbeđuje čitanje svojstva. Šta god da radi, na kraju mora pomodu naredbe *return* da vrati vrednost koja de onda da predstavlja vrednost svojstva.

Slično tome, metod za postavljanje je funkcija koja se poziva kada se zadaje vrednost svojstva. Ova funkcija mora imati jedan parametar koji predstavlja zadatu vrednost. Negde „u pozadini” se ta vrednost beleži - najčešće u nekom svojstvu objekta, koje se ne „eksponira”. Naravno, JavaSkript nema mehanizam kojim bi to svojstvo bilo zaista potpuno sakriveno, tako da je u pitanju prosto konvencija - dogovor.

Primer. Ilustruje rad sa metodama za postavljanje i čitanje osobina:

```
const gomila = {
  elementi: ["ljuska jajeta", "kora pomorandze", "crv", "stara novina"],
  get visina() {
    return this.elementi.length;
  },
  set visina(vrednost) {
    console.log(`Pokusaj da visina gomile bude ${vrednost} je ignorisan.`);
  }
};
console.log(gomila.visina);
gomila.visina = 100;
```

U objektno orjentisanim jezicima, kao što su Java i C++, neka svojstva se mogu deklarirati kao privatna, tako da je nemoguće upravljati njima izvan te klase.

JavaSkript ne podržava direktno postavljanje svojstva kao privatnog. Međutim, sakrivanje od spoljašnjosti može da se postigne tako što se čitanje i upisivanje svojstva može realizovati samo preko posebnih pristupnih metoda, koji koriste zatvorenja.

Primer. Ilustruje, na primeru objekata koji predstavljaju tačke, kako imitirati privatna polja. Ovde se u tu svrhu koriste zatvorenja i metodi za čitanje osobina.

```
class Tacka {
  constructor(x = 0, y = 0) {
    this.getX = function() { return x; }
    this.getY = function() { return y; }
  }
  get x(){ return this.getX(); }
  get y(){ return this.getY(); }
}
```

U ovom kodu se radi lakšeg čitanja i upisivanja vrednosti može pristupati samo preko posebnih pristupnih metoda, a pristupne metode ne mogu se naslediti od prototipa.

11. Asinhroni JavaScript

1. Tipični modeli izvršavanja programa.

Prema načinu izvršavanja programa, tipično se govori o sledećim modelima izvršavanja:

1. **sinhroni** (program se izvršava sukcesivno tj. jedno za drugim)
2. **višenitni** (više procesa se izvršava uporedo)
3. **asinhroni** (program se izvršava na preskok, kako šta postaje dostupno)

2. Sinhroni model programiranja.

Sinhroni model je najstariji i najjednostavniji model programiranja. Zadaci se izvršavaju sukcesivno (pojedinačno), jedan po jedan, i tek kada se izvrši prethodni, počinje sledeći.

U sinhronom modelu, naredba koja sledi de biti izvršena tek nakon što se izvrši naredba koja njoj prethodi. Ako se dogodi da je naredba koja prethodi skupa i da dugo traje, naredba de biti blokirana, tj. morade da čeka. Ovo je ozbiljan problem kada se razvijaju sistemi visokih performansi.

Postoji još jedan problem kod sinhronog modela, koji se manifestuje kod korisničkog interfejsa. Dok program izvršava zadatak koji može da potraje neko vreme, nema mogućnosti da se blokiraju korisnici, koji mogu da unose nešto u polje za unos dok se izvršava skupi zadatak. Sa druge strane, ne bi bilo dobro da se blokira unos korisnika tokom izvršavanja skupe operacije jer zahtevni zadaci treba da se izvršavaju u pozadini.

3. Višenitni model programiranja.

Jedno od rešenja ovog problema je da se svaki zadatak podeli na **programske niti kontrole**. Ovo se zove **višenitni model** (engl. **multithreading**). U višenitnom modelu svaki zadatak se izvršava u nitima kontrole. Nitima, obično, upravlja operativni sistem i one mogu da se izvršavaju uporedo na drugim procesorima. Zahvaljujući modernim procesorima, višenitni model može da ima izuzetno dobre performanse. Nekoliko jezika podržava ovaj model (C#, C++, Java, Rust...).

Višenitni model može biti kompleksan za implementiranje. Naime, niti treba međusobno da sarađuju, što može vrlo brzo da postane „nezgodno“. Međutim, postoje varijacije višenitnog modela u kojima je stanje nepromenljivo i tada se model pojednostavljuje, jer svaka nit je odgovorna za nepromenljivo stanje i nema potrebe da se upravlja stanjima između niti.

4. Asinhroni model programiranja.

Model **asinhronog programiranja** ima jednu nit kontrole, unutar koje se zadaci prepliduju. Kada se izvršava jedan zadatak, može se biti siguran da je samo taj zadatak izvršen. U asinhronom modelu nije potreban složen mehanizam za komunikaciju između niti, pa je zato predvidljiviji.

U kojim situacijama je asinhroni model bolji od sinhronog?

Kada god program čeka nešto - učitavanje podataka sa diska, upit prema bazi podataka ili mrežne zahteve. Ovo su sve blokirajuće operacije. U slučajevima kada program ima mnogo ulaza/izlaza iz izvora kao što su učitavanje diska ili mrežni pozivi, kašnjenje se ne može predvideti. U sinhronom programu nepredvidljivost je „recept“ za lošu performansu. Kada se asinhroni program suoči sa blokirajućim zadatkom, izvršava se naredni zadatak, bez čekanja da se blokirajuća operacija završi.

5. JavaScript okruženje i asinhrono programiranje.

Jezik JavaScript je nastao kako bi se olakšalo veb programiranje. Specifičnost veba je i u tome što, kod malo složenijih aplikacija, često se dolazi u situaciju da se prilikom izvršavanja programa (na pola posla) moraju čekati podaci sa servera. Tada se mora zaustaviti rad, tako da sve može da se nastavi kada podaci budu stigli.

To praktično znači da treba da se „prepolovi“ JavaScript program na deo koji „poziva“ i deo koji „dočekuje“ podatke. Ideja je da se, u momentu kada server odgovori, pozove funkcija povratnog poziva koja služi kao početna tačka za nastavak rada. U primerima koji slede kašnjenje u komunikaciji sa serverom de biti simulirano pozivom funkcije **setTimeout()** - njeni argumenti su funkcija povratnog poziva i vreme čekanja dato u milisekundama.

JavaScript okruženje za izvršavanje se sastoji od sledećih komponenti:

- **JavaScript mašina** (engl. JS Engine) (Hip i stek)
- **Spoljašnji API-ji**
- **Red povratnih poziva** (Red zadataka)
- **Petlja za događaje**

Asinhrono programiranje se realizuje tako što se naredbe izvršavaju jedna za drugom - programski kod se postavlja na Stek i izvršava se onaj kod koji je na vrhu steka. Međutim, ako su neke naredbe/pozivi funkcija takvi da ih ne treba odmah izvršiti, njihov programski kod se stavlja u odvojeni red za čekanje, pa se (po ispunjenju uslova) premešta u Red zadataka. U međuvremenu se nastavi sa izvršavanjem naredbi koje slede - one se guraju na Stek i tamo se izvršavaju (ako su takve da ih treba odmah izvršiti).

Kada stek postane prazan, petlja za događaje uzima elemente iz Reda zadataka, prebacuje ih na Stek i onda taj programski kod biva izvršen. Ovim je postignuto da se ne čeka prilikom izvršavanja „skupih“ naredbi, već se nastavlja sa izvršavanjem.

Postoji više programskih pristupa u JavaScriptu kojima se realizuje asinhrono programiranje. Istorijski najstariji je rad pomodu **funkcija povratnog poziva** (engl. **callback function**).

6. Asinhrono programiranje i povratni pozivi. Primeri.

Primer. Ilustruje rad sa povratnim pozivom:

```
let povratniPoziv = () => {  
  console.log(`Ziv sam!`)  
}  
console.log(`Pokrenuto...`)  
setTimeout(povratniPoziv, 2000)  
console.log(`Zavrsava...`)
```

JavaScript okruženje izvršava skriptu naredbu po naredbu:

1. Prvo stavi na stek prvu **console.log** metodu i zatim je izvrši.
2. Nakon toga okruženje izvršava naredbu gde se poziva **setTimeout**, pa ovu metodu stavlja na vrh steka i izvršava je.
3. Nakon izvršenja **setTimeout** metode ona se sklanja sa steka, a njena funkcija povratnog poziva (engl. **callback**) na koju referiše promenljiva **povratniPoziv** se prebacuje u red povratnih poziva na čekanju i uključuje časovnik koji broji milisekunde.
4. Potom okruženje nastavlja sa obradom naredne naredbe koda gde nailazi na drugi poziv **console.log** metode, koju stavlja na stek i izvršava je.
5. Dok se izvršava ostali deo programa, a po isteku vremena zadatog pozivu **setTimeout** (ovde je to **2000**) funkcija povratnog poziva se prebacuje u red zadataka i tamo čeka da se stek isprazni da bi mogla da se izvrši
6. Petlja za događaje potom, kada stek postane prazan, a ovaj zadatak "dođe na red" prebacuje zadatak (tj. funkciju povratnog poziva) iz reda zadatka na stek.
7. Kada se funkcija povratnog poziva nađe na steku, ona se izvršava i nakon toga sklanja sa steka.

Prema tome, izlaz koji de se pokazati na konzoli je:

```
Pokrenuto...
Završava...
Ziv sam!
```

Da bi se mogao uspešno realizovati asinhroni program, neophodno je da postoji **mehanizam za sinhronizaciju**, tj. obezbeđivanje čekanja tamo gde je to neophodno.

U primeru koji sledi opisan je JavaScript kod bez ikakve sinhronizacije.

Primer. Prikaz (sa čekanjem) prvih pet slova u proizvoljnom redosledu. (Prikaz pojedinačnog slova je realizovan korišćenjem funkcije za prikaz niske posle čekanja:)

```
function prikaziNisku(niska) {
    setTimeout(
        () => {
            console.log(niska);
        },
        Math.floor(Math.random() * 50) + 1
    );
}
function prikaziSve() {
    prikaziNisku("A");
    prikaziNisku("B");
    prikaziNisku("C");
    prikaziNisku("D");
    prikaziNisku("E");
}
prikaziSve();
```

Funkcija za prikaz niske posle čekanja, tj. **prikaziNisku()** je realizovana preko funkcije **setTimeout()**, gde je povratni poziv u okviru te funkcije lambda-izraz kojim se niska šalje na konzolu.

Redosled prikaza zavisi od vrednosti koju vrada generator pseudoslučajnih brojeva, pa se u ponovljenim izvršavanjima skripte dobijaju različite vrednosti.

Ovakvo rešenje je najefikasnije, u smislu da je ukupno čekanje minimalno, ali nema nikakve garancije o tome kojim de se redosledom izvršiti metode.

Slededi primer pokazuje kako se može obezbediti **sinhronizacija korišćenjem povratnih poziva**.

Primer. Prikaz (sa čekanjem) prvih pet slova uz ograničenje u redosledu tako da se slovo V mora uvek prikazati pre slova G.

I u ovom primeru je prikaz pojedinačnog slova realizovan korišćenjem funkcije za prikaz niske posle čekanja, samo što, za razliku od prethodnog primera, ta funkcija ima i dodatni argument - funkcija povratnog poziva, a u telu funkcije se povratni poziv realizuje **posle** slanja niske na konzolu.

Dakle, ako treba postidi da se slovo G uvek prikazuje posle slova V, onda se funkcija za prikaz slova G postavlja kao povratni poziv kod funkcije za prikaz slova V. U svim ostalim slučajevima, funkcija povratnog poziva je funkcija koja ne radi ništa, ovde zapisana pomodu lambda-izraza `()=>{}`.

```
function prikaziNisku(niska, povratniPoziv) {
    setTimeout(
        () => {
            console.log(niska);
            povratniPoziv();
        },
        Math.floor(Math.random() * 50) + 1
    );
}
function prikaziSve() {
    prikaziNisku("A", ()=>{});
    prikaziNisku("B", ()=>{});
    prikaziNisku("C", ()=>{prikaziNisku("D", ()=>{})});
    prikaziNisku("E", ()=>{});
}
prikaziSve();
```

Prikaz:

```
V D V D
D A B B
A B G V
B V A A
G G D G
```

Kao što se vidi iz prethodnih rezultata, slova **V i G** ne moraju biti neposredno jedno ispred drugog, ali u svakom slučaju (tj. prilikom svakog izvršavanja skripte) de slovo **V** prethoditi slovu **G**.

Daljom primenom prethodno opisane ideje mogude je postidi **serijalizovano izvršavanje** željenih operacija u okviru asinhronog koda.

Primer. Prikaz (sa čekanjem) prvih pet slova po azbučnom redosledu.

```
function prikaziNisku(niska, povratniPoziv) {
    setTimeout(
        () => {
```



```

        console.log(niska);
        if(typeof povratniPoziv === 'function')
            PovratniPoziv()
    },
    Math.floor(Math.random() * 50) + 1
);
}
function prikaziSveRedom() {
    prikaziNisku("A",
    ()=>{
        prikaziNisku("B",
        ()=>{
            prikaziNisku("C",
            ()=>{
                prikaziNisku("D");
            });
        });
    });
}
prikaziSveRedom();

```

I ovde funkcija za prikaz niske očekuje dva parametra: nisku koju treba prikazati na konzoli i funkciju povratnog poziva. Međutim, telo funkcije je malo izmenjeno, pa posle prikaza niske povratni poziv bude realizovan samo ako drugi parametar zaista jeste funkcija.

Prilikom svakog izvršavanja gornje skripte, slova de se pojavljivati uvek u istom redosledu (uz čekanje koje varira između različitih izvršavanja).

Uočava se da ovakvo strukturisanje programa, sa visokim stepenom ugnježdavanja funkcija, nije pogodno u nešto složenijim situacijama, koje su u praksi česte. Mogude je “ulančati” pozive, tj. funkcijama povratnog poziva proslediti rezultat rada tj. vrednost koju vrada prethodno izvršena funkcija.

Primer. Prikaz (sa čekanjem) prva tri slova po azbučnom redosledu (ovaj primer ilustruje asinhroni rad sa funkcijama koje vrađaju rezultate)

```

function dodajNisku(prethodna, tekuca, povratniPoziv) {
    setTimeout(
        () => {
            povratniPoziv(prethodna + ' ' + tekuca);
        },
        Math.floor(Math.random() * 50) + 1
    );
}
function dodajSveRedom() {
    dodajNisku('', 'A', result => {
        dodajNisku(result, 'B', result => {
            dodajNisku(result, 'V', result => {

```

```

        console.log(result);
    });
    });
    });
}
dodajSveRedom();

```

Rezultat je: A B V.

Dodatnu komplikaciju u intenzivnom korišćenju ovog principa predstavlja činjenica da treba obezbediti smislen odgovor ukoliko prilikom izvršavanja neke od funkcija povratnog poziva bude došlo do greške.

7. Asinhrono programiranje i obedanja. Primeri.

U JavaSkriptu postoji izraz za deo koda sa ugnježđenim funkcijama povratnih poziva, pod nazivom “**pakao povratnih poziva**” ili “**piramida propasti**”. Debugiranje takvog programskog koda (pa čak i samo razumevanje) je veoma otežano.

Sa standardom ES2015 došla je nova konstrukcija pod nazivom **obedanje** (engl. *promise*), koja sa svojim API-jem obezbeđuje bolji i pregledniji način za organizovanje funkcija povratnih poziva. Ovo se naročito primenjuje u radu sa asinhronim operacijama jer *sintaksa obedanja* veoma liči na standardnu sinhronu sintaksu.

Obedanje je JavaSkript objekat u kome se čuvaju rezultati asinhronih funkcija sve dok traje izvršavanje asinhronih operacija. Preciznije iskazano, obedanje je sinhrono vraden objekat pri izvršenju asinhronih operacija, koji predstavlja privremenu zamenu za moguće rezultate te asinhronih operacija. On umesto krajnje vrednosti, daje obedanje da de dostaviti tu vrednost u nekom trenutku u budućnosti (otud mu i ime).

Pošto su objekti, obedanja, privremena zamena za buduću eventualnu vrednost, to omogućava da programer preko obedanja pridruži rukovaoce budućem rezultatu asinhronih operacija. Na ovaj način i sinhronih i asinhronih operacija mogu da vradaju neku vrednost, s tim što sinhronih odmah vradaju krajnji podatak, a asinhronih referencu za buduću podatak.

Asinhrona funkcija može imati dva moguća krajnja rezultata, a to su “uspešno izvršena operacija” ili “neuspešno izvršena operacija”.

Obedanje se može nalaziti u jednom od tri stanja:

1. **Na čekanju** (engl. *pending*) – kada se asinhrona radnja još uvek izvršava
2. **Ispunjeno** (engl. *fulfill*) – kada je asinhrona radnja završena uspešno
3. **Odbijeno** (engl. *reject*) – kada je asinhrona radnja neuspešno završena greškom

Prvi korak u radu sa obedanjima je kreiranje obedanja unutar asinhronih funkcija. Naime, da bi se buduću krajnji rezultat asinhronih funkcija mogao zameniti sa obedanjem, potrebno je da funkcija kao rezultat vrati objekat-obedanje. Svakom novom obedanju se kroz parametar prosleđuje funkcija-izvršitelj (engl. **executor function**) koja obrađuje samu asinhronu operaciju i buduću rezultate te asinhronih operacija.

Pseudo-kod asinhronih funkcija koja koristi obedanja je dat u produžetku:

```

function asinhronaFunkcija() {
    return new Promise(function (resolve, reject) {

```

```

        //...kod za asinhronu operaciju...
        if (uspešna operacija) ,
            resolve(result_value);
        } else {
            reject(error);
        }
    });
}

```

U zavisnosti od uspešnosti operacije, *funkcija-izvršitelj* poziva jednu od dve funkcije koje su joj prosleđene kao parametri:

- **Funkcija za razrešavanje** (engl. **resolve**) se poziva u delu koda koji obrađuje uspešno završenu asinhronu operaciju. Parametar ove funkcije predstavlja dobijeni podatak iz uspešno završene operacije, stoga se funkcija za razrešavanje koristi da kroz svoj parametar prosledi rezultujući podatak odgovarajućem rukovaocu.
- **Funkcija za odbijanje** (engl. **reject**) se poziva u delu koda koji obrađuje slučaj kada se pojavi problem sa izvršavanjem asinhronne operacije. Ona kroz svoj parametar prosleđuje razlog neuspešnosti asinhronne operacije odgovarajućem rukovaocu.

Primer. Prikaz (sa čekanjem) prvih pet slova.

```

function prikaziNisku(niska) {
    return new Promise((razresi, odbij) => {
        setTimeout(
            () => {
                console.log(niska);
                razresi();
            }, Math.floor(Math.random() * 50) + 1);
    });
}

function prikaziSve() {
    prikaziNisku("A");
    prikaziNisku("B");
    prikaziNisku("C");
    prikaziNisku("D");
    prikaziNisku("E");
}

prikaziSve();

```

U ovom primeru, prikaz niske sa čekanjem je izdvojen u posebnu funkciju koja koristi obedanja. Ovde je, što je atipično, predviđeno da se asinhronne operacije uvek uspešno realizovati i zato se struktura te funkcije ne poklapa u potpunosti sa pseudokodom koji joj prethodi.

Rezultat rada skripte dovodi da se prikažu slova **A-G**, svako posle izvesnog čekanja, pri čemu redosled prikaza nije fiksiran, niti je na njemu postavljeno bilo kakvo ograničenje.

Obično se nakon promene stanja obedanja (iz stanja “na čekanju” u stanje “ispunjeno” ili u stanje “odbijeno”) poziva metod **then()**. Ovaj metod, definisan u prototipu obedanja, prihvata dva parametra koji su tipa funkcije. Prvi parametar opisuje šta da se radi ako je obedanje ispunjeno - to je funkcija koja prihvata jedan parametar kroz koji joj se prosleđuje podatak dobijen asinhronom operacijom.

Drugi parametar opisuje šta de se raditi ako je obedanje odbijeno - ta funkcija takođe prihvata jedan parametar kroz koji joj se prosledjuje razlog neuspeha.

Primer. Prikaz (sa čekanjem) prvih pet slova uz ograničenje u redosledu tako da se slova **A**, **B** i **V** moraju uvek prikazati u tom redosledu.

```
function prikaziNisku(niska) {
  return new Promise((razresi, odbij) => {
    setTimeout(
      () => {
        console.log(niska);
        razresi();
      }, Math.floor(Math.random() * 50) + 1);
  });
}
function prikaziTriRedom() {
  prikaziNisku("A")
    .then(() => {
      return prikaziNisku("B")
    })
    .then(() => prikaziNisku("V"));
  prikaziNisku("");
  prikaziNisku("A");
}
prikaziTriRedom();
```

Rezultat izvršavanja nije determinističan, ali svaki put se slovo **V** mora pojaviti posle slova **B**, koje se mora pojaviti posle slova **A** - mada ne neposredno posle, ved između njih može biti "umetnut" i prikaz nekog drugog slova.

Primer. Prikaz (sa čekanjem) prva tri slova po azbučnom redosledu (U ovom primeru je "ulančavanje" rezultata realizovano preko obedanja, pri čemu su funkcije povratnog poziva date kao lambda-izrazi.)

I ovom slučaju je isti rezultat: prikaz prva tri slova azbuke u rastudem redosledu: A B V

```
function dodajNisku(prethodna, tekuca) {
  return new Promise((razresi, odbij) => {
    setTimeout(
      () => {
        razresi(prethodna + ' ' + tekuca);
      }, Math.floor(Math.random() * 50) + 1);
  });
}
function dodajSveRedom() {
  dodajNisku("", 'A')
    .then(result => dodajNisku(result, 'B'))
    .then(result => dodajNisku(result, 'V'))
    .then(result => console.log(result));
}
```

8. Asinhrono programiranje i naredbe `async` i `await`. Primeri.

Korišćenje samo jednog obedanja je jasno i jednostavno, međutim kada se program zakomplikuje asinhronom logikom, rad sa obedanjima se rapidno otežava.

Sa standardnom ES2017 je stigla i nova sintaksa **`async/await`**, koja olakšava rad sa obedanjima i omogućava jednostavnije predstavljanje serije asinhronih obedanja. Korišćenje **`async/await`** sintakse omogućava da se čitljivije prikažu višestruke međusobno zavisne asinhronne radnje, i da se na taj način izbegne tzv. “pakao obedanja”.

Treba napomenuti da se uz **`async` funkcije** ne mogu koristiti “obične” funkcije povratnih poziva.

Sintaksa `async/await` ne isključuje obedanja, nego menja način konzumiranja obedanja. Ova sintaksa omogućava programeru da piše asinhroni kod prema sekvencijalnom redosledu izvršavanja tako da liči na sinhroni kod.

Asinhrona funkcija, obeležena ključnom reči **`async`**, funkcija je unutar koje se izvršava asinhroni kod. Tako obeležena funkcija daje do znanja izvršnom okruženju da de se unutar nje izvršiti asinhrona operacija.

Prilikom izvršenja asinhronne funkcije, u pozadini se izvršavaju sledede aktivnosti:

1. Automatski konvertuje regularnu funkciju u obedanje
2. Sve što je vradeno naredbom **`return`** u telu funkcije, nakon uspešne operacije to biva prosleđeno funkciji za razrešavanje obedanja iz tačke 1. Asinhrona funkcija uvek vrada obedanje. Čak i ako vradena vrednost funkcije nije obedanje, asinhrona funkcija de obavijati svaku vradenu vrednost i prosleđivati je kao obedanje.
3. Asinhrona funkcija omogućava korišćenje **`await`** operatora.

Primer. Prikaz (sa čekanjem) prvih pet slova (u ovom slučaju je funkcija za prikaz svih elemenata napravljena kao asinhrona funkcija.)

```
function prikaziNisku(niska) {  
    return new Promise((razresi, odbij) => {  
        setTimeout(  
            () => {  
                console.log(niska);  
                razresi();  
            }, Math.floor(Math.random() * 50) + 1);  
    });  
}  
  
async function prikaziSve() {  
    prikaziNisku("A");  
    prikaziNisku("B");  
    prikaziNisku("C");  
    prikaziNisku("D");  
    prikaziNisku("E");  
}  
  
prikaziSve();
```

U ovom slučaju, proglašavanje funkcije asinhronom dopušta da se naredbe unutar metoda asinhrono izvršavaju, pa redosled prikaza slova nije predefinisani.

Operator await može da se koristi jedino u okviru asinhronone funkcije, gde pauzira njeno izvršenje. Naime, rezervisana reč **await** pauzira izvršenje asinhronone funkcije sve dok ne dobije rezultate operacije tj. dok se ne vrati obedanje. Ako je obedanje ispunjeno, onda rezultat koji je “dočekao” **await** vrada vrednost ispunjeno, a ako je dočekano obedanje odbijeno onda **await** vrada vrednost odbijeno.

Primer. Prikaz (sa čekanjem) prvih pet slova, tako da prvo bude prikazano A, potom B i da nema drugih ograničenja na redosled (Zahtevana ograničenja su realizovana preko rezervisane reči **await**.)

```
function prikaziNisku(niska) {
    return new Promise((razresi, odbij) => {
        setTimeout(
            () => {
                console.log(niska);
                razresi();
            },
            Math.floor(Math.random() * 50) + 1
        );
    });
}
async function prikaziDvaRedom() {
    await prikaziNisku("A");
    await prikaziNisku("B");
    prikaziNisku("B");
    prikaziNisku("C");
    prikaziNisku("D");
}
```

Primer. Prikaz (sa čekanjem) prva tri slova po azbučnom redosledu (U ovom primeru je “ulančavanje” rezultata realizovano preko **await** naredbe, pri čemu su funkcije povratnog poziva date kao lambda-izrazi.)

```
function dodajNisku(prethodna, tekuca) {
    return new Promise((razresi, odbij) => {
        setTimeout(
            () => {
                razresi(prethodna + ' ' + tekuca);
            },
            Math.floor(Math.random() * 50) + 1
        );
    });
}
async function dodajSveRedom() {
    let ret = "";
    ret = await dodajNisku(ret, 'A');
    ret = await dodajNisku(ret, 'B');
    ret = await dodajNisku(ret, 'B');
}
dodajSveRedom();
```

Rezultat izvršenja skripte je prikaz prva tri slova azbuke u rastuđem redosledu: A B V.

12. Rukovanje greskama

Ovde de biti opisani mehanizmi otkrivanja grešaka i rukovanja greškama, oslanjajući se na koncept izuzetaka.

1. Hvatanje grešaka u programskom kodu. Primeri.

Preporučuje se da se u skripti uključi tzv. striktni mod.

Kada se uključi striktni mod, okruženje za izvršavanje kontroliše da li su deklarisanе promenljive koje se koriste u kodu.

Primer. Ilustruje kako u jeziku JavaScript mogu ostati sakrivene greške koje se odnose na nedeklarisane promenljive.

```
function gdeLiJeProblem() {  
    for (brojac = 0; brojac < 4; brojac++)  
        console.log("Срећа, срећа, радпсу!");  
}  
gdeLiJeProblem();
```

Kada se zaboravi da se deklarise promenljiva u JavaScriptu (kao što je ovde slučaj sa promenljivom brojac) JavaScript okruženje za izvršavanje nede prijaviti grešku, ved de u tišini kreirati **globalnu** promenljivu i programski kod de izvršiti prikaz na konzolu.

Dakle, rezultat rada gornjeg skripta je:

Срећа, срећа, радпсу! Срећа, срећа, радпсу! Срећа, срећа, радпсу! Срећа, срећа, радпсу!

Ako se uključi opcija za striktni mod, kao u kodu koji sledi:

```
function gdeLiJeProblem() {  
    "use strict";  
    for (brojac = 0; brojac < 5; brojac++)  
        console.log("Срећа, срећа, радпсу!");  
}  
gdeLiJeProblem();
```

Tada ce se prijaviti greska i rezultat ce biti:

```
for (brojac = 0; brojac < 5; brojac++)  
    ^
```

ReferenceError: brojac is not defined at gdeLiJeProblem...

Prilikom rada u striktnom modu, tada se vezivanja tipa funkcije koja se odnose na vrednost undefined nede pozivati kao metodi objekta. Međutim, ako se poziv te vrste izvršava u nestriktnom modu, tada de this predstavljati globalni objekat, kreiranjem i čitanjem globalne promenljive.

Primer. Ilustruje kako u jeziku JavaScript mogu ostati sakrivene greške koje se odnose na kreiranje objekata. Ovde je umesto da se napravi objekat, pristupljeno globalnom objektu, što de nadalje predstavljati problem.

```
function Osoba(ime) {  
    this.ime = ime;  
} // Greska, zaboravljen 'new' ?  
let mikiMaus = Osoba("Мики Маус");  
console.log(ime);
```

Dakle, rezultat rada gornjeg skripta je: Мики Маус

Međutim, ako se u striktnom modu metod konstruktor pozove na nekorektan način (bez operatora `new`), JavaScript okruženje de zaustaviti izvršavanje i prikazati poruku o grešci:

```
"use strict";
function Osoba(ime) {
    this.ime = ime;
}
// Greska, zaboravljen 'new' ?
let mikiMaus = Osoba("Мики Маус");
```

Greška de biti prijavljena kod naredbe kojom se pokušava očitati vrednost polja u okviru *this*.

```
this.ime = ime;
```

^

TypeError: Cannot set property 'ime' of undefined at Osoba

2. Reagovanje na greške. Primeri.

I u jeziku JavaScript se može signalizirati da je prilikom izvršavanja funkcije došlo do greške tako što de biti vradena “specijalna” povratna vrednost.

Primer. Ilustruje kako se u jeziku JavaScript može reagovati na grešku.

U ovom slučaju imamo funkciju na koju referiše promenljiva *slucajanBrojIliMiki* - ta funkcija nekada vrada pozitivan ceo broj manji ili jednak od 9, nekada negativan ceo broj vedi ili jednak -9, a nekada nisku Miki Maus. Neka druga funkcija računa kvadratni koren rezultata koji proizvede prva funkcija.

```
"use strict";
let slucajanBrojIliMiki = function () {
    if (Math.random() < 0.4)
        return Math.floor(Math.random() * 10);
    if (Math.random() < 0.8)
        return Math.floor(-Math.random() * 10);

    return "Miki Maus";
}
function kvadratniKoren(izvorPodataka) {
    let broj = Number(izvorPodataka());
    let rezultat = Math.sqrt(broj);

    return { "broj": broj, "rezultat": rezultat };
}
console.log(kvadratniKoren(slucajanBrojIliMiki));
```

Funkcija za računanje kvadratnog korena očekuje da se vrednost dobije izvršenjem funkcije koja joj se prosledjuje kao argument. Sama funkcija vrada objekat, čija osobina *broj* sadrži broj koji se korenuje, a osobina *rezultat* sadrži rezultat korenovanja. U ovom primeru ne preduzimaju se nikakve specijalne mere za detekciju greške, samo se prikaže rezultat rada funkcije.

Kada se detektuje greška u telu pozvane funkcije, jedna od opcija je da se postojanje greške signalizira tako što de naredbom `return` biti vradena neka specijalna vrednost koja signalizira da je došlo od greške. Ta specijalna vrednost može sadržati enkodiranu informaciju o tome šta je prouzrokovalo grešku.

Primer. Ilustruje kako se u jeziku JavaScript u okviru funkcije može reagovati na grešku vratanjem specijalne vrednosti.

```
"use strict";
let slucajanBrojIliMiki = function () { ...}
function kvadratniKoren(izvorPodataka) {
    let broj = Number(izvorPodataka());
    if (isNaN(broj)) {
        let rezultat = "nemoguće korenovati nesto sto nije broj";
        return { "broj": broj, "rezultat": rezultat };
    }
    if (broj < 0) {
        let rezultat = "nemoguće korenovati negativan broj";
        return { "broj": broj, "rezultat": rezultat };
    }

    let rezultat = Math.sqrt(broj);
    return { "broj": broj, "rezultat": rezultat };
}
console.log(kvadratniKoren(slucajanBrojIliMiki));
```

U ovom slučaju funkcija vrada objekat, čija osobina *broj* sadrži broj koji se korenuje, a osobina *rezultat* de biti broj ukoliko je operacija izvršena bez problema ili de biti niska sa opisom uzroka, u slučaju da se pojavila greška.

U scenariju kada se greška signalizira preko povratne vrednosti, uobičajeno je da, u slučaju da je detektovana greška, funkcija vrati vrednost **null** ili **undefined**.

Međutim, struktuiranje koda na ovakav način ima i određene **negativne efekte**, odnosno može dovesti do potencijalnih problema:

1. Šta de se dogoditi ako funkcija ved vrada sve mogude vrednosti iz domena? Za takvu funkciju bi bilo teško nadi "specijalnu" vrednost, koja de služiti za signalizaciju da je došlo do greške.
2. Korišćenje specijalnih "povratnih" vrednosti za signalizaciju greške može dovesti do toga da struktura programskog koda bude zagađena, teška za čitanje, razumevanje, nadogradnju i održavanje.

3. Izuzetci. Primeri (u 4. 5.).

Izuzetci su mehanizam koji omogućuje da programski kod koji se izvršava izbaci izuzetak - vrednost koja ukazuje da je došlo do problema.

Izbacivanje izuzetka donekle podseda na "pojačani" povratak iz funkcije. Naime, na taj način ne samo što tekuda funkcija završava rad, ved se iskače iz lanca njenih pozivaoca, tj. ide se naniže kroz stek poziva sve do onog poziva koji je inicirao to izvršavanje i koji "zna" kako da obradi izuzetak. Prethodno opisani proces se naziva **odmotavanje steka poziva**.

Dakle, izuzetak se spušta naniže i biva izbačen kroz sve kontekste poziva na steku.

Ako bi se izbačeni izuzetak spustio do polaznog poziva, izvršavanje bi se prekinulo, što programeru nije od velike pomodi. Mod izuzetaka leži u činjenici da se izuzetku prilikom spuštanja kroz stek poziva mogu postaviti elementi koji ga **hvataju** (obično i obrađuju) i na taj način prekidaju njegovo spuštanje naniže. Posle hvatanja i obrade izuzetka program de nastaviti rad od naredbe iza mesta gde je dati izuzetak uhvađen.

4. Izbacivanje izuzetka. Primeri.

Primer. Ilustruje kako se u jeziku JavaScript izbacuju izuzetci, na primeru korenovanja.

```
"use strict";
let slucajanBrojIliMiki = function () { ... }
function kvadratniKoren(izvorPodataka) {
  let broj = Number(izvorPodataka());
  if (isNaN(broj))
    throw new Error("nemoguće korenovati nešto što nije broj");
  if (broj < 0)
    throw new Error("nemoguće korenovati negativan broj");
  let rezultat = Math.sqrt(broj);
  return { "broj": broj, "rezultat": rezultat };
}
for (let i = 0; i < 10; i++) console.log(kvadratniKoren(slucajanBrojIliMiki));
```

Primer. Ilustruje kako se u jeziku JavaScript izbacuju izuzetci i kako se vrši odmotavanje steka.

```
"use strict";
let pravac = function () {
  if (Math.random() < 0.3)
    return "levo";
  if (Math.random() < 0.6)
    return "desno";
  if (Math.random() < 0.8)
    return "gore";
  return "dole";
}
function voziAuto(usmerenje) {
  let rezultat = usmerenje();
  if (rezultat.toLowerCase() == "levo")
    return "L";
  if (rezultat.toLowerCase() == "desno")
    return "R";
  if (rezultat.toLowerCase() == "gore")
    throw new Error("Auto ne leti: " + rezultat);
  if (rezultat.toLowerCase() == "dole")
    throw new Error("Auto nije krtica: " + rezultat);
  throw new Error("Nekorektno usmerenje za auto");
}
function pogled() {
  if (voziAuto(pravac) == "L")
    return "Sa ove strane se nalazi livada";
  else
    return "Sa ove strane su planine";
}
for (let i = 0; i < 10; i++)
  console.log(` ${i} Gledas iz auta. ${pogled()}`);

// 0 Gledas iz auta. Sa ove strane se nalazi livada      // 1 Gledas iz auta. Sa ove strane su planine
// Uncaught Error: Auto ne leti: gore at voziAuto (<anonymous>:18:11) at pogled (<anonymous>:24:7)
// at <anonymous>:30:40
```

5. Hvatanje izuzetaka. Primeri.

Primer. Ilustruje kako se u jeziku JavaScript izbacuju i hvataju izuzetci, na primeru korenovanja.

```
"use strict";
let slucajanBrojIliMiki = function () { ... }
function kvadratniKoren(izvorPodataka) {
let broj = Number(izvorPodataka());
if (isNaN(broj))
  throw new Error("nemoguce korenovati nesto sto nije broj");
if (broj < 0)
  throw new Error("nemoguce korenovati negativan broj");
let rezultat = Math.sqrt(broj);
return { "broj": broj, "rezultat": rezultat };
}
for (let i = 0; i < 10; i++)
  try {
    console.log(kvadratniKoren(slucajanBrojIliMiki));
  } catch (error) {
    console.log("Nesto je jako pogresno: *** " + error + " ***");
  }
```

Primer. Ilustruje kako se u jeziku JavaScript izbacuju i hvataju izuzetci i kako se vrši odmotavanje steka.

```
"use strict";
let pravac = function () { ... }
function voziAuto(usmerenje) {
  let rezultat = usmerenje();
  if (rezultat.toLowerCase() == "levo")
    return "L";
  if (rezultat.toLowerCase() == "desno")
    return "R";
  if (rezultat.toLowerCase() == "gore")
    throw new Error("Auto ne leti: " + rezultat);
  if (rezultat.toLowerCase() == "dole")
    throw new Error("Auto nije krtica: " + rezultat);
  throw new Error("Nekorektno usmerenje za auto");
}
function pogled() {
  if (voziAuto(pravac) == "L")
    return "Sa ove strane se nalazi livada";
  else
    return "Sa ove strane su planine";
}
for (let i = 0; i < 10; i++)
  try {
    console.log(` ${i} Gledas iz auta. ${pogled()}`);
  } catch (error) {
    console.log("Nesto je jako pogresno: *** " + error + " ***");
  }
```

6. Finalno pospremanje kod izuzetaka. Primeri.

Kao što je ved istaknuto, prilikom izbacivanja izuzetaka vrši se “odmotavanje” steka. Postavlja se pitanje: da li se prilikom odmotavanja može dodati u situaciju da konteksti (sadrzaji) funkcija koji se izvršavaju (koji se takođe čuvaju na steku) budu “izgubljeni”?

Primer. Ilustruje kako u jeziku JavaScript izbacivanje izuzetka unutar tela funkcije **može** dovesti do toga da podaci koje je ta funkcija postavila ne budu adekvatno pospremljeni.

Ovde funkcija *izvrsiSaKontekstom* treba da obezbedi da tokom svog izvršavanja promenljiva kontekst (koja je za nju globalna) sadrži datu vrednost. Naravno, na završetku svog rada, vrednost te globalne promenljive treba restaurisati na onu vrednost koju je ona imala na početku rada te funkcije.

```
"use strict";
let kontekst = null;
console.log(kontekst); // null
function izvrsiSaKontekstom(noviKontekst, teloFunkcije) {
  let stariKontekst = kontekst;
  kontekst = noviKontekst;
  let rezultat = teloFunkcije();
  kontekst = stariKontekst;
  return rezultat;
}
izvrsiSaKontekstom(-25, () => console.log(Math.sqrt(kontekst))); // Nan
console.log(kontekst); // null
try {
  izvrsiSaKontekstom(16, function (x) {
    if (kontekst < 0)
      throw new Error("Nemoguće izračunati koren negativnog broja!");
    console.log(Math.sqrt(kontekst)); // 4
  });
} catch (e) {
  console.log("Ignorise se izuzetak: " + e);
}
console.log(kontekst); // null
try {
  izvrsiSaKontekstom(-16, function () {
    if (kontekst < 0)
      throw new Error("Nemoguće izračunati koren negativnog broja!");
    console.log(Math.sqrt(kontekst));
  });
} catch (e) {
  console.log("Ignorise se izuzetak: " + e); // Ignorise se izuzetak: negativan...
}
console.log(kontekst); // - 16
```

Međutim, šta se dešava ukoliko dođe do izbacivanja izuzetka u telu funkcije koju izvršava funkcija *izvrsiSaKontekstom()*, a na koju referiše parametar *teloFunkcije*?

U tom slučaju poziv naredbe kojom se restauriše vrednost promenljive *kontekst* de nestati sa steka usled njegovog odmotavanja, pa nede biti moguda restauracija vrednosti promenljive *kontekst* na njenu originalnu vrednost.

Iza naredbe **try** može slediti **finally** blok, koji se može nadi umesto **catch** bloka, ili eventualno može slediti iza **catch** bloka.

Ovaj **finally** blok označava da se kod u njemu izvršiti iza pokušaja izvršenja **try** bloka **u svakom slučaju** - i ako se kod u **try** bloku normalno izvršio i ako je bio izbačen izuzetak tokom izvršavanja naredbi **try** bloka. Dakle, ako ima nešto šta treba "pospremiti", onda se kod za pospremanje obično smešta u **finally** blok.

Primer.

U ovom primeru, za razliku od prethodnog, nema više potreba da se vrednost rezultata dobijenog izvršavanjem funkcije na koju referiše parametar **teloFunkcije** smešta u posebnu promenljivu.

```
"use strict";
let kontekst = null;
console.log(kontekst);
function izvrsiSaKontekstom(noviKontekst, teloFunkcije) {
  let stariKontekst = kontekst;
  kontekst = noviKontekst;
  try {
    return teloFunkcije();
  } finally {
    kontekst = stariKontekst;
  }
}
try {
  izvrsiSaKontekstom(-16, function () {
    if (kontekst < 0)
      throw new Error("Nemoguće izračunati koren negativnog broja!");
    console.log(Math.sqrt(kontekst));
  });
} catch (e) {
  console.log("Ignorise se izuzetak: " + e);
}
console.log(kontekst); // null
```

Uočava se da de, čak i kada se naredba **return** izvrši direktno u **try** bloku, naredbe unutar **finally** bloka biti izvršene.

7. Selektivno hvatanje izuzetaka. Primeri.

Programski jezik **JavaScript** ne sadrži direktnu podršku za selektivno hvatanje izuzetaka. Drugim rečima, ili de se hvatati svi izuzetci, ili se nede hvatati nijedan. Stoga bi, na prvi pogled, programer mogao podrazumevati da je izuzetak koji je izbačen baš onaj izuzetak na koji se mislilo kada je pisan **catch** blok.

Međutim, to često ne biva slučaj - bivaju prekršene neke druge pretpostavke, ili se negde drugo pojavio **bag** zbog kog je izbačen izuzetak različit od onog kog je programer očekivao.

Stoga se sugerše i u **JavaScriptu** da se ne pravi rukovalac za obradu svih mogućih izuzetaka, osim u svrhu usmeravanja tih izuzetaka njihovom pravom rukovaocu (a i u tom slučaju pažljivo treba razmotriti kako realizovati koncept sakrivanja informacija).

U jednom od prethodnih primera koji se bavio usmeravanjem automobila, u slučaju svih neregularnih situacija je, bez obzira na uzrok koji je doveo do takve situacije, izbacivan objekat tipa **Error**. To je moglo dovesti do komplikacije prilikom hvatanja izuzetka i njegove dalje obrade.

Naravno, postoji mogućnost da se pri rukovanju analizira poruka pridružena izuzetku tokom njegovog kreiranja i na osnovu toga zaključi šta je uzrok izuzetka, ali takav pristup nije dobar:

- poruke su predviđene da ih čita krajnji korisnik, a ne da se na osnovu njih donose odluke o toku izvršenja programa
- pristup nije robustan zbog skrivenih zavisnosti - čim bi neko promenio (npr. preveo) poruku, program više ne bi radio ono za šta je predviđen.

Stoga, poželjno je da se u takvim slučajevima definišu novi tipovi izuzetaka i da se prilikom hvatanja i obrade oni identifikuju koriscenjem **instanceof**.

U programskom kodu koji sledi, ti novi tipovi *NeMozeIspodZemljeError* i *NeMozePrekoNebaError* su kreirani pomodu prototipova.

```
"use strict";
function NeMozeIspodZemljeError(message) {
    this.message = message;
    this.stack = (new Error()).stack;
}
NeMozeIspodZemljeError.prototype = Object.create(Error.prototype);
NeMozeIspodZemljeError.prototype.name = "NeMozeIspodZemljeError";

function NeMozePrekoNebaError(message) {
    this.message = message;
    this.stack = (new Error()).stack;
}
NeMozePrekoNebaError.prototype = Object.create(Error.prototype);
NeMozePrekoNebaError.prototype.name = "NeMozePrekoNebaError";

let pravac = function () { ... }
function voziAuto(usmerenje) {
    let rezultat = usmerenje();
    if (rezultat.toLowerCase() == "levo") return "L";
    if (rezultat.toLowerCase() == "desno") return "R";
    if (rezultat.toLowerCase() == "gore") throw new NeMozePrekoNebaError("Auto ne leti: " + rezultat);
    if (rezultat.toLowerCase() == "dole") throw new NeMozeIspodZemljeError("Auto nije krtica: " + rezultat);
    throw new Error("Nekorektno usmerenje za auto");
}

function pogled() { ... }
for (let i = 0; i < 50; i++)
    try { console.log(` ${i} Gledas iz auta. ${pogled()}`);
    } catch (error) {
        if (error instanceof NeMozeIspodZemljeError)
            console.log("Podzemlje: " + error + " ***");
        else if (error instanceof NeMozePrekoNebaError)
            console.log("Nebeski svod: " + error + " ***");
        else {
            console.log("Nesto je jako pogresno: *** " + error + " ***");
            throw error;
        }
    } }
```

8. Tvrdnje. Primeri.

Tvrdnje (engl. assertions) su alat za osnovnu **kontrolu "zdravlja" programa**, koji olakšavaju pronalazak bagova. Tvrdnje daju kompaktan način da se eksplicitno iskažu očekivanja (preduslovi) za uspešan rad delova sistema i istovremeno obezbeđuju da se program zaustavi odmah čim se utvrdi da eksplicitno iskazana očekivanja nisu ispunjena.

Drugim rečima, tvrdnje predstavljaju način da se obezbedi da greške dovode do prekida izvršavanja tamo gde je nastala greška, kako bi se sprečilo da ta greška u tišini proizvede besmislenu vrednost, koja bi mogla da izazove problem u nekom drugom delu sistema.

Za razliku od nekih drugih programskih jezika, JavaScript direktno ne podržava tvrdnje, ali se odgovarajući mehanizam može implemetirati oslanjajući se na izuzetke.

Primer. Ilustruje kako se u jeziku JavaScript mogu realizovati tvrdnje.

Funkcija *lastElement()*, koja vrada poslednji element niza, bi (kad ne bi bila postavljena tvrdnja) vratila undefined ako bi se kao argument prosledio prazan niz. S obzirom da nema mnogo smisla tražiti poslednji element praznog niza, sigurno se radi o nekoj vrsti greške i tu se može postaviti tvrdnja.

Analogna situacija je i kod funkcije *element()* koja vrada element niza na datoj poziciji. Ovde su postavljene tvrdnje da "niz ne sme biti prazan", da "indeks niza mora biti broj", da "indeks niza ne sme biti negativan" i da "indeks niza mora biti manji od broja članova".

```
"use strict";
function AssertionFailed(message) {
  this.message = message;
}
AssertionFailed.prototype = Object.create(Error.prototype);
function assert(test, message) {
  if (!test)
    throw new AssertionFailed(message);
}
function lastElement(array) {
  assert(array.length > 0, "niz ne sme biti prazan");
  return array[array.length - 1];
}
function element(array, index) {
  assert(array.length > 0, "niz ne sme biti prazan");
  assert(typeof(index) == Number, "indeks niza mora biti broj" )
  assert(index >= 0, "indeks niza ne sme biti negativan");
  assert(index < array.length, "indeks niza mora biti manji od broja članova");
  return array[index];
}
let niz1 = [];
let niz2 = ["Paja", "Miki", "Mini", "Silja"];
console.log(lastElement(niz2));           // Uncaught AssertionFailed {message: 'niz ne sme biti prazan'}
```

Potreban uslov za uspešan rad sa tvrdnjama koje su realizovane na ovaj način je da se nigde gde se rukuje sa izuzetcima ne vrši hvatanje izuzetaka tipa AssertionFailed, niti njegova obrada.

13. Moduli

1. Nativni ES5 moduli, preko funkcijskih izraza koji se odmah izvršavaju. Primeri.

Modularni koncept programiranja je trenutno jedan od najzastupljenijih koncepata u modernom JavaScript programiranju. Zasniva se na razbijanju jedne velike aplikacije na manje, enkapsulirane, nezavisne delove koda – **module**. Cilj modularnosti jeste da se smanji kompleksnost prema principu “podeli pa vladaj”.

Najznačajnije prednosti modularnog programiranja su:

1. preglednija i razumljivija aplikacija
2. lakše kontrolisanje opsega definisanosti promenljivih
3. sprečavanje da globalni opseg definisanosti bude “zagađen”
4. ponovna upotrebljivost koda
5. mogućnost rada na istom projektu više različitih timova ili programera koji rade odvojene manje zadatke
6. lakše debugiranje

U zavisnosti od verzije JavaScripta postoje različiti pristupi realizaciji modularnog programiranja, pa samim tim i različite sintakse koje omogućavaju modularno programiranje:

- Nativna ES5 sintaksa
- ES5 sintaksa sa spoljnim bibliotekama
- Sintaksa ES6 modula

Problem učitavanja modula nije trivijalan (moduli zavise od drugih modula, treba se efikasno realizovati, itd.) pa se za rešavanje ovog problema koriste posebni programi tzv. **alati za učitavanje modula** (engl. module loader) i **alati za uvezivanje modula** (engl. module bundler).

Nativni ES5 moduli

U vreme pisanja JavaScripta, modularno programiranje nije bilo planirano kao način programiranja, tako da JavaScript sve do verzije ES6 (još se označava i sa ES2015) nema ugrađenu podršku za module. U okviru nativnog ES5 se koriste razni **obrasci** (engl. pattern) koji imaju sličnosti sa modularnim programiranjem, ali koji nemaju baš sve karakteristike “čistokrvnog modularnog šablona”.

Efekat modula kod kojih je enkapsuliran njihov kod je u ovom slučaju mogude postidi na dva načina: korišćenjem funkcijskih izraza koji se odmah izvršavaju ili korišćenjem konstruktora.

U oba slučaja, osnovna ideja je ista: ono što treba da bude dostupno spoljašnjosti se vrada iz funkcije pomodu rezervisane reči **return**. Ovaj jednostavan šablon se može primenjivati bilo gde i ne zahteva korišćenje dodatnih biblioteka. Nadalje, u okviru jedne datoteke je, ako je to potrebno, mogude definisati više modula.

Pored navedenih prednosti ovaj pristup ipak ima i svojih mana:

1. “zagađivanje” globalnog opsega imena nazivima modula (mada je telo modula sakriveno u lokalni domen korišćenjem funkcija)
2. potreba za “ručnim” određivanjem redosleda učitavanja modula, što može biti prilično komplikovano kod velikih i kompleksnih aplikacija
3. nemogućnost asinhronog učitavanja modula

Funkcijski izrazi koji se odmah izvršavaju (engl. Immediately-Invoked Function Expressions - **IIFE** (izgovara se "ifi"), su funkcijski izrazi koji se izvršavaju odmah nakon stvaranja. Koriste se **funkcijski izrazi**, a ne deklaracije funkcija, zato što deklarirana funkcija ne može biti odmah pozvana u istoj naredbi dok funkcijski izraz može.

Primer. Slededa naredba (**function()**,- **()**); predstavlja funkcijski izraz koji se izvršava odmah po kreiranju.

Funkcijski izrazi koji se odmah izvršavaju se koriste za modularno programiranje, jer sav kod unutar ovakve funkcije ostaje privatn.

Korišćenjem funkcijskih izraza koji se odmah izvršavaju može da se izabere koje podatke/funkcije de se zadržati kao privatni, a koje podatke/funkcije de biti objavljene. Javno dostupni de biti oni delovi koda koji se vrate sa rezervisanom reči **return**. Na ovaj način se može u globalnom prostoru imena proslediti svi tipovi podataka.

Primer: potrebno je izvršiti proračunavanje (u ovom slučaju kvadriranje) nad datim argumentom. Pri tome, vrednost argumenta je enkapsulirana u modulu, isto kao i funkcija koja vrši proračunavanje.

U datoteci vrednost.js je u poseban modul izdvojeno enkapsuliranje vrednosti za podatakKojiSeCuva. Sam modul je, preko funkcijskih izraza koji se odmah izvršavaju, izložio metode za pristup *getPodatak()* i za promenu *setPodatak()*.

```
const vrednost = function () {  
    let podatakKojiSeCuva = ' '; // ovo je privatn podatak  
    function setPodatak(noviPodatak) { // funkcija za postavljanje vrednosti  
        podatakKojiSeCuva = noviPodatak;  
    }  
    function getPodatak() { // funkcija za ocitavanje vrednosti  
        return podatakKojiSeCuva;  
    }  
    return { // publikovanje "javnih "funkcija  
        setPodatak: setPodatak,  
        getPodatak: getPodatak  
    };  
}();
```

U modulu koji sadrži datoteka proracun.js je izložena funkcija *izracunajKvadrat()*.

```
const proracun = function () {  
    function izracunajKvadrat() {  
        let x = vrednost.getPodatak(); // pozvan je metod iz vrednost.js  
        return x * x; // ovde ide deo koda vezan za proracun  
    }  
    return { // publikovanje "javne" funkcije  
        izracunajKvadrat: izracunajKvadrat,  
    };  
}();
```

Da bi bilo obezbeđeno učitavanje modula, koristide se spoljašnji API veb pregledača. Izvršavanje počinje od datoteke *index.html*. Ovde je, na početku izvršavanja, potrebno izvršiti učitavanje modula. Redosled učitavanja modula zavisi od funkcionalnosti aplikacije, što je najteži deo za programera naročito kod velikih i kompleksnih aplikacija. U ovom primeru se prvo učitava *vrednost.js* jer se njegove metode koriste i u okviru *proracun.js*.

```

<html>
<head lang="en">
  <meta charset="UTF-8">
  <title>Modularna aplikacija</title>
</head>
<body>
  <script src="vrednost.js" type="text/javascript"></script>           <!-- učitavanje modula -->
  <script src="proracun.js" type="text/javascript"></script>
  <script type="text/javascript">                                     <!-- skripta koja koristi publikovane elemente modula -->
    let argument = 10;
    vrednost.setPodatak(argument);
    console.log(proracun.izracunajKvadrat());
  </script>
</body>
</html>

```

Kao što vidimo, dakle, učitavanje i “registracija” modula je realizovano izvršavanjem skripti koje sadrže programski kod modula. Promena redosleda učitavanja tj. promena redosleda `<script>` elemenata bi dovela do greške u izvršavanju JavaScript koda.

Kad se veb strana pogleda u veb pregledaču, ako je sve kako treba, kao rezultat rada de u konzoli pregledača biti upisan broj 100.

2. Nativni ES5 moduli, preko konstruktora. Primeri.

Konstruktor omogućava pravljenje **više objekata** na osnovu konstruktorske funkcije.

Primer. Ilustruje modularno programiranje preko konstruktora. I ovde je zadatak isti kao u prethodnom primeru: potrebno je izvršiti proračunavanje (kvadriranje) nad datim argumentom. Pri tome, vrednost argumenta je enkapsulirana u modulu, isto kao i funkcija koja vrši proračunavanje.

Izmene koda u odnosu na prethodni primer su **male** - telo funkcija ostaje nepromenjeno, a prilagođavaju se slededi elementi:

- kako se radi o konstruktoru, to naziv promenljive kojoj se dodeljuje funkcija treba da počinje velikim slovom;
- funkcijski izraz koji se odmah izvršava se transformiše u običnu funkciju, brisanjem zagrada koje odmah pozivaju funkciju, jer se standardna konstruktorska funkcija poziva sa rezervisanom reči **new**;
- unutar modula koji poziva metodu iz drugog modula potrebno je instancirati novi objekat, da bi bila dostupna njegova metoda.

U datoteci vrednost.js je u poseban modul izdvojeno enkapsuliranje vrednosti za podatakKojiSeCuva.

```

const Vrednost = function() {
  let podatakKojiSeCuva = "";           // ovo je privatan podatak
  function setPodatak(noviPodatak) {   // funkcija za postavljanje vrednosti
    podatakKojiSeCuva = noviPodatak; }
  function getPodatak() {              // funkcija za očitavanje vrednosti
    return podatakKojiSeCuva;  }
}

```

```

return {                                     // publikovanje "javnih" funkcija
  setPodatak: setPodatak,
  getPodatak: getPodatak
};

```

U modulu koji sadrži datoteka *proracun.js* je preko konstruktora izložena funkcija *izracunajKvadrat()*.

```

const Proracun = function () {
  function izracunajKvadrat() {
    let x = vrednost.getPodatak();           // pozvan je metod iz vrednost.js
    return x * x;                           // ovde ide deo koda vezan za proracun
  }
  return {                                  // publikovanje "javne" funkcije
    izracunajKvadrat: izracunajKvadrat,
  };
};

```

Kao i u prethodnom primeru, da bi bilo obezbeđeno učitavanje modula, koristide se spoljašnji API veb pregledača. I ovde izvršavanje počinje od datoteke *index.html*. I u ovom primeru je redosled učitavanja modula eksplicitno dat redosledom `<script>` elemenata: prvo učitava *vrednost.js*, pa potom *proracun.js*.

```

<html>
<head lang="en">
  <meta charset="UTF-8">
  <title>Modularna aplikacija</title>
</head>
<body>
  <script src="vrednost.js" type="text/javascript"></script>           <!-- učitavanje modula -->
  <script src="proracun.js" type="text/javascript"></script>
  <script type="text/javascript">                                     <!-- skripta koja koristi publikovane elemente modula -->
    let argument = 10;
    let vrednost = new Vrednost();
    vrednost.setPodatak(argument);
    let proracun = new Proracun();
    console.log(proracun.izracunajKvadrat());
  </script>
</body>
</html>

```

Kao rezultat rada JavaScript modularnog koda, u konzoli pregledača de biti upisan broj 100. Isto kao u prethodnom primeru, eventualna promena redosleda učitavanja tj. promena redosleda `<script>` elemenata bi dovela do greške u izvršavanju.

3. ES5 moduli preko spoljnih biblioteka. Asinhrona definicija modula. Primeri.

Upravo zbog nedostatka podrške za module u ES5, kreirane su spoljašnje sintakse koje jeziku daju nedostajuću funkcionalnost:

- Asinhrona definicija modula (engl. Asynchronous Module Definition - **AMD**)
- CommonJS moduli
- Univerzalna definicija modula (engl. Universal modul definition - **UMD**)

Asinhrona definicija modula, kao što joj i samo ime kaže, podržava asinhrono učitavanje modula što je pogodno za rad sa modulima u pregledaču.

CommonJS učitava module sinhrono i zbog toga se najčešće koristi za rad sa modulima na serverskoj strani u okruženju node.js. Iako nije planiran za rad na klijentskoj strani, tj. unutar pregledača, uz pomoć alata za uvezivanje modula je moguće prilagoditi CommonJS radu u veb pregledaču.

Univerzalna definicija modula je kompatibilna i sa AMD i sa CommonJS definicijom i koristi se uglavnom ukoliko ima potrebe da se isti modul učitava na serveru i u veb pregledaču.

Asinhrona definicija modula

Pod AMD sintaksom se podrazumeva dogovoreni skup pravila i specifikacija koje ukazuju kako treba da izgleda kod za kreiranje modula.

Osnova AMD sintakse je funkcija **define()**, koja preko prosleđenih argumenata definiše sam modul i zavisnosti od drugih modula.

Funkcija **define()** ima tri parametra:

- **id** – niska koja predstavlja naziv modula (bez ekstenzije) - ovaj parametar je opcionalan, tj. nije obavezan.
- **dependencies** – niz niski sa nazivima potrebnih modula ili relativnih putanja do svih modula koji su potrebni za uspešan rad datog modula. Ako je ovaj niz prazan, to znači da modul ne zavisi od drugih modula. Redosled u nizu je bitan jer definiše redosled učitavanja.
- **factory** – funkcija povratnog poziva koja kreira objekat-modul. Ukoliko modul zavisi od drugih modula, onda ti moduli od kojih ovaj zavisi moraju biti parametri ove funkcije. Ona vrednost (funkciju, objekat, itd) koji funkcija za kreiranje vrati kao rezultat de biti vrednost koju modul izvozi - tj. izlaže spoljnom svetu na korišćenje.

Treba istadi da je implementacija asinhronne definicije modula moguda tek uz pomoć alata za učitavanje modula, pa je pored korišćenja same sintakse potrebno učitati odgovarajući alat za učitavanje modula - ovde de se za tu svrhu koristiti **require.js**.

U slučaju kada se kao alat za učitavanje modula koristi require.js biblioteka, programeru je na raspolaganju funkcija **requirejs()**, koja omogućava izvršavanje programskog koda koji zavisi od modula. Funkcija **requirejs()** ima dva parametra:

- **niz niski** koji sadrži informacije (imena ili putanje) o modulima od kojih dati kod zavisi
- **funkciju povratnog poziva** koja de biti izvršena kada budu učitani svi moduli iz prethodnog niza (naravno, ako neki od tih modula zavisi od drugih, onda de ti drugi moduli biti učitani pre zavisnog).

Primer. Ilustruje modularno programiranje preko asinhronne definicije modula. Opet je zadatak isti kao u prethodnim primerima: potrebno je izvršiti proračunavanje (kvadriranje) nad datim argumentom. Pri tome, vrednost argumenta treba biti enkapsulirana u modulu, a isto tako i funkcija koja vrši izračunavanje.

S obzirom da de polazna tačka izvršavanja JavaSkript koda biti veb strana, onda je biblioteka require.js dovučena i smeštena u isti direktorijum u kom se nalaze sve ostale datoteke, a alat za učitavanje modula biva pokrenut korišćenjem elementa script sa atributom src koji ima vrednost putanje do te biblioteke i sa atributom data-main koji ima vrednost putanje do JavaSkript datoteke od koje počinje učitavanje modula. Za razliku od prethodnih slučajeva, ovde proces učitavanja realizuje biblioteka require.js, pa nema potrebe da

programer ručno podešava redosled učitavanja modula, niti može dodati do greške zato što je taj redosled pogrešan.

Modul *vrednost.js* ne zavisi od drugih modula, tako da argument zadužen za naziv modula (ili relativnu putanju do modula) ostaje prazan:

```
define([], function() {  
    let podatakKojiSeCuva = ' ';           // ovo je privatan podatak  
    function setPodatak(noviPodatak) {    // funkcija za postavljanje vrednosti  
        podatakKojiSeCuva = noviPodatak;  
    }  
    function getPodatak() {               // funkcija za očitavanje vrednosti  
        return podatakKojiSeCuva;  
    }  
    return {                              // publikovanje "javnih" funkcija  
        setPodatak: setPodatak,  
        getPodatak: getPodatak  
    };  
});
```

Modulu *proracun.js* je potreban modul *vrednost.js*, pa je stoga dodata relativna putanja do tog modula u niz. Pored toga, ubačeni modul je prosleđen kao argument funkcije:

```
define(['./vrednost'], function (vrednost) {  
    function izracunajKvadrat() {  
        let x = vrednost.getPodatak();    // pozvan je metod iz vrednost.js  
        return x * x;                    // ovde ide deo koda vezan za proracun  
    }  
    return {                             // publikovanje "javne" funkcije  
        izracunajKvadratAMD: izracunajKvadrat  
    };  
});
```

U gornjem modulu je odlučeno da prema spoljašnjosti bude izložena funkcija *izracunajKvadrat()*, pri čemu de joj se iz spoljašnjosti pristupati preko imena *izracunajKvadratAMD()*.

Modul *index.js* zavisi i od modula *vrednost.js* i od modula *proracun.js*, pa su argumentu koji sadrži niz zavisnosti dodate putanje do oba modula i funkcija povratnog poziva ima dva argumenta:

```
define(['./vrednost', './proracun'], function (vrednost, proracun) {  
    function pokreni() {  
        let argument = 10;  
        vrednost.setPodatak(argument);  
        console.log(proracun.izracunajKvadratAMD());  
    }  
    return {                             // publikovanje "javne" funkcije  
        pokreniAMD: pokreni,  
    };  
});
```

U prethodnom skriptu je, dakle, napravljen modul, u njemu napravljena funkcija *pokreni()* i ta funkcija je izložena za korišćenje pod imenom *pokreniAMD()*.

Izvršavanje programskog koda počinje od datoteke *index.html*. Sada, umesto više uzastopno učitanih skripti, imamo samo jednu koja učitava *require.js*, a on brine o učitavanju svih ostalih modula. Obezbeđivanje da se učitaju zavisni moduli pre izvršenja date JavaScript funkcije, postignuto je pomodu funkcije *requirejs()*:

```
<html>
<head lang="en">
  <meta charset="UTF-8">
  <title>Modularna aplikacija</title>
</head>
<body>
  <script data-main="index.js" src="./require.js"></script>           <!-- učitavanje modula -->
  <script type="text/javascript">                                     <!-- skripta koja koristi publikovane elemente modula -->
    // poziv publikovane funkcije
    requirejs(["index"], function (index) {
      index.pokreniAMD();
    });
  </script>
</body>
</html>
```

Ako je sve kako treba, prilikom pregleda ove veb strane u konzoli pregledača de biti upisan broj 100.

AMD sintaksa rešava najvede nedostatke koje u radu sa modulima iskazuje nativni ES5 pristup:

- ovde je ugrađen menadžment modula u sklopu alata za učitavanje koji zamenjuje “ručno” određivanje redosleda učitavanja
- globalni prostor imena je “zagađen” samo sa jednim imenom *require* umesto sa svim nazivima modula.

Međutim, i ovaj pristup ima određene **mane**:

- kod funkcije *define()* lista zavisnosti u nizu mora da se slaže sa listom argumenata prosleđenih funkciji, što može da bude teško kada ima puno zavisnosti.
- sintaksa zahteva da je ceo kod obavijen sa *define()* funkcijom, što zahteva dodatno uvlačenje koda.
- u slučaju korišćenja alata za učitavanje modula i protokola HTTP 1.1, pri učitavanju velikog broja puno malih JavaScript datoteka može da se javi problem sa performansama (ovo može da se prevaziđe korišćenjem HTTP/2 protokola ili korišćenjem alata za uvezivanje modula)

Smatra se da je uz AMD bolje koristiti alate za učitavanje nego alate za uvezivanje, jer tada asinhroni rad dolazi do izražaja i pokazuje svoj pun potencijal. Naime, pri učitavanju se dovlače samo potrebni moduli, dok se pri uvezivanju dovlači jedna velika datoteka koji obuhvata sve module.

4. ES5 moduli preko spoljnih biblioteka. CommonJS moduli. Primeri.

CommonJS sintaksa je prvenstveno planirala za koriscenje na serveru, za rad u sinhronom modu. Koriscenje alata za učitavanje modula (*SystemJS*) je logican izbor za rad na serveru.

Ukoliko imamo potrebu da CommonJS sintaksu koristimo u okruženju veb pregledaca, potrebno je da koristimo alate za uvezivanje (*browserify* ili *webpack*), koji mogu da prilagode ovu sintaksu radu u pregledacu. Za razliku od AMD-a, gde telo modula treba da bude obavijeno funkcijom, kod CommonJS-a nema nikakvog omotaca (smatra se da je svaka JS datoteka jedan modul).

Izvoz metoda se može izvršiti na dva načina:

1. dodeljivanjem svake metode pojedinačno objektu *module.export*:

```
module.export.imeMetodeZaSpoljasnost1 = imeMetodelzModula1  
module.export.imeMetodeZaSpoljasnost2 = imeMetodelzModula2
```

2. dodeljivanjem objekta sa zeljenim metodama objektu *module.export*:

```
module.export = {  
  imeMetodeZaSpoljasnost1 : imeMetodelzModula1,  
  imeMetodeZaSpoljasnost2 : imeMetodelzModula2  
}
```

Uvoz modula (koriscenje metoda iz drugih modula) se omogucava definisanjem nove promenljive i dodelom njene vrednosti koristeći funkciju *require()*. Ona za parametar ima nisku – relativnu putanju do modula koji se zahteva, a kao rezultat vraca referencu na zahtevani modul.

```
var promenljivaKojaReferiseNaDrugiModul = require('./drugiModul');
```

Uvezeni modul je samo **kopija** izvezene vrednosti, tj. kopija modula. Kopija vracena sa *require()* je prekinula vezu sa originalom.

Primer. Ilustruje modularno programiranje preko CommonJS. Zadatak je isti kao i u prethodnim primerima (ali u ovom primeru se kod izvrsava na serverskoj strani).

Modul *vrednost.js* izvozi funkcije *getPodatak()* i *setPodatak()*, a vrednost promenljive *podatakKojiSeCuva* ostaje sakrivena:

```
let podatakKojiSeCuva = ' '; // ovo je privatan podatak  
function setPodatak(noviPodatak) { // funkcija za postavljanje vrednosti  
  podatakKojiSeCuva = noviPodatak;  
}  
function getPodatak() { // funkcija za ocitavanje vrednost  
  return podatakKojiSeCuva;  
}  
exports.setPodatak = setPodatak; // publikovanje "javnih" funkcija  
exports.getPodatak = getPodatak;
```

Modul *proracun.js* zahteva funkcionalnost modula *vrednost.js* na koji referise promenljiva *vrednost*, koristi funkciju *getPodatak()* iz tog modula, te kreira i izvozi funkciju za kvadriranje, pod imenom *izracunajKvadratCommonJS()*:

```
const vrednost = require("./vrednost");  
function izracunajKvadrat() {  
  let x = vrednost.getPodatak(); // pozvan je metod iz vrednost.js  
  return x * x; // ovde ide deo koda vezan za proracun  
}  
exports.izracunajKvadratCommonJS = izracunajKvadrat;
```

Datoteka *index.js* je ulazna tacka za izvršenje programa:

```
const vrednost = require("./vrednost");  
const proracun = require("./proracun");  
const argument = 10;  
vrednost.setPodatak(argument);  
console.log(proracun.izracunajKvadratCommonJS());
```

Prilikom izvršavanja ovog skripta u okruženju node, na konzoli terminala će biti prikazan broj 100.

CommonJS pristup kao i AMD rešava problem “ručnog” upravljanja redosledom učitavanja modula i smanjuje “zagađenje” globalnog opsega definisanosti i to sa još jednostavnijom sintaksom nego kod AMD.

Međutim, ovaj pristup ima i svojih **mana**:

- sinhroni rad nije baš najbolji za pregledač i u tom kontekstu je obavezno je korišćenje alata za uvezivanje
- svaki modul je potrebno smestiti u jednu odvojenu datoteku
- ne podržava ciklične zavisnosti
- CommonJS ima dinamičku strukturu modula, koja se definiše tek u vreme izvršavanja koda, pa u nekim slučajevima nije lako ispratiti šta se u stvari izvozi sve dok se ne izvrši kod.

5. ES6 moduli. Primeri.

Uz ES6 stiže podrška za modularni sistem koja je ugrađena i u sam jezik – ECMAScript 6 moduli. Time je integrisana podrška za modularno programiranje:

- Ovaj pristup podržava module koji se smeštaju u datoteke – jedan modul po datoteci.
- Moduli su unikati, ili jedinstveni primerci, ili singlotni (engl. singleton) - svaki modul se izvršava samo jednom.
- ES6 moduli mogu da rade i sinhrono i asihrono.
- Podržana je ciklična zavisnost modula.
- Sintaksa je još kompaktnija od CommonJS.

ES6 moduli imaju statičku strukturu. Pošto je struktura modula nepromenjiva, obično je dovoljno da se pregleda kod da bi se shvatilo šta se gde uvozi. Ovo nije slučaj kod dinamičke strukture koja karakteriše CommonJS, gde je često potrebno da se programski kod izvrši da bi se videlo šta se uvozi. Stoga, kod ovog pristupa, eventualne greške mogu da se otkriju i u vreme prevodjenja (sa alatom za uvezivanje modula), jer se sve radnje koje se odnose na uvoz i izvoz modula određuju u trenutku uvezivanja modula.

Uvoz modula ima sledeće karakteristike:

- Deklaracija promenjive ili funkcije se u fazi prevođenja diže na početak (vrh) oblasti definisanosti. (to znaci da poziv funkcije može da se nadje pre naredbe uvoza te funkcije i neće doći do greske)
- Svi uvezeni elementi su nepromenjivi iz modula koji vrši uvoz. Svaka operacija dodeljivanja vrednosti za uvezeni element u okviru modula (koji je uvezao) bi prouzrokovala grešku tipa `TypeError`.
- Nije dozvoljeno korišćenje promenljivih u naredbi `import`. ES6 moduli su statični pa, izvršenje naredbe `import` ne sme da zavisi od nečega što je dobijeno u vreme izvršenja.
- Uslovno učitavanje modula samo sa ES6 sintaksom nije mogude. Razlog za pomenuto je to što `import` naredba uvek mora biti na skroz spoljašnjem nivou, pa se ne može nadi u bloku, samim tim ni u telu naredbe grananja.

ES6 ne pravi kopiju svojstva već deli vezu na to svojstvo. Ovo važi čak i za deljenje primitivnih svojstava (svojstava koja su tipa broj ili niska).

Standardni izvoz se postiže pomoću rezervisane reči **export** - njenim postavljanjem ispred deklaracije promenljive/funkcije.

Primer. Ilustruje standardni izvoz

```
export var foo = 1;  
export var foo = function () {};
```



```
export var bar;  
export let foo = 2;  
export let bar;  
export const foo = 3;  
export function foo () {}  
export class foo {}
```

Imenovani izvoz je vrsta izvoza kojom se na kraju modula definišu elementi koji se izvoze. Naziv je potekao iz činjenice da se za definisanje elemenata koji se izvoze koriste njihova imena. Nekom izvezenom elementu se može dati drugo ime korišćenjem ključne reči **as**.

Primer. Ilustruje imenovani izvoz

```
export {};  
export {foo};  
export {foo, bar};  
export {foo as bar};
```

Preporuka je da se pri izvozu nekog modula definiše deo modula koji se podrazumevano izvozi. Podrazumevani izvoz se definiše sa ključnom reči **default**. U jednom modulu je dozvoljen samo jedan podrazumevani izvoz. Izvezeni deo se u drugom modulu koristi pod imenom default.

Primer. Ilustruje podrazumevani izvoz

```
export default 42;  
export default {};  
export default [];  
export default (1 + 2);  
export default foo;  
export default function () {}  
export default class {}  
export default function foo () {}  
export default class foo {}
```

Moguće je i kombinovanje podrazumevanog i imenovanog izvoza

Primer:

```
export {foo as default};  
export {foo as default, bar};
```

Ukoliko modul ima priličan broj “razbacanih” elemenata koji se izvoze, preporuka je da se na vrhu modula jasno i pregledeno definiše API kao u sledecem primeru:

```
const api = { add, subtract, multiply, divide };           // calculator.js  
function add(a,b) {...}  
function subtract(a,b) {...}  
function multiply(a,b) {...}  
function divide(a,b) {...}  
function somePrivateHelper() {...}  
export default api;
```

Uvoz modula se realizuje korišćenjem rezervisane reči **import**, iza koje sledi spisak funkcija koje se uvoze (eventualno i njihovih alijasa), zatim ključna reč **from**, pa niska koja sadrži putanju do datoteke sa modulom. Za imenovani uvoz koristi se ključna rec **as**. Uvoz podrazumevanih vrednosti se vrši jednostavnim pozivanjem imena modula ili pozivanjem preko imena.

Primeri:

```
import {} from "foo";
import {bar, baz} from "foo";
import {bar as baz} from "foo";
import {bar as baz, xyz} from "foo";
import foo from "foo";
import {default as foo} from "foo";
```

Uvoz celog prostora imena modula, još se zove i globalni uvoz, se koristi ukoliko treba da se u jednoj naredbi učitaju svi izvezeni elementi jednog modula. Ovakav globalni uvoz kositi sintaksu *** as**.

Primer:

```
import * as bar from "foo"
//nakon ovog globalog uvoza dostupni su nam svi elementi modula foo u modulu pod nazivom bar
bar.x; bar.baz();
```

Nova ES6 sintaksa još uvek nije podržana od strane svih pregledača, pa je potrebno koristi alate (npr. Babel) da bi se transpilirala u ES5. Taj scenario je prihvatljiv i za izvršavanje u okviru pregledača i na serverskoj strani, samo što zahteva instalaciju dodatnih biblioteka.

Pored toga, mogude je izvršiti ES6 modularni kod i na platformi node bez instalacije dodatnih biblioteka. Za uspešno pokretanje ES6 modularnog koda na node platformi i bez dodatnih biblioteka-transpilatora (u ovom trenutku) potrebno je da:

- verzija okruženja node bude 12 ili više
- datoteke koje sadrže takav kod imaju ekstenziju mjs (a ne js kao što je to dosad bio slučaj)
- izvršavanje koda pokrede sa opcijom --experimental-modules

Primer. Ilustruje modularno programiranje preko ES6. Potrebno je izvršiti proračunavanje (kvadriranje) nad datim argumentom. U ovom primeru de se programski kod izvršavati na serverskoj strani.

→ Modul vrednost.mjs koriscenjem naredbe export izvozi funkcije getPodatak() i setPodatak():

```
let podatakKojiSeCuva = ""; // ovo je privatan podatak
const _setPodatak = function (noviPodatak) {
  podatakKojiSeCuva = noviPodatak;
};
export { _setPodatak as setPodatak };
const _getPodatak = function() {
  return podatakKojiSeCuva;
};
export { _getPodatak as getPodatak };
```

→ modul proracun.mjs uvozi funkciju getPodatak() iz modula vrednost.mjs, pa kreira i izvozi funkciju za kvadriranje, pod imenom izracunajKvadratES6()

```
import { getPodatak } from "./vrednost";
function izracunajKvadrat() {
  let x = getPodatak(); // pozvan je metod iz vrednost.js
  return x * x; // ovde ide deo koda vezan za proracun
}
export const izracunajKvadratES6 = izracunajKvadrat;
```

→ datoteka *index.mjs* je ulazna tacka za izvršavanje programa, tu se uvozi funkcija `setPodatak()` i funkcija `izracunajKvadratES6()`, a potom se one koriste za postavljanje vrednosti argumenata i izracunavanje:

```
import { setPodatak } from './vrednost';
import { izracunajKvadratES6 } from './proracun';
let argument = 10;
setPodatak(argument);
console.log(izracunajKvadratES6());
```

→ Pokretanje skripte uz dodatne parametre u node okruženju:

```
node --experimental-modules index.mjs
```

→ Dobija se rezultat:

```
(node:14480) ExperimentalWarning: The ESM module loader is experimental.
```

```
100
```

6. Alati za učitavanje modula.

Ukoliko se programer odluči za modularni način programiranja aplikacije, on de se susresti sa problemom organizovanja učitavanja potrebnih modula, kao i modula od kojih su oni zavisni (engl. dependencies). Alati za učitavanje (engl. modul loader) i za uvezivanje modula (engl. modul bundler) su neophodna podrška programeru za jednostavniji i efikasniji rad sa modulima.

Alati za učitavanje modula omogućavaju dinamičko učitavanje modula vodeći računa o rasporedu učitavanja, kako bi se zavisni moduli ranije učitali. Najpoznatiji alati za učitavanje modula su:

- RequireJS – implementira AMD modularni sistem.
- SystemJS – je univerzalni alat za učitavanje, koji može učitavati module u bilo kom popularnom formatu (CommonJS, UMD, AMD, ES6).

Smatra se da je uz AMD bolje koristiti alate za učitavanje nego alate za uvezivanje, jer tada asinhroni rad dolazi do izražaja i pokazuje svoj pun potencijal. Naime, pri učitavanju se dovlače samo potrebni moduli, dok se pri uvezivanju dovlači jedna velika datoteka koji obuhvata sve module.

7. Alati za uvezivanje modula.

Alati za uvezivanje modula rešavaju problem tako što kompajliraju sve module u jednu datoteku prema određenom redu, vodeći računa o tome da neki modul od kojeg zavisi drugi bude učitao na vreme.

Najpoznatiji alati za uvezivanje modula:

- Browserify – implementira CommonJS i u okruženju veb pregledača. Može da se nadograđuje raznim dodacima i uz pomoć izvršilaca zadataka (engl. task runner) (najčešće su to alati Gulp ili Grunt) može da izvrši različite poslove.
- Webpack – može učitavati module u bilo kom popularnom formatu (CommonJS, UMD, AMD, ES6), dolazi kao jedan paket i nisu mu potrebni dodaci. Pored učitavanja modula može da izvrši posao transpilovanja ES6 u ES5, kao i transpilovanje SASS u CSS.

Ukoliko imamo potrebu da CommonJS sintaksu koristimo u okruženju veb pregledaca, potrebno je da koristimo alate za uvezivanje, koji mogu da prilagode ovu sintaksu radu u pregledacu.

14. JavaScript programiranje koriscenjem okruzenja node

1. Okruzenje node.

node.js je višeplatformsko radno okruženje (engl. runtime) koje se bazira na Google Chrome V8 mašini - JavaScript izvršnom okruženju visokih performansi, napisanom u C++ (prvobitno za potrebe veb pregledača Google Chrome).

Node ima jedinstvenu prednost jer milioni programera koji pisu JavaScript za pregledac, pored klijentske strane (za sta je JavaScript primarno bio namenjen), sada mogu da pisu i serversku stranu, a da ne moraju da koriste potpuno novi programski jezik.

2. Karakteristike okruzenja node

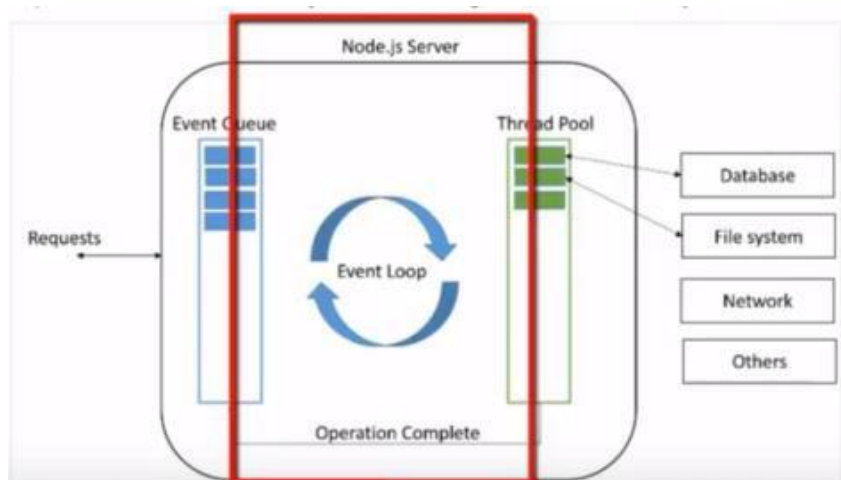
Programer Rajan Dal je razvio node.js 2009. godine, na mašini V8 kao serversko radno okruženje.

Velika prednost node.js je to što je on **u potpunosti JavaScript**. Ranije je JavaScript bio korišćen isključivo za razvoj na klijentskoj strani ili tzv. prednjem kraju (engl. frontend) - bilo kao "čisti" JavaScript (engl. Vanilla JS) bilo kao JQuery, ili neka druga JavaScript biblioteka. Od pojave okruženja node.js, JavaScript jezik se široko koristi i za programiranje na serverskoj strani, odnosno na zadnjem kraju (engl. backend). Pošto se radi o istom jeziku, ako se izabere ovaj pristup, razlika u samoj sintaksi između klijentskih i serverskih programa je vrlo mala ili čak ne postoji.

Druga velika prednost node.js je **modularnost** - koncept proširenja funkcionalnosti samog node.js i pojednostavljivanje zadataka instaliranjem određenih paketa, koji predstavljaju module.

Instalacija modula vrši se pomoću alata koji se zove **npm** – upravljač paketima okruženja node (engl. node package manager).

Osnovu node.js čini paradigma asinhronog programiranja, odnosno programiranje upravljano **događajima**. Događaji u JavaScriptu mogu biti klik mišem, pritisak na taster, zahtev za nekim resursom na mreži, itd. Okruženje node.js čini modnim upravo upotreba programskog modela baziranog na događajima, takozvani neblokirajući model, čiji je princip rada opisan na dijagramu koji sledi.



Princip rada neblokirajućeg modela odvija se pomoću jedne niti u node.js:

- 1) Kada node.js primi zahtev od strane klijenta, taj zahtev se smešta u red događaja
- 2) Ukoliko zahtev ne sadrži neku blokirajuću operaciju (npr. operacija sa datotekom, operacija za resursom sa mreže, rad sa bazom podataka i sl.), okruženje node.js de jednostavno obraditi zahtev i rezultat vratiti klijentu.
- 3) Ukoliko zahtev sadrži blokirajuću operaciju, okruženje node.js de taj zahtev dodati skupu niti (engl. threading pool), kako bi bila realizovana blokirajuća operacija. Često je blokirajuća operacija povezana sa funkcijom povratnog poziva, kojom se specificira šta treba uraditi po završetku blokirajuće operacije.
- 4) Kada je blokirajuća operacija završena, radna nit (engl. worker thread) priprema odgovor i šalje ga petlji za događaje, koja taj odgovor šalje klijentu. Ako je postojao povratni poziv pridružen blokirajućoj operaciji, sada nastupa vreme kada je omogućeno njegovo izvršenje.

Na ovaj način izvršavanje glavnog programa nije blokirano od strane blokirajućih operacija. Kao što je malopre opisano, okruženje node.js koristi petlju za događaje u jednoj niti (engl. Single Thread Event Loop) za naš programski kod, dok se sve ostalo izvršava paralelno. Zato ovo okruženje odlikuju visoke performanse, naročito kod modernih veb aplikacija gde se zahteva velika brzina prilikom postojanja velikog broja simultanih zahteva od strane korisnika.

3. Menadžer paketa npm. Primeri.

Kao što je ved istaknuto modularnost je velika prednost okruženja node.js. Modularnost podrazumeva nadogradnju node.js, inkorporaciju određenih funkcionalnosti u vidu programskog koda koji proširuju mogućnosti node.js. Kao što je opisano u delu koji se odnosio na module, oni se mogu posmatrati kao neka vrsta JavaScript biblioteke, a zapravo predstavljaju skup razvijenih funkcija koje programer želi uključiti u aplikaciju. Postoje **ugrađeni moduli**, a takođe postoje i moduli - **paketi**, koje programer instalira po potrebi.

Moduli se instaliraju pomoću alata **upravljač paketima okruženja node** tj. **npm**. Alat, tj. aplikacija npm je napisana u JavaScriptu i ona na osnovu zadatih parametara pretražuje bazu modula, tj. registar modula. Ako modula nema u lokalnom registru, alat de dovede modul sa registra na Internetu i instalirati ga u node.js okruženje. Počev od verzije 5+, alat npm automatski dodaje module u listu zavisnosti koja se nalazi u datoteci package.json.

Okruženje node.js u ovom trenutku podrazumevano koristi CommonJS sintaksu za uvoz i izvoz modula, pa se unutar programskog koda moduli uvoze pomoću funkcije require().

Ovde su pobrojani neki od najvažnijih node.js modula:

- assert - obezbeđuje skup testova.
- buffer - vrši operacije nad binarnim podacima. Omogućava interakciju sa TCP tokom podataka.
- child_process - omogućava rad sa procesom-detetom.
- cluster – omogućava deljenje jednog node.js procesa u više manjih procesa, čime se omogućava upotreba sistema sa više jezgara.
- command line – naredbe node.js se mogu koristiti iz terminala.
- C/C++ module – omogućava da se C/C++ koristi kao i svaki drugi modul.
- crypto – upravlja OpenSSL kriptografskim funkcijama.
- **dgram** – omogućava implementaciju UDP datagram soketa.
- debugger – omogućava debugiranje u node.js datoteci.
- dns – omogućava veze sa DNS serverima.
- errors - modul omogućava obradu grešaka.

- **events** - modul omogućava obradu događaja.
- **fs** – upravlja fajl sistemom.
- **http** – funkcionalnosti http servera.
- **https** - omogućava http pomodu TLS/SSL protokola.
- **modules** - omogućava sistem za učitavanje modula za node.js.
- **net** – omogućava kreiranje servera i klijenta.
- **os** – omogućava informacije i pristup operativnom sistemu.
- **path** – vrši operacije nad putanjama datoteka.
- **querystring** – omogućava formatiranje URL adrese.
- **readline** - omogućava interfejs za čitanje informacija iz toka podataka.
- **repl** – omogućava kreiranje komandne linije.
- **stream** – omogućava upravljanje podacima toka.
- **string_decoder** – dekodira bafer objekte u nisku.
- **timers** – omogućuje izvršavanje funkcije posle zadate vrednosti u milisekundama.
- **tls** – implementira TLS i SSL protokole.
- **tty** – omogućava klase korišćene od strane terminala teksta.
- **url** – raščlanjuje URL niske.
- **util** - pristup korisnim funkcijama, omogućava podršku za različite aplikacije i module.
- **V8** – Pristup informacijama o V8 mašini za izvršavanje JavaScripta.
- **vm** – omogućava kompajliranje JavaScript koda na virtualnoj mašini V8.
- **zlib** - kompresija i dekompresija datoteka, pomodu modula Gzip.

(podebljani su oni moduli sa kojima ćemo kasnije više raditi)

4. Događaji kod okruženja node. Primeri.

Poznato nam je da je node.js aplikacija koja se izvršava u jednoj niti i koja podržava konkurentnu obradu kroz koncept događaja i povratnih poziva. Svaki od API-ja u okviru node.js je asinhroni, a kako se izvršava u aplikaciji sa jednom niti, za to se koriste *async* funkcije u cilju obezbeđivanja konkurentnosti.

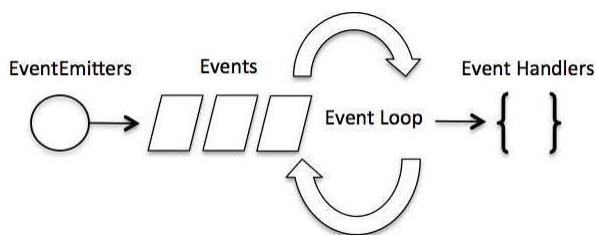
Okruženje node koristi obrazac dizajna posmatraca (observer pattern). Dakle, jedna nit node aplikacije se vrti u okviru petlje za događaje, sve dok se zadatak ne završi, a potom ispaljuje odgovarajući događaj kojim se signalizira da treba da bude izvršena funkcija koja osluškuje taj događaj.

```
const Događaj = require('events');
class EmiterDogađaja extends Događaj {}
const emiter = new EmiterDogađaja();
emiter.on('događaj', () => console.log('Odgovaram na emitovani događaj!'));
emiter.emit('događaj');
emiter.emit('događaj');
```

5. Programiranje upravljano događajima. Primeri.

Okruženje node.js koristi događaje i to je jedna od prednosti ovog okruženja u odnosu na slične tehnologije (dosta je brze). Pri pokretanju servera koji koristi node.js, odmah čim se inicijalizuju promenljive i deklarise funkcije, programski kod servera samo čeka da se dogode događaji, na čije je osluškivanje pretplacen – kada se dogodi događaj datog tipa, pokrene se funkcija koja rukuje sa takvim događajem.

Kod aplikacija upravljanih događajem, u opstem slučaju, postoji jedna glavna petlja u kojoj se oslušuju događaji i potom pokreću funkcije povratnih poziva kada se detektuju događaji odgovarajućeg tipa.



Iako događaji izgledaju slično kao povratni pozivi, razlika je u tome što se povratni pozivi desavaju kada asinhrona funkcija vrati rezultat, a rukovaoci događajima se pokreću u skladu sa načinom rada obrasca dizajna posmatraca (kod događaja se funkcije, koje oslušuju događaje, ponasaju kao posmatraci: kad događaj bude ispaljen, oslušivac događaja počinje sa izvršavanjem).

Okruženje node.js obezbeđuje podršku za rad sa događajima kroz modul *events* i kroz klasu *EventEmitter*. Kod aplikacija koje rade u node.js okruženju, svaka asinhrona funkcija ima kao poslednji parametar funkciju povratnog poziva, koja će biti izvršena po okončanju ove asinhronne funkcije.

6. Klasa EventEmitter. Primeri.

Klasa *EventEmitter* se nalazi u modulu *events*. Ova klasa obezbeđuje da objekti tipa *EventEmitter* sadrže osobine kao što su **on** i **emit**. Osobina **on** se koristi radi vezivanja funkcije za dati događaj, a osobina **emit** služi za ispaljivanje događaja.

Najvažniji metodi za objekte tipa *EventEmitter* su:

- *addListener(događaj, oslušivac)* - dodaje funkciju *oslušivac()* na kraj niza oslušivaca datog događaja *događaj*. Prilikom dodavanja ne vrši se provera da li je taj *oslušivac* ranije već bio dodan u niz oslušivaca datog događaja. Funkcija vraća emiter, tako da se može koristiti u ulančanim pozivima.
- *on(događaj, oslušivac)* - dodaje funkciju *oslušivac()* na kraj niza oslušivaca datog događaja *događaj*. Prilikom dodavanja se ne vrši provera da li je taj *oslušivac()* ranije već bio dodan u niz oslušivaca datog događaja. Funkcija vraća emiter, tako da se može koristiti u ulančanim pozivima.
- *once(događaj, oslušivac)* - jednokratno dodaje funkciju *oslušivac()* na kraj niza oslušivaca datog događaja *događaj*. Ovaj oslušivac će se izvršiti jednom po ispaljivanju događaja, posle čega će biti uklonjen iz niza oslušivaca. Funkcija vraća emiter, tako da se može iskoristiti u ulančanim pozivima.
- *removeListener(događaj, oslušivac)* - uklanja *oslušivac()* iz niza oslušivaca za dati događaj *događaj*. Ova funkcija će ukloniti ne više od jedne pojave datog oslušivaca iz niza, pa ako je prethodno oslušivac bio dodan više puta, onda ova funkcija neće ukloniti sve njegove pojave. Funkcija vraća emiter, tako da se može koristiti u ulančanim pozivima.
- *RemoveAllListeners([događaj])* - uklanja sve oslušivace ako je ispusten argument koji predstavlja događaj. Ako je prilikom poziva funkcije prosledjen događaj, onda će biti uklonjeni svi oslušivaci za dati događaj. Funkcija vraća emiter, tako da se može koristiti u ulančanim pozivima.
- *setMaxListeners()* - podrazumevano ponasanje je da objekat tipa *EventEmitters* šalje upozorenje ako je broj njegovih oslušivaca postao veći od 10. Ova funkcija menja to ponasanje, tako što na *n* postavlja granicnu vrednost za upozoravanje na previše oslušivaca. Ako se korišćenjem ove funkcije maksimalni broj oslušivaca postavi na 0, upozorenje se neće pojavljivati.
- *Listeners(događaj)* - vraća niz oslušivaca za dati događaj, prosledjen kao argument funkcije

- `emit(dogadjaj, *arg1+, *arg2+, *...+)` - emituje dogadjaj sa datim argumentima, sto ce dovesti do izvršavanja svih osluskivaca tog dogadjaja, pri cemu ce u tom izvršavanju argumenti poziva biti prosledjeni rukovaocima dogadjaja. Funkcija vraca `true` ako dogadjaj *dogadjaj* ima osluskivace, inace vraca `false`.

Primer. Ilustruje rukovanje greskama kod dogadjaja

```
const EventEmitter = require('events');
class EmitterDogadjaja extends EventEmitter {}
const emitter = new EmitterDogadjaja();
emitter.on('dogadjaj', () =>{
    console.log('A');
});
emitter.on('error', (err) => {
    console.error(`Paznja! doslo je do greske. Greska: ${err}`);
});
emitter.emit('dogadjaj'); // A
emitter.emit('error'); // Paznja! doslo je do greske. Greska: undefined
emitter.emit('dogadjaj'); // A
```

I nad dogadjajima se mogu praviti dogadjaji – preciznije, dva tipa dogadjaja:

- *NewListener* – emituje se svaki put kada se doda osluskivac za neki dogadjaj
- *RemoveListener* – emituje se svaki put kada osluskivac bude uklonjen

Primer. Ilustruje rad sa dogadjajem *newListener*

```
const EventEmitter = require('events');
class EmitterDogadjaja extends EventEmitter {}
const emitter = new EmitterDogadjaja();
emitter.once('newListener', (event, listener) => {
    if (event === 'dogadjaj') {
        emitter.on('dogadjaj', () => { // Ubaci novi dogadjaj na pocetak
            console.log('B');
        });
    }
});
emitter.on('dogadjaj', () =>{
    console.log('A');
});
emitter.emit('dogadjaj'); // B A
emitter.emit('dogadjaj'); // B A
```

Primer: emitovanje i jednostruko hvatanje dogadjaja

```
const KlasaZaEmitovanjeDogadjaja = require('events');
class EmitterDogadjaja extends KlasaZaEmitovanjeDogadjaja {}
const emitter = new EmitterDogadjaja();
let m = 0;
emitter.on('dogadjaj1', () =>{ console.log(++m); });
```



```

emitor.emit('dogadjaj1'); // 1
emitor.emit('dogadjaj1'); // 2
emitor.emit('dogadjaj1'); // 3
let n = 0;
emitor.once('dogadjaj2', () =>{
  console.log(++n);
});

emitor.emit('dogadjaj2'); // 1
emitor.emit('dogadjaj2'); //

```

7. Rad sa datotekama. Direktan rad sa datotekama. Primeri.

Primer. Ilustruje citanje sadržaja datoteke i njen prikaz na konzolu.

Datoteka *test.txt* treba da se nadje u istom direktorijumu u kom se nalazi ova skripta.

```

const fs = require('fs')
fs.readFile('test.txt', (err, data) => {
  if (err) {
    console.log(err);
  }
  console.log(data);
});

```

Prilikom izvršenja skripte, na konzoli se pojavio sledeći sadržaj:

```
<Buffer 54 72 6c 61 0d 0a 20 20 62 61 62 61 20 0d 0a 20 20 20 20 6c 61 6e 0d 0a 0d 0a 44 61 20 6a 6f 6a 20
70 72 6f c4 91 65 20 64 61 6e 21>
```

Podaci koji su poslati da se prikazu na konzoli su sekvenca bajtova, koja predstavlja kodove slova.

Primer. Ilustruje citanje sadržaja tekstualne datoteke i njen prikaz na konzolu.

Datoteka *test.txt* treba da se nadje u istom direktorijumu u kom se nalazi ova skripta.

```

const fs = require('fs')
fs.readFile('test.txt', 'utf-8', (err, data) => {
  if (err) {
    console.log(err);
  }
  console.log(data);
});

```

Razlika u odnosu na prethodni primer je u tome što je prilikom citanja datoteke specificiran i kodni raspored.

Sada se prilikom izvršenja skripte, na konzoli pojavio sledeći sadržaj:

„Trla

Baba

Ian

Da joj prođe dan!“

To je isti sadržaj tekstualne datoteke *test.txt* koji mogao da se vidi pomoću bilo kog editora.

Primer. Ilustruje upis teksta u tekstualnu datoteku.

```
const fs = require('fs');
```

```
fs.writeFile('text3.txt', 'Ovo je neka mala proba', 'utf8', (err) =>{
    if(err){
        console.log(err);
    }
});
```

Primer. Ilustruje sinhrono, tj. blokirajuće citanje sadržaja tekstualne datoteke, kao i sinhroni upis u datoteku.

```
const fs = require('fs');
let mojCitac = fs.readFileSync('test.txt', 'utf8');
console.log(mojCitac);
let mojPisac = fs.writeFileSync('test2.txt',
`Ovo je proba!
I treba pokušavati!
`,
'utf8');
```

8. Rad sa datotekama. Rad sa datotekama preko tokova. Primeri.

Primer. Ilustruje citanje sadržaja velike tekstualne datoteke, koriscenjem tokova.

```
const fs = require('fs');
console.log("\n");
let tokZaCitanje = fs.createReadStream('lorem.txt');
tokZaCitanje.setEncoding('utf8');
tokZaCitanje.on('data', (prispeliPodaci) => console.log(prispeliPodaci));
```

Primer. Ilustruje prebrojavanje koliko je bilo citanja pri ocitavanju sadržaja velike tekstualne datoteke, koriscenjem tokova

```
const fs = require('fs');
console.log("\n");
let tokZaCitanje = fs.createReadStream('lorem.txt');
tokZaCitanje.setEncoding('utf8');
let brojac = 0;
tokZaCitanje.on('data',
    () => {
        brojac++;
        console.log(brojac);
    });
tokZaCitanje.on('end',
    () => console.log("---\n" + brojac));
```

Primer. Ilustruje upis u tekstualnu datoteku, koriscenjem tokova.

```
const fs = require('fs');
let tokZaUpis = fs.createWriteStream('copy1.txt');
tokZaUpis.write('Ппздрав за слушапце курса УВИТ!');
```

Primer. Ilustruje nadovezivanje tokova.

```
const fs = require('fs');
let tokZaCitanje = fs.createReadStream('lorem.txt');
tokZaCitanje.setEncoding('utf8');
let brojac = 0;
tokZaCitanje.on('data', () => brojac++);
tokZaCitanje.on('end', () => console.log(brojac));
let tokZaUpis = fs.createWriteStream('copy1.txt');
tokZaCitanje.pipe(tokZaUpis);
```

Nakon izvršenja ovog koda, sadržaj datoteke *copy1.txt* će biti tekst koji je procitan iz datoteke *lorem.txt*.

`tokZaCitanje.pause()` → pauzira citanje

9. Rad sa datotekama. Događaji i tokovi podataka. Primeri.

Primer. Ilustruje postavljanje više osluskivaca na jedan događaj toka za citanje

```
const fs = require('fs');
let tokZaCitanje = fs.createReadStream('lorem.txt');
tokZaCitanje.setEncoding('utf8');
let brojac = 0;
tokZaCitanje.addListener('data', brojiCitanja);
tokZaCitanje.addListener('data', prikazujeCitanja);
function brojiCitanja(prispeliPodaci) {
    brojac = brojac + 1;
    console.log("Citanje broj: " + brojac);
}
function prikazujeCitanja(prispeliPodaci) {
    console.log('Duzina prispelih podataka: ' + prispeliPodaci.length);
}
tokZaCitanje.addListener('end',
    function () {
        console.log("---\nUkupno citanja: " + brojac);
    });
```

Rezultat izvršavanja ove skripte je:

```
Citanje broj: 1
Duzina prispelih podataka: 65536
Citanje broj: 2
Duzina prispelih podataka: 65536
...
Citanje broj: 20
Duzina prispelih podataka: 65536
Citanje broj: 21
Duzina prispelih podataka: 49786
---
Ukupno citanja: 21
```

Primer. Ilustruje dinamičko uklanjanje svih osluskivaca za događaj *data* tokom procesa citanja podataka iz datoteke preko tokova.

```
const fs = require('fs');
let tokZaCitanje = fs.createReadStream('lorem.txt');
tokZaCitanje.setEncoding('utf8');
let brojCitanja = 0;
let brojCitanja = (prispeliPodaci) => {
    brojCitanja = brojCitanja + 1;
    if (brojCitanja == 3)
        tokZaCitanje.removeAllListeners('data');
};
tokZaCitanje.addListener('data', brojCitanja);
tokZaCitanje.addListener('data', (prispeliPodaci) => console.log('дужина приспелих ппдауака: ' +
                                                                prispeliPodaci.length));
tokZaCitanje.addListener('end', () => console.log(brojCitanja));
```

Prilikom izvršavanja bi se dobio izlaz sledećeg oblika:

дужина приспелих ппдауака: 65536

дужина приспелих ппдауака: 65536

дужина приспелих ппдауака: 65536

3

U trenutku kada je *brojObracanja* dostigao vrednost 3, izvršeno je uklanjanje svih osluskivaca koji osluskuju *data* događaj.

15. JavaScript serversko programiranje koriscenjem okruzenja node

1. JavaScript serversko programiranje korišćenjem okruženja node. Mrežne aplikacije.

Sa funkcionalnog stanovista, internet se definise preko **usluga** koje nudi svojim korisnicima. Internet je mrezna infrastruktura koja omogucava rad distribuiranim aplikacijama koje korisnici koriste. Ove aplikacije ukljucuju:

- Veb (World Wide Web) - omogucava korisnicima pregled hipertekstualnih dokumenata
- elektronsku postu (e-mail)
- prenos datoteka (ftp, scp) izmedju racunara
- upravljanje racunarima na daljinu preko prijavljivanja na udaljene racunare (telnet, ssh)
- slanje instant poruka (im)...

Vremenom se gradi sve veci i veci broj novih aplikacija. One medjusobno komuniciraju preko svojih specificnih aplikacionih protokola (npr. HTTP, SMTP, POP3, ...). Svi aplikacioni protokoli komuniciraju koriscenjem dva transportna protokola:

TCP - (engl. Transmision Control Protocol) protokol sa uspostavljanjem konekcije, garantuje da ce podaci biti dostavljeni ispravno, u potpunosti i u redosledu u kome su poslali

UDP - (engl. User Datagram Protocol) protokol bez uspostavljanja konekcije i ne daje nikakve garancije o dostavljanju.

2. JavaScript serversko programiranje korišćenjem okruženja node. TCP aplikacije.

Primer. Ilustruje jednostavno mrezno programiranje preko TCP protokola koriscenjem modula *net*.

U ovom primeru je napravljen jednostavan server, koji svaki put kada klijent uspostavi vezu sa njim, salje pozdravnu poruku.

Programski kod servera se nalazi u datoteci *server.js*:

```
let net = require('net');
let server = net.createServer(
  (socket) => {
    socket.write(`Pozdrav od servera\n`);
    socket.pipe(socket);
  });

adresa = '127.0.0.1';
port = 1337;
server.listen(port, adresa);
console.log(`Server slusa na adresi ${adresa}, port ${port} `);
```

Prilikom pokretanja servera, na konzoli se dobija poruka sledeceg oblika:

Server slusa na adresi 127.0.0.1, port 1337

Sa pokrenutim serverom se korisnik moze povezati na dva nacina:

- Koriscenjem Unix naredbe *netcat* iz komandne linije:

```
\$ netcat 127.0.0.1 1337
```

Tada bi odgovor trebalo da bude:

```
> Pozdrav od servera
```

- Pokretanjem klijenta definisanog u datoteci *client.js*:

```
let net = require('net');
let klijent = new net.Socket();
adresa = '127.0.0.1';
port = 1337;
klijent.connect(port, adresa,
  () => {
    console.log('Povezan sa serverom');
    klijent.write('Pozdrav za server! Pozdrav salje klijent.');
```

```
  });
klijent.on('data',
  (data) => {
    console.log('Primljeno: ' + data);
    klijent.destroy(); // posle odgovora servera, klijent se unistava
  });
klijent.on('close',
  () => console.log('Veza je zatvorena'));
```

Uocavamo da klijent po povezivanju, na konzoli prikaze status i serveru posalje pozdravnu poruku, a da svaki put kada prima podatke od servera, prikaze te podatke na konzoli.

Pokretanjem klijenta, tokom vremena kada server radi, dobija se sledeci izlaz na konzoli:

Povezan sa serverom

Primljeno: Pozdrav od servera

Veza je zatvorena

3. JavaSkript serversko programiranje korišćenjem okruženja node. UDP aplikacije.

Primer. Ilustruje jednostavno mrežno programiranje preko UDP protokola koriscenjem modula *dgram*.

Programski kod servera se nalazi u datoteci *server.js*:

```
const PORT = 33333; const HOST = '127.0.0.1';
const dgram = require('dgram');
let server = dgram.createSocket('udp4');
server.on('listening', function () {
  let address = server.address();
  console.log('UDP Server listening on ' + address.address + ':' + address.port);
});
server.on('message', function (message, remote) {
  console.log(remote.address + ':' + remote.port + ' - ' + message);
});
server.bind(PORT, HOST);
```

Vazno je napomenuti da:

- Kod izvršenja funkcije *bind* nad serverom, parametar *HOST* je opcionalan - njegova podrazumevana vrednost je *0.0.0.0*.
- Dogadjaj *message* se ispaljuje kada UDP paket stigne na odrediste za ovaj server.
- Dogadjaj *listening* se ispaljuje kada server bude inicijalizovan i potpuno spreman da prima UDP pakete. Metod *createSocket()* modula *dgram* moze da prihvati jednu od dve vrednosti: nisku 'udp4' ili nisku 'udp6'. Prva oznacava da ce biti koriscen IPv4, a druga da ce biti koriscen IPv6.

Klijent je definisan u datoteci *client.js*:

```
const PORT = 33333;
const HOST = '127.0.0.1';
const dgram = require('dgram');
let message = new Buffer('My KungFu is Good!');
let client = dgram.createSocket('udp4');
client.send(message, 0, message.length, PORT, HOST,
  function (err, bytes) {
    if (err)
      throw err;
    console.log('UDP message sent to ' + HOST + ':' + PORT);
    client.close();
  });
```

Vazno je napomenuti da:

- Funkcija *send()* zahteva postojanje pravog node.js objekta tipa *Buffer*, a ne nisku ili broj (objekat tipa *Buffer* je prvi parametar funkcije *send()* za slanje datagrama preko soketa).
- Drugi parametar funkcije *send()* predstavlja poziciju u baferu od koje pocinje UDP paket koji se salje. U gornjem kodu, vrednost odgovarajuceg argumenta je *0*, sto znaci da se poruka iz bafera prenosi od pocetka.
- Treci parametar je broj bajtova bafera, pocev od date pozicije, koji ce se sadrzati u UDP paketu za slanje. U prethodnom slucaju, vrednost tog argumenta je *message.length* (tj. 16) - s obzirom da je bafer mali, ovde ceo bafer moze biti poslat u okviru jednog UDP paketa. U nekim slucajevima, kada je bafer veliki, bice potrebno da se iterira kroz bafer i da se sadrzaj salje u vecem broju manjih UDP paketa.
- Protokol UDP je napravljen tako da ako poruka predje dopustenu velicinu paketa, tada se paket nece preneti, ali nece biti prijavljena nikakva greska.
- Objekat *err* u funkciji privatnog poziva funkcije *send()* ce se odnositi na upit nad DNS serverom
- Da bi osigurali da ce paketi stici na odrediste, neophodno je da host i/ili IP adresa budu u saglasnosti sa verzijom IP-a, koja se koristi. Ako to nije slucaj, komunikacija nece postojati.

4. JavaSkript veb programiranje korišćenjem okruženja node. Procesiranje zahteva i generisanje odgovora.

Veb aplikacija bide razvijena isključivo pomodu okruženja *node.js*.

Primer. Ilustruje jednostavan veb server korišćenjem modula *http*.

Programski kod servera se nalazi u datoteci *web-server.js*:

```

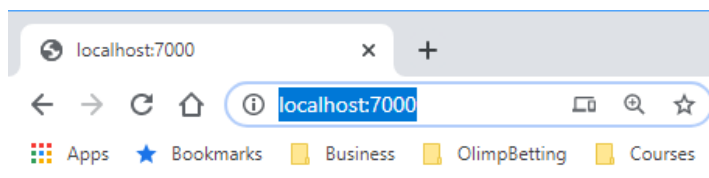
let http = require('http');
const port = 7000;
http.createServer(
  function (zahtev, odgovor) {
    odgovor.writeHead(200, { 'Content-type': 'text/plain' });
    odgovor.write('Napraavljeni veb server koristi node.js');
    odgovor.end();
  }).listen(port);
console.log(`Veb server osluskuje zahteve na portu ${port}...\n`);

```

Kakav god zahtev stigao ovom serveru, on de uvek prosledivati isti odgovor. Pokretanje ovog servera se realizuje na isti način kao i kod prethodnih primera: node veb-server.js

S obzirom da se radi o veb serveru, sada nema potrebe da se pravi posebni klijent, ved kao klijent može da posluži:

- **veb pregledač**, pri čemu u adresnu liniju treba uneti adresu **http://localhost:7000**
- **alat curl**, pri čemu u terminal treba uneti naredbu **curl http://localhost:7000**
- **alat Postman**, gde treba oformiti get zahtev i poslati taj zahtev



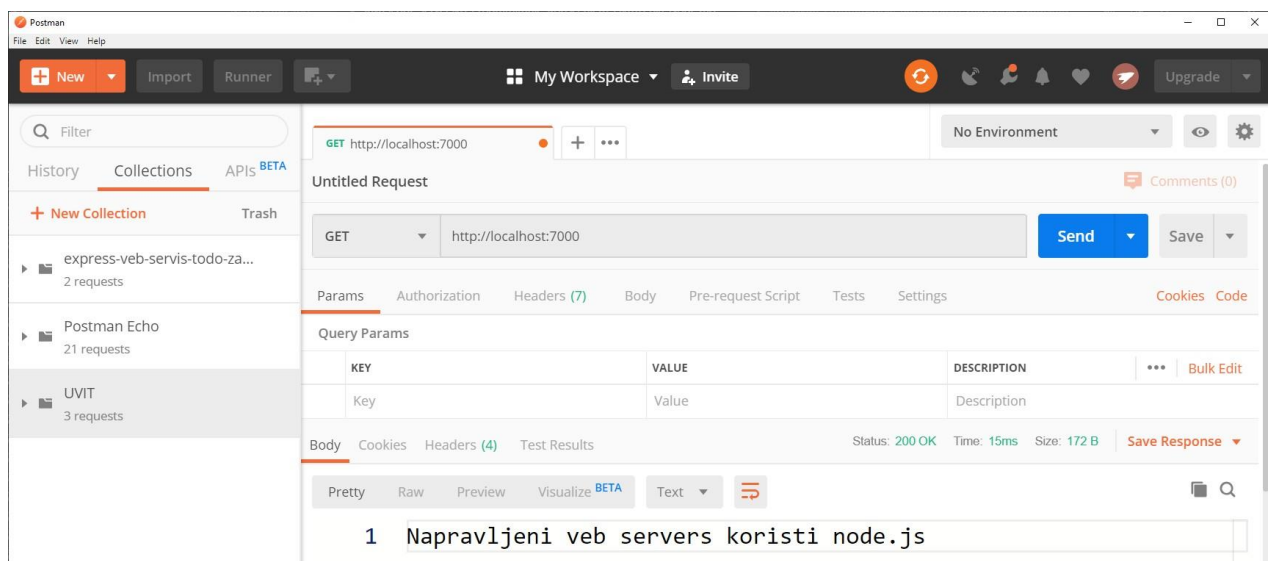
Napraavljeni veb servers koristi node.js

1.

2. StatusCode 200

StatusDescription : OK

Content : Napraavljeni veb servers koristi node.js RawContentLength 39



3.

U svim ovim slučajevima je dobijen isti odgovor od servera, samo se načini prikaza tog dobijenog odgovora razlikuju zato što su korišćeni različiti tipovi klijenata.

Primer. Jednostavni veb server koji prilikom obrade zahteva i generisanja odgovora, zahtev prikazuje na serverskoj konzoli.

```
const http = require('http');
const port = 7000;
http.createServer(function (zahtev, odgovor) {
    odgovor.writeHead(200, { 'Content-type': 'text/plain' });
    odgovor.write(`Napravljeni veb servers koristi node.js \n`);
    odgovor.write(`i zahteve upisuje u konzolu servera.`);
    odgovor.end();
    let tekuceVreme = new Date();
    console.log("---" + tekuceVreme + "---");
    console.log(zahtev);
}).listen(port);
console.log(`Veb server osluskuje zahteve na portu ${port}...\n`);
```

I ovaj veb server osluškuje na portu **7000**. Po pokretanju ovog servera i prilikom slanja zahteva prema njemu, dobide se odgovarajući odgovor, a na konzoli servera de biti prikazano vreme obrade zahteva i objekat koji predstavlja prispeli zahtev (u JSON formatu).

Veb server osluskuje zahteve na portu 7000...

---Wed Nov 27 2019 14:02:35 GMT+0100 (Central European Standard Time)---

IncomingMessage {

_readableState: ...

5. JavaSkript veb programiranje korišćenjem okruženja node. Određivanje putanje i upita kod zahteva.

Prilikom opisa protokola HTTP opisani su URL, putanja (engl. path) i upit (engl. querystring), kao i njihova funkcija prilikom obrade zahteva i generisanja odgovora. Prilikom procesiranja na strani veb servera, od celog URL koji je dao korisnik, programera interesuje samo onaj deo koji opisuje gde se zahtevani resurs nalazi u okviru veb servera. (Jasno je da programerima veb servera treba jednostavan i elegantan način da izdvoje ove elemente iz zahteva.)

Primer. Jednostavni veb server koji prilikom obrade zahteva, na konzoli servera prikazuje URL i putanju. (Programski kod servera sa nalazi u datoteci *veb-server.js*)

```
let http = require('http');
let url = require('url');
const port = 7000;
http.createServer(function (zahtev, odgovor) {
    odgovor.writeHead(200, { 'Content-type': 'text/plain' });
    odgovor.write(`Napravljeni veb servers koristi node.js \n`);
    odgovor.write(`i upisuje URL i putanju u konzolu servera.`);
    odgovor.end();
    let tekuceVreme = new Date();
    console.log("---" + tekuceVreme + "---");
```

```

    console.log('url: ' + zahtev.url);
    let putanja = url.parse(zahtev.url).pathname;
    console.log('putanja: ' + putanja);
  }).listen(port);
  console.log(`Veb server osluskuje zahteve na portu ${port}...\n`);

```

Ako, po pokretanju ovog servera, korisnik u adresi veb pregledača unese npr.

http://localhost:7000/paja=patak?sestric=raja, dobide se ovakve poruke na konzoli:

```

---Wed Nov 27 2019 14:37:00 GMT+0100 (Central European Standard Time)---
url:      /paja=patak?sestric=raja
putanja:  /paja=patak

```

Primer. Jednostavni veb server koji prilikom obrade zahteva, na konzoli servera prikazuje URL, putanju i upit. (Programski kod servera sa nalazi u datoteci *veb-server.js*)

```

const http = require('http');
const url = require('url');
const port = 7000;
http.createServer(function (zahtev, odgovor) {
  odgovor.writeHead(200, { 'Content-type': 'text/plain' });
  odgovor.write('Napraavljeni veb servers koristi node.js \n');
  let tekuceVreme = new Date();
  odgovor.write('vreme (sa servera): ' + tekuceVreme + "\n");
  console.log("---" + tekuceVreme + "---");
  odgovor.write('url: ' + zahtev.url + "\n");
  console.log('url: ' + zahtev.url);
  let putanja = url.parse(zahtev.url).pathname;
  odgovor.write('putanja: ' + putanja + "\n");
  console.log('putanja: ' + putanja);
  let upit = url.parse(zahtev.url).query;
  odgovor.write('upit: ' + upit + "\n");
  console.log('upit: ' + upit);
  odgovor.end();
}).listen(port);
console.log(`Veb server osluskuje zahteve na portu ${port}...\n`);

```

Prilikom obrade zahteva, na konzoli de biti prikazani putanja i upit. Ako se, po pokretanju servera, pošalje zahtev oblika **http://localhost:7000/paja=patak?sestric1=raja**, na konzoli de se prikazati:

```

---Wed Nov 27 2019 14:47:15 GMT+0100 (Central European Standard Time)---
url:      /paja=patak?sestric1=raja
putanja:  /paja=patak
upit:     sestric1=raja

```

Ako bi se, po pokretanju servera, poslao zahtev oblika

http://localhost:7000/paja=patak?sestric1=raja&sestric2=gaja na konzoli servera bi se prikazalo:

```

---Wed Nov 27 2019 14:50:57 GMT+0100 (Central European Standard Time)---
url:      /paja=patak?sestric1=raja&sestric2=gaja
putanja:  /paja=patak          upit:     sestric1=raja&sestric2=gaja

```

6. JavaScript veb programiranje korišćenjem okruženja node. Slanje datoteka kao odgovora.

Primer. Jednostavni veb server koji prilikom obrade zahteva konsultuje sistem datoteka na serveru u pokušaju da nađe zahtevani resurs. (Kao i u prethodnim primerima, programski kod servera sa nalazi u datoteci *veb-server.js*)

```
const http = require('http');
const url = require('url');
const fs = require('fs');
const port = 7000;
http.createServer(function (zahtev, odgovor) {
  pathName = url.parse(zahtev.url).pathname;
  fs.readFile(__dirname + pathName, function (err, data) {
    if (err) {
      odgovor.writeHead(404, { 'Content-type': 'text/plain' });
      odgovor.write(`Page Was Not Found ${JSON.stringify(err)}`);
      odgovor.end();
    } else {
      odgovor.writeHead(200);
      odgovor.write(data);
      odgovor.end();
    }
  });
}).listen(port);
console.log(`Veb server osluskuje zahteve na portu ${port}...\n`);
```

Bilo koji resurs (HTML datoteka, slika, PDF dokument i sl.) koji se nalazi u direktorijumu u kom je i veb-server.js ili nekom njegovom poddirektorijumu de biti pristupačan iz veb pregledača - samo treba navesti putanju do tog resursa u adresnoj liniji.

7. JavaScript veb programiranje korišćenjem okruženja node. Mapa sadržaja.

Mapa sadržaja (engl. **content map**) kod veb servera je mapa koja preslikava putanje tj. rute (engl. **route**) pristiglih zahteva u podatke na osnovu kojih de se realizovati generisanje odgovora.

Ova konstrukcija je dobila ime *mapa sadržaja* zato što se obično realizuje kao mapa (rečnik) - sadrži parove ključ-vrednost smeštene u stukturu koja je optimizovana za brz pristup vrednosti preko ključa (heš tabela). Kod mape sadržaja **ključevi** su **putanje**, a **vrednosti** su **podaci** koji govore kako generisati odgovor ako je u zahtevu bila putanja koja se poklapa sa ključem.

Primer. Jednostavni veb server koji prilikom obrade zahteva i generisanja odgovora konsultuje mapu sadržaja. (Celokupan programski kod servera sa nalazi u datoteci **veb-server.js**. U ovom primeru promenljiva **contentMap** referiše na mapu sadržaja)

```
let http = require('http')
let url = require('url');
```

```

http.createServer(onRequest).listen(8888);
console.log('Server has started');
function onRequest(request, response) {
    let pathName = url.parse(request.url).pathname;
    console.log(pathName);
    prikazStrane(response, pathName);
}
let contentMap = {
    '/': '<h1> Dobrodosli na sajt </h1>',
    '/kontakt': '<h1> Kontaktne informacije </h1>',
    '/opis': '<h1> Opis sajta </h1>',
    '/korisnici': '<h1> Spisak korisnika veb sajta </h1>',
    '/privatno': '<h1> Privatni podaci </h1>'
}
function prikazStrane(response, pathName) {
    if (contentMap[pathName]) {
        response.writeHead(200, { 'Content-Type': 'text/html' });
        response.write(contentMap[pathName]);
        response.end();
    } else {
        response.writeHead(404, { 'Content-Type': 'text/html' })
        response.write('404 Page not found');
        response.end();
    }
}

```

Odgovor de zavisiti od putanje zahteva - jedan de se odgovor slati ako je putanja /, drugi ako je putanja /kontakt, a sasvim treći za putanju /korisnici...

Da bi se olakšao razvoj i održavanje programskog koda veb servisa, obično se primenjuje *modularna organizacija*, kako je i opisano u ranijem poglavlju. Ta modularna organizacija se može primeniti i kada veb servis koristi mapu sadržaja i tada mapa bude izdvojena u posebnoj datoteci.

8. JavaSkript veb programiranje korišćenjem okruženja node. Obrada veb formulara. Metod GET.

Protokol HTTP dopušta da se, pored različitih putanja, veb serveru od strane korisnika prosleđuju i dodatni podaci koji mogu uticati na tok i ishod obrade zahteva.

Elementi HTML strana koji omogućavaju da se takve dodatne informacije proslede serveru se obično nalaze u okviru elementa **<form>**. Veb strana koja korisniku omogućava da unese neke podatke i prosledi ih prema veb serveru se naziva **veb formular** (engl. **web form**).

Primer. Ilustracija veb formulara. (Veb formular iz primera koji prema serveru prosleđuje dva tekstovna podatka koji predstavljaju ime i adresu elektronske pošte).

```

<html>
<body>
    <form action="/pozdrav" method="get">

```

```

    Ime:
      <input type="text" name="ime">
    <br> E-mail:
      <input type="text" name="email">
    <br>   <input type="submit" value="Dobro dosli!">
  </form>
</body>
</html>

```

Može se primetiti da se u okviru elementa **<form>** nalaze elementi **<input>**, koji u zavisnosti od njihove funkcionalnosti mogu imati različite vrednosti atributa **type**. Ako je vrednost ovog atributa **text**, onda se radi o tekstualnom polju, a ako je vrednost **submit** onda se radi o dugmetu čijim pritiskom korisnik inicira slanje unetih podataka prema serveru.

Metod GET

Primer. Modularno organizovan, veb server koji obrađuje zahtev sa metodom GET. (Veb strana sa formularom se nalazi na serveru, tj. u direktorijumu gde se nalazi serverski kod). Njen naziv je **dobro-dosli-start.html**, a sadržaj je isti kao u prethodnom primeru.

Programski kod servera je raspoređen u više modula, koji se učitavaju pomoću ranije opisanog **CommonJS mehanizma**. Polazna tačka za pokretanje servera se nalazi u datoteci **veb-server.js**)

```

const http = require('http');
const url = require('url');
const querystring = require('querystring');
const prikaz = require('./prikaz-strane');
const port = 7000;
http.createServer(onRequest).listen(port);
console.log('Server has started');
console.log(`Veb server osluskuje zahteve na portu ${port}...\n`);
function onRequest(request, response) {
  if (request.method == 'GET') {
    let pathName = url.parse(request.url).pathname;
    let queryText = url.parse(request.url).query;
    let queryData = querystring.parse(queryText);
    let tekuceVreme = new Date();
    console.log("---" + tekuceVreme + "---");
    console.log(pathName);
    console.log(queryText);
    prikaz.prikazStrane(response, pathName, queryData);
  }
}

```

U prethodnom kodu se, ako je zahtev sa metodom GET, iz zahteva izvlače putanja, upit i parametri upita i na osnovu njih se prikazuje odgovarajuća strana. Funkcija za prikaz strane na osnovu ovih parametara se nalazi u posebnom modulu, u datoteci **prikaz-strane.js**:

```

const fs = require('fs');
const sadrzaj = require('./sadrzaj');

```

```

function prikazStrane(response, pathName, queryData) {
  if (sadrzaj.contentMap[pathName]) {
    let izbor = sadrzaj.contentMap[pathName];
    if (izbor == "pozdrav") {
      response.writeHead(200);
      response.write(
        `<html>
        <body>
        Dobro dosli, ${queryData.ime}<br>
        Vasa email adresa je: ${queryData.email} <br>
        </body>
        </html>`);
      response.end();
    } else {
      fs.readFile(__dirname + '/' + izbor, function (err, data) {
        if (err) {
          response.writeHead(500, { 'Content-type': 'text/plain' });
          response.write(
            `Error in processing page ${JSON.stringify(err)}`);
          response.end();
        } else {
          response.writeHead(200);
          response.write(data);
          response.end();
        }
      });
    }
  } else {
    response.writeHead(404, { 'Content-Type': 'text/html' });
    response.write('404 Page not found');
    response.end();
  }
}
exports.prikazStrane = prikazStrane;

```

Može se primetiti da se prilikom odluke kako generisati odgovor konsultuje mapa sadržaja za ovaj veb sajt. Ova mapa sadržaja se nalazi u JavaScript datoteci **sadrzaj.js** i ima slededi oblik:

```

const mapaSadrzaja = {
  '/': 'dobro-dosli-start.html',
  '/dobro-dosli-start.html': 'dobro-dosli-start.html',
  '/pozdrav': 'pozdrav'
}
exports.contentMap = mapaSadrzaja;

```

Dakle, ako je putanja / ili **/dobro-dosli-start.html**, tada de biti prikazana veb strana sa veb formularom. Ako je putanja **/pozdrav**, tada de se, kao odgovor na zahtev, u veb strani koja predstavlja odgovor upisati parametri koje je korisnik prosledio. Ako putanja nije ništa od ovoga, onda de se smatrati da putanja opisuje datoteku iz sistema datoteka na serveru, pa de datoteka “prozvana” putanjom biti vrađena kao odgovor.

9. JavaScript veb programiranje korišćenjem okruženja node. Obrada veb formulara. Metod POST.

Primer. Modularno organizovan, veb server koji obrađuje i GET i POST zahteve. |

Ovde de na serveru (preciznije, u direktorijumu gde se nalazi serverski kod) postojati dve veb strane sa formularom - jedna za omogudavanje da korisnik pošalje GET zahtev, a druga koja omoguduje slanje POST zahteva.

Veb strana sa formularom preko koga se generiše POST zahtev se čuva u datoteci **dobro-dosli-post.html**:

```
<html>
<body>
  <form action="/pozdrav" method="post">
    Ime:
    <input type="text" name="ime">
    <br> E-mail:
    <input type="text" name="email">
    <br>
    <input type="submit" value="Dobro dosli!">
  </form>
</body>
</html>
```

Uočava se da veb formulari iz prethodne dve veb strane po unosu podataka od strane korisnika i njegovom pritisku na dugme za slanje prema serveru, "gađaju" istu putanju **/pozdrav**, s tim što se u prvom slučaju kreira zahtev sa metodom GET, a u drugom slučaju sa metodom POST.

Programski kod servera je raspoređen u više modula dinamički učitavanih pomodu **CommonJS mehanizma**. (Polazna tačka za pokretanje servera se nalazi u datoteci **veb-server.js**)

```
const http = require('http');
const url = require('url');
const querystring = require('querystring');
const prikaz = require('./prikaz-strane');
const PORT = 7000;
http.createServer(onRequest).listen(PORT);
console.log('Server je pokrenut');
function onRequest(request, response) {
  let pathName = url.parse(request.url).pathname;
  if (request.method == 'GET') {
    let queryText = url.parse(request.url).query;
    let queryData = querystring.parse(queryText);
    console.log("GET:" );
    console.log(queryData );
    prikaz.prikazStrane(response, pathName, queryData);}
}
else if (request.method == 'POST') {
  let body = "";
  request.on('data', function (data) {
    body += data;
```

```

});
request.on('end', function () {
    let postData = querystring.parse(body);
    console.log("POST:");
    console.log(postData);
    prikaz.prikazStrane(response, pathName, postData);
});
}
}

```

U gornjem kodu se, prvo iz zahteva izvuče putanja.

Ako se radi o zahtevu koji je poslat metodom POST, tada se (preko funkcija sa povratnim pozivima koje se odnose na telo zahteva) izvlače podaci koje je prosledio korisnik i na osnovu njih se generiše i prikazuje odgovarajuća strana.

Funkcija za prikaz strane koja se poziva ima tri parametra (bez obzira na to da li je zahtev koji se obrađuje sa metodom GET ili sa metodom POST): tok u koji se upisuje odgovor, putanju i podatke koje je poslao korisnik prema veb serveru. Ova funkcija se nalazi u posebnom modulu, u datoteci **prikaz-strane.js**, koja ima slededi sadržaj:

```

const fs = require('fs');
const sadrzaj = require('./sadrzaj');
function prikazStrane(response, pathName, submittedData) {
    if (sadrzaj.contentMap[pathName]) {
        let izbor = sadrzaj.contentMap[pathName];
        if (izbor == "pozdrav") {
            response.writeHead(200);
            response.write(
                `<html>
                <body>
                Dobro dosli, ${queryData.ime}<br>
                Vasa email adresa je: ${queryData.email} <br>
                </body>
                </html>`);
            response.end();
        } else {
            fs.readFile(__dirname + '/' + izbor, function (err, data) {
                if (err) {
                    response.writeHead(500, { 'Content-type': 'text/plain' });
                    response.write(
                        `Error in processing page ${JSON.stringify(err)}`);
                    response.end();
                } else {
                    response.writeHead(200);
                    response.write(data);
                    response.end();
                }
            });
        }
    }
}

```



```
} else {  
    response.writeHead(404, { 'Content-Type': 'text/html' })  
    response.write('404 Page not found');  
    response.end();  
}  
}  
exports.prikazStrane = prikazStrane;
```

Može se primetiti da se prilikom odluke kako generisati odgovor konsultuje mapa sadržaja koja se nalazi u JavaScript datoteci **sadrzaj.js** i ima sledeći oblik:

```
const mapaSadrzaja = {  
    '/': 'dobro-dosli-get.html',  
    '/dobro-dosli-post': 'dobro-dosli-post.html',  
    '/dobro-dosli-get': 'dobro-dosli-get.html',  
    '/pozdrav': 'pozdrav'  
}  
exports.contentMap = contentMap;
```

Dakle, ako je putanja **/** ili **/dobro-dosli-get.html**, tada de biti prikazana veb strana sa veb formularom koji omogudava generisanje GET zahteva. Ako je putanja **/dobro-dosli-post**, tada de biti prikazana veb strana sa veb formularom koji omogudava generisanje POST zahteva. Ako je putanja **/pozdrav**, tada de se, kao odgovor na zahtev, u veb strani koja predstavlja odgovor upisati parametri koje je korisnik prosledio. Ako putanja nije ništa od ovoga, onda de se smatrati da putanja opisuje datoteku iz sistema datoteka na serveru, pa de datoteka “prozvana” putanjom biti vradena kao odgovor.

16. JavaScript veb programiranje na klijentskoj strani

1. JavaScript i veb strane.

Iz klijentskog JavaScript-a se može programski pristupiti elementima na veb strani, čitati ih, menjati njihove attribute, dodavati ih ili brisati, po potrebi.

Svaka veb strana omogućuje da se iz JavaScript koda pristupa sledećim objektima:

- window: objekat najvišeg nivoa; sadrži svojstva primenjiva na celokupan prozor
- location: sadrži svojstva tekude URL lokacije
- history: sadrži svojstva prethodno posedenih
- document: sadrži svojstva sadržaja tekudeg dokumenta, kao što su naslov (title), boja pozadine (bgcolor), forme itd.

Primer. Primeri postavljanja svojstava za objekte:

```
document.title = 'Probni dokument'; //naziv dokumenta (title)
document.bgColor = '#070707';
document.fgColor = '#00FFFF'
history.length = 7;
location.href = 'http://uvit.math.rs';
```

Dopušten je pristup elementima na veb strani iz JavaScript-a. Jedan način za programski pristup elementima na veb strani je referisanje preko vrednosti atributa name.

Primer. U programskom kodu koji sledi, elementima na veb strani se može pristupiti na osnovu dodeljenog imena:

```
let forma = document.mojaForma;
let txt = forma.textIme;
console.log(txt.outerHTML);
let btn = forma.buttonOk;
console.log(btn.outerHTML);
```

Naravno, JavaScript objekti kojima smo pristupili na prethodno opisani način mogu imati svoje attribute.

Primer. U programskom kodu koji sledi, postavljamo attribute za elemente na veb strani.

```
document.mojaforma.action = "http://www.matf.bg.ac.rs/primeri/obrada.html";
document.mojaforma.method = get;
document.mojaforma.length = 5;
document.mojaforma.button1.value = "Klikni ovde";
document.mojaforma.text1.name = TekstPolje1;
document.mojaforma.check1.defaultChecked = true;
```

2. Objektni model dokumenta.

Objektni model dokumenta (eng. Document Object Model - DOM) je interfejs koji omogućava programima tj. skriptovima da dinamički pristupe i izmene sadržaj, strukturu ili stil Veb dokumenta. DOM je nezavistan od jezika iz kojega se koristi i platforme na kojoj se koristi. U okviru DOM, elementi dokumenta se predstavljaju

objektima (eng. objects) koji imaju svoja svojstva (eng. properties) i metode (eng. methods). Promenom vrednosti svojstava objekta i pozivom metoda nad objektima, menja se sadržaj, struktura ili stil dokumenta. Dakle, DOM obezbeđuje uniforman način da se iz bilo kojeg skript jezika pristupi, promeni, doda ili obriše proizvoljni HTML element.

DOM je nastao još 1990-tih, u vreme ratova pregledaca. Godine 1996., kompanija Netscape u okviru njihovog pregledaca ugrađuje podršku za prvu verziju jezika JavaScript koja Veb stranice proširuje mogućnošću interakcije na klijentskoj strani. Istovremeno Microsoft u Internet Explorer 3.0 ugrađuje podršku za jezik JScript, zasnovan na Netscape-ovom jeziku JavaScript. Način na koji ova dva jezika komuniciraju sa okružujućim HTML dokumentom predstavio je osnovu za definisanje DOM. DOM iz ovog perioda se danas označava kao „DOM Level 0” ili „Legacy DOM”. Ovaj model nije zvanično standardizovan, ali je delimično opisan u okviru specifikacije jezika HTML 4. Legacy DOM omogućava pristup samo određenim elementima HTML dokumenta.

Godine, 1997. sa novijim verzijama Netscape Navigator i Internet Explorer pregledaca, javila se mnogo bolja podrška za dinamički HTML, što je zahtevalo proširenje objektnog modela dokumenta. Dodata je mogućnost pristupa većini HTML elemenata, mogućnost kontrolisanja izgleda stranica modifikovanjem CSS svojstava od strane skriptova, bogatiji model događaja za interakciju sa korisnikom i slično. Na nesreću, u jeku ratova pregledaca, obe kompanije nezavisno vrše proširenja rudimentarnog Legacy DOM i time narušavaju koliko toliko postojeću sinhronizovanost. Ove, međusobno različite verzije DOM danas se nazivaju „Intermediate DOM”.

Uvidevši probleme nesinhronizovanog razvoja, pod okriljem W3C, krajem 1990-tih započinje rad na standardizaciji klijentskih skript jezika, a nakon toga i DOM. Definiše se standard ECMAScript koji biva usvojen i u okviru JavaScript i JScript jezika. Nakon ovoga, 1998. definiše se standardna verzija DOM, poznata kao DOM Level 1. Ovaj model se dalje proširuje novim mogućnostima i novom funkcionalnošću i 2000., definiše se DOM Level 2, a nakon toga, 2004. godine i DOM Level 3.

DOM predstavlja sliku HTML/XML dokumenta u memoriji, u obliku drveta čiji su čvorovi DOM objekti. Standardizacijom HTML-a i DOM-a, postalo je jasno da prva komponenta svakog pregledaca mora biti rasclanjavač (eng. parser) koji čita tekst HTML/XML datoteke i u memoriji ga predstavlja u obliku DOM drveta. Ove komponente se nazivaju raspoređivačkim masinama (eng. layout engine) i počinju da se razvijaju u okviru zasebnih projekata, nezavisno od samih pregledaca.

U okviru DOM, svakom delu dokumenta pridružen je zaseban DOM objekat.

Objekti imaju svoja svojstva (atribute) i metode kojima se obično pristupa korišćenjem sintakse objekat.svojstvo i objekat.metod(parametri). Svaki DOM objekat ima svoj tip, pri čemu tip objekta određuje svojstva i metode koje ga odlikuju. Tipovi, pa samim tim i svojstva i metodi su određeni tzv. interfejsima (eng. interface). Interfejsi su organizovani u hijerarhiju nasleđivanja.

DOM se može posmatrati na 3 nivoa:

- _ Core DOM - standardni model za bilo koji struktuirani dokument
- _ XML DOM - standardni model XML dokumenata
- _ HTML DOM - standardni model HTML dokumenata

Core DOM definiše interfejse karakteristične za sve dokumente (npr. Interfejs [Node](#) koji je zajednički za sve čvorove DOM drveta), XML DOM definiše proširenje ovih interfejsa potrebna za predstavljanje strukture XML dokumenata (npr. interfejs [CDATASection](#) kojim se predstavlja [CDATA](#) sekcija XML dokumenta), dok HTML DOM definiše interfejse potrebne za predstavljanje strukture HTML dokumenata (npr. interfejs [HTMLDocument](#) kojim se predstavlja HTML dokument ili npr. [HTMLImageElement](#) kojim se predstavljaju elementi [img](#) HTML dokumenta).

3. Događaji kod veb programiranja.

Događaji su pojave, koje su najčešće rezultat nečega što korisnik uradi (klik mišem, klik na tastaturi, drag and drop....), mada mogu biti izazvane i od strane sistema, browsera... **Registrowanje događaja** na nekom HTML elementu podrazumeva vezivanje **oslušivača događaja** (eng. *event listener*) za HTML elemenat i definisanje posledice tog događaja. *Event listener* nakon izvršenog događaja poziva na akciju **obrađivače događaja** (eng. *event handler*). **Event handler** je callback funkcija koja se aktivira kao posledica nekog događaja.

Postoje više različitih metoda za registrowanje događaja u JavaScript-u.

Inline event handlers registruje događaj kroz **HTML atribut** i pripada **zastarelom načinu** za registrowanje događaja. HTML atribut se dobija tako što se uz naziv događaja dodaje prefiks **“on”** tj. *on{eventtype}*.

```
<button id="testing" onclick="alert('uradi nešto posle klika');">Klikni</button>
```

Primer2:

```
<button id="demo" onclick="nekaCallbackFunkcija()">Klikni me</button>
<script>
  function nekaCallbackFunkcija() {
    // Uradi nešto posle klika
  }
</script>
```

Ovakav postupak i dalje može da *“uradi posao”* mada nije najbolja praksa jer ima više mana:

- Ne odvaja JS od HTML-a
- Sprečava dodavanje više *“event handlers”* (funkcija koje se izvršavaju nakon JS događaja) po istom događaju
- Nije praktičano ukoliko treba da se primeni odredjeni događaj na više istih vrsta HTML tag-ova, jer treba upisati kod u svaki tag.
- Pri korišćenju *inline event handler-a* **ključna reč THIS ukazuje na globalni objekat** tj. *window* objekat (jer HTML atribut nije vlasnik funkcije nego samo *function call-a*).
- U slučaju da je *inline event handler* ugrađen u tag ``, javljaju se problemi sa redosledom događaja jer se takodje pokrede atribut **href** koji de otvoriti drugu stranicu.

Traditional event registration model

```
var element = document.getElementById('foo');
element.onclick = function () , /* Uradi nešto posle klika */ -;
```

Ovaj model je bolji od *inline event handler-a* jer razdvaja JS od HTML koda i ključna reč **THIS** ukazuje na targetirani element ali i dalje ne može da se dodaje više od jedne funkcije na isti događaj.

Jedini način da imamo više od jednog event handler-a na istom događaju je pokazan na slededem primeru:

```
var element = document.getElementById('foo');
element.onclick = function () {prvaFunkcija(); drugaFunkcija()}
prvaFunkcija() , // Uradi nešto posle klika -
drugaFunkcija() , // Uradi nešto posle klika -
```

“addEventListener” (W3C model)

Ovaj model je **najprihvađeniji** način registrovanja događaja i omogućava registrovanje više događaja na jednom elementu. Metoda `addEventListener` podržava i **bubbling** i **capturing** redosled izvršavanja događaja a to definiše kroz treći opcioni parametar. Ovaj parametar je boolean-ova vrednost koja definiše da li je redosled izvršenja *capture* ili ne. Stoga ako je ovaj parametar *false* znači da nije *capturing* već je *bubbling*, dok vrednost parametra *true* jasno definiše *capturing*. Defaultna vrednost je *false*, stoga ako se ne koristi ovaj treći opcioni parametar smatra se da je u pitanju *bubbling redosled*.

Primer br.1

```
var element = document.getElementById('foo');
element.addEventListener("click", function(event) {
  /* Uradi nešto posle klika */
});
```

Primer br.2

Da bi se izbeglo ponavljanje koda, preporuka je da se umesto anonimne funkcije koristi imenovana:

```
var element1 = document.getElementById('foo');
var element2 = document.getElementById('bar');
element1.addEventListener('click', uradiNesto, false);
element2.addEventListener('click', uradiNesto, false);
function uradiNesto() {
  this.style.backgroundColor = '#cc0000';
}
```

Event objekat

Kada god se desi neki događaj (eng. event), javascript smešta sve relevantne podatke vezane za događaj u event objekat (npr. gde je bio mouse pointer, kojim dugmetom je kliknuto, koje element je kliknut...). Ovaj event objekat se uvek prosledjuje callback funkciji a podacima smeštenim u njemu možemo pristupiti preko svojstva event objekta.

Primer

```
function nekaCallbackFunkcija (e) {
  console.log(e.target); // Vrada: HTML element koji inicirao događaj
  e.target.style.visibility = 'hidden'; // Sakriva HTML elemenat
}
```

Redosled izvršavanja pri propagaciji

Ukoliko imamo HTML elemente koji su “*ugnježdjeni*” jedan unutar drugog a svi imaju zakačen neki *event listener*, događaji se ne izvršavaju odjednom nego prema odredjenom redosledu. Redosled izvršavanja događaja može biti:

a) Bubbling

Ovakav redosled podrazumeva aktiviranje događaja od unutrašnjeg elementa ka spoljnom (od deteta ka roditelju). Skoro svi događaji su bubbling osim par izuzetaka kao npr. *focus* događaj kod koga je default-ni redosled *capturing*.

b) Capturing

Ovakav redosled podrazumeva aktiviranje događaja od spoljnog ka unutrašnjem (od roditelja ka detetu)

Prevenција podrazumevanog ponašanja

Pri programiranju se javljaju zahtevi za prevencijom podrazumevanog ponašanja DOM elemenata nakon nekog događaja. Najčešće je to vezano za sprečavanje link taga da otvori stranicu ili da se forma submituje nakon klika na submit dugme. Najčešće se problem rešava sa slededom sintaksom:

event.preventDefault();

Medjutim u slučaju da je DOM element kome želimo da promenimo podrazumevano ponašanje ugnježđen, a pri tom njegov roditeljski element takodje ima “zakačen” neki “*eventListener*”, sa prethodnim načinom nedemo sprečiti da se “*okine*” događaj na roditeljskom elementu. Stoga je potrebno da sprečimo propagaciju sa **event.stopPropagation()**.

Prethodni primer može da se uradi i na drugačiji način dodavanjem **return false** u okviru funkcije (govori funkciji da treba da se prekine izvršavanje, a samim tim nede dodati do reakcije na događaj). Ova linija koda menja obe linije koda iz prethodnog primera.

Dinamičko registrovanje osluškivača događaja

Kada se koristeći JavaScript dinamički grade novi HTML elementi oni se ne “radjaju” sa prikazanim *event*

4. HTML komponente. Definisane. Primeri.

Primer 1 bez strukture:

```
<html lang="sr">
<head>
  <title>Nove HTML komponente</title>
  <meta charset="UTF-8" />
  <script>
    let StudentComponent = customElements.define('uvit-student');
  </script>
</head>
<body>
  <h1>Илустрација нових <code>HTML</code> кмпнент</h1>
  <script>
    document.body.appendChild(new StudentComponent());
  </script>
  <p>
    Суауичан садржај веб суране...
  </p>
  <uvit-student></uvit-student>
</body>
</html>
```

Primer 2 sa strukturom:

```
<!DOCTYPE html>
<html lang="sr">
<head>
```

```

<title>Nove HTML komponente</title>
<meta charset="UTF-8" />
<script>
  class Student extends HTMLElement {
    constructor() {
      super(); // always call super() first in the constructor.
    }
    connectedCallback() {
      this.innerHTML = '<h2>Ovo je student!</h2>';
    }
    disconnectedCallback() {}
    attributeChangedCallback(attrName, oldVal, newVal) {}
  }
  // new element
  let StudentComponent = customElements.define('uvit-student', Student);
</script>
</head>
<body>
  <h1>Илусурација нпвих <code>HTML</code> кпмпннени</h1>
  <p>
    <script>
      document.body.appendChild(new StudentComponent());
    </script>
  </p>
  <p>
    Суауичан садржај веб суране...
  </p>
  <uvit-student></uvit-student> // prikaze se Ovo je student
</body>
</html>

```

5. HTML komponente. Shadow DOM. Primeri.

Index.html:

```

<!DOCTYPE html>
<html lang="sr">
<head>
  <title>Nove HTML komponente</title>
  <meta charset="UTF-8" />
  <script src='student-component.js'></script>
</head>
<body>
  <h1>Илусурација нпвих <code>HTML</code> кпмпннени</h1>
  <p>
    Суауичан садржај веб суране...
  </p>

```

```

    <uvit-student></uvit-student> // Ovo je student (ShadowDOM)!
    <uvit-student></uvit-student> // Ovo je student (ShadowDOM)!
</body>
</html>

```

student-component.js:

```

class Student extends HTMLElement {
  constructor() {
    super(); // увек у кпнсурукупру на ппчеуку ппзвауи super()
    const senkaKoren = this.attachShadow({ mode: 'open' }); // придруживаое ДОМ сенке уз кпреп
    senkaKoren.innerHTML =
      `<h2>
        Ovo je student (ShadowDOM)!
      </h2>`;
  }
  connectedCallback() {}
  disconnectedCallback() {}
  attributeChangedCallback(attrName, oldVal, newVal) {}
}
let StudentComponent = customElements.define('uvit-student', Student);

```

6. HTML komponente. Šabloni. Primeri.

Index.html:

```

<!DOCTYPE html>
<html lang="sr">
<head>
  <title>Nove HTML komponente</title>
  <meta charset="UTF-8" />
  <script src='student-component.js'></script>
</head>
<body>
  <h1>Илусурација нпвих <code>HTML</code> кпмпнненуи</h1>
  <p>
    Суауичан садржај веб суране...
  </p>
  <uvit-student></uvit-student> // Ovo je student (ShadowDOM)!
  <uvit-student></uvit-student> // Ovo je student (ShadowDOM)!
</body>
</html>

```

student-component.js:

```

const sablon = document.createElement('uvit-student-template');
sablon.innerHTML =
  `<style>
    h2 {

```



```

        color: white;
        background-color: #999;
        padding: 5px;
    }
</style>
<h2>
    Ovo je student (template)!
</h2>`;
class Student extends HTMLElement {
    constructor() {
        super();
        const senkaKoren = this.attachShadow({ mode: 'open' }).appendChild(sablon.cloneNode(true));
    }
    connectedCallback() {}
    disconnectedCallback() {}
    attributeChangedCallback(attrName, oldVal, newVal) {}
}
// нпви елеменау
let StudentComponent = customElements.define('uvit-student', Student);

```

7. HTML komponente. Atributi. Primeri.

Index.html:

```

<!DOCTYPE html>
<html lang="sr">
<head>
    <title>Nove HTML komponente</title>
    <meta charset="UTF-8" />
    <script src='fioka-component.js'></script>
</head>
<body>
    <h1>Илустрација нпвих <code>HTML</code> кпмпнненуи</h1>
    <p>
        Суауичан садржај веб суране...
    </p>
    <template id="uvit-fioka-template">
        <style>
            h2 {
                color: white;
                background-color: #999;
                padding: 5px;
            }
        </style>
        <h2>
            Fioka (template)!
        </h2>
    </template>

```

```

</template>

<uvit-fioka otvoreno></uvit-fioka>
<uvit-fioka></uvit-fioka>
</body>
</html>

```

fioka-component.js:

```

class Fioka extends HTMLElement {
  constructor() {
    super();
    const sablon = document.getElementById('uvit-fioka-template').content;
    const senkaKoren = this.attachShadow({ mode: 'open' }).appendChild(sablon.cloneNode(true));
    this.addEventListener('click', e => {
      if (this.disabled)
        return;
      this.pomeriFioku();
    });
  }
  get otvoreno() { // чиуаое пспбине otvoreno
    return this.hasAttribute('otvoreno');
  }
  set otvoreno(val) { // ппсуављаое пспбине otvoreno
    if (val) { // Reflect the value of the open property as an HTML attribute.
      this.setAttribute('otvoreno', '');
    } else {
      this.removeAttribute('otvoreno');
    }
  }
  pomeriFioku() { // пперација
    this.otvoreno = !this.otvoreno;
    console.log(`Fioka je pomerena. Status fioke: ${this.hasAttribute('otvoreno')?'otvoreno':'zatvoreno'}`);
  }
  connectedCallback() {}
  disconnectedCallback() {}
  attributeChangedCallback(attrName, oldVal, newVal) {}
}
let FiokaComponent = customElements.define('uvit-fioka', Fioka);

```

8. HTML komponente. Iscrtavanje. Primeri.

Index.html:

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">

```

```

<title>Живпуни циклус ппврауних ппзива</title>
<style>
  custom-square {
    margin: 20px;
  }
</style>
<script defer src="kocka-component.js"></script>
</head>
<body>
  <h1>Life cycle callbacks test</h1>
  <div>
    <button class="add">Дпдај увиу-кпцку у DOM</button>
    <button class="update">Ажурирај аүрибүүе</button>
    <button class="remove">Уклпни увиу-кпцку из DOM</button>
  </div>
</body>
</html>

```

kocka-component.js:

```

class Kocka extends HTMLElement {                                // Create a class for the element
                                                                // Specify observed attributes so that
                                                                // attributeChangedCallback will work

  static get observedAttributes() {
    return ['c', 'l'];
  }

  constructor() {
    super();
    const shadow = this.attachShadow({ mode: 'open' });
    const div = document.createElement('div');
    const style = document.createElement('style');
    shadow.appendChild(style);
    shadow.appendChild(div);
  }

  connectedCallback() {
    console.log('Custom square element added to page.');
    updateStyle(this);
  }

  disconnectedCallback() {
    console.log('Custom square element removed from page.');
  }

  adoptedCallback() {
    console.log('Custom square element moved to new page.');
  }

  attributeChangedCallback(name, oldValue, newValue) {
    console.log('Custom square element attributes changed.');
    updateStyle(this);
  }
}

customElements.define('uvit-kocka', Kocka);

```

```

function updateStyle(elem) {
  const shadow = elem.shadowRoot;
  shadow.querySelector('style').textContent =
    `div {
      width: ${elem.getAttribute('l')}px;
      height: ${elem.getAttribute('l')}px;
      background-color: ${elem.getAttribute('c')};
    }`;
}

const add = document.querySelector('.add');
const update = document.querySelector('.update');
const remove = document.querySelector('.remove');
let kockica;
update.disabled = true;
remove.disabled = true;
function random(min, max) {
  return Math.floor(Math.random() * (max - min + 1) + min);
}

add.onclick = function() {
  kockica = document.createElement('uvit-kocka');           // Create a custom square element
  kockica.setAttribute('l', '100');
  kockica.setAttribute('c', 'red');
  document.body.appendChild(kockica);
  update.disabled = false;
  remove.disabled = false;
  add.disabled = true;
};

update.onclick = function() {
  kockica.setAttribute('l', random(50, 200));               // Randomly update square's attributes
  kockica.setAttribute('c', `rgb(${random(0, 255)}, ${random(0, 255)}, ${random(0, 255)})`);
};

remove.onclick = function() {
  document.body.removeChild(kockica);                       // Remove the square
  update.disabled = true;
  remove.disabled = true;
  add.disabled = false;
};

```