






PROGRAMIRANJE ZA VEB

SKRIPTA

Ajzenhamer Nikola
Zečević Anđelka



Matematički fakultet, Univerzitet u Beogradu
2021



Copyright ©2021 Nikola Ajzenhamer, Anđelka Zečević

IZDATO OD STRANE "WWW.NIKOLAAJZENHAMER.RS"

[HTTPS://WWW.NIKOLAAJZENHAMER.RS/ASSETS/PDF/PZV.PDF](https://www.nikolaaajzenhamer.rs/assets/pdf/pzv.pdf)

Ovo delo je zaštićeno licencom Creative Commons Attribution-NonCommercial 3.0 Unported License ("Licenca"). Ovo delo se ne može koristiti osim ako nije u skladu sa Licencom. Detalji Licence mogu se videti na veb adresi <http://creativecommons.org/licenses/by-nc/3.0>. Dozvoljeno je umnožavanje, distribucija i javno saopštavanje dela, pod uslovom da se navedu imena autora. Upotreba dela u komercijalne svrhe nije dozvoljena. Prerada, preoblikovanje i upotreba dela u sklopu nekog drugog nije dozvoljena.

Prvo izdanje, Mart 2021.

Poslednja izmena: 2021-03-17 00:53

Sadržaj

Predgovor	7
------------------------	----------

I	Uvod u programiranje za veb	
1	Pregled osnovnih tehnologija	11
1.1	Klijentsko-serverski model	12
1.2	Rad veb pregledača	12
1.2.1	Mašina za prikazivanje	13
1.3	HTTP protokol	14
1.3.1	Transakcije i poruke	16
1.3.2	HTTPS	27
1.3.3	HTTP 2.0	27
	Literatura za ovu oblast	27
2	JavaScript	29
2.1	Izvršavanje JavaScript koda	29
2.2	Osnovni elementi jezika	32
2.2.1	Tipovi i operatori	32
2.2.2	Programske strukture	35
2.3	Funkcije	37
2.3.1	Domet važenja promenljivih	38
2.3.2	Izdizanje deklaracija	40
2.3.3	Opcioni i podrazumevani argumenti	41
2.3.4	Anonimne funkcije	42

2.3.5	Samoizvršavajuće funkcije	42
2.3.6	Funkcije kao podaci — nastavak	43
2.3.7	Zatvorenje	45
2.4	Strukture podataka: Nizovi i objekti	47
2.4.1	Nizovi	47
2.4.2	Objekti	47
2.4.3	Svojstva u objektima	48
2.4.4	Metodi	50
2.4.5	Deskriptorski objekat	51
2.4.6	Referenciranje objekata	52
2.4.7	Još malo o nizovima	53
2.4.8	Još malo o niskama	55
2.4.9	Funkcije sa proizvoljnim brojem parametara	56
2.4.10	Dekonstrukcija	56
2.4.11	JSON	57
2.5	Objektno-orijentisana paradigma	59
2.5.1	Objekti (metodi) i vrednost <code>this</code>	59
2.5.2	Konstruktorske funkcije	61
2.5.3	Leksičko <code>this</code>	64
2.5.4	Prototipovi	65
2.5.5	Nasleđivanje	73
2.5.6	Klase i objektno-orijentisana podrška u ECMAScript 6	84
2.6	Obrada grešaka	88
2.6.1	Striktan režim rada	88
2.6.2	Rad sa izuzecima	94
2.7	Moduli	97
2.7.1	Paketi	97
2.7.2	Načini za kreiranje modula	98
2.8	Asinhrona paradigma programiranja	100
2.8.1	JavaScript je jednonitni programski jezik	100
2.8.2	Podrška za asinhrono programiranje u jeziku JavaScript	106
2.8.3	Interfejsi veb pregledača za ostvarivanje asinhronih zahteva	130
	Literatura za ovu oblast	137
3	TypeScript	139
3.1	Prevođenje TypeScript programa	140
3.1.1	Upravljanje TypeScript projektima	141
3.2	Tipovi	141
3.2.1	Promenljive i primitivni tipovi	142
3.2.2	Enumeracije	142
3.2.3	Unije	143
3.2.4	Presek tipova	143
3.2.5	Nizovi	143
3.2.6	Funkcije	144
3.3	Klase	145
3.3.1	Konstruktori	145
3.3.2	Modifikatori pristupa	146
3.3.3	Svojstva i metodi	146

3.3.4	Nasleđivanje i interfejsi	149
3.3.5	Apstraktne klase	154
3.4	Polimorfizam	155
3.4.1	Hijerarhijski polimorfizam	155
3.4.2	Parametarski polimorfizam	158
3.4.3	Ograničenja tipa nad šablonskim parametrima	161
3.5	Dekoratori	162
3.5.1	Konfigurabilni dekoratori	163
3.5.2	Kompozicija dekoratora	164
3.5.3	Tipovi dekoratora	165
	Literatura za ovu oblast	170
4	Reaktivna paradigma	171
4.1	Asinhrono programiranje metodom eliminacije petlji	171
4.1.1	Asinhroni metodi za obradu nizova	172
4.1.2	Lančanje asinhronih metoda	173
4.1.3	Rad sa ugnežđenim strukturama	174
4.2	Uvod u reaktivno programiranje	175
4.3	Biblioteka RxJS	177
4.3.1	Tok u RxJS biblioteci	178
4.3.2	Kreiranje tokova	181
4.3.3	Vrste tokova	184
4.3.4	Ulančavanje operatora	188
4.3.5	Još neke funkcije za kreiranje tokova	188
4.3.6	Transformisanje tokova	192
4.3.7	Filterovanje tokova	195
4.3.8	Kombinovanje tokova	198
4.3.9	Obrada grešaka	199
	Literatura za ovu oblast	201

II

Programiranje serverskih aplikacija

5	Razvojno okruženje Node.js	205
5.1	Sistem zasnovan na događajima	205
5.1.1	Blokirajuće i neblokirajuće operacije	205
5.1.2	Petlja događaja	207
5.2	Kreiranje serverskih aplikacija	208
5.3	Razvojno okruženje Express.js	218
5.3.1	Opisivanje statičkih i dinamičkih sadržaja	219
5.3.2	Postavljanje REST arhitekture	224
	Literatura za ovu oblast	236
6	Baza podataka MongoDB	237
7	Mongoose ORM	239

8	Radni okvir Angular	243
8.1	O verzionisanju	243
8.2	Angular CLI i kreiranje novog projekta	244
8.3	Osnovni pogled na Angular aplikacije — podizanje aplikacije	246
8.3.1	Uključivanje Bootstrap 4 biblioteke za stilizovanje elemenata	250
8.4	Komponente i vezivanje podataka	250
8.4.1	Kreiranje komponenti	250
8.4.2	Vezivanje podataka	257
8.5	Ugrađene Angular direktive	266
8.5.1	Strukturne direktive	268
8.5.2	Atributske direktive	272
8.6	Komponente i vezivanje podataka — napredniji koncepti	274
8.6.1	Tok podataka kroz hijerarhijsku organizaciju komponenti	275
8.6.2	Referencne promenljive šablona	284
8.6.3	Dekorator @ViewChild	285
8.6.4	Element <ng-content> i dekorator @ContentChild	287
8.6.5	Životni tok komponenti i metodi za osluškivanje događaja iz životnog toka	287
8.7	Kreiranje direktiva	291
8.8	Mehanizam servisa i ubrizgavanje zavisnosti	291
8.8.1	Kreiranje servisa	291
8.8.2	Ubrizgavanje zavisnosti i korišćenje servisa u komponentama	292
8.9	Rutiranje	293
8.9.1	Specifikovanje putanja za rutiranje na klijentu	294
8.10	Filteri	298
8.10.1	Ugrađeni filteri	298
8.10.2	Korisnički-definisani filteri	299
8.11	Rad sa formularima	300
8.11.1	Kreiranje formulara u reaktivnom pristupu	300
8.11.2	Validacija formulara u reaktivnom pristupu	302
8.12	HTTP komunikacija u Angular aplikacijama	304
8.12.1	Modul HttpClientModule i servis HttpClient	305
8.12.2	Slanje HTTP zahteva	305
8.12.3	Obrada HTTP odgovora	307
8.12.4	Uvođenje tipiziranosti zahteva	308
8.12.5	Filter AsyncPipe	309
8.12.6	Dodavanje tela HTTP zahtevu	310
8.12.7	Obrada grešaka	312
8.12.8	Dodatne mogućnosti	316
	Literatura za ovu oblast	317

Predgovor

Ovaj tekst predstavlja skripta iz izbornog kursa "Programiranje za veb", na 4. godini smjera Informatika na Matematičkom fakultetu Univerziteta u Beogradu. Ova skripta su pre svega prateći materijal za časove vežbi studentima koji ovaj kurs slušaju u okviru svojih studija, ali i svima drugima koji žele da se upoznaju sa razvojem savremenih veb aplikacija. Ovaj materijal ne može zameniti pohađanje vežbi niti drugu preporučenu literaturu.

Sadržaj skriptata je napisan tako da ga i početnici u veb programiranju mogu razumeti. Ipak, iako će čitaocu biti objašnjene teme poput komunikacije na vebu, rad veb pregledača, kategorisanje sadržaja, kao i to da će biti dat uvod u programske jezike JavaScript i TypeScript, podrazumevamo osnovno poznavanje jezika za obeležavanje teksta — HTML i CSS, kao i interfejs modela objekata dokumenta (engl. *Document Object Model*, skr. *DOM*). Preporučeni resurs za savladavanje ovih tema predstavljaju elektronska skripta "*Uvod u veb i internet programiranje*" koja je dostupna besplatno na vezi <https://matfuvit.github.io/UVIT/vezbe/knjiga/>.

Sadržaj skriptata je podeljen u tri dela, od kojih je svaki podeljen u nekoliko poglavlja:

- Deo prvi je posvećen uvodu čitaoca u koncepte važne za razumevanje veb tehnologija i razvoja veb aplikacija.
 1. Prvo poglavlje se bavi uvođenjem elementarnih pojmova vezanih za **web**. Pored samih tehničkih definicija, čitalac će u ovom poglavlju pronaći diskusiju o tradicionalno najpopularnijoj arhitekturi nad kojom je veb izgrađen – u pitanju je klijent-server arhitektura. Ostatak poglavlja diskutuje o HTTP protokolu koji služi kao osnova za ostvarivanje komunikacije između veb aplikacija na aplikacionom nivou.
 2. Drugo poglavlje predstavlja uvod čitaoca u programski jezik **JavaScript**. Iako će biti diskutovani elementi ovog jezika na detaljnom nivou, očekuje se da čitalac ima prethodnog programerskog iskustva u nekom programskom jeziku višeg reda, kao što su C, Java, C++ i sl.

3. Treće poglavlje proširuje diskusiju iz prethodnog poglavlja uvođenjem programskog jezika **TypeScript**.
 4. Četvrto poglavlje uvodi čitaoca u **reaktivnu** paradigmu programiranja koja predstavlja osnovu za razumevanje nekih naprednih radnih okvira o kojima se diskutuje u kasnijim delovima. Pored objašnjavanja koncepata kao što su tok i posmatrač, u ovom poglavlju će biti dat veliki broj primera kroz jednu konkretnu implementaciju reaktivne paradigme programiranja pomoću biblioteke RxJS.
- Deo drugi je posvećen razvoju serverskih veb aplikacija.
 5. U petom poglavlju započinjemo razvoj serverskih veb aplikacija korišćenjem okruženja **Node.js**. Čitalac se uvodi u mehanizme funkcionisanja ovog okruženja. Takođe, poglavlje ilustruje razvoj REST arhitekture.
 6. U šestom poglavlju prikazujemo **MongoDB** sistem za upravljanje bazom podataka. Ovaj SUBP predstavlja jedan od primarnih izbora za razvoj savremenih veb aplikacija. Zasnovan je na nerelacionim tehnologijama, konkretno, predstavlja bazu dokumenata.
 7. U sedmom poglavlju naučimo kako da povežemo Node.js serverske aplikacije i MongoDB SUBP kako bismo omogućili trajno skladištenje podataka na nivou servera. U tu svrhu, koristimo popularni radni okvir **Mongoose** ORM.
 - Deo treći je posvećen razvoju klijentskih veb aplikacija.
 8. Osmo poglavlje uvodi čitaoca u **Angular**, savremeni radni okvir za razvoj klijentskih veb aplikacija koji se zasniva na programskom jeziku TypeScript i biblioteci RxJS za reaktivno programiranje. Nakon što savlada elementarne pojmove ovog radnog okvira, čitalac se uvodi u naprednije koncepte ovog radnog okvira, kao što su servisi, reaktivni formulari i njihova obrada, kao i asinhrona HTTP komunikacija sa serverskim aplikacijama.

Ovaj tekst je u veoma ranoj fazi formiranja i kao takav sklon je velikom broju grešaka. Ukoliko ste pažljivi čitalac ovih skripta, i ukoliko uočite bilo kakvu grešku ili propust, možete se javiti autorima putem elektronske pošte na matf@nikolaajzenhamer.rs sa naslovom *Programiranje za veb - skripta*. Svi komentari, sugestije, kritike, ali i pohvale vezane za ovaj materijal su dobrodošli.

Zahvalnice

Na veoma pažljivom čitanju i brojnim korisnim savetima zahvaljujemo se kolegama sa Matematičkog fakulteta, Ivanu Čukiću i Jeleni Marković koji su svojim izmenama ili dopunama u tekstu i predlozima zadataka učinili da ovaj tekst postane prijemčiviji studentima.

Autori

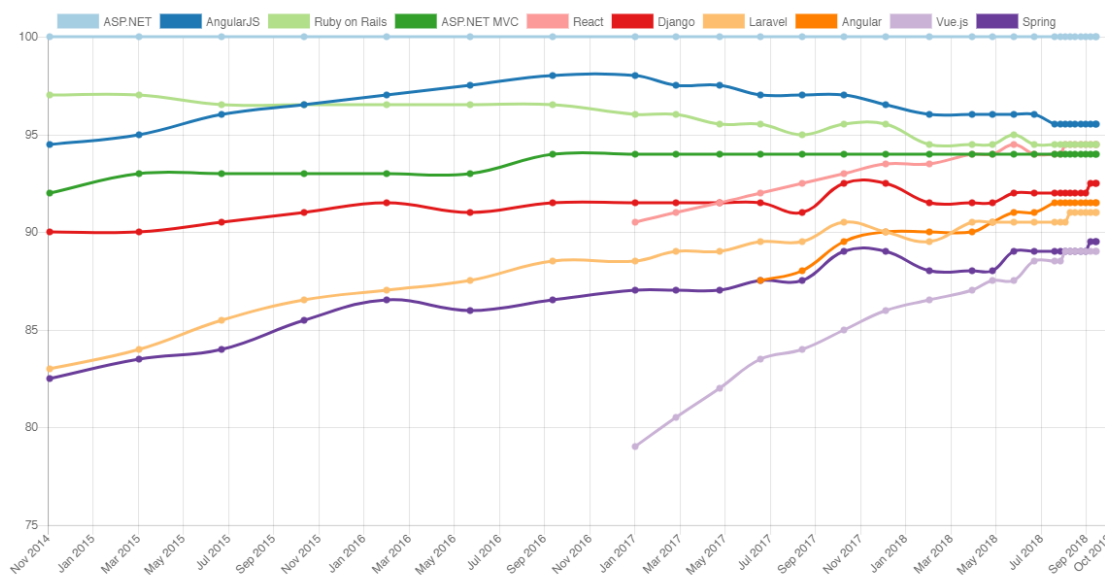


Uvod u programiranje za veb

1	Pregled osnovnih tehnologija	11
1.1	Klijentsko-serverski model	
1.2	Rad veb pregledača	
1.3	HTTP protokol	
	Literatura za ovu oblast	
2	JavaScript	29
2.1	Izvršavanje JavaScript koda	
2.2	Osnovni elementi jezika	
2.3	Funkcije	
2.4	Strukture podataka: Nizovi i objekti	
2.5	Objektno-orijentisana paradigma	
2.6	Obrada grešaka	
2.7	Moduli	
2.8	Asinhrona paradigma programiranja	
	Literatura za ovu oblast	
3	TypeScript	139
3.1	Prevođenje TypeScript programa	
3.2	Tipovi	
3.3	Klase	
3.4	Polimorfizam	
3.5	Dekoratori	
	Literatura za ovu oblast	
4	Reaktivna paradigma	171
4.1	Asinhrono programiranje metodom eliminacije petlji	
4.2	Uvod u reaktivno programiranje	
4.3	Biblioteka RxJS	
	Literatura za ovu oblast	

1. Pregled osnovnih tehnologija

Veb tehnologije su u stalnom napretku. Novi alati i nova okruženja za razvoj se pojavljuju često, a oni koji su dobro poznati programerima se konstantno unapređuju u takmiče u popularnosti (slika 1.1).



Slika 1.1: Popularnost okruženja za razvoj kroz vreme. Grafik je preuzet sa <https://hotframeworks.com/>.

Cilj ovog teksta jeste da se čitalac uvede u dinamično područje veb tehnologija, kao i da bude osposobljen za dalji samostalni rad. U tu svrhu, u ovom poglavlju biće predstavljeni osnovne tehnologije za rad na vebu. Poglavlje započinjemo objašnjenjem rada veb pregledača, koji predstavljaju prozor korisnika ka sadržaju na vebu. Zatim ćemo detaljno opisati

HTTP protokol i osvrnuti se na HTTP 2.0 i novine koje nam ovo unapređenje donosi. Konačno, poglavlje završavamo pregledom HTML5 verzije jezika za obeležavanje teksta.

U tu svrhu, započinjemo ovo poglavlje definisanjem pojmova veb i internet.

Definicija 1.0.1 — Internet. *Internet* (engl. *Internet*) predstavlja skup različitih mreža u kojima se koriste neki zajednički protokoli i obezbeđuju neke zajedničke usluge.

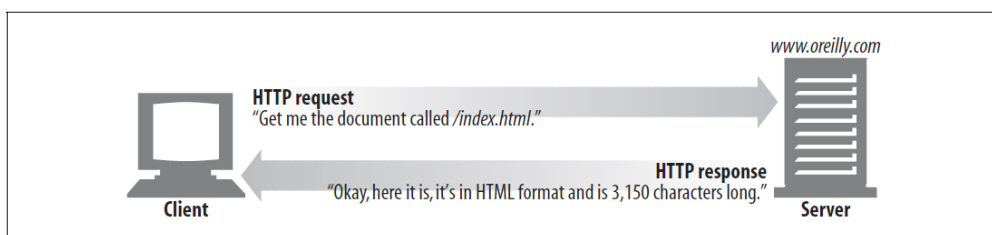
Definicija 1.0.2 — Veb. *Veb* (engl. *world wide web*) predstavlja najpoznatiji distribuirani sistem koji se izvršava preko Interneta. Distribuirani sistem korisnicima prikazuje jedinstven model koji apstrahuje skup nezavisnih računara. Dakle, to je softverski sistem koji se izvršava u mreži, zaklanja je i daje joj visoki stepen ujednačenosti.

1.1 Klijentsko-serverski model

Jedna od najznačajnijih upotreba računarskih mreža predstavlja deljenje sadržaja. Internet i veb omogućavaju deljenje sadržaja, čiji je cilj da se podaci učine dostupni svima koji se nalaze u mreži, bez obzira na njihovu stvarnu fizičku lokaciju.

Jedan mogući način podrazumeva postojanje dve vrste uređaja — server i klijent. U ovom modelu, podaci su uskladišteni na računarima koje nazivamo *serveri* (engl. *server*). U pitanju su uglavnom moćni računari kojima upravljaju administratori sistema. Sa druge strane, imamo korisničke, često jednostavnije računare, koje nazivamo *klijenti* (engl. *client*), pomoću kojih se pristupa udaljenim sadržajima.

Opisani sistem se naziva *klijentsko-serverski model* (engl. *client-server model*) i predstavlja najupotrebljiviji model na Internetu. Na slici 1.2 dat je jednostavan grafički prikaz komunikacije između klijenta i servera u opisanom modelu. Prvo, na osnovu korisničkog upita, klijent šalje zahtev serveru i traži od njega da mu isporuči sadržaj. Nakon primanja i obrade zahteva, server isporučuje odgovor klijentu, koji klijent dalje obrađuje i prikazuje korisniku.



Slika 1.2: Klijent-server model.

U ovom modelu vidimo da korisnik zadaje upit na svom klijentskom računaru. Takođe, vidimo da klijentski računar treba da prikaže rezultat upita korisniku. Postavlja se pitanje na koji način se vrši ova komunikacija između klijenta i korisnika. Odgovor na ovo pitanje daju aplikacije koje se nazivaju veb pregledači, o kojima govorimo u nastavku. Na komunikaciju u klijent-server modelu ćemo se detaljnije osvrnuti u sekciji 1.3.

1.2 Rad veb pregledača

Definicija 1.2.1 — Veb pregledač. *Veb pregledač* (engl. *web browser*) predstavlja korisničku aplikaciju čiji je glavni cilj predstavljanje sadržaja na vebu, dohvatanjem sadržaja sa servera i njihovim prikazivanjem u prozoru pregledača.

Primeri veb pregledača su: Chrome, Firefox, Internet Explorer, Microsoft Edge, Safari, Opera i dr. Pod sadržajem na vebu podrazumevamo: obeleženi tekst (na primer, HTML i CSS datoteke), izvorni kod za upravljanje dinamičkim sadržajem (na primer, JavaScript datoteke), multimedijalne datoteke (na primer, datoteke sa ekstenzijama .png, .jpeg, .gif, .mp4, .mp3, .ogg, .swf, i dr.), informacije sadržane u tekstualnim datotekama i bazama podataka i sl. Na samom startu ćemo napraviti razliku između veb pregledača i pretraživačke mašine, s obzirom da se ta dva pojma često razmatraju kao sinonimi, iako to nisu.

Definicija 1.2.2 — Pretraživačka mašina. *Pretraživačka mašina* (engl. *search engine*) predstavlja veb prezentaciju koja sadrži informacije o drugim veb prezentacijama.

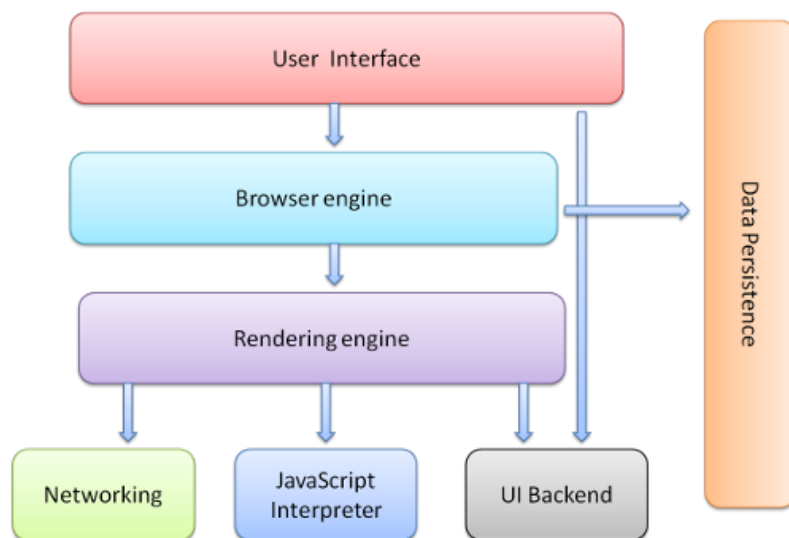
Sada ćemo se upoznati sa glavnim komponentama koje čine veb pregledač. Na slici 1.3, dat je pregled komponenti i njihov međusobni uticaj. Komponente veb pregledača su:

1. Korisnički interfejs (engl. *user interface*) predstavlja komponentu koja je vidljiva korisniku i koja prikazuje rezultate pretrage. Glavni delovi korisničkog interfejsa su: prozor za prikaz sadržaja, adresna linija za pretragu sadržaja, razne kontrole za navigaciju kroz sadržaje, njihovo memorisanje zarad bržeg pronalaženja, opcije za prikazivanje i upravljanje samom aplikacijom i dr.
2. Mašina pregledača (engl. *browser engine*) predstavlja softverski deo aplikacije koja izvršava operacije upita i upravljanja mašinom za prikazivanje.
3. Mašina za prikazivanje (engl. *rendering engine*) odgovorna je za prikaz traženog sadržaja. Na primer, ukoliko je korisnik zatražio HTML stranicu, ovaj deo aplikacije će izvršiti parsiranje i prikaz te HTML stranice.
4. Umrežavač (engl. *networking*) obavlja slanje i prihvatanje poruka preko mreže, na primer, HTTP zahteve. Koristi različite implementacije za različite platforme korišćenjem platformski-nezavisnog interjfsa.
5. Grafička mašina (engl. *UI backend*) koristi se za iscrtavanje osnovnih kontrola poput dugmića i prozora. Ovaj deo predstavlja platformski-nezavisan interfejs ispod kojeg se nalaze pozivi ka grafičkoj mašini operativnog sistema.
6. JavaScript interpreter (engl. *JavaScript interpreter*) koristi se za parsiranje i izvršavanje JavaScript koda¹.
7. Skladište (engl. *data persistence*) predstavlja sloj za skladištenje različitih informacija potrebnih veb pregledaču, kao što su kolačići. Većina savremenih veb pregledača podržava mehanizme za skladištenje, kao što su `localStorage`, `IndexedDB`, `WebSQL` i `FileSystem`.

1.2.1 Mašina za prikazivanje

Podrazumevano, mašina za prikazivanje može da prikaže HTML i XML dokumenta i slike. Ostali tipovi podataka se takođe mogu prikazivati, kroz instalaciju *dodataka* (engl. *plug-in*) ili *proširenja* (engl. *extensions*). Različiti veb pregledači koriste različite mašine za

¹Iako se ovaj deo softvera naziva *interpreter*, proces izvršavanja JavaScript koda iz izvornog koda je nešto složeniji od jednostavne interpretacije.



Slika 1.3: Glavni elementi veb pregledača.

prikazivanje: Internet Explorer koristi Trident, Firefox koristi Gecko, Safari koristi WebKit, a Chrome i Opera koriste Blink, koji se razvio iz WebKit-a.

Mašina za prikazivanje započinje svoj rad dohvaćanjem sadržaja iz umreživača. Nakon toga, vrši se parsiranje HTML dokumenta i formiraju se elementi DOM stabla², od kojeg nastaje *stablo sadržaja* (engl. *content tree*). Mašina dodatno vrši parsiranje podataka o stilu dokumenta i na osnovu ovih informacija i stabla sadržaja, formira se *stablo prikazivanja* (engl. *render tree*). Stablo prikazivanja sadrži pravougaonike sa informacijama poput boje i dimenzije. Pravougaonici su raspoređeni u redosledu kojim se iscrtavaju na ekranu.

Naredna faza jeste raspoređivanje elemenata, odn. dodeljivanje svakom elementu stabla tačne koordinate na ekranu. Konačno, dolazi se do faze iscrtavanja gde se vrši obilazak stabla prikazivanja i svaki element se iscrtava korišćenjem grafičke mašine.

Ovaj proces je inkrementalan, odn. mašina za prikazivanje će pokušavati da iscrta elemente na ekranu što je pre moguće. Drugim rečima, ona ne čeka da se parsiranje HTML dokumenta izvrši celo da bi se prešlo na naredne faze, već deo-po-deo dokumenta prolazi kroz opisan proces.

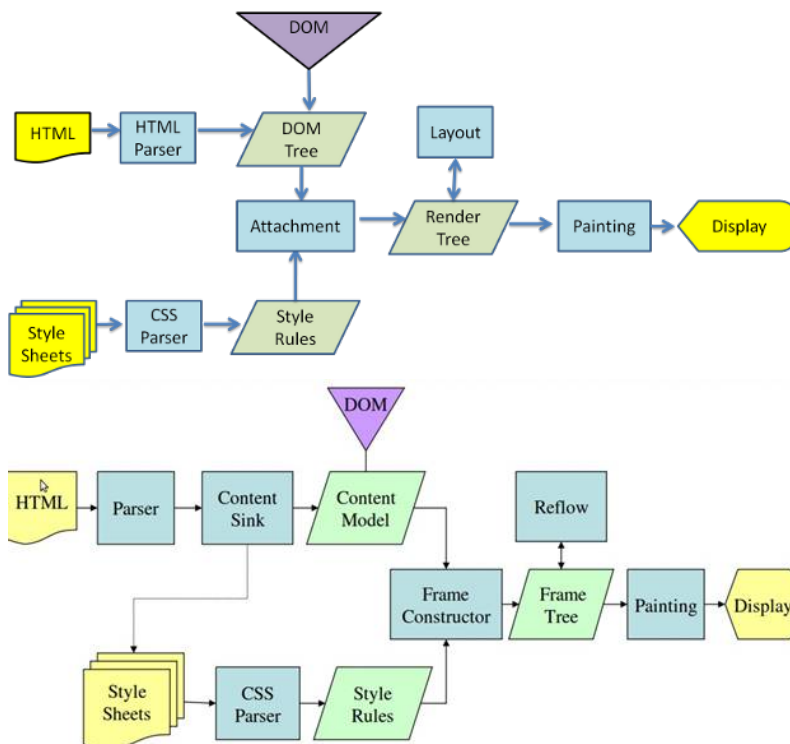
Na slici 1.4 prikazan je opisani proces prikazivanja za dve različite mašine za prikazivanje — WebKit i Gecko. Iako je terminologija koja se koristi drugačija, tok procesa se praktično ne razlikuje.

1.3 HTTP protokol

Klijenti, veb pregledači, serveri i aplikacije koje se nalaze na vebu svoju komunikaciju vrše koristeći HTTP. Možemo reći da je HTTP zajednički jezik svih podsistema na vebu.

Definicija 1.3.1 — Protokol. *Protokol* (engl. *protocol*) predstavlja dogovor između dve jedinice o tome kako treba da teče njihova međusobna komunikacija.

²Više informacija o DOM-u se može pronaći na adresi <https://www.w3.org/DOM/DOMTR>.



Slika 1.4: Grafički prikazi procesa prikazivanja u različitim mašinama za prikazivanje: WebKit (gore) i Gecko (dole).

Definicija 1.3.2 — HTTP. *Protokol za prenos hiperteksta* (engl. *HyperText Transfer Protocol*, skr. HTTP) predstavlja široko korišćen protokol za komunikaciju između aplikacija.

S obzirom da Internet sadrži ogroman broj različitih tipova sadržaja, HTTP pažljivo označava svaki sadržaj koji se deli oznakom tipa sadržaja koji se naziva *MIME* (engl. *Multi-purpose Internet Mail Extension*). U klijent-serveru modelu komunikacije, server dodaje MIME tip svim sadržajima koje isporučuje u odgovoru. MIME tip se šalje kao tekstualna oznaka u obliku `<primarni tip>/<sekundarni tip>`. Na primer,

- HTML dokument ima MIME tip `text/html`;
- Neformatiran tekstualni dokument zapisan u ASCII kodnoj shemi ima MIME tip `text/plain`;
- Slika zapisana u PNG formatu ima MIME tip `image/png`;
- Dokument zapisan u Microsoft PowerPoint Presentation formatu ima MIME tip `application/vnd.ms-powerpoint`.

MIME tipova ima mnogo, što se može videti na listi dostupnoj na adresi <https://www.freeformatter.com/mime-types-list.html>.

Naredno pitanje koje možemo postaviti sebi jeste kako je klijent u stanju da pronađe server koji sadrži sadržaj koji je korisniku potreban. Jedan način je praćenjem *hiperveza* (engl. *hyperlink*) koji se nalaze na tekućoj veb stranici ili odlaskom na prethodno sačuvane veze. Međutim, šta ako se trenutno ne nalazimo ni na jednoj veb stranici ili nemamo sačuvane veze? Možemo potražiti sadržaj uz pomoć neke pretraživačke mašine, poput Google-ove.

Međutim, rekli smo da su pretraživačke mašine takode veb prezentacije, te je potrebno doći i do njih. Rešenje na sve ove probleme jeste pridruživanje jedinstvenih imena svim resursima na Internetu.

Definicija 1.3.3 — URI. *Jedinstveni identifikator resursa* (engl. *Uniform Resource Identifier*, skr. *URI*) predstavlja jedinstveni identifikator koji se dodeljuje veb sadržaju i važi za ceo svet.

Na osnovu URI-a, klijent može da pošalje HTTP protokol ka datom sadržaju, nakon čega se ostvaruje konekcija između tog klijenta i odgovarajućeg servera. Način razrešavanja ove komunikacije neće biti tema ovog teksta. Na slici 1.5 prikazano je na koji način URI dohvata veb sadržaj. Postoje dve vrste URI-a: URL-e i URN-i. Mi ćemo se fokusirati samo na URL-e.

Definicija 1.3.4 — URL. *Jedinstvena adresa resursa* (engl. *Uniform Resource Locator*, skr. *URL*) predstavlja najčešću formu identifikacije sadržaja zasnovanu na specifičnoj lokaciji sadržaja na određenom serveru.

URL ima sledeće osnovne delove:

- *Shema* (engl. *scheme*) koja opisuje protokol koji se koristi da bi se dohvatio resurs.
- DNS ime računara, odn. adresa servera na kojem se sadržaj nalazi.
- (Opcioni) Broj *porta* (engl. *port*) na kojem se vrši konekcija ka serveru. Obično se koristi ukoliko na serveru postoji više aplikacija koje očekuju komunikaciju sa različitim klijentima putem istog protokola (na primer, HTTP protokola). U tom slučaju je potrebno precizirati sa kojom aplikacijom želimo da komuniciramo i to se obavlja putem broja porta. Ukoliko se vrši konekcija putem HTTP protokola, pri čemu je broj porta izostavljen, obično se podrazumeva broj porta 80.
- Lokalno ime koje jedinstveno identifikuje sadržaj (što je obično ime datoteke na računaru na kome se sadržaj nalazi).

Na primer, URL identifikator koji identifikuje ovu skriptu i predstavlja vezu ka njoj je

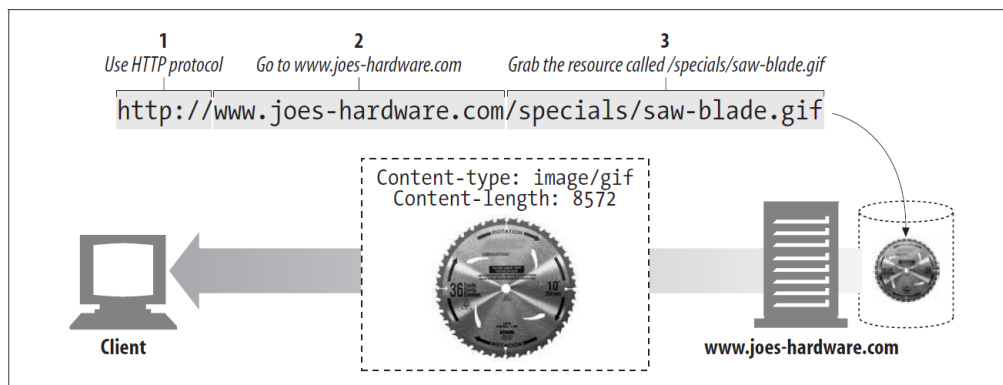
`http://www.math.rs/~nikola_ajzenhamer/kursevi/pzv/pzv.pdf`

Ovaj URL ima tri dela: protokol (*http*), DNS ime računara (*www.math.rs*) i ime/putanja datoteke (*~nikola_ajzenhamer/kursevi/pzv/pzv.pdf*), sa odgovarajućom interpunkcijom između delova.

1.3.1 Transakcije i poruke

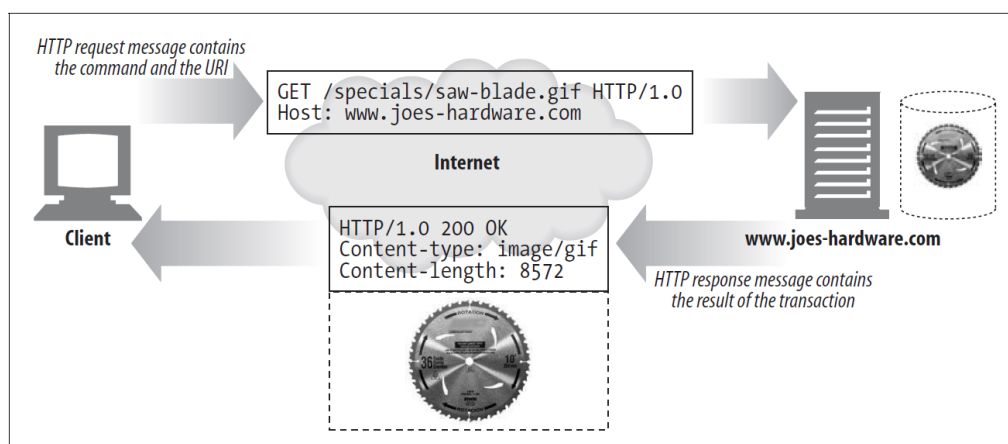
Hajde da detaljnije pogledamo kako klijent i server komuniciraju putem HTTP protokola. Kao što smo ranije opisali, klijent prvo šalje *zahtev* (engl. *request*) serveru, a zatim od servera dobija *odgovor* (engl. *response*). Upravo ove dve razmene čine jednu transakciju u HTTP protokolu, što je prikazano na slici 1.6. Istaknimo važnost ovog pojma narednom definicijom.

Definicija 1.3.5 — HTTP transakcija. *HTTP transakcija* (engl. *HTTP transaction*) sastoji se od HTTP zahteva od klijenta ka serveru i HTTP odgovora od servera ka klijentu.



Slika 1.5: URL-i specificuju protokol, server i lokalni resurs.

Definicija 1.3.6 — HTTP poruka. *HTTP poruka* predstavlja jedinicu komunikacije u HTTP transakciji. Sastoji se od formatiranih blokova podataka i može predstavljati bilo zahtev ili odgovor.



Slika 1.6: HTTP transakcije se sastoje od zahteva i odgovora.

HTTP poruke se sastoje od sledećih delova:

1. *Početna linija* (engl. *start line*) sadrži opis poruke.
2. *Zaglavlja* (engl. *header*) sadrže atribut koji se vezuju za poruku.
3. (Opciono) *Telo poruke* (engl. *message body*) koje sadrži podatke.

Početna linija i zaglavlja se sastoje od nekoliko linija teksta zapisanih u kodnoj shemi ASCII razdvojenih znacima za novi red (CRLF). Telo poruke može sadržati tekst, binarne podatke ili može biti prazno. Dodatno, telo je odvojeno od zaglavlja jednom praznom linijom.

Kao što smo rekli, sve HTTP poruke se dele u dve grupe: zahtevi i odgovori. Oba tipa poruka imaju osnovnu strukturu koju smo opisali. Ipak, u zavisnosti od vrste, sadržaj delova poruke se razlikuje. Tako za HTTP zahtev imamo osnovnu strukturu oblika

```
<metod> <url zahteva> <verzija>
<zaglavlja>
```

<telo>

dok za HTTP odgovor imamo osnovnu strukturu oblika (primetimo da se zahtev i odgovor razlikuju samo u početnoj liniji)

```
<verzija> <statusni kod> <statusna poruka>
<zaglavlja>
```

<telo>

U nastavku teksta govorimo detaljnije o svakom elementu opisanih struktura.

Zaglavlja

Zaglavlja se sastoje od nula ili više linija teksta. Zaglavlje je oblika

```
<ime>:[ ]<vrednost><CRLF>
```

Već smo napomenuli da se zaglavlja od tela zahteva odvajaju jednim CRLF karakterom. Neke verzije HTTP protokola zahtevaju da se određena zaglavlja uključe da bi zahtev ili odgovor bili validni.

HTTP specifikacije definiše neka polja zaglavlja. Naravno, aplikacije mogu definisati i svoja zaglavlja. HTTP zaglavlja se mogu klasifikovati u naredne kategorije: *opšta zaglavlja* (engl. *general header*), *zaglavlja zahteva* (engl. *request header*), *zahteva odgovora* (engl. *response header*), *zaglavlja podataka* (engl. *entity header*) i *proširena zaglavlja* (engl. *extension header*).

Opšta zaglavlja

Mogu se pojaviti i u zahtevu i u odgovoru. Oni daju osnovne informacije o poruci. Neka opšta zaglavlja su data u tabeli 1.1.

Tabela 1.1: Opšta zaglavlja i njihovi opisi.

<i>Zaglavlje</i>	<i>Opis</i>
Connection	Allows clients and servers to specify options about the request/response connection.
Date	Provides a date and time stamp telling when the message was created.
MIME-Version	Gives the version of MIME that the sender is using.
Upgrade	Gives a new version or protocol that the sender would like to "upgrade" to using.
Via	Shows what intermediaries (proxies, gateways) the message has gone through.

Zaglavlja zahteva

Sadrže više informacija o zahtevu. Server može koristiti ove informacije da bi prilagodio odgovor klijentu. Na primer, zaglavlja tipa **Accept** daju način klijentu da specifikuje šta žele, šta mogu da koriste, i, najvažnije od svega, šta ne žele. Uslovna zaglavlja omogućavaju da klijent postavi neka ograničenja na zahtev. Sigurnost zaglavlja predstavljaju jednostavan mehanizam za autentikaciju zahteva. Zaglavlja proksija omogućavaju rad sa proksijima. Neka zaglavlja zahteva su data u tabeli 1.2.

Tabela 1.2: Zaglavlja zahteva i njihovi opisi.

<i>Zaglavlje</i>	<i>Opis</i>
Client-IP	Provides the IP address of the machine on which the client is running.
From	Provides the email address of the client's user.
Host	Gives the hostname and port of the server to which the request is being sent.
Referer	Provides the URL of the document that contains the current request URI.
User-Agent	Tells the server the name of the application making the request.
Accept	Tells the server what media types are okay to send.
Accept-Charset	Tells the server what charsets are okay to send.
Accept-Encoding	Tells the server what encodings are okay to send.
Accept-Language	Tells the server what languages are okay to send.
Expect	Allows a client to list server behaviors that it requires for a request.
If-Match	Gets the document if the entity tag matches the current entity tag for the document.
If-Modified-Since	Restricts the request unless the resource has been modified since the specified date.
If-None-Match	Gets the document if the entity tags supplied do not match those of the current document.
If-Range	Allows a conditional request for a range of a document.
If-Unmodified-Since	Restricts the request unless the resource has not been modified since the specified date.
Range	Requests a specific range of a resource, if the server supports range requests.
Authorization	Contains the data the client is supplying to the server to authenticate itself.
Cookie	Used by clients to pass a token to the server—not a true security header, but it does have security implications.
Max-Forwards	The maximum number of times a request should be forwarded to another proxy or gateway on its way to the origin server—used with the TRACE method.
Proxy-Authorization	Same as Authorization, but used when authenticating with a proxy.
Proxy-Connection	Same as Connection, but used when establishing connections with a proxy.

Zaglavlja odgovora

Sadrže više informacija o odgovoru. U njima se klijentu dostavljaju informacije o tome ko šalje odgovor, mogućnosti servera ili specijalne instrukcije koje se tiču odgovora. Tako, na primer, bezbednosna zaglavlja predstavljaju *zahtev* (engl. *challenge*) od klijenta koja se tiču bezbednosti, a na koja klijent treba da odgovori slanjem odgovarajućih bezbednosnih zaglavlja u novom zahtevu. Neka zaglavlja odgovora su data u tabeli 1.3.

Tabela 1.3: Zaglavlja odgovora i njihovi opisi.

<i>Zaglavlje</i>	<i>Opis</i>
Age	How old the response is.
Public	A list of request methods the server supports for its resources.
Retry-After	A date or time to try back, if a resource is unavailable.
Server	The name and version of the server's application software.
Title	For HTML documents, the title as given by the HTML document source.
Warning	A more detailed warning message than what is in the reason phrase.
Proxy-Authenticate	A list of challenges for the client from the proxy.
Set-Cookie	Not a true security header, but it has security implications; used to set a token on the client side that the server can use to identify the client.
WWW-Authenticate	A list of challenges for the client from the server.

Zaglavlja podataka

Opisuju veličinu i sadržaj tela, ili sam veb sadržaj. S obzirom da i zahtevi i odgovori mogu da imaju sadržaj, ova zaglavlja se mogu javiti o obema vrstama poruka. Najjednostavniji tip zaglavlja podataka su informaciona zaglavlja koja govore o tipovima operacija koje se mogu primeniti nad sadržajem, kao i o lokaciji sadržaja. Zaglavlja tipa **Content** daju specifične informacije o sadržaju, kao što su tip, veličina i druge korisne informacije za njegovu obradu. Zaglavlja keširanja daju uputstva o tome kada i kako treba keširati sadržaj. Neka zaglavlja podataka su data u tabeli 1.4.

Tabela 1.4: Zaglavlja podataka i njihovi opisi.

<i>Zaglavlje</i>	<i>Opis</i>
Allow	Lists the request methods that can be performed on this entity.
Location	Tells the client where the entity really is located; used in directing the receiver to a (possibly new) location (URL) for the resource.
Content-Base	The base URL for resolving relative URLs within the body.
Content-Encoding	Any encoding that was performed on the body.
Content-Language	The natural language that is best used to understand the body.
Content-Length	The length or size of the body.
Content-Location	Where the resource actually is located.
Content-MD5	An MD5 checksum of the body.
Content-Range	The range of bytes that this entity represents from the entire resource.
Content-Type	The type of object that this body is.
Expires	The date and time at which this entity will no longer be valid and will need to be fetched from the original source.

Last-Modified	The last date and time when this entity changed.
---------------	--

Proširena zaglavlja

Ona nisu opisana HTTP specifikacijom, već su definisana specifikacijom aplikacije.

Zaglavlja se mogu definisati i u više linija tako što se naredne linije uvuku barem jednim karakterom razmaka ili tabulatora. Na primer,

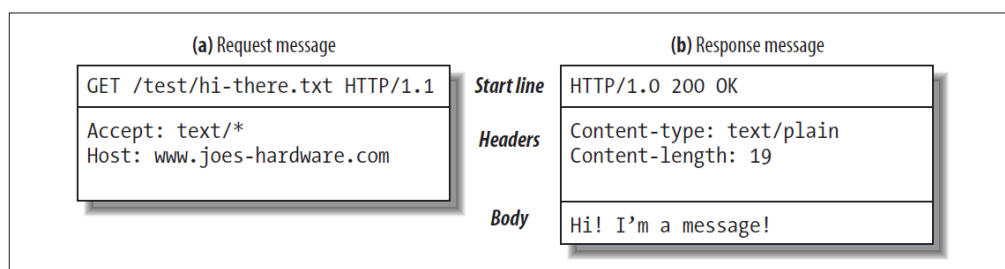
```
HTTP/1.0 200 OK
Content-Type: image/gif
Content-Length: 8572
Server: Test Server
    Version 1.0
```

U ovom primeru, HTTP odgovor zadrži zaglavlje sa imenom **Server** čija je vrednost razlomljena u dve linije. Očigledno, efekat ove akcije je u povećanoj čitljivosti.

Telo

Već smo rekli da telo HTTP poruke može biti prazno ili sadržati nekakve podatke. Obično se, ukoliko telo sadrži neke podatke, u poruku uključuju zaglavlja o tipu podataka i njihovoj dužini. Jedan takav primer dat je na slici 1.7.

HTTP poruke mogu da sadrže različite tipove digitalnih podataka: slike, video snimke, HTML dokumente, softverske aplikacije, transakcije kreditnim karticama, elektronsku poštu i mnoge druge.



Slika 1.7: Primeri HTTP zahteva i odgovora.

HTTP metodi

HTTP protokol podržava nekoliko različitih komandi zahteva koji se nazivaju HTTP metodi. Svaki HTTP zahtev ima tačno jedan metod.

Definicija 1.3.7 — HTTP metod. *HTTP metod* (engl. *HTTP method*) govori serveru koji akciju treba da preduzme.

HTTP specifikacija definiše skup najčešćih metoda. Neki od tih metoda sa njihovim objašnjenjima su:

- **GET:** Koristi se za potražnju imenovanog sadržaja od servera ka klijentu. Telo HTTP odgovora sadrži traženi sadržaj. Često serveri dodaju informaciju o tipu sadržaja i njegovoj veličini u zaglavlje odgovora.
- **HEAD:** Ponašanje je isto kao za GET metod, sa razlikom da server vraća samo HTTP zaglavlja iz odgovora za dati imenovani sadržaj. Telo odgovora je uvek prazno. Ovim

se omogućuje da klijent dohvati informacije o veb sadržaju, bez dohvaćanja samog sadržaja, ili ispitati da li sadržaj postoji. Dodatno, može se koristiti za ispitivanje da li je došlo do modifikacije prethodno dohvaćenog sadržaja. Razvijaoaci servera bi trebalo da obezbede da odgovor na **HEAD** metod vrati zaglavlja koja bi bila vraćena u slučaju **GET** metoda.

- **PUT**: Ovim metodom se vrši uskladištavanje podataka od klijenta u imenovani serverski sadržaj. Može se smatrati inverznom operacijom metoda **GET**. Semantika **PUT** metoda podrazumeva da server preuzme sadržaj iz tela zahteva i koristi ga bilo za kreiranje novog sadržaja ili ažuriranja sadržaja već postojećeg dokumenta.
- **POST**: Koristi se za slanje podataka od klijenta ka serverskoj aplikaciji za njihovo procesiranje. U praksi se najčešće koristi za slanje informacija iz veb formulara koje je korisnik uneo za dalju obradu na serveru. Treba biti svestan razlike između **PUT** metoda i ovog metoda — sadržaj koji se šalje metodom **PUT** se skladišti u datoteku, dok **POST** šalje podatke za rad serverske aplikacije.
- **TRACE**: Na putu od klijenta do servera, **HTTP** zahtev prolazi kroz različite proksi-servere, zaštitne zidove, druge aplikacije i sl. te svaka ova "prepreka" može izmeniti originalni **HTTP** zahtev. Korišćenjem **TRACE** metoda, poruka se prati kroz proksi-servere do ciljanog servera, i koristi se da klijent sazna kako zahtev izgleda kada dođe do servera. U telu **HTTP** odgovora od servera se nalazi celokupan potpis **HTTP** zahteva koji je stigao do njega.
- **OPTIONS**: Koristi se za određivanje metoda koji su dostupni na serveru, bilo su opštim ili specifičnim slučajevima. Ovim se obezbeđuje da klijentska aplikacija odredi najbolju strategiju za pristup različitim veb sadržajima bez da im zapravo pristupa. Dostupni metodi su nabrojani kao vrednost zaglavlja sa imenom **Allow**.
- **DELETE**: Predstavlja naredbu za brisanjem imenovanog sadržaja sa servera. Ipak, klijentskoj aplikaciji se ne garantuje da je operacija izvršena. **HTTP** specifikacija dozvoljava da server pregazi zahtev od klijenta bez da mu to kaže.

Trebalo bi da bude očigledno da metodi **POST** i **PUT** zahtevaju sadržaj u telu zahteva, dok ostali metodi to ne zahtevaju. Dodatno, server nije u nužnosti da implementira sve prethodno opisane metode. Da bi bio saglasan sa verzijom protokola **HTTP** 1.1, servera mora da implementira samo **GET** i **HEAD** metode za svoje sadržaje. Čak i kad server implementira sve metode, neki metodi će često imati ograničenu upotrebu. Na primer, serveri koji dozvoljavaju metode **DELETE** ili **PUT** često ne dozvoljavaju bilo kome da vrši izmenu ili dodavanje resursa. Ovakva podešavanja se implementiraju u konfiguraciji servera.

Takođe, s obzirom da je **HTTP** protokol dizajniran tako da se može lako nadograditi, serveri mogu da implementiraju i druge metode. Ovakve metode se nazivaju *proširene metode* (engl. *extension method*). Hajde da se nešto detaljnije pozabavimo **HTTP** metodima i njihovim svojstvima. Primer proširene metode bi mogao biti metod **LOCK** kojim se omogućava da server "zaključa" datoteku za menjanje dok klijent ne završi sa njom.

Definicija 1.3.8 — Bezbedan metod. *Bezbedan metod* (engl. *safe method*) predstavlja **HTTP** metod čijom obradom se na serveru ne izvršava nikakva akcija promene.

Pod *akcijom promene* podrazumevamo da neće doći do promene na serveru, bilo njegove konfiguracije ili ponašanja, ali češće do promene podataka koje se nalaze na serveru. Na primer, kupovinom knjige iz veb knjižare, na serveru se mogu izvršiti različite akcije promene, kao što su, na primer: dodavanje nove transakcije, ažuriranje baze podataka o raspoloživosti knjiga, povlačenje novca sa kreditne kartice i sl.

Metodi **GET** i **HEAD** treba da budu bezbedni. Naravno, ne postoji garancija da bezbedan metod zaista neće dovesti do akcije promene — to je zaduženje veb razvijaoca. Nebezbedni metodi treba da sugerišu razvijaocima HTTP aplikacija da upozore korisnike da je akcija koju će preuzeti nebezbedna. U prethodnom primeru, aplikacija bi mogla da prikaze prozor u kojem se traži od korisnika da potvrdi da želi da izvrši transakciju.

URL zahteva

Kao što smo rekli, da bismo jedinstveno identifikovali željeni resurs, potrebno je da navedemo njegovu preciznu lokaciju datu URL-om. Često je moguće dati samo lokalno ime ukoliko se komunicira direktno sa serverom jer server može da pretpostavi sebe za ime računara, kao i broj porta.

Verzija

Definicija 1.3.9 — Verzija. Verzija predstavlja oznaku verzije HTTP protokola koji se koristi.

Ovim se postiže način kako da HTTP aplikacije govore jedna drugoj kojoj verziji protokola treba da se pridržavaju. Verzija je oblika

HTTP/<broj verzije>.<podbroj verzije>

gde su <broj verzije> i <podbroj verzije> celi brojevi. Verzija treba da govori koji je najveći broj verzije HTTP protokola kojem aplikacija može da se povinuje. Do sada je bilo reči o verziji HTTP 1.1, dok ćemo se posebno osvrnuti na novine koje donosi verzija HTTP 2.0 u podsekciji 1.3.3.

Statusni kod i statusna poruka

Predstavlja trocifreni broj koji opisuje rezultat izvršavanja zahteva na serveru. Na primer, u početnoj liniji odgovora "HTTP/1.0 200 OK", statusni kod je 200.

S obzirom da se zahtev koji šaljemo serveru može završiti na različite načine, to postoje i razni statusni kodovi. Svi oni su organizovani u grupe i dati su u tabeli 1.5. Statusni kodovi su grupisani na osnovu značenja. Za svaku kategoriju u tabeli dat je opšti interval, kao i interval u kojem se nalaze kodovi koji su predefinisani HTTP specifikacijom. Prva cifra govori o opštem statusu zahteva (na primer, da li je zahtev obrađen kako treba ili je došlo do neke greške). Preostale cifre govore detaljnije o samoj grešci.

Tabela 1.5: Kategorije statusnih kodova i njihovi opšti intervali i intervali koji sadrže kodove definisane HTTP specifikacijom.

<i>Opšti interval</i>	<i>Predefinisani interval</i>	<i>Kategorija</i>
100 – 199	100 – 101	Informacioni kodovi
200 – 299	200 – 206	Kodovi uspešnosti
300 – 399	300 – 305	Redirekcioni kodovi
400 – 499	400 – 417	Kodovi klijentskih grešaka
500 – 599	500 – 505	Kodovi serverskih grešaka

Kao što vidimo iz prethodne tabele, HTTP specifikacija definiše relativno mali broj statusnih kodova. To znači da prilikom rada sa serverom, možemo dobiti statusni kod, na primer 530. Iako on nije deo HTTP specifikacije, ipak bi ga trebalo tretirati kao statusni kod iz kategorije kodova serverskih grešaka.

Statusna poruka predstavlja čitljivu reprezentaciju statusnog koda. Njen sadržaj je ceo tekst do kraja početne linije. Ipak, ukoliko dve poruke imaju isti statusni kod, ali potpuno drugačije statusne poruke, one bi trebalo biti jednako tretirane, bez obzira na to što statusne poruke sugerišu drugačije. Postoji konvencija dodeljivanja statusnih poruka odgovarajućim statusnim kodovima i dobra je praksa pratiti te smernice. Naravno, ove sugestije ne predstavljaju tvrda pravila, već su upravo samo to — sugestije.

Tabela 1.6: Statusni kodovi zadati HTTP specifikacijom, njihove statusne poruke i objašnjenja.

<i>Statusni kod</i>	<i>Statusna poruka</i>	<i>Objašnjenje</i>
100	Continue	Indikuje da je inicijalni deo zahteva prihvaćen i da klijent može nastaviti dalje. Nakon slanja ovakvog odgovora, server treba da reaguje nakon dobijanja zahteva. Koristi se za optimizovanje HTTP zahteva kada klijent želi prvobitno da proveriti da li će server prihvatiti sadržaj pre nego što ga zapravo i pošalje.
101	Switching protocol	Indikuje da server menja protokol, kako je specifikovano od strane klijenta, na onaj koji je specifikovan u zaglavlju sa imenom Upgrade.
200	OK	Zahtev je u redu, telo sadrži traženi sadržaj. Služi za zahteve koji kreiraju sadržaj na serveru (na primer, PUT). Telo odgovora bi trebalo da sadrži URL-e za referenciranje kreiranog sadržaja, zajedno sa zaglavljem sa imenom Location koji sadrži najspecifičniju referencu. Server mora da kreira datoteku pre slanja ovog statusnog koda.
201	Created	Zahtev je prihvaćen, ali server još uvek nije preduzeo nikakvu akciju. Nema garancije da će server kompletirati zahtev; ovo samo govori da je zahtev delovao u redu kada ga je server prihvatio. Server bi trebalo da uključi opis statusa zahteva u telo odgovora, i eventualno procenu kada će biti izvršen zahtev (ili referencu ka ovim informacijama). Informacije koje se sadrže u zaglavlju podataka (videti deo Zaglavlja ispod) dolaze ne od originalnog servera već od kopije sadržaja. Ovo se može desiti ako je međuserver imao kopiju sadržaja, ali nije mogao da validira ili nije validirao metainformacije (zaglavlja) koje je poslao u vezi sa sadržajem.
202	Accepted	Odgovor sadrži početnu liniju i zaglavlja, ali ne i telo. Najčešće se koristi za ažuriranje pregledača bez prebacivanja na novi dokument (na primer, osvežavanje veb formulara).
203	Non-Authoritative Information	Govori pregledaču da obriše sadržaj elemenata HTML formulara na trenutnoj stranici.
204	No Content	Klijenti mogu tražiti deo dokumenta korišćenjem specijalnih zaglavlja. Ovaj statusni kod govori da je zahtev za delom dokumenta uspešno izveden.
205	Reset Content	
206	Partial Content	

300	Multiple Choices	Vraća se kada klijent zahteva URL koji referiše na više sadržaja, kao što je server koji sadrži srpsku i englesku verziju HTML dokumenta. Ovaj kod se vraća zajedno sa listom opcija; korisnik može da odabere koji dokument želi. Server može da priloži najpoželjniji URL u zaglavlju sa imenom Location. Koristi se kada je sadržaj na datom URL-u pomeren na drugu lokaciju. Odgovor treba da sadrži URL lokacije na koju je sadržaj prebačen u zaglavlju sa imenom Location.
301	Moved Permanently	Poput koda 301, sa razlikom da je URL zadat u zaglavlju sa imenom Location privremenog tipa. Budući zahtevi bi trebalo da koriste stari URL. Koristi se da se kaže klijentu da bi resurs trebalo da se dohvata korišćenjem drugog URL-a, navedenog u zaglavlju sa imenom Location. Njegovo glavno korišćenje je da dozvoli odgovore za POST zahteve za usmeravanje klijenta ka sadržaju.
302	Found	Klijenti mogu da šalju uslovne zahteve, na primer, da zatraže sadržaj GET metodom ukoliko nije menjan skorije, ovaj kod se šalje kao indikator da sadržaj nije menjan. Odgovori koji šalju ovaj kod treba da imaju prazno telo.
303	See Other	Koristi se za indicaciju da se sadržaju mora pristupati preko proksija. Lokacija proksija je data u zaglavlju sa imenom Location. Napomenimo da je ovo naznaka klijentu da koristi proksi za taj konkretan resurs, a ne uopšteno.
304	Not Modified	Ovaj statusni kod nije specifikovan u HTTP specifikaciji.
305	Use Proxy	Poput koda 302.
306	(Unused)	
307	Temporary Redirect	
400	Bad Request	Služi za obaveštenje klijentu da je poslat loš zahtev.
401	Unauthorized	Vraća se zajedno sa odgovarajućim zaglavljima koja zahtevaju od klijenta da izvrši autentikaciju pre nego što dobije pristup sadržaju.
402	Payment Required	Trenutno nije korišćen, ali je rezervisan za buduću upotrebu.
403	Forbidden	Obaveštenje klijenta da je server odbio da izvrši zahtev. Server može da uključi u telo odgovora razlog za odbijanje, ali često se to ne radi.
404	Not Found	Koristi se kao indicacija da server ne može da pronađe traženi URL. Često se prosleđuje sadržaj u telu odgovora koji klijent treba da prikaže korisniku.
405	Method Not Allowed	Koristi se kada klijent pokuša da izvrši metod koji nije implementiran na serveru. Zaglavlje sa imenom Allow bi trebalo da se uključi, a njegova vrednost da se postavi na listu metoda koje server može da podrži.

406	Not Acceptable	Klijenti mogu da naznače koje tipove sadržaja mogu da prihvate. Ovaj kod se koristi kad server nema sadržaj koji odgovara URL-u koji odgovara klijentu.
407	Proxy Authentication Required	Poput koda 401, ali se koristi od proksi-servera koji zahtevaju autentikaciju za sadržaj.
408	Request Timeout	Ako klijent potroši previše vremena da završi zahtev, server može da pošalje ovaj kod i da zatvori konekciju. Dužina zavisi od podešavanja servera.
409	Conflict	Indikuje neku vrstu konflikta koju zahtev izaziva nad sadržajem. Odgovor treba da sadrži detalje o konfliktu u svom telu.
410	Gone	Poput koda 404, ali uz signalizaciju da je server nekada imao traženi sadržaj. Obično ga koriste sistem administratori da obaveste klijente kada je sadržaj pomeren.
411	Length Required	Koristi se kada server zahteva da klijent dostavi zaglavlje sa imenom Content-Length u svom zahtevu.
412	Precondition Failed	Koristi se ako klijent napravi uslovni zahtev i jedan od uslova ne uspe. Uslovni zahtevi se javljaju kada klijent uključi zaglavlje sa imenom Expect.
413	Request Entity Too Large	Koristi se kada klijent šalje telo sadržaja koje je veće nego što server može ili želi da procesira.
414	Request URI Too Long	Koristi se kada klijent šalje zahtev sa URL-om koji je veći nego što server može ili želi da obradi.
415	Unsupported Media Type	Koristi se kada klijent šalje sadržaj one vrste koji server ne razume ili ne podržava.
416	Requested Range Not Satisfiable	Koristi se kada klijent šalje zahtev koji traži određeni deo sadržaja i taj deo je nevažeći ili nije mogao biti ispunjen.
417	Expectation Failed	Koristi se kada zahtev koji sadrži očekivanje u zaglavlju sa imenom Expect server nije mogao zadovoljiti.
500	Internal Server Error	Koristi se kada server dođe do greške koja onemogućuje server da ispuni zahtev.
501	Not Implemented	Koristi se kada klijent napravi zahtev koji je van mogućnosti servera (na primer, korišćenje metoda zahteva koji server ne podržava).
502	Bad Gateway	Koristi se kada server koji deluje kao posrednik susreće lažni odgovor od sledeće veze u lancu odgovora na zahtev.
503	Service Unavailable	Koristi se da naznači da server trenutno ne može da ispuni zahtev, ali da će biti u mogućnosti u budućnosti. Ako server zna kada će sadržaj postati dostupan, onda može uključiti zaglavlje sa imenom Retry-After u odgovoru.

504	Gateway Timeout	Slično kodu 408, osim što odgovor dolazi iz proksija koji je istekao čekajući na svoj zahtev od drugog servera.
505	HTTP Version Not Supported	Koristi se kada server primi zahtev u verziji protokola koji ne može ili neće podržati. Neke serverske aplikacije biraju da ne podržavaju starije verzije protokola.

1.3.2 HTTPS

1.3.3 HTTP 2.0

Literatura za ovu oblast

- [GG05] A. Grosskurth i M. W. Godfrey. „A reference architecture for web browsers”. U: Washington: IEEE Computer Society, 2005., strane 661–664. ISBN: 0769523684.
- [05] *Proceedings of the 21st IEEE international conference on software maintenance (ICSM'05), Volume 00, held in Budapest, Hungary, September 25-30, 2005.* Washington: IEEE Computer Society, 2005. ISBN: 0769523684.
- [Tan02] A. Tanenbaum. *Computer Networks*. 4th. Prentice Hall Professional Technical Reference, 2002. ISBN: 0130661023.
- [Tot+02] B. Totty i drugi. *Http: The Definitive Guide*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2002. ISBN: 1565925092.

2. JavaScript

JavaScript je nastao 1995. godine kao jezik za veb pregledač Netscape Navigator, pomoću kojih je bilo moguće dodati programske sposobnosti veb stranicama. Od tada je jezik usvojen od strane svih modernih veb pregledača. Vremenom je napisan dokument kojim se standardizuje ponašanje veb pregledača za koje se tvrdi da podržavaju JavaScript, i taj standard se naziva ECMAScript standard. Mi ćemo koristiti oba termina podjednako, s obzirom da to i jesu dva naziva za isti jezik.

U ovom poglavlju će se najpre upoznati sa jednostavnijim elementima JavaScript jezika, a zatim ćemo uroniti u nešto kompleksnije elemente. Na kraju ovog poglavlja, čitalac bi trebalo da bude u stanju da razume osnovno funkcionisanje jezika JavaScript i da bude osposobljen za pisanje programa u ovom jeziku. U nastavku kursa, JavaScript će nam služiti kao jezik za sve programske delove naših zadataka.

2.1 Izvršavanje JavaScript koda

Pre nego što započnemo diskusiju o sintaksi i semantici programskog jezika JavaScript, potrebno je da razumemo kako je moguće da izvršimo izvorni kod koji budemo napisali. JavaScript kod se izvršava u okviru *okruženja za izvršavanje* (engl. *engine*) koje nam omogućava bezbedno okruženje za izvršavanje koda. Sve definicije promenljivih, funkcija i dr. se izvršavaju u okviru ovog okruženja, odvojenog od operativnog sistema. Okruženje za izvršavanje predstavlja kompleksan softverski sistem koji se sastoji od mnogo delova, a njegov najznačajniji deo jeste *interpreter* (engl. *interpreter*) koji služi za kompiliranje¹ i izvršavanje JavaScript koda. Jedan od najpopularnijih okruženja za izvršavanje jeste V8². Suštinski, V8 predstavlja C++ program kojem se prosleđuje JavaScript, koji on zatim kompilira i izvršava. Zadaci koje V8 obavlja su:

¹Termin *kompiliranje* u kontekstu izvršavanja JavaScript jezika nema isto značenje kao kompiliranje jezika poput C++ čiji je finalni proizvod mašinski kod koji se izvršava na operativnom sistemu.

²Zvanična veb prezentacija se može pronaći na adresi <https://v8.dev/>.

- Kompiliranje i izvršavanje JavaScript koda.
- Upravljanje stekom poziva funkcija.
- Upravljanje dinamičkom memorijom.
- Sakupljanje otpadaka.
- Obezbeđivanje tipova podataka, operatora, objekata i funkcija.

Predimo sada na načine za izvršavanje JavaScript koda:

1. *JavaScript na klijentu.* Ovaj način predstavlja tradicionalni pristup korišćenja JavaScript jezika u kojem se on koristi za razvoj veb aplikacija koje se izvršavaju u okviru veb pregledača na klijentskom računaru. Svi savremeni veb pregledači imaju kao deo svog sistema okruženje za izvršavanje (na primer, Google Chrome ima V8) koji se koristi za izvršavanje JavaScript koda.
2. *JavaScript na serveru.* Ovaj način predstavlja relativno noviji pristup korišćenja JavaScript jezika u kojem se on koristi za razvoj veb aplikacija koje se izvršavaju u okviru operativnog sistema na serverskom računaru. Savremeni razvoj serverskih veb aplikacija se sve više usmerava ka JavaScript jeziku, pri čemu je najpopularniji izbor softverski sistem Node.js³ koji je izgrađen nad V8 okruženjem za izvršavanje. Korišćenjem Node.js sistema, moguće je konstruisati serverske aplikacije koje su se tradicionalno razvijale korišćenjem, na primer, programskih jezika PHP ili Python.

JavaScript na klijentu

Da bismo izvršili neki JavaScript kod u veb pregledaču, potrebno je da veb pregledaču otvori `html` datoteku koja u svojoj strukturi sadrži JavaScript kod. Međutim, taj kod se ne može javiti bilo gde u HTML kodu, već se mora navesti kao sadržaj elementa `script`, kao u narednom kodu:

```
1 <!DOCTYPE html>
2 <html>
3
4 <head>
5   <meta charset="UTF-8">
6   <title>JavaScript</title>
7 </head>
8
9 <body>
10   <script>
11     // JavaScript kod ide ovde
12     var x = 1;
13   </script>
14 </body>
15
16 </html>
```

Element `script` se ipak može naći bilo gde u sadržaju elemenata `head` ili `body`. Štaviše, možemo imati više `script` elemenata.

Alternativno, možemo JavaScript kod pisati u eksternoj datoteci sa ekstenzijom `js`. Ovakva datoteka se zatim uključuje u HTML kod ponovo pomoću elementa `script`, navođenjem putanje do te datoteke kao vrednost atributa `src`, kao u narednom kodu:

Datoteka `index.html`:

```
1 <!DOCTYPE html>
2 <html>
```

³Zvanična veb prezentacija se može pronaći na adresi <https://nodejs.org/en/>.

```
3
4 <head>
5   <meta charset="UTF-8">
6   <title>JavaScript</title>
7 </head>
8
9 <body>
10  <script src="index.js"></script>
11 </body>
12
13 </html>
```

Datoteka `index.js`:

```
1 // JavaScript kod ide ovde
2 var x = 1;
```

U oba slučaja će veb pregledač, čim naide na element `script`, proslediti JavaScript kod JavaScript interpreteru koji taj kod odmah izvršava. Ovo je važna napomena koju uvek treba imati na umu. Ako postoji JavaScript kod koji menja neki deo veb stranice, onda je potrebno da je taj deo stranice definisan u HTML kodu pre tog JavaScript koda. U suprotnom, može doći do pojave grešaka.

JavaScript na serveru

U poglavlju 5 detaljno ćemo diskutovati o razvoju serverskih veb aplikacija i načinima za njihovo izvršavanje, pod odgovarajućoj arhitekturom koja nam se prirodno nameće u radu sa Node.js sistemom. Za sada, prikazaćemo samo elementarno izvršavanje JavaScript koda na serveru.

Nakon instalacije Node.js sistema, na raspolaganju nam je program `node` koji se pokreće iz komandne linije. Ukoliko ga pokrenemo bez argumenata, biće nam prikazan Node.js *REPL* (skraćeno od *Read-Evaluate-Print-Loop*), u kojem možemo da kucamo kod:

```
C:\WINDOWS\system32>node
Welcome to Node.js v12.4.0.
Type ".help" for more information.
>
```

Izvršavanje koda se vrši unošenjem naredbi jezika kada vidimo oznaku `>`. REPL takođe podržava izračunavanje proizvoljnih izraza:

```
C:\WINDOWS\system32>node
Welcome to Node.js v12.4.0.
Type ".help" for more information.
> var x = 1;
undefined
> x
1
> .exit
```

```
C:\WINDOWS\system32>
```

Problem sa ovim načinom jeste što nam je najčešće nezgodno da kucamo kod u terminalu; preferiramo korišćenje tekstualnih editora, kao što je Visual Studio Code⁴. Dodatno, jednom kada završimo rad u REPL-u, sav kod koji smo napisali se gubi. Umesto toga,

⁴Zvanična veb prezentacija se može pronaći na adresi <https://code.visualstudio.com/>.

bolje je sačuvati kod u neku datoteku, na primer, `kod.js`, a zatim taj kod izvršiti pozivom programa `node` i prosleđivanjem putanje do datoteke kao njegov argument:

```
C:\WINDOWS\system32>node kod.js
```

2.2 Osnovni elementi jezika

Ova sekcija je posvećena prikazivanju osnovnih elemenata jezika JavaScript, kao što su tipovi podataka, literali, definisanju promenljivih, ispisivanju vrednosti u konzoli i poređenju vrednosti po jednakosti.

2.2.1 Tipovi i operatori

Brojevi

Najjednostavniji tip podataka je *numerički* (engl. *number*). Svi numerički tipovi su zapisani u 64 bita. Nad numeričkim tipovima su definisani standardni binarni operatori `+`, `-`, `*`, `/`, `%`, `<`, `<=`, `>`, `>=` i unarni operator `-`, sa standardnim pravilima za prednost. Zagrade (`(` i `)`) mogu da promene prednost izračunavanja. Takođe, podrazumeva se leva asocijativnost. JavaScript podržava:

- celobrojne vrednosti — `0`, `7`, `-42`, ...
- razlomljene vrednosti — `9.81`, `2.998e8` ($= 2.998 \cdot 10^8$), ...
- "beskonačnosti" — `Infinity`, `-Infinity`
- `NaN` — označava da "nije broj" (skr. *Not a Number*), iako je njegov tip numerički. Ovaj rezultat možemo dobiti ako, na primer, pokušamo da izračunamo `0 / 0` ili `Infinity - Infinity`, ili bilo koju drugu nedefinisanu operaciju.

Niske

Sledeći osnovni tip su *niske* (engl. *string*). Niske se koriste za reprezentaciju teksta. Kodna shema koja se koristi je Unicode. Možemo konstruisati literal ovog tipa pomoću jednostrukih ili dvostrukih navodnika, kao i "nakošenih" navodnika (tzv. *šablon-literali* (engl. *template literals*)). Na primer:

- `'Programiranje u JavaScript-u je kul!'`
- `"Koriscenje jQuery-a je bolje!"`
- ``A tek da vidis Angular!``

Treba voditi računa o tome da kad započnemo literal niske jednim od ova tri karaktera navodnika, prvi sledeći put kada se nađe na taj karakter, to će se smatrati krajem niske. Zato ne možemo direktno ugnežđavati, na primer, jednostruke navodnike, već ih moramo *označiti* (engl. *escape*). Označavanje se vrši stavljanjem karaktera `\` ispred karaktera koji želimo da označimo. U narednom primeru, prva linija predstavlja neispravno, a druga linija ispravno ugnežđavanje navodnika:

```
1 'A onda je rekao: 'Uradicu to!'' // Neispravno
2 'A onda je rekao: \'Uradicu to!\'' // Ispravno
```

Ako želimo da imamo višelinijske niske, to možemo uraditi na dva načina:

- Korišćenjem označenog karaktera za novi red `\n` na mestu gde želimo prelomiti u novi red. Na primer:

```
1 'Ovo je sve prva linija\nA odavde kreće druga linija'
```

- Korišćenjem šablon-literal a i prelamanjem tasterom za novi red na mestu gde želimo prelomiti u novi red. Na primer,


```
1 `Ovo je sve prva linija`
2 `A oдавde kreće druga linija`
```

Niske se mogu *nadovezivati* (engl. *concatenate*) operatorom `+`. Na primer, niska `'nadovezivanje'` se može dobiti izračunavanjem narednog izraza:

```
1 `na' + 'do' + 'vezivanje'
```

Poređenje niski pomoću standardnih operatora za poređenje `<`, `<=`, `>` i `>=` se vrši tako što se niske upoređuju leksikografski.

Šablon literali imaju još jedno zanimljivo svojstvo. U njih se mogu ugnežđavati razni drugi izrazi. Na primer, pri izračunavanju izraza

```
1 `Dva plus dva je jednako ${2 + 2}`
```

prvo će se izračunati vrednost izraza zapisan između `${` i `}`, zatim će se njegova vrednost konvertovati u nisku, i na kraju, ta niska će biti ugnežđena na tu poziciju. Dakle, prethodni izraz će biti izračunat u nisku

```
1 `Dva plus dva je jednako 4`
```

Bulove vrednosti

JavaScript takođe podržava izračunavanja koja proizvode Bulove vrednosti — `true` (tačno) i `false` (netačno). Binarni operatori poređenja regularno proizvode Bulove vrednosti nakon izračunavanja. Kao i u drugim programskom jezicima, Bulove vrednosti se najčešće koriste kao uslovi u naredbama grananja, petlji, i dr.

Baratanje Bulovim vrednostima se može jednostavno obaviti korišćenjem standardnih binarnih operatora `&&` (konjunkcije), `||` (disjunkcije), unarnog operatora `!` (negacije) i ternarnog operatora `?:` (uslovni operator).

Nedostajuće vrednosti

Postoje dve specijalne vrednosti, `null` i `undefined`, koje se koriste da označe odsustvo "vrednosti koje ima značenje". One same predstavljaju vrednosti, ali ne nose nikakvu informaciju. Mnoga izračunavanja u jeziku JavaScript koja ne proizvode vrednosti koje imaju značenje (u nastavku poglavlja ćemo se upoznati sa njima) proizvode `undefined` jednostavno zato što moraju da dobiju nekakvu vrednost.

Operator `typeof`

Operator `typeof` kao argument dobija proizvoljan izraz, a rezultat njegove primene na taj izraz jeste niska koja sadrži naziv tipa izraza nad kojim se primenjuje:

```
1 typeof 1;           // 'number'
2 typeof 'Tekstualni sadržaj'; // 'string'
3 typeof true;        // 'boolean'
4 typeof undefined;   // 'undefined'
5 typeof null;        // 'object'
```

Kao što vidimo, iako smo rekli da izrazi `undefined` i `null` nose isto značenje, njihovi tipovi su očigledno različiti. Ovo će nam biti važna napomena za nadalje i posebno, kada budemo govorili o konceptu *objekata* u JavaScript jeziku.

Implicitne konverzije

JavaScript je programski jezik koji može da "proguta" razne programske izraze koji uključuju osnovne tipove i da izračuna od njih nekakve vrednosti, koje možda ne bismo očekivali. Na primer:

- Vrednost izraza `8 * null` je `0`.
- Vrednost izraza `'5' - 1` je `4`.
- Vrednost izraza `'5' + 1` je `'51'`.
- Vrednost izraza `'pet' * 2` je `NaN`.
- Vrednost izraza `!''` je `true`.

Objašnjenje za ovakvo ponašanje leži u *implicitnoj konverziji*. Naime, kada se nekom operatoru proslede tipovi koji su različiti (primeri 1 – 4) ili tip koji on ne očekuje (primer 5), JavaScript će pokušati da, nekim komplikovanim mehanizmom, konvertuje tipove tako da može da primeni operator na njih. U slučaju da to ne uspe, rezultat je `NaN`. Na primer:

- U prvom primeru, prazna vrednost `null` se konvertuje u broj `0`, pa se izračunava množenje brojeva `8 * 0`.
- U drugom primeru, vrednost niske `'5'` se konvertuje u broj `5`, pa se izračunava oduzimanje brojeva `5 - 1`.
- U trećem primeru, vrednost broja `1` se konvertuje u nisku `'1'`, pa se izračunava nadovezivanje niski `'5' + '1'`.
- U četvrtom primeru, vrednost niske `'pet'` se ne može mapirati u brojevanu vrednost na očigledan način (isto važi i za `undefined`), pa ta konverzija rezultuje vrednošću `NaN`, pa se izračunava množenje `NaN * 2`.
- U petom primeru, vrednost niske `''` se konvertuje u `false`, pa se izračunava negacija `!false`.

Poređenje po jednakosti i nejednakosti

Jedan slučaj implicitne konverzije koji treba posebno razmotriti jeste kada se dve vrednosti porede po jednakosti ili nejednakosti. Kada se dve vrednosti istog tipa porede operatorom `==`, rezultat je jednostavno predvideti: rezultat će biti `true` ako su vrednosti iste, osim u slučaju vrednosti `NaN` (videti ispod). Međutim, kada se tipovi razlikuju, onda nije jednostavno i, štaviše, često je zbunjujuće utvrditi rezultat poređenja.

U slučaju poređenja vrednosti `null` i `undefined`, njihovo međusobno poređenje će proizvesti `true`. Poređenje sa nekom drugom vrednošću će proizvesti `false`. Ovo može biti korisno za testiranje da li je neka vrednost smislena ili ne.

Vratimo se na slučaj poređenja dve vrednosti različitih tipova. Šta ukoliko bismo želeli da poredimo da li je neka vrednost baš `false`? Prva ideja jeste da je poredimo pomoću operatora `==` sa vrednošću `false`. Međutim, naredni primer nam pokazuje da to neće raditi dobro:

- Vrednost izraza `false == 0` je `true`.

Dakle, nismo dobili željeni efekat. Želeli bismo da nekako "preciznije" poredimo vrednosti. U tu svrhu, u jeziku JavaScript postoji operator `===` koji pored toga što proverava da li su dve vrednosti jednake, proverava i da li su te vrednosti istog tipa! Ovim rešavamo prethodni problem, što naredni primer ilustruje:

- Vrednost izraza `false === 0` je `false`.
- Vrednost izraza `false === false` je `true`.

Poređenje dve `NaN` vrednosti će u jeziku JavaScript uvek proizvesti vrednost `false`. Smisao ovoga je da, s obzirom da `NaN` predstavlja rezultat neodređenog izračunavanja, ono ne može biti jednako nekom "drugom" neodređenom izračunavanju. Za ispitivanje da li je vrednost izraza `NaN`, može se iskoristiti funkcija `Number.isNaN`, koja kao argument prihvata neki izraz, a rezultat je `true` ukoliko taj rezultat proizvodi `NaN`, a `false` inače.

Izračunavanje kratkog spoja

Bulovi operatori `&&` i `||` imaju jedno zanimljivo svojstvo koje dolazi do izražaja kada se primenjuju nad vrednostima čiji tipovi nisu Bulove vrednosti:

- Operator `||` će vratiti vrednost levo od njega onda kada je tu vrednost moguće konvertovati u `true`. Inače, vraća vrednost desno od njega. Na primer,
 - Vrednost izraza `null || 'korisnik'` je `'korisnik'`.
 - Vrednost izraza `'ime' || 'prezime'` je `'ime'`.
- Operator `&&` će vratiti vrednost levo od njega onda kada je tu vrednost moguće konvertovati u `false`. Inače, vraća vrednost desno od njega. Na primer,
 - Vrednost izraza `'' && 42` je `''`.
 - Vrednost izraza `'ti' && 'ja'` je `'ja'`.

Ovakvo ponašanje se naziva *izračunavanje kratkog spoja* (engl. *short-circuit evaluation*). Ono se najčešće koristi u slučaju kada je potrebno iskoristiti neku vrednost, ali ukoliko ona nije dostupna, možemo koristiti neku drugu, na primer, podrazumevanu:

```
1 var port = PORT_NUM || 3000;
```

Ukoliko promenljiva `PORT_NUM` nema vrednost (ili je implicitno konvertibilna u `false`), tada će vrednost promenljive `port` biti `3000`. U suprotnom će vrednost promenljive `port` biti jednaka vrednosti promenljive `PORT_NUM`

Slično tome, izračunavanje kratkog spoja se može koristiti za osiguravanje da neka akcija ne dovede do nepoželjnih bočnih efekata:

```
1 var condition =  
2   !(denom == 0 || (num == Infinity || num == 0) && denom == Infinity);  
3  
4 if (condition && num / denom) {  
5   // ensures that calculating num/denom never results in NaN  
6 }
```

2.2.2 Programske strukture

Kao što smo videli do sada, osnovna struktura programa koja ne pravi *bočne efekte* je *izraz*. Ipak, kako ćemo mi praviti programe koji su složeniji od jednostavnog izračunavanja izraza, potrebne su nam *naredbe* koje imaju bočne efekte, kontrole toka, funkcije, i dr.

Komentari

U programskom jeziku JavaScript, jednolinijski komentari se navode iza `//`, dok se višelinjski komentari navode između `/*` i `*/`.

Uvođenje promenljivih

Promenljive se uvode u *domet* (engl. *scope*) pomoću naredbe dodele. Ona može imati oblik kao u narednom primeru:

```
1 let imePromenljive = 7 * 7;
```

Ključna reč `let` označava da će ova naredba dodeliti vrednost novoj promenljivoj čiji je identifikator zadat sa `imePromenljive`. Ukoliko samo deklariramo novu promenljivu, bez defisanja njene vrednosti, a zatim zatražimo njenu vrednost, kao rezultat dobićemo `undefined`. Možemo definisati više vrednosti odvojenih zapetom:

```
1 let a = 7, b = 42;
```

Pored ključne reči `let`, postoje još dve ključne reči za uvođenje promenljivih: `var` i `const`. Ključnom reči `var` se postiže isti efekat kao sa `let` (uz dodatne razlike, videti paragraf "Domet važenja promenljivih"), a ključnom reči `const` uvodimo konstantne promenljive, tj. one promenljive kojima samo jednom možemo dodeliti vrednost. Ono što je bitno da razumemo jeste da ako je promenljiva uvedena rečju `const`, to ne znači da vrednost te promenljive nije moguće promeniti u nekim slučajevima. Na primer, ukoliko je vrednost konstantne promenljive `x` objekat⁵, nije moguće dodeliti novu vrednost promenljivoj `x`, ali je moguće izmeniti unutrašnju strukturu objekta koji joj je dodeljen.

Identifikatori mogu sadržati karaktere slova, brojeva, `_` i `$`, ali ne smeju počinjati brojem. Dodatno, zbog specijalnih značenja koje nose ključne reči, one se ne mogu koristiti kao identifikatori.

Pored operatora dodele `=`, postoje i `+=`, `-=`, `*=`, `/=`, prekisni i infiksni `++` i `--`.

Kontrole toka

Pored standardnog, linearnog toka programa, u programskom jeziku JavaScript postoje uslovni i ponavljajući tokovi. Uslovni tokovi se kreiraju pomoću ključnih reči `if`, `else if` i `else`, čija je osnovna struktura:

```
1 if (uslov1) {
2     // Ako je uslov1 ispunjen
3 }
4 else if (uslov2) {
5     // Ako uslov1 nije ispunjen i uslov2 jeste ispunjen
6 }
7 // ...
8 else {
9     // Ako nijedan od prethodnih uslova nije ispunjen
10 }
```

Uslovni tok se može postići i korišćenjem `switch` konstrukta, čija je forma:

```
1 switch (izraz) {
2     case vrednost1:
3         // Ako je izraz === vrednost1
4         break;
5     case vrednost2:
6         // Ako je izraz !== vrednost1 i izraz === vrednost2
7         break;
8     // ...
9     default:
10        // Ako izraz nema nijednu od zadatih vrednosti
11        break;
12 }
```

Što se tiče ponavljajućih tokova, odn. *petlji*, postoje tri vrste ovakvih tokova:

1. "Sve dok — radi"

```
1 while (uslov) {
2     // Telo se izvršava sve dok je uslov ispunjen
3 }
```

2. "Radi — sve dok"

```
1 do {
2     // Telo se izvršava barem jednom,
```

⁵Više o objektima u sekciji 2.4.

```
3     // a zatim onda sve dok je uslov ispunjen
4 } while (uslov);
```

3. "Tradicionalna for-petlja"

```
1 for (inicijalizator; uslov; korak) {
2     // Na pocetku se prvo izvrši inicijalizator.
3     // Zatim se ponavlja sledece:
4     // - Ako je uslov ispunjen:
5     //     - Izvrši telo
6     //     - Uradi korak
7 }
```

Ključnom reči `break` se prekida izvršavanje petlje, dok se ključnom reči `continue` kontrola toka postavlja iza kraja tela petlje (drugim rečima, preskače se ostatak tela petlje) i nastavlja se sa narednom iteracijom.

2.3 Funkcije

S obzirom da funkcije igraju izuzetno važnu ulogu u jeziku JavaScript, njima ćemo posvetiti posebnu pažnju. Funkcije se mogu definisati na više načina. Jedan od njih je dodeljivanje funkcije promenljivoj:

```
1 const imeFunkcije = function(arg1, arg2 /*, ..., argN */) {
2     // Telo funkcije
3 };
```

Funkcije su takođe podaci i one imaju svoj tip:

```
1 console.log(typeof imeFunkcije); // 'function'
```

U primeru iznad primećujemo dve stvari. Prva je to da funkcije možemo posmatrati i kao podatke koje dodeljujemo promenljivama kao vrednosti. Ovo je važna napomena koja će nas pratiti do kraja poglavlja. Drugo, primetimo da smo funkciju dodelili kao vrednost konstantnoj promenljivoj. Ovo je standardna praksa da bismo sprečili da taj identifikator dobije novu vrednost koja nije funkcija.

Funkcije se *izračunavaju* (kažemo i *pozivaju*) navođenjem njihovih imena i vrednostima za argumente. Na primer:

```
1 const stepen = function(osnova, eksponent) {
2     let rezultat = 1;
3     for (let broj = 0; broj < eksponent; broj++) {
4         rezultat *= osnova;
5     }
6     return rezultat;
7 };
8
9 stepen(2, 3); // Vratice rezultat 8
```

Ključna reč `return` određuje povratnu vrednost funkcije. Ukoliko ne stavimo vrednost nakon ove ključne reči, ili ukoliko uopšte nemamo `return` naredbu, smatra se da je povratna vrednost funkcije `undefined`.

Postoji jedna posebna funkcija, koja se može pozvati na sledeći način:

```
1 console.log(izraz);
```

Njena uloga je da na *tekstualni izlaz* ispiše vrednost *izraz*. U veb pregledaču, tekstualni izlaz je JavaScript konzola, dok je u **node** programu tekstualni izlaz terminal u kojem je program pokrenut.

Deklaraciona notacija za funkcije

Postoji kraća notacija za uvođenje funkcija, čiji je oblik:

```
1 function imeFunkcije(arg1, arg2 /*, ..., argN */) {  
2     // Telo funkcije  
3 }
```

Osim što je zapis kraći, ne zahteva se karakter `;` na kraju, kao kod dodeljivanja funkcije konstantnoj promenljivoj.

O rekurzivnim funkcijama

Sasvim očekivano, JavaScript programski jezik nudi mogućnost programerima da konstruišu i koriste rekurzivne funkcije. Naravno, treba voditi računa o veličini steka, i ne preterivati sa rekurzivnim pozivima. Primer rekurzivne funkcije koja računa n -ti Fibonačijev broj, a koju smo sada svi u stanju da napišemo, izgleda:

```
1 function fib(n) {  
2     if (n == 0 || n == 1) {  
3         return 1;  
4     }  
5     return fib(n-1) + fib(n-2);  
6 }
```

Često je bolje implementirati iterativnu varijantu algoritma (ukoliko je moguće) nego koristiti rekurziju.

Zadatak 2.1 Napisati iterativnu varijantu algoritma `fib(n)`. ■

2.3.1 Domet važenja promenljivih

Uvođenjem nove promenljive definišemo domet važenja te promenljive, odn. definišemo u kojim sve delovima programskog koda možemo koristiti tu promenljivu. U zavisnosti od načina uvođenja promenljivih, možemo razlikovati globalni domet, domet funkcije i blokovski domet. O svakom od ovih će biti reči u nastavku teksta.

JavaScript jezik ima veoma intrigantan sistem za kreiranje dometa važenja promenljivih, koji može dovesti do veoma čudnih ponašanja, makar iz ugla programera koje dolaze iz drugih programskih jezika. Za početak, bitno je napomenuti da se promenljive uvode u dometu funkcija, što znači da ako je promenljiva deklarisanu u okviru funkcije, ona nije vidljiva u dometu izvan te funkcije. Za uvođenja koja se dešavaju izvan svih blokova i funkcija, domet je ceo program — ovakve promenljive su *globalne*.

Argumenti funkcije su vidljivi samo u telu te funkcije. Takođe, promenljive koje uvodimo u telu funkcije su vidljive samo u telu te funkcije. Ovakve promenljive se nazivaju *lokalne*.

Što se tiče blokovskog dometa, u JavaScript jeziku (tj. do verzije ES6), blokovski domet ne postoji. Preciznije, samo promenljive koje su uvedene ključnim rečima **let** ili **const** su lokalne za blok u kojem su deklarisanu. Ukoliko imamo ugneždene blokove, ta dva bloka definišu svoje domete, pri čemu uži blok ima pristup širem bloku, ali ne i obrnuto. Za razliku od njih, promenljive uvedene ključnom reči **var** važe u celom dometu funkcije. Na primer:

```
1 const varTest = function(n) {
2     var x = 1;
3     if (true) {
4         // Ista promenljiva kao u sirem dometu!
5         var x = 2;
6         console.log(x); // 2
7     }
8     console.log(x);      // 2
9 };
10
11 const letTest = function(n) {
12     let x = 1;
13     if (true) {
14         // Uvodi se nova promenljiva u uzem dometu
15         let x = 2;
16         console.log(x); // 2
17     }
18     console.log(x);      // 1
19 };
```

Pre standarda ECMAScript 6 (kada su uvedene ključne reči `let` i `const`), programeri su bili prinuđeni da uvode promenljive ključnom reči `var`, što je moglo da dovede do situacije prikazane u prethodnom primeru. Zbog toga, podrazumevaćemo da smo sve promenljive uveli korišćenjem ključnih reči `let` ili `const` (osim u slučajevima kada je to eksplicitno naznačeno). Redeklaracija promenljive pomoću ključnih reči `let` i `const` u istom dometu se smatra greškom, i JavaScript okruženje će nam javiti da je došlo do greške tipa `SyntaxError`.

Ipak, treba razumeti ponašanje promenljivih uvedenih promenljivom `var`, o čemu ćemo pričati detaljnije u nastavku.

U narednom primeru, funkcija `f` ima pristup globalnoj promenljivoj `global`, kao i lokalno uvedenoj promenljivoj `local`. Izvan te funkcije, promenljiva `local` ne postoji:

```
1 var global = 1;
2 function f() {
3     var local = 2;
4     global++;
5     local++;
6     return global;
7 }
8
9 f();    // 2
10 f();   // 3
11
12 local; // ReferenceError: local is not defined
```

Napomenimo i da ukoliko ne deklariramo promenljivu korišćenjem ključne reči `var`, JavaScript okruženje će kreirati promenljivu u globalnom dometu, što možda nije nešto što je očekivano:

```
1 function f() {
2     local = 2;
3 }
4
5 local; // ReferenceError: local is not defined
6 f();   // Izvršava se naredba dodele u funkciji f
7 local; // 2
```

Smatra se dobrom praksom ukoliko minimizujemo uvođenje globalnih promenljivih — pre svega da ne bismo dolazili u koliziju sa drugim kodovima. Takođe, da bismo sprečili pojavljivanje prethodnog problema, dobro je uvek na početku funkcije staviti deklaracije svih promenljivih.

Zbog svih ovih napomena, ukoliko je moguće, treba uvoditi promenljive ključnom reči `const` ukoliko ne planiramo da promenljive redefinišemo. U slučaju da je redefinisavanje promenljive neophodno, onda treba koristiti ključnu reč `let` i izbegavati `var`.

2.3.2 Izdizanje deklaracija

Programski jezik JavaScript podržava koncept koji se naziva *izdizanje deklaracija* (engl. *hoisting*). Ova koncept podrazumeva da se deklaracije promenljivih i funkcija stavljaju u memoriju tokom faze kompiliranja. Drugim rečima, možemo u kodu, na primer, izvršavati pozive ka funkcijama pre njihove definicije. Na primer:

```
1 nazivZivotinje('pas');
2
3 function nazivZivotinje(ime) {
4     console.log(ime);
5 }
```

JavaScript izdiže samo deklaracije, ne i njihove definicije. U zavisnosti od toga kojom ključnom reči je promenljiva uvedena, razlikujemo dva ponašanja:

1. Ako je promenljiva uvedena ključnom reči `var` i ako pristupamo vrednosti ove promenljive pre nego što je ona definisana, njena vrednost će biti `undefined`.
2. Ako je promenljiva uvedena ključnom reči `let` ili `const` i ako pristupamo vrednosti ove promenljive pre nego što je ona definisana, JavaScript okruženje će javiti da je došlo do greške tipa `ReferenceError`.

Naredni primer ilustruje ovo ponašanje:

```
1 function izdizanjeDeklaracija() {
2     console.log(x); // undefined
3     console.log(y); // ReferenceError
4     var x = 1;
5     let y = 2;
6 }
```

Napomenimo da izdizanje deklaracija u kontekstu funkcija radi samo za deklaracionu notaciju, a ne i za dodeljivanje funkcije (konstantnoj) promenljivoj. U drugom slučaju, JavaScript okruženje javlja da je došlo do greške tipa `ReferenceError`, što je i očekivano u skladu sa prethodnom napomenom.

Posebno je bitno razumeti kombinacije ponašanja izdizanja deklaracije i prethodne diskusije o dometu važenja promenljivih. Na primer, neko ko razmatra naredni primer može doći do zaključka da će prvo biti ispisana vrednost 123, a zatim 1:

```
1 var a = 123;
2
3 function f() {
4     console.log(a);
5     var a = 1;
6     console.log(a);
7 }
8
9 f();
```


Ono što će zapravo biti ispisano jeste vrednost `undefined`, a zatim vrednost 1. Ovo ponašanje direktno sledi iz prethodnih diskusija: s obzirom da lokalni domet ima prioritet nad globalnim, u funkciji `f` prvo dolazi do izdizanja deklaracije lokalne promenljive `a`, zatim se ispisuje njena vrednost (koja je u tom trenutku jednaka `undefined`), zatim joj se dodeljuje vrednost 1, koja se zatim ispisuje.

2.3.3 Opcioni i podrazumevani argumenti

JavaScript je poprilično slobodan u načinu rukovanja argumentima funkcija. Pogledajmo naredne primere:

```
1 function ispisi1(x) {
2   console.log(x);
3 }
4
5 function ispisi2(a, b) {
6   console.log(a);
7   console.log(b);
8 }
9
10 ispisi1(1, 2, 'Niska'); // Ispisace samo prvi argument (1),
11                        // drugi se odbacuju
12 ispisi2(7); // Ispisace 7 za prvi argument,
13            // i undefined za drugi argument
```

Pozivom `ispisi1(1, 2, 'Niska')` neće nastati nikakav problem — kako funkcija ima samo jedan argument, a mi smo joj prosledili tri, JavaScript će dodeliti argumentu `x` prvu prosledenu vrednost 1, a ostale dve prosledene vrednosti će odbaciti.

Slično, ni poziv `ispisi2(7)` neće napraviti problem — kako funkcija ima dva argumenta, a mi smo joj prosledili samo jedan, JavaScript će dodeliti prvom argumentu `a` prvu prosledenu vrednost 7, a drugi argument `b` će dobiti vrednost `undefined`.

Dobra strana ovog ponašanja jeste da funkcija može raditi različite stvari u zavisnosti od toga sa kojim brojem argumenata je pozvana. Veoma loša strana jeste da možemo da pogrešimo broj argumenata funkcije prilikom njenog pozivanja, a da nismo ni svesni da smo napravili grešku.

Ono što možemo uraditi jeste da dodelimo *podrazumevane vrednosti* argumentima funkcije, čime oni neće dobiti `undefined` vrednosti. Na primer:

```
1 const stepen = function(osnova, eksponent = 2) {
2   let rezultat = 1;
3   for (let broj = 0; broj < eksponent; broj++) {
4     rezultat *= osnova;
5   }
6   return rezultat;
7 };
8
9 stepen(2, 3); // Vratice rezultat 8
10 stepen(10); // Vratice rezultat 100
```

Zadatak 2.2 Napisati funkciju `pozoviMe(ime)` koja na tekstualni izlaz ispisuje naredne poruke u zavisnosti od toga kako je pozvana:

```
1 pozoviMe(); // 'Zdravo, osobo bez imena!'
2 pozoviMe('Ana'); // 'Zdravo, Ana!'
```

Ne koristiti uslovne kontrole toka.

2.3.4 Anonimne funkcije

Ukoliko nije potrebno da imenujemo funkciju, već samo da se izvrši jedanput, možemo koristiti anonimne funkcije. Zapravo, već smo videli anonimne funkcije, a da to nismo ni znali! Kada definišemo funkciju putem dodeljivanja konstantnoj funkciji, izraz sa desne strane jednakosti čini jednu anonimnu funkciju. Na primer, u fragmentu koda

```
1 const kvadrat = function(x) {  
2   return x*x;  
3 };
```

vrednost

```
1 function(x) {  
2   return x*x;  
3 }
```

predstavlja anonimnu funkciju.

Ovu vrednost, međutim, ne možemo da koristimo samu po sebi, već ju je moguće, na primer, dodeliti promenljivoj, proslediti kao argument funkcije, vratiti kao povratnu vrednost funkcije ili upotrebiti na neki drugi način (JavaScript ima moćne alate za rad sa funkcijama, o čemu će biti reči u kasnijem delu teksta).

Dodatno, anonimne funkcije imaju i skraćenu notaciju, koja se naziva *lambda funkcija*. Prethodni primer zapisan kao lambda funkcija izgleda:

```
1 (x) => {  
2   return x*x;  
3 }
```

Dakle, osnovna struktura lambda funkcije je *lista parametara => telo funkcije*. Kao što vidimo iz primera, lista parametara se navodi između zagrada (i). Dodatno, ukoliko funkcija ima jedan parametar, zagrade nisu neophodne, a ukoliko funkcija nema parametre, onda lista parametara izgleda (). Na primer:

```
1 x => {  
2   return x*x;  
3 }  
  
1 () => {  
2   console.log('Funkcija nema parametre!');  
3 }
```

Operator *strelica* označen kao => odvaja listu parametara od tela funkcije. Telo funkcije ne mora da sadrži vitičaste zagrade { i } ukoliko ima jednu naredbu (koja upravo predstavlja i povratnu vrednost funkcije, te u tom slučaju nije potrebno navesti ni `return` ključnu reč). Na primer:

```
1 x => x * x
```

2.3.5 Samoizvršavajuće funkcije

Anonimne funkcije imaju jednu interesantnu primenu u kreiranju tzv. *samoizvršavajućih funkcija* (engl. *immediate function*). U pitanju su funkcije koje se pozivaju odmah po njihovom definisanju. Na primer:

```
1 (
2     function () {
3         alert('Pozdrav!');
4     }
5 )();
```

odnosno, ako imaju argumente:

```
1 (
2     function (name) {
3         alert('Zdravo, ' + name + '!');
4     }
5 )('svete');
```

Iako sintaksa na prvi pogled može da izgleda kompleksno, zapravo nije — potrebno je staviti obične zagrade oko anonimne funkcije, a zatim dodati još jedan par zagrada nakon toga (što zapravo predstavlja poziv te funkcije). Alternativna sintaksa podrazumeva da se prva otvorena zagrada zatvara nakon poziva funkcije. Prvobitno uvedena sintaksa i alternativna sintaksa (koju ćemo mi koristiti u nastavku), date su narednim primerom:

```
1 (function () {
2     // ...
3 })();
4
5 // ili
6
7 (function () {
8     // ...
9 })();
```

Može delovati čudno zašto bismo koristili ovako definisane (i pozvane) funkcije. Jedna česta primena jeste ukoliko želimo da izvršimo nekakve akcije bez kreiranja globalnih promenljivih. Naravno, treba uzeti u obzir da ovakve funkcije možemo izvršavati samo jedanput. Iz ove dve napomene možemo zaključiti da su samoizvršavajuće funkcije korisne za poslove inicijalizacije.

Samoizvršavajuće funkcije mogu vratiti i vrednosti, i u tom slučaju nam spoljašnji par zagrada nije neophodan, što je ilustrovano na slici 2.1. Međutim, ovo može biti konfuzno za čitanje jer ne znamo da li definišemo funkciju ili samoizvršavajuću funkciju sve do kraja definicije, kao na slici 2.2. Zbog toga treba uvek stavljati zagrade oko samoizvršavajućih funkcija.

<pre>1 var f = (function () { 2 // something complex with 3 // temporary local variables 4 5 // return something; 6 })();</pre>	<pre>1 var f = function () { 2 // something complex with 3 // temporary local variables 4 5 // return something; 6 }();</pre>
---	---

Slika 2.1: Primer dva fragmenta koda u kojem je definisana samoizvršavajuća funkcija sa spoljnim zagradama (levo) i bez spoljnih zagrada (desno).

2.3.6 Funkcije kao podaci — nastavak

Kao što smo rekli, funkcije možemo posmatrati kao i sve druge objekte, tj. kao podatke. Ništa nas ne sprečava da definišemo funkciju unutar neke druge funkcije:

<pre> 1 var f = function () { 2 // something complex with 3 // temporary local variables 4 5 // return something; 6 }; </pre>	<pre> 1 var f = function () { 2 // something complex with 3 // temporary local variables 4 5 // return something; 6 }(); </pre>
---	---

Slika 2.2: Primer dva fragmenta koda u kojem nije očigledno da li je promenljiva `f` funkcija (levo) ili povratna vrednost samoizvršavajuće funkcije (desno), sve dok ne pročitamo poslednju liniju koda.

```

1 function outer(param) {
2     function inner(theinput) {
3         return theinput * 2;
4     }
5     return 'The result is ' + inner(param);
6 }
7
8 outer(2);    // 'The result is 4'
9 inner(2);    // ReferenceError: inner is not defined

```

Kao što vidimo, poziv funkcije `outer` će izvršiti poziv funkcije `inner`. Dodatno, vidimo da funkcija `inner` nije dostupna u globalnom dometu. Zbog toga se funkcije poput `inner` često nazivaju *unutrašnje* (engl. *inner*) ili *privatne* (engl. *private*) funkcije.

Korišćenje privatnih funkcija ima nekoliko prednosti. Za početak, smanjuje se zagađivanje globalnog dometa dodatnim imenima. Dodatno, moguće je izložiti "svetu" samo one funkcije koje mi odaberemo da budu dostupne, dok neke druge ostaju sakrivene.

Takođe, funkcije je moguće vraćati kao vrednosti, što naredni primer ilustruje:

```

1 function a() {
2     console.log('A!');
3
4     return function () {
5         console.log('B!');
6     };
7 }
8
9 var newFunc = a(); // A!
10 newFunc();        // B!
11
12 // Ukoliko ne zelimo da deklarismo novu promenljivu
13 // da bismo pozvali funkciju,
14 // mozemo jednostavno dodati par zagrada
15 a()();             // B!

```

S obzirom da funkcije mogu da vraćaju druge funkcije, da li je moguće da funkcija kao povratna vrednost pregazi spoljašnju funkciju? Odgovor je da, što naredni primer ilustruje:

```

1 function a() {
2     console.log('A!');
3
4     return function () {
5         console.log('B!');
6     };
7 }
8

```

```

9  a = a();      // A!
10 a();          // B!
11 a();          // B!

```

Očigledno, ova strategija nam može služiti za pisanje funkcija koji prvo izvršavaju neki posao inicijalizacije, a zatim postanu funkcije koje rade neku akciju. Štaviše, možemo ići korak dalje i definisati funkciju koja će sama sebe da redefiniše:

```

1  function a() {
2      console.log('A!');
3
4      a = function () {
5          console.log('B!');
6      };
7  }

```

Prvi put kada je funkcija `a` pozvana, ispisaće se `'A!'` (ovo možemo smatrati kao posao inicijalizacije), a svaki naredni poziv će ispisati `'B!'`, zato što se redefiniše globalna promenljiva `a` tako što joj se dodeljuje nova funkcija.

Još jedan korak dalje ilustruje kombinaciju do sada prikazanih tehnika:

```

1  var a = (function () {
2      function someSetup() {
3          var setup = 'done';
4      }
5
6      function actualWork() {
7          console.log('Worky-worky');
8      }
9
10     someSetup();
11     return actualWork;
12 })();

```

Zadatak 2.3 Posmatrajući prethodni primer, dati odgovore na naredna pitanja:

1. Šta će biti ispisano u konzoli kada se završi naredba dodele promenljivoj `a`?
2. Šta će biti ispisano u konzoli kada se izvrši poziv `a()`; nakon izvršavanja prethodno pomenute naredbe dodele?

2.3.7 Zatvorenje

Mogućnost da se funkcije tretiraju kao vrednosti, kombinovano sa činjenicom da se lokalno-vedene promenljive rekreiraju svaki put kada je funkcija pozvana, dovodi do interesantnog pitanja. Šta se dešava sa životnim vekom lokalno-vedenih promenljiva kada poziv funkcije koji ih je kreirao više nije aktivan? Naredni kod demonstrira sledeći primer: definišemo funkciju `obuhvatiVrednost` koja kreira lokalno-vedenu promenljivu. Ona zatim vraća funkciju koja dohvata i vraća tu lokalno-vedenu promenljivu.

```

1  function obuhvatiVrednost(n) {
2      let lokalnaP = n;
3      return () => lokalnaP; // Povratna vrednost je funkcija!
4  }
5
6  let obuhvati1 = obuhvatiVrednost(1); // lokalnaP je bila postavljena na 1

```

```

7 let obuhvati2 = obuhvatiVrednost(2); // lokalnaP je bila postavljena na 2
8
9 console.log(obuhvati1()); // 1
10 console.log(obuhvati2()); // 2

```

Ovakvo ponašanje je dozvoljeno i radi onako kako i očekujemo — obema instancama lokalno-uvodenih promenljivih i dalje možemo da pristupimo. Ovakva situacija je dobar prikaz činjenice da se lokalno-uvodne promenljive kreiraju iznova za svaki poziv, i drugi pozivi ne mogu da utiču međusobno na vrednosti ovakvih promenljivih.

Ova sposobnost jezika — da je moguće referisati specifičnu instancu lokalno-uvodne promenljive u obuhvatajućem dometu — naziva se *zatvorenje* (engl. *closure*). Sama funkcija koja referiše na lokalno-uvodne promenljive iz lokalnih dometa oko nje naziva se *zatvorenje*. Ovo ponašanje ne samo da nam olakšava razumevanje životnog veka lokalnih promenljivih, ali takođe predstavlja i alat za korišćenje funkcija kao vrednosti na zanimljive načine.

Jedno od klasičnih pitanja vezanih za JavaScript na razgovorima za posao jeste da se napiše zatvorenje koje sabira broj sa nekim proizvoljnim brojem. Mala izmena u prethodnom primeru daje nam rešenje:

```

1 function saberi(sabirak) {
2     return broj => broj + sabirak;
3 }
4
5 let saberiSa2 = saberi(2);
6 console.log(saberiSa2(10)); // 12

```

Objasnimo još malo kako zatvorenja funkcionišu. Možemo o funkcijama kao vrednostima razmišljati na sledeći način — one sadrže kod u svom telu, ali i okruženje u kojem su kreirane. Kada se pozovu, telo funkcije vidi okruženje u kojem je kreirano, a ne okruženje u kojem je pozvano. U našem primeru, `saberi` je pozvana i kreirala je okruženje u kojem je argument `sabirak` vezan vrednošću 2. Funkciju kao vrednost koju ona vraća, a koja je sačuvana u `saberiSa2`, pamti ovo okruženje. Zato onda kada je pozvana, sabraće argument sa vrednošću 2.

Zadatak 2.4 Napisati zatvorenja `mnozi1` i `mnozi2` koja množe tri broja. Zatvorenja napisati tako da su sledeći pozivi ispravni: `mnozi1(2)(3, 4)` i `mnozi2(2)(3)(4)`. ■

Zadatak 2.5 Razmisлити o narednom fragmentu koda koji za ideju ima da popuni niz `arr` funkcijama koje vraćaju odgovarajuću vrednost `i` u zavisnosti od toga koja je funkcija pozvana korišćenjem tehnike zatvorenja:

```

1 function F() {
2     var arr = [], i;
3
4     for (i = 0; i < 3; i++) {
5         arr[i] = function () {
6             return i;
7         };
8     }
9
10    return arr;
11 }
12

```

```
13 var arr = F();
```

Odgovoriti na naredna pitanja:

1. Šta očekujemo da naredni fragment koda ispiše?

```
1 arr[0]();  
2 arr[1]();  
3 arr[2]();
```

2. Izvršiti date fragmente koda u nekom JavaScript okruženju i proveriti da li se očekivanje poklopilo sa dobijenim vrednostima. Dati obrazloženje za dobijeno ponašanje.



2.4 Strukture podataka: Nizovi i objekti

2.4.1 Nizovi

Nizovi predstavljaju jednostavnu sekvencionalnu strukturu podataka. U JavaScript jeziku, literali nizova se zapisuju između zagrada [i], a vrednosti koje se skladište u nizu se razdvajaju zapetom. Na primer:

```
1 let nizBrojeva = [1, 2, 3, 4, 5];  
2 let nizNiski = ['Niska1', 'Niska2'];
```

Tip nizova je:

```
1 console.log(typeof nizBrojeva); // 'object'
```

Vrednostima u nizu se može pristupiti pomoću *indeksnog operatora*, koji takođe koristi zagrade [i], pri čemu se indeks elementa navodi između zagrada. Nizovi su indeksirani počevši od 0:

```
1 console.log(nizBrojeva[0]); // 1  
2 console.log(nizNiski[1]); // Niska2
```

Ono što nije očigledno iz prethodnih primera jeste da nizovi mogu da čuvaju vrednosti različitog tipa. Na primer:

```
1 let niz = [1, 2, 'Niska', { broj: 2 }];  
2  
3 console.log(niz[0]); // 1  
4 console.log(niz[1]); // 2  
5 console.log(niz[2]); // Niska  
6 console.log(niz[3]); // { broj: 2 }
```

2.4.2 Objekti

U prethodnom primeru vidimo nešto što do sada nismo videli — četvrti element (odno, treći element ako posmatramo sa stanovišta računara) strukture `niz` sadrži nešto što nazivamo literal *objekta* (engl. *object*).

Tip objekata je:

```
1 console.log(typeof { broj: 2 }); // 'object'
```

Objekti su još jedna struktura podataka i možemo ih posmatrati kao kolekcije nečega što nazivamo *svojstva* (engl. *properties*). Svojstva nisu ništa drugo nego vrednosti koje se

čuvaju u okviru drugih objekata. Tako, iz prethodnog primera, objekat { broj: 2 } ima samo jedno svojstvo koje se naziva broj i čija je vrednost 2.

Literali objekata se konstruišu navođenjem vitičastih zagrada { i }, unutar kojih se navode parovi imeSvojstva : vrednostSvojstva, odvojenih zapetom između tih zagrada. Na primer, recimo da želimo da napravimo objekat koji predstavlja log kada se neki korisnik ulogovao na sistem, kao i koji put je posetio sistem:

```
1 let log1 = {
2   datumVreme: '31.10.2018. 00:00',
3   korisnik: 'admin',
4   'broj posete': 42
5 };
6
7 console.log(log1.datumVreme);      // 31.10.2018. 00:00
8 console.log(log1.korisnik);        // admin
9 console.log(log1['broj posete']);   // 42
10 console.log(log1.sifra);           // undefined
```

U ovom primeru vidimo da možemo predstaviti objekte u više linija, što povećava čitljivost koda. Takođe, vidimo da se imena svojstva koja nisu ispravni identifikatori promenljivih (kao što su brojevi ili niske sa razmakom) pišu kao niske. Konačno, vidimo da se pristupanje svojstvu koje ne postoji u objektu izračunava na vrednost **undefined**.

2.4.3 Svojstva u objektima

Skoro sve JavaScript vrednosti imaju neka svojstva. Izuzetak od ovog pravila su **null** i **undefined**, i ukoliko pokušamo da pristupimo nekom svojstvu ovih vrednosti, JavaScript okruženje će se pobuniti i javiti nam da je došlo do greške tipa **TypeError**:

```
1 // TypeError: Cannot read property 'svojstvo' of null
2 null.svojstvo;
```

U prethodnom primeru smo videli jedan način za pristup svojstvima neke vrednosti — *tačka-notacijom*. Ona podrazumeva da navedemo vrednost, iza koje sledi karakter ., a zatim da navedemo ime svojstva. Postoji još jedan način za dohvatanje svojstava, a to je pomoću uglastih zagrada. Tako, na primer, korišćenjem vrednost.svojstvo i vrednost[svojstvo] možemo dohvatiti svojstvo objekta vrednost — međutim, ovo ne mora biti nužno isto svojstvo!

Razlika je u tome što korišćenjem tačka-notacije reč nakon karaktera tačke predstavlja naziv tog svojstva. Sa druge strane, izraz između zagrada se izračunava da bi se dobio ime svojstva. Dakle, ako iskoristimo vrednost.svojstvo, time kažemo programu da potraži svojstvo u objektu vrednost koje se zove "svojstvo", dok ako iskoristimo vrednost[svojstvo], JavaScript će svojstvo smatrati kao izraz, pa će vrednost izraza svojstvo biti izračunata, i to što se dobije će biti implicitno konvertovano u nisku, da bi se potražilo svojstvo sa tim imenom.

Tako da, ako znamo da se svojstvo neke vrednosti boja zove ljubicasta, možemo mu pristupiti sa boja.ljubicasta. Ako želimo da iskoristimo vrednost sačuvanu u promenljivoj indeks, možemo koristiti boja[indeks], itd. Imena svojstava su niske, a kao što znamo, niske mogu sadržati najrazličitija imena, na primer, mogu početi brojem ili sadržati razmak. Međutim, tačka-notacija ne može biti upotrebljena u ovakvim slučajevima, te moramo koristiti uglaste zagrade, na primer, vrednost[2] ili vrednost['Ime i prezime'].

Elementi u nizu se čuvaju kao svojstva tog niza, te nam je sada jasno zašto mogu sadržati vrednosti različitih tipova. Međutim, oni koriste isključivo cele brojeve kao imena svojstava, te zbog toga za pristupanje elementima nizova uvek koristimo uglaste zagrade, kao što smo i pokazali.

Nad nizovima postoji definisano svojstvo `length` koje nam izračunava broj elemenata niza. Ono se najčešće koristi u tačka-notaciji jer unapred znamo ime tog svojstva. Naredni primer koristi ovo svojstvo da prođe kroz sve elemente niza `nekiNiz` i ispise njegove vrednosti na tekstualni izlaz:

```
1 let nekiNiz = ['Ovaj', 'niz', 'ima', 5, 'elementa'];
2 for(let i = 0; i < nekiNiz.length; i++) {
3   console.log(nekiNiz[i]);
4 }
```

Isto svojstvo sadrže i niske, na primer:

```
1 let nekaNiska = 'Ja sam neka niska!'
2 for(let i = 0; i < nekaNiska.length; i++) {
3   console.log(nekaNiska[i]);
4 }
```

Dodeljivanje novih svojstava

Možemo dodeliti vrednost svojstvu pomoću operatora `=`. Ovim ćemo pregaziti vrednost svojstva ukoliko već postoji ili će kreirati novo svojstvo ukoliko ne postoji. Na primer, možemo krenuti od narednog objekta `stablo` i dodati mu dva nova svojstva:

```
1 let stablo = { cvor: 2 };
2 stablo.leviSin = { cvor: 1 };
3 stablo.desniSin = { cvor: 3 };
```

Nakon ovoga, vrednost promenljive `stablo` se može zapisati na sledeći način:

```
1 {
2   cvor: 2,
3   leviSin: {
4     cvor: 1
5   },
6   desniSin: {
7     cvor: 3
8   }
9 }
```

Uklanjanje postojećih svojstava

Ako želimo da uklonimo svojstvo iz objekta, možemo koristiti unarni operator `delete`. Na primer, ako želimo da obrišemo svojstvo `leviSin` iz prethodnog primera, to možemo uraditi na sledeći način:

```
1 delete stablo.leviSin;
```

Ako pokušamo da pristupimo obrisanom svojstvu, biće nam vraćena vrednost `undefined`.

Ispitivanje postojanja svojstava

Ako želimo da proverimo da li objekat sadrži neko svojstvo, možemo koristiti binarni operator `in`, koji je infiksnog tipa i primenjuje se na nisku i objekat:

```
1 console.log('leviSin' in stablo); // false
2 console.log('desniSin' in stablo); // true
```

Postoji razlika između brisanja svojstva i postavljanja njegove vrednosti na `undefined`. U prvom slučaju, operator `in` će vratiti vrednost `false`, dok u drugom slučaju, objekat i dalje ima to svojstvo i operator `in` će vratiti `true`.

Ukoliko želimo da dobijemo niz sa imenima svih svojstava nekog objekta, možemo koristiti funkciju `Object.keys` čiji je argument objekat čiji niz imena svojstava želimo da dobijemo:

```
1 console.log(Object.keys({x: 0, y: 0, z: 2})); // ['x', 'y', 'z']
```

Skraćeno vezivanje vrednosti svojstva

Posmatrajmo naredni primer:

```
1 let boja = 'zuta';
2 let limun = { tip: 'voce', boja };
```

Naizgled, objekat `limun` ima nedefinisano svojstvo `'boja'` jer mu nismo dodelili nikakvu vrednost. Međutim, ako bismo ispisali taj objekat

```
1 console.log(limun); // { tip: 'voce', boja: 'zuta' }
```

vidimo da mu je dodeljena vrednost na osnovu istoimene promenljive `boja` u istom dometu. Ovo je svojstvo jezika JavaScript, čime možemo izbeći pisanja oblika `{ boja: boja }`.

Dinamičko određivanje naziva svojstva

U ES6 standardu moguće je, pored kreiranja statičkih imena za svojstava, objektima dodeljivati imena koja su dinamička:

```
1 let vehicle = "car";
2
3 function vehicleType(){
4     return "truck";
5 }
6
7 let car = {
8     [vehicle+"_model"]: "Ford";
9 };
10
11 let truck = {
12     [vehicleType() + "_model"]: "Mercedes";
13 };
14
15 console.log(car); // {"car_model":"Ford"}
16 console.log(truck); // {"truck_model":"Mercedes"}
```

2.4.4 Metodi

Svojstva objekata ne moraju biti brojevi, niske i sl. već mogu biti i funkcije. Svojstva čije su vrednosti funkcije se u literaturi najčešće nazivaju *metodi* (engl. *method*), da bi se razlikovala od svojstava čije vrednosti nisu funkcije. Tako, na primer, vrednost svojstva `toUpperCase` koje sadrže sve niske predstavlja funkciju koja izračunava kopiju niske nad kojom je pozvana u kojoj su svi karakteri pretvoreni u velika slova. Zbog toga što je vrednost ovog svojstva funkcija, `toUpperCase` predstavlja primer metoda. Radi kompletnosti, navedimo u funkciju `toLowerCase` koja radi suprotno — vraća kopiju niske nad kojom je pozvana u kojoj su transformisana velika slova u mala slova. Ovo možemo testirati narednim primerom:

```
1 let niska = 'nIsKa';
2 console.log(typeof niska.toUpperCase); // 'function'
```

```
3
4 // Naredni uslov je ispunjen
5 if (typeof niska.toUpperCase === 'function') {
6     console.log(niska.toUpperCase()); // NISKA
7 }
```

Nad nizovima su definisana dva korisna metoda — `push(elem)`, koji dodaje vrednost `elem` na kraj niza nad kojim je pozvan i `pop()`, koji uklanja poslednji element iz niza i vraća ga kao povratnu vrednost. Na primer:

```
1 let sekvenca = [1, 2, 3];
2
3 sekvenca.push(4);
4 sekvenca.push(5);
5
6 console.log(sekvenca); // [1, 2, 3, 4, 5]
7 console.log(sekvenca.pop()); // 5
8 console.log(sekvenca); // [1, 2, 3, 4]
```

2.4.5 Deskriptorski objekat

Podsetimo se osobine proširivosti JavaScript objekata:

```
1 let point = {
2     x: 10,
3     y: 5,
4 };
5
6 point.color = 'orange';
7
8 console.log(point);
```

U primeru je, nakon što je kreiran objekat `point` koji predstavlja tačku u ravni, dodato svojstvo `color` koje je ravnopravno sa preostalim svojstvima objekta.

Funkciju koju do sada nismo pominjali, a koja dozvoljava proširivanje objekata uz dodatna podešavanja je `Object.defineProperty()`.

```
1 Object.defineProperty(point, 'name', {
2     enumerable: true,
3     writable: true,
4     configurable: true,
5     value: 'A'
6 });
7
8 console.log(point);
```

Prethodnim kodom smo dodali objektu `point` svojstvo `name` sa vrednošću `'A'`. Prvim parametrom funkcije `Object.defineProperty` navodi se objekat koji se proširuje, drugim parametrom se navodi ime svojstva koje se dodaje, a trećim parametrom, takozvanim *deskriptorskim objektom* (engl. *property descriptor*), osobine koje novo svojstvo treba da ima.

Svojstvo `value` deskriptor objekta predstavlja vrednost novog svojstva i u našem slučaju to je niska `'A'`. Svojstvom `writable` se kontroliše da li se vrednost može menjati operatorom dodele. Na primer, narednim kodom

```
1 let user = {
2     name: 'Paul'
```

```
3 };
4
5 Object.defineProperty(user, 'status', {
6     value: 'active',
7     write: false
8 });
```

definiše se svojstvo `status` objekta `user` koje se ne može menjati. Tako bi, na primer, naredba `user.status = 'inactive'`; bila bez efekta ili bi, ako se kod izvršava u striktnom modu⁶, rezultirala greškom.

Slično, `enumerable` svojstvom deskriptor objekta se određuje da li svojstvo učestvuje u iteriranju kroz objekat `for-in` petljom, dok `configurable` svojstvo određuje da li se objekat na ovaj način u daljem radu može konfigurisati ili ne. Na primer, naredni kod

```
1 Object.defineProperty(user, 'password', {
2     value: '123456',
3     enumerable: false
4 });
5
6 for (prop in user){
7     console.log(prop);
8 }
```

ispisaće svojstva `name` i `status`, ali ne i svojstvo `password`.

Funkcija `Object.defineProperty` se može koristiti i za ažuriranje objekata i promenu njegovih postojećih svojstva. Ukoliko se navede ime svojstva koje već postoji, umesto dodavanja svojstava vrši se njegovo ažuriranje. Zbog toga će nam deskriptorski konfiguracioni objekat biti važan i u priči o dekoratorima⁷.

2.4.6 Referenciranje objekata

Kada razmišljamo o dvema vrednostima nekog osnovnog tipa, na primer, `120` i `120`, smatramo ih istim brojevima, iako njihova fizička reprezentacija u memoriji možda nije ista⁸. Kada su objekti u pitanju, postoji razlika između:

- postojanja dve reference nad istim objektom
- postojanje dva objekta koji imaju ista svojstva

Posmatrajmo naredni primer:

```
1 let objekat1 = {vrednost: 10};
2 let objekat2 = objekat1;
3 let objekat3 = {vrednost: 10};
4
5 console.log(objekat1 === objekat2); // true
6 console.log(objekat1 === objekat3); // false
7
8 objekat1.vrednost = 15;
9
10 console.log(objekat2.vrednost); // 15
11 console.log(objekat3.vrednost); // 10
```

⁶Više o striktnom režimu u sekciji 2.6.1.

⁷Ovu temu ćemo obraditi u poglavlju posvećenu jeziku TypeScript. Videti sekciju 3.5.

⁸Ovde, pod *fizičkom reprezentacijom u memoriji* smatramo da je jedan broj upisan na adresi, na primer, 0000019190CDC420, a drugi broj na različitoj adresi, na primer, 0000019190CDC880.

Promenljive `objekat1` i `objekat2` vezuju isti objekat, odnosno, možemo reći da *referišu* na isti objekat. Zbog toga, promena svojstva u `objekat1` podrazumeva i promenu vrednosti u `objekat2`. Kažemo da ovi objekti imaju isti *identitet* (engl. *identity*). Promenljiva `objekat3` referiše na drugi objekat, koji "slučajno" sadrži ista svojstva kao i `objekat1`, ali ima svoj životni vek.

Već smo napomenuli u podsekciji "Uvođenje promenljivih" da uvođenje konstantnih objekata ne znači da nije moguće promeniti svojstva tog objekta, već samo da toj promenljivoj nije moguće dodeliti novu vrednost. U ovom trenutku možemo dati i fragment koda koji ilustruje ovo ponašanje, na primer:

```
1 const konstantanObj = {vrednost: 1};
2
3 konstantanObj.vrednost = 0; // Korektno
4 console.log(konstantanObj); // {vrednost: 0}
5
6 // TypeError: Assignment to constant variable.
7 konstantanObj = {vrednost: 10};
```

Poređenje objekata operatorima `==` i `===` se vrši tako što se porede odgovarajući identiteti. Rezultat izračunavanja je `true` ako objekti imaju isti identitet, a inače je `false`, čak i ako imaju identična svojstva. Dakle, ne postoji ugrađen *operator dubokog poređenja*, ali je moguće napisati sopstven.

Zadatak 2.6 Napisati funkciju `dubokoPoredjenje(obj1, obj2)` koja vraća `true` ako su argumenti iste vrednosti ili ako su objekti sa istim svojstvima. ■

2.4.7 Još malo o nizovima

U ovom delu ćemo dati opise još nekih funkcionalnosti vezanih za nizove u jeziku JavaScript.

Ispitivanje sadržaja

Nizovi imaju ugrađen metod `includes` koji proverava da li se neka vrednost nalazi kao element tog niza. Na primer:

```
1 let niz = [1, 2, 'tri'];
2
3 console.log(niz.includes(1));           // true
4 console.log(niz.includes('cetiri'));    // false
```

For-of petlja

Jedna standardna stvar koja se radi sa nizovima jeste prolazak kroz svaki element niza, dohvatanje vrednosti na tekućem indeksu i zatim korišćenje te vrednosti za dalje izračunavanje. Na primer:

```
1 let niz = [1, 2, 'tri'];
2
3 for (let i = 0; i < niz.length; ++i) {
4   let element = niz[i];
5
6   console.log(element);
7 }
```

Postoji kraća varijanta za istu svrhu, koja se često naziva *for-of* petlja. U programskom jeziku JavaScript njena forma je:

```
1 for (let idElementa of idNiza) {
2   // Radi nesto sa promenljivom idElementa
3 }
```

Konkretno, prethodni primer se može refaktorisati u *for-of* petlju na sledeći način:

```
1 let niz = [1, 2, 'tri'];
2
3 for (let element of niz) {
4   console.log(element);
5 }
```

For-in petlja

Komplementarno *for-of* petlji, postoji i *for-in* petlja kojom se iterira kroz indekse nizova:

```
1 for (let ind in niz) {
2   const element = niz[ind];
3   console.log(ind, ':', element);
4 }
```

Za razliku od *for-of* petlje, ova petlja se može koristiti za iteriranje kroz ključeve objekata:

```
1 let obj = { x: 1, y: 2, z: 3 };
2 for (let key in obj) {
3   const val = obj[key];
4   console.log(key, ':', val);
5 }
```

Metod forEach

Prolazak kroz petlju se takođe može izvršiti i metodom `forEach`, čija je deklaracija zadata sa:

```
1 arr.forEach(function callback(currentValue[, index[, arrayRef]]) {
2   //your iterator
3 });
```

Dakle, ova funkcija zahteva *funkciju povratnog poziva* (engl. *callback function*) koja će biti izračunava za svaki element iz niza `arr`, redom. Za ovu funkciju povratnog poziva važi da će se za njen prvi argument vezati tekući element niza u iteraciji. Dodatno, možemo specifikovati drugi argument za vezivanje vrednosti indeksa na kojoj se tekući element nalazi, kao i treći argument za vezivanje reference na niz nad kojim se poziva metod `forEach`. Demonstriranje korišćenje ovog metoda na osnovu prethodnih primera, dato je u nastavku:

```
1 let niz = [1, 2, 'tri'];
2
3 niz.forEach(element => {
4   console.log(element);
5 });
6
7 niz.forEach((element, indeks) => {
8   console.log(`Element ${element} se nalazi na poziciji ${indeks}`);
9 });
10
11 niz.forEach((element, indeks, nizRef) => {
12   console.log(`Element ${element} se nalazi na poziciji ${indeks} u nizu [${
13     nizRef}]`);
14 });
```

Pretraga vrednosti

Metod `includes` služi da izračuna da li se neki element nalazi u nizu, ali ne i na kojoj poziciji. Za to nam služi metod `indexOf`. Metod kreće od početka niza i prolazi do kraja tražeći element u nizu. Ako ga pronađe, vratiće indeks na kojem se element nalazi. U suprotnom, vratiće `-1`. Ako želimo da tražimo od kraja niza umesto od početka, možemo koristiti metod `lastIndexOf`. Na primer:

```
1 console.log([1, 2, 3, 2, 1].indexOf(2)); // 1
2 console.log([1, 2, 3, 2, 1].lastIndexOf(2)); // 3
```

Oba metoda imaju opcioni drugi argument kojim se specifikuje od kog indeksa da se započne pretraga.

Biranje podniza

Metod `slice`, koji uzima dva indeksa, vraća podniz niza nad kojim se primenjuje, a koji je definisan upravo tim indeksima, pri čemu je prvi indeks inkluzivan, a drugi nije. Ukoliko ne specifikujemo drugi indeks, onda će se za kraj podniza uzeti kraj niza. Na primer:

```
1 console.log([0, 1, 2, 3, 4].slice(2, 4)); // [2, 3]
2 console.log([0, 1, 2, 3, 4].slice(2)); // [2, 3, 4]
```

Nadovezivanje nizova

Metod `concat`, koji prima niz kao argument, nadovezaće taj argument na kraj niza nad kojim se poziva, i taj nadovezani niz vratiti kao povratnu vrednost (originalni niz se ne menja). Takođe, ako kao prosledimo bilo šta što nije niz, ta vrednost će biti dodata na kraj, kao da je prosleđen niz sa jednim elementom. Na primer:

```
1 let a = [1, 2, 3];
2
3 console.log(a.concat([4, 5])); // [1, 2, 3, 4, 5]
4 console.log(a.concat(6)); // [1, 2, 3, 6]
```

2.4.8 Još malo o niskama

Slično kao i nizovi, i niske imaju relativno veliki broj ugrađenih metoda. Na primer, kao i nizovi, i niske imaju metode `slice` i `indexOf`, koje vraćaju podnisku, odnosno, indeks tražene niske, redom. Jedna razlika između metoda `indexOf` kod nizova i istoimenog metoda kod niske jeste ta da se kod nizova traži jedan element, a kod niski možemo tražiti pojavljivanje neke druge niske, ne samo jednog karaktera. Na primer:

```
1 console.log('Ja sam niska!'.indexOf('sam')); // 3
```

Uklanjanje okolnih praznih karaktera

Često se pri parsiranju niski javlja potreba da se uklone početni i krajnji prazni karakteri (razmaci, novi redovi, tabulatori i sl.). Za to nam služi metod `trim`. Na primer:

```
1 console.log(' Ja imam okolni whitespace \n '.trim()); // Ja imam okolni
  whitespace
```

Razdvajanje niski po graničniku

Jednu nisku možemo podeliti na više tako što definišemo koja njena podniska treba da predstavlja *graničnik* (engl. *delimiter*) između dve niske, a zatim taj graničnik da iskoristimo kao argument metoda `split`. Rezultat je niz podniski. Na primer:

```
1 let recenica = 'Ja sam niska';
2 let reci = recenica.split(' ');
```

```
3
4 console.log(reci); // ['Ja', 'sam', 'niska']
```

Spajanje niski po graničniku

Inverzna operacija prethodno opisanoj se može postići metodom `join`, koja takođe koristi graničnik. Na primer:

```
1 let reci = ['Ja', 'sam', 'niska'];
2 let recenica = reci.join(' ');
3
4 console.log(recenica); // Ja sam niska
```

2.4.9 Funkcije sa proizvoljnim brojem parametara

Može biti korisno napisati funkcije koje prihvataju bilo koji broj parametara. Na primer, funkcija `Math.max` računa maksimum svih prosleđenih argumenata, koliko god da ih ima. Da bismo napisali ovakvu funkciju, potrebno je da stavimo tri tačke ispred poslednjeg parametra funkcije, na primer:

```
1 function nabrojSve(...stvari) {
2     for (let stvar of stvari) {
3         console.log(stvar);
4     }
5 }
6
7 nabrojSve('majica', 'sorts', 'dres', 'torba');
```

Ovakav parametar se naziva *parametar ostataka* (engl. *rest parameter*). Kada se ovakve funkcije pozivaju, parametar ostataka se vezuje za niz koji sadrži sve preostale argumente. Ukoliko postoje još neki parametri ispred parametra ostataka, oni se neće naći u nizu. U primeru funkcije `nabrojSve`, svi prosleđeni argumenti će biti deo tog niza jer ona ima samo taj jedan parametar.

Možemo koristiti sličnu notaciju sa tri tačke da pozovemo funkciju sa nizom argumenata, na primer:

```
1 let stvariZaPut = ['pasos', 'karte', 'kofer'];
2 nabrojSve(...stvariZaPut);
```

Ovakva notacija će *razbiti* (engl. *spread*) niz na pojedinačne argumente u pozivu funkcije. Možemo koristiti ovu notaciju u kombinaciji sa drugim argumentima, na primer:

```
1 let stvariZaPut = ['pasos', 'karte', 'kofer'];
2 nabrojSve('naocare za sunce', ...stvariZaPut, 'sesir');
```

2.4.10 Dekonstrukcija

U JavaScript jeziku je moguće vezati promenljive za elemente niza umesto za sam niz, a zatim te promenljive dalje koristiti u izračunavanjima. Ovo može biti korisno u, na primer, argumentima funkcija:

```
1 function veciOdDva_1(nizOd2) {
2     let prvi = nizOd2[0];
3     let drugi = nizOd2[1];
4
5     return prvi > drugi ? prvi : drugi;
6 }
7
```



```

8 function veciOdDva_2([prvi, drugi]) {
9     return prvi > drugi ? prvi : drugi;
10 }

```

Ova funkcionalnost radi i za promenljive uvedene pomoću `let`, `const` i `var`. Na primer:

```

1 function nadjiBrojManjiOd(niz, granicniBroj) {
2     for (let i = 0; i < niz.length; ++i) {
3         tekuciBroj = niz[i];
4         if (tekuciBroj < granicniBroj) {
5             return [tekuciBroj, i];
6         }
7     }
8
9     return [null, -1]
10 }
11
12 let niz = [5, 4, 3, 2, 1];
13 [broj, indeks] = nadjiBrojManjiOd(niz, 3);
14
15 if (indeks !== -1) {
16     console.log(broj); // 2
17 }

```

Sličan trik radi za objekte, samo što se umesto uglastih zagrada koriste vitičaste zgrade. Na primer:

```

1 let {ime} = {ime: 'Stefan', godine: 27};
2 console.log(ime); // Stefan

```

2.4.11 JSON

Podaci koji su sačuvani u memoriji mogu se *serijalizovati* (engl. *serialize*), odn. konvertovati u prenosivi zapis, na različite načine. Neki od njih su XNS, XML, razni binarni formati, i mnogi drugi. Jedan, za JavaScript programere prirodan zapis se naziva *JavaScript Object Notation* (skr. *JSON*). JSON izgleda slično kao pisanje literala nizova i objekata, uz neka ograničenja. Sva imena svojstava moraju biti okružena dvostrukim navodnicima, i samo su jednostavni tipovi podržani, dakle, nema funkcija kao vrednosti, uvođenja promenljivih ili bilo šta što podrazumeva izračunavanje. Čak ni komentari nisu dozvoljeni u JSON formatu.

Na primer, neka je dat naredni niz objekata koji predstavljaju osobe i neke njihove podatke:

```

1 let osobe = [
2     {
3         ime: 'Nikola',
4         godine: 24,
5         jezici: ['srpski', 'engleski'],
6         zeliSladoled: false
7     },
8     {
9         ime: 'Stefan',
10        godine: 27,
11        jezici: ['srpski', 'engleski', 'spanski'],
12        zeliSladoled: true
13    },
14    {
15        ime: 'Yuuki',
16        godine: 21,
17        jezici: ['japanski'],

```

```

18     zeliSladoled: true
19   },
20 ];

```

U JavaScript jeziku postoje funkcija `JSON.stringify` koja konvertuje argument koji joj se prosledi u nisku zapisanu u JSON formatu. Naredni fragment koda

```

1 let osobeJSON = JSON.stringify(osobe);
2 console.log(osobeJSON);

```

proizvodi sledeći izlaz:

```

1 '[{"ime":"Nikola","godine":24,"jezici":["srpski","engleski"],"zeliSladoled":
   false}, {"ime":"Stefan","godine":27,"jezici":["srpski","engleski","spanski"],
   "zeliSladoled":true}, {"ime":"Yuuki","godine":21,"jezici":["japanski"],"
   zeliSladoled":true}]'

```

Ovakva niska se zatim može proslediti kroz mrežu do udaljenog računara kako bi ti podaci bili vidljivi na njemu.

Ukoliko nam je na raspolaganju niska zapisana u JSON formatu, onda možemo iskoristiti ugrađenu funkciju `JSON.parse` da bismo iz takve niske dobili objekat koji možemo koristiti za dalja izračunavanja. Ovaj proces predstavlja inverznu operaciju serijalizaciji, te se naziva *deserijalizacija* (engl. *deserialize*). Na primer:

```

1 let osobeJSON = '[{"ime":"Nikola","godine":24,"jezici":["srpski","engleski"],
   "zeliSladoled":false}, {"ime":"Stefan","godine":27,"jezici":["srpski","
   engleski","spanski"],"zeliSladoled":true}, {"ime":"Yuuki","godine":21,"
   jezici":["japanski"],"zeliSladoled":true}]';
2
3 let osobe = JSON.parse(osobeJSON);
4 console.log(osobe[0].zeliSladoled); // false

```

Ako pokušamo da parsiramo nisku koja nije zapisana u validnom JSON formatu, JavaScript okruženje će prijaviti da je došlo do greške tipa `SyntaxError`.

Zadatak 2.7 U nastavku je zadataka niska zapisana u JSON formatu koja sadrži informacije o nekim proteinima. Analizirati date podatke i uraditi naredne zadatke:

- Napisati funkciju koja izračunava naziv proteina koji ima najveću dužinu poznate sekvence.
- Napisati funkciju koja kreira niz svih k -mera od fragmenta sekvence. Jedan k -mer predstavlja niz koji se sastoji od k uzastopnih aminokiselina (karaktera) iz datog fragmenta sekvence.
- Napisati funkciju koja grupiše proteine na osnovu njihove funkcije, tj. vraća objekat čiji su ključevi nazivi funkcija proteina, a vrednosti nizovi koji sadrže same proteine.
- Napisati funkciju koja izračunava naziv funkcije proteina koja ima najveći broj proteina koji tu funkciju vrše.

```

1 // Podaci:
2 let bazaProteina = '[{"sekvenca":"MSMDISDFYQTFDEADELL","naziv":"Chem.
   protein CheA","duzina":654,"organizam":"EC","sadrzaj uredjenosti":0.0993,
   "uredjenost":"Disorder","funkcije":["Flexible linker/spacer"]}, {"sekvenca":
   "MTTQVPPSALLPLNPEQLAR","duzina":599,"naziv":"SRF a-c", "organizam":"EC",
   "sadrzaj uredjenosti":0.2888,"uredjenost":"Disorder","funkcije":["Molecular
   recognition effector"]}, {"sekvenca

```

```

    "":"MKG TARVQTAREMKINGEIR", "duzina":180, "organizam":"EC", "naziv":"TIF
    333", "sadrzaj_uredjenosti":0.0667, "uredjenost":"Order", "funkcije":["
    Flexible linker/spacer", "Nuclear magnetic resonance", "Protein
    binding"]], {"sekvenca":"MGDEDWEAEINPHMSSYVPI", "duzina":690, "
    organizam":"HS", "naziv":"Protein 947", "sadrzaj_uredjenosti":0.2956, "
    uredjenost":"Disorder", "funkcije":["Liquid-liquid phase separation
    ", "Protein binding"]], {"sekvenca":"MASSCAVQVKLELGHRAQVR", "duzina
    ":768, "organizam":"HS", "naziv":"Isoform 2 of PATP-dep. RNA helix.
    DDX478", "sadrzaj_uredjenosti":0.139, "uredjenost":"Order", "funkcije
    ":["Molecular recognition assembler", "Protein binding"]}]';
3
4 // Resenje:
5 function f1() {
6 }
7
8 function f2(protein, k) {
9 }
10
11 function f3() {
12 }
13
14 function f4() {
15 }
16
17 // Testovi:
18 const protein_sa_najduzom_sekvencom = f1();
19 console.log(protein_sa_najduzom_sekvencom === 'Isoform 2 of PATP-dep.
    RNA helix. DDX478');
20
21 const k_meri = f2(bazaProteina[0], 3);
22 console.log(k_meri.length === 17); // Provera broja k-mera
23 console.log(k_meri[0].length === 3); // Provera duzine jednog k-mera
24 console.log(k_meri.join(' ') === 'M,S,M S,M,D M,D,I D,I,S I,S,D S,D,F D,
    F,Y F,Y,Q Y,Q,T Q,T,F T,F,F F,F,D F,D,E D,E,A E,A,D A,D,E D,E,L');
25
26 const recnik = f3();
27 console.log(recnik['Flexible linker/spacer'].length === 2);
28
29 const najzastupljenija_funkcija = f4();
30 console.log(najzastupljenija_funkcija === 'Protein binding');

```

2.5 Objektno-orijentisana paradigma

Kao što znamo, objektno-orijentisani pristup programiranju podrazumeva struktuiranje programa u klase i činjenicu da se izvršavanje programa svodi na interakciju objekata koje se instanciraju iz tih klasa. Enkapsulacija, odnosno, podela delova klase u javne i privatne, predstavlja jednu od glavnih paradigmi u objektno-orijentisanom pristupu programiranja. Neki programski jezici nude jasan pristup enkapsulaciji, poput C++ jezika, dok JavaScript (trenutno) to ne podržava. Ipak, to ne znači da se u JavaScript-u ne može programirati pomoću objektno-orijentisanog pristupa, već da su OOP tehnike drugačije od drugih jezika opšte namene.

2.5.1 Objekti (metodi) i vrednost this

Znamo da objektima možemo dinamički dodavati i uklanjati svojstva. Recimo da imamo prazan objekat koji ćemo dodeliti promenljivoj `zec` i neka je potrebno dodeliti tom objektu

metod `govori` koji će omogućiti zecu da kaže neku rečenicu:

```
1 let zec = {};  
2 zec.govori = function(recenica) {  
3   console.log(`Ovaj zec kaze: '${recenica}'`);  
4 };  
5  
6 zec.govori('Ja sam živ!'); // Ovaj zec kaze: 'Ja sam živ!'
```

Neka sada imamo dva zeca, `beliZec` i `gladanZec`, koji se razlikuju po tipu:

```
1 let beliZec = { tip: 'beli' };  
2 let gladanZec = { tip: 'gladan' };
```

Neka je sada potrebno da svaki od njih može da kaže neku rečenicu, pri čemu je potrebno reći o kojem je tipu zeca reč. Jedan način da ovo implementiramo jeste da njima dodelimo zasebnu funkciju `govori` koja će na zaseban način da označi o kojem je tipu reč:

```
1 beliZec.govori = function(recenica) {  
2   console.log(`Ovaj beli zec kaze: '${recenica}'`);  
3 };  
4  
5 gladanZec.govori = function(recenica) {  
6   console.log(`Ovaj gladan zec kaze: '${recenica}'`);  
7 };
```

Iako smo uradili šta je trebalo, ovi metodi ne uzimaju u obzir vrednosti svojstava `tip` koje ovi zečevi poseduju. Ovo može biti problem ukoliko odlučimo da nekom zecu promenimo njegov tip. To možemo ispraviti na naredni način:

```
1 beliZec.govori = function(recenica) {  
2   console.log(`Ovaj ${beliZec.tip} zec kaze: '${recenica}'`);  
3 };  
4  
5 gladanZec.govori = function(recenica) {  
6   console.log(`Ovaj ${gladanZec.tip} zec kaze: '${recenica}'`);  
7 };
```

Iako definitivno predstavlja poboljšanje, nekoliko očiglednih mana se odmah uočava. Pored toga što se praktično isti kod duplira, zamislimo situaciju u kojem bi trebalo implementirati ovaj metod nad stotinu različitih objekata zečeva. Ručno implementiranje svakog od metoda predstavlja zahtevan posao.

Ono što rešava oba problema jeste činjenica da možemo da koristimo specijalnu ključnu reč u okviru metoda objekta da dobijemo referencu ka objektu čiji je to metod. U pitanju je ključna reč `this`:

```
1 beliZec.govori = function(recenica) {  
2   console.log(`Ovaj ${this.tip} zec kaze: '${recenica}'`);  
3 };  
4  
5 gladanZec.govori = function(recenica) {  
6   console.log(`Ovaj ${this.tip} zec kaze: '${recenica}'`);  
7 };
```

Sada vidimo da možemo ići korak dalje, i izdvojiti ova dva metoda u jednu funkciju, s obzirom da imaju identičnu implementaciju:

```
1 function govori(recenica) {  
2   console.log(`Ovaj ${this.tip} zec kaze: '${recenica}'`);
```

```
3 }
4
5 let beliZec = { tip: 'beli', govori };
6 let gladanZec = { tip: 'gladan', govori };
7
8 // Ovaj beli zec kaze: 'Vidi koliko ima sati!'
9 beliZec.govori('Vidi koliko ima sati!');
10
11 // Ovaj gladan zec kaze: 'Dobro bi mi dosla sargarepa.'
12 gladanZec.govori('Dobro bi mi dosla sargarepa.');
```

2.5.2 Konstruktorske funkcije

Postoji još jedan način za kreiranje objekata. U pitanju je korišćenje *konstruktorskih funkcija* (engl. *constructor function*). Naredni primer ilustruje konstruktorsku funkciju za kreiranje zečeva:

```
1 function Zec(tip) {
2   this.tip = tip;
3   this.vrsta = 'zeka';
4   this.govori = function(recenica) {
5     console.log(`Ovaj ${this.tip} zec kaze: '${recenica}'`);
6   };
7 }
```

Ovu konstruktorsku funkciju možemo koristiti za kreiranje novih objekata zečeva na sledeći način:

```
1 // Obratiti paznju na ključnu rec new!
2 let braonZec = new Zec('braon');
3
4 console.log(braonZec.tip);      // 'braon'
5 console.log(braonZec.vrsta);    // 'zeka'
```

Kao što vidimo, možemo u konstruktorskoj funkciji dodeliti svojstva (samim tim i metode) objektu koji se kreira pomoću ključne reči **this**, a možemo im prosledivati argumente kao i bilo kojim drugim funkcijama.

Prema konvenciji, uvek bi trebalo konstruktorske funkcije nazivati velikim početnim slovom, čime se one razlikuju od običnih funkcija.

Dodatno, da bismo uspešno kreirali novi objekat pomoću konstruktorske funkcije, neophodno je da ispred njenog poziva navedemo ključnu reč **new**. Ukoliko to ne uradimo, JavaScript neće prijaviti grešku i na prvi pogled se dešava nešto veoma čudno:

```
1 let noviZec = Zec('novi');
2
3 // undefined
4 console.log(typeof noviZec);
5
6 // TypeError: Cannot read property 'tip' of undefined
7 console.log(typeof noviZec.tip);
```

Bez operatora **new**, konstruktorska funkcija se ponaša kao i bilo koja druga funkcija. Iz definicije funkcije **Zec**, vidimo da ona ne vraća nikakvu povratnu vrednost, a znamo da je u tom slučaju podrazumevana povratna vrednost funkcija **undefined**, koja se dodeljuje promenljivoj **noviZec**.

Međutim, ono što je dodatno čudno jeste naredno ponašanje:

```
1 console.log(tip); // 'novi'
```

O čemu je ovde reč. U funkciji `Zec` se referiše na vrednost `this`. Ukoliko je ta funkcija pozvana kao konstruktorska funkcija, `this` će predstavljati referencu ka novokreiranom objektu. U slučaju da `Zec` nije pozvana kao konstruktorska funkcija, onda se `this` odnosi na specijalan *globalni objekat*. Znamo da se JavaScript kod izvršava u nekom okruženju, bilo to veb pregledač ili konzolno okruženje kao što je Node.js. U svakom slučaju, u takvim okruženjima je definisan globalni objekat, i sve globalne promenljive zapravo predstavljaju svojstva tog globalnog objekta.

Na primer, ukoliko je JavaScript okruženje veb pregledač, onda se globalni objekat naziva `window`. U ovom okruženju (kao i u većini drugih okruženja), drugi način da se pristupi ovom globalnom objektu jeste pomoću ključne reči `this` izvan konstruktorskih funkcija, na primer, u globalnom dometu. Tako, na primer, možemo deklarirati globalnu promenljivu `a`, i pristupiti joj bilo kroz njeno ime, ili pomoću `this.a` ili `window.a` (u veb pregledaču):

```
1 var a = 1;
2
3 console.log(a); // 1
4 console.log(this.a); // 1
5 console.log(window.a); // 1
```

Ovim se objašnjava ono što smo smatrali za čudno ponašanje kada smo zaboravili da navedemo ključnu reč `new` ispred poziva konstruktorske funkcije.

Hajde da obradujemo novog zeca tako što ćemo ga ispravno kreirati:

```
1 let noviZec = new Zec('novi');
```

Svojstvo constructor

Kada je objekat kreiran, njemu se dodeljuje specijalno svojstvo koje se naziva `constructor`. Ovo svojstvo sadrži referencu ka konstruktorskoj funkciji koja se koristila za kreiranje tog objekta:

```
1 console.log(noviZec.constructor); // [Function: Zec]
```

Ovo svojstvo možemo iskoristiti za kreiranje novih objekata od već postojećih:

```
1 let josJedanNoviZec = new noviZec.constructor('drugi novi');
2
3 console.log(josJedanNoviZec.tip); // 'drugi novi'
```

Operator instanceof

Često je korisno znati da li je neki objekat kreiran pomoću neke konstruktorske funkcije. JavaScript jezik definiše infiksni operator `instanceof`, koji primenjen nad objektom i konstruktorskom funkcijom vraća `true` akko je objekat kreiran tom konstruktorskom funkcijom:

```
1 console.log(noviZec instanceof Zec); // true
```

Fabrika-funkcije

S obzirom da funkcije mogu da vraćaju objekte, onda ne moramo ni da koristimo konstruktorske funkcije da kreiramo objekte:

```
1 function fabrikaZeceva(tip) {
2     return {
3         tip: tip
```

```

4     };
5 }
6
7 let zec1 = fabrikaZeceva('1');
8 console.log(zec1.tip); // 1

```

U tom slučaju će `constructor` biti specijalna konstruktorska funkcija `Object`:

```

1 console.log(zec1.constructor); // [Function: Object]

```

Ono što je dodatno interesantno jeste da možemo da pregazimo podrazumevano ponašanje konstruktorske funkcije ukoliko postavimo da konstruktorska funkcija vrati objekat, što je ilustrovano slikom 2.3. Primer levo ilustruje standardno korišćenje konstruktorske funkcije. U primeru desno, umesto da bude vraćen objekat `this` koja sadrži svojstvo `a`, konstruktor vraća drugi objekat koji nema to svojstvo, ali koje ima svojstvo `b`. Napomenimo da ovo ponašanje dolazi do izražaja samo ukoliko je povratna vrednost konstruktorske funkcije objekat, dok u ostalim slučajevima biva vraćena vrednost `this`.

<pre> 1 function C() { 2 this.a = 1; 3 } 4 5 let c = new C(); 6 7 // 1 8 console.log(c.a); </pre>	<pre> 1 function C2() { 2 this.a = 1; 3 return {b: 2}; 4 } 5 6 let c2 = new C2(); 7 8 // 'undefined' 9 console.log(typeof c2.a); 10 11 // 2 12 console.log(c2.b); </pre>
---	--

Slika 2.3: Primer dva fragmenta koda koji koriste konstruktorske funkcije u normalnom scenariju (levo) i u situaciji kada konstruktorska funkcija vraća objekat, čime se podrazumevano ponašanje pregazuje (desno).

Rešavanje problema poziva konstruktorskih funkcija bez operatora `new`

Prethodna tehnika se može iskoristiti za rešavanje problema kada korisnik poziva konstruktorsku funkciju bez navođenja `new` ispred poziva funkcije. Rešenje se može izvesti na naredni, jednostavan način. Ono što je potrebno uraditi jeste ispitati da li je objekat instanca tipa koji definiše konstruktorska funkcija. U slučaju da nije, onda je potrebno rekursivno pozvati konstruktorsku funkciju, ali ovoga puta dodavanjem ključne reči `new` ispred poziva, i to vratiti kao povratnu vrednost funkcije. Naredni primer ilustruje ovo ponašanje:

```

1 function MyObject(arg) {
2     if (!(this instanceof MyObject)) {
3         return new MyObject(arg);
4     }
5
6     this.arg = arg;
7 }
8
9 var obj = new MyObject(1);
10 console.log(obj);           // { arg: 1 }
11 console.log(obj instanceof MyObject); // true
12

```

```

13 var obj2 = MyObject(2);
14 console.log(obj2);           // { arg: 2 }
15 console.log(obj2 instanceof MyObject); // true
16
17 console.log(typeof arg);      // undefined

```

S obzirom da poslednja linija u primeru ispisuje vrednost `undefined`, onda smo sigurni da poziv funkcije bez `new` nije kreirao globalne promenljive.

2.5.3 Leksičko `this`

O vrednosti `this` možemo da razmišljamo kao o dodatnom parametru koji se prosleđuje na drugačiji način. Ako želimo da ga eksplicitno pozovemo, možemo pozvati metod `call` nad određenom funkcijom, čiji je prvi argument ono što će biti smatrano za `this`, a ostali parametri su regularni parametri te funkcije:

```

1 function govori(recenica) {
2   console.log(`Ovaj ${this.tip} zec kaze: '${recenica}'`);
3 }
4 let gladanZec = { tip: 'gladan' };
5
6 // Ovaj gladan zec kaze: 'Najeo sam se'
7 govori.call(gladanZec, 'Najeo sam se');

```

Kako svaka funkcija ima svoju vrednost za `this`, čija vrednost zavisi od načina pozivanja, ne možemo da referišemo na `this` iz dometa oko funkcije, ako je funkcija definisana ključnom reči `function`. Na primer:

```

1 'use strict';
2 let greeter = {
3   default: "Hello ",
4   greet: function (names) {
5     names.forEach(function(name) {
6       // TypeError: Cannot read property 'default' of undefined
7       console.log(this.default + name);
8     });
9   }
10 }
11
12 console.log(greeter.greet(['world', 'heaven']));

```

U prethodnom primeru dolazi do greške jer pokušavamo da pristupimo svojstvu `default` iz dometa anonimne funkcije (koju ćemo nazivati podrutina), umesto iz dometa oko nje. Ova podrutina ima nedefinisanu vrednost za `this` i nažalost, nema pristup vrednosti `this` iz spoljnjeg dometa. Ono što je ovoj podrutini potrebno jeste tzv. *leksičko* (engl. *lexical*) `this`, odnosno, da izvede vrednost za `this` iz spoljašnjeg dometa. Tradicionalan pristup rešavanju ovog problema jeste dodeljivanje vrednosti `this` nekoj promenljivoj, koja će biti dostupna unutrašnjoj funkciji:

```

1 'use strict';
2 let greeter = {
3   default: "Hello ",
4   greet: function (names) {
5     let self = this;
6
7     names.forEach(function(name) {
8       console.log(self.default + name);
9     });
10  }

```



```

11 }
12
13 console.log(greeter.greet(['world', 'heaven']));
14 // Hello world
15 // Hello heaven

```

Međutim, zahvaljujući ES6 standardu, isti efekat se može postići pomoću lambda funkcija. Lambda funkcije su drugačije — one ne vezuju svoju vrednost za `this` i mogu da vide `this` iz spoljašnjeg dometa oko njih. Drugim rečima, ako anonimnu funkciju iz prethodnog primera zamenimo anonimnom lambda funkcijom, dobićemo ispravan rezultat:

```

1 'use strict';
2 let greeter = {
3   default: "Hello ",
4   greet: function (names) {
5     names.forEach((name) => {
6       console.log(this.default + name);
7     });
8   }
9 }
10
11 console.log(greeter.greet(['world', 'heaven']));
12 // Hello world
13 // Hello heaven

```

2.5.4 Prototipovi

Pogledajmo naredni scenario:

```

1 let empty = {};
2
3 console.log(empty.toString()); // [Function: toString]
4 console.log(empty.toString()); // [object Object]

```

Izvlačenje svojstva iz praznog objekta podseća na trik izvlačenja zeca iz praznog šesira. Međutim, nikakva magija nije u pitanju (kao ni u starom mađioničarskom triku), već samo do sada nismo spomenuli da objekti, pored svojstava, takođe imaju i *prototip* (engl. *prototype*). Prototip predstavlja objekat koji služi kao izvor svojstava za pretragu — kada objekat dobije zahtev za svojstvom koji se ne nalazi direktno u njemu, onda će se pretražiti njegov prototip i videti da li tu ima svojstvo. Ukoliko ima, iskoristiće se, a inače će se pretraživati prototip prototipa, itd.

Šta je onda prototip praznog objekta? Gotovo svi objekti imaju istog pretka — prototip `Object.prototype`:

```

1 console.log(Object.getPrototypeOf({}) === Object.prototype); // true
2 console.log(Object.getPrototypeOf(Object.prototype)); // null

```

Kao što vidimo, metod `Object.getPrototypeOf` vraća prototip objekta.

Prototipski odnos JavaScript objekata formira drvoliku strukturu, a u korenu ove strukture nalazi se `Object.prototype` (iz prethodnog primera vidimo da ovaj objekat nema svoj prototip). On daje nekoliko metoda koje se pojavljuju u svim objektima, kao što je metod `toString`, koji konvertuje objekat u reprezentaciju pomoću niske.

Mnogi objekti nemaju direktno `Object.prototype` kao njihov prototip već umesto toga imaju drugi objekat koji daje drugi skup podrazumevanih svojstava. Funkcije se izvode iz `Function.prototype`, dok se nizovi izvode iz `Array.prototype`:

```

1 console.log(Object.getPrototypeOf(Math.max) === Function.prototype); // true
2 console.log(Object.getPrototypeOf([]) === Array.prototype);           // true

```

Ovakvi prototipovi će i sami imati svoje prototipove, često baš `Object.prototype`, tako da će indirektno imati metode poput `toString`.

Sada kada smo se upoznali sa pojmom prototipa i time šta nas očekuje u ovoj podsekciji, hajde da detaljnije diskutujemo o nekim svojstvima jezika JavaScript koji se tiču prototipova.

Augmentacija prototipa

U prethodnog tekstu, naučili smo kako da definišemo konstruktorske funkcije koje možemo da koristimo za kreiranje (konstruisanje) novih objekata. Glavna ideja je da, unutar funkcije koja je pozvana ključnom rečju `new`, dobijamo pristup vrednosti `this`, koja referiše na objekat koji će biti vraćen konstruktorskom funkcijom. *Augmentacija* (engl. *augmenting*), odnosno, dodavanje svojstava i metoda, predstavlja način za obogaćivanje konstruisanog objekta dodatnim funkcionalnostima.

Pogledajmo konstruktorsku funkciju `Gadget` koja koristi `this` da doda dva svojstva i jedan metod objektu koji kreira:

```

1 function Gadget(name, color) {
2     this.name = name;
3     this.color = color;
4     this.whatAreYou = function () {
5         return 'I am a ' + this.color + ' ' + this.name;
6     };
7 }

```

Pored augmentacije vrednosti `this`, može se augmentisati i sam prototip, dostupan kroz svojstvo `prototype` konstruktorske funkcije, čime se dodaje funkcionalnost objektima koje konstruktor proizvede. Dodajmo još dva svojstva `price` i `rating`, kao i metod `getInfo`. Kako `prototype` već pokazuje na objekat, možemo jednostavno dodavati svojstva i metode na njega:

```

1 Gadget.prototype.price = 100;
2 Gadget.prototype.rating = 3;
3 Gadget.prototype.getInfo = function () {
4     return `Rating: ${this.rating}, price: ${this.price}`;
5 };

```

Alternativno, umesto dodavanje svojstva objektu prototipa jedan po jedan, možemo pregaziti prototip u potpunosti, menjajući ga objektom po izboru, kao u narednom primeru:

```

1 Gadget.prototype = {
2     price: 100,
3     rating: ... /* and so on... */
4 };

```

Sva svojstva i svi metodi koje smo dodali prototipu su dostupni čim konstruišemo novi objekat korišćenjem konstruktorske funkcije:

```

1 var newtoy = new Gadget('webcam', 'black');
2 console.log(newtoy.name);           // 'webcam'
3 console.log(newtoy.color);          // 'black'
4 console.log(newtoy.whatAreYou());    // 'I am a black webcam'
5 console.log(newtoy.price);           // 100
6 console.log(newtoy.rating);          // 3
7 console.log(newtoy.getInfo());       // 'Rating: 3, price: 100'

```

Važno je razumeti da je prototip "živ". Kao što znamo, objekti se u JavaScript jeziku prosleđuju po njihovoj adresi⁹, tako da se prototip ne kopira sa svakom novom instancom objekta. To znači da možemo da modifikujemo prototip u bilo kom trenutku, i svi objekti, čak i oni koji su kreirani pre izmene, videće te izmene. Nastavimo prethodni primer dodavanjem novog metoda prototipu:

```
1 Gadget.prototype.get = function (what) {  
2     return this[what];  
3 };
```

Bez obzira što je objekat `newtoy` kreiran pre nego što je definisan metod `get`, objekat `newtoy` ipak ima pristup novom metodu, što se vidi narednim fragmentom koda:

```
1 console.log(newtoy.get('price')); // 100  
2 console.log(newtoy.get('color')); // 'black'
```

U prethodnom primeru, metod `getInfo` se koristio interno da pristupi svojstvu objekta. Mogli smo da iskoristimo `Gadget.prototype` da postignemo isti efekat:

```
1 Gadget.prototype.getInfo = function () {  
2     return `Rating: ${Gadget.prototype.rating}, ` +  
3         `price: ${Gadget.prototype.price}`;  
4 };
```

U čemu je onda razlika? Da bismo došli do odgovora, potrebno je da razumemo kako prototip funkcioniše. Pogledajmo objekat `newtoy` opet:

```
1 var newtoy = new Gadget('webcam', 'black');
```

Kada pokušamo da pristupimo svojstvu objekta `newtoy`, na primer, `newtoy.name`, JavaScript mašina pretražuje kroz sva svojstva objekta i pokušava da pronađe ono svojstvo koje se naziva `name`. Ukoliko ga nađe, onda će dohvatiti njegovu vrednost:

```
1 console.log(newtoy.name); // 'webcam'
```

Šta ukoliko bismo pokušali da pristupimo svojstvu `rating`? JavaScript mašina ispituje sva svojstva objekta `newtoy` i ne pronalazi svojstvo čiji je naziv `rating`. Zatim, mašina identifikuje prototip konstruktorske funkcije koja se koristila da kreira ovaj objekat. Ukoliko je svojstvo pronađeno u objektu prototipa, onda će ono biti dohvaćeno:

```
1 console.log(newtoy.rating); // 3
```

U ovom slučaju smo svojstvu mogli da pristupimo iz prototipa direktno. Kao što znamo, svaki objekat ima svojstvo `constructor`, što predstavlja referencu na funkciju koja je kreirala objekat, te se pristupanje svojstva može uraditi na sledeći način:

```
1 console.log(newtoy.constructor === Gadget); //true  
2 console.log(newtoy.constructor.prototype.rating); // 3
```

Hajde da sada ovu pretragu proširimo još jedan korak dalje. Svaki objekat ima `constructor`. Prototip je objekat, stoga on mora da ima konstruktor takođe, koji opet ima svoj prototip. Penjanjem uz ovaj lanac prototipova u jednom trenutku ćemo završiti sa ugrađenim objektom tipa `Object`, koji se nalazi na vrhu tog lanca. U praksi, ovo znači da ukoliko pokušamo da pristupimo metodi `newtoy.toString()` i `newtoy` nema svoj metod `toString`, kao i njegov prototip, na kraju ćemo dohvatiti `toString` metod iz `Object`:

```
1 console.log(newtoy.toString()); // '[object Object]'
```

⁹Dakle, njihova referenca se prenosi po vrednosti.

Prevazilaženje svojstava iz prototipa

Kao što prethodna diskusija prikazuje, ako jedan od naših objekata nema neko svojstvo, on može da koristi istoimeno svojstvo, ukoliko ono postoji, negde u lancu prototipova. Međutim, šta se dešava ukoliko objekat sadrži neko svojstvo, a i njegov prototip takođe sadrži to svojstvo? U takvoj situaciji, prednost ima svojstvo koje se nalazi u objektu u odnosu na svojstvo u prototipu. Razmotrimo naredni scenario u kojem svojstvo `name` postoji i na nivou svojstva i na nivou prototipa:

```
1 function Gadget(name) {
2     this.name = name;
3 }
4 Gadget.prototype.name = 'mirror';
```

Kreiranje novog objekta i pristup svojstvu `name` rezultuje u dohvatanju svojstva iz objekta:

```
1 var toy = new Gadget('camera');
2 console.log(toy.name); // 'camera'
```

Da bismo otkrili gde je svojstvo definisati, možemo koristiti metod `hasOwnProperty`, kao u narednom primeru:

```
1 console.log(toy.hasOwnProperty('name')); // true
```

Ako bismo obrisali svojstvi `name` iz objekta `toy`, tada bi istoimeno svojstvo iz prototipa bilo vidljivo:

```
1 delete toy.name;
2
3 console.log(toy.name); // 'mirror'
4 console.log(toy.hasOwnProperty('name')); // false
```

Naravno, uvek možemo rekreirati svojstvo u objektu:

```
1 toy.name = 'camera';
2 console.log(toy.name); // 'camera'
```

Možemo koristiti metod `hasOwnProperty` da pronađemo poreklo svojstva koje nas interesuje. Na primer, možemo videti da `toString` zaista potiče iz `Object.prototype`:

```
1 console.log(toy.toString()); // '[object Object]'
2
3 console.log(toy.hasOwnProperty('toString')); // false
4 console.log(toy.constructor.hasOwnProperty('toString')); // false
5 console.log(toy.constructor.prototype.hasOwnProperty('toString')); // false
6 console.log(Object.hasOwnProperty('toString')); // false
7 console.log(Object.prototype.hasOwnProperty('toString')); // true
```

Ispitivanje prototipa

Objekti takođe imaju i metod `isPrototypeOf`. Ovim metodom možemo proveriti da li je specifičan objekat korišćen kao prototip drugog objekta. Pogledajmo ovo na narednom primeru. Neka nam je dat objekat `monkey`:

```
1 var monkey = {
2     hair: true,
3     feeds: 'bananas',
4     breathes: 'air'
5 };
```

Sada, kreirajmo konstruktorsku funkciju `Human` i postavimo njeno svojstvo `prototype` da pokazuje na `monkey`:

```
1 function Human(name) {  
2   this.name = name;  
3 }  
4 Human.prototype = monkey;
```

Sada, ukoliko kreiramo novi objekat tipa `Human` sa imenom `george` i pitamo da li je `monkey` prototip od `george`, dobićemo potvrđan odgovor:

```
1 var george = new Human('George');  
2 console.log(monkey.isPrototypeOf(george)); // true
```

Primetimo da je potrebno da znamo, ili da makar pretpostavimo koji objekat je prototip, da bismo pitali da li je neki objekat prototip drugom objektu, u cilju potvrđivanja hipoteze. Međutim, šta raditi u slučaju da nemamo ideju koji objekat bi mogao da bude prototip? Da li je ovaj odgovor uopšte moguće dobiti? Odgovor je potvrđan, ali sa napomenom da nije moguće u svim veb pregledačima. Većina pregledača u poslednjim verzijama imaju implementiran dodatak ES5 standardu: metod `Object.getPrototypeOf`.

```
1 console.log(Object.getPrototypeOf(george).feeds); // 'bananas'  
2 console.log(Object.getPrototypeOf(george) === monkey); // true
```

Za neka JavaScript okruženja koja ne podržavaju ES5 standard i metod `Object.getPrototypeOf`, može se koristiti specijalno svojstvo `__proto__`.

Svojstvo `__proto__`

Kao što već znamo, svojstvo `prototype` se konsultuje prilikom pretraživanja svojstva koje ne postoji u objektu nad kojim se ono dohvata. Razmotrimo drugi objekat koji se naziva `monkey`, i iskoristimo ga kao prototip prilikom kreiranja objekata konstruktorskom funkcijom `Human`:

```
1 var monkey = {  
2   feeds: 'bananas',  
3   breathes: 'air'  
4 };  
5 function Human() {}  
6 Human.prototype = monkey;
```

Sada, kreirajmo objekat `developer`, i dajmo mu naredna svojstva:

```
1 var developer = new Human();  
2 developer.feeds = 'pizza';  
3 developer.hacks = 'JavaScript';
```

Sada, potražimo ova svojstva u objektu (na primer, svojstvo `hacks` je svojstvo objekta `developer`):

```
1 console.log(developer.hacks); // 'JavaScript'
```

Svojstvo `feeds` se takođe može pronaći u objektu:

```
1 console.log(developer.feeds); // 'pizza'
```

Za razliku od njih, svojstvo `breathes` ne postoji kao svojstvo na nivou objekta `developer`, te se pretražuje prototip, kao da postoji tajna spona između objekta i prototipa ili hodnik koji vodi do objekta prototipa:

```
1 console.log(developer.breathes);    // 'air'
```

Ta tajna spona je izložena u većini savremenih JavaScript okruženja kroz svojstvo `__proto__` — ime svojstva obuhvata reč `proto` sa po dva karaktera podvlake ispred i iza te reči:

```
1 console.log(developer.__proto__ === monkey);    // true
```

Možemo koristiti ovo tajno svojstvo prilikom procesa učenja, ali nije dobra ideja da ga koristimo prilikom pisanja skriptova u praksi, pre svega zato što ne postoji u svim pregledačima (na primer, IE), tako da skriptovi neće biti portabilni.

Prilikom učenja se često preleti preko jedne važne teme, a to je razlika između `__proto__` i `prototype`. Svojstvo `__proto__` je svojstvo instanci tipa (objekata), dok je `prototype` svojstvo konstruktorskih funkcija koje se koriste za kreiranje tih objekata:

```
1 console.log(typeof developer.__proto__);          // 'object'
2
3 console.log(typeof developer.prototype);          // 'undefined'
4 console.log(typeof developer.constructor.prototype); // 'object'
```

Još jednom napomenimo da bi se svojstvo `__proto__` trebalo koristiti samo tokom procesa učenja ili debugovanja. Alternativno, ukoliko je dovoljno da kod radi u skladu sa ES5 standardom, onda možemo koristiti `Object.getPrototypeOf`.

Kreiranje objekata sa određenim prototipom

Možemo koristiti metod `Object.create` da bismo kreirali objekat sa specifičnim prototipom. Na primer:

```
1 let protoZec = {
2     govori(recenica) {
3         console.log(`Ovaj ${this.tip} zec kaze: '${recenica}'`);
4     }
5 };
6
7 let tuzanZec = Object.create(protoZec);
8 tuzanZec.tip = 'tuzan';
9 tuzanZec.govori('Nema vise sargarepe :(');
10 // Ovaj tuzan zec kaze: 'Nema vise sargarepe :('
```

Svojstvo `govori(recenica)` iz prethodnog primera, kao deo izraza objekta, predstavlja kraći način za definisanje metoda. Ovim se kreira svojstvo koje se naziva `govori` čija je vrednost funkcija.

”Proto” zec služi kao šablon za svojstva koja dele svi zečevi. Individualni objekti-zečevi, poput tužnog zeca, sadrže svojstva koja se vezuju samo za njih — u ovom primeru njihov tip — i nasleđuju deljena svojstva iz svog prototipa.

Više o pregazivanju prototipova

Naredna dva ponašanja su veoma bitna da se razumeju i zapamte prilikom rada sa prototipovima:

- Lanac prototipova je ”živ”, osim ukoliko u potpunosti pregazimo objekat prototipa.
- Metod `prototype.constructor` nije pouzdan.

Kreirajmo jednostavnu konstruktorsku funkciju i dva objekta:

```
1 function Dog() {
2     this.tail = true;
```

```
3 }
4 var benji = new Dog();
5 var rusty = new Dog();
```

Do sada smo naučili da čak i nakon što smo kreirali objekte `benji` i `rusty`, dodavanjem novih svojstava prototipu tipa `Dog` omogućićemo da postojeći objekti imaju pristup njima. Dodajmo metod `say` na nivou prototipa:

```
1 Dog.prototype.say = function () {
2   return 'Woof!';
3 };
```

Očigledno, oba postojeća objekta imaju pristup novom metodu:

```
1 console.log(benji.say()); // 'Woof!'
2 console.log(rusty.say()); // 'Woof!'
```

Do ovog trenutka, ukoliko bismo se konsultovali na način psima i pitali ih na osnovu koje konstruktorske funkcije su kreirani, oni će nam reći ispravan odgovor:

```
1 console.log(benji.constructor === Dog); // true
2 console.log(rusty.constructor === Dog); // true
```

Hajde sada da kompletno pregazimo objekat prototipa potpuno novim objektom:

```
1 Dog.prototype = {
2   paws: 4,
3   hair: true
4 };
```

Ono što se ispostavlja je da stari objekti nemaju pristup svojstvima novog prototipa; oni i dalje sadrže tajnu sponu koja pokazuje na objekat starog prototipa, što se vidi iz narednog fragmenta koda:

```
1 console.log(typeof benji.paws);           // 'undefined'
2 console.log(benji.say());                 // 'Woof!'
3
4 console.log(typeof benji.__proto__.say);   // 'function'
5 console.log(typeof benji.__proto__.paws);  // 'undefined'
```

Za razliku od postojećih psa, svaki novi objekat koji kreiramo od ovog trenutka će koristiti ažurirani prototip:

```
1 var lucy = new Dog();
2 console.log(lucy.say()); // TypeError: lucy.say is not a function
3 console.log(lucy.paws);  // 4
```

Tajna spona `__proto__` pokazuje na novi objekat prototipa, što se vidi iz narednih linija koda:

```
1 console.log(typeof lucy.__proto__.say);   // 'undefined'
2 console.log(typeof lucy.__proto__.paws);  // 'number'
```

Međutim, sada svojstvo `constructor` novog objekta ne izveštava ispravnu konstruktorsku funkciju. Ono što očekujemo jeste da svojstvo pokazuje na konstruktorsku funkciju `Dog`, međutim, ono pokazuje na `Object`, što se može videti iz narednog primera:

```

1 console.log(lucy.constructor);
2 // function Object() { [native code] }
3
4 console.log(benji.constructor);
5 // function Dog() {
6 //     this.tail = true;
7 // }

```

Na našu sreću, ova konfuzija se jednostavno rešava resetovanjem svojstva `constructor` nakon pregazivanja prototipa, kao u narednom primeru:

```

1 function Dog() {}
2 Dog.prototype = {};
3
4 console.log(new Dog().constructor === Dog); // false
5
6 // Resetovanje svojstva constructor
7 Dog.prototype.constructor = Dog;
8
9 console.log(new Dog().constructor === Dog); // true

```

Zbog ovog ponašanja uvek treba imati na umu naredno pravilo:

Prilikom pregazivanja prototipa, potrebno je resetovati svojstvo `constructor`.

Učauravanje

Kada je objektno-orijentisani pristup programiranja u pitanju, jedna od najznačajnijih pitanja prilikom projektovanje klase jeste koja svojstva i metodi će biti vidljivi spoljašnjem svetu, odnosno, dostupni korisniku na korišćenje, a šta će biti sakriveno od njega. Zatvaranje unutrašnje strukture objekta i sakrivanje detalja implementacije od korisnika se naziva *učauravanje* (engl. *encapsulation*).

Sa znanjem koje smo stekli do sada, ne bi trebalo da bude komplikovano razumeti način na koji se učauravanje implementira u JavaScript jeziku. Pogledajmo naredni primer:

```

1 function MyObject(publicVar) {
2     // Private area
3     var privateVar = 'result';
4
5     function privateF() {
6         console.log('Doing something...');
7         return privateVar;
8     }
9
10    // Public area
11    this.publicVar = publicVar;
12    this.publicF = function() {
13        return privateF();
14    };
15 }

```

Ukoliko želimo da učaurimo promenljive, dovoljno je da ih deklariramo unutar tela funkcije. Promenljive će biti dostupne javno tek kada ih dodelimo kao svojstva novokreiranoj instanci pomoću `this` vrednosti. Slično važi i za metode. Kao što vidimo, samo su javni članovi tipa dostupni korisniku, dok pristupanje privatnim članovima tipa rezultuje vraćanjem `undefined` vrednosti:

```

1 var obj = new MyObject(1);
2

```



```
3 // Accessing public members
4 console.log(obj.publicVar);      // 1
5 console.log(obj.publicF());     // Doing something...
6                                 // result
7
8 // Accessing private members
9 console.log(typeof obj.privateVar); // undefined
10 console.log(typeof obj.privateF);  // undefined
```

Očigledno, s obzirom da se privatni članovi definišu isključivo u konstruktorskoj funkciji, to znači da će svaka instanca sadržati svoju kopiju tih članova.

2.5.5 Nasleđivanje

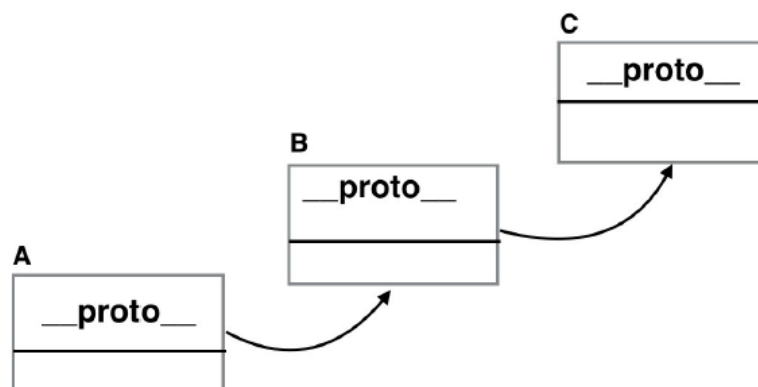
Nasleđivanje se u JavaScript jeziku može implementirati na više načina. U skladu sa time, u ovoj podsekciji ćemo prikazati jedan od standardnih obrazaca za implementiranje koncepta nasleđivanja — *lanac prototipova*. Nakon toga ćemo diskutovati o nekim specifičnostima klasične implementacije i pokušati da je nadogradimo.

Lanac prototipova

Pojam lanca prototipova smo do sada pominjali nekoliko puta, međutim, u ovom delu teksta ćemo se nešto detaljnije osvrnuti na njega, s obzirom da on igra važnu ulogu u nasleđivanju u jeziku JavaScript.

Kao što već znamo, svaka funkcija sadrži svojstvo `prototype`, koja referiše na neki objekat. Kada je funkcija pozvana uz operator `new`, kreira se novi objekat (instanca tipa konstruktorske funkcije) i vraća se kao povratna vrednost. Ovaj novokreirani objekat ima tajnu sponu ka objektu prototipa. Tajna spona (koja se naziva `__proto__` u nekim JavaScript okruženjima) dozvoljava da se svojstva i metodi objekta prototipa koriste kao da pripadaju novokreiranom objektu.

Objekat prototipa je regularan objekat, te stoga i on sadrži tajnu sponu ka njegovom prototipu. Na taj način se formira lanac prototipova, što je ilustrovano na slici 2.4. Na ovoj ilustraciji, objekat **A** sadrži neka svojstva među kojima je i sakriveno svojstvo `__proto__`, koje referiše na drugi objekat koji se naziva **B**. Svojstvo `__proto__` objekta **B** dalje referiše na objekat **C**. Ovaj lanac se završava objektom `Object.prototype`, pretkom svih objekata.



Slika 2.4: Ilustracija lanca prototipova.

Ovo je dobro znati, ali kako nam to može pomoći? Praktična strana ove pojave jeste da kada objekat **A** ne sadrži neko svojstvo, ali ono se nalazi u objektu **B**, onda **A** i dalje može

da mu pristupi kao da je njegovo. Slično važi i ako **B** ne sadrži neko svojstvo, ali **C** ga sadrži. Upravo ovo ponašanje oslikava način na koji se nasleđivanje ostvaruje — objekat može da pristupi svojstvu koji se nalazi negde u lancu nasleđivanja.

Lanac prototipova je osnovni metod za implementaciju nasleđivanja. Da bismo ilustrovali način na koji ovaj metod funkcioniše, definišimo tri konstruktorske funkcije:

```

1 function Shape() {
2     this.name = 'Shape';
3     this.toString = function () {
4         return this.name;
5     };
6 }
7
8 function TwoDShape() {
9     this.name = '2D shape';
10 }
11
12 function Triangle(side, height) {
13     this.name = 'Triangle';
14     this.side = side;
15     this.height = height;
16     this.getArea = function () {
17         return this.side * this.height / 2;
18     };
19 }
```

Kod koji omogućava "magiju" nasleđivanja je:

```

1 TwoDShape.prototype = new Shape();
2 Triangle.prototype = new TwoDShape();
```

Kakav je efekat ovih dveju linija koda? Prvo, mi uzimamo objekat koji se nalazi u svojstvu **prototype** tipa **TwoDShape**, i umesto da ga augmentiramo individualnim svojstvima, mi ga u potpunosti pregazimo drugim objektom, koji je kreiran pozivom funkcije **Shape** kao konstruktorske funkcije, čime se uspostavlja odnos između tipova **Shape** i **TwoDShape**. Isti proces se može ispratiti za **Triangle** — njegov prototip se zamenjuje objektom kreiranim pozivom **new TwoDShape()**.

Važno je zapamtiti da JavaScript radi sa objektima, a ne sa klasama. Zbog toga je neophodno kreirati instance tipova (kao što smo to radili pozivom, na primer, **new Shape()**), čime se nasleđuju njegova svojstva; ne vršimo nasleđivanje pozivom funkcije direktno (na primer, **Shape()**). Dodatno, nakon nasleđivanja, možemo modifikovati konstruktor **Shape**, pregaziti ga ili ga čak obrisati, i ovo neće imati efekta na **TwoDShape**, zato što je neophodno postojanje barem jedne instance iz koje se vrši nasleđivanje.

Već smo govorili o tome da pregazivanje objekta prototipa (nasuprot njegovoj augmentaciji) ima bočni efekat na svojstvo **constructor**. Zbog toga, dobra je ideja resetovati to svojstvo nakon nasleđivanja. Iz prethodnog primera zaključujemo da treba dodati naredne dve linije koda:

```

1 TwoDShape.prototype.constructor = TwoDShape;
2 Triangle.prototype.constructor = Triangle;
```

Sada, testirajmo šta je do sada urađeno. Kreiranje objekta **Triangle** i pozivanje metoda **getArea** koje on sadrži radi kako je i očekivano:

```

1 var my = new Triangle(5, 10);
2 console.log(my.getArea()); // 25
```

Iako objekat `my` ne sadrži metod `toString`, on nasleđuje istoimeni metod iz tipa `Shape` koji možemo da pozivamo. Primetimo jednu važnu stvar — nasleđeni metod `toString` vezuje referencu `this` na objekat `my`:

```
1 console.log(my.toString()); // 'Triangle'
```

U nastavku slede koraci koje JavaScript mašina uradi tokom poziva `my.toString()`:

- Izvršava se pretraživanje svih svojstava objekta `my` i ne pronalazi se metod sa nazivom `toString`.
- Pretraga se pomera u objekat na koji referiše svojstvo `my.__proto__`, što je instanca objekta kreirana pomoću poziva `new TwoDShape()` u procesu nasleđivanja.
- Izvršava se pretraživanje svih svojstava instance `TwoDShape` i ne pronalazi se metod sa nazivom `toString`.
- Pretraga se pomera u objekat na koji referiše svojstvo `__proto__` tekućeg objekta. Ovog puta, `__proto__` referiše na instancu kreiranu pomoću poziva `new Shape()`.
- Izvršava se pretraživanje svih svojstava instance `Shape` i pronalazi se metod sa nazivom `toString`.
- Pronađeni metod se poziva u kontekstu objekta `my`, što praktično znači da `this` u metodi `toString` referiše na `my`.

Ukoliko bismo upitali našeg trougla `my` ko je njegov konstruktor, on se neće zbuniti i reći će nam ispravan odgovor jer smo resetovali svojstvo `constructor` nakon nasleđivanja:

```
1 console.log(my.constructor === Triangle); // true
```

Korišćenjem operatora `instanceof`, možemo se uveriti da je objekat `my` zaista instanca svih triju konstruktora:

```
1 console.log(my instanceof Shape); // true
2 console.log(my instanceof TwoDShape); // true
3 console.log(my instanceof Triangle); // true
4
5 console.log(my instanceof Array); // false
```

Isto se primećuje prilikom poziva metoda `isPrototypeOf` nad prototipovima konstruktora i prosleđivanjem objekta `my`:

```
1 console.log(Shape.prototype.isPrototypeOf(my)); // true
2 console.log(TwoDShape.prototype.isPrototypeOf(my)); // true
3 console.log(Triangle.prototype.isPrototypeOf(my)); // true
4
5 console.log(String.prototype.isPrototypeOf(my)); // false
```

Zadatak 2.8 Posmatrajmo naredni fragment koda:

```
1 // TwoDShape test
2 var td = new TwoDShape();
3
4 console.log(td.constructor === TwoDShape);
5
6 if (typeof td.toString !== 'undefined') {
7     console.log(td.toString());
8 }
9 else {
10     console.log('td.toString is undefined');
11 }
```

```

12
13 if (typeof td.getArea !== 'undefined') {
14     console.log(td.getArea());
15 }
16 else {
17     console.log('td.getArea is undefined');
18 }
19
20 // Shape test
21 var s = new Shape();
22
23 console.log(s.constructor === Shape);
24
25 if (typeof s.toString !== 'undefined') {
26     console.log(s.toString());
27 }
28 else {
29     console.log('s.toString is undefined');
30 }
31
32 if (typeof s.getArea !== 'undefined') {
33     console.log(s.getArea());
34 }
35 else {
36     console.log('s.getArea is undefined');
37 }

```

Podvući pozive `console.log` metoda koji će biti pozvani i za svaki od njih napisati šta će ispisati na standardni izlaz. ■

Skladištenje zajedničkih svojstava u prototipu

Prilikom kreiranja objekata korišćenjem konstruktorske funkcije, svojstva koja pripadaju tom objektu možemo kreirati korišćenjem veze koja nam je dostupna kroz ključnu reč `this`. Međutim, ovakav način čuvanja podataka može biti neefikasan za ona svojstva koja se ne menjaju među instancama. U prethodnom primeru, tip `Shape` definisan je narednom konstruktorskom funkcijom:

```

1 function Shape() {
2     this.name = 'Shape';
3 }

```

Ovo znači da svaki put kada kreiramo novi objekat pozivom `new Shape()`, novo svojstvo `name` biva kreirano iznova i skladišti se negde u memoriji. S obzirom da će sve instance imati istu vrednost za ovo svojstvo, ovo predstavlja veliki gubitak memorije. Alternativna opcija jeste da svojstvo `name` vezemo za prototip i delimo ga među svim instancama:

```

1 function Shape() {}
2
3 Shape.prototype.name = 'Shape';

```

Na ovaj način, svaki put kada se kreira novi objekat pozivom `new Shape()`, taj objekat neće dobiti svoje svojstvo `name`, već će koristiti ono svojstvo kojim je augmentiran prototip. Ovim se dobija na efikasnosti. Naravno, ovaj metod je moguće koristiti samo za svojstva koja se ne menjaju između instanci. Zbog toga su metodi idealni kandidati za ovaj način deljenja.

Hajde da poboljšamo prethodni primer dodavanjem svih metoda i podesnih svojstava u

odgovarajuće prototipove. U slučaju tipova `Shape` i `TwoDShape`, sva svojstva su podesni kandidati za premeštanje u prototip:

```
1 // constructor
2 function Shape() {}
3
4 // augment prototype
5 Shape.prototype.name = 'Shape';
6 Shape.prototype.toString = function () {
7     return this.name;
8 };
9
10 // another constructor
11 function TwoDShape() {}
12
13 // take care of inheritance
14 TwoDShape.prototype = new Shape();
15 TwoDShape.prototype.constructor = TwoDShape;
16
17 // augment prototype
18 TwoDShape.prototype.name = '2D shape';
```

Ono što dodatno primećujemo jeste da je potrebno prvo rešiti pitanje nasleđivanja pre nego što se augmentira prototip. U suprotnom, sve što se dodaje objektu `TwoDShape.prototype` biće pregaženo prilikom nasleđivanja.

Konstruktor `Triangle` je nešto drugačiji. S obzirom da svaki objekat koji on kreira je novi trougao, koji će imati svoje dimenzije, bolje je čuvati dužinu stranice i visinu kao sopstvena svojstva, a ostalo deliti među instancama. Sa druge strane, metod `getArea` je isti, bez obzira na stvarne dimezije svakog trougla. Ponovo, prvo rešavamo nasleđivanje, a zatim augmentiramo prototip:

```
1 function Triangle(side, height) {
2     this.side = side;
3     this.height = height;
4 }
5
6 // take care of inheritance
7 Triangle.prototype = new TwoDShape();
8 Triangle.prototype.constructor = Triangle;
9
10 // augment prototype
11 Triangle.prototype.name = 'Triangle';
12 Triangle.prototype.getArea = function () {
13     return this.side * this.height / 2;
14 };
```

Prethodni kod za testiranje radi isto, što se vidi na sledećem primeru:

```
1 var my = new Triangle(5, 10);
2 console.log(my.getArea()); // 25
3 console.log(my.toString()); // 'Triangle'
```

Postoji jedna relativno mala razlika u pozadini izvršavanja prilikom pozivanja `my.toString()` između ovog i prethodnog pristupa. Razlika je u tom da postoji još jedna potraga pre nego što je metod pronađen u `Shape.prototype`, za razliku od prethodnog pristupa u kojem je metod pronađen u instanci tipa `Shape`.

Možemo se takođe poigrati pozivom metoda `hasOwnProperty` da vidimo razliku između sopstvenog svojsta i svojstva koje dolazi iz lanca prototipa:

```
1 console.log(my.hasOwnProperty('side')); // true
2 console.log(my.hasOwnProperty('name')); // false
```

Pozivi metoda `isPrototypeOf` i korišćenje operatora `instanceof` iz prethodnog primera rade na identičan način, na primer:

```
1 console.log(TwoDShape.prototype.isPrototypeOf(my)); // true
2 console.log(my instanceof Shape); // true
```

Korišćenjem ovog metoda nasleđivanja, možemo otići korak dalje i primeniti još jedno poboljšanje. Posmatrajmo naredne dve linije koda za koje smo rekli da čine "magiju" nasleđivanja:

```
1 TwoDShape.prototype = new Shape();
2 Triangle.prototype = new TwoDShape();
```

Zaista, pozivanjem funkcija `Shape` i `TwoDShape` kao konstruktora dobićemo objekat koji ima odgovarajući prototip, te ovaj kod nije pogrešan. Međutim, ovde se javlja jedan problem — vrši se poziv konstruktora pre izvršavanja dodele, što znači da ukoliko konstruktor izvršava akcije koje imaju bočne efekte, kao što je logovanje, promena stanja, korišćenje drugih objekata u neke svrhe, dodavanje svojstava vrednosti `this` i slično, sve te akcije će biti izvršene i trenutku izvršavanja dodele, umesto samo kada je potrebno (pri kreiranju objekata podtipa, kad je to očekivano). Na ovo ponašanje ćemo se još jednom kratko osvrnuti kada budemo govorili o pozivu konstruktora nadtipa radi inicijalizacije podataka.

Umesto toga, možemo se osloniti na metod `Object.create`, uvedenu standardom ES5, koji će za nas kreirati objekat sa onim prototipom koji prosledimo kao argument metoda. Tako se prethodne dve linije koda mogu zameniti narednim:

```
1 TwoDShape.prototype = Object.create(Shape.prototype);
2 Triangle.prototype = Object.create(TwoDShape.prototype);
```

Ovaj metod neće pozvati konstruktorsku funkciju, tako da neće doći do izvršavanja bočnih efekata koje konstruktorske funkcije najčešće imaju. Ipak, s obzirom da je metod `Object.create` uveden tek u verziji ES5, ukoliko je potrebno da naš kod radi i na starijim pregledačima, onda je neophodno koristiti `new Shape()` sintaksu.

Ono na šta treba posebno obratiti pažnju jeste da se na ovaj način neće pronaći svojstva koja su dodeljena u konstruktoru nadtipa, što ilustruje naredni primer:

```
1 function Nadtip() {
2     this.deljenoSvojstvo = 'Ovo bi trebalo da bude dostupno!';
3     this.deljeniMetod = function() {
4         console.log(this.deljenoSvojstvo);
5     };
6 }
7
8 Nadtip.prototype.zaistaDeljeniMetod = function() {
9     console.log('Ovo je sigurno dostupno');
10 };
11
12 function Podtip() {}
13
14 // Nasledjivanje
15 Podtip.prototype = Object.create(Nadtip.prototype);
16
17 // Augmentacija prototipa podtipa
18 Podtip.prototype.sopstveniMetod = function() {
```

```

19     console.log('Ja se nalazim u podtipu');
20 };
21
22 // Testiranje
23 var x = new Podtip();
24
25 x.sopstveniMetod(); // Ja se nalazim u podtipu
26 x.zaistaDeljeniMetod(); // Ovo je sigurno dostupno
27
28 console.log(typeof x.deljenoSvojstvo); // undefined
29 console.log(typeof x.deljeniMetod); // undefined

```

Do ovog ponašanja dolazi zato što, kao što smo rekli, metod `Object.create` ne poziva konstruktorsku funkciju nadtipa, tako da svaka inicijalizacija svojstava koja se vrši u konstruktorskoj funkciji nadtipa, neće biti vidljiva na nivou prototipa nadtipa.

Postoje dva rešenja ovog problema:

1. Izdvajanje *svih* deljivih svojstava u prototip (što i jeste osnovna ideja ovog metoda nasleđivanja), čime će oni biti dostupni, čak i ukoliko se nasleđivanje ostvaruje metodom `Object.create`.
2. Pozivom konstruktora nadtipa u konstrukturu podtipa.

Pogledajmo primer koda u kojem je neophodno da zovemo konstruktor nadtipa u konstrukturu podtipa radi inicijalizacije podataka:

```

1 function NamedShape(name) {
2     this.name = name;
3 }
4
5 NamedShape.prototype.toString = function() {
6     return this.name;
7 };
8
9 function NamedTriangle(name, side, height) {
10     this.side = side;
11     this.height = height;
12
13     // Poziv konstruktorske funkcije nadtipa
14     NamedShape.call(this, name);
15 }
16
17 NamedTriangle.prototype = Object.create(NamedShape.prototype);
18 NamedTriangle.prototype.constructor = NamedTriangle;
19
20 NamedTriangle.prototype.getArea = function() {
21     return this.side * this.height / 2;
22 };
23
24 var my = new NamedTriangle('My Triangle', 5, 10);
25 console.log(my.getArea()); // 25
26 console.log(my.toString()); // My Triangle

```

Prikažimo kako izgleda objekat prototipa tipa `NamedTriangle` kada se prototipi povezuju pozivom funkcije `Object.create`, kao u prethodnom primeru:

```

1 NamedTriangle
2 { constructor: [Function: NamedTriangle], getArea: [Function] }

```

Ukoliko bismo sada taj poziv zamenili narednim

```
1 // NamedTriangle.prototype = Object.create(NamedShape.prototype);
2 NamedTriangle.prototype = new NamedShape();
```

i prikazali tekuće stanje objekta `NamedTriangle.prototype`

```
1 NamedTriangle {
2   name: undefined,
3   constructor: [Function: NamedTriangle],
4   getArea: [Function] }
```

vidimo da ono sadrži svojstvo `name` koje je *undefined* (naravno, to je zato što konstruktoru nismo prosledili nijednu vrednost). Dakle, umesto dodatnog poziva konstruktorske funkcije za ostvarivanje nasleđivanja, preporučuje se korišćenje metoda `Object.create`, naravno, ukoliko je metod dostupan.

Zadatak 2.9 Prisetimo se narednog fragmenta koda i rezultata njegovog izračunavanja:

```
1 function Nadtip() {
2   this.deljenoSvojstvo = 'Ovo bi trebalo da bude dostupno!';
3   this.deljeniMetod = function() {
4     console.log(this.deljenoSvojstvo);
5   };
6 }
7
8 Nadtip.prototype.zaistaDeljeniMetod = function() {
9   console.log('Ovo je sigurno dostupno');
10 };
11
12 function Podtip() {}
13
14 // Nasledjivanje
15 Podtip.prototype = Object.create(Nadtip.prototype);
16
17 // Augmentacija prototipa podtipa
18 Podtip.prototype.sopstveniMetod = function() {
19   console.log('Ja se nalazim u podtipu');
20 };
21
22 // Testiranje
23 var x = new Podtip();
24
25 x.sopstveniMetod(); // Ja se nalazim u podtipu
26 x.zaistaDeljeniMetod(); // Ovo je sigurno dostupno
27
28 console.log(typeof x.deljenoSvojstvo); // undefined
29 console.log(typeof x.deljeniMetod); // undefined
```

Ispraviti kod tako da objekat `x` vidi svojstvo `deljenoSvojstvo` i metod `deljeniMetod`:

1. Metodom izdvajanja deljenih svojstava u prototip.
2. Metodom poziva konstruktora nadtipa.

Ne menjati liniju `Podtip.prototype = Object.create(Nadtip.prototype);`. ■

Nasleđivanje samo prototipa

Kao što objasnili ranije, radi postizanja efikasnosti, prototipu bi trebalo dodavati svojstva i metode koji služe za višestruku upotrebu. Ukoliko to uradimo, onda je dobra ideja naslediti samo prototip, zato što se sav višestruko upotrebljiv kod nalazi u njemu. To znači da na-

sleđivanje, na primer, objekta `Shape.prototype` je bolje od nasleđivanja objekta kreiranog pozivom `Object.create(Shape.prototype)`. Uostalom, `Object.create(Shape.prototype)` nam daje sopstvena svojstva koja nisu namenjena za višestruku upotrebu (u suprotnom bi se ona nalazila u prototipu). Zaključak je da možemo dobiti na dodatnoj efikasnosti ukoliko:

- Ne vršimo kreiranje novog objekta samo radi nasleđivanja.
- Imamo manje potraga tokom faze izvršavanja (kao što je slučaj sa potraživanjem metoda `toString`).

Pokažimo ovo na primeru ažuriranog koda u odnosu na prethodnu implementaciju:

```
1 function Shape() {}
2
3 // augment prototype
4 Shape.prototype.name = 'Shape';
5 Shape.prototype.toString = function () {
6     return this.name;
7 };
8
9 function TwoDShape() {}
10
11 // take care of inheritance
12 TwoDShape.prototype = Shape.prototype;
13 TwoDShape.prototype.constructor = TwoDShape;
14
15 // augment prototype
16 TwoDShape.prototype.name = '2D shape';
17 function Triangle(side, height) {
18     this.side = side;
19     this.height = height;
20 }
21
22 // take care of inheritance
23 Triangle.prototype = TwoDShape.prototype;
24 Triangle.prototype.constructor = Triangle;
25
26 // augment prototype
27 Triangle.prototype.name = 'Triangle';
28 Triangle.prototype.getArea = function () {
29     return this.side * this.height / 2;
30 };
```

Naredni test kod daje iste rezultate:

```
1 var my = new Triangle(5, 10);
2 console.log(my.getArea()); // 25
3 console.log(my.toString()); // 'Triangle'
```

U čemu je razlika prilikom pozivanja `my.toString()`? Prvo, po običaju, JavaScript mašina pretražuje metod `toString` u samom objektu. Kako ga ne pronalazi, onda pomera pretragu u prototipu. Međutim, sada prototip pokazuje na isti objekat na koji prototip tipa `TwoDShape` pokazuje i takođe na isti objekat na koji `Shape.prototype` pokazuje. Prisetimo se da se objekti ne kopiraju po vrednosti, već se šalje njihova referenca¹⁰. Dakle, potraga se sada sastoji od dva koraka umesto od četiri (kao u prethodnoj implementaciji) ili od tri (kao u prvoj implementaciji).

¹⁰Opet, njihova referenca se kopira po vrednosti.

Jednostavno kopiranje prototipa je efikasnije, ali boluje od jednog bočnog efekta. S obzirom da sada svi prototipovi podtipova i nadtipova pokazuju na isti objekat, kada neki podtip augmentira prototip, svi njegovi nadtipovi, a samim tim i svi ostali podtipovi vide izmenu.

Razmotrimo efekat izvršavanja naredne linije koda:

```
1 Triangle.prototype.name = 'Triangle';
```

Ova linija menja svojstvo `name`, tako da efektivno menja i `Shape.prototype.name`. Ukoliko kreiramo instancu tipa `Shape`, njegovo svojstvo `name` će nam vratiti vrednost `'Triangle'`:

```
1 var s = new Shape();
2 console.log(s.name);    // 'Triangle'
```

Ovaj metod nasleđivanja je više efikasan, ali zbog opisanog bočnog efekta, nije prikladan za sve slučajeve upotrebe. Ipak, ovu "manu" ne treba posmatrati kao propust (kao što neka literatura to čini), već kao dodatni mehanizam koji može biti koristan u nekim slučajevima. Naravno, na programerima je odgovornost da razumeju upotrebu, kao i potencijalne probleme koji se mogu pojaviti u slučajevima kada nasleđivanje samo prototipa nije adekvatno.

Metod privremenog konstruktora

Rešenje prethodno opisanog "problema" u kojem svi prototipovi referišu na isti objekat i nadtipovi dobijaju svojstva podtipova predstavlja korišćenje *posrednika* (engl. *intermediary*) radi razbijanja lanca. Posrednik je u formi privremene konstruktorske funkcije. Kreiranje prazne funkcije `F()` i postavljanje njegovog prototipa na prototip konstruktora nadtipa dozvoljava nam da pozovemo `new F()` i kreiramo objekte koji nemaju sopstvena svojstva, ali nasleđuju sve iz prototipa nadtipa.

Razmotrimo narednu izmenjenu implementaciju:

```
1 function Shape() {}
2
3 // augment prototype
4 Shape.prototype.name = 'Shape';
5 Shape.prototype.toString = function () {
6     return this.name;
7 };
8
9 function TwoDShape() {}
10
11 // take care of inheritance
12 var F = function () {};
13 F.prototype = Shape.prototype;
14 TwoDShape.prototype = new F();
15 TwoDShape.prototype.constructor = TwoDShape;
16
17 // augment prototype
18 TwoDShape.prototype.name = '2D shape';
19
20 function Triangle(side, height) {
21     this.side = side;
22     this.height = height;
23 }
24
25 // take care of inheritance
26 var F = function () {};
27 F.prototype = TwoDShape.prototype;
28 Triangle.prototype = new F();
```

```

29 Triangle.prototype.constructor = Triangle;
30
31 // augment prototype
32 Triangle.prototype.name = 'Triangle';
33 Triangle.prototype.getArea = function () {
34     return this.side * this.height / 2;
35 };

```

Kreiranje trougla i testiranje metoda:

```

1 var my = new Triangle(5, 10);
2 console.log(my.getArea()); // 25
3 console.log(my.toString()); // 'Triangle'

```

Korišćenjem ovog pristupa, lanac prototipa ostaje na mestu:

```

1 console.log(my.__proto__ === Triangle.prototype); // true
2 console.log(my.__proto__.constructor === Triangle); // true
3 console.log(my.__proto__.__proto__ === TwoDShape.prototype); // true
4 console.log(my.__proto__.__proto__.__proto__.constructor === Shape); // true

```

Dodatno, svojstva nadtipova nisu pregažena od strane podtipova:

```

1 var s = new Shape();
2 console.log(s.name); // 'Shape'
3 console.log('I am a ' + new TwoDShape()); // 'I am a 2D shape'

```

Takođe, ovaj pristup podržava ideju da samo svojstva i metodi koji se dodaju u prototipu treba da se naslede, a da se sopstvena svojstva ne nasleđuju. Razlog iza ove ideje je taj da su sopstvena svojstva uglavnom dovoljno specifična da ne mogu biti upotrebljena za višestruku upotrebu.

Primitimo da smo ovde koristili `new F()` umesto `Object.create(F.prototype)`, iako smo govorili o problemima tog pristupa. Međutim, kako se u ovom metodu nasleđivanja podrazumeva da konstruktorska funkcija `F` ima prazno telo, oba pristupa su prikladna.

Zadatak 2.10 Implementirati narednu hijerarhiju tipova korišćenjem stečenog znanja o prototipovima i nasleđivanju:

- Kreirati naredne tipove sa odgovarajućim svojstvima:
 - `Person`: `name`, `surname`
 - `Employee`: nasleđuje `Person` i ima `job` i `salary`. Plata ne sme biti dostupna spoljnom svetu.
 - `Developer`: nasleđuje `Employee` i ima `specialization`
 - `Manager`: nasleđuje `Employee` i ima `department`
- Sve instance `Person` imaju naredni metod:
 - `getData` koji ispisuje ime i prezime
- Sve instance `Employee` imaju naredne metode:
 - `getSalary` koji ispisuje platu
 - `increaseSalary` koji uvećava platu zaposlenog za 10%
- Sve instance `Developer` imaju naredni metod:
 - `getSpecialization` koji ispisuje naziv specijalizacije
- Sve instance `Manager` imaju naredne metode:
 - `getDepartment` koji ispisuje naziv odeljenja
 - `changeDepartment` koji menja naziv odeljenja

- `getData` koji ispisuje ime, prezime i naziv odeljenja

Ispravno implementirati učauravanje svojstava i metoda. Omogućiti da kreiranje instanci datih tipova narednim fragmentom koda ne proizvodi greške pod striktnim tipom:

```
1 var laza = Person('Laza', 'Lazic');
2 var mara = Employee('Mara', 'Maric', 'teacher', 300);
3 var pera = Developer('Pera', 'Peric', 'JS programmer', 1000, 'Front-End');
4 var koja = Manager('Koja', 'Kojic', 'manager', 1000, 'D1')
```

Omogućiti da naredni fragment koda proizvodi odgovarajuće ispise date u komentarima

```
1 laza.getData();           // Laza Lazic
2
3 mara.getSalary();         // 300
4 mara.increaseSalary();
5 mara.getSalary();         // 330
6
7 pera.getData();           // Pera Peric
8 pera.getSpecialization(); // Front-End
9 pera.getSalary();         // 1000
10
11 koja.getData();           // Koja Kojic D1
12 koja.changeDepartment('D3');
13 koja.getDepartment();    // D3
14 koja.getData();           // Koja Kojic D3
```

2.5.6 Klase i objektno-orijentisana podrška u ECMAScript 6

U novijim verzijama JavaScript jezika, postoji pristojnija notacija za definisanje klasa:

```
1 class Zec {
2     // Konstruktor funkcija
3     constructor(tip) {
4         this.tip = tip;
5     }
6
7     govori(recenica) {
8         console.log(`Ovaj ${this.tip} zec kaze: '${recenica}'`);
9     }
10 }
11
12 let tuzanZec = new Zec('tuzan');
13 let crniZec = new Zec('crni');
```

Ono što je potrebno razumeti jeste da klasna deklaracija u JavaScript jeziku predstavlja ništa drugo do sintaksno poboljšanje — u pozadini se izvršava prototipsko nasleđivanje o kojem smo pričali u prethodnoj podsekciji. Da dokažemo ovu tvrdnju, ispitajmo tip "klase" `Zec`:

```
1 console.log(typeof Zec);    // function
```

Trenutno, deklaracije klasa dozvoljavaju samo metode da se dodaju prototipu. Nefunkcijska svojstva se mogu dodavati direktnom izmenom prototipa nakon što se definiše klasa.

Dodatno, iako su klase u pozadini zapravo funkcije, JavaScript jezik neće izvršavati izdizanje deklaracija klasa, kao što to radi sa funkcijama. Dakle, klase su dostupne tek od trenutka njihovog definisanja.

Metod constructor

U prethodnom primeru smo koristili jednu specijalnu funkciju koja se naziva **constructor**. U pitanju je metod koji se koristi za kreiranje i inijalizaciju objekta prilikom instanciranja objekta klase. Klasa može imati samo jedan metod **constructor**.

Jedna od prednosti korišćenja klase nad konstruktorskim funkcijama jeste da se u okviru metoda **constructor** može pozivati metod **super**, koji se odnosi na konstruktor natklase. O ovom metodu ćemo detaljnije pričati kada se budemo dotakli nasleđivanja.

Očitavanje i postavljanje svojstava. Statička svojstva.

Neke klase se često sastoje samo od metoda, ali je sasvim korektno uključiti svojstva koja ne sadrže funkcije kao vrednosti. Na primer, postoji klasa **Map** koja implementira strukturu podataka rečnik, a koja ima svojstvo **size** koje nosi informaciju o broju ključeva koje se sadrže u rečniku. Očigledno, nije neophodno (a najčešće ni poželjno) da ovakva svojstva se računaju i čuvaju u instancama klase direktno. Čak i svojstva koja se dohvataju direktno mogu da kriju poziv metoda. Takvi metodi se nazivaju *očitači* (engl. *getter*), i oni se definišu postavljanjem ključne reči **get** ispred naziva metoda u literali objekta ili deklaraciji klase:

```
1 let promenljivaVelicina = {
2   get velicina() {
3     return Math.floor(Math.random() * 100);
4   }
5 };
6
7 console.log(promenljivaVelicina.velicina); // 73
8 console.log(promenljivaVelicina.velicina); // 49
```

Kada god se očitava vrednost svojstva **velicina** ovog objekta, odgovarajući metod se poziva. Možemo koristiti sličan metod i kada se menja vrednost svojstvu, korišćenjem *postavljača* (engl. *setter*), koji se definišu ključnom reči **set**:

```
1 class Temperatura {
2   constructor(celzijus) {
3     this.celzijus = celzijus;
4   }
5
6   get farenhajt() {
7     return this.celzijus * 1.8 + 32;
8   }
9
10  set farenhajt(vrednost) {
11    this.celzijus = (vrednost - 32) / 1.8;
12  }
13
14  static izFarenhajta(vrednost) {
15    return new Temperatura((vrednost - 32) / 1.8);
16  }
17 }
```

Klasa **Temperatura** dozvoljava očitavanje i postavljanje vrednosti za temperaturu bilo u celzijusima ili farenhajtima, ali interno čuva samo celzijuse i automatski vrši konverziju u tu jedinicu prilikom očitavača i postavljača **farenhajt**:

```
1 let temp = new Temperatura(32);
2 console.log(temp.farenhajt); // 71.6
3 temp.farenhajt = 86;
4 console.log(temp.celzijus); // 30
```

Nekada želimo da dodamo svojstva direktno konstruktoru umesto prototipu. Takvi metodi nemaju pristup instancama klase, ali mogu, na primer, biti iskorišćeni kao alternativni načini za kreiranje instanci. Unutar deklaracije klase, metodi koji imaju ključnu reč `static` ispred naziva se čuvaju na nivou konstruktora. Zbog toga, klasa `Temperatura` dozvoljava da napišemo izraz `Temperatura.izFarenhajta(100)` da kreiramo temperaturu korišćenjem jedinice farenhajt (umesto celzijusa, što je podrazumevano).

Zadatak 2.11 Napisati klasu `Vec` koja predstavlja dvodimenzionalni vektor. Klasa ima dva svojstva, `x` i `y` (brojevi). Klasa ima dva metoda, `plus` i `minus`, koji imaju drugi vektor kao parametar i vraćaju novi vektor koji predstavlja zbir, odnosno, razliku "this" vektora i prosleđenog vektora. Dodati postavljач `duzina` koja izračunava udaljenost tačke (x,y) od koordinatnog početka. ■

Nasleđivanje

Pretpostavimo da nam je na raspolaganju naredna klasa `Osoba`:

```
1 class Osoba {
2     constructor(ime, prezime, godine) {
3         this.ime = ime;
4         this.prezime = prezime;
5         this.godine = godine;
6     }
7
8     identifikujSe() {
9         console.log(`Zdravo! Moje ime je ${this.ime} ${this.prezime}. Imam ${
10             this.godine} godina.`);
11     }
12 }
```

Ukoliko bismo želeli da kreiramo klasu `Student` kojeg želimo da identifikujemo na sličan način kao instancu klase `Osoba`, mogli bismo pisati klasu ispočetka, ali tada ćemo imati mnogo koda koji se ponavlja. Umesto toga, možemo da iskoristimo postojeću klasu `Osoba`. JavaScript sistem prototipova omogućava kreiranje nove klase od neke postojeće, koja izgleda kao postojeća klasa, uz nove definicije nekih od svojstava. Prototip nove klase se izvodi iz prototipa postojeće klase, ali dodaje nove definicije. U kontekstu objektno-orijentisane paradigme, ovo se naziva *nasleđivanje* (engl. *inheritance*). Kažemo da nova klasa nasleđuje svojstva i ponašanje iz postojeće klase:

```
1 class Student extends Osoba {
2     constructor(ime, prezime, godine, fakultet) {
3         super(ime, prezime, godine);
4         this.fakultet = fakultet;
5     }
6
7     identifikujSe() {
8         super.identifikujSe();
9         console.log(`Ja sam student ${this.fakultet}.`);
10    }
11 }
```

Korišćenjem ključne reči `extends` indikujemo da se nova klasa bazira na već postojećoj klasi umesto na prototipu `Object`. Postojeća klasa se naziva *natklasa* (engl. *superclass*), a izvedena klasa se naziva *potklasa* (engl. *subclass*).

Za inicijalizaciju instance klase `Student`, konstruktor poziva konstruktor iz natklase pomoću ključne reči `super`. Dodatno, konstruktor potklase obavlja još neke radnje (u ovom

slučaju, inicijalizuje dodatno svojstvo koje imaju samo instance klase `Student`). Slično, metod `identifikujSe` poziva istoimeni metod iz natklase, jer hoćemo da zadržimo originalno ponašanje, uz dodatne izmene. Zbog toga ponovo koristimo ključnu reč `super`, da bismo pozvali ispravan metod. Primer izvršavanja je dat narednim kodom:

```
1 let osoba = new Osoba('Pera', 'Peric', 24);
2 osoba.identifikujSe();
3 // Zdravo! Moje ime je Pera Peric. Imam 24 godina.
4
5 let student = new Student('Ana', 'Jovanovic', 20, 'MATF');
6 student.identifikujSe();
7 // Zdravo! Moje ime je Ana Jovanovic. Imam 20 godina.
8 // Ja sam student MATF.
```

Napomenimo da je moguće implementirati referencu ka nadtipu i korišćenjem konstruktorskih funkcija i lanca prototipova sa efektom poput ključne reči `super` u ECMAScript 5 standardu, ali to nećemo prikazivati.

Operator `instanceof`

Kao što znamo, često je korisno znati da li je neki objekat izveden iz specifične klase. JavaScript jezik definiše infiksni operator `instanceof`, koji primenjen nad objektom i klasom vraća `true` akko je objekat instanca date klase:

```
1 console.log(student instanceof Student); // true
2 console.log(student instanceof Osoba);   // true
3 console.log(osoba instanceof Student);   // false
4 console.log([1] instanceof Array);       // true
```

Operator `instanceof` razume koncept nasleđivanja, tako da će uspeti da prepozna da je objekat `student` instanca klase `Osoba`, naravno, indirektna instanca te klase.

Ulančano nasleđivanje

Još jedna tehnika koju je moguće implementirati u ES5 standardu, ali čija je implementacija mnogo jednostavnija u ES6 standardu jeste *višestruko nasleđivanje* (engl. *multiple inheritance*). JavaScript jezik u svom jezgru podržava isključivo jednostruko nasleđivanje. Međutim, korišćenjem nekih od opisanih tehnika moguće je svesti koncept višestrukog nasleđivanja na *ulančano nasleđivanje*.

Koncept višestrukog nasleđivanja u jezicima u kojima je ono podržano u najvećem broju slučajeva podrazumeva da se, pored izvođenja potklase iz bazne klase koja implementira glavnu logiku, potklasa dopuni dodatnim implementacijama koje ne postoje u baznoj klasi. Na primer, neka je data naredna jednostavna hijerarhija klasa:

```
1 class Osoba {}
2 class Zaposleni extends Osoba {}
```

Neka je za klasu `Zaposleni` potrebno implementirati metod za proveru istorije zaposlenosti i metod za pristupanje. Očigledno, jedan način da ovo uradimo jeste da te metode implementiramo kao metode klase `Zaposleni`. Međutim, ukoliko su nam ove metode neophodne i u drugim klasama, njihova ponovna implementacija dovodi do umnožavanja koda. Zbog toga, bilo bi dobro smestiti metode u zasebne klase i izvršiti višestruko nasleđivanje (ovime bi problem bio rešen u, na primer, C++ programskom jeziku):

```
1 class ProveraIstorije {
2   prover() {}
3 }
4
```

```

5 class PristupanjeSistemu {
6     izdajKarticu() {}
7 }

```

Međutim, kao što smo rekli, višestruko nasleđivanje nije moguće u JavaScript jeziku. Umešto toga, moguće je implementirati klase `ProveraIstorije` i `PristupanjeSistemu` (koje se u literaturi nazivaju *umešavači* (engl. *mixin*)) kao funkcije koje kao parametar imaju natklasu, a kao povratnu vrednost imaju potklasu izvedenu iz te natklase:

```

1 class Osoba {}
2
3 const ProveraIstorije = Alati => class extends Alati {
4     prover() {}
5 };
6
7 const PristupanjeSistemu = Alati => class extends Alati {
8     izdajKarticu() {}
9 };
10
11 class Zaposleni extends ProveraIstorije(PristupanjeSistemu(Osoba)) {}

```

Ovim se praktično ostvaruje da je `Zaposleni` potklasa `ProveraIstorije`, što je potklasa `PristupanjeSistemu`, što je potklasa `Osoba`.

2.6 Obrada grešaka

Kao što smo videli do sada, JavaScript jezik je poprilično labilan u smislu da je u stanju da izračuna razne izraze i programske fragmente bez da se žali, iako možda rezultat nije nešto što bismo očekivali. Ovakvih situacija ima mnogo, te je potrebno upoznati se sa sistemom za kontrolu i obradu grešaka, kada do njih dođe.

2.6.1 Striktni režim rada

Striktni režim rada je uveden standardom ECMAScript5 i on podrazumeva postroživanje izvršnog konteksta JavaScript okruženja na nivou celog skripta ili pojedinačnih funkcija. Ovaj režim rada obuhvata više različitih ponašanja okruženja, a sva ona se mogu svrstati u naredne tri kategorije:

1. Neke tradicionalne tihe greške se sada prijavljuju od strane JavaScript okruženja.
2. Ispravlja se greške koje onemogućavaju JavaScript mašinu da izvede optimizacije.
3. Zabranjuje korišćenje određenog dela JavaScript sintakse.

Omogućivanje izvršavanja JavaScript okruženja se, kao što smo rekli, može postići na nivou celog skripta ili na nivou pojedinačne funkcije. U prvom slučaju, dovoljno je napisati naredbu `'use strict'`; na početku skripta. U drugom slučaju, ista naredba se navodi kao prva linija tela funkcije. Ovo je ilustrovano na slici 2.5.

<pre> 1 'use strict'; 2 3 function mojaGlobalnaFunkcija () {} </pre>	<pre> 1 function mojaGlobalnaFunkcija () { 2 'use strict'; 3 4 // Ostatak tela funkcije 5 } </pre>
--	--

Slika 2.5: Ilustracija omogućavanja striktnog režima rada JavaScript okruženja na nivou celog skripta (levo) i na nivou funkcije (desno).

JavaScript skriptove je moguće nadovezivati. Ukoliko nadovežemo skript koji radi u striktnom režimu i skript koji ne radi u striktnom režimu, onda će celo nadovezivanje biti striktno! Slično, nadovezivanje nestriktnog i striktnog skripta proizvodi nestriktno ponašanje. Zbog toga treba biti oprezan sa omogućavanjem striktnog režima rada na nivou celog skripta.

Pre nego što predstavimo razlike u ponašanju između striktnog i nestriktnog režima rada, napomenimo da će u tekstu biti predstavljen tek pravi podskup izmena ponašanja koje se uvode omogućavanjem striktnog režima rada.

Prijavljivanje tradicionalno tihih grešaka

Tihe greške (engl. *silent error*) predstavljaju one greške koje dovode do nekorektnog ponašanja u odnosu na očekivani tok izvršavanja, ali ne rezultuju prijavljivanjem greške od strane JavaScript okruženja, već se okruženje sa ovakvim greškama izборi na odgovarajući način.

Jedna od semantičkih grešaka koje programeri mogu napraviti jeste ukoliko pogrešno spe-luju naziv promenljive, misleći da koriste ispravno imenovanje. Primer na slici 2.6 ilustruje razliku između nestriktnog i striktnog ponašanja u ovom slučaju. U nestriktnom režimu, kreira se nova globalna promenljiva (levo), dok se u striktnom režimu prijavljuje greška tipa `ReferenceError` (desno).

<pre>1 // 'use strict'; 2 3 var myVar = 1; 4 5 // Pogresno spelovanje kreira 6 // novu globalnu promenljivu 7 myVr = 2; 8 9 10 console.log(myVar); // 1 11 console.log(myVr); // 2</pre>	<pre>1 'use strict'; 2 3 var myVar = 1; 4 5 // ReferenceError: 6 // myVr is not defined 7 myVr = 2; 8 9 // Nedostizan deo koda 10 console.log(myVar); 11 console.log(myVr);</pre>
--	---

Slika 2.6: Ilustracija slučajnog kreiranja globalne promenljive u nestriktnom režimu (levo), i sprečavanje ovog ponašanja greškom u striktnom režimu rada (desno).

Dodeljivanje vrednosti konstruktima koje sadrže imutabilne vrednosti, kao što su `NaN`, `undefined` ili `Infinity`, rezultuju greškom tipa `TypeError` u striktnom režimu rada, što ilustruje slika 2.7.

<pre>1 NaN = 7; 2 undefined = 7; 3 Infinity = 7; 4 5 // Vrednosti nisu izmenjene, 6 // ali kod prolazi</pre>	<pre>1 'use strict'; 2 3 // Svaka od dodela ispod 4 // proizvodi TypeError 5 NaN = 7; 6 undefined = 7; 7 Infinity = 7;</pre>
--	--

Slika 2.7: Ilustracija dodeljivanja vrednosti specijalnim konstruktima jezika bez pojave greške (levo) i sa pojavom greške u striktnom režimu rada (desno).

Brisanje nebrišućih svojstava objekata u striktnom režimu proizvodi grešku tipa `TypeError`,

što ilustruje slika 2.8.

```

1 delete Object.prototype;
2
3 // Nije obrisano,
4 // ali ne proizvodi gresku.
5
6 // true
7 console.log(Object.prototype
8     === Object.getPrototypeOf({}));

```

```

1 'use strict';
2
3 // TypeError
4 delete Object.prototype;

```

Slika 2.8: Ilustracija brisanja nebrišućeg svojstva objekta bez pojave greške (levo) i sa pojavom greške u striktnom režimu rada (desno).

Uvođenje dva svojstava koji imaju isti naziv rezultuje greškom tipa `SyntaxError`. U ECMAScript6 standardu, ovo je oslabljeno i dozvoljava se da objekat ima dva svojstava istog naziva, pri čemu će vrednost tog svojstva biti poslednja dodeljena vrednost tom svojstvu. Primer na slici 2.9 ilustruje ovo ponašanje.

```

1 var obj = {
2     test: 'Test 1',
3     test: 2
4 };
5
6 console.log(obj.test); // 2

```

```

1 var obj = {
2     test: 'Test 1',
3     test: 2
4 };
5
6 // ES5: SyntaxError
7 // ES6: 2
8 console.log(obj.test);

```

Slika 2.9: Ilustracija navođenja više od jednog svojstva sa istim imenom u istom objektu bez pojave greške (levo) i sa pojavom greške u striktnom režimu rada u ECMAScript5 standardu (desno).

Ukoliko deklariramo da više od jednog parametra funkcije ima isti naziv, onda će poslednji parametar sakriti vrednosti ostalih parametara. U striktnom režimu ovo nije dozvoljeno ponašanje, i rezultuje greškom tipa `SyntaxError`, što ilustruje slika 2.10.

```

1 function sum(a, a, c) {
2     return a + a + c;
3 }
4
5 console.log(sum(1, 2, 3)); // 7

```

```

1 function sum(a, a, c) {
2     // ^^^^
3     // SyntaxError
4     'use strict';
5     return a + a + c;
6 }
7
8 console.log(sum(1, 2, 3));

```

Slika 2.10: Ilustracija navođenja više od jednog parametra sa istim imenom u listi parametara funkcije bez pojave greške (levo) i sa pojavom greške u striktnom režimu rada (desno).

Još jedna promena u striktnom modu je da vrednost `this` ima vrednost `undefined` u funkcijama koje nisu pozvane kao metodi. Kada takav poziv obavljamo izvan striktnog moda,

`this` referiše na globalni objekat, što je objekat čija su svojstva globalno vezana. Tako da ako slučajno pozovemo metod ili konstruktor neispravno u striktnom modu, JavaScript će proizvesti grešku tipa `TypeError` čim pokuša da pročita nešto iz `this`, umesto da bez ikakvih problema piše po globalnom prostoru. Primer na slici 2.11 zove konstruktor funkciju bez ključne reči `new`, odakle sledi da `this` u toj funkciji neće referisati na novokreirani objekat. Na sreću, konstruktori kreirani sa `class` notacijom će se uvek žaliti ako zaboravimo ključnu reč `new`.

<pre> 1 function Osoba(ime) { 2 this.ime = ime; 3 } 4 5 let ana = Osoba('Ana'); 6 console.log(ime); // Ana </pre>	<pre> 1 'use strict'; 2 3 function Osoba(ime) { 4 this.ime = ime; 5 // ^^^^^^^ 6 // TypeError 7 } 8 9 let ana = Osoba('Ana'); </pre>
---	--

Slika 2.11: Ilustracija navođenja više od jednog parametra sa istim imenom u listi parametara funkcije bez pojave greške (levo) i sa pojavom greške u striktnom režimu rada (desno).

Onemogućavanje optimizacije

Striktni režim pojednostavljuje kako se identifikatori promenljivih mapiraju odgovarajućim definicijama promenljivih u kodu. Mnoge optimizacije kompilatora se oslanjaju na mogućnost da se "promenljiva `x` nalazi na *toj* lokaciji". Ovo jednostavno ponašanje u JavaScript jeziku može biti nemoguće za određivanje sve do faze izvršavanja. U tom duhu, predstavimo naredna dva poboljšanja koja uvodi striktni režim rada.

Jedna zanimljiva JavaScript funkcija jeste `eval`, čiji je efekat proizvodnja koda i njegovo izračunavanje u fazi izvršavanja programa. Ovoj funkciji možemo proslediti nisku koja sadrži JavaScript kod, i ona će taj fragment koda izvršiti. S obzirom na veliki broj mogućnosti koje ova funkcija nudi, bilo one smatrane za dobre ili loše, mi se nećemo upuštati u analizu njenog ponašanja, već ćemo navesti jedno od njih, koje je ilustrovano na slici 2.12. Ukoliko radimo u nestriktnom režimu rada, funkcija `eval` može kreirati promenljive u širem opsegu i time vršiti zagađivanje globalnog okruženja. Međutim, u striktnom režimu rada, sve promenljive koje se uvode u kodu su lokalne za kontekst izvršavanja koda koji se prosleđuje kao niska-argument funkcije `eval`.

<pre> 1 eval("var x = 2;"); 2 console.log(x); // 2 </pre>	<pre> 1 'use strict'; 2 3 eval("var x = 2;"); 4 5 // ReferenceError: 6 // x is not defined 7 console.log(x); </pre>
---	---

Slika 2.12: Ilustracija poziva funkcije `eval` koja u nestriktnom režimu kreira globalne promenljive (levo), dok u striktnom režimu se promenljive kreiraju lokalno za kontekst izvršavanja koda koji se prosleđuje kao argument funkcije (desno).

Druga stvar o kojoj ćemo diskutovati je konstrukt `with`. U striktnom režimu rada, ovaj

konstrukt nije uopšte dostupan za upotrebu. Njegova sintaksa je:

```
1 with (expression) {  
2     statements  
3 }
```

Osnovna upotrebna vrednost `with` konstrukta jeste uvođenje izraza `expression` u domet pretrage vrednosti prilikom izračunavanja `statements`. Da bismo pojasnili ovo ponašanje, daćemo jedan primer. U JavaScript-u su nam standardne matematičke funkcije i konstante dostupne kroz tip `Math`, na primer, `Math.sin` ili `Math.PI`. Naredni primer ilustruje korišćenje ovih funkcija:

```
1 var a, x, y;  
2 var r = 10;  
3  
4 a = Math.PI * r * r;  
5 x = r * Math.cos(Math.PI);  
6 y = r * Math.sin(Math.PI / 2);
```

Pomoću `with` konstrukta, možemo izbeći pisanje `Math` tako što ćemo ga uvesti u domet pretrage vrednosti, na sledeći način:

```
1 var a, x, y;  
2 var r = 10;  
3  
4 with (Math) {  
5     a = PI * r * r;  
6     x = r * cos(PI);  
7     y = r * sin(PI / 2);  
8 }
```

Problem sa konstruktom `with` jeste da bilo koje ime unutar njegovog bloka može da se mapira bilo na svojstvo objekta `expression` koji mu se prosleđuje ili da veže promenljivu u okolnom (ili globalnom) dometu, tokom faze izvršavanja. Zbog toga je nemoguće ovo odrediti pre te faze, što dovodi do toga da kompilator ne može da vrši optimizacije. U striktnom modu, korišćenje `with` konstrukta rezultuje greškom tipa `SyntaxError`, čime se sprečava ovo dvosmisleno ponašanje, što ilustruje slika 2.13.

<pre>1 var x = 17; 2 var obj = {}; 3 4 if (Math.random() < 0.5) { 5 obj.x = 42; 6 } 7 8 with (obj) { 9 // Poznato je tek 10 // u fazi izvršavanja 11 console.log(x); // ??? 12 }</pre>	<pre>1 'use strict'; 2 3 var x = 17; 4 var obj = {}; 5 6 if (Math.random() < 0.5) { 7 obj.x = 42; 8 } 9 10 with (obj) { // SyntaxError 11 console.log(x); 12 }</pre>
---	---

Slika 2.13: Ilustracija korišćenja `with` konstrukta koji dovodi do dvosmislenog ponašanja u nestriktnom režimu rada — vrednost `x` unutar bloka može biti bilo 17 ili 42, te kompilator ne može da zaključi o kojoj promenljivoj je reč (levo). U striktnom režimu rada, `with` konstrukt nije dozvoljen.

Zabranjivanje određenih delova JavaScript sintakse

Budući ECMAScript standardi će najverovatnije uvesti novu sintaksu, i pod striktnim režimom rada u ECMAScript5 standardu se olakšava tranzicija koda tako što se neke ključne reči ne mogu koristiti, kao što su, na primer, `implements`, `interface`, `let`, `package`, `private`, `protected`, `public`, `static` i `yield`. Primer na slici 2.14 ilustruje ovo ponašanje.

<pre> 1 function package(protected) { 2 var implements; 3 4 interface: 5 while (true) { 6 break interface; 7 } 8 9 function private() {} 10 } 11 12 function fun(static) {} 13 14 package(); 15 fun(); 16 17 // Pokretanje se završava 18 // bez gresaka </pre>	<pre> 1 function package(protected) { 2 // ^^^^^^ ^^^^^^^ 3 // SyntaxError 4 'use strict'; 5 6 var implements; 7 // ^^^^^^^ 8 // SyntaxError 9 10 interface: 11 // ^^^^^^^ 12 // SyntaxError 13 14 while (true) { 15 break interface; 16 // ^^^^^^^ 17 // SyntaxError 18 } 19 20 function private() {} 21 // ^^^^^^ 22 // SyntaxError 23 } 24 25 function fun(static) { 'use strict' 26 ; } 27 // ^^^^^^ 28 // SyntaxError 29 30 package(); 31 fun(); </pre>
---	---

Slika 2.14: Ilustracija korišćenja ključnih reči rezervisanih za naredne ECMAScript standarde bez prijavljivanja grešaka u nestruktnom režimu (levo) i sa greškama u striktnom režimu rada (desno).

Zadatak 2.12 Precizno podvucite deo koda koji će dovesti do greške u narednom fragmentu koda:

```

1 function mozeteLiPronaciProblem() {
2     'use strict';
3     for (counter = 0; counter < 10; counter++) {
4         console.log('99 little bugs in the code');
5     }
6 }
7
8 mozeteLiPronaciProblem();

```

2.6.2 Rad sa izuzecima

Ne mogu svi problemi da budu sprečeni od strane programera. Ako program komunicira sa spoljašnjim svetom u bilo kom smislu, moguće je da će dobiti ulaz koji nije ispravan, da bude preopterećen poslom ili da dođe do problema na mreži u toku njegovog rada.

Neka imamo funkciju `pitajZaBroj` koja pita korisnika za celi broj i vraća ga. Šta bi se desilo ako korisnik unese "Igra prestola"? Jedna opcija bi bila da funkcija vrati specijalnu vrednost, na primer, `null`, `undefined` ili `-1`.

```
1 function pitajZaBroj(pitanje) {
2     let rezultat = Number(prompt(pitanje));
3     if (Number.isNaN(rezultat)) {
4         return null;
5     }
6     return rezultat;
7 }
8
9 console.log(pitajZaBroj('Koliko Vesteros kuca znate da nabrojite?'));
```

Funkcija `prompt` u veb pregledaču prikazuje prozor u kojem korisnik može da unese tekst. Sada bilo koji kod koji zove funkciju `pitajZaBroj` mora da proverava da li je zapravo pročitani broj `i`, ukoliko to nije, nekako da pokuša da se oporavi. Ovakav pristup ima mnogobrojne probleme — Šta ako funkcija može da vrati bilo koju vrednost (na primer, funkcija koja vraća poslednji element niza)? Da li to znači da svaki put kada se poziva funkcija, ona mora iznova da ima provere? Da li to podrazumeva da i funkcije koje zovu funkciju koja zove problematičnu funkciju moraju da imaju iste provere?

Kada funkcija ne može da nastavi sa regularnim tokom, ono što bismo želeli jeste da prekinemo taj tok i da "skočimo" na deo koda koji zna kako da upravlja problemom. U programskim jezicima visokog nivoa, ovaj koncept se naziva *rad za izuzecima* (engl. *exception handling*). Izuzeci su mehanizam koji čine mogućim da kod koji naiđe na problem *podigne* (engl. *raise*) (ili *ispali* (engl. *throw*)) izuzetak. Izuzetak može biti proizvoljna vrednost. Ispaljivanjem izuzetka se iskače ne samo iz trenutke funkcije, već iz svih funkcija koje je zovu, sve do prvog poziva koji je započeo trenutni proces. Ovo se naziva *odmotavanje steka* (engl. *unwinding the stack*).

Očigledno, da je jedina uloga izuzetaka samo odmotavanje steka do prvog poziva, to bi značilo da oni predstavljaju glorifikovani način za eksploziranje naših programa. Njihova prava moć leži u tome da možemo da određena mesta u steku *uhvatiti* (engl. *catch*) izuzetak prilikom odmotavanja steka. Jednom kada smo uhvatili izuzetak, možemo da uradimo proizvoljnu akciju sa njim čime želimo da pokušamo da rešimo problem i da nastavimo sa izvršavanjem problema.

Pogledajmo naredni primer:

```
1 function pitajZaSmer(pitanje) {
2     let rezultat = prompt(pitanje);
3     if (rezultat.toLowerCase() == 'levo') {
4         return 'L';
5     }
6     else if (rezultat.toLowerCase() == 'desno') {
7         return 'D';
8     }
9     throw new Error('Nemoguc smer: ' + rezultat);
10 }
11
12 function pogledaj() {
```

```
13     if (pitajZaSmer('U kom smeru da pogledam?') == 'L') {
14         return 'kucu od slatkisa';
15     }
16     return 'dva gladna medveda';
17 }
18
19 try {
20     console.log('Vidim', pogledaj());
21 }
22 catch (error) {
23     console.log('Nesto je poslo naopako: ' + error);
24 }
```

Ključnom reči `throw` se ispaljuje izuzetak. Hvatanje izuzetka se vrši tako što se: (1) kod koji može da ispali izuzetak smesti u `try`-blok i (2) na taj kod se nadoveže `catch`-blok. Kada kod u `try`-bloku izazove ispaljivanje izuzetka, `catch`-blok se izračunava, a za promenljivu u zagradama se veže vrednost izuzetka. Nakon što se ili `catch`-blok završi ili se `try`-blok završi bez ispaljivanja izuzetka¹¹, program nastavlja sa izvršavanjem nakon celog `try/catch` izraza.

U ovom primeru smo izuzetak napravili korišćenjem konstruktora `Error`. Ovo je standardan konstruktor jezika JavaScript koji konstruiše objekat sa svojstvom `message`. U većini JavaScript okruženja, instance konstruktora takođe sakupljaju informaciju o steku koji je odmotan ispaljivanjem te instance. Ova informacija se čuva u svojstvu `stack` i može biti korisna za debugovanje.

Primetimo da funkcija `pogledaj` potpuno ignoriše mogućnost da funkcija `pitajZaSmer` može da ispali izuzetak. Ovo je prednost rada sa izuzecima: kod za obradu greške je neophodan samo na mestu gde je greška nastala i na mestu gde se obrađuje. Sve funkcije između ta dva mesta mogu da zaborave na obradu grešaka.

Nekada je i u slučaju ispaljivanja izuzetka i u slučaju da nema izuzetaka potrebno izvršiti neko parče koda. Ovo se može uraditi pomoću `finally`-bloka, koji se nalazi iza `catch`-bloka (odnosno, iza `try`-bloka, ukoliko `catch`-blok ne postoji).

Zadatak 2.13 Naredni primer ilustruje veoma loš kod za rad sa računima. Primetimo da program prvo skida novac sa računa pre nego što možda dođe do izuzetka, čime se prekida rad i može doći do gubitka novca:

```
1  const accounts = {
2      a: 100,
3      b: 0,
4      c: 20
5  };
6
7  function getAccount() {
8      let accountName = prompt('Enter an account name');
9      if (!accounts.hasOwnProperty(accountName)) {
10         throw new Error(`No such account: ${accountName}`);
11     }
12     return accountName;
13 }
14
15 function transfer(from, amount) {
```

¹¹Primetimo da se u rečenici nalazi ekskluzivna disjunkcija — desiće se tačno jedna od dve stvari.

```

16     if (accounts[from] < amount) {
17         return;
18     }
19     accounts[from] -= amount;
20     accounts[getAccount()] += amount;
21 }

```

Analizirati naredni kod i pronaći sva poboljšanja u odnosu na prvobitnu verziju funkcije `transfer`:

```

1 function transfer(from, amount) {
2     if (accounts[from] < amount) {
3         return;
4     }
5     let progress = 0;
6     try {
7         accounts[from] -= amount;
8         progress = 1;
9         accounts[getAccount()] += amount;
10        progress = 2;
11    } finally {
12        if (progress == 1) {
13            accounts[from] += amount;
14        }
15    }
16 }

```

Hvatanje različitih izuzetaka

Nekada će se desiti da deo koda može da ispali različite izuzetke, na primer:

```

1 function pitajZaSmer(pitanje) {
2     let rezultat = prompt(pitanje);
3     if (rezultat.toLowerCase() == 'levo') {
4         return 'L';
5     } else if (rezultat.toLowerCase() == 'desno') {
6         return 'D';
7     }
8     throw new Error('Nemoguc smer: ' + rezultat);
9 }
10
11 for (let i = 0; i < 1e4; ++i) {
12     try {
13         let smer = pitajZaSmer('Kuda?');
14         console.log('Odabrali ste ', smer);
15         break;
16     } catch (e) {
17         console.log('Nije ispravan smer. Pokusajte ponovo.');

```

Napravili smo grešku u kucanju naziva funkcije `pitajZaSmer`. Iako mi očekujemo da bude ispaljena greška za pogrešan korisnički unos, biće ispaljen izuzetak za nepostojeću referencu `pitjZaSmer`. Nažalost, u ovom kodu mi samo "progutamo" izuzetak i nastavljamo dalje, što će proizvesti beskonačnu petlju.

Rešenje problema jeste u izvođenju posebne klase za grešku:

```

1 class InputError extends Error {}

```



```

2
3 function pitajZaSmer(pitanje) {
4     let rezultat = prompt(pitanje);
5     if (rezultat.toLowerCase() == 'levo') {
6         return 'L';
7     }
8     else if (rezultat.toLowerCase() == 'desno') {
9         return 'D';
10    }
11    throw new InputError('Nemoguc smer: ' + rezultat);
12 }
13
14 for (;;) {
15     try {
16         let smer = pitajZaSmer('Kuda?'); // typo!
17         console.log('Odabrali ste ', smer);
18         break;
19     } catch (e) {
20         if (e instanceof InputError) {
21             console.log('Nije ispravan smer. Pokusajte ponovo.');

```

Zadatak 2.14 Napisati funkciju `cudnoMnozenje` koja prilikom pozivanja sa verovatnoćom od 20% množi dva broja i vraća njihov rezultat, a sa verovatnoćom od preostalih 80% ispaljuje izuzetak tipa `MultiplierUnitFailure`. Napisati funkciju `ispravnoMnozenje` koja pokušava da pomnoži dva broja pomoću funkcije `cudnoMnozenje` sve dok ne uspe, nakon čega vraća rezultat. Obezbediti da naredni fragment koda ispiše vrednost 70:

```
1 console.log(ispravnoMnozenje(10, 7)); // 70
```

Napomena: obrađivati samo neophodne izuzetke. ■

2.7 Moduli

Modul (engl. *module*) je deo programa koji specifikuje na koje se druge delove programa on oslanja i koje mogućnosti on daje ostalim delovima programa. Te funkcionalnosti koje su izložene "svetu" zajedno se nazivaju *interfejs* (engl. *interface*) modula. Sve ostalo se smatra privatnim za taj modul, i o tome svet ne mora da vodi računa. Korišćenjem modula se smanjuje celokupna povezanost delova programa, odnosno, tendencija da "sve zna za sve", što se smatra lošom praksom za programiranje.

Odnosi između modula se nazivaju *zavisnosti* (engl. *dependency*). Kada modul zahteva deo iz nekog drugog modula, kažemo da taj modul zavisi od drugog modula. Da bismo razdvojili module, potrebno je da svaki modul ima svoj, privatni domaćin.

2.7.1 Paketi

Paketi (engl. *package*) predstavljaju delove koda koji se mogu distribuirati (kopirati i instalirati). Paket može da sadrži jedan ili više modula i ima informacije o tome od kojih drugih paketa zavisi. Paket obično sadrži i dokumentaciju. Kada se problem pronađe u paketu ili se doda nova mogućnost, paket se ažurira. Tada, programi koji zavise od paketa (koji takođe mogu biti drugi paketi) mogu da se jednostavno ažuriraju na novu verziju.

Rad na ovakav način zahteva infrastrukturu — mesto za skladištenje i pronalaženje paketa, kao i jednostavan način za njihovo instaliranje i ažuriranje. U svetu JavaScript jezika, ovakva infrastruktura je obezbeđena od strane NPM (<https://npmjs.org>). NPM predstavlja dve stvari: (1) veb servis odakle je moguće preuzeti (i postaviti) pakete i (2) program (koji dolazi uz Node.js okruženje za JavaScript) koji pomaže instaliranje i upravljanje paketima. U tekstu ćemo videti kako možemo instalirati različite pakete za naše programe koje budemo pisali.

2.7.2 Načini za kreiranje modula

U nastavku teksta opisujemo dva najpoznatija metoda za kreiranje modula: CommonJS i ECMAScript module.

CommonJS

Ovaj način predstavlja verovatno najrasprostranjeniji način za kreiranje JavaScript modula. Sistem Node.js koristi upravo ovaj način, a takođe i većina paketa na NPM sistemu, te ćemo mu se zbog toga i posvetiti.

Glavni koncept u CommonJS modulima jeste funkcija `require`. Kada pozovemo ovu funkciju sa imenom modula od kojeg zavisi naš modul, funkcija se postara da je modul učitani i vraća nam njegov interfejs. Pošto učitavanje postavlja omotač nad kodom modula u funkciji, moduli automatski dobijaju svoj lokalni domet. Sve što oni treba da urade jeste da `require` pozivima dohvataju pakete od kojih zavise, a svoj interfejs stave u objekat koji se vezuje za `exports`.

Sada ćemo napraviti jedan primer modula koji sadrži funkciju za formatiranje datuma. Ovaj modul koristi dva paketa iz sistema NPM — `ordinal`, da bi mogao da konvertuje brojeve u niske poput `'1st'`, a `date-names` za dobijanje imena dana i meseci na engleskom jeziku. Interfejs ovog modula čini jedna funkcija `formatDate`, čiji su argumenti `Date` objekat i šablonsku nisku. Šablonska niska može da sadrži kodove kojima se definiše format, poput `YYYY` za četvorocifrenu godinu i `Do` za redni dan u mesecu. Na primer, ukoliko prosledimo nisku `'MMMM Do YYYY'`, mogli bismo očekivati rezultat tipa `"June 12th 2019"`.

```

1  const ordinal = require('ordinal');
2  const {days, months} = require('date-names');
3
4  exports.formatDate = function(date, format) {
5      return format.replace(/YYYY|M(MM)?|Do?|dddd/g, tag => {
6          if (tag === 'YYYY') {
7              return date.getFullYear();
8          }
9          else if (tag === 'M') {
10             return date.getMonth();
11          }
12          else if (tag === 'MMMM') {
13              return months[date.getMonth()];
14          }
15          else if (tag === 'D') {
16              return date.getDate();
17          }
18          else if (tag === 'Do') {
19              return ordinal(date.getDate());
20          }
21          else if (tag === 'dddd') {
22              return days[date.getDay()];
23          }
24      });

```

25 }

Interfejs modula `ordinal` čini jedna funkcija, dok modul `date-names` eksportuje objekat kojeg čini više stvari — `days` i `months` su nizovi imena. Dekonstrukcija je veoma korisna tehnika kada kreiramo vezivanja za interfejse koje učitavamo.

Ovaj modul dodaje funkciju u svoj interfejs `exports` tako da moduli koji zavise od njega imaju pristup funkciji. Recimo da je modul sačuvan u datoteci `format-date.js` u tekućem direktorijumu. Ako želimo da ga koristimo u drugom fajlu u istom direktorijumu, dovoljno je da napišemo:

```
1 const {formatDate} = require('./format-date');
2
3 console.log(formatDate(new Date(2019, 6, 12), 'dddd the Do'));
4 // Friday the 12th
```

Ono što smo možda primetili jeste da je interfejs paketa `ordinal` funkcija, a ne objekat. Zanimljiva stavka CommonJS modula je da, iako sistem modula kreira prazan objekat-interfejs za nas (koji je vezan za promenljivu `exports`), mi ga možemo zameniti bilo kojom vrednošću tako što pregazimo vrednost `module.exports`. Ovo je urađeno u velikom broju postojećih modula da bi se exportovala jedna vrednost umesto objekta-interfejsa.

Kada putanja za učitavanje paketa nije relativna, Node.js će pogledati instalirane pakete i potražiće u njima paket sa prosleđenim imenom. Instaliranje paketa u korenom direktorijumu projekta se može izvršiti komandom

```
$ npm install imePaketa
```

ECMAScript moduli

JavaScript standard iz 2015. godine je doneo novi sistem za upravljanje modulima. Ovaj sistem se obično naziva ES moduli, gde je ES skraćenica sa ECMAScript. Osnovni koncept zavisnosti i interfejsa ostaje isti, ali se detalji razlikuju. Za početak, notacija koja se koristi je integrisana u jezik. Umesto pozivanja funkcije za dohvatanje zavisnih paketa, koristi se ključna reč `import`:

```
1 import ordinal from 'ordinal';
2 import {days, months} from 'date-names';
3
4 export function formatDate(date, format) { /* ... */ }
```

Slično, ključna reč `export` se koristi za eksportovanje stvari kao deo interfejsa. Može se pojaviti ispred funkcije, klase ili promenljive (`let`, `const` ili `var`).

Interfejs ES modula nije jedna vrednost već skup imenovanih vrednosti. Prethodni primer vezuje ime `formatDate` za funkciju. Kada učitavamo interfejs iz drugog modula, mi zapravo učitavamo vezivanja, a ne vrednosti, što znači da eksportovani modul može da promeni vrednost imenovanja u bilo kom trenutku, tako da modul koji ga učitava vidi tu novu vrednost.

Tamo gde je definisano vezivanje pod nazivom `default`, ono se tretira kao glavna stvar koja se izvozi u modulu. Ako uvozimo modul poput `ordinal` u primeru, bez zagrada oko imena vezivanja, dobijamo njegovo `default` vezivanje. Takvi moduli i dalje mogu da eksportuju druga vezivanja pored njihovog `default` izvoženja.

Da bismo kreirali `default` izvoženje, dovoljno je napisati `export default` ispred izraza, deklaracije funkcije ili deklaracije klase:

```
1 export default ['Zima', 'Prolece', 'Leto', 'Jesen'];
```

Jedna bitna razlika je da se uvoženje ES modula događa pre nego što je skript modula pokrene. Ovo znači da `import` deklaracije ne mogu da se nađu unutar funkcija ili blokova, i nazivi zavisnih paketa moraju biti niske pod navodnicima, ne proizvoljni izrazi.

Angular sistem za razvoj intenzivno koristi ES module, te ih je potrebno razumeti i naučiti.

2.8 Asinhrona paradigma programiranja

U *sinhronom* modelu programiranja, stvari se dešavaju sekvencionalno, jedna za drugom. Kada pozovemo funkciju koja izvršava neku akciju, ona se vraća tek kada je akcija završena i tada može da vrati neki rezultat. Ovim se program stopira za ono vreme koliko je bilo potrebno toj akciji da se završi.

Asinhroni model programiranja dozvoljava više stvari da se dešavaju u isto vreme. Kada pokrenemo neku akciju, program nastavlja sa radom. Onog trenutka kada akcija završi, program biva informisan o završetku akcije i dobija pristup rezultatu.

Poređenje između sinhronog i asinhronog programiranja se može jednostavno sagledati kroz naredni primer. Neka je potrebno napisati program koji dohvata dva resursa sa interneta, a zatim kombinuje rezultate. U sinhronom modelu, najjednostavniji način jeste da se zahtevi za resursima vrše jedan za drugim. Problem ovog pristupa jeste da se drugi zahtev započinje tek onda kada se prvi završi. Ukupno vreme rada programa je najmanje zbir trajanja dohvaćanja odgovora. U asinhronom modelu, moguće je poslati dva zahteva jedan za drugim, a zatim u trenutku kada su obe vrednosti dostupne, vrši se kombinovanje rezultata. Prednost ovog modela je u tome što se preklapa vreme potrebno za dohvaćanje dva resursa, čime se značajno ubrzava rad programa.

2.8.1 JavaScript je jednonitni programski jezik

Izvršavanje JavaScript koda koji neki programer napiše se izvršava u jednoj niti, što znači da se u jednom trenutku može izvršiti samo jedna naredba. Ovo ponašanje povlači neke dobre i loše osobine. Sa jedne strane, ne moramo da vodimo računa o *utrkivanju* (engl. *race condition*) više niti nad istim resursima. Međutim, ukoliko postoji neka skupa operacija koju JavaScript mašina za izvršavanje mora da izračuna, postoji mogućnost da će blokirati neke druge operacije i stvoriti nepoželjne efekte.

Na primer, pretpostavimo da imamo veb aplikaciju koja prikazuje nekakav glavni sadržaj, a pored toga, prikazuje informacije o vremenskoj prognozi i vestima sa strane (kao sporedne elemente). Neka je aplikacija implementirana tako da izvršava naredne korake jedan za drugim:

- Dohvati podatke o vremenskoj prognozi (sa spoljašnjeg API-ja).
- Dohvati podatke o poslednjim vestima (sa spoljašnjeg API-ja).
- Prikaži dohvaćene informacije o vremenskoj prognozi u bočnoj traci.
- Prikaži dohvaćene informacije o poslednjim vestima u bočnoj traci.
- Prikaži glavni sadržaj HTML stranice.

Ukoliko bi dohvaćanje podataka u prvim koracima trajalo po, na primer, 5 sekundi, onda bi korisnik prilikom otvaranja veb aplikacije u veb pregledaču prvih 10 sekundi gledao u „prazan” ekran. Mnogo bi pogodnije bilo da veb pregledač prikazuje glavni sadržaj aplikacije (što je i najvažniji deo) odmah nakon učitavanja, dok čeka da mu pristignu podaci sa spoljašnjeg API-ja.

Naravno, rešenje ovog konkretnog problema jeste u preraspoređivanju koraka tako da se najvažnije operacije izvršavaju prve, a sporedne na kraju. Međutim, u složenijim aplikacijama u praksi koje treba da „istovremeno” izvršavaju hiljade operacija, često nije moguće precizno odrediti važnost tih operacija. U ovakvim slučajevima, asinhroni model opisan iznad igra važnu ulogu.

Postavlja se pitanje kako je moguće implementirati asinhroni model u programskom jeziku koji se izvršava u jednoj niti. Kako bismo to razumeli, neophodno je da objasnimo neke elemente od kojih se sastoji JavaScript okruženja za izvršavanje. Započnimo prvo sa elementima izvršavanja sinhronog JavaScript koda. Pogledajmo naredni primer:

```
1  const message = 'Hello there!';
2
3  function second() {
4      console.log(message);
5  }
6
7  function first() {
8      const message = 'Hi there!';
9      console.log(message);
10
11     second();
12
13     function third() {
14         console.log(message);
15     }
16
17     third();
18 }
19
20 first();
```

Kako bismo razumeli kako se kod iznad izvršava, potrebno je da razumemo koncepte *konteksta izvršavanja* i *steka poziva funkcija*.

Kontekst izvršavanja

U pitanju je apstraktan koncept okruženja u kojem se JavaScript kod prevodi i izvršava. Svaki fragment JavaScript koda se izvršava u okvirima nekakvog konteksta izvršavanja. Tako, na primer, kod koji se nalazi u nekoj funkciji se izvršava u kontekstu izvršavanja te funkcije, dok se kod koji se nalazi u globalnom opsegu izvršava u globalnom kontekstu izvršavanja. Svaka funkcija ima svoj kontekst izvršavanja. Koncepti, kao što su postavljanje vrednosti promenljivih ili pretraga identifikatora zavise od konteksta izvršavanja.

Stek poziva funkcija

U pitanju je LIFO struktura podataka koja se koristi za skladištenje svih konteksta izvršavanja koji se kreiraju tokom izvršavanja koda. S obzirom da je JavaScript jednonitni programski jezik, okruženje za izvršavanje sadrži samo jedan stek poziva funkcija. Kada se pozove neka funkcija *f*, kreira se njen kontekst izvršavanja i stavlja se na vrh steka. Zatim, funkcija *f* se izvršava od početka do kraja. Kada se izvršavanje funkcije *f* okonča, njen kontekst izvršavanja se uklanja sa vrha steka. U ovom trenutku, njen kontekst će biti uništen, osim ako ne postoji neka druga funkcija *g* koja ima potrebu da zapamti kontekst izvršavanja funkcije *f* (tj. ako kontekst izvršavanja funkcije *f* ne ulazi u zatvorenje funkcije *g*).

Pogledajmo na slici 2.15 kako se menja stek poziva funkcija iz primera iznad.



Slika 2.15: Promena sadržaja steka poziva funkcija tokom izvršavanja JavaScript koda.

Svaki kontekst izvršavanja sadrži informaciju o kontekstu izvršavanja koji mu „prethodi” (sa izuzetkom globalnog konteksta izvršavanja koji nema „prethodnika”), što je na slici označeno podebljanom ljubičastom strelicom sa leve strane. Primetimo da ovi „prethodnici” zavise od toga na koji način su funkcije definisane. Na primer, funkcije `first`, `second` i `console.log` su definisane u globalnom kontekstu izvršavanja, te zbog toga je upravo on njihov „prethodnik”. Za razliku od toga, funkcija `third` je definisana u kontekstu izvršavanja funkcije `first`, te je zbog toga „prethodnik” njenog konteksta izvršavanja upravo kontekst izvršavanja funkcije `first`.

Razliku između ova dva ponašanja možemo videti na slikama 2.15(f) i 2.15(j):

- Na prvoj od njih, prilikom pozivanja funkcije `console.log` unutar funkcije `second`, promenljiva `message` se prvo pretražuje u kontekstu izvršavanja funkcije `second`. S obzirom da nije tu pronađena, razmatra se njegov „prethodnik”, što je globalni kontekst izvršavanja. U njemu se pronalazi promenljiva čija je vrednost `"Hello there!"` i ta poruka se ispisuje.
- Na drugoj od njih, prilikom pozivanja funkcije `console.log` unutar funkcije `third`, promenljiva `message` se prvo pretražuje u kontekstu izvršavanja funkcije `third`. S obzirom da nije tu pronađena, razmatra se njegov „prethodnik”, što je kontekst izvršavanja funkcije `first`. U njemu se pronalazi promenljiva čija je vrednost `"Hi there!"` i ta poruka se ispisuje.

Sada kada smo usvojili ove koncepte, trebalo bi da bude jasno da će prethodni kod ispisati sledeće u konzoli:

```
Hi there!  
Hello there!  
Hi there!
```

Pređimo sada na implementaciju asinhronog izvršavanja. U tu svrhu, razmotrimo naredni kod.

```
1 function networkRequest() {  
2   setTimeout(function() {  
3     console.log('Async Code');  
4   }, 2000);  
5 };  
6  
7 networkRequest();  
8 console.log('Hello World');
```

Ovaj kod će ispisati naredne poruke u konzoli:

```
Hello World  
Async Code
```

Kako bismo objasnili zašto prethodni kod ispisuje dati izlaz u konzoli, neophodno je da uvedemo još neke koncepte. *Petlja događaja*, *redovi zadataka* i *mikrozadataka* i *web API* predstavljaju delove koji nisu direktno ugrađeni u JavaScript mašinu, već se nalaze kao deo okruženja u kojima se JavaScript izvršava (bilo kao deo veb pregledača ili NodeJS okruženja).

Petlja događaja

Zadatak petlje događaja jeste da sakuplja sve događaje koji se okidaju tokom izvršavanja programa i da za svaki događaj koji ima pridruženu funkciju dodaje tu funkciju na kraj reda zadataka ili mikrozadataka. Zatim, ona izvršava zadatke koji stoje na čekanju. Konačno, u zavisnosti od okruženja, izvršavaju se neke dodatne funkcionalnosti pre nego što se pređe na narednu iteraciju petlje. Na primer, u slučaju veb pregledača vrše se neophodna iscertavanja.

Konkretnije, petlja događaja prati stanje steka poziva funkcija. Ukoliko je stek poziva funkcija prazan, on kontaktira redove zadataka i mikrozadataka i pita da li postoji neki zadatak (funkcija) koji čeka na izvršenje. Ukoliko postoji, onda će petlja događaja povući zadatak sa vrha reda i staviti ga na vrh steka.

Red zadataka

U pitanju je FIFO struktura podataka koja skladišti zadatke koji čekaju da budu procesirani. Svakom zadatku je pridružena funkcija koja implementira taj zadatak. Da bi se zadatak procesirao, potrebno je da se odgovarajuća funkcija povuče iz reda zadataka i smesti na stek poziva funkcija. S obzirom da se zadatak (funkcija) izvršava tek kada se pojavi na vrhu steka poziva funkcija, kao i činjenica da se izvršava u celosti (tj. od početka do kraja funkcije, bez prekida), čini asinhroni model izvršavanja imunim na probleme konkurentnog izvršavanja. Naravno, kao što smo primetili i ranije, ukoliko je jedan zadatak vrlo dugo izvršava, onda će on blokirati ostale zadatke koji se nalaze iza njega u redu zadataka.

Red mikrozadataka

Red mikrozadataka predstavlja istu strukturu podataka kao red zadataka, ali postoje neke razlike. Kada se zadaci izvršavaju sa reda zadataka, onda će se tačno jedan zadatak izvršiti u jednoj iteraciji petlje događaja. Dodatno, zadaci koji se dodaju na red zadataka u nekoj iteraciji, neće biti izvršeni sve do naredne iteracije. Za razliku od toga, kada neka funkcija na steku poziva funkcija završi sa radom, i ako je stek poziva funkcija prazan, onda će se svi zadaci koji se nalaze u redu mikrozadataka izvršiti jedan za drugim. Razlika je u tome što se izvršavanje mikrozadataka nastavlja sve dok se red mikrozadataka ne isprazni, čak i ako se u međuvremenu doda novi mikrozadatak. Drugim rečima, mikrozadaci mogu da dodaju nove mikrozadatke u red i svi oni će biti izvršeni u celosti i to u istoj iteraciji petlje događaja.

Veb API

Kako bi se omogućilo ostvarivanje asinhronne paradigme programiranja, JavaScript okruženja implementiraju razne interfejsse za programiranje aplikacija. Veb API¹² predstavlja kolekciju velikog broja interfejsa, među kojima se nalaze i oni za ostvarivanje asinhronne paradigme programiranja. Neki od njih su:

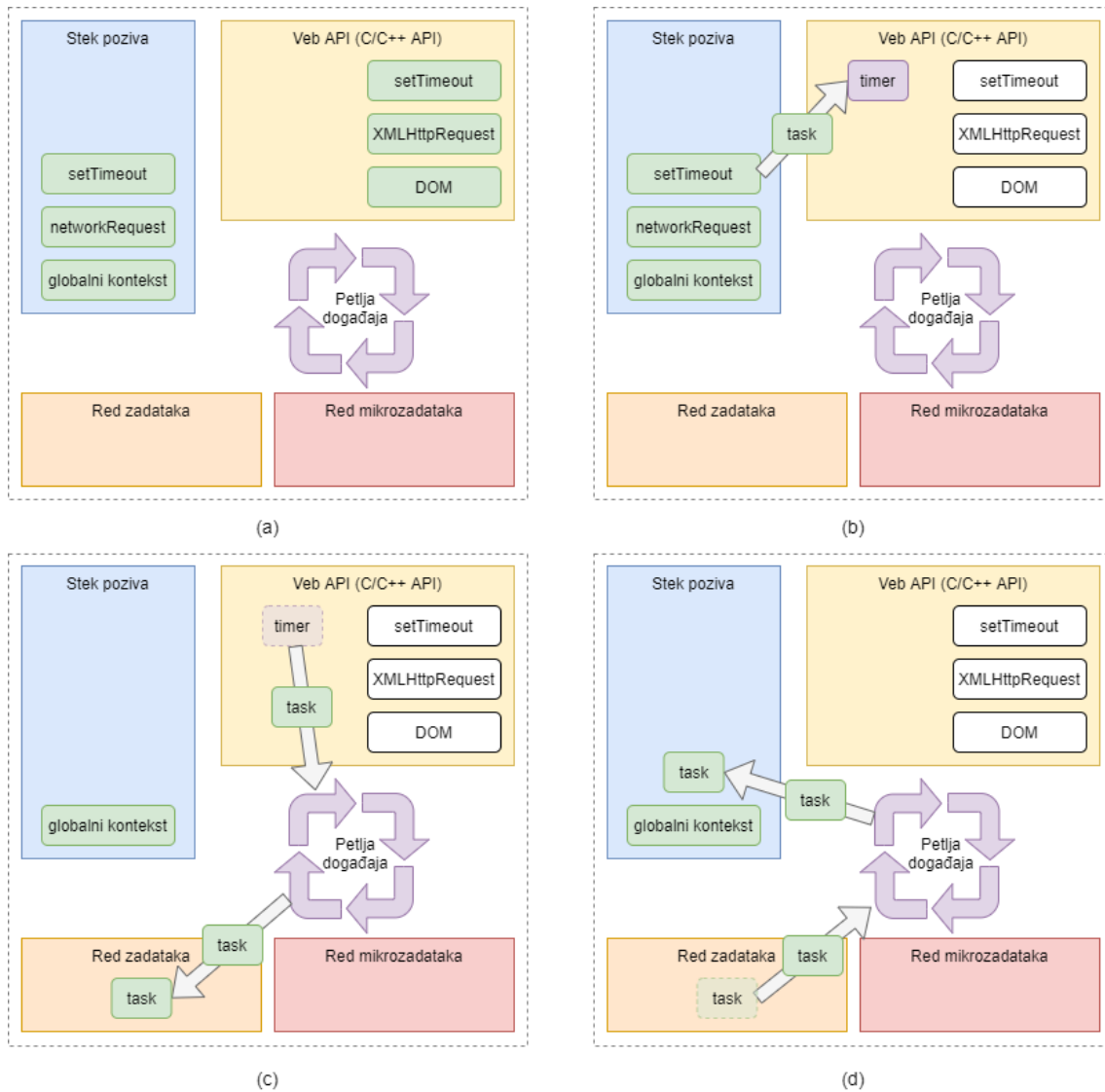
- Funkcije `setTimeout` i `setInterval` služe za dodavanje funkcija u red zadataka (koja se postavlja kroz prvi argument ovih funkcija). Ove funkcije postavljaju i tajmer koji će pozvati odgovarajuću funkciju na određen broj milisekundi (koji se postavlja kroz drugi argument ovih funkcija). Razlika između ovih funkcija je u tome što će se prva funkcija pozvati tačno jednom, a druga funkcija će biti učestalo pozivana. Tajmeri koje ove funkcije postavljaju se mogu ukloniti pozivom funkcija `clearTimeout` i `clearInterval`, redom. Njihov argument je povratna vrednost funkcija `setTimeout` i `setInterval`.
- Funkcija `queueMicrotask` dodaje funkciju koja joj se prosleđuje u red mikrozadataka. Ova funkcija će biti pozvana na kraju tekuće iteracije petlje događaja.
- Klasa `XMLHttpRequest` i funkcija `fetch` se mogu koristiti za dodavanje funkcija u red zadataka, odnosno, red mikrozadataka koji će biti pozvani nakon što se dobije odgovor na HTTP zahtev.
- Podskup interfejsa DOM API-ja koji služi za asinhrono izvršavanje zadataka nad događajima koji se okidaju u pogledu veb pregledača (klik miša na neki HTML elementa, unos podataka u formular, i sl.).
- IndexedDB API služi za asinhronu komunikaciju sa sistemom za upravljanje nerelacionim bazama podataka (sa fiksnim kolonama) za potrebe skladištenja struktuiranih podataka na klijentskoj strani.
- ...

¹²Ili C/C++ API u Node.js okruženju.

Podsetimo se koda koji treba da analiziramo nakon što smo usvojili ove koncepte.

```

1 function networkRequest() {
2   setTimeout(function() {
3     console.log('Async Code');
4   }, 2000);
5 };
6
7 networkRequest();
8 console.log('Hello World');
```



Slika 2.16: Promena sadržaja JavaScript okruženja tokom izvršavanja JavaScript koda.

Na slici 2.16 su prikazana stanja steka poziva funkcija, redova zadataka i mikrozadataka i Veb API-ja prilikom izvršavanja datog koda u različitim trenucima:

- Slika 2.16(a) prikazuje trenutak tik pred poziva funkcije `setTimeout`. Kao što vidimo, u tom trenutku su redovi zadataka prazni.
- Slika 2.16(b) prikazuje trenutak tik nakon poziva funkcije `setTimeout`. Kao što vidi-mo, u tom trenutku su redovi zadataka i dalje prazni. Međutim, sada u okviru veb

API-ja imamo kreiran tajmer u trajanju od 2 sekunde (2000 milisekundi–drugi argument funkcije `setTimeout`). Uz ovaj tajmer je pridružena i funkcija koja predstavlja zadatak koji treba da se izvrši nakon 2 sekunde. U istoj iteraciji petlje događaja, funkcije `setTimeout` i `networkRequest` završavaju sa radom i njihovi konteksti se uklanjaju sa steka. Zatim, izvršava se poslednja linija u kodu kojom se u konzoli ispisuje poruka `'Hello World'`.

- Slika 2.16(c) prikazuje trenutak kada je tajmer istekao. Tajmer se uklanja iz veb API-ja (na slici je taj tajmer prikazan prozirnom pozadinom) i pridružena funkcija se prosleđuje petlji događaja koja ovaj zadatak dodaje na kraj reda zadataka.
- Slika 2.16(d) prikazuje trenutak u narednoj iteraciji petlje događaja u odnosu na sliku 2.16(c). Petlja događaja prvo uklanja zadatak sa vrha reda zadataka (na slici je taj zadatak prikazan prozirnom pozadinom), a zatim od JavaScript mašine zahteva da se kreira kontekst izvršavanja za taj zadatak i dodaje ga na vrh steka poziva funkcija, čime se započinje izvršavanje tog zadatka (tj. pridružene funkcije). U funkciji se poziva `console.log` koji će na konzoli ispisati poruku `'Async Code'`.

2.8.2 Podrška za asinhrono programiranje u jeziku JavaScript

Do sada smo videli različite mehanizme kojima se ostvaruje asinhrono izvršavanje koda. Međutim, svi ovi koncepti su deo okruženja u kojem se JavaScript kod izvršava, ali oni nisu vezani za sam jezik. Sada ćemo predstaviti elemente koji su ugrađeni u sam JavaScript kojima je moguće programirati asinhronu aplikacije.

Asinhrono programiranje u programskom jeziku JavaScript možemo ostvariti kroz naredne koncepte:

- Funkcije povratnog poziva
- Obećanja
- Asinhronu funkcije
- Generatorske funkcije¹³

N Za izvršavanje koda u ostatku ove sekcije, neophodno je u direktorijumu gde se nalaze ovi primeri (na putanji `primeri/javascript/asinhrono-programiranje/`) pokrenuti veb server. Ukoliko samo otvorimo `.html` datoteku u veb pregledaču, on će koristiti `file://` protokol za učitavanje te datoteke. Međutim, neki od primera koriste HTTP komunikaciju za dohvaćanje podataka (kako bi se ilustrovala asinhronost akcija kroz praktične primere), te je neophodno pristupati primerima preko `http://` protokola.

Ovo možemo najjednostavnije uraditi u alatu Visual Studio Code, u okviru kojeg je dostupna ekstenzija naziva „Live Server” autora Ritvik Deja (eng. Ritwick Dey). Nakon instalacije ekstenzije, potrebno je otvoriti direktorijum u Visual Studio Code alatu. Zatim, desnim klikom na datoteku `index.html` otvoriti pomoćni meni iz koga je potrebno odabrati opciju „Open with Live Server”. Alternativno, moguće je aktivirati „Live Server” klikom na dugme „Go Live” u statusnoj liniji alata. Ako se otvorio veb pregledač na adresi `http://127.0.0.1:5500/index.html`, onda je veb server korektno pokrenut. Otvorena stranica sadrži veze ka ostalim primerima iz ove sekcije zarad lakšeg pristupa i izvršavanja. Poželjno je otvoriti i *konzolu* u *alatima za razvijanje*.

Funkcije povratnog poziva

Jedan pristup asinhronom programiranju jeste da se funkcije koje izvršavaju duge ili spore akcije konstruišu tako da prihvataju dodatni argument, funkciju koja će biti izvršena

¹³Navodimo ih ovde radi kompletnosti, ali o njima neće biti reči.

asinhrono. Ovakve funkcije se u kontekstu asinhronih izvršavanja nazivaju u literaturi kao *funkcije povratnog poziva* (engl. *callback function*). Ideja je da će funkcije povratnog poziva predstavljati zadatke koji će biti izvršeni asinhrono, u trenutku kada programer definiše da se ta funkcija pozove (na primer, nakon isteka tajmera ili nakon dobijanja HTTP odgovora od servera).

Kao primer ovog modela, možemo razmotriti funkciju `setTimeout`, dostupnu i u veb pregledačima i u Node.js platformi, koja prihvata dva argumenta: funkciju povratnog poziva i broj milisekundi. Funkcija `setTimeout` postavlja tajmer koji traje prosleđeni broj milisekundi i po isteku tajmera, poziva funkciju povratnog poziva u red zadataka. Pogledajmo naredni primer.

Kod 2.1: javascript/asinhrono-programiranje/callback1.html

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <title>Asinhrono programiranje - funkcije povratnog poziva</title>
6   </head>
7   <body>
8     <h3>setTimeout</h3>
9
10    <script>
11      document.body.appendChild(document.createTextNode(" 1 "));
12      setTimeout(function () {
13        document.body.appendChild(document.createTextNode(" 2 "));
14      }, 500);
15      document.body.appendChild(document.createTextNode(" 3 "));
16    </script>
17  </body>
18 </html>
```

Kao što vidimo, funkcija povratnog poziva koja se prosleđuje funkciji `setTimeout` biće izračunata nakon pola sekunde, dok se ostatak koda izvršava odmah. Zbog toga, u veb pregledaču vidimo da se prvo prikazuju brojevi „1” i „3”, pa tek onda broj „2”. Ovo je ponašanje koje bismo očekivali čak i da ne znamo ništa o izvršavanju asinhronog koda.

Međutim, pogledajte naredni primer i zapitajte se šta će biti ispisano u veb pregledaču.

Kod 2.2: javascript/asinhrono-programiranje/callback2.html

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <title>Asinhrono programiranje - funkcije povratnog poziva</title>
6   </head>
7   <body>
8     <h3>setTimeout</h3>
9
10    <script>
11      document.body.appendChild(document.createTextNode(" 1 "));
12      setTimeout(function () {
13        document.body.appendChild(document.createTextNode(" 2 "));
14      }, 0);
15      document.body.appendChild(document.createTextNode(" 3 "));
16    </script>
17  </body>
18 </html>
```

S obzirom da se funkcija povratnog poziva koja prikazuje broj „2” poziva nakon nula sekundi, oni koji su preskočili prethodnu sekciju bi stekli naivan zaključak da se prikazuje „1 2 3”. Međutim, pokretanjem primera u veb pregledaču vidimo da je rezultat prikaza „1 3 2”. Okrepljeni znanjem iz prethodne sekcije, znamo da se funkcije koje se dodaju u red zadataka izvršavaju tek u narednoj iteraciji petlje događaja (i kada je stek poziva funkcija prazan). Dakle, JavaScript okruženje prvo izvršava „sinhroni deo koda” od početka do kraja bez prekida, pa u narednoj iteraciji petlje događaja uklanja funkciju povratnog poziva iz reda zadataka i izvršava je. Naravno, isti efekat bismo dobili i da smo umesto `setTimeout` pozvali `queueMicrotask`, sa razlikom da bi se funkcija povratnog poziva pozvala na kraju iste iteracije petlje događaja (ali i dalje nakon „sinhronog dela koda”).

Naredni primer koristi pomenutu klasu `XMLHttpRequest` iz veb API-ja da kreira HTTP zahtev čija će se obrada događaja izvršiti asinhrono pomoću funkcije povratnog poziva. Cilj je napisati funkciju `httpGet` koja za dati URL kreira HTTP GET zahtev ka resursu koji je identifikovan tim URL-om. Pretpostavka je da će biti dohvaćeni podaci u JSON formatu. S obzirom da objekti klase `XMLHttpRequest` koriste funkcije povratnih poziva kako bi implementirali asinhrono izvršavanje funkcije, onda i naša funkcija `httpGet` mora da prihvati funkciju povratnog poziva kao argument `callback` koju će koristiti zajedno sa objektom klase `XMLHttpRequest`.

Svaki put kada se promeni interno stanje objekta klase `XMLHttpRequest` (što označava promenu stanja HTTP zahteva), poziva se funkcija povratnog poziva koja se smešta kao metod `onreadystatechange` tog objekta. Ispitivanjem vrednosti svojstva `readyState` možemo proveriti da li je stigao HTTP odgovor od servera (vrednost ovog svojstva je u tom slučaju jednaka broju 4). Ako jeste, onda možemo ispitati statusni kod i pozvati funkciju `callback`. Primetite da očekujemo da funkcija `callback` treba biti definisana sa 2 argumenta. Prvi predstavlja objekat greške i prosleđuje se samo ako je došlo do greške u HTTP zahtevu, a inače ima vrednost `null`. Drugi argument predstavlja dohvaćene podatke od servera u slučaju uspeha, odnosno, `null` u slučaju neuspeha. Prosleđivanje objekta greške funkciji povratnog poziva kao prvi argument nije bilo neophodno uraditi, ali predstavlja dobru praksu i vrlo je popularno u Node.js bibliotekama.

Prilikom poziva funkcije `httpGet`, prosleđujemo i funkciju povratnog poziva sa 2 argumenta koja su opisana iznad. Prvo proveravamo da li je došlo do greške, a zatim implementiramo odgovarajuću operaciju nad dohvaćenim podacima. U ovom primeru, prikazujemo informacije o studentima.

Kod 2.3: javascript/asinhrono-programiranje/callback3.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <title>Asinhrono programiranje – funkcije povratnog poziva</title>
6   </head>
7   <body>
8     <h3>Studenti:</h3>
9
10    <script>
11      function httpGet(url, callback) {
12        const httpReq = new XMLHttpRequest();
13        httpReq.onreadystatechange = function () {
14          if (httpReq.readyState == 4) {
15            if (httpReq.status == 200) {
16              const data = JSON.parse(httpReq.responseText);

```

```

17         callback(null, data);
18     } else {
19         const err = new Error(httpReq.statusText);
20         callback(err, null);
21     }
22 }
23 };
24 httpReq.open("GET", url, true);
25 httpReq.send();
26 }
27
28 httpGet("../data/students.json", function (err, students) {
29     // Greska se moze simulirati izmenom naziva datoteke iznad u, na primer
30     // , studenti.json
31     if (err) {
32         console.log("Doslo je do greske: " + err.message);
33         return;
34     }
35     for (const student of students) {
36         const node = document.createElement("p");
37         node.appendChild(
38             document.createTextNode(
39                 `${student.name} ${student.surname} (${student.index})`
40             )
41         );
42         document.body.appendChild(node);
43     }
44 });
45 </script>
46 </body>
47 </html>

```

Nešto složeniji primer obuhvata dohvaćanje podataka o ispitima koje su dali studenti položili. Prikažimo prvo kod primera, pa prodiskutujemo o njegovim osobinama.

Kod 2.4: javascript/asinhrono-programiranje/callback4.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <title>Asinhrono programiranje – funkcije povratnog poziva</title>
6   </head>
7   <body>
8     <h3>Studenti i ispiti:</h3>
9
10    <script>
11      function httpGet(url, callback) {
12        const httpReq = new XMLHttpRequest();
13        httpReq.onreadystatechange = function () {
14          if (httpReq.readyState == 4) {
15            if (httpReq.status == 200) {
16              const data = JSON.parse(httpReq.responseText);
17              callback(null, data);
18            } else {
19              const err = new Error(httpReq.statusText);
20              callback(err, null);
21            }
22          }
23        };

```

```
24     httpReq.open("GET", url, true);
25     httpReq.send();
26 }
27
28 // Za svakog studenta...
29 httpGet(
30     "./data/students.json",
31     /* callback 1 */ function (err, students) {
32         if (err) {
33             console.log(
34                 "Doslo je do greske pri dohvananju studenata: " + err.message
35             );
36             return;
37         }
38
39         for (const student of students) {
40             const node = document.createElement("p");
41             node.appendChild(
42                 document.createTextNode(
43                     `${student.name} ${student.surname} (${student.index})`
44                 )
45             );
46             document.body.appendChild(node);
47
48             const examNodeList = document.createElement("ul");
49             document.body.appendChild(examNodeList);
50
51             // ...dohvatamo podatke o ispitima.
52             // Primetimo da se ovde vrsi ugnezdjavanje funkcija povratnog
53             // poziva.
54             httpGet(
55                 "./data/exams.json",
56                 /* callback 2 */ function (err, exams) {
57                     if (err) {
58                         console.log(
59                             "Doslo je do greske pri dohvananju ispita: " + err.message
60                         );
61                         return;
62                     }
63
64                     for (const exam of exams) {
65                         if (exam.studentId !== student.id) {
66                             continue;
67                         }
68
69                         const examNode = document.createElement("li");
70                         examNode.appendChild(
71                             document.createTextNode(
72                                 `${exam.final} ${exam.year} - grade: ${exam.grade},
73                                 status: ${exam.status}`
74                             )
75                         );
76                         examNodeList.appendChild(examNode);
77                     }
78                 }
79             );
80         }
81     }
82 </script>
</body>
```

83 `</html>`

S obzirom da se obrada podataka o studentima vrši u funkciji povratnog poziva koja je označena kao `callback1`, onda i dohvaćanje ispita mora biti deo te funkcije. Dakle, u okviru funkcije `callback1` pravimo još jedan poziv `httpGet` funkcije koja dohvata podatke o ispitima. Naravno, i ovom pozivu moramo proslediti funkciju povratnog poziva, koja je u kodu označena kao `callback2`. Ova funkcija povratnog poziva pronalazi ispite za tekućeg studenta i prikazuje podatke o njegovim ispitima. Dakle, imamo ugnežđavanje funkcija povratnog poziva.

Naredni primer proširuje prethodnu implementaciju tako što za svaki ispit jednog studenta dohvata podatke o predmetima kako bi prikazao informacije o predmetu koji je student polagao na tom ispitu. Pažljivi čitalac će prepoznati da je i za ovaj korak neophodno pozvati funkciju `httpGet`, čime dobijamo i treću ugnežđenu funkciju povratnog poziva, u kodu ispod označenu kao `callback3`.

Kod 2.5: javascript/asinhrono-programiranje/callback5.html

```

1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <meta charset="UTF-8" />
5      <title>Asinhrono programiranje - funkcije povratnog poziva</title>
6    </head>
7    <body>
8      <h3>Studenti i ispiti (sa informacijama o predmetima):</h3>
9
10   <script>
11     function httpGet(url, callback) {
12       const httpReq = new XMLHttpRequest();
13       httpReq.onreadystatechange = function () {
14         if (httpReq.readyState == 4) {
15           if (httpReq.status == 200) {
16             const data = JSON.parse(httpReq.responseText);
17             callback(null, data);
18           } else {
19             const err = new Error(httpReq.statusText);
20             callback(err, null);
21           }
22         }
23       };
24       httpReq.open("GET", url, true);
25       httpReq.send();
26     }
27
28     // Ilustracija tzv. "pakla funkcija povratnog poziva".
29     // Kod ispod je vrlo tesko debugovati.
30     httpGet(
31       "./data/students.json",
32       /* callback 1 */ function (err, students) {
33         if (err) {
34           console.log(
35             "Dosló je do greske pri dohvatanju studenata: " + err.message
36           );
37           return;
38         }
39
40         for (const student of students) {
41           const node = document.createElement("p");

```

```

42     node.appendChild(
43         document.createTextNode(
44             `${student.name} ${student.surname} (${student.index})`
45         )
46     );
47     document.body.appendChild(node);
48
49     const examNodeList = document.createElement("ul");
50     document.body.appendChild(examNodeList);
51
52     httpGet(
53         "./data/exams.json",
54         /* callback 2 */ function (err, exams) {
55             if (err) {
56                 console.log(
57                     "Doslo je do greske pri dohvatanju ispita: " + err.message
58                 );
59                 return;
60             }
61
62             for (const exam of exams) {
63                 if (exam.studentId !== student.id) {
64                     continue;
65                 }
66
67                 httpGet(
68                     "./data/courses.json",
69                     /* callback 3 */ function (err, courses) {
70                         if (err) {
71                             console.log(
72                                 "Doslo je do greske pri dohvatanju predmeta: " +
73                                 err.message
74                             );
75                             return;
76                         }
77
78                         for (const course of courses) {
79                             if (exam.courseId !== course.id) {
80                                 continue;
81                             }
82
83                             const examNode = document.createElement("li");
84                             examNode.appendChild(
85                                 document.createTextNode(
86                                     `${exam.final} ${exam.year} - ${course.name} (${
87                                         course.espb}) - grade: ${exam.grade}, status: $
88                                         {exam.status}`
89                                     )
90                                 );
91                             examNodeList.appendChild(examNode);
92                         }
93                     }
94                 );
95             }
96         }
97     );
98 }
99 </script>
100 </body>

```


101 `</html>`

Ono što zaključujemo jeste da programiranje iole složenijih aplikacija koje koriste mehanizam funkcija povratnih poziva zahtevaju ugnežđavanje tih funkcija. Ovo dovodi do pojave nečega što se naziva *pakao funkcija povratnih poziva* (eng. *callback hell*). Osnovna osobina pakla funkcija povratnih poziva jeste kod koji je izuzetno težak za održavanje, ali i za debugovanje.

Ako pogledamo koliko HTTP zahteva smo generisali za implementaciju prethodnog primera (na primer, korišćenjem alata za razvijanje), primetićemo da je taj broj značajno veliki. Ako smo prvim HTTP zahtevom dohvatili n studenata, pri čemu i -ti student ima položeno I_i ispita, onda je ukupan broj HTTP zahteva jednak $1 + n + \sum_{i=1}^n I_i$. Konkretno, u podacima koji se nalaze u direktorijumu `primeri/javascript/asinhrono-programiranje/data/` studenti 1, 2 i 3 imaju redom $I_1 = 17$, $I_2 = 11$ i $I_3 = 18$ ispita, te je ukupan broj HTTP zahteva jednak 50. Ipak, ovaj kod je moguće prezapisati tako da se ostvari najviše 3 zahteva.

Zadatak 2.15 Izmeniti implementaciju primera `javascript/asinhrono-programiranje/callback5.html` tako da se ostvari najviše 3 HTTP zahteva. ■

Obećanja

Kako bi sprečili nastajanje pakla funkcija povratnih poziva, javila se potreba za uvođenjem naprednijih elemenata jezika JavaScript. Standardom ES6 je uveden koncept *obećanja* (engl. *promise*) koji rešava ovaj problem.

Razmotrimo koji je odnos između funkcija povratnih poziva i vrednosti koje se izračunavaju asinhrono. Da bi funkcija povratnog poziva mogla biti pozvana, morali smo da specifikujemo tu funkciju kao argument neke druge funkcije, a zatim da je pozovemo onog trenutka kada znamo da smo dobili odgovarajuću vrednost. Dakle, naša implementacija je zavisila od *mesta u kodu gde će biti poznata vrednost* koja se prosleđuje funkciji povratnog poziva.

Obećanja invertuju ovo ponašanje na sledeći način. Jedno obećanje predstavlja objekat (vrednost u širem smislu) koje će u nekom trenutku dobiti vrednost koja se asinhrono izračunava (vrednost u užem smislu). Funkcija koja zahteva tu asinhrono izračunatu vrednost se „prosleđuje” objektu obećanja i ona će biti pozvana *onda kada obećanje dobije vrednost*. Dakle, zavisnost koja je u slučaju funkcija povratnih poziva bila *prostornog tipa* (tj. morali smo da vodimo računa *u kom mestu* u kodu ćemo pozvati funkciju povratnog poziva), u slučaju obećanja je *vremenskog tipa*. Mi nećemo morati da vodimo računa gde se ta funkcija zaista poziva – obećanje će to uraditi za nas, onog trenutka kada bude dobio asinhrono izračunatu vrednost.

Konstruisanje obećanja

Pređimo sada na konkretizaciju ovog apstraktnog opisa obećanja. U programskom jeziku JavaScript nam je na raspolaganju konstruktor `Promise`. Objekti klase `Promise` predstavljaju asinhronu akciju koje mogu biti završene u nekom trenutku u budućnosti i koje će proizvesti vrednost. Obećanje ima mogućnost da obavesti bilo koga ko je zainteresovan kada je njegova vrednost dostupna.

Obećanje može biti u jednom od narednih stanja:

- *Na čekanju* (engl. *pending*) — Obećanje je u ovom stanju prilikom inicijalizacije.

- *Ispunjeno* (engl. *fulfilled*) — Obećanje prelazi u ovo stanje ukoliko se asinhrona operacija završila uspešno.
- *Odbačeno* (engl. *rejected*) — Obećanje prelazi u ovo stanje ukoliko se asinhrona operacija završila neuspešno.

Jednom kada se neko obećanje ispuni ili odbaci, ono nadalje ostaje u tom stanju.

Konstruktoru **Promise** se prosleđuje funkcija koja ima 2 argumenta (koju ćemo označavati **executor**):

1. argument predstavlja funkciju (koju ćemo označavati kao **resolve**) i ovu funkciju treba pozvati u telu funkcije **executor** onda kada nam je asinhrona vrednost dostupna. Pozivanjem funkcije **resolve** se obećanje ispunjava.
2. argument predstavlja funkciju (koju ćemo označavati kao **reject**) i ovu funkciju treba pozvati u telu funkcije **executor** onda kada postoji problem zbog kojeg asinhrona vrednost ne može biti izračunata. Pozivanjem funkcije **resolve** se obećanje odbacuje.

Prilikom poziva obe funkcije **resolve** ili **reject**, vrednost koja se prosleđuje kao argument ovih funkcija predstavlja upravo onu vrednost kojom se obećanje ispunjava ili odbacuje, redom.

Ispunjavanje obećanja

Sada se postavlja pitanje kako možemo reći obećanju da pozove neku funkciju kada bude ispunjeno. Da bismo to uradili, potrebno je da nad objektom klase **Promise** pozovemo metod **then** i kojem prosleđujemo odgovarajuću funkciju (koju ćemo označavati kao **onfulfilled**). Zanimljivo je da to možemo uraditi proizvoljan broj puta i svaka funkcija **onfulfilled** koja je prosleđena nekom pozivu metoda **then** biće pozvana asinhrono sa istom vrednošću kojom je obećanje ispunjeno. Naredni primer ilustruje korišćenje konstruktora **Promise** i poziva **then** metoda. S obzirom da u funkciji **executor** pozivamo samo funkciju **resolve**, rezultujuće obećanje će uvek biti ispunjeno vrednošću niske **'Obećanje je ispunjeno!'**.

Kod 2.6: javascript/asinhrono-programiranje/promise1.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <title>Asinhrono programiranje - obećanja</title>
6   </head>
7   <body>
8     <h3>Promise konstruktor i then</h3>
9
10    <script>
11      document.body.appendChild(document.createTextNode(" 1 "));
12
13      const promise = new Promise(function (resolve, reject) {
14        // Primitimo da se ova funkcija pozvala sinhrono sa globalnim kodom!
15        document.body.appendChild(document.createTextNode(" 2 "));
16
17        // Za sada cemo samo ispuniti obećanje
18        resolve("Obećanje je ispunjeno!");
19      });
20
21      document.body.appendChild(document.createTextNode(" 3 "));
22
23      // Da li ce poziv funkcije povratnog poziva u setTimeout
24      // biti pre ili posle poziva funkcije u .then() metodu ispod?
25      setTimeout(function () {

```

```

26     document.body.appendChild(document.createTextNode(" 4 "));
27   }, 0);
28
29   // Da bismo dohvatili vrednost iz ispunjenog obećanja,
30   // ulancavamo metodom .then() funkciju koja dobija vrednost kojom je
    obećanje ispunjeno.
31   promise.then(function (value) {
32     // Primitimo da se ova funkcija pozvala asinhrono u odnosu na globalni
        kod!
33     document.body.appendChild(document.createTextNode(value));
34   });
35   </script>
36   </body>
37 </html>

```

Važno je primetiti da se funkcija `executor` pozvala sinhrono sa celim kodom. Ovo možemo zaključiti zbog toga što se u veb pregledaču prvo prikazuje broj „2”, pa tek onda „3” (koji dolazi nakon poziva konstruktora `Promise`). Dakle, funkcija `executor` se ne dodaje u red zadataka, već se poziva direktno (tj. njen kontekst izvršavanja se direktno dodaje na vrh steka funkcija u tekućoj iteraciji petlje događaja).

Ono što dodatno primećujemo u kodu jeste poziv `setTimeout` funkcije kojim smo dodali jednu funkciju na vrh reda zadataka. Pitanje koje se ovde postavlja jeste da li će prvo biti pozvana funkcija iz poziva `setTimeout` ili iz metoda `then` u obećanju? S obzirom da se funkcijom `setTimeout` dodaje funkcija na vrh reda zadataka, a da se metodom `then` iz obećanja funkcija dodaje na vrh reda mikrozadataka, onda možemo zaključiti da će prvo biti izvršena funkcija iz obećanja.

Odbacivanje obećanja

U narednom primeru ilustrujemo šta se dešava ukoliko se u funkciji `executor` prilikom konstrukcije nekog obećanja pozove funkcija `reject`, a nad tim obećanjem je pozvan samo metod `then`.

Kod 2.7: javascript/asinhrono-programiranje/promise2.html

```

1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <meta charset="UTF-8" />
5      <title>Asinhrono programiranje - obećanja</title>
6    </head>
7    <body>
8      <h3>Promise i izuzeci prilikom odbacivanja</h3>
9
10     <script>
11       const promise = new Promise(function (resolve, reject) {
12         // Simuliramo uspesnost obećanja
13         const isSuccessful = Math.random() < 0.3;
14         if (isSuccessful) {
15           // Ispunjenje obećanja
16           resolve("Obećanje je ispunjeno!");
17         } else {
18           reject("Obećanje nije ispunjeno!");
19         }
20       });
21
22       // Funkcija koja je ulancana metodom .then() neće biti pozvana ako je
        obećanje odbaceno.
23       // Umesto toga, u konzoli ćemo videti poruku da nije obradjen slucaj.

```

```

24     promise.then(function (value) {
25         document.body.appendChild(document.createTextNode(value));
26     });
27 </script>
28 </body>
29 </html>

```

Izvršavanjem primera u veb pregledaču nekoliko puta primećujemo naredno ponašanje. U slučaju uspeha, poruka o uspehu se regularno prikazuje ispod naslova. Međutim, u nekim slučajevima ispod naslova nemamo nikakav ispis. Otvaranjem konzole u alatima za razvijanje primećujemo da u takvim slučajevima postoji greška. U pitanju je neobrađen izuzetak. Ono što možemo zaključiti jeste da, ukoliko obećanje može biti odbačeno, a takva su većina obećanja u praksi, neophodno je obraditi i taj slučaj kako bi nam se kod izvršavao korektno.

Naredni primer ilustruje kako se vrši obrađivanje slučaja kada se obećanje odbacuje. Dakle, potrebno je pozvati metod `catch` kojim se prosleđuje funkcija (koju ćemo označavati `onrejected`). Funkcija `onrejected` će biti pozvana samo ako je obećanje odbačeno.

Kod 2.8: javascript/asinhrono-programiranje/promise3.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <title>Asinhrono programiranje - obećanja</title>
6   </head>
7   <body>
8     <h3>Promise konstruktor i catch</h3>
9
10    <script>
11      const promise = new Promise(function (resolve, reject) {
12        // Simuliramo uspesnost obećanja
13        const isSuccessful = Math.random() < 0.3;
14        if (isSuccessful) {
15          // Ispunjenje obećanja
16          resolve("Obećanje je ispunjeno!");
17        } else {
18          // Odbacivanje obećanja
19          reject("Obećanje nije ispunjeno!");
20        }
21      });
22
23      // Da bismo dohvatili vrednost iz odbacenog obećanja,
24      // ulancavamo metodom .catch() funkciju koja dobija vrednost kojom je
25      // obećanje odbaceno.
26      promise
27        .then(function (value) {
28          document.body.appendChild(document.createTextNode(value));
29        })
30        .catch(function (reason) {
31          document.body.appendChild(document.createTextNode(reason));
32        });
33    </script>
34  </body>
35 </html>

```

Dodatno, primetimo da smo poziv `catch` metoda *ulančali* (engl. *chain*) uz poziv `then` metoda, umesto da smo ga pozvali nezavisno. Ulančavanje je tehnika koja potiče iz objektno-

orijentisane paradigme. Naime, ono što nismo još uvek rekli jeste da metodi `then` i `catch` kao povratnu vrednost vraćaju novo obećanje. Upravo zbog toga možemo iskoristiti ulančavanje. O naprednijem korišćenju tehnike ulančavanja će biti reči nešto kasnije.

Izuzeci u obećanjima

Prilikom izvršavanja koda u funkciji `executor` može doći do ispaljivanja izuzetaka. Naredni primer ilustruje pokušaj da se izuzeci koji se ispaljuju u funkciji `executor` uhvate i obrade pomoću `try/catch` blokova.

Kod 2.9: javascript/asinhrono-programiranje/promise4.html

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <title>Asinhrono programiranje - obećanja</title>
6   </head>
7   <body>
8     <h3>Promise i obrada izuzetaka</h3>
9
10    <script>
11      const promise = new Promise(function (resolve, reject) {
12        throw new Error("Doslo je do greske");
13      });
14
15      // Izuzeci u obećanjima se ne mogu uhvatiti try/catch blokom,
16      // zato sto oni hvataju "sinhrone izuzetke".
17      try {
18        promise.then(function (value) {
19          document.body.appendChild(document.createTextNode(value));
20        });
21      } catch (err) {
22        document.body.appendChild(document.createTextNode(err));
23      }
24    </script>
25  </body>
26 </html>
```

Izvršavanjem primera u veb pregledaču, i otvaranjem konzole, primećujemo da je prijavljen neobrađen izuzetak. Dakle, zaključujemo da na ovaj način nije moguće vršiti obradu izuzetaka u obećanjima.

Na sreću, ukoliko dođe do izuzetka prilikom izvršavanja funkcije `executor`, JavaScript automatski odbacuje obećanje koje se konstruiše. Vrednost kojom se obećanje odbacuje jeste upravo objekat izuzetka koji je ispaljen. Dakle, možemo zaključiti da bi obrađivanje izuzetaka u obećanjima trebalo raditi pomoću `catch` metoda. Naredni primer ilustruje da je naš zaključak korektan.

Kod 2.10: javascript/asinhrono-programiranje/promise5.html

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <title>Asinhrono programiranje - obećanja</title>
6   </head>
7   <body>
8     <h3>Promise i obrada izuzetaka</h3>
9
10    <script>
```

```

11     const promise = new Promise(function (resolve, reject) {
12         throw new Error("Doslo je do greske");
13     });
14
15     // Ulančavanje metodom .catch() se koristi i za obradu izuzetaka koje
16     // nastaju u obećanju.
17     promise
18     .then(function (value) {
19         document.body.appendChild(document.createTextNode(value));
20     })
21     .catch(function (reason) {
22         document.body.appendChild(document.createTextNode(reason));
23     });
24 </script>
25 </body>
26 </html>

```

Ulančavanje više metoda

Kao što smo rekli, metodi `then` i `catch` vraćaju obećanja, te ih možemo ulančavati na proizvoljne načine. Sada se prirodno postavlja pitanje kojim vrednostima se ova obećanja, koja se vraćaju kao povratne vrednosti, ispunjavaju (ili odbacuju?). Odgovor je da će ova obećanja uvek biti ispunjena povratnom vrednošću funkcije `onfulfilled`, osim ukoliko se ne ispali izuzetak prilikom izračunavanja te funkcije. U tom slučaju će obećanje biti odbačeno objektom izuzetka. Naredni primer ilustruje lanac poziva `then` metoda.

Kod 2.11: javascript/asinhrono-programiranje/promise6.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <title>Asinhrono programiranje - obećanja</title>
6   </head>
7   <body>
8     <h3>Promise i ulančavanje vise then metoda</h3>
9
10    <script>
11      const promise = new Promise(function (resolve, reject) {
12        resolve(0);
13      });
14
15      // Ako funkcija ulancana metodom .then() vrati neku vrednost,
16      // onda ce sam metod .then() vratiti obećanje na koje mozemo ulancati
17      // drugu funkciju.
18      promise
19      .then(function (value) {
20        document.body.appendChild(document.createTextNode(value + ","));
21        return value + 1;
22      })
23      .then(function (value) {
24        document.body.appendChild(document.createTextNode(value + ","));
25        return value + 1;
26      })
27      .then(function (value) {
28        document.body.appendChild(document.createTextNode(value + ","));
29      })
30      // I ovde je moguće vrsiti ulančavanje, samo sto ce funkciji biti
31      // prosledjena vrednost undefined.
32      .then(function (value) {
33        document.body.appendChild(document.createTextNode(value));
34      });
35    </script>
36  </body>
37 </html>

```

```

32     })
33     .catch(function (reason) {
34         document.body.appendChild(
35             document.createTextNode("--> catch: " + reason)
36         );
37     });
38 </script>
39 </body>
40 </html>

```

Ukoliko u bilo kom delu lanca poziva `then` metoda bude ispaljen izuzetak, onda se lanac `then` poziva prekida i izvršavanje se nastavlja u funkciji `onrejected` koja je specifikovana *prvim ulančanim* pozivom metoda `catch`. Naredni primer ilustruje prekid lanca poziva `then` metoda.

Kod 2.12: javascript/asinhrono-programiranje/promise7.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <title>Asinhrono programiranje - obećanja</title>
6   </head>
7   <body>
8     <h3>Promise i obrada izuzetaka u slucaju vise then metoda</h3>
9
10    <script>
11      const promise = new Promise(function (resolve, reject) {
12        resolve(0);
13      });
14
15      // Ako funkcija ulancana metodom .then() vrati neku vrednost,
16      // onda ce sam metod .then() vratiti obećanje na koje mozemo ulancati
17      // drugu funkciju.
18      promise
19        .then(function (value) {
20          if (value > 0) {
21            throw new Error("Vrednost je veka od 0");
22          }
23          document.body.appendChild(document.createTextNode(value + ","));
24          return value + 1;
25        })
26        .then(function (value) {
27          if (value > 0) {
28            throw new Error("Vrednost je veka od 0");
29          }
30          document.body.appendChild(document.createTextNode(value + ","));
31          return value + 1;
32        })
33        .then(function (value) {
34          if (value > 0) {
35            throw new Error("Vrednost je veka od 0");
36          }
37          document.body.appendChild(document.createTextNode(value + ","));
38        })
39        .catch(function (reason) {
40          document.body.appendChild(
41            document.createTextNode("--> catch: " + reason)
42          );
43        });
44    </script>

```

```
44 </body>
45 </html>
```

Ovo ponašanje omogućava pisanje vrlo kvalitetnog koda, koji sa jedne strane jasno opisuje efekat koji kod postiže, a sa druge strane omogućava vrlo naprednu obradu grešaka, oporavak od grešaka i debugovanje.

Obećanja i XMLHttpRequest

Naredni primer ilustruje kako možemo kombinovati stečeno znanje o obećanjima da bismo kreirali HTTP zahteve čiji odgovori se asinhrono obrađuju. Funkcija `httpGet` koju treba napisati za slanje HTTP GET zahteva više ne prihvata funkciju povratnog poziva kao argument. Umesto toga, ona kreira obećanje koje se vraća kao povratna vrednost. Funkcija `executor` ovog obećanja ima već poznati kod, sa jednom izmenom – umesto da se poziva funkcija povratnog poziva sa 2 argumenta, u slučaju uspeha se obećanje ispunjava pozivom funkcije `resolve`, a u slučaju neuspeha se obećanje odbacuje pozivom funkcije `reject`.

Primer takođe ilustruje i upotrebu ove funkcije za dohvaćanje podataka o studentima i prikazivanje tih podataka, kao i obradu grešaka.

Kod 2.13: javascript/asinhrono-programiranje/promise8.html

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <title>Asinhrono programiranje – funkcije povratnog poziva</title>
6   </head>
7   <body>
8     <h3>Studenti:</h3>
9
10    <script>
11      // Funkcija httpGet vise ne prihvata funkciju povratnog poziva,
12      // vec vraca obećanje koje ce ili biti ispunjeno vrednoscu iz
13      // odgovarajuće datoteke
14      // ili ce ispaliti izuzetak.
15      function httpGet(url) {
16        return new Promise(function (resolve, reject) {
17          const httpReq = new XMLHttpRequest();
18          httpReq.onreadystatechange = function () {
19            if (httpReq.readyState == 4) {
20              if (httpReq.status == 200) {
21                const data = JSON.parse(httpReq.responseText);
22                // Ispunjenje obećanja
23                resolve(data);
24              } else {
25                const err = new Error(httpReq.statusText);
26                // Odbacivanje obećanja
27                reject(err);
28              }
29            }
30          };
31          httpReq.open("GET", url, true);
32          httpReq.send();
33        });
34
35        httpGet("./data/students.json")
36          .then(function (students) {
37            for (const student of students) {
```



```

38     const node = document.createElement("p");
39     node.appendChild(
40         document.createTextNode(
41             `${student.name} ${student.surname} (${student.index})`
42         )
43     );
44     document.body.appendChild(node);
45 }
46 })
47 .catch(function (err) {
48     console.log("Doslo je do greske: " + err.message);
49 });
50 </script>
51 </body>
52 </html>

```

Obećanja i Fetch API

Pored konstruktora `XMLHttpRequest`, veb API sadrži i podskup interfejsa koji se naziva Fetch API. Poput `XMLHttpRequest`, i ovaj API služi za kreiranje HTTP zahteva. Važna razlika u odnosu na `XMLHttpRequest` je u tome što Fetch API koristi objektno-orijentisane tehnike i zasnovan je na obećanjima. U osnovi Fetch API-ja se nalazi funkcija `fetch`. U sekciji 2.8.3 biće više reči o ovom API-ju. Funkcija `fetch` prihvata URL do resursa koji se dohvata i vraća obećanje koje se ispunjava odgovorom od servera. Odgovor je predstavljen objektom klase `Response` koji ima na raspolaganju metod `json` za deserijalizaciju odgovora iz JSON formata.

Za sada navodimo primer koji koristi Fetch API za implementiranje zahteva iz prethodnog primera.

Kod 2.14: javascript/asinhrono-programiranje/promise9.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <title>Asinhrono programiranje - funkcije povratnog poziva</title>
6   </head>
7   <body>
8     <h3>Studenti:</h3>
9
10    <script>
11      function httpGet(url) {
12        // Umesto da koristimo XMLHttpRequest, veb pregledaci implementiraju
13        // tzv. Fetch API.
14        // Na raspolaganju nam je funkcija fetch() kojom mozemo slati HTTP
15        // zahteve.
16        // Za razliku od XMLHttpRequest objekata, funkcija fetch() podrzava rad
17        // sa obećanjima.
18        return fetch(url).then(function (response) {
19          // Povratna vrednost funkcije fetch() je obećanje koje ce biti
20          // ispunjeno objektom klase Response.
21          // Ovaj objekat ima na raspolaganju metod .json() kojim se vrši
22          // deserijalizacija HTTP odgovora.
23          // Metod .json() takodje vraca obećanje, tako da mozemo ulancati then
24          // () metodom,
25          // kao sto je prikazano ispod.
26          return response.json();
27        });
28      }
29    }

```

```

23
24     httpGet("./data/students.json")
25     .then(function (students) {
26         for (const student of students) {
27             const node = document.createElement("p");
28             node.appendChild(
29                 document.createTextNode(
30                     `${student.name} ${student.surname} (${student.index})`
31                 )
32             );
33             document.body.appendChild(node);
34         }
35     })
36     .catch(function (err) {
37         console.log("Dosló je do greske: " + err.message);
38     });
39 </script>
40 </body>
41 </html>

```

Primitimo da nema razlike u pozivu različitih implementacija iz ovog i prethodnog primera.

Zavisnost između više obećanja

U nekim implementacijama će biti neophodno da koristimo podatke koje dobijamo iz različitih obećanja. Ovo može izgledati kao potencijalno problematična stvar za implementirati, ali videćemo da postoji nekoliko tehnika kojima je ovo moguće ostvariti kroz zadatak prikazivanja ispita za studente (iz primera `javascript/asinhrono-programiranje/callback4.html`). neke od mogućih tehnika su:

- *Ugnežđavanje poziva metoda `then`*. Ova tehnika podrazumeva da se prilikom obrade vrednosti iz jednog obećanja, ugnezdi obrada drugog obećanja. Ovaj pristup, iako jednostavan, dovodi do toga da i ugneždjena obrada obećanja mora da sadrži svoj poziv `catch` metoda za obradu grešaka. Dodatno, veliki broj ugnežđavanja dovodi do pojave slične paklu funkcija povratnih poziva. Naredni primer ilustruje ovu tehniku.

Kod 2.15: `javascript/asinhrono-programiranje/promise10.html`

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <title>Asinhrono programiranje - funkcije povratnog poziva</title>
6   </head>
7   <body>
8     <h3>Studenti i ispiti (ugnezđjavanje .then() poziva):</h3>
9
10    <p>
11      U ovom primeru je neophodno da za svakog studenta dohvatimo njegove
12      ispite.<br />
13      Ovo mozemo implementirati na vise nacina.
14    </p>
15
16    <script>
17      function httpGet(url) {
18        return fetch(url).then(function (response) {
19          return response.json();
20        });
21      }

```

```

22
23     httpGet("./data/students.json")
24     .then(function (students) {
25         for (const student of students) {
26             const node = document.createElement("p");
27             node.appendChild(
28                 document.createTextNode(
29                     `${student.name} ${student.surname} (${student.index})`
30                 )
31             );
32             document.body.appendChild(node);
33
34             const examNodeList = document.createElement("ul");
35             document.body.appendChild(examNodeList);
36
37             // Ugnezdjavanje obrade drugog obecanja.
38             // Ovaj pristup podseca na pakao funkcija povratnog poziva.
39             // Kao i u tom primeru, primetimo da se ovde generise "veliki
40             // " broj nepotrebnih HTTP zahteva
41             // (za svakog studenta dohvatamo podatke o svim ispitima
42             // iznova).
43             // Naravno, ovo se moze ispraviti u 2 HTTP zahteva. Kako?
44             httpGet("./data/exams.json").then(function (exams) {
45                 // Problem sa ovim pristupom jeste sto donji .catch() nece
46                 // uhvatiti izuzetak
47                 // koji se mozda javi u funkciji koja se prosledjuje ovoj .
48                 // then() metodi.
49                 // tako da bi ispravnija implementacija podrazumevala da
50                 // postoji jos jedan .catch() poziv
51                 // za obecanje koje vrati poziv httpGet() ispod.
52
53                 // Otkomentarisati narednu liniju kako bi se u konzoli
54                 // prikazao neobradjen izuzetak.
55                 // throw new Error("Ovaj izuzetak nece biti uhvacen .catch
56                 // () metodom ispod");
57
58                 for (const exam of exams) {
59                     if (exam.studentId !== student.id) {
60                         continue;
61                     }
62
63                     const examNode = document.createElement("li");
64                     examNode.appendChild(
65                         document.createTextNode(
66                             `${exam.final} ${exam.year} - grade: ${exam.grade},
67                             status: ${exam.status}`
68                         )
69                     );
70                     examNodeList.appendChild(examNode);
71                 }
72             });
73         }
74     });
75     .catch(function (err) {
76         console.log("Doslo je do greske: " + err.message);
77     });
78 }
79 </script>
80 </body>
81 </html>

```

- Ulančavanje poziva metoda `then`. Ova tehnika podrazumeva da se, prilikom obra-

de vrednosti u prvom obećanju kreira novo obećanje koje će biti ispunjeno nizom sa dve vrednosti – prva vrednost je vrednost iz prvog obećanja, a druga vrednost je vrednost iz drugog obećanja. Ovde je neophodno koristiti niz, s obzirom da se obećanje ispunjava tačno jednom vrednošću. Ova tehnika predstavlja unapređenje prethodne tehnike s obzirom da nemamo kod koji liči na pakao funkcija povratnih poziva. Međutim, uvodi dodatnu kompleksnost u ručnom kreiranju „međuobećanja” koje se ispunjava nizom vrednosti. Dodatno, ne rešava problem sa dupliciranim `catch` pozivima. Naredni primer ilustruje ovu tehniku.

Kod 2.16: javascript/asinhrono-programiranje/promise11.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <title>Asinhrono programiranje - funkcije povratnog poziva</title>
6   </head>
7   <body>
8     <h3>Studenti i ispiti (ulancavanje .then() poziva):</h3>
9
10    <p>
11      U ovom primeru je neophodno da za svakog studenta dohvatimo njegove
12      ispite.<br />
13      Ovo mozemo implementirati na vise nacina.
14    </p>
15
16    <script>
17      function httpGet(url) {
18        return fetch(url).then(function (response) {
19          return response.json();
20        });
21      }
22
23      // U ovom pristupu imamo tacno 2 HTTP poziva, sto je znacajno
24      // unapredjenje u odnosu na prethodni primer.
25      httpGet("./data/students.json")
26        .then(function (students) {
27          return new Promise(function (resolve, reject) {
28            // Problem sa ovim pristupom jeste sto donji .catch() nece
29            // uhvatiti izuzetak
30            // koji se mozda javi u funkciji koja se prosledjuje ovoj .
31            // then() metodi.
32            // tako da bi ispravna implementacija podrazumevala da
33            // postoji jos jedan .catch() poziv
34            // za obećanje koje vrati poziv httpGet() ispod.
35            httpGet("./data/exams.json").then(function (exams) {
36              // Otkomentarisati narednu liniju kako bi se u konzoli
37              // prikazao neobradjen izuzetak.
38              // throw new Error("Ovaj izuzetak nece biti uhvacen .catch
39              // () metodom ispod");
40
41              resolve([students, exams]);
42            });
43          });
44        })
45        .then(function ([students, exams]) {
46          for (const student of students) {
47            const node = document.createElement("p");
48            node.appendChild(
49              document.createTextNode(

```

```

44         `${student.name} ${student.surname} (${student.index})`
45     )
46 );
47 document.body.appendChild(node);
48
49 const examNodeList = document.createElement("ul");
50 document.body.appendChild(examNodeList);
51
52 for (const exam of exams) {
53     if (exam.studentId !== student.id) {
54         continue;
55     }
56
57     const examNode = document.createElement("li");
58     examNode.appendChild(
59         document.createTextNode(
60             `${exam.final} ${exam.year} - grade: ${exam.grade},
              status: ${exam.status}`
61         )
62     );
63     examNodeList.appendChild(examNode);
64 }
65 }
66 })
67 .catch(function (err) {
68     console.log("Dosló je do greske: " + err.message);
69 });
70 </script>
71 </body>
72 </html>

```

- *Korišćenje niza obećanja.* Ova tehnika podrazumeva da se sva obećanja koja su zavisna jedna od drugih smeste u jedan niz. Taj niz se zatim prosledi kao argument funkciji `Promise.all` koja kreira obećanje koje će biti ispunjeno samo ako se sva pojedinačna obećanja iz prosleđenog niza ispune. Velika prednost ove tehnike jeste u tome što nije važan redosled kojim se obećanja iz niza ispunjavaju – obećanje koje se dobija kao povratna vrednost funkcije `Promise.all` će biti ispunjeno tek onda kada se poslednje obećanje iz niza ispuni, nezavisno od toga koje je bilo poslednje. Vrednost kojom se obećanje ispunjava jeste niz vrednosti kojim su ispunjena pojedinačna obećanja, pri čemu je *i*-ti element u rezultujućem nizu vrednost kojim je ispunjeno *i*-to obećanje iz prosleđenog niza. Dodatno, ukoliko je makar jedno obećanje iz prosleđenog niza odbačeno, onda se i obećanje dobijeno pozivom funkcije `Promise.all` odbacuje. Ovime se osiguravamo da ćemo ili dobiti sve neophodne podatke ili grešku. Očigledno, ova tehnika rešava sve probleme prethodnih tehnika, te se zbog toga i preporučuje za korišćenje. Naredni primer ilustruje ovu tehniku.

Kod 2.17: javascript/asinhrono-programiranje/promise12.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <title>Asinhrono programiranje - funkcije povratnog poziva</title>
6   </head>
7   <body>
8     <h3>Studenti i ispiti (kolekcije obećanja):</h3>
9
10    <p>
11      U ovom primeru je neophodno da za svakog studenta dohvatimo njegove

```

```
12     ispite.<br />
13     Ovo mozemo implementirati na vise nacina.
14 </p>
15
16 <script>
17     function httpGet(url) {
18         return fetch(url).then(function (response) {
19             return response.json();
20         });
21     }
22
23     // Ovaj pristup resava sve probleme prethodna dva primera.
24     // Gresku mozemo simulirati promenom naziva jedne od datoteka ispod
25     .
26     const promises = [
27         httpGet("./data/students.json"),
28         httpGet("./data/exams.json"),
29     ];
30     // Metod Promise.all() vraća obećanje koje će biti ispunjeno samo
31     // ako je svako obećanje u nizu ispunjeno.
32     // Ako je makar jedno obećanje odbaceno, onda će i Promise.all()
33     // obećanje takodje biti odbaceno.
34     // Ako je ispunjeno, onda će biti prosledjen niz vrednosti kojima
35     // su ispunjena obećanja iz datog niza.
36     Promise.all(promises)
37     .then(function ([students, exams]) {
38         for (const student of students) {
39             const node = document.createElement("p");
40             node.appendChild(
41                 document.createTextNode(
42                     `${student.name} ${student.surname} (${student.index})`
43                 )
44             );
45             document.body.appendChild(node);
46
47             const examNodeList = document.createElement("ul");
48             document.body.appendChild(examNodeList);
49
50             for (const exam of exams) {
51                 if (exam.studentId !== student.id) {
52                     continue;
53                 }
54
55                 const examNode = document.createElement("li");
56                 examNode.appendChild(
57                     document.createTextNode(
58                         `${exam.final} ${exam.year} - grade: ${exam.grade},
59                         status: ${exam.status}`
60                     )
61                 );
62                 examNodeList.appendChild(examNode);
63             }
64         }
65     })
66     .catch(function (err) {
67         console.log("Dosló je do greske: " + err.message);
68     });
69 </script>
70 </body>
71 </html>
```

Naravno, postoje i druge, složenije tehnike o kojima neće biti reči u tekstu.

Zadatak 2.16 Implementirati zahtev iz primera javascript/asinhrono-programiranje/callback5.html korišćenjem tehnika zasnovanih na obećanjima umesto funkcija povratnih poziva. ■

Asinhrone funkcije

Sada kada smo upoznati sa popularnim načinom za upravljanje asinhronim kodom pomoću obećanja, možemo nadgraditi ovaj šablon rada nekim novinama u JavaScript jeziku. ECMAScript2017 standard daje način za pisanje *asinhronih funkcija* tako da izgledaju kao sinhronne funkcije, ali naravno, ponašaju se asinhrono, pomoću ključnih reči `async` i `await`. Pogledajmo naredni primer.

Kod 2.18: javascript/asinhrono-programiranje/async1.html

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <title>Asinhrono programiranje - asinhrone funkcije</title>
6   </head>
7   <body>
8     <h3>async/await</h3>
9
10    <script>
11      // Asinhron kod koji koristi obećanja mozemo uciniti "sinhronim" u 2
12      // koraka:
13      // 1) Potrebna nam je funkcija koja je oznacena kljucnom recju async.
14      // Ovo je neophodno zato sto kljucna rec await (koja se koristi ispod)
15      // ne moze se koristiti u globalnom opsegu u veb pregledacu (osim ako
16      // nije u modulu).
17      async function doStuff() {
18        const promise = new Promise(function (resolve, reject) {
19          resolve("Obećanje je ispunjeno!");
20        });
21
22        // 2) Ispred obećanja dodajemo kljucnu rec await koja ce "sacekati" da
23        // se obećanje ispuni.
24        // Vrednost promenljive value ispod ce biti vrednost kojom je obećanje
25        // ispunjeno.
26        const value = await promise;
27        // Kod "ispod" await kljucne reci ce se izvršiti asinhrono.
28        document.body.appendChild(document.createTextNode(value));
29      }
30
31      doStuff();
32    </script>
33  </body>
34 </html>
```

Kada funkciju označimo ključnom reči `async`, funkcija će uvek vraćati obećanje, koje može biti ispunjeno nekom vrednošću, dok `await` suspenduje izvršavanje asinhrone funkcije sve dok se obećanje koje ga prati nije završeno.

Napomenimo da ključna reč `await` radi samo za obećanjima, što znači da će konvertovati vrednost u obećanje ako ta vrednost to već nije. Ključnu reč `async` možemo koristiti i u funkcijama kao vrednostima, definicijama metoda i lambda funkcijama.

Objasnimo nešto detaljnije rad ključnom reči `await`. Operator `await` čeka na operand, odnosno, na obećanje da se ispuni ili odbaci — u prvom slučaju njegova vrednost predstavlja vrednost ispunjenog obećanja, a u drugom slučaju ispaljuje izuzetak čija je vrednost vrednost odbačenog izuetka. Da li ovo znači da možemo koristiti `try/catch`-blokove za rad sa obećanjima? Odgovor je potvrđan, što ilustruje naredni primer.

Kod 2.19: javascript/asinhrono-programiranje/async2.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <title>Asinhrono programiranje - asinhronne funkcije</title>
6   </head>
7   <body>
8     <h3>Obrada izuzetaka u asinhronim funkcijama</h3>
9
10    <script>
11      async function doStuff() {
12        // Glavna prednost koriscenja funkcija jeste u obradi izuzetaka.
13        // U ovom primeru vidimo da kljucna rec await radi korektno i ispred
14          poziva funkcije koja vraca obećanje.
15        function getPromise() {
16          return new Promise(function (resolve, reject) {
17            throw new Error("Obećanje nije ispunjeno!");
18          });
19        }
20
21        // Kod ispod izgleda kao svaki drugi sinhron kod (pa cak i obrada
22          gresaka!),
23        // samo sto se zapravo izvrsava asinhrono.
24        try {
25          const value = await getPromise();
26          document.body.appendChild(document.createTextNode(value));
27        } catch (err) {
28          document.body.appendChild(document.createTextNode(err));
29        }
30      }
31    </script>
32  </body>
33 </html>

```

Uvođenje asinhronih funkcija u JavaScript je značajno unapredilo pisanje asinhronog koda. Naredni primer ilustruje implementaciju zahteva iz primera `javascript/asinhrono-programiranje/callback3.html` i `javascript/asinhrono-programiranje/promise9.html` u kojem se dohvataju podaci putem HTTP GET zahteva, a zatim se asinhrono, po dohvatanju tih podataka, prikazuju u veb pregledaču. Kod izgleda kao da se sve izvršava sinhrono, iako to nije slučaj.

Kod 2.20: javascript/asinhrono-programiranje/async3.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <title>Asinhrono programiranje - asinhronne funkcije</title>
6   </head>
7   <body>

```



```

8     <h3>Studenti:</h3>
9
10    <script>
11        function httpGet(url) {
12            return fetch(url).then(function (response) {
13                return response.json();
14            });
15        }
16
17        // Asinhrone funkcije cine pisanje asinhronog koda neverovatno
18        // jednostavnim.
19        async function doStuff() {
20            try {
21                const students = await httpGet("./data/students.json");
22
23                for (const student of students) {
24                    const node = document.createElement("p");
25                    node.appendChild(
26                        document.createTextNode(
27                            `${student.name} ${student.surname} (${student.index})`
28                        )
29                    );
30                    document.body.appendChild(node);
31                }
32            } catch (err) {
33                console.log("Dosló je do greske: " + err.message);
34            }
35
36            doStuff();
37        }
38    </script>
39 </body>
</html>

```

Korišćenje asinhronih funkcija u slučaju kada više obećanja zavise jedni od drugih je trivijalno, kao što naredni primer ilustruje.

Kod 2.21: javascript/asinhrono-programiranje/async4.html

```

1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <meta charset="UTF-8" />
5      <title>Asinhrono programiranje - asinhrone funkcije</title>
6    </head>
7    <body>
8      <h3>Studenti i ispiti:</h3>
9
10     <script>
11         function httpGet(url) {
12             return fetch(url).then(function (response) {
13                 return response.json();
14             });
15         }
16
17         // Primetite koliko je kod ispod jednostavniji od svih pristupa koje smo
18         // videli do sada,
19         // pri čemu je očuvana efikasnost (2 HTTP poziva).
20         async function doStuff() {
21             try {
22                 const students = await httpGet("./data/students.json");

```

```

22     const exams = await httpGet("./data/exams.json");
23
24     for (const student of students) {
25         const node = document.createElement("p");
26         node.appendChild(
27             document.createTextNode(
28                 `${student.name} ${student.surname} (${student.index})`
29             )
30         );
31         document.body.appendChild(node);
32
33         const examNodeList = document.createElement("ul");
34         document.body.appendChild(examNodeList);
35
36         for (const exam of exams) {
37             if (exam.studentId !== student.id) {
38                 continue;
39             }
40
41             const examNode = document.createElement("li");
42             examNode.appendChild(
43                 document.createTextNode(
44                     `${exam.final} ${exam.year} – grade: ${exam.grade}, status: $
45                         {exam.status}`
46                 )
47             );
48             examNodeList.appendChild(examNode);
49         }
50     } catch (err) {
51         console.log("Dosló je do greske: " + err.message);
52     }
53 }
54
55 doStuff();
56 </script>
57 </body>
58 </html>

```

Asinhronne funkcije su izuzetno moćan alat za pisanje asinhronog koda koji je visokog kvaliteta, jednostavan za čitanje, održavanje i debugovanje. Ipak, ne treba zaboraviti da postoje složeni koncepti koji stoje u pozadini ovog alata, a koje treba razumeti ukoliko kreiramo asinhronne aplikacije.

Zadatak 2.17 Implementirati zahtev iz primera `javascript/asinhrono-programiranje/callback5.html` korišćenjem tehnika zasnovanih na asinhronim funkcijama umesto funkcija povratnih poziva. ■

2.8.3 Interfejsi veb pregledača za ostvarivanje asinhronih zahteva

Sada kada smo razumeli asinhronne koncepte u JavaScript jeziku, konačno možemo da pređemo na temu konstruisanja asinhronih zahteva ka resursima, bilo lokalnim, bilo udaljenim. Ovo je moguće uraditi na više načina, a mi ćemo prikazati neke od najosnovnijih.

XMLHttpRequest objekat

Da bismo poslali HTTP zahtev, potrebno je kreirati objekat klase `XMLHttpRequest`, otvoriti url ka resursu, i poslati zahtev. Nakon što se transakcija završi, objekat će sadržati korisne informacije poput tela HTTP odgovora i statusnog koda. Ovaj objekat prolazi kroz razna

stanja, kao što je "otvorena konekcija", "finalno stanje" i dr. Svako stanje ima svoj kod.

Slanje zahteva

Kao što smo rekli, prvo je potrebno kreirati objekat, koji se inicijalizuje u stanje "UNSET" (kod je 0):

```
1 let xhr = new XMLHttpRequest();
```

Zatim je potrebno formirati HTTP zahtev pozivom metoda `open`. Njegovi argumenti su: (1) HTTP metod, (2) URL servera, (3) Bulova vrednost koja označava da li se zahtev vrši sinhrono ili asinhrono (podrazumevana vrednost je `true`, što označava asinhronost). Metod ima i dodatne argumente za korisničko ime i lozinku.

```
1 xhr.open('GET', 'http://api.icndb.com/jokes/random');
```

U ovom trenutku se mogu postaviti zaglavlja HTTP zahteva pomocu `setRequestHeader` metode, na primer, `xhr.setRequestHeader(imePolja, vrednost)`. Zatim je potrebno dodati funkcije povratnog poziva koje će vršiti nekakvu obradu pristiglog odgovora ili, ono što se često radi, obradu pri promeni stanja asinhronog zahteva. Odgovor zahteva je sadržan u svojstvima `response`, `responseText` ili `responseXML` u zavisnosti od tipa odgovora. U slučaju greške, svojstvo `statusText` sadrži statusnu poruku koja odgovara HTTP statusnoj poruci. Na primer:

```
1 xhr.onload = () => {
2     // Proveravamo da li je odgovor servera sa vrednoscu 200 preko polja
      zahteva status
3     if (xhr.status === 200) {
4         console.log(xhr.response);
5     }
6     else {
7         console.log(new Error(xhr.statusText));
8     }
9 };
```

Događaj `onload` se izvršava kada objekat pređe u finalno stanje "DONE" (kod je 4). Događaj `onerror` se izvršava kada dođe do greske prilikom zahteva:

```
1 xhr.onerror = () => console.log(new Error('Problem prilikom slanja zahteva'));
```

Alternativa je moguća sa `onreadystatechange` osluskivacem koji će pozvati funkciju povratnog poziva prilikom svake promene stanja:

```
1 xhr.onreadystatechange = () => {
2     switch(xhr.readyState)
3     {
4         case XMLHttpRequest.DONE:
5             if (xhr.status === 200) {
6                 console.log(xhr.response);
7             }
8             else {
9                 console.log(new Error(xhr.statusText));
10            }
11        }
12    }
```

Konačno, potrebno je poslati HTTP zahtev, što se vrši metodom `send`. U zavisnosti od tipa zahteva, opcioni argument metode `send` predstavlja telo zahteva.

```
1 xhr.send();
```

Pristup zasnovan na obećanjima

Da bismo zamenili pristup zasnovan na funkcijama povratnog poziva pristupom zasnovanim na obećanjima, ono što je dovoljno uraditi jeste obmotati `XMLHttpRequest` objekat u obećanje i izmeniti funkcije povratnog poziva za `onload` i `onerror` metode. U tu svrhu, konstruisaćemo jednostavnu klasu koja će predstavljati jedan HTTP zahtev.

Kod 2.22: javascript/promiseXmlhttprequest.js

```

1 class ZahtevError extends Error {}
2
3 class Zahtev
4 {
5     constructor(metod, url, telo = null, zaglavlja = null)
6     {
7         this.metod = metod;
8         this.url = url;
9         this.telo = telo;
10        this.zaglavlja = zaglavlja;
11    }
12
13    posaljiZahtev()
14    {
15        return new Promise((ispuni, odbaci) => {
16            let xhr = new XMLHttpRequest();
17            xhr.open(this.metod, this.url);
18
19            (this.zaglavlja || []).forEach(zaglavlje => {
20                xhr.setRequestHeader(zaglavlje.ime, zaglavlje.vrednost);
21            });
22
23            xhr.onload = () => {
24                if (xhr.status === 200) {
25                    ispuni(xhr.response);
26                } else {
27                    odbaci(new ZahtevError(xhr.statusText));
28                }
29            };
30            xhr.onerror = () => odbaci(new ZahtevError(xhr.statusText));
31
32            xhr.send(this.telo);
33        });
34    }
35 }
36
37 let zahtev1 = new Zahtev("GET", "http://api.icndb.com/jokes/random");
38
39 zahtev1.posaljiZahtev()
40     .then((rezultat) => {
41         console.log(JSON.parse(rezultat).value.joke);
42         // It is scientifically impossible for Chuck Norris to have had a
43           mortal father. The most popular theory is that he went back in time
44           and fathered himself.
45     })
46     .catch((err) => {
47         console.log(err.message);
48     });

```

Fetch API

Fetch API omogućava korišćenje JavaScript interfejsa koji služi za pristupanje i upravljanje delovima HTTP toka, kao što su zahtevi i odgovori. U osnovi ovog API-ja leži globalna

funkcija `fetch` kojom se na jednostavan način udaljeni resursi dohvataju preko mreže na asinhroni način, a koja već koristi prednosti rada sa obećanjima.

Iako se ovo može postići pomoću tipa `XMLHttpRequest`, funkcija `fetch` predstavlja bolju alternativu, što se može videti kroz neke njene osobine:

- Jednostavnost za korišćenje, pogotovo kao deo drugih API-ja (kao što je, na primer, Service Workers API).
- Obećanje koje funkcija `fetch` vraća neće biti odbačeno ukoliko se od servera dobije statusni kod koji označava grešku, kao što je 404 ili 500, već se odbacuje samo ukoliko dođe u problema sa mrežnom komunikacijom ili do nečega što onemogućuje uspešno završavanje zahteva.
- Kredencijali (kolačići, TLS klijentski sertifikati, autentikacione pristupnice za HTTP¹⁴) podrazumevano bivaju poslani samo ukoliko se zahtev šalje sa istog izvora kao i server¹⁵.

Naredni jednostavan primer ilustruje osnovno korišćenje funkcije `fetch`:

```
1 fetch('https://jsonplaceholder.typicode.com/todos/7')
2   .then(function(response) {
3     return response.json();
4   })
5   .then(function(myJson) {
6     console.log('Fetch 1:', JSON.stringify(myJson));
7   });
```

Napomenimo da je Fetch API namenjen za korišćenje u klijentskim aplikacijama, tako da je potrebno pokrenuti prethodni primer u veb pregledaču. Na primer, dovoljno je napraviti narednu html datoteku i pokrenuti je u nekom veb pregledaču:

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   <meta http-equiv="X-UA-Compatible" content="ie=edge">
7   <title>Fetch API</title>
8 </head>
9 <body>
10  <h1>Otvoriti JavaScript konzolu</h1>
11  <script src="fetch1.js"></script>
12 </body>
13 </html>
```

U skladu sa ovom napomenom, podrazumevaćemo da se svi naredni JavaScript kodovi izvršavaju u okviru veb pregledača¹⁶.

Funkcija `fetch`

Kao što je vidljivo iz prethodnog primera, najjednostavniji način za korišćenje funkcije `fetch` jeste njeno pozivanje sa niskom-argumentom koja sadrži URI ka udaljenom resursu. U tom slučaju se podrazumevano šalje `GET` metod za dohvaćanje podataka.

¹⁴Torke korisničkog imena, lozinki i izvora koje se koriste za HTTP autentikaciju i pridruženi su jednom zahtevu ili više njih.

¹⁵Originalno ponašanje je bilo da se kredencijali nikad ne šalju implicitno, ali je to izmenjeno (videti <https://github.com/whatwg/fetch/pull/585>).

¹⁶Postoje i implementacije ovog API-ja u vidu polifila (<https://github.com/github/fetch>) ili drugih implementacija (<https://www.npmjs.com/package/node-fetch>).

Puna deklaracija ove funkcije data je u nastavku:

```
1 Promise<Response> fetch(input[, init]);
```

Iz ovoga možemo videti sledeće:

- Parametri funkcije su:
 - `input` — Definiše resurs ka kojem želimo da izvršimo asinhroni HTTP zahtev. Ovaj parametar može biti niska koja sadrži URI resursa ili objekat tipa `Request`.
 - `init` — Opcioni parametar koji predstavlja objekat sa dodatnim podešavanjima koja želimo da budu primenjena na zahtev koji se šalje. Neka podešavanja koja izdvajamo su:
 - * `method` — Definiše HTTP metod zahteva (`'GET'`, `'POST'`, itd.).
 - * `headers` — Definiše zaglavlja koja se šalju uz zahtev. Može biti literal objekta ili objekat tipa `Headers`.
 - * `body` — Definiše telo zahteva. Može biti tipa `Blob`, `BufferSource`, `FormData`, `URLSearchParams` ili niska. Napomenimo da metodi `GET` i `HEAD` ne smeju imati telo.
 - * `mode` — Definiše režim zahteva i može biti jedna od vrednosti `'cors'`, `'no-cors'`, `'same-origin'`. Podrazumevano je `'no-cors'`.
 - * `credentials` — Definiše u kojoj situaciji će kredencijali biti poslani i može biti jedna od vrednosti `'omit'`, `'same-origin'` i `'include'`. Podrazumevano je `'same-origin'`.
 - * `cache` — Definiše na koji način se izvršava keširanje zahteva i može biti jedna od vrednosti `'default'`, `'no-store'`, `'reload'`, `'no-cache'`, `'force-cache'` i `'only-if-cached'`. Podrazumevano je `'default'`.
- Povratna vrednost funkcije je:
 - Obećanje koje se ispunjava objektom tipa `Response` ili koje se odbacuje izuzetkom tipa `AbortError` ukoliko se zahtev prekine ili izuzetkom tipa `TypeError` ukoliko, na primer, URI sadrži kredencijale (`http://user:password@example.com`) ili je CORS pogrešno konfigurisan na serveru.

Jedan primer koji ilustruje neke od opisanih opcija dat je u nastavku:

```
1 const myInit = {
2   method: 'POST',
3   body: JSON.stringify({
4     title: 'foo',
5     body: 'bar',
6     userId: 1
7   }),
8   headers: {
9     "Content-type": "application/json; charset=UTF-8"
10  }
11 };
12
13 fetch('https://jsonplaceholder.typicode.com/posts', myInit)
14   .then(response => response.json())
15   .then(json => console.log('Fetch 2:', JSON.stringify(json)));
```

Response

Tip `Response` predstavlja deo Fetch API-ja kojim se reprezentuje HTTP odgovor od servera na zahtev koji je poslat od klijenta. Mada postoji konstruktorska funkcija kojom se mogu kreirati odgovori, značajno je češća situacija da ćemo koristiti objekte odgovora koji se

kreiraju za nas. U nastavku ćemo prikazati korisna svojstva i metode definisane nad ovim tipom, koje možemo koristiti prilikom obrade odgovora od servera:

- Nepromenljiva svojstva nad tipom `Response`:
 - `headers` — Sadrži objekat tipa `Headers` koji predstavlja zaglavlja odgovora od servera
 - `ok` — Bulovala vrednost koja označava da li je statusni kod od servera iz intervala [200,299].
 - `redirected` — Bulovala vrednost koja označava da li je odgovor rezultat preusmeravanja
 - `status` — Statusni kod odgovora
 - `statusText` — Statusna poruka odgovora
 - `url` — Niska koja sadrži URI odgovora (nakon svih eventualnih preusmeravanja)
 - `body` — Čitajući tok koji sadrži telo odgovora od servera. Čitajući tok je implementiran kroz tip `ReadableStream` iz Stream API-ja i još uvek je visoko eksperimentalan, te ga treba koristiti pažljivo u produkcijskom kodu. Umesto njega, preporučuje se korišćenje metoda koji će dohvatiti telo odgovora u odgovarajućem formatu.
- Metodi nad tipom `Response`:
 - `arrayBuffer` — Čita ceo čitajući tok do kraja i vraća obećanje koje se ispunjava objektom tipa `ArrayBuffer`.
 - `blob` — Čita ceo čitajući tok do kraja i vraća obećanje koje se ispunjava objektom tipa `Blob`.
 - `formData` — Čita ceo čitajući tok do kraja i vraća obećanje koje se ispunjava objektom tipa `FormData`.
 - `json` — Čita ceo čitajući tok do kraja kao JSON nisku i vraća obećanje koje se ispunjava objektom koji se parsira iz tela.
 - `text` — Čita ceo čitajući tok do kraja i vraća obećanje koje se ispunjava niskom koja predstavlja tekstualni sadržaj tela.

U prethodnim primerima smo videli dohvaćanje objekata zapisanih u JSON formatu. Na redni primer ilustruje dohvaćanje slike i njeno transformisanje u `Blob` objekat, da bismo to iskoristili kao atribut `src` za HTML element `img` na stranici:

```

1 const image = document.querySelector('.my-image');
2
3 fetch('https://upload.wikimedia.org/wikipedia/commons/thumb/9/99/
  Unofficial_JavaScript_logo_2.svg/1200px-Unofficial_JavaScript_logo_2.svg.
  png')
4   .then(response => response.blob())
5   .then(blob => {
6     const objectURL = URL.createObjectURL(blob);
7     image.src = objectURL;
8     console.log('Fetch 3:', objectURL);
9   });
```

Request

Tip `Request` predstavlja deo Fetch API-ja kojim se reprezentuje HTTP zahtev od klijenta ka serveru. Kreiranje novog zahteva se može izvršiti konstruktorom čiji je potpis dat sa:

```
1 Request Request(input[, init]);
```

Parametri `input` i `init` predstavljaju identične parametre kao i za funkciju `fetch`. U slu-

čaju da se za `init` prosledi objekat tipa `Request`, efektivno će biti napravljena kopija prosleđenog objekta. U nastavku dajemo korisna svojstva i metode definisane nad ovim tipom:

- Nepromenljiva svojstva nad tipom `Request`:
 - Svako od polja koje se postavlja (implicitno ili eksplicitno) kroz parametar `init` ima odgovarajuće istoimeno svojstvo koje dohvata odgovarajuću vrednost.
 - `url` — Niska koja sadrži URI zahteva
- Metodi nad tipom `Request`:
 - Identični metodi kao za `Response`, samo se odnose na telo zahteva.

Primer kreiranja i korišćenja objekta tipa `Request`:

```

1  const image2 = document.querySelector('.my-image2');
2
3  const myInit2 = {
4    method: 'GET',
5    mode: 'cors',
6    cache: 'no-cache'
7  };
8  const myRequest = new Request('https://upload.wikimedia.org/wikipedia/commons/
  thumb/9/99/Unofficial_JavaScript_logo_2.svg/1200px-
  Unofficial_JavaScript_logo_2.svg.png', myInit2);
9
10 fetch(myRequest)
11   .then(response => response.blob())
12   .then(blob => {
13     const objectURL = URL.createObjectURL(blob);
14     image2.src = objectURL;
15     console.log('Fetch 4:', objectURL);
16   });

```

Headers

Interfejs `Headers` predstavlja deo Fetch API-ja koji omogućava izvršavanje raznih akcija nad zaglavlja odgovarajućih HTTP zahteva ili odgovora. Ove akcije podrazumevaju: očitavanje i postavljanje vrednosti zaglavlja, dodavanje novih zaglavlja ili uklanjanje postojećih zaglavlja.

Objekat interfejsa `Headers` sadrži listu zaglavlja, koja je inicijalno prazna, i koja se sastoji od nula ili više parova (*ime, vrednost*). Čitanje svojstava se može vršiti metodom `get`. Dodavanje elemenata u ovoj listi se vrši metodima kao što je `append`. Važno je napomenuti da nazivi zaglavlja nisu osetljivi na velika i mala slova. Sledi primer čitanja vrednosti zaglavlja `content-type` iz HTTP odgovora:

```

1  const myInit = {
2    method: 'GET',
3    mode: 'cors',
4    cache: 'no-cache'
5  };
6  const myRequest = new Request('https://upload.wikimedia.org/wikipedia/commons/
  thumb/9/99/Unofficial_JavaScript_logo_2.svg/1200px-
  Unofficial_JavaScript_logo_2.svg.png', myInit);
7
8  fetch(myRequest)
9    .then(response => {
10      if (response.headers.has('content-type')) {
11        console.log(response.headers.get('content-type'));
12      }

```



```

13     else {
14         console.log('Server nije postavio tip resursa...');
15     }
16 });

```

Iz sigurnosnih razloga, nekim zaglavlja se može upravljati isključivo od strane veb pregledača, odnosno, nije ih moguće menjati programerski. Spisak takvih zaglavlja zahteva se može pronaći na adresi https://developer.mozilla.org/en-US/docs/Glossary/Forbidden_header_name, dok se spisak takvih zaglavlja odgovora može pronaći na adresi https://developer.mozilla.org/en-US/docs/Glossary/Forbidden_response_header_name.

Na neka zaglavlja se može drugačije uticati metodima `set`, `delete` i `append` u zavisnosti od tipa čuvara (engl. *guard*) koji je definisan nad tim zaglavlja. Čuvar može da ima jednu od vrednosti `immutable`, `request`, `request-no-cors`, `response` ili `none`. Više o čuvarima se može pronaći na adresi https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Basic_concepts#Guard.

Objekat klase `Headers` se može očitati iz svojstava `Request.headers` i `Response.headers` odgovarajućih objekata u zavisnosti od toga da li dohvatamo zaglavlja zahteva ili odgovora, redom. Novi objekti se mogu konstruisati korišćenjem konstruktorske funkcije `Headers`.

Objekat koji implementira interfejs `Headers` se može direktno koristiti u *for-of* petlji ili se može iterirati kroz iterator koji se dobija metodom `entries`. Na primer:

```

1  const myInit = {
2      method: 'GET',
3      mode: 'cors',
4      cache: 'no-cache'
5  };
6  const myRequest = new Request('https://upload.wikimedia.org/wikipedia/commons/
   thumb/9/99/Unofficial_JavaScript_logo_2.svg/1200px-
   Unofficial_JavaScript_logo_2.svg.png', myInit);
7
8  fetch(myRequest)
9      .then(response => {
10         for (const header of response.headers) {
11             console.log(header);
12         }
13     });

```

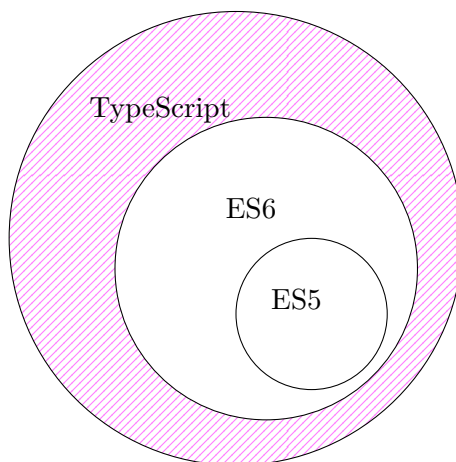
Literatura za ovu oblast

- [AS17] Ved Antani i Stoyan Stefanov. *Object-Oriented JavaScript - Third Edition*. 3rd. Packt Publishing, 2017. ISBN: 178588056X, 9781785880568.
- [doch] MDN web docs. *JavaScript*. URL: <https://developer.mozilla.org/en-US/docs/Web/JavaScript>.
- [GK17] Deepak Grover i Hanu Prateek Kunduru. *ES6 for Humans: The Latest Standard of JavaScript ES2015 and Beyond*. 1st. Berkely, CA, USA: Apress, 2017. ISBN: 1484226224, 9781484226223.
- [Hav18] Marijn Haverbeke. *Eloquent JavaScript: A Modern Introduction to Programming*. 3rd. San Francisco, CA, USA: No Starch Press, 2018. ISBN: 1593275846, 9781593275846.
- [Par15] Daniel Parker. *JavaScript with Promises*. 1st. O'Reilly Media, Inc., 2015. ISBN: 1449373216, 9781449373214.

- [WHA] WHATWG. *XMLHttpRequest Standard*. URL: <https://xhr.spec.whatwg.org>.

3. TypeScript

TypeScript predstavlja jezik koji se intenzivno i rado koristi u programiranju Angular aplikacija. Neko bi se mogao zapitati zbog čega je potrebno učiti ceo novi jezik kada već znamo da radimo u JavaScript jeziku. Odgovor je da, iako bismo mogli programirati Angular aplikacije u čistom JavaScript jeziku, TypeScript nam nudi neke dodatne mogućnosti koje nam olakšavaju razvoj. Još jedan dobar argument za korišćenje TypeScript jezika je njegova veza sa JavaScript jezikom — TypeScript ne predstavlja ceo novi jezik, već ga možemo smatrati *dijalektom* JavaScript jezika u smislu da ga on prirodno proširuje. Već smo pričali o ECMAScript-u i standardima JavaScript jezika. Poznato nam je da ono što se najčešće smatra jezikom JavaScript jeste standard ECMAScript 5. Nova verzija standarda ECMAScript 6 je donela fundamentalna proširenja, a TypeScript nadograđuje ova proširenja. Dijagram odnosa ova tri jezika je dat na slici 3.1.



Slika 3.1: Odnos standarda ECMAScript 5, ECMAScript 6 i dijalekta TypeScript jezika JavaScript.

U ovom poglavlju biće dati opisi samo neophodnih konstrukata jezika TypeScript. Za početak, da bismo mogli da koristimo TypeScript, potrebno je da pokrenemo komandu:

```
npm install -g typescript
```

3.1 Prevođenje TypeScript programa

Pre nego što se upustimo u same elemente jezika TypeScript, pogledajmo na koji način se vrši prevođenje izvornog koda napisanog na jeziku TypeScript.

Kao što znamo, prilikom prevođenja programa napisanih na jezicima kao što su C, C++, Java i slično, od izvornog koda se generiše izvršni kod, koji se sastoji od mašinskih instrukcija, a koji se zatim izvršava od strane operativnog sistema ili neke virtualne mašine. Ovaj proces je poznat pod imenom *kompiliranje* (engl. *compilation*) i izvršava ga program koji se naziva *kompilator* (engl. *compiler*).

Za razliku od ovih jezika, programi napisani u programskom jeziku TypeScript se ne kompiliraju, već se ovaj izvorni kod pretvara u odgovarajući JavaScript kod, koji se potom interpretira od strane nekog JavaScript interpretera (bilo u veb pregledaču, iz komandne linije ili iz nekog drugog okruženja). Ovaj proces je poznat pod nazivom *transpiliranje* (engl. *transpilation*) i izvršava ga program koji se naziva *transpilator* (engl. *transpiler*).

Ukoliko u Node.js okruženju instaliramo paket **typescript** na način opisan u prethodnoj sekciji, na raspolaganju nam je transpilator **tsc**. Pretpostavimo da se neki TypeScript izvorni kod nalazi u datoteci **app.ts**. Transpiliranje ove datoteke se može vršiti pozivanjem naredne komande, čime se kao rezultat generiše datoteka **app.js**:

```
tsc app.ts
```

Nakon što se dobije odgovarajuća JavaScript verzija TypeScript koda, ona se dalje može izvršiti ili u **node** okruženju ili povezivanjem sa odgovarajućom HTML datotekom. Na primer, ako je **app.js** ime transpilirane datoteke, može se izvršiti komanda **node app.js** ili navesti linija `<script type='text/javascript' src='app.js'>` u odgovarajućoj HTML datoteci. Važno je napomenuti da se u drugom slučaju uvek navodi ime datoteke sa ekstenzijom **.js**, a ne originalna **.ts** datoteka. O izvršavanju JavaScript koda je bilo reči u sekciji 2.1.

Ukoliko je ulazna datoteka zavisila od drugih modula napisanih u jeziku TypeScript, i ti moduli će takođe biti prevedeni. Više informacija o podešavanjima transpilatora **tsc** može se dobiti izvršavanjem komande:

```
tsc --help
```

Neka odabrana podešavanja su data postavljanjem narednih zastavica:

- **--module** — Specifikuje koji se šablon uvođenja drugih modula koristi. Neke vrednosti koje je moguće odabrati su: **umd** (Universal Module Definition), **amd** (Asynchronous Module Definition), **commonjs** i **esnext**.
- **--target** — Specifikuje najvišu verziju standarda koja se koristi. Na primer, ukoliko želimo da omogućimo da se ne koriste elementi ECMAScript 6 standarda, moguće je transpilatoru proslediti opciju **--target ES5**, čime se, na primer, neće koristiti deklaracije klasa, **let**, **const** i ostali ECMAScript 6 koncepti. Neke vrednosti koje je moguće odabrati su: **ES3**, **ES5**, **ES6**, **ESNext**.
- **--outFile** — Nadovezuje izlaz iz svih datoteka u jednu datoteku, čiji naziv se specifikuje kao vrednost zastavice.

3.1.1 Upravljanje TypeScript projektima

Naravno, možemo koristiti TypeScript za pisanje celih projekata, koji se zatim prevode u JavaScript i izvršavaju u odgovarajućem okruženju. U tu svrhu, poželjno je inicijalizovati novi projekat i podesiti odgovarajuće opcije za njega. Inicijalizacija se može izvršiti pozivanjem komande:

```
tsc --init
```

čime se kreira datoteka `tsconfig.json` koji sadrži informacije o projektu. Inicijalno, specifikovane su samo opcije za transpilator kroz svojstvo `"compilerOptions"` korenog objekta. Ukoliko želimo da specifikujemo koje se sve datoteke koriste u projektu, to možemo uraditi bilo dodavanjem novog svojstva `"files"`, čija je vrednost niz imena datoteka; na primer:

```
1 "files": [  
2   "app.ts",  
3   "module.ts"  
4 ]
```

bilo korišćenjem svojstava `"include"` and `"exclude"`, kojima se specifikuje koje se sve datoteke uključuju, odnosno, isključuju iz projekta, redom; na primer:

```
1 "include": [  
2   "src/**/*"  
3 ],  
4 "exclude": [  
5   "node_modules",  
6   "**/*.spec.ts"  
7 ]
```

Svojstvo `"files"` koristi apsolutne ili relativne nazive datoteka, dok svojstva `"include"` i `"exclude"` koriste GLOB-šablone za pronalaženje datoteka. GLOB-šabloni mogu da koriste sledeće *višeobeležavajuće* (engl. *wildcard*) karaktere:

- `*` — Obeležava nula ili više karaktera (bez karaktera za razdvajanje direktorijuma)
- `?` — Obeležava tačno jedan karakter (bez karaktera za razdvajanje direktorijuma)
- `*` — Rekurzivno obeležava svaki poddirektorijum

Više o podešavanju projekta kroz datoteku `tsconfig.json` može se pronaći na adresi <https://www.typescriptlang.org/docs/handbook/tsconfig-json.html>.

Ovako postavljen projekat se zatim transpilira jednostavno pozivanjem komande:

```
tsc
```

u korenom direktorijumu projekta koji sadrži datoteku `tsconfig.json`.

Sada pređimo na elemente TypeScript jezika.

3.2 Tipovi

TypeScript uvodi statičku tipiziranost u JavaScript jezik pomoću sistema tipova. To znači da prilikom transpiliranja dolazi do provere tipova zarad provere slučajnog dodeljivanja neispravnih vrednosti. Ukoliko bismo ipak želeli da koristimo dinamičku tipiziranost u nekom delu koda, to je moguće uraditi dodeljivanjem dinamičkog tipa podataka. U nastavku ćemo videti na koje sve načine možemo koristiti sistem tipova jezika TypeScript.

3.2.1 Promenljive i primitivni tipovi

Da bismo definisali promenljivu sa identifikatorom `promenljiva` koja je tipa `tip` i ima vrednost `nekaVrednost`, potrebno je da uradimo:

```
1 let promenljiva: tip = nekaVrednost;
```

Na primer, možemo definisati promenljivu koja sadrži ime na sledeći način:

```
1 let ime: string = 'Stiven';
```

Primitivni tipovi koji su nam na raspolaganju su:

- `string` — sekvenca UTF-16 karaktera
- `boolean` — `true` ili `false`
- `number` — brojevana vrednost u pokretnom zarezu zapisana u 64 bita¹
- `symbol` — jedinstven, nepromenljivi simbol, koji se može koristiti umesto niske kao ime svojstva objekta

Sistem tipova sadrži i nekoliko tipova koji predstavljaju specijalne vrednosti:

- `undefined` — predstavlja vrednost promenljive kojoj još uvek nije dodeljena vrednost
- `null` — može se koristiti za predstavljanje namernog odsustva vrednosti objekta. Na primer, ako bismo imali metod koji pretražuje niz objekata da pronade neki zadati objekat u njemu, metod bi mogao da vrati `null` ukoliko nije uspeo da ga pronade
- `void` — predstavlja slučajeve kada ne postoji vrednost. Na primer, koristi se da prikaže da funkcija ne vraća nikakvu vrednost
- `never` — predstavlja nedostižan deo koda. Na primer, funkcija koja ispaljuje izuzetak ima `never` kao tip povratne vrednosti

Sve što nije primitivan tip u jeziku TypeScript potpada kao potklasa tipa `object`. Većina tipova koja se nalazi u našim programima potpada pod ovu definiciju.

Poslednji tip koji ćemo prikazati jeste tip `any`. On se koristi da predstavi bilo koji tip. Kada koristimo ovaj dinamički tip, ne vrši se provera tipova tokom kompilacije programa. Ovaj tip se takođe koristi u situacijama kada kompilator ne može da automatski zaključi tip, mada se ovo ponašanje može kontrolisati zadavanjem odgovarajuće *zastavice* (engl. *flag*) kompilatoru.

3.2.2 Enumeracije

Moguće je definisati i enumeracije:

```
1 enum TipPrevoza
2 {
3     Automobil,
4     Autobus,
5     Tramvaj,
6     Ostalo
7 }
```

Enumeracije možemo jednostavno koristiti navođenjem naziva i vrednosti enumeracije odvojene tačkom:

```
1 let mojTip = TipPrevoza.Automobil;
```

¹Ne postoji specijalan tip za predstavljanje celih brojeva ili drugih specijalnih varijanti numeričkih vrednosti jer ne bi bilo praktično izvršavati statičku analizu da bi se osiguralo da sve moguće vrednosti bivaju tačne.

Moguće je dohvatiti naziv vrednosti enumeracije iz tipa tretiranjem enumeracije kao niz:

```
1 let imeMogTipa = TipPrevoza[mojTip];    // 'Automobil'
```

3.2.3 Unije

Unije se definišu navođenjem više tipova odvojenih operatorom `|`:

```
1 let union: boolean | number;
2 // OK: number
3 union = 5;
4 // OK: boolean
5 union = true;
6 // Error: Type "string" is not assignable to type 'number | boolean'
7 union = 'string';
```

Moguće je uvesti i alias za uniju (kao i za bilo koji drugi tip), što je preporučljivo da bi se izbegavalo dupliranje unija:

```
1 type StringOrError = string | Error;
2
3 let sorExample: StringOrError = 'greska';
```

3.2.4 Presek tipova

Ako želimo da napravimo tip koji ima elemente više drugih tipova, možemo koristiti operator preseka tipova (`&`). Na primer, naredna dva interfejsa² `Skier` i `Shooter` imaju po jedan metod:

```
1 interface Skier {
2     slide(): void;
3 }
4
5 interface Shooter {
6     shoot(): void;
7 }
```

Možemo uvesti novi tip `Biathlete` koji će imati oba metoda `slide` i `shoot` korišćenjem operatora preseka na sledeći način:

```
1 type Biathlete = Skier & Shooter;
```

3.2.5 Nizovi

Nizovi u TypeScript-u imaju precizan tip za svoj sadržaj. Da bismo specifikovali tip niza, dovoljno je dodati uglaste zagrade nakon naziva tipa, na primer:

```
1 let nizNiski: string[];
```

ili je moguće koristiti varijantu `Array<T>`, gde je `T` neki tip:

```
1 let nizNiski: Array<string>;
```

Ovakva specifikacija važi za sve tipove, bez obzira na to da li su oni primitivni ili korisnički. Prilikom dodavanja elementa u niz, dolazi do provere saglasnosti njegovog tipa sa tipom niza. Ovo je takođe korisno jer svi savremeni alati za razvoj mogu da iskoriste ovu informaciju za *dopunjavanje koda* (engl. *code autocompletion*) prilikom pristupa elementima niza, zato što im je poznat njihov tip.

²O interfejsima će biti više reči u sekciji 3.3.4.

3.2.6 Funkcije

Uvođenjem sistema tipova i funkcijama je moguće pridružiti specifikaciju tipova za parametre, kao i za povratnu vrednost:

```
1 function saberi(a: number, b: number): string
2 {
3     return `Zbir brojeva ${a} i ${b} je ${a + b}`;
4 }
```

Kao što vidimo, za svaki argument se zasebno navodi tip nakon identifikatora argumenta. Dodatno, nakon svih argumenata, a pre tela funkcije, navodi se povratni tip funkcije. Ukoliko ne navedemo povratni tip funkcije, TypeScript će pokušati sam da ustanovi tip, što je posebno korisno u slučaju da imamo putanje u telu koje vraćaju različite tipove (na primer, `number` i `string`), čime će TypeScript transpilator ustanoviti uniju (na primer, `number | string`).

Već smo videli da je u JavaScript jeziku moguće pozivati funkcije sa manjim ili većim brojem argumenata, u odnosu na broj parametara koji je specifikovan pri definiciji funkcije. S obzirom da se u jeziku TypeScript argumenti temeljno proveravaju, potrebno je specifikovati ukoliko je neki parametar opcioni pri deklaraciji funkcije, što se vrši navođenjem karaktera `?` nakon identifikatora parametra, a ispred deklaracije tipa, na primer:

```
1 function getAverage(a: number, b: number, c?: number): string {
2     let total = a;
3     let count = 1;
4
5     total += b;
6     count++;
7
8     if (typeof c !== 'undefined')
9     {
10         total += c;
11         count++;
12     }
13
14     const average = total / count;
15     return 'The average is ' + average;
16 }
17
18 // 'The average is 5'
19 const result = getAverage(4, 6);
```

Kao što vidimo, prilikom korišćenja opcionih parametara, moramo proveravati da li je vrednost inicijalizovana, što se može izvršiti operatorom `typeof`, kao u prethodnom primeru.

Podrazumevani parametri se navode kao i u jeziku JavaScript. Naredni primer ilustruje korišćenje podrazumevanih parametara:

```
1 function concatenate(items: string[],
2     separator = ',',
3     beginAt = 0,
4     endAt = items.length)
5 {
6     let result = '';
7
8     for (let i = beginAt; i < endAt; i++)
9     {
```



```
10     result += items[i];
11     if (i < (endAt - 1))
12     {
13         result += separator;
14     }
15 }
16
17 return result;
18 }
19
20 const items = ['A', 'B', 'C'];
21
22 const result = concatenate(items); // 'A,B,C'
23 const partialResult = concatenate(items, '-', 1); // 'B-C'
```

Jedna razlika jeste da prilikom korišćenja podrazumevanih parametara u TypeScript jeziku, generisani JavaScript kod sadrži proveru tipa podrazumevanih parametara, slično kao što smo mi sami pisali proveru u slučaju opcionih parametara. Ovo znači da je provera pomerena izvan tela funkcije, što ga čini kraćim i čitljivijim. Zbog pomeranje provere, moguće je za podrazumevanu vrednost uzeti i neku dinamičku vrednost (kao što je to slučaj sa parametrom `endAt` u prethodnom primeru), za razliku od mnoštva drugih programskih jezika koji omogućavaju korišćenje isključivo konstantnih vrednosti za podrazumevane parametre.

3.3 Klase

U ovoj sekciji ćemo predstaviti samo nove stvari koje TypeScript uvodi u odnosu na JavaScript.

3.3.1 Konstruktori

Sve klase u TypeScript-u imaju konstruktor, ili eksplicitni, ukoliko ga mi navedemo, ili implicitni, koji će biti dodat od strane transpilatora u slučaju da mi ne navedemo eksplicitni. Za klasu koja ne nasleđuje neku drugu klasu, implicitni konstruktor nema parametre i inicijalizuje svojstva klase. U suprotnom, implicitni konstruktor će ispoštovati potpis konstruktora natklase, i proslediće parametre konstruktoru potklase pre nego što inicijalizuje svoja svojstva.

Pogledajmo naredni primer:

```
1 class Song {
2     private artist: string;
3     private title: string;
4
5     constructor(artist: string, title: string) {
6         this.artist = artist;
7         this.title = title;
8     }
9
10    play() {
11        console.log('Playing ' + this.title + ' by ' + this.artist);
12    }
13 }
14
15 const song = new Song('Lady Gaga', 'Poker Face');
16 song.play();
```

U ovom primeru smo u klasi `Song` eksplicitno specifikovali svojstva `artist` i `title`, te je bilo neophodno da ih u konstruktoru inicijalizujemo na vrednosti prosledene kao argumente konstruktora. U metodi `play` vidimo da se ove vrednosti koriste pomoću ključne reči `this`. Ipak, postoji bolji način da se ova inicijalizacija izvede, što je demonstrirano u narednom primeru:

```
1 class Song {
2     constructor(private artist: string, private title: string) {
3     }
4
5     play() {
6         console.log('Playing ' + this.title + ' by ' + this.artist);
7     }
8 }
9
10 const song = new Song('Lady Gaga', 'Poker Face');
11 song.play();
```

Ukoliko parametru konstruktora dodelimo *modifikator pristupa* (engl. *access modifier*), kao što je `private`, taj parametar će za nas biti automatski preslikan u odgovarajuće istoimeno svojstvo klase i biće mu dodeljena odgovarajuća vrednost, što ubrzava razvoj klase i smanjuje broj linija koda. Takođe, primetićemo da se njihova upotreba nije uopšte promenila u metodi `play`, što je svakako korisno.

3.3.2 Modifikatori pristupa

U prethodnom primeru smo pomenuli modifikatore pristupa. U pitanju su ključne reči kojima se specifikuje vidljivost svojstava klase. U TypeScript-u postoje naredna tri modifikatora pristupa, poredana po specifikaciji vidljivosti od najmanje do najveće:

- `private` — Vršiti restrikciju vidljivosti svojstava na nivou klase u kojoj su deklarirani. Pokušaj pristupa privatnih svojstava rezultuje greškom.
- `protected` — Vršiti restrikciju vidljivosti svojstava na nivou klase i svih njenih potklasa. Pokušaj pristupa iz bilo kojih drugih izvora rezultuje greškom.
- `public` — Ne vrši nikakvu restrikciju vidljivosti svojstava. Ovaj modifikator pristupa se koristi kao podrazumevani, ukoliko se pri deklaraciji ne navede eksplicitno nijedan modifikator. Stoga njega nije potrebno navoditi eksplicitno, osim u slučaju da želimo da mapiramo parametre konstruktora u svojstva klase.

Naravno, modifikatori pristupa nakon transpilacije bivaju uklonjeni, te se ne vrši nikakva provera pristupa u fazi interpretacije, već samo u fazi prevođenja.

3.3.3 Svojstva i metodi

Svojstva TypeScript klase se tipično deklariraju ispred konstruktora. Deklaracija svojstva se sastoji od tri elementa: (1) opcionog modifikatora pristupa, (2) identifikatora i (3) anotacije tipa. Na primer: `public name: string;`. Svojstva se mogu i definisati nekom vrednošću, na primer: `public name: string = 'Jane';`. Ukoliko se definiše vrednost, onda tip nije potrebno navesti jer će biti zaključen iz konteksta, na primer: `public name = 'Jane';`. Prilikom transpilacije programa, inicijalizatori svojstava se premeštaju na početak konstruktora, tako da će biti izvršeni pre bilo kakvog drugog koda u konstruktoru. Svojstvima se može pristupiti korišćenjem ključne reči `this`. Ako je svojstvo deklarirano sa javnim modifikatorom pristupa, onda mu se može pristupiti preko identifikatora instance klase. Ukoliko se ne navede modifikator pristupa, podrazumevano se koristi modifikator `public`.

Moguće je dodavanje statičkih svojstava klase, tj. svojstava koja su dostupna svim instancama klase. Način deklarisanja statičkih svojstava je isti kao onaj za deklarisanje nestatičkih svojstava, sa razlikom da se dodaje ključna reč `static` između modifikatora pristupa (ako je naveden) i identifikatora svojstava. Vrednosti statičkih svojstava se očitavaju ili postavljaju kroz naziv klase.

Moguće je definisati, bilo nestatička ili statička, svojstva da budu samo za čitanje kako bi se sprečilo njihovo menjanje. Ovo se omogućava navođenjem ključne reči `readonly` ispred identifikatora svojstva, a iza ključne reči `static` (za statička svojstva).

Naredni primer ilustruje definicije privatnog nestatičkog svojstva `songs` i javnog statičkog svojstva `maxSongCount`. Dodatno, statičko svojstvo `maxSongCount` je postavljeno za `readonly`, odnosno, nije ga moguće menjati, već koristiti isključivo za čitanje.

```
1 class Song {
2     constructor(private artist: string, private title: string) {
3     }
4
5     play() {
6         console.log('Playing ' + this.title + ' by ' + this.artist);
7     }
8 }
9
10 class Playlist {
11     private songs: Song[] = [];
12     public static readonly maxSongCount = 30;
13
14     constructor(public name: string) {
15     }
16
17     addSong(song: Song) {
18         if (this.songs.length >= Playlist.maxSongCount) {
19             throw new Error('Playlist is full');
20         }
21         this.songs.push(song);
22     }
23 }
24
25 // Creating a new instance
26 const playlist = new Playlist('My Playlist');
27
28 // Accessing a public instance property
29 const name = playlist.name;
30
31 // Calling a public instance method
32 playlist.addSong(new Song('Kylie Minogue', 'New York City'));
33
34 // Accessing a public static property
35 const maxSongs = Playlist.maxSongCount;
36
37 // Error: Cannot assign to a readonly property
38 Playlist.maxSongCount = 20;
```

U prethodnom primeru možemo videti i primer definicije metoda `addSong`. Metodi su svojstva klasa čije su vrednosti funkcije. Oni se definišu poput klasičnih funkcija, o čemu smo diskutovali ranije, samo bez navođenja ključne reči `function`. Slično kao i za funkcije, ukoliko se ne navede tip povratne vrednosti, za tip se smatra `void`, odn. u pitanju je procedura. Metode je, poput svojstava, moguće definisati kao statičke ili nestatičke, kao

i dodeliti im modifikatore pristupa.

TypeScript podržava i upotrebu *očitavača* (engl. *getter*) i *postavljača* (engl. *setter*) korišćenjem ključnih reči `get` i `set`, redom. Ovakva svojstva u definiciji klase predstavljaju metode kojima se vrednosti mogu očitavati, odnosno, postavljati, ali se u kodu koriste kao svojstva. Naredni primer ilustruje ova svojstva:

```

1  class Song {
2      constructor(public artist: string, public title: string) {
3      }
4
5      play() {
6          console.log('Playing ' + this.title + ' by ' + this.artist);
7      }
8  }
9
10 class Playlist {
11     private songs_: Song[] = [];
12     public static readonly maxSongCount = 30;
13
14     constructor(public name: string) {
15     }
16
17     addSong(song: Song) {
18         if (this.songs_.length >= Playlist.maxSongCount) {
19             throw new Error('Playlist is full');
20         }
21         this.songs_.push(song);
22     }
23
24     // Primer očitavača – služi samo za očitavanje privatnog svojstva songs_
25     get songs(): Song[] {
26         return this.songs_;
27     }
28
29     // Primer očitavača – na osnovu svojstva songs_ kreiramo niz imena
30     get artistNames(): String[] {
31         // Ovde koristimo "this.songs_" očitavač, umesto da pristupamo direktno
32         // "this.songs_".
33         // Iako se očitavači definišu kao metodi u klasi,
34         // u kodu se koriste kao "klasična" svojstva.
35         return this.songs.map(song => song.artist);
36     }
37
38     // Primer postavljača – popunjavamo niz pesama na osnovu drugog Playlist
39     // objekta
40     set allSongs(p: Playlist) {
41         this.songs_ = [];
42         for (const song of p.songs) {
43             this.addSong(song);
44         }
45     }
46
47     const pop = new Playlist('Best of Pop 2019');
48     pop.addSong(new Song('Kylie Minogue', 'New York City'));
49     pop.addSong(new Song('Katy Perry', 'Harleys in Hawaii'));
50     pop.addSong(new Song('Adam Lambert', 'Superpower'));
51     pop.addSong(new Song('Carly Rae Jepsen', 'Party for One'));
52
53     const pop2 = new Playlist('Great pop songs of 2019');

```

```

53 pop2.addSong(new Song('Camila Camello', 'Shameless'));
54 pop2.addSong(new Song('Ava Max', 'Torn'));
55
56 // Korišćenje očitavača:
57 console.log(pop2.songs.length); // 2
58
59 // Korišćenje postavljača
60 pop2.allSongs = pop;
61
62 console.log(pop2.songs.length); // 4

```

Napomenimo i to da ukoliko očitavač i postavljač imaju isto ime, onda se povratna vrednost očitavača i argument postavljača moraju poklapati, inače dolazi do greške. Takođe, očitavači i postavljači su dostupni samo ako se transpilira za verziju ECMAScript 5 ili veću.

3.3.4 Nasleđivanje i interfejsi

Postoje tri tipa nasleđivanja u jeziku TypeScript:

1. *Nasleđivanje klase*. Klasa može da nasleđuje drugu klasu, čime se ostvaruje da instance potklase dobijaju svojstva i metode iz natklase. Ovaj vid nasleđivanja opisuje relaciju *jeste* (engl. *is-a*).
2. *Implementiranje interfejsa*. Klasa može da implementira *interfejs* (engl. *interface*) kojim se zadaju ograničenja koja potklasa mora da ispunjava. Ovaj vid nasleđivanja opisuje relaciju *ispunjava* (engl. *has-a*).
3. *Kombinovano nasleđivanje*. Predstavlja kombinaciju prethodna dva tipa.

Nasleđivanje klase

Klasa nasleđuje drugu klasu tako što se navede ključna reč `extends` nakon naziva klase koja nasleđuje, a iza nje se navodi naziv klase iz koje se nasleđuje. Klauza `extends` postavlja novu klasu za *izvedenu klasu* ili *potklasu*, dok se klasa iz koje se nasleđuje naziva *bazna klasa* ili *natklasa*. Instance potklase dobijaju sve metode iz natklase.

```

1 class Song {
2     constructor(private artist: string, private title: string) {
3     }
4
5     play(): void {
6         console.log('Playing ' + this.title + ' by ' + this.artist);
7     }
8 }
9
10 class Playlist {
11     constructor(public songs: Song[]) {
12     }
13
14     protected play() {
15         if (this.songs.length === 0) {
16             console.log('The song queue is empty');
17             return;
18         }
19         const song = this.songs.pop();
20         song.play();
21     }
22 }
23
24 class PopPlaylist extends Playlist {

```

```

25     constructor(songs: Song[]) {
26         super(songs);
27     }
28
29     playNextSong() {
30         console.log('Playing the next song in the pop playlist...');
31         super.play();
32     }
33 }
34
35 const songs: Song[] = [
36     new Song('Kylie Minogue', 'New York City'),
37     new Song('Katy Perry', 'Harleys in Hawaii'),
38     new Song('Adam Lambert', 'Superpower'),
39     new Song('Carly Rae Jepsen', 'Party for One')
40 ];
41 const pop = new PopPlaylist(songs);
42 pop.playNextSong();
43 pop.playNextSong();
44 pop.playNextSong();
45 pop.playNextSong();
46 pop.playNextSong();

```

Konstruktor potklase `PopPlaylist` iz prethodnog primera može da se izostavi, zato što TypeScript generiše *podrazumevani konstruktor* koji ima istu definiciju. Naravno, da smo imali neki dodatni kod koji se izvršava u konstruktoru, onda bismo morali da ga eksplicitno definišemo. Na primer, ako konstruktor potklase zahteva neke dodatne argumente, onda poziv konstruktora iz natklase mora biti prva naredba u telu konstruktora potklase. Dodatno, TypeScript zabranjuje da se navede restriktivniji modifikator za parametar potklase u odnosu na onaj koji je naveden u natklasi.

```

1  class Song {
2      constructor(protected artist: string, protected title: string) {
3      }
4  }
5
6  class SongWithFeatures extends Song {
7      private hasFeaturedArtists: boolean;
8
9      // Konstruktor potklase se razlikuje od konstruktora natklase,
10     // pa ga je potrebno eksplicitno napisati
11     constructor(artist: string, title: string, private featuredArtists: string
12         []) {
13         super(artist, title);
14
15         this.hasFeaturedArtists = this.featuredArtists.length > 0;
16     }
17
18     play(): void {
19         let songString = 'Playing ' + this.title;
20         if (this.hasFeaturedArtists) {
21             songString += ' (feat. ';
22         }
23         for (let i = 0; i < this.featuredArtists.length; ++i) {
24             const feat = this.featuredArtists[i];
25             songString += feat + (i === this.featuredArtists.length-1 ? '' : ',
26                 ');
27         }
28         if (this.hasFeaturedArtists) {
29             songString += ')';
30         }
31     }
32 }

```

```

28     }
29     songString += ' by ' + this.artist;
30     console.log(songString);
31 }
32 }
33
34 const song = new SongWithFeatures('Rita Ora', 'Girls', ['Cardi B', 'Bebe Rexha'
35   , 'Charli XCX']);
36 song.play();

```

Postoje određena pravila koja se moraju ispoštovati za nasleđivanje klasa:

- Klasa može da nasleđuje najviše jednu natkласu.
- Klasa ne može da nasleđuje sebe, bilo direktno ili kroz lanac nasleđivanja.

Interfejsi

Interfejsi se deklariraju ključnom reči `interface` i sadrže niz anotacija kojima se opisuju svojstva i funkcije klase koje opisuju. Svojstva i metodi se deklariraju korišćenjem sada već poznate sintakse. U interfejsima možemo specificovati i konstruktore, koji se deklariraju ključnom reči `new`, a čiji je povratni tip, očigledno, naziv interfejsa. Naredni primer ilustruje neke interfejse:

```

1 interface Point {
2     // Properties
3     x: number;
4     y: number;
5 }
6
7 interface Passenger {
8     // Properties
9     name: string;
10 }
11
12 interface Vehicle {
13     // Constructor
14     new(): Vehicle;
15
16     // Properties
17     currentLocation: Point;
18
19     // Methods
20     travelTo(point: Point): void;
21     addPassenger(passenger: Passenger): void;
22     removePassenger(passenger: Passenger): void;
23 }

```

Klasa implementira interfejs navođenjem ključne reči `implements` nakon svog naziva, a koju prati naziv interfejsa koji se implementira. U suštini, kada se interfejs implementira, deklaracija pomoću ključne reči `implements` je u potpunosti opcionalna jer nas ništa ne sprečava da definišemo metode koje interfejs nalaže bez da eksplicitno označimo da se odgovarajući interfejs implementira. Međutim, ukoliko navedemo da klasa implementira interfejs navođenjem ključne reči `implements`, prilikom transpilacije biće provereno da li ta klasa ispunjava uslove koje definicija interfejsa nalaže. Ovo može biti korisno, na primer, radi provere da li su svi željeni metodi implementirani.

Naredni primer ilustruje klasu `Song` koja implementira interfejs `Audio`. Kako taj interfejs postavlja ograničenje da svaka klasa koja ga implementira mora da implementira metod `play` (sa datom deklaracijom), to će transpiler proveriti da li klasa `Song` implementira taj

metod. Ukoliko to nije slučaj, biće prijavljena greška. Naravno, klasa koja implementira interfejs može imati i druga svojstva i metode osim onih koji su navedeni u interfejsu. Tako, na primer, klasa `Song` ima statički metod `Comparer` koji definiše leksikografski poredak dve instance klase `Song`.

```

1 interface Audio {
2     play(): any;
3 }
4
5 class Song implements Audio {
6     constructor(private artist: string, private title: string) {
7     }
8
9     play(): void {
10         console.log('Playing ' + this.title + ' by ' + this.artist);
11     }
12
13     static Comparer(a: Song, b: Song) {
14         if (a.title === b.title) {
15             return 0;
16         }
17         return a.title > b.title ? 1 : -1;
18     }
19 }

```

Klasa može da implementira više interfejsa, pri čemu se interfejsi navode nakon ključne reči `implements`, razdvojeni karakterom zapete. Na primer:

```

1 interface HearingSense {
2     listen(): void;
3 }
4
5 interface SeeingSense {
6     look(): void;
7 }
8
9 class Human implements HearingSense, SeeingSense {
10     constructor(private name: string) {
11     }
12
13     listen() {
14         console.log('A human called ' + this.name + ' can hear');
15     }
16
17     look() {
18         console.log('A human called ' + this.name + ' can see');
19     }
20 }
21
22 const ana = new Human('Ana');
23 ana.listen();
24 ana.look();

```

Metod klase koji implementira ograničenje postavljeno u interfejsu može imati manje parametara nego što odgovarajuće ograničenje iz interfejsa ima. Ovim se postiže da klasa ignoriše parametre koji joj nisu neophodni da bi se metod izvršio. Na primer:

```

1 interface SabiračBrojeva {
2     saberi(a: number, b: number, c: number, d: number, e: number): number;
3 }

```



```
4
5 class SabiračTriBroja implements SabiračBrojeva {
6     saberi(a: number, b: number, c: number): number {
7         return a + b + c;
8     }
9 }
10
11 const sabirač = new SabiračTriBroja();
12 console.log(sabirač.saberi(0, 1, 2)); // 3
```

O korišćenju interfejsa

Interfejsi se u TypeScript-u mogu koristiti u različite svrhe. Naravno, moguće je koristiti interfejs kao apstraktan tip, koji može biti implementiran u konkretnim klasama, ali oni se mogu koristiti za definisanje strukture u programima.

Interfejsi su korisni i za deklarisanje specifikacije koje neke vrednosti moraju da ispunjavaju. Ukoliko koristimo primitivne vrednosti, ovakva specifikacija nam nije neophodna, na primer:

```
1 function typeGuardExample(stringNumber: string | number) {
2     // Bilo da se prosledi broj ili niska,
3     // tacno jedna od naredne dve naredbe dodele ce izbaciti narednu gresku:
4     // Error: Property does not exist
5     // const a = stringNumber.length;
6     // const b = stringNumber.toFixed();
7
8     // Type guard
9     if (typeof stringNumber === 'string') {
10         // OK
11         return stringNumber.length;
12     }
13     else {
14         // OK
15         return stringNumber.toFixed();
16     }
17 }
```

Ipak, u nekim kompleksnijim situacijama, ovo nam neće biti dovoljno. Na primer, ukoliko je potrebno da neki objekti imaju odgovarajuće metode, možemo definisati interfejs koji postavlja ova ograničenja. Međutim, operator `typeof` primenjen na objekat bilo koje klase izračunava nisku `'object'`. Umesto toga, možemo definisati funkciju koja će ispitivati zadovoljivost ograničenja, korišćenjem specijalne klauze `is` u anotaciji povratne vrednosti funkcije, kao u narednom primeru:

```
1 interface InclineControllable {
2     increaseIncline(): void;
3     decreaseIncline(): void;
4 }
5
6 interface SpeedControllable {
7     increaseSpeed(): void;
8     decreaseSpeed(): void;
9     stop(): void;
10 }
11
12 function isSpeedControllable(moveable: SpeedControllable | any): moveable is
    SpeedControllable {
13     if (moveable.increaseSpeed && moveable.decreaseSpeed && moveable.stop) {
14         return true;
15     }
16 }
```

```

15     }
16     return false;
17 }

```

Sada se ova funkcija može iskoristiti isto kao u slučaju korišćenja operatora `typeof`:

```

1 function increaseSpeedIfPossible(moveable: SpeedControllable |
  InclineControllable) {
2     if (isSpeedControllable(moveable)) {
3         // OK
4         moveable.increaseSpeed();
5     }
6 }

```

Primer koda koji testira opisanu funkcionalnost:

```

1 class Treadmill implements InclineControllable {
2     increaseIncline(): void {
3         console.log('Treadmill incline increased');
4     }
5
6     decreaseIncline(): void {
7         console.log('Treadmill incline decreased');
8     }
9 }
10
11 class Car implements SpeedControllable {
12     increaseSpeed(): void {
13         console.log('Car speed increased');
14     }
15
16     decreaseSpeed(): void {
17         console.log('Car speed decreased');
18     }
19
20     stop(): void {
21         console.log('Car stopped');
22     }
23 }
24
25 const c = new Car();
26 increaseSpeedIfPossible(c); // Will call increaseSpeed: 'Car speed increased'
27
28 const t = new Treadmill();
29 increaseSpeedIfPossible(t); // Will not call increaseSpeed AND will not produce
    error

```

Za kraj, napomenimo da se interfejsi ne prevode ni u jedan JavaScript kod. Oni se koriste u fazi dizajniranja projekta, radi omogućavanja dopunjavanja koda i u fazi prevođenja radi provere tipova.

3.3.5 Apstraktne klase

Apstraktna klasa (engl. *abstract class*) predstavlja klasu koja se može koristiti kao bazna klasa, ali se ne može direktno instancirati. Apstraktne klase mogu sadržati implementirane metode, kao i *apstraktne metode*, odnosno, metodi koji nemaju definiciju. Apstraktni metodi se moraju implementirati u nekoj izvedenoj klasi. Apstraktne klase se kreiraju navođenjem ključne reči **abstract** ispred ključne reči **class**; apstraktni metodi se deklariraju navođenjem ključne reči **abstract** ispred identifikatora metoda.

Naredni primer sadrži apstraktnu klasu `Logger` koja služi za logovanje poruka i koja sadrži dva metoda: (1) apstraktni metod `notify` i (2) implementirani zaštićeni metod `getMessage`. Svaka potklasa koja nasleđuje klasu `Logger` mora da implementira metod `notify`, ali sve one mogu da dele implementaciju metoda `getMessage` iz bazne klase.

```

1 // Abstract class
2 abstract class Logger {
3     abstract notify(message: string): void;
4
5     protected getMessage(message: string): string {
6         return `Information: ${new Date().toUTCString()} ${message}`;
7     }
8 }
9
10 class ConsoleLogger extends Logger {
11     notify(message) {
12         console.log(this.getMessage(message));
13     }
14 }
15
16 class DetailedConsoleLogger extends Logger {
17     notify(message) {
18         console.log(`[LOG][Console][Production] ${this.getMessage(message)}`);
19     }
20 }
21
22 // Error. Cannot create an instance of an abstract class
23 // const logger = new Logger();
24
25 // Create an instance of a sub-class
26 const logger1 = new ConsoleLogger();
27 logger1.notify('Hello World');
28
29 const logger2 = new DetailedConsoleLogger();
30 logger2.notify('Hello World');
```

Apstraktne klase su slične interfejsima u smislu da postavljaju ograničenje implementacije koje mora biti ispunjeno. Za razliku od interfejsa, apstraktne klase mogu takođe sadržati implementacije nekih metoda i mogu da specifikuju modifikatore pristupa uz metode.

3.4 Polimorfizam

Za neki izraz ili vrednost u programu kažemo da su *polimorfni* ako mogu da imaju za domen više različitih tipova. U skladu sa time, za neki *programski kod* kažemo da je *polimorfan* ako sadrži polimorfne vrednosti ili izraze.

U jeziku TypeScript, kod koji ispoljava polimorfno ponašanje se može ostvariti na dva načina, odnosno, korišćenjem naredna dva koncepta:

1. Hijerarhijski polimorfizam.
2. Parametarski polimorfizam.

3.4.1 Hijerarhijski polimorfizam

Hijerarhijski polimorfizam predstavlja tehniku u kojoj se moguće *prevazići* (engl. *override*) svojstva i metode tako što se u izvedenoj klasi navede svojstvo/metod sa istim imenom i tipom. U narednom primeru, klasa `RepeatingPlaylist` nasleđuje klasu `Playlist` i koristi

svojstvo `songs` koje dolazi iz bazne klase navođenjem `this.songs`, ali, u isto vreme, prevazilazi metod `play` definisanjem specijalizovane implementacija koja ciklično pušta narednu pesmu u nizu.

```

1  class Song {
2      constructor(public artist: string, public title: string) {
3      }
4
5      play() {
6          console.log('Playing ' + this.title + ' by ' + this.artist);
7      }
8  }
9
10 class Playlist {
11     constructor(public songs: Song[]) {
12     }
13
14     play() {
15         if (this.songs.length === 0) {
16             console.log('The song queue is empty');
17             return;
18         }
19         const song = this.songs.pop();
20         song.play();
21     }
22 }
23
24 class RepeatingPlaylist extends Playlist {
25     private songIndex = 0;
26
27     constructor(songs: Song[]) {
28         super(songs);
29     }
30
31     play() {
32         if (this.songIndex === 0) {
33             console.log('-----');
34         }
35
36         this.songs[this.songIndex].play();
37         this.songIndex++;
38         if (this.songIndex >= this.songs.length) {
39             this.songIndex = 0;
40         }
41     }
42 }
43
44 const pop = new RepeatingPlaylist([
45     new Song('Kylie Minogue', 'New York City'),
46     new Song('Katy Perry', 'Harleys in Hawaii'),
47     new Song('Adam Lambert', 'Superpower'),
48     new Song('Carly Rae Jepsen', 'Party for One')
49 ]);
50
51 pop.play();
52 pop.play();
53 pop.play();
54 pop.play();
55
56 // Za razliku od obicne Playlist-e,
57 // naredni poziv nece ispisati da je red pesama ispraznjen,

```

```
58 // vec ce prikazati prvu pesmu iz reda.  
59 pop.play();
```

Takođe, u jeziku TypeScript moguće je čuvati referencu na objekat potklase kao referencu na objekat natklase. Ovo proizilazi iz činjenice da objekat potklase *jest*e objekat natklase. Obrnuto ne važi. U narednom primeru, niz `zoo` deklarisan je kao niz referenci objekata `Animal`, a on sadrži reference objekata `Cat`, `Dog` i `Animal`. Vidimo da, prilikom poziva metoda `say`, jezik TypeScript će odabrati ispravno prevazilaženje u zavisnosti od toga na koji tačno tip referiše tekuća referenca iz niza, čime se osigurava ispravno polimorfno ponašanje.

```
1 class Animal {  
2     say(): void {  
3         console.log('I can only speak abstractly');  
4     }  
5 }  
6  
7 class Cat extends Animal {  
8     say(): void {  
9         console.log('Meow!');  
10    }  
11 }  
12  
13 class Dog extends Animal {  
14     say(): void {  
15         console.log('Woof!');  
16     }  
17 }  
18  
19 const zoo: Animal[] = [];  
20 zoo.push(new Cat());  
21 zoo.push(new Dog());  
22 zoo.push(new Dog());  
23 zoo.push(new Cat());  
24 zoo.push(new Animal());  
25  
26 for (const animal of zoo) {  
27     animal.say();  
28 }
```

Dodatno, moguće je eksplicitno proveriti tip klase pomoću operatora `instanceof` i izvršiti *eksplicitno kastovanje* (engl. *type assertion*) u odgovarajući tip potklase operatorom *kastovanja*. U slučaju da se kastuje u tip koji nije odgovarajući, neće biti prijavljena greška, sve dok se ne pokuša sa pristupanjem svojstava ili metoda koji ne postoje, kao što naredni primer ilustruje:

```
1 abstract class Animal {  
2     abstract say(): void;  
3 }  
4  
5 class Dog extends Animal {  
6     say(): void {  
7         console.log('Woof!');  
8     }  
9  
10    chewOn(something: string): void {  
11        console.log('I like to chew on ' + something);  
12    }  
13 }
```

```

14
15 class Cat extends Animal {
16     say(): void {
17         console.log('Meow!');
18     }
19
20     purr(): void {
21         console.log('I like to purr');
22     }
23 }
24
25 const dog: Animal = new Dog();
26
27 // Error: Property 'chewOn' does not exist on type 'Animal'
28 // dog.chewOn('a tasty bone');
29
30 if (dog instanceof Dog) {
31     (<Dog>dog).chewOn('a tasty bone'); // OK
32 }
33
34 const cat: Cat = <Cat>dog;
35 try {
36     cat.purr(); // TypeError
37 } catch (err) {
38     console.log('Dogs can\'t purr...');
39 }

```

3.4.2 Parametarski polimorfizam

Parametarski polimorfizam dozvoljava način programiranja u kojem je moguće odložiti navođenje konkretnih tipova podataka za kasnije. Umesto navođenja konkretnih tipova podataka, koriste se *šablonski parametri* koji će u nekom trenutku biti zamenjeni konkretnim tipovima. Parametarski polimorfizam može biti koristan da bi se izbeglo kodiranje identičnih algoritama za različite tipove podataka. U jeziku TypeScript, parametarski polimorfizam se ostvaruje kodiranjem šablonskih funkcija, šablonskih klasa i šablonskih interfejsa.

Šablonske funkcije

Da bismo neku funkciju učinili šablonskom, potrebno je da nakon njenog identifikatora navedemo zagrade manje-veće (< i >) između kojih se navodi jedan šablonski parametar ili više njih razdvojenih zaptetama. Šablonski parametri koji su deklarirani na ovaj način se mogu koristiti za anotiranje tipa parametara funkcije, povratne vrednosti ili kao tip u telu funkcije.

Prilikom poziva šablonske funkcije, moguće je eksplicitno specificovati konkretan tip koji će zameniti šablonski parametar navođenjem zagrada veće-manje nakon identifikatora funkcije između kojih se navodi konkretan tip podataka. Ako se tipovi šablonskih parametara mogu zaključiti automatski (na primer, na osnovu tipova argumenata koji se prosleđuju funkciji), onda se ne mora eksplicitno specificovati konkretan tip.

```

1 function reverse<T>(list: T[]): T[] {
2     const reversedList: T[] = [];
3     for (let i = (list.length - 1); i >= 0; i--) {
4         reversedList.push(list[i]);
5     }
6     return reversedList;
7 }
8

```

```

9  const letters = ['a', 'b', 'c', 'd'];
10 const reversedLetters = reverse<string>(letters);
11 console.log(reversedLetters); // [ 'd', 'c', 'b', 'a' ]
12
13 const numbers = [1, 2, 3, 4];
14 const reversedNumbers = reverse<number>(numbers);
15 console.log(reversedNumbers); // [ 4, 3, 2, 1 ]

```

Primer iznad ilustruje implementaciju algoritma koji pravi kopiju niza proizvoljnog tipa podataka, pri čemu rezultujući niz sadrži elemente u obrnutom redosledu u odnosu na originalni niz. Primetimo da šablonska funkcija radi korektno i za `T = string` i za `T = number`. Dodatno, u primeru iznad, prilikom poziva šablonske funkcije nije bilo nužno navesti eksplicitan konkretan tip podataka za šablonski parametar `T`.

Šablonske klase

Šablonska klasa se definiše navođenjem šablonskih parametara nakon naziva klase u zagradama manje-veće. Šablonski parametri se dalje mogu koristiti za anotaciju parametara metoda, svojstava, povratnih vrednosti i lokalnih promenljivih u definiciji te klase. Naredni primer definiše šablonsku klasu `DomainId` koja definiše domen podatka za identifikatore:

```

1  class DomainId<T> {
2      constructor(private id: T) {
3      }
4
5      get value(): T {
6          return this.id;
7      }
8  }

```

Na osnovu ove šablonske klase, definisaćemo dve izvedene klase koje predstavljaju domen identifikatora za narudžbine i domen identifikatora za naloge, `OrderId` i `AccountId`, redom:

```

1  class OrderId extends DomainId<number> {
2      constructor(orderIdValue: number) {
3          super(orderIdValue);
4      }
5  }
6
7  class AccountId extends DomainId<string> {
8      constructor(accountIdValue: string) {
9          super(accountIdValue);
10     }
11 }

```

Naredni kod sadrži dve funkcije, `onlyAcceptsOrderId` i `acceptsAnyDomainId`, koje prihvataju argumente iz odgovarajućih domena. U pratećem kodu vidimo da poziv funkcije `onlyAcceptsOrderId` sa argumentom koji se ne uklapa proizvodi transpilatorsku grešku, dok funkcija `acceptsAnyDomainId` može da prihvati objekte iz proizvoljnog domena.

```

1  // Examples of compatibility
2  function onlyAcceptsOrderId(orderId: OrderId) {
3      console.log('Accepted an orderId: ' + orderId.value);
4  }
5
6  function acceptsAnyDomainId(id: DomainId<any>) {
7      console.log('Accepted id: ' + id.value);
8  }
9
10 const accountId = new AccountId('GUID-1');

```

```

11 const orderId = new OrderId(5);
12
13 // Error: Argument of type 'AccountId' is not assignable to parameter of type '
   //   OrderId'
14 // onlyAcceptsOrderId(accountId);
15
16 onlyAcceptsOrderId(orderId);
17 acceptsAnyDomainId(accountId);

```

Šablonski interfejsi

Da bismo kreirali šablonske interfejse, dovoljno je da šablonske parametre navedemo nakon naziva interfejsa u zagradama manje-veće. Šablonski parametri se dalje mogu koristiti bilo gde u definiciji tog interfejsa za anotiranje. Naredni primer ilustruje šablonski interfejs `Repository` koji definiše operacije koje je potrebno da svako skladište podataka sadrži. Operacija `getById` treba da na osnovu identifikatora podatka (šablonski parametar `TId`) dohvati objekat tog podatka (šablonski parametar `T`). Operacija `persist` treba da sačuva objekat novog podatka u skladište.

```

1 interface Repository<T, TId> {
2     getById(id: TId): T;
3     persist(model: T): void;
4 }

```

Želimo da napišemo skladište koje čuva informacije o našim kupcima. U tu svrhu, definišimo klasu `CustomerId` koja predstavlja jedinstveni identifikator svakog kupca i klasu `Customer` koja sadrži informacije o kupcu:

```

1 class CustomerId {
2     constructor(private customerIdValue: number) {
3     }
4
5     get value() {
6         return this.customerIdValue;
7     }
8 }
9
10 class Customer {
11     constructor(public id: CustomerId, public name: string) {
12     }
13 }
14
15 type CustomerKV = { 'key': CustomerId, 'value': Customer };

```

Primetimo da smo definisali alijas `CustomerKV` za objekat koji sadrži informaciju o identifikatoru kupca (svojstvo `key`) i informaciju o kupcu (svojstvo `value`). Mogli smo koristiti i klasu, ali za ovako jednostavan tip podataka je alijas dovoljan.

Prilikom definisanja klase `CustomerRepository` koja predstavlja skladište podataka o korisnicima i koja implementira šablonski interfejs, potrebno je da navedemo konkretne tipove za šablonske parametre interfejsa. Za `TId` ćemo koristiti `CustomerId`, dok ćemo za `T` koristiti `Customer`. Prilikom transpiliranja, telo klase `CustomerRepository` će biti provereno da li zadovoljava ograničenja koja interfejs postavlja nad konkretnim tipovima:

```

1 class CustomerRepository implements Repository<Customer, CustomerId> {
2     constructor(private customers: CustomerKV[] = []) {
3     }
4
5     getById(id: CustomerId): Customer {

```



```

6         const foundCustomer = this.customers.find(val => val.key.value === id.
            value);
7         if (foundCustomer) {
8             return foundCustomer.value;
9         }
10        return null;
11    }
12
13    persist(customer: Customer) {
14        for (const cmap of this.customers) {
15            if (cmap.key.value === customer.id.value) {
16                throw new Error('Duplicate Id detected!');
17            }
18        }
19
20        this.customers.push({ key: customer.id, value: customer });
21    }
22 }

```

Napišimo i deo koda koji testira celu implementaciju:

```

1 const customer1 = new Customer(new CustomerId(1245454122), 'Marija Begovic');
2 const customer2 = new Customer(new CustomerId(6904484054), 'Pera Peric');
3 const customer3 = new Customer(new CustomerId(1241574054), 'Ilija Mitrovic');
4
5 const repo = new CustomerRepository();
6 repo.persist(customer1);
7 repo.persist(customer2);
8 repo.persist(customer3);
9
10 console.log(repo.getById(customer2.id).name === 'Pera Peric'); // true
11
12 const customer4 = new Customer(new CustomerId(6904484054), 'Nikola Petrovic');
13
14 try {
15     repo.persist(customer4); // Ispalice izuzetak jer vec postoji identifikator
16 } catch (err) {
17     console.log('Doslo je do greske prilikom cuvanja korisnika');
18 }

```

3.4.3 Ograničenja tipa nad šablonskim parametrima

U opštem slučaju, šablonske parametre mogu zameniti proizvoljni tipovi podataka. Nekada ipak ovo ponašanje nije poželjno. Na sreću, moguće je postaviti *ograničenje tipa* (engl. *type constraint*) nad šablonskim parametrima bilo šablonskih funkcija, klasa ili interfejsa. Ograničenje tipa se izvodi pomoću ključne reči **extends** koja se navodi iza šablonskog parametra, a koju prati naziv klase, interfejsa ili anotacije tipa koja opisuje ograničenje koje se postavlja. Ako konkretan tip koji se navodi ne zadovoljava postavljeno ograničenje, transpilator će prijaviti grešku.

```

1 interface HasName {
2     name: string;
3 }
4
5 class Personalization {
6     static greet<T extends HasName>(obj: T) {
7         console.log('Hello, ' + obj.name);
8     }
9 }
10

```

```

11 class Human implements HasName {
12     constructor(public name: string) {
13     }
14 }
15
16 Personalization.greet(new Human('Max'));
17 Personalization.greet({ name: 'Max' });
18
19 // Error:
20 // Personalization.greet('Pricilla');

```

U ograničenju tipa je moguće navesti najviše jedan interfejs, klasu ili anotaciju tipa. Ovo se može prevazići tako što se definiše tip podataka (na primer, klasa) koji implementira veći broj drugih interfejsa, pri čemu svaki od interfejsa nameće odgovarajuća ograničenja koja moraju da važi za dati šablonski parametar. Na taj način, konkretan tip koji se bude koristio kao zamena šablonskog parametra mora da zadovoljava sva ograničenja koja nameću svi interfejsi.

3.5 Dekoratori

Kada je u pitanju organizacija koda, većina tehnika za organizaciju se može podeliti na horizontalne i vertikalne. Tipična horizontalna organizaciona tehnika je *n-slojna arhitektura* (engl. *n-tier architecture*), u kojoj se aplikacija deli na slojeve koji upravljaju nekom logičkom celinom kao što su korisnički interfejs, poslovna logika, upravljanje podacima, i sl. Sa druge strane, jedna od sve popularnijih vertikalnih organizacionih tehnika jeste *arhitektura zasnovana na mikroservisima* (engl. *microservice architecture*), u kojoj svaki vertikalni sloj predstavlja ograničeni kontekst, kao što su: "plaćanja", "kupci", "korisnici", i sl.

Pojam *dekoratora* (engl. *decorator*) u jeziku TypeScript je relevantan za oba tipa organizacionih tehnika, ali se pre svega ističe u vertikalnim arhitekturama. Dekoratori se mogu koristiti za upravljanje sveobuhvatnim zaduženjima aplikacije kao što su: logovanje, autorizacija operacija, validacija podataka, i sl. Ukoliko se koristi korektno, ovaj stil *aspektno-orijentisanog programiranja* može minimizovati broj linija koda koji je neophodan za implementiranje ovih deljenih zaduženja.

Sintaksa dekoratora je jednostavna. Iako dekoratori mogu dolaziti u različitom obliku, naredni primer prikazuje funkciju `log` koja predstavlja dekoratorsku funkciju, kao i primer njene upotrebe kao dekorator metoda `square` iz klase `Calculator`. Dekorator se primenjuje korišćenjem simbola `@`.

```

1 // Decorator Function
2 function log(target: any, key: string, descriptor: any) {
3     console.log('-----');
4     console.log(target);
5     console.log('-----');
6     console.log(key);
7     console.log('-----');
8     console.log(descriptor);
9     console.log('-----');
10 }
11
12 class Calculator {
13     // Using the decorator
14     @log
15     square(n: number) {

```

```

16         return n * n;
17     }
18 }
19
20 console.log('-----');
21 console.log('MAIN')
22 console.log('-----');
23
24 const c = new Calculator();
25 console.log(c.square(5));

```

Prikažimo ispis u konzoli za prethodni kod:

```

-----
Calculator { square: [Function] }
-----
square
-----
{
  value: [Function],
  writable: true,
  enumerable: true,
  configurable: true
}
-----
MAIN
-----
25

```

Kao što vidimo, funkcija `log` je ispisala u konzoli neke informacije o metodi `square` klase `Calculator` i to prilikom definisanja tog metoda, a ne prilikom poziva tog metoda, zato što se metod `square` poziva nakon ispisa teksta `MAIN` u konzoli! Ovo je važna napomena koju treba imati na umu pri radu sa dekoratorima.

N TypeScript dekoratori predstavljaju eksperimentalni dodatak, te je potrebno navesti zastavicu `experimentalDecorators` prilikom transpiliranja. Dodatno, ciljna verzija JavaScript jezika mora biti postavljena na barem ECMAScript 5:

```
tsc --target ES5 --experimentalDecorators
```

Alternativno, ukoliko se koristi TypeScript projekat, potrebno je dodati naredne opcije u `tsconfig.json`:

```

{
  "compilerOptions": {
    "target": "ES5",
    "experimentalDecorators": true
  }
}

```

3.5.1 Konfigurabilni dekoratori

Dekoratori se mogu konfigurisati tako što se dekoratorska funkcija pretvori u *dekoratorsku fabriku* (engl. *decorator factory*). Dekoratorska fabrika je funkcija koja vraća dekoratorsku funkciju i koja može imati proizvoljan broj parametara koji se mogu koristiti za konfigurisanje dekoratorske funkcije koja se vraća kao njena povratna vrednost.

U narednom primeru, definisana je dekoratorska fabrika `log` koja pravi različite dekoratorske funkcije u zavisnosti od parametra `memberType`. U zavisnosti od tog parametra, poruka koja se loguje će imati različite vrednosti.

```

1 function log(memberType: string) {
2     if (memberType === 'STATIC') {
3         return function(target: any, key: string, descriptor: any) {
4             console.log(`Successfully decorated static member ${key} of class $
                        {(<Function>target).name}`);
5         };
6     }
7     return function(target: any, key: string, descriptor: any) {
8         console.log(`Successfully decorated instance member ${key}`);
9     };
10 }
11
12 class Calculator
13 {
14     // Using the configurable decorator
15     @log('NONSTATIC')
16     square(num: number) {
17         return num * num;
18     }
19
20     // Using the configurable decorator
21     @log('STATIC')
22     static doubled(num: number) {
23         return 2 * num;
24     }
25 }

```

Prikažimo ispis u konzoli:

```

Successfully decorated instance member square
Successfully decorated static member doubled of class Calculator

```

3.5.2 Kompozicija dekoratora

Na neku deklaraciju je moguće primeniti više dekoratora, i to:

- U jednoj liniji:

```
1 @f @g x
```

- U više linija:

```

1 @f
2 @g
3 x

```

Bez obzira koji je stil pisanja koda koristi, efekat je isti. Kada se dekoratori primene na istu deklaraciju, njihovo izračunavanje odgovara kompoziciji funkcija u matematičkim izračunavanjima. Iz gornjih primera, prilikom kompozicija funkcija `f` i `g`, rezultujući kompozit je ekvivalentan pozivu `f(g(x))`.

Dakle, prilikom izračunavanja većeg broja dekoratora nad jednom deklaracijom u jeziku TypeScript, dolazi do izvršavanja narednih koraka:

- Izrazi za svaki dekorator se izračunavaju odozgo nadole.
- Rezultati tih izračunavanja se zatim pozivaju kao funkcije odozdo nagore.

Da bismo ilustrovali ovo ponašanje, kreiraćemo dekoratorske fabrike `f` i `g`. Ove dve funkcije redom anotiraju isti metod `method` klase `C`:

```
1 function f()
2 {
3     console.log("f(): evaluated");
4     return function (target, propertyKey: string, descriptor:
5         PropertyDescriptor)
6     {
7         console.log("f(): called");
8     }
9 }
10 function g()
11 {
12     console.log("g(): evaluated");
13     return function (target, propertyKey: string, descriptor:
14         PropertyDescriptor)
15     {
16         console.log("g(): called");
17     }
18 }
19 class C
20 {
21     @f()
22     @g()
23     method() {}
24 }
```

Prikažimo izlaz u konzoli:

```
f(): evaluated
g(): evaluated
g(): called
f(): called
```

3.5.3 Tipovi dekoratora

Jezik TypeScript razlikuje nekoliko tipa dekoratora na osnovu toga koji je koncept jezika TypeScript na koji se neki dekorator primenjuje. Tako postoje:

- Dekoratori klasa
- Dekoratori metoda
- Dekoratori očitavača/postavljača
- Dekoratori svojstava
- Dekoratori parametara

Svaki tip dekoratora zahteva drugačiji potpis funkcije zbog toga što se dekoratoru prosleđuju različiti parametri u zavisnosti od tipa dekoratora. U nastavku ćemo prikazati praktične primere za svaki tip dekoratora.

Videćemo da dekoratorske funkcije očekuju parametre koje ustaljeno zovemo `target`, `key` i `descriptor`. Parametar `target` će u zavisnosti od tipa dekoratora predstavljati konstruktorsku funkciju, objekat nad kojim se metod poziva ili odgovarajući prototip, `key` će predstavljati ime bilo klase, metode ili svojstva, dok će `descriptor`, za svaki od dekoratora, predstavljati deskriptorski objekat sa svojstvima `value`, `writable`, `enumerable` i `configurable` preko kojeg će biti moguće menjanje objekata nad kojim se primenjuju.

Postoji precizno definisan redosled izvršavanja ukoliko postoji više dekoratora unutar jedne klase. Ovaj redosled je opisan narednim koracima:

1. Za svaki nestatički član klase izvršava se dekorator parametara koji je praćen dekoratorom metoda, dekoratorom očitavača/postavljača ili dekoratorom svojstva.
2. Za svaki statički član klase izvršava se dekorator parametara koji je praćen dekoratorom metoda, dekoratorom očitavača/postavljača ili dekoratorom svojstva.
3. Izvršava se dekorator parametara za konstruktor.
4. Izvršava se dekorator klase.

Dekoratori klasa

Dekorator klase se navodi tik ispred deklaracije klase. Dekorator klase se primenjuje nad konstruktorom klase i može se koristiti za posmatranje, modifikaciju ili zamenu u definiciji klase. Dekorator klase će biti pozvan kao funkcija tokom faze izvršavanja, a njegovi argumenti su dati u nastavku:

1. Konstruktor dekorisane klase.

Ako dekorator klase ima povratnu vrednost, onda će ona zameniti deklaraciju klase sa datom konstruktorskom funkcijom. Ukoliko želimo da vratimo novu konstruktorsku funkciju, onda je neophodno da vodimo računa da se originalni prototip održao.

Naredni primer definiše dekorator klasa `sealed` koji, kada se primeni na neku klasu, zabranjuje dalje menjanje konstruktora i prototipa pomoću `Object.seal`³.

```

1 // Transpilirati sa verzijom ECMAScript 6
2 // da bi se izracunalo ime klase u dekoratoru:
3 // tsc --target ES6 --experimentalDecorators
4
5 function sealed(constructor: Function) {
6     console.log(constructor.name + ' is being sealed');
7
8     Object.seal(constructor);
9     Object.seal(constructor.prototype);
10 }
11
12 @sealed
13 class Greeter {
14     greeting: string;
15
16     constructor(message: string) {
17         this.greeting = message;
18     }
19
20     greet() {
21         console.log('Hello, ' + this.greeting);
22     }
23 }
24
25 console.log('-----');
26 console.log('MAIN')
27 console.log('-----');
28
29 const louieGreeter = new Greeter('Louie');
30 louieGreeter.greet();
31
```

³Više o upotrebi funkcije `Object.seal` se može pronaći na vezi https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/seal.

```
32 const mayaGreeter = new Greeter('Maya');
33 mayaGreeter.greet();
34
35 mayaGreeter.constructor.prototype.randomNumber = 5;
36 console.log(mayaGreeter.constructor.prototype.randomNumber); // undefined
```

Dekoratori metoda

Dekorator metoda se navodi tik ispred deklaracije metoda. Dekorator se primenjuje na deskriptorski objekat metoda i može se koristiti za posmatranje, modifikaciju ili zamenu definicije metoda. Dekorator metoda će biti pozvan kao funkcija tokom faze izvršavanja, a njegovi argumenti su dati u nastavku:

1. Ili konstruktorska funkcija u slučaju statičkih metoda ili prototip klase za nestatičke metode.
2. Naziv metoda koji se dekoriše.
3. Deskriptorski objekat za dati metod. Ukoliko cilj transpiliranja nije barem ECMAScript 5, ova vrednost će biti `undefined`.

Ako dekorator metoda ima povratnu vrednost, onda će ta vrednost biti korišćena kao deskriptorski objekat za dati metod. Ukoliko cilj transpiliranja nije barem ECMAScript 5, povratna vrednost dekoratora metoda će biti ignorisana.

Da bismo ilustrovali ovo ponašanje, definišimo dekoratorsku fabriku `enumerable` koja će vratiti dekorator metoda koji postavlja svojstvo `enumerable` metoda na `true` ili `false`, u zavisnosti od prosleđenog argumenta dekoratorske fabrike.

```
1 function enumerable(value: boolean) {
2   return function (target: any, propertyKey: string, descriptor:
3     PropertyDescriptor) {
4     descriptor.enumerable = value;
5   };
6 }
7 class Greeter {
8   greeting: string;
9   constructor(message: string) {
10    this.greeting = message;
11  }
12
13  @enumerable(false)
14  greet() {
15    return "Hello, " + this.greeting;
16  }
17 }
18
19 const mayaGreeter = new Greeter('Maya');
20
21 let found = false;
22 for (const key in mayaGreeter) {
23   if (key === 'greet') {
24     found = true;
25     break;
26   }
27 }
28 console.log('Is it found? ' + (found ? 'Yes' : 'No')); // Is it found? No
```

Dekoratori očitavača/postavljača

Dekoratori očitavača/postavljača imaju istu logiku kao dekoratori metoda, sa jednom dodatnom razlikom. TypeScript zabranjuje dekorisanje i očitavača i postavljača za isto svojstvo. Umesto toga, svi dekoratori očitavača/postavljača se moraju primeniti na ono koje se prvo javi u dokumentu. Ovo i ima smisla s obzirom da, ukoliko želimo da dekorišemo neko svojstvo i za čitanje i za pisanje, umesto da koristimo dekoratore očitavača/postavljača, možemo jednostavno koristiti dekorator svojstva, koji je opisan u nastavku.

Dekoratori svojstva

Dekorator svojstva se navodi tik ispred deklaracije svojstva. Dekorator svojstva će biti pozvan kao funkcija tokom faze izvršavanja, a njegovi argumenti su dati u nastavku:

1. Ili konstruktorska funkcija u slučaju statičkih metoda ili prototip klase za nestatičke metode.
2. Naziv metoda koji se dekoriše.

Povratna vrednost dekoratora svojstva se uvek ignoriše. Naredni primer ilustruje kako je moguće kreirati dekorator svojstva koji definiše funkcije za očitavanje i postavljanje vrednosti, pri čemu se dato svojstvo uklanja iz objekta.

```
1 function log(target: any, key: string) {
2     let value = target[key];
3
4     // Replacement getter
5     const getter = function () {
6         console.log(`[LOG] Getter for ${key} returned ${value}`);
7         return value;
8     };
9
10    // Replacement setter
11    const setter = function (newVal) {
12        console.log(`[LOG] Set ${key} to ${newVal}`);
13        value = newVal;
14    };
15
16    // Replace the property
17    if (delete this[key]) {
18        Object.defineProperty(target, key, {
19            get: getter,
20            set: setter,
21            enumerable: true,
22            configurable: true
23        });
24    }
25 }
26
27 class Greeter {
28     @log
29     greeting: string;
30
31     constructor(message: string) {
32         this.greeting = message; // setter
33     }
34
35     greet() {
36         console.log('Hello, ' + this.greeting); // getter
37     }
38 }
39
```



```

40 const louieGreeter = new Greeter('Louie');
41 louieGreeter.greet();

```

Prikažimo i ispis u konzoli:

```

[LOG] Set greeting to Louie
[LOG] Getter for greeting returned Louie
Hello, Louie

```

Dekoratori parametara

Dekorator parametra se navodi tik ispred deklaracije parametra. Dekorator se primenjuje na funkciju koja predstavlja ili konstruktor klase ili deklaraciju metoda. Dekorator parametra će biti pozvan kao funkcija tokom faze izvršavanja, a njegovi argumenti su dati u nastavku:

1. Ili konstruktorska funkcija u slučaju statičkih metoda ili prototip klase za nestatičke metode.
2. Naziv metoda koji se dekoriše.
3. Redni broj parametra u listi parametara funkcije.

Dekorator parametra se može koristiti samo za posmatranje da je parametar definisan nad metodom. Povratna vrednost dekoratora parametra se ignoriše.

Naredni primer ilustruje dekorator parametra `notnull` koji služi za proveru da li je vrednost nekog parametra metoda `null`. Zapravo, dekorator samo sakuplja informaciju o metodima koji imaju parametre koji ne smeju imati vrednost `null` u niz `notNullableArgs`. Logika koja vrši proveru se nalazi u dekoratoru metoda `validate`, koji, primenjen na metod, zamenjuje taj metod funkcijom koja prvo vrši proveru, a zatim izvršava metod. U slučaju da neki od dekorisanih parametara ima vrednost `null` (na osnovu prethodno sakupljenih informacija iz niza `notNullableArgs`), biće ispaljen izuzetak. U suprotnom, metod se poziva sa prosleđenim argumentima.

```

1  const notNullableArgs: { target: any, propertyKey: string | symbol,
   parameterIndices: number[] }[] = [];
2
3  function notnull(target: Object, propertyKey: string | symbol, parameterIndex:
   number) {
4      let found = false;
5      for (const nna of notNullableArgs) {
6          if (nna.target === target && nna.propertyKey === propertyKey) {
7              nna.parameterIndices.push(parameterIndex);
8              found = true;
9              break;
10         }
11     }
12     if (!found) {
13         notNullableArgs.push({ target, propertyKey, parameterIndices: [
14             parameterIndex ] });
15     }
16 }
17 function validate(target: any, propertyKey: string, descriptor:
   PropertyDescriptor) {
18     let method = descriptor.value;
19
20     descriptor.value = function () {
21         for (const nna of notNullableArgs) {
22             if (nna.target === target && nna.propertyKey === propertyKey) {

```

```

23         for (let i = 0; i < arguments.length; ++i) {
24             if (nna.parameterIndices.indexOf(i) !== -1 && arguments[i]
25                 === null) {
26                 throw new Error(`The method call ${propertyKey} is
27                     missing required argument ${i}.`);
28             }
29         }
30     }
31     }
32     return method.apply(this, arguments);
33 }
34 }
35
36 class Caller {
37     private lastPhoneNumber: string;
38     private lastPrefix: string;
39
40     @validate
41     callMe(@notnull phoneNumber: string, @notnull prefix: string, callerName:
42         string) {
43         this.lastPhoneNumber = phoneNumber;
44         this.lastPrefix = prefix;
45
46         console.log('Calling number ' + this.getLastCalledNumber());
47     }
48
49     @validate
50     getLastCalledNumber(): string {
51         return this.lastPrefix + '/' + this.lastPhoneNumber;
52     }
53 }
54
55 const c = new Caller();
56
57 c.callMe('1245655', '011', null); // OK
58 // c.callMe('1245655', null, null); // Error
59 console.log(c.getLastCalledNumber());

```

Literatura za ovu oblast

- [Fen18] Steve Fenton. *Pro TypeScript: Application-Scale JavaScript Development*. 2nd. Berkely, CA, USA: Apress, 2018. ISBN: 9781484232484.
- [Mic] Microsoft. *TypeScript — JavaScript that scales*. URL: <https://www.typescriptlang.org/index.html>.

4. Reaktivna paradigma

U prethodnim poglavljima smo se susretali sa pojmovima poput "asinhrono programiranje" i "posmatrač". Videli smo različite načine na koje je moguće izvršiti asinhronu akciju u programskom jeziku JavaScript, dok smo se sa posmatračem samo posredno susreli, bez ulaženja u detalje. Cilj ovog poglavlja jeste produbljivanje ovih termina kroz praktične primere. Na početku, govorićemo o jednoj veoma zanimljivoj upotrebi po-potrebi-pozivnih funkcija, a to je u eliminaciji klasičnih `for` petlji. JavaScript petlje se mogu izvršavati isključivo sinhrono, i ne mogu se koristiti za ponavljanje asinhronih operacija. Jednom kada savladamo programiranje bez petlji, upoznaćemo se detaljnije sa obrascem za projektovanje koji se naziva posmatrač, kao i koja je njegova upotrebna vrednost i zašto ga treba razumeti. Na samom kraju, upoznaćemo se sa bibliotekom RxJS za rad sa posmatračima, što će nam pomoći da savladamo reaktivnu paradigmu programiranja i bolje razumemo koncepte u Angular okruženju za razvoj koji se oslanjaju na nju.

4.1 Asinhrono programiranje metodom eliminacije petlji

Pretpostavimo da imamo listu objekata koji predstavljaju studente koji imaju broj indeksa, ime i prezime, odnosno, neka su nam na raspolaganju objekti klase `Student`:

```
1 class Student
2 {
3     constructor(indeks, ime, prezime)
4     {
5         this.indeks = indeks;
6         this.ime = ime;
7         this.prezime = prezime;
8     }
9 }
10
11 let studenti = [
12     new Student('1/2017', 'Pera', 'Peric'),
13     new Student('2/2017', 'Jovana', 'Jovanovic'),
14     new Student('3/2017', 'Nikola', 'Nikolic'),
```

```
15  ];
```

4.1.1 Asinhroni metodi za obradu nizova

Neka nam je zadatak da napišemo funkciju `dohvatiIndekse` čiji je argument niz studenata, a čiji je zadatak da vrati niz koji se sastoji samo od indeksa tih studenata. Jedan način da to uradimo jeste pomoću `for` petlje:

```
1  function dohvatiIndekse(studenti)
2  {
3      let indeksi = [], student;
4
5      for (let i = 0; i < studenti.length; ++i)
6      {
7          student = studenti[indeks];
8          indeksi.push(student.indeks);
9      }
10
11     return indeksi;
12 }
```

U ovom primeru prethodni kod radi savršeno dobro, međutim, ima jednu manu koja nije očigledna. Klasična `for` petlja radi isključivo sinhrono, što znači da, ako bismo u asinhronom okruženju dobavljali informacije o studentima, onda ne bismo mogli da je koristimo.

Alternativa se sastoji u korišćenju metoda `forEach` kojeg poseduju svi `Array` objekti. Ovaj metod kao argument uzima po-potrebi-pozivnu funkciju `callback(currentValue[, index[, array]])`, koja može imati barem jedan, a najviše tri argumenta, čije su vrednosti:

- promenljiva za koju će biti vezana vrednost tekućeg elementa niza,
- promenljiva za koju će biti vezan indeks tekućeg elementa niza,
- promenljiva za koju će biti vezan sam niz.

Telo ove funkcije se izvršava za svaku element niza, redom. Tako prethodni primer možemo refaktorisati u:

```
1  function dohvatiIndekse(studenti)
2  {
3      let indeksi = [];
4
5      studenti.forEach(student => {
6          indeksi.push(student.indeks);
7      });
8
9      return indeksi;
10 }
```

Ako se malo detaljnije pozabavimo prethodnim primerom, vidimo da smo formulaciju zadatka da od niza studenata konstruišemo novi niz koji sadrže indekse tih studenata, mogli da preformulišemo tako da glasi da niz studenata transformišemo u niz njihovih indeksa. Ponovo, kao u prethodnom primeru, mogli bismo da napišemo `for` petlju kojom ćemo prolaziti kroz elemente niza, izvršiti transformaciju nad njima i čuvati transformisanu verziju u novi niz, i na kraju vratiti novi niz. Ono što se ispostavlja jeste da postoji brži način da to kodiramo, a to je korišćenjem metoda `map`.

Argument metoda `map` je po-potrebi-pozivna funkcija čiji je potpis isti kao i u slučaju `forEach` metoda. Metod `map` poziva funkciju `callback` za svaki element niza nad kojim se

poziva i konstruiše novi niz od rezultata. Niz se konstruiše tako što se umesto elementa koji se obrađuje, na njegovu poziciju u novom nizu smešta vrednost koju vraća funkcija `callback`.

Prethodni primer se korišćenjem ovog metoda može refaktorisati u:

```
1 function dohvatiIndekse(studenti)
2 {
3     return studenti.map(student => {
4         return student.indeks
5     });
6 }
```

Jedan vrlo čest zadatak u programiranju jeste da izdvojimo sve elemente nekog niza koji zadovoljavaju neki kriterijum. Na primer, neka je potrebno izdvojiti studente čiji je indeks iz 2018. godine. Umesto korišćenja `for` petlje, možemo iskoristiti metod `filter` definisan nad nizovima, koji uzima argument funkciju kao i `forEach` i `map`, a koja predstavlja test koji će biti primenjen nad svakim elementom niza. Ukoliko funkcija vrati vrednost `true`, taj element će biti deo rezultujućeg niza, a ukoliko funkcija vrati vrednost `false`, onda taj element neće biti deo rezultujućeg niza. U narednom primeru vidimo da funkcija `dohvati2018` za svaki element niza prvo izvlači informaciju u godini iz indeksa, a zatim je poredi sa `'2018'` da bi izvršila test da li tekući student treba da se pronade u rezultujućem nizu ili ne:

```
1 function dohvati2018(studenti)
2 {
3     return studenti.filter(student => {
4         let godina = student.indeks.substr(student.indeks.indexOf('/') + 1);
5         return godina === '2018';
6     });
7 }
```

4.1.2 Lančanje asinhronih metoda

Kao što smo rekli, povratne vrednosti metoda `map` i `filter` su novokonstruisani nizovi, što znači da nad njima možemo ponovo primeniti iste funkcije, kao i metod `forEach`. Ovo nam omogućava da kreiramo kompleksnije programe, a da pritom ne napišemo nijednu `for` petlju. Na primer, neka je potrebno od niza studenata ispisati imena i prezimena onih čiji je indeks iz 2018. godine. Naredni fragment programskog koda radi upravo to:

```
1 studenti
2     .filter(student => {
3         let godina = student.indeks.substr(student.indeks.indexOf('/') + 1);
4         return godina === '2018';
5     })
6     .map(student => {
7         return `${student.ime} ${student.prezime}`;
8     })
9     .forEach(imePrezime => {
10         console.log(imePrezime);
11     });
```

Već sada možemo da vidimo izuzetnu moć ovih metoda. Naravno, ukoliko bi nam filterovan i transformisan niz bio potreban dalje, mogli smo ga prvo sačuvati u promenljivu, pa onda nad njom pozvati `forEach` metod. Iako nije greška izvršiti prvo `map`, pa onda `filter` metod, više ima smisla smanjiti dimenziju niza pomoću metoda `filter`, pa onda primeniti transformaciju nad tim, smanjenim nizom.

4.1.3 Rad sa ugnežđenim strukturama

Često imamo situaciju da radimo sa strukturama podataka koje su ugneždene jedne u druge, na primer, da radimo sa nizovima nizova. U takvim situacijama, nekada nam je značajno da ugneždene nizove spojimo u jedan niz. Na primer, pretpostavimo da imamo smerove na fakultetu, implementirane kao niz nizova studenata:

```
1 let smerovi = [  
2   [  
3     new Student('1/2017', 'Pera', 'Peric'),  
4     new Student('3/2017', 'Nikola', 'Nikolic')  
5   ],  
6   [  
7     new Student('2/2017', 'Jovana', 'Jovanovic'),  
8     new Student('4/2017', 'Ana', 'Nikolic'),  
9     new Student('5/2017', 'Mirjana', 'Lucic'),  
10    new Student('6/2017', 'Stefan', 'Jovanovic'),  
11  ]  
12 ];
```

Neka nam je zadatak da ispišemo indekse svih studenata na svim smerovima. Jedan pristup jeste pomoću ugnežđenih `forEach` metoda, slično kao da smo pisali ugneždene `for` petlje:

```
1 smerovi.forEach(smer => {  
2   smer.forEach(student => {  
3     console.log(student.indeks);  
4   });  
5 });
```

Međutim, ukoliko bi trebalo da sada ispišemo imena i prezimena svih studenata na svim smerovima, morali bismo ponovo da koristimo isti princip, što bi dovelo do dupliranja koda. Nažalost, za razliku od `map` i `filter` metoda, nemamo postojeću funkciju koja bi ovo uradila za nas. Na sreću, JavaScript prototipi se mogu menjati, pa možemo mi sami da je napišemo:

```
1 Array.prototype.applyAll = function(callback) {  
2   let results = [];  
3  
4   this.forEach(subArray => {  
5     subArray.forEach(element => {  
6       results.push(callback(element));  
7     });  
8   });  
9  
10  return results;  
11 };
```

Sada se prethodni kod pretvara u

```
1 smerovi.applyAll(student => {  
2   console.log(student.indeks);  
3 });
```

a ispisivanje svih imena i prezimena u

```
1 smerovi.applyAll(student => {  
2   console.log(`${student.ime} ${student.prezime}`);  
3 });
```

Primetimo dve stvari:

1. Definicija metoda `applyAll` zahteva da koristimo standardnu notaciju funkcije kao vrednosti umesto lambda funkcije jer lambda funkcije nemaju svoju vrednost `this`, već posmatraju svoju okolinu. Očigledno, ako bismo u ovom slučaju koristili lambda funkciju, vrednost za `this` bi bio prazan objekat.
2. Definicija metoda `applyAll` konstruiše novi niz od povratnih vrednosti koje će vratiti funkcija `callback` pri svakom pozivu. Ovim smo omogućili da možemo koristiti metod `applyAll` i za transformaciju svih elemenata u ugneždjenim nizovima. Na primer, naredni fragment koda će ispisati niz indeksa svih studenata na svim smerovima:

```
1 console.log(smerovi.applyAll(student => student.indeks));
```

Ukoliko u našim programima nećemo koristiti povratne vrednosti `callback` poziva, ona ovo predstavlja utoliko značajno usporenje što više primenjujemo metod `applyAll`.

Napišimo i funkciju `concatAll` koja jednostavno smanjuje nivo ugnežđenosti za jedan:

```
1 Array.prototype.concatAll = function() {
2   let results = [];
3
4   this.forEach(subArray => {
5     subArray.forEach(element => {
6       results.push(element);
7     });
8   });
9
10  return results;
11 };
```

Nizovi svoje podatke čuvaju u memoriji. Zbog toga, ti podaci su nam odmah dostupni i metodi poput `forEach` se zapravo izvršavaju sinhrono. To znači da ako želimo da izvršimo neki kod po završetku obrade niza, dovoljno je da taj kod smestimo nakon poziva `forEach` metoda. Slično, obrada grešaka se svodi na obuhvatanje `forEach` metoda u `try/catch` blokove. Zbog čega je onda cela ova priča do sada bila ispričana? Odgovor leži u tome da se rad sa asinhronim nizova podataka zasniva na promeni razmišljanja, odnosno, na tome da umesto da razmišljamo o klasičnoj `for` petlji, treba da razumemo i druge načine za obrađivanje nizova podataka. Do kraja poglavlja ćemo razumeti zašto je bitno da smo razumeli novi način razmišljanja.

4.2 Uvod u reaktivno programiranje

Reaktivno programiranje je programiranje sa asinhronim tokovima podataka. Sa nekim od asinhronih akcija smo se susretali kada smo programirali osluškivače na razne DOM događaje. Reaktivna paradigma uzima ovaj koncept i uopštava ga — možemo kreirati tokove podataka od bilo čega, ne samo od klikova miša ili prelaženja kursora preko elementa. Tokovi mogu biti: promenljive, korisnički unosi, svojstva, strukture podataka, HTTP zahtevi i odgovori, i dr. Na primer, objave na našim Instagram nalozima se mogu posmatrati kao tokovi na isti način kao i klikovi. Možemo osluškivati taj tok i reagovati na njega.

Da stvar bude još bolja, tokovi mogu koristiti koncepte iz funkcionalnog programiranja — jedan dok može biti ulaz u drugi tok. To nam daje na raspolaganje moćne funkcije za izvršavanje najrazličitih operacija nad tokovima kao što su:

- spajanje dva toka u jedan
- filtriranje tokova po raznim uslovima radi izdvajanja događaja za koje smo zainteresovani

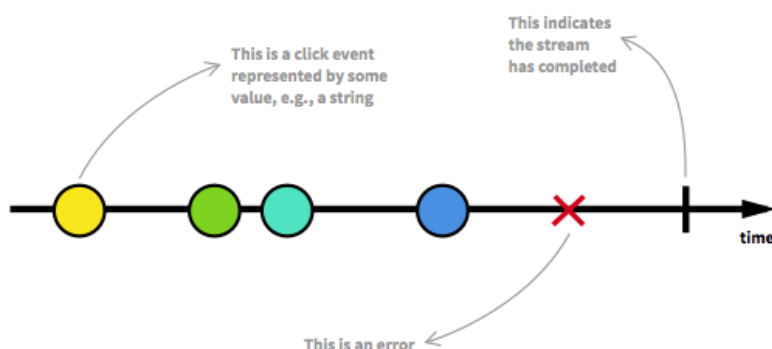
- transformisanje vrednosti u toku u tok sa drugim vrednostima
- i mnogi drugi

Sve vreme govorimo o tokovima, a još uvek ih nismo uveli. Pa, hajde to da uradimo sada:

Definicija 4.2.1 — Tok. *Tok* (engl. *stream*) predstavlja sekvencu događaja koji se dešavaju, uređenih po vremenu u kojem su se desili.

Tok može da emituje tri vrste događaja, što je ilustrovano na slici 4.1:

1. vrednost odgovarajućeg tipa,
2. grešku, ukoliko do nje dođe,
3. signal da je tok završen.



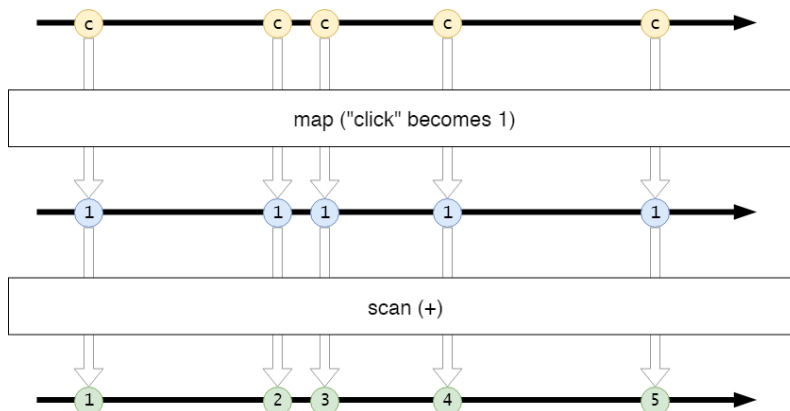
Slika 4.1: Grafički prikaz primera toka i događaja koje on može da emituje.

Mi smo u stanju da reagujemo na ove događaje isključivo asinhrono, definisanjem jedne funkcije koje će se izvršiti kada je vrednost emitovana, druge funkcije kada je greška emitovana i treće funkcije kada je završni signal emitovan. Naravno, reagovanje se vrši ako smo prvo rekli da želimo da "oslušujemo" šta se emituje na toku. Uvedimo novu terminologiju za ovo.

Definicija 4.2.2 — Pretplaćivanje, posmatrač i subjekat. "Osluškiivanje" nekog toka se naziva *pretplaćivanje* (engl. *subscribe*). Funkcije koje mi definišemo su *posmatrači* (engl. *observers*). Sam tok je *subjekat* (engl. *subject* ili *observable*) koji se posmatra.

Ovim je opisan obrazac sa projektovanje "Posmatrač". Za više detalja, pogledati izvor https://en.wikipedia.org/wiki/Observer_pattern.

Jedan jednostavan primer rada sa tokovima je dat u nastavku. Pretpostavimo da imamo dugme i tok koji emituje događaj `'click'` svaki put kada se klikne na dugme. Želimo da ovaj tok pretvorimo u tok koji emituje broj koliko je puta dugme kliknuto. Na slici 4.2 data je ilustracija ovog procesa. Prvo se početni tok klikova transformiše u tok jedinica pomoću operatora `map`, i dobija se novi tok (početni ostaje nepromenjen). Operator `map(f)` primenjuje funkciju transformacije `f` za svaki emitovan događaj. U našem slučaju, funkcija `f` je funkcija koja će svaki klik jednostavno mapirati u broj 1. Funkcija `scan(g)` agregira sve prethodne vrednosti u toku i proizvodi vrednost `x = g(akumulirano, tekuće)`, gde je `g` jednostavna funkcija sabiranja u ovom slučaju. Zbog toga, poslednji tok će emitovati



Slika 4.2: Grafički prikaz transformacije toka klikova mišem u tok brojeva.

ukupan broj klikova svaki put kada se emituje događaj klika u prvom toku.

Dajmo još jedan primer. Neka imamo isti tok klikova kao u prethodnom primeru i neka želimo da napravimo tok događaja "dvostrukih klikova" od njega. Zapravo, neka je zadatak da se svi višestruki klikovi tretiraju kao dvostruki klikovi u novom toku. U tradicionalnom pristupu imperativnog programiranja, bilo bi nam veoma naporno da ovo izvedemo. Pogledajmo reaktivni pristup ovom problemu. Na slici 4.3 data je ilustracija ovog procesa.

Prvo akumuliramo pojedinačne klikove koji su dovoljno blizu jedni drugih u liste klikova, pri čemu se za granicu između listi bira "tišina" od `250ms` ("tišina" u ovom kontekstu označava da nema emitovanih događaja u tom periodu). Zatim se tok listi transformiše u tok gde svaki događaj odgovara broju elemenata u listi iz početnog toka. Konačno, možemo da filtriramo sve one događaje koji emituju vrednosti veće ili jednake od 2, što zapravo znači da su upravo tada nastali višestruki klikovi mišem.

Ono što je fenomenalno saznanje je da, bez ulaženja u detalje, cela opisana logika se može implementirati u samo 4 linije koda, koje dajemo ovde radi kompletnosti i motivisanja čitaoca pre nego što se upustimo u jednu popularnu biblioteku za reaktivno programiranje, RxJS.

```
1 var multiClickStream = clickStream
2   .buffer(function() { return clickStream.throttle(250); })
3   .map(function(list) { return list.length; })
4   .filter(function(x) { return x >= 2; });
```

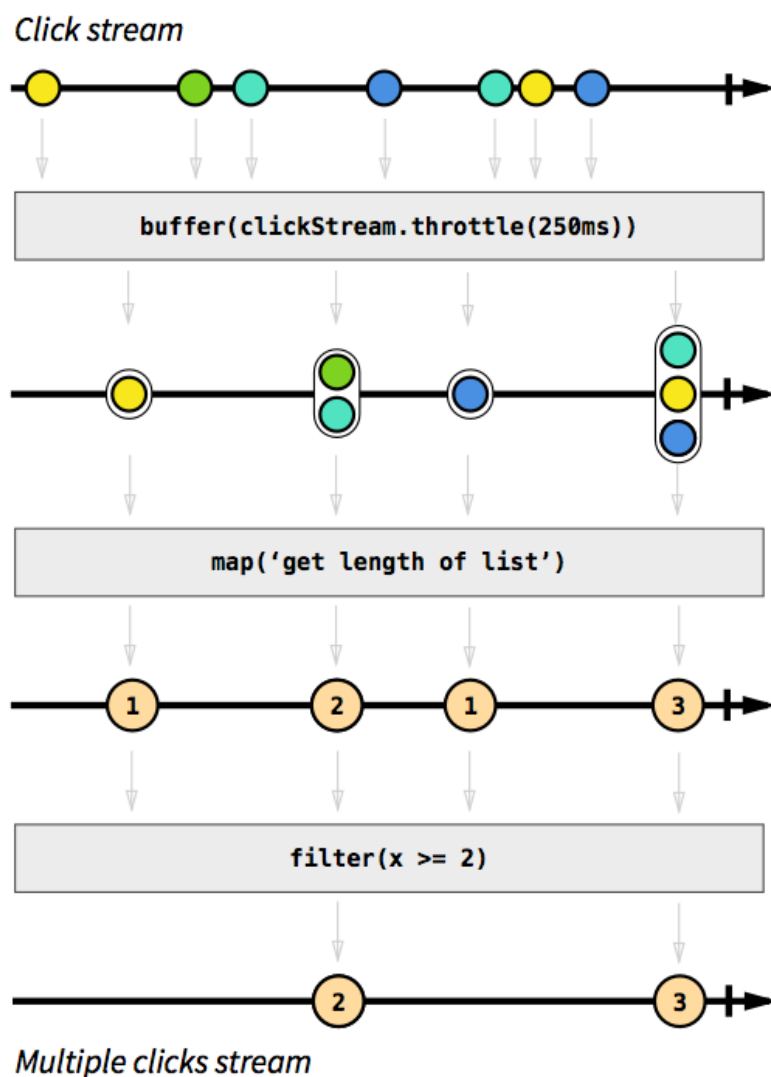
4.3 Biblioteka RxJS

U biblioteci RxJS, tokovi su implementirani kroz tip podataka koji se naziva **Observable**. Iako se u dokumentaciji tokovi tako nazivaju, mi ćemo nastaviti da ih zovemo tokovima, ali ovo ne bi trebalo da bude zbunjujuće. Zvanična veb prezentacija sa dokumentacijom biblioteke i drugim informacijama je dostupna na adresi <https://rxjs-dev.firebaseapp.com/>. Da bismo učitali biblioteku možemo je ili instalirati pomoću NPM:

```
$ npm install rxjs
```

ili iskoristiti mrežu za dostavljanje sadržaja (engl. *content delivery network*, skr. CDN) UNPKG:

```
1 <script src='https://unpkg.com/@reactivex/rxjs@6.3.3/dist/global/rxjs.umd.js'
  ></script>
```



Slika 4.3: Grafički prikaz transformacije toka klikova mišem u tok višestrukih klikova mišem.

Ukoliko smo biblioteku instalirali, možemo je uvesti kao ES6 modul:

```
1 import * as rxjs from 'rxjs';
```

ili kao CommonJS modul:

```
1 const rxjs = require('rxjs');
```

Ukoliko smo biblioteku ugnježili kao deo HTML stranice, onda su nam klase, funkcije i ostali elementi biblioteke dostupni kroz objekat `rxjs`.

4.3.1 Tok u RxJS biblioteci

Funkcionalnost toka je u RxJS biblioteci implementirana kroz šablonsku klasu `Observable` `<T>`, i terminologija u literaturi na engleskom koja se koristi pri radu sa ovom bibliotekom je *observable*, mada ćemo mi i dalje govoriti *tok*. Kao što smo rekli, o tokovima možemo razmišljati kao o asinhronim nizovima. Da bismo dobili ove vrednosti kada one pristignu

asinhronim operacijama, potrebno je da se pretplatimo. Za pretplaćivanje možemo koristiti funkciju `subscribe`, čiji je argument objekat, koji nazivamo *posmatrač*, a koji implementira interfejs `Observer` (<https://rxjs-dev.firebaseapp.com/api/index/interface/Observer>), ili samo tri funkcije, koje imaju naredna značenja:

- funkcija `next` koja prima argument `value: T` biće izvršena svaki put kada nova vrednost bude emitovana u toku.
- funkcija `error` koja prima argument `err: any` biće izvršena svaki put kada se emituje greška u toku.
- funkcija `complete` biće izvršena jednom, kada tok prestaje da emituje vrednosti. Naravno, moguće je da neki tokovi nikada neće prestati da emituju vrednosti (na primer, tok događaja klika miša na nekom dugmetu).

Naredni primeri ilustruju pretplaćivanje na tok dat promenljivom `tokPrimer$` (koji emituje vrednosti 1, 2 i 3, a zatim završava). Primer u nastavku definiše posmatrač kao objekat:

Kod 4.1: reaktivno-programiranje/rxjs/pretplata.1.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6     <title>RxJS Pretplata 1</title>
7   </head>
8
9   <body>
10    <script src="rxjs.umd.js"></script>
11    <script>
12      // Primer toka koji emituje tri broja
13      const tokPrimer$ = rxjs.from([1, 2, 3]);
14
15      // Primer posmatraca koji sabira vrednosti iz toka
16      const sumObserver = {
17        sum: 0,
18        // Funkcije (1) next, (2) error i (3) complete bice pozvane
19        // u slucaju da posmatrac dobije:
20        // (1) regularnu vrednost iz toka
21        // (2) gresku iz toka
22        // (3) nakon sto tok iscrpi sve vrednosti
23        next(value) {
24          console.log('Adding: ' + value);
25          this.sum = this.sum + value;
26        },
27        error() {
28          console.log('An error occured');
29        },
30        complete() {
31          console.log('Sum equals: ' + this.sum);
32        }
33      };
34
35      // Metod subscribe koristimo za pretplacivanje posmatraca na tok
36      tokPrimer$.subscribe(sumObserver);
37    </script>
38  </body>
39 </html>

```

Na sličan način možemo dobiti isti rezultat korišćenjem funkcija:

Kod 4.2: reaktivno-programiranje/rxjs/pretplata.2.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6     <title>RxJS Pretplata 2</title>
7   </head>
8
9   <body>
10    <script src="rxjs.umd.js"></script>
11    <script>
12      // Primer toka koji emituje tri broja
13      const tokPrimer$ = rxjs.from([1, 2, 3]);
14
15      // Posmatrac moze biti zadat trima funkcijama
16      let sum = 0;
17
18      tokPrimer$.subscribe(
19        function(value) {
20          console.log('Adding: ' + value);
21          sum = sum + value;
22        },
23        // Nemamo greske u toku, pa "ignorisemo" funkciju za obradu gresaka
24        undefined,
25        function() {
26          console.log('Sum equals: ' + sum);
27        }
28      );
29    </script>
30  </body>
31 </html>

```

Važno je znati da je moguće *ukinuti pretplatu* (engl. *unsubscribe*) na tok. Ukidanje pretplate podrazumeva da prethodno registrovani posmatrač više neće dobijati emitovane vrednosti iz toka. Informaciju o pretplati nekog posmatrača na tok možemo dobiti prilikom pretplaćivanja tog objekta na taj tok kroz objekat klase `Subscription`. Ovaj objekat ima jedan važan metod, `unsubscribe`, kojim se odgovarajući posmatrač uklanja iz liste posmatrača za odgovarajući tok. Naredni primer ilustruje tok koji na svaku sekundu emituje jednu vrednost koju će posmatrač ispisati u konzoli. Nakon 5 sekundi, pretplata se ukida i u konzoli prestaje ispisivanje daljih vrednosti.

Kod 4.3: reaktivno-programiranje/rxjs/ukidanje.pretplate.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6     <title>RxJS Ukidanje Pretplate</title>
7   </head>
8
9   <body>
10    <script src="rxjs.umd.js"></script>
11    <script>
12      const tokPrimer$ = rxjs.interval(1000);
13      const posmatrac = {
14        next(x) {
15          console.log('Dobijena vrednost:', x);
16        }

```

```

17     };
18
19     const pretplata = tokPrimer$.subscribe(posmatrac);
20     setTimeout(() => {
21         pretplata.unsubscribe();
22     }, 5000);
23 </script>
24 </body>
25 </html>

```

Ukidanje pretplate je važno zbog smanjenja resursa koje aplikacija održava zbog aktivnih pretplata na tokove. Ukoliko u nekom trenutku ne želimo više da osluškujemo tok, važno je da se pretplata na taj tok ukine.

4.3.2 Kreiranje tokova

Postoji više načina na koje možemo kreirati nove tokove. U ovom delu teksta ćemo razgovarati o RxJS funkcijama koje kreiraju nove tokove.

DOM događaji kao tokovi

Kreiranje novog toka od DOM događaja se može izvršiti funkcijom `fromEvent`. Njegov prvi argument je čvor DOM stabla, a drugi argument je naziv događaja od kojeg želimo da kreiramo tok. Na primer, ako želimo da se pretplatimo na klik dugmeta (element `<input type="button">`), to možemo uraditi na sledeći način:

Kod 4.4: reaktivno-programiranje/rxjs/dom.click.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6     <title>RxJS Dom Click</title>
7   </head>
8
9   <body>
10    <input type="button" id="dugme" value="Klikni me!" />
11
12    <script src="rxjs.umd.js"></script>
13    <script>
14      const dugme = document.getElementById('dugme');
15      const clickTok$ = rxjs.fromEvent(dugme, 'click');
16
17      clickTok$.subscribe(
18        event => {
19          document.body.appendChild(document.createTextNode('Klik!'));
20        },
21        error => {
22          document.body.appendChild(
23            document.createTextNode(`Desila se greska: ${error.message}`)
24          );
25        },
26        () => {
27          // DOM eventi nikada ne prestaju,
28          // pa ovaj tok nikada nece biti zavrsen
29          document.body.appendChild(document.createTextNode('Gotovo!'));
30        }
31      );
32    </script>
33  </body>

```

34 `</html>`

Slično, možemo se pretplatiti na događaj za korisnikov unos u `<input>` polju formulara na sledeći način:

Kod 4.5: reaktivno-programiranje/rxjs/dom.keyup.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6     <title>RxJS Dom Keyup</title>
7   </head>
8
9   <body>
10    <input type="text" id="unos" placeholder="Kucajte tekst ovde" />
11    <br />
12    <span id="rezultat"></span>
13
14    <script src="rxjs.umd.js"></script>
15    <script>
16      const unos = document.getElementById('unos');
17      const keyupTok$ = rxjs.fromEvent(unos, 'keyup');
18
19      keyupTok$.subscribe(
20        event => {
21          document.getElementById('rezultat').textContent = event.target.value;
22        }
23        // Ostale dve funkcije su opcione
24      );
25    </script>
26  </body>
27 </html>

```

Možemo se pretplatiti na bilo koji događaj bilo kog DOM elementa, pa čak i na ceo dokument:

Kod 4.6: reaktivno-programiranje/rxjs/dom.mousemove.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6     <title>RxJS Dom Mousemove</title>
7   </head>
8
9   <body>
10    <span id="mis">Pomerajte kursor misa po dokumentu</span>
11
12    <script src="rxjs.umd.js"></script>
13    <script>
14      const clickTok$ = rxjs.fromEvent(document, 'mousemove');
15
16      clickTok$.subscribe(event => {
17        document.getElementById(
18          'mis'
19        ).textContent = `Kursor se nalazi na poziciji (${event.clientX}, ${
20          event.clientY})`;
21      });

```

```

21     </script>
22   </body>
23 </html>

```

Vrednosti kao tokovi

U nastavku teksta ćemo podrazumevati postojanje modula `getSubscriber.js` koji definiše narednu funkciju za kreiranje posmatrača:

Kod 4.7: reaktivno-programiranje/rxjs/getSubscriber.js

```

1  const getSubscriber = (function() {
2    function Subscriber(id, transformFn) {
3      this.id = id;
4      this.transformFn = transformFn;
5      this.next = function(emittedValue) {
6        if (typeof this.transformFn === 'function') {
7          console.log(`${this.id}: ${this.transformFn(emittedValue)}`);
8        } else {
9          console.log(`${this.id}: ${emittedValue}`);
10       }
11     };
12     this.error = function(error) {
13       console.log(`${this.id}: An Error occured - ${error.message || error}`);
14     };
15     this.complete = function() {
16       console.log(`${this.id}: Completed!`);
17     };
18   }
19
20   return (id, transformFn) => {
21     return new Subscriber(id, transformFn);
22   };
23 })();

```

Ova funkciju smo implementirali tako da prihvata naziv posmatrača kako bi bilo lakše da identifikujemo više posmatrača u konzoli i, opciono, funkciju koja transformiše vrednost iz toka pre ispisivanja u konzolu.

Za kreiranje toka od niza vrednosti koristi se funkcija `from`. U ovakvom slučaju, tok se završava kada se iscrpe svi elementi u nizu:

Kod 4.8: reaktivno-programiranje/rxjs/from.1.html

```

1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <meta charset="UTF-8" />
5      <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6      <title>RxJS From 1</title>
7    </head>
8
9    <body>
10     <script src="rxjs.umd.js"></script>
11     <script src="getSubscriber.js"></script>
12     <script>
13       const nums = [1, 2, 3, 4, 5];
14       const nums$ = rxjs.from(nums);
15       nums$.subscribe(getSubscriber('nums'));
16     </script>
17   </body>
18 </html>

```

Korišćenje funkcije `from` za kreiranje toka nije korisno samo za kreiranje toka od niza. Ovu funkciju možemo iskoristiti i za kreiranje toka od drugih kolekcija, kao i od niski:

Kod 4.9: reaktivno-programiranje/rxjs/from.2.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6     <title>RxJS From 2</title>
7   </head>
8
9   <body>
10    <script src="rxjs.umd.js"></script>
11    <script src="getSubscriber.js"></script>
12    <script>
13      // Tok od Set objekta
14      const s = new Set(['Foo', 43, { name: 'Jeff' }]);
15      const s$ = rxjs.from(s);
16      s$.subscribe(getSubscriber('set'));
17
18      // Tok od Map objekta
19      const m = new Map([
20        [1, 2],
21        [3, 4],
22        [5, 6]
23      ]);
24      const m$ = rxjs.from(m);
25      m$.subscribe(getSubscriber('map'));
26
27      // Tok od niske
28      const str = 'Hello World';
29      const str$ = rxjs.from(str);
30      str$.subscribe(getSubscriber('str'));
31    </script>
32  </body>
33 </html>

```

4.3.3 Vrste tokova

Ne ponašaju se svi tokovi isto. Ova činjenica zauzima važno mesto pri radu sa tokovima, te ćemo o njoj detaljno diskutovati. Prema načinu emitovanja vrednosti, razlikujemo dve vrste tokova:

- *Topli tok* (engl. *hot observable*) emituje vrednosti bez obzira na posmatrača koji su pretplaćeni na njega. Posmatrač koji je pretplaćen na topli tok dobiće vrednosti emitovane tačno od trenutka kada se pretplati na taj tok. Svaki drugi posmatrač pretplaćen u tom trenutku dobija istu vrednost.
- *Hladni tok* (engl. *cold observable*) emituju celu sekvencu vrednosti od početka svakom pojedinačnom posmatraču. Za razliku od toplog toka, hladni tok emituje vrednosti samo kada postoji posmatrač koji je pretplaćen na njega.

DOM događaj kursora miša predstavlja primer toplog toka zato što će događaj biti emitovan bez obzira da li postoji posmatrač koji je pretplaćen na taj događaj. Dodatno, svaki novi pretplatnik dobija informaciju o samo onim događajima od trenutka pretplate na tok, a ne o svim događajima koji su se desili:

Kod 4.10: reaktivno-programiranje/rxjs/topli.tok.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6     <title>RxJS Topli Tok</title>
7   </head>
8
9   <body>
10    <script src="rxjs.umd.js"></script>
11    <script src="getSubscriber.js"></script>
12    <script>
13      const onClick$ = rxjs.fromEvent(document, 'click');
14      let preostaloKlikova = 3;
15
16      // Pretplati prvog posmatraca
17      onClick$.subscribe(
18        getSubscriber('Subscriber1', e => `(${e.clientX}, ${e.clientY})`)
19      );
20
21      const subscription = onClick$.subscribe({
22        next() {
23          --preostaloKlikova;
24
25          if (preostaloKlikova === 0) {
26            // Nakon tri klika, pretplati drugog posmatraca.
27            // S obzirom da je tok topli, ovaj posmatrac dobija samo nove
28              vrednosti
29            onClick$.subscribe(
30              getSubscriber('Subscriber2', e => `(${e.clientX}, ${e.clientY})`)
31            );
32            // Otplati ovog posmatraca
33            subscription.unsubscribe();
34            console.log('Drugi posmatrac je pretplacen!');
35          } else {
36            console.log(
37              `Preostalo je jos ${preostaloKlikova} klik(a) do pretplacivanja
38                drugog posmatraca...`
39            );
40          }
41        }
42      });
43    </script>
44  </body>
45 </html>

```

Sa druge strane, tok koji nastaje funkcijom `from` je hladni tok. Svaki novi posmatrač koji se pretplati dobija iznova sve vrednosti.

Kod 4.11: reaktivno-programiranje/rxjs/hladni.tok.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6     <title>RxJS Hladni Tok</title>
7   </head>
8
9   <body>

```

```

10 <script src="rxjs.umd.js"></script>
11 <script src="getSubscriber.js"></script>
12 <script>
13   const nums = [1, 2, 3, 4, 5];
14   const nums$ = rxjs.from(nums);
15
16   // Odmah se sve vrednosti procesiraju posmatracem Nums1
17   nums$.subscribe(getSubscriber('Nums1'));
18
19   setTimeout(() => {
20     // Nakon jedne sekunde,
21     // ponovo se sve vrednosti procesiraju posmatracem Nums2
22     nums$.subscribe(getSubscriber('Nums2'));
23   }, 1000);
24 </script>
25 </body>
26 </html>

```

Razumevanje da li je tok topli ili hladni može biti krucijalno u razrešavanju bagova koji se potencijalno neočekivani. Na primer, funkcija `interval` kreira tok koji emituje celobrojne vrednosti u pravilnim vremenskim intervalima. Ukoliko ne razumemo koncept hladnog i toplog toka, mogli bismo pretpostaviti da će nam ovaj tok isporučivati uvek tekuću vrednost iz intervala, bez obzira kada se pretplatimo nekim posmatračem na njega. Na primer:

Kod 4.12: reaktivno-programiranje/rxjs/interval.1.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6     <title>RxJS Interval 1</title>
7   </head>
8
9   <body>
10    <script src="rxjs.umd.js"></script>
11    <script src="getSubscriber.js"></script>
12    <script>
13      const source$ = rxjs.interval(1000);
14
15      source$.subscribe(getSubscriber('Observer 1'));
16      source$.subscribe(getSubscriber('Observer 2'));
17    </script>
18  </body>
19 </html>

```

Iz ispisa u konzoli može nam delovati da je naša pretpostavka tačna, s obzirom da oba posmatrača dobijaju iste vrednosti. Međutim, ukoliko bismo se drugim pretplatnikom pretplatili 3 sekunde nakon prvog pretplatnika, videli bismo da drugi pretplatnik dobija *već emitovane vrednosti* umesto da nastavi od poslednje emitovane vrednosti:

Kod 4.13: reaktivno-programiranje/rxjs/interval.2.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6     <title>RxJS Interval 2</title>

```

```

7   </head>
8
9   <body>
10    <script src="rxjs.umd.js"></script>
11    <script src="getSubscriber.js"></script>
12    <script>
13      const source$ = rxjs.interval(1000);
14
15      source$.subscribe(getSubscriber('Observer 1'));
16      setTimeout(() => {
17        source$.subscribe(getSubscriber('Observer 2'));
18      }, 3000);
19    </script>
20  </body>
21 </html>

```

Objašnjenje za ovo ponašanje leži u tome što funkcija `interval` proizvodi hladan, a ne topli tok. Ukoliko ne razumemo razliku između ove dve vrste tokova, velika je šansa da ćemo se pronaći u scenariju koji nismo očekivali.

Pretvaranje hladnog toka u topli

Možemo kreirati topli tok iz hladnog toka korišćenjem *operatora* `publish`. Pozivom ovog operatora dobija se tok koji služi kao proksi za originalni tok i za razliku od `Observable` tipa, ovaj tok je tipa `ConnectableObservable`. Nad ovim tokom nam je dostupna funkcija `connect` koju je potrebno pozvati da bismo započeli emitovanje vrednosti:

Kod 4.14: reaktivno-programiranje/rxjs/publish.html

```

1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <meta charset="UTF-8" />
5      <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6      <title>RxJS Publish</title>
7    </head>
8
9    <body>
10     <script src="rxjs.umd.js"></script>
11     <script src="getSubscriber.js"></script>
12     <script>
13       // Kreiramo tok koji emituje vrednost svake sekunde
14       const source$ = rxjs.interval(1000);
15       const publisher$ = source$.pipe(rxjs.operators.publish());
16
17       // Čak i ako se pretplatimo, nijedna vrednost se još uvek ne emituje
18       publisher$.subscribe(getSubscriber('Observer 1'));
19
20       // "Izdavac" (publisher) se povezuje i zapocinje emitovanje vrednosti
21       publisher$.connect();
22
23       setTimeout(() => {
24         // 5 sekundi kasnije, drugi pretplatnik se pretplacuje
25         // i dobija vrednosti počevši od tekuće emitovane vrednosti
26         // umesto od početka sekvence
27         publisher$.subscribe(getSubscriber('Observer 2'));
28       }, 5000);
29     </script>
30  </body>
31 </html>

```

4.3.4 Ulančavanje operatora

U prethodnom primeru vidimo nešto drugačiju sintaksu do sada: poziv operatora `publish` se ne izvršava direktno nad objektom toka, već posredstvom metoda `pipe`. O čemu je reč? Počevši od verzije 5.5, u biblioteci RxJS, ne možemo direktno pozvati operator `publish` nad tokom, već se koristi metod `pipe`. Za više informacija pogledati članak koji se nalazi na adresi <https://github.com/ReactiveX/rxjs/blob/master/doc/pipeable-operators.md>. U ovom tekstu se koristi termin *operator* (engl. *operator*) da označi one funkcije koje se koriste upravo na ovaj način. Ulančavanje više od jednog operatora se vrši jednostavnim nabravanjem, odvojenih zapetom, kao u narednom primeru:

Kod 4.15: reaktivno-programiranje/rxjs/pipe.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6     <title>RxJS Pipe</title>
7   </head>
8
9   <body>
10    <script src="rxjs.umd.js"></script>
11    <script src="getSubscriber.js"></script>
12    <script>
13      const source$ = rxjs.interval(100);
14
15      source$
16        .pipe(
17          rxjs.operators.take(50),
18          rxjs.operators.filter(x => x % 2 === 0),
19          rxjs.operators.map(x => x + x),
20          rxjs.operators.scan((acc, x) => acc + x, 0)
21        )
22        .subscribe(getSubscriber('Pipe example'));
23    </script>
24  </body>
25 </html>

```

U ovom primeru, operatori su `take`, `filter`, `map` i `scan`. O ovim, ali i o još nekim operatorima, biće reči u daljem tekstu. U slučaju da koristimo RxJS biblioteku u veb pregledaču (korišćenjem UMD verzije, kao što smo do sada to činili), operatori su dostupni kroz globalni objekat `rxjs.operators`. Sa druge strane, ukoliko se biblioteka koristi kao deo ES6 projekta, potrebno je uključiti odgovarajuća zaglavlja iz modula `'rxjs/operators'`:

```
1 import { take, filter, map, scan } from 'rxjs/operators';
```

Veoma je bitno razumeti da nakon primere jednog ili više operatora nad izvornim tokom, dobija se tok koji sada može da ima neke druge vrednosti. Upravo ulančavanje operatora služi za kreiranje novih tokova od postojećih. U daljem tekstu ćemo videti još neke primere operatora.

4.3.5 Još neke funkcije za kreiranje tokova

Kao što smo rekli, funkcija `interval` kreira tok koji emituje cele brojeve na određen broj milisekundi koji se prosleđuje kao argument funkciji. Ovaj tok se nikad ne završava, osim ukoliko ne ulančamo operator `take` koji emituje prvih n vrednosti u toku, gde je n argument koji se prosleđuje tom operatoru:

Kod 4.16: reaktivno-programiranje/rxjs/interval.take.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6     <title>RxJS Interval Take</title>
7   </head>
8
9   <body>
10    <script src="rxjs.umd.js"></script>
11    <script src="getSubscriber.js"></script>
12    <script>
13      const source$ = rxjs.interval(1000).pipe(rxjs.operators.take(3));
14
15      source$.subscribe(getSubscriber('Take'));
16    </script>
17  </body>
18 </html>

```

Funkcija `timer(dueTime, periodOrScheduler)` kreira tok kojim se emituju celobrojne vrednosti slično kao i kod funkcije `interval`, na svakih *periodOrScheduler* milisekundi, ali se za emitovanjem vrednosti započinje tek nakon *dueTime* milisekundi.

Kod 4.17: reaktivno-programiranje/rxjs/timer.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6     <title>RxJS Timer</title>
7   </head>
8
9   <body>
10    <script src="rxjs.umd.js"></script>
11    <script src="getSubscriber.js"></script>
12    <script>
13      rxjs
14        .timer(2000, 100)
15        .pipe(rxjs.operators.take(10))
16        .subscribe(getSubscriber('Timer'));
17    </script>
18  </body>
19 </html>

```

Funkcija `range(start, count)` kreira tok koji emituje celobrojne vrednosti u intervalu $[start, start + count)$ jedan za drugim:

Kod 4.18: reaktivno-programiranje/rxjs/range.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6     <title>RxJS Range</title>
7   </head>
8
9   <body>
10    <script src="rxjs.umd.js"></script>

```

```

11     <script src="getSubscriber.js"></script>
12     <script>
13         rxjs.range(5, 10).subscribe(getSubscriber('Range'));
14     </script>
15 </body>
16 </html>

```

Funkcija `of(...args)` kreira tok od proizvoljnih vrednosti koji se proslede kao argumenti ove funkcije:

Kod 4.19: reaktivno-programiranje/rxjs/of.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6     <title>RxJS Of</title>
7   </head>
8
9   <body>
10    <script src="rxjs.umd.js"></script>
11    <script src="getSubscriber.js"></script>
12    <script>
13        rxjs.of(45, ['Niz', 'niski'], {}).subscribe(getSubscriber('Of'));
14    </script>
15  </body>
16 </html>

```

Funkcija `defer(observableFactory)` kreira tok koji, pri svakom pretplaćivanju, poziva funkciju `observableFactory` da bi kreirala tok za svakog novog pretplatnika. Dakle, argument ove funkcije je druga funkcija koja konstruiše tok na koji će se pretplatiti novi pretplatnik:

Kod 4.20: reaktivno-programiranje/rxjs/defer.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6     <title>RxJS Defer</title>
7   </head>
8
9   <body>
10    <script src="rxjs.umd.js"></script>
11    <script src="getSubscriber.js"></script>
12    <script>
13        let count = 0;
14        const source$ = rxjs.defer(() => {
15            ++count;
16
17            return rxjs.of(count);
18        });
19
20        source$.subscribe(getSubscriber('Defer 1'));
21        source$.subscribe(getSubscriber('Defer 2'));
22        source$.subscribe(getSubscriber('Defer 3'));
23    </script>
24  </body>
25 </html>

```

Funkcija može da vrati i obećanje, koje će biti pretvoreno u tok. Funkcija `defer` čeka dok se neki pretplatnik ne pretplati na tok koji ona kreira, a zatim generiše novi tok za svakog pretplatnika, tako da iako pretplatnik misli da se pretplaćuje na isti tok (u primeru iznad izgleda kao da se pretplatioci `Defer 1`, `Defer 2` i `Defer 3` pretplaćuju na isti tok `source$`), zapravo svaki pretplatnik dobija svoj tok na koji se pretplaćuje (u primeru iznad se ti novi tokovi dobijaju pozivom funkcije `of` unutar argumenta funkcije `defer`).

Funkcija `merge` kreira tok koji konkurentno emituje vrednosti iz svih ulaznih tokova. Drugim rečima, sve emitovane vrednosti, iz svakog toka koji se prosleđuje funkciji, spajaju se u jedinstven tok u kojem se te vrednosti emituju nepromenjene.

Kod 4.21: reaktivno-programiranje/rxjs/merge.html

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6     <title>RxJS Merge</title>
7   </head>
8
9   <body>
10    <script src="rxjs.umd.js"></script>
11    <script src="getSubscriber.js"></script>
12    <script>
13      const click$ = rxjs.fromEvent(document, 'click');
14      const interval$ = rxjs.interval(1000);
15
16      rxjs.merge(click$, interval$).subscribe(getSubscriber('Merge'));
17    </script>
18  </body>
19 </html>
```

Funkcija `forkJoin` kreira jedinstven tok od niza drugih tokova tako što sačeka da sve vrednosti iz ulaznih tokova budu emitovane, a zatim emituje niz poslednjih emitovanih vrednosti.

Kod 4.22: reaktivno-programiranje/rxjs/forkJoin.html

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6     <title>RxJS ForkJoin</title>
7   </head>
8
9   <body>
10    <script src="rxjs.umd.js"></script>
11    <script src="getSubscriber.js"></script>
12    <script>
13      const observable = rxjs.forkJoin([
14        rxjs.of(1, 2, 3, 4),
15        Promise.resolve(8),
16        rxjs.timer(4000)
17      ]);
18      observable.subscribe(getSubscriber('ForkJoin'));
19    </script>
20  </body>
21 </html>
```

Funkcija `zip` kombinuje više tokova radi kreiranja novog toka čije vrednosti su izračunate na osnovu emitovanih vrednosti ulaznih tokova, za svaki od ulazni tok, redom. Razlika u odnosu na `forkJoin` je u tome što se ne čeka da se ulazni tokovi završe, već se nova vrednost emituje, po emitovanju narednih vrednosti iz svih ulaznih tokova. Posledica toga je da će broj emitovanih vrednosti u rezultujućem toku biti jednak broju emitovanih vrednosti u toku koji se "prvi završi"¹. Dodatno, moguće je proslediti funkciju kao poslednji argument koja prihvata vrednosti iz ulaznih tokova i izračunava vrednost koja će biti emitovana u rezultujućem toku.

Kod 4.23: reaktivno-programiranje/rxjs/zip.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6     <title>RxJS Zip</title>
7   </head>
8
9   <body>
10    <script src="rxjs.umd.js"></script>
11    <script src="getSubscriber.js"></script>
12    <script>
13      let age$ = rxjs.of(27, 25, 29, 30, 50, 17, 14);
14      let name$ = rxjs.of('Maddie', 'Kai', 'Luke');
15      let isDev$ = rxjs.of(true, true, false, true, false, false);
16
17      rxjs.zip(age$, name$, isDev$).subscribe(getSubscriber('Zip 1'));
18
19      function createValue(age, name, isDev) {
20        return `${name} (${age}yo) ${isDev ? 'is' : "isn't"} a developer`;
21      }
22
23      rxjs
24        .zip(age$, name$, isDev$, createValue)
25        .subscribe(getSubscriber('Zip 2'));
26    </script>
27  </body>
28 </html>

```

4.3.6 Transformisanje tokova

Ponekad nam je dovoljno da samo dobijamo emitovane vrednosti iz tokova i da takve vrednosti koristimo u aplikacijama. Međutim, često nam te vrednosti same po sebi nisu ključne, već ih koristimo za dobijanje drugih vrednosti ili, slično tome, često su nam samo neke od emitovanih vrednosti bitne, dok neke druge možemo da zanemarujemo. Biblioteka RxJS nudi veliki broj operatora kojima je moguće vršiti najrazličitije transformacije emitovanih vrednosti u tokovima.

Pre nego što prikazemo neke operatore za transformisanje tokova, potrebno je da uskladimo terminologiju koju ćemo koristiti u nastavku. Kažemo da je tok *izvorni* (engl. *source*) ako je u pitanju tok nad kojim se primenjuje odgovarajući operator korišćenjem metoda `pipe`. Kažemo da je tok *unutrašnji* (engl. *inner*) ako se koristi kao argument odgovarajućeg

¹Ovde se završavanje ne odnosi na vremenski period, već na broj emitovanog vrednosti pre završavanja. Na primer, ako primenimo operator `zip` na tri toka koji redom emituju 10, 11 i 12 vrednosti pre nego što se završe, rezultujući tok će emitovati tačno 10 vrednosti.

operatora ili u kontekstu nekog od argumenata tog operatora. Na primer, posmatranjem narednog fragmenta koda

```
1 rxjs.fromEvent(document, 'click').pipe(
2   rxjs.operators.flatMap(function (click) {
3     return rxjs.interval(500).pipe(rxjs.operators.take(10));
4   })
5 )
6 // ...
```

za operator `flatMap` vezujemo jedan izvorni tok koji se dobija pozivom funkcije `rxjs.fromEvent` i jedan unutrašnji tok koji se dobija pozivom funkcije `rxjs.interval`. U prikazanom fragmentu koda, unutrašnji tok ne predstavlja direktno argument operatora `flatMap`, ali učestvuje kao deo funkcije koja predstavlja argument tog operatora, tako da se i dalje smatra za njegov unutrašnji tok.

Pređimo sada na neke operatore za transformisanje tokova.

Jedan od svakako najkorisnijih operatora za transformisanje tokova jeste operator `map`. Ovaj operator transformiše svaku vrednost koja je emitovana iz toka, tako što primenjuje funkciju koja se prosleđuje kao argument nad svakom emitovanom vrednošću. Naredni primer ilustruje kreiranje toka operatorom `map` koji se dobija tako što se brojevi, koji su emitovani iz ravnomernog vremenskog intervala, kvadrirani.

Kod 4.24: reaktivno-programiranje/rxjs/map.html

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6     <title>RxJS Map</title>
7   </head>
8
9   <body>
10    <script src="rxjs.umd.js"></script>
11    <script src="getSubscriber.js"></script>
12    <script>
13      rxjs
14        .interval(1000)
15        .pipe(
16          rxjs.operators.take(5),
17          rxjs.operators.map(value => value * value)
18        )
19        .subscribe(getSubscriber('Map'));
20    </script>
21  </body>
22 </html>
```

Operator `reduce` primenjuje akumulatorsku funkciju nad vrednostima toka i vraća akumuliranu vrednost tek kada se sve vrednosti emituju. Akumulator se primenjuje nad opcionim neutralom. Naredni primer ilustruje kreiranje toka operatorom `reduce` koji će emitovati zbir brojeva iz intervala $[0, 10)$ sa neutralom 0, pri čemu se taj zbir emituje kada izvorni tok emituje sve brojeve.

Kod 4.25: reaktivno-programiranje/rxjs/reduce.html

```
1 <!DOCTYPE html>
2 <html lang="en">
```

```

3   <head>
4     <meta charset="UTF-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6     <title>RxJS Reduce</title>
7   </head>
8
9   <body>
10    <script src="rxjs.umd.js"></script>
11    <script src="getSubscriber.js"></script>
12    <script>
13      rxjs
14        .range(0, 10)
15        .pipe(rxjs.operators.reduce((acc, val) => acc + val, 0))
16        .subscribe(getSubscriber('Reduce'));
17    </script>
18  </body>
19 </html>

```

Operator `scan` funkcionira slično kao i operator `reduce` sa razlikom da emituje akumuliranu vrednost pri svakom emitovanju iz izvornog toka. Naredni primer ilustruje kreiranje toka operatorom `scan` koji će emitovati zbir brojeva iz intervala $[0, 10)$ sa neutralom 0, pri čemu se emituje ne samo konačni zbir, već i svi međuzbirovi koji nastaju emitovanjem vrednosti iz izvornog toka.

Kod 4.26: reaktivno-programiranje/rxjs/scan.html

```

1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <meta charset="UTF-8" />
5      <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6      <title>RxJS Scan</title>
7    </head>
8
9    <body>
10     <script src="rxjs.umd.js"></script>
11     <script src="getSubscriber.js"></script>
12     <script>
13       rxjs
14         .range(0, 10)
15         .pipe(rxjs.operators.scan((acc, val) => acc + val, 0))
16         .subscribe(getSubscriber('Scan'));
17     </script>
18   </body>
19 </html>

```

Operator `buffer` skladišti emitovane vrednosti iz izvornog toka i emituje niz skladištenih vrednosti svaki put kada unutrašnji tok emituje vrednost. Naredni primer ilustruje kreiranje toka operatorom `buffer` koji će emitovati niz emitovanih brojeva iz izvornog intervala koji je sakupio između svaka dva klika.

Kod 4.27: reaktivno-programiranje/rxjs/buffer.html

```

1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <meta charset="UTF-8" />
5      <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6      <title>RxJS Buffer</title>
7    </head>

```

```

8
9   <body>
10     <script src="rxjs.umd.js"></script>
11     <script src="getSubscriber.js"></script>
12     <script>
13       const onClick$ = rxjs.fromEvent(document, 'click');
14
15       rxjs
16         .interval(1000)
17         .pipe(rxjs.operators.buffer(onClick$))
18         .subscribe(getSubscriber('Buffer'));
19     </script>
20   </body>
21 </html>

```

Operator `throttleTime` se koristi za odbacivanje svih emitovanih vrednosti koje su emitovane u okviru nekog vremenskog intervala. Naredni primer ilustruje kreiranje toka operatorom `throttleTime` koji će emitovati događaje iz unosa u tekstualno polje, pri čemu će odbacivati sve one događaje koji su se dogodili u trajanju od jedne sekunde od prethodno emitovanog događaja.

Kod 4.28: reaktivno-programiranje/rxjs/throttleTime.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6     <title>RxJS ThrottleTime</title>
7   </head>
8
9   <body>
10     <input type="text" id="unos" />
11     <br />
12     <span id="rezultat"></span>
13
14     <script src="rxjs.umd.js"></script>
15     <script src="getSubscriber.js"></script>
16     <script>
17       const keydown$ = rxjs.fromEvent(
18         document.getElementById('unos'),
19         'keydown'
20       );
21
22       keydown$
23         .pipe(rxjs.operators.throttleTime(1000))
24         .subscribe(function(event) {
25           document.getElementById('rezultat').textContent = event.target.value;
26         });
27     </script>
28   </body>
29 </html>

```

4.3.7 Filterovanje tokova

Operator `filter` prihvata funkciju kojom testira svaku vrednost iz toka. Samo ukoliko vrednost ispunjava uslov, ta vrednost će biti *propuštena* kroz filter. Drugim rečima, ovaj operator kreira novi tok koji emituje samo one vrednosti iz ulaznog toka za koje funkcija kojom se testira vrati `true`:

Kod 4.29: reaktivno-programiranje/rxjs/filter.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6     <title>RxJS Filter</title>
7   </head>
8
9   <body>
10    <script src="rxjs.umd.js"></script>
11    <script src="getSubscriber.js"></script>
12    <script>
13      rxjs
14        .range(0, 10)
15        .pipe(rxjs.operators.filter(value => value % 2 === 0))
16        .subscribe(getSubscriber('Filter'));
17    </script>
18  </body>
19 </html>

```

Operator `distinct` će porediti narednu vrednost iz toka u odnosu na prethodno emitovane vrednosti. Ako naredna vrednost predstavlja "kopiju" neke od prethodno emitovanih vrednosti, ta vrednost neće biti emitovana. Podrazumevano se vrednosti porede po jednakosti, ali je moguće proslediti funkciju koja transformiše emitovanu vrednost u ključ po kojem će vršiti poređenje:

Kod 4.30: reaktivno-programiranje/rxjs/distinct.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6     <title>RxJS Distinct</title>
7   </head>
8
9   <body>
10    <script src="rxjs.umd.js"></script>
11    <script src="getSubscriber.js"></script>
12    <script>
13      rxjs
14        .of(1, 1, 2, 2, 2, 1, 2, 3, 4, 3, 2, 1)
15        .pipe(rxjs.operators.distinct())
16        .subscribe(getSubscriber('Distinct 1'));
17
18      rxjs
19        .of(1, 1, 2, 2, 2, 1, 2, 3, 4, 3, 2, 1)
20        .pipe(
21          rxjs.operators.distinct(val => {
22            return val === 1;
23          })
24        )
25        .subscribe(getSubscriber('Distinct 2'));
26    </script>
27  </body>
28 </html>

```

Operatori `first` i `last` emituju samo prvu, odnosno, poslednju vrednost iz toka, redom. Operator `elementAt` emituje samo vrednost čiji indeks odgovara indeksu koji se prosleđuje

operatoru:

Kod 4.31: reaktivno-programiranje/rxjs/first.last.elementAt.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6     <title>RxJS First Last ElementAt</title>
7   </head>
8
9   <body>
10    <script src="rxjs.umd.js"></script>
11    <script src="getSubscriber.js"></script>
12    <script>
13      rxjs
14        .of(1, 2, 3, 4, 5)
15        .pipe(rxjs.operators.first())
16        .subscribe(getSubscriber('First'));
17
18      rxjs
19        .of(1, 2, 3, 4, 5)
20        .pipe(rxjs.operators.last())
21        .subscribe(getSubscriber('Last'));
22
23      rxjs
24        .of(1, 2, 3, 4, 5)
25        .pipe(rxjs.operators.elementAt(2))
26        .subscribe(getSubscriber('ElementAt'));
27    </script>
28  </body>
29 </html>

```

Operator `debounce` odlaže emitovanje vrednosti iz izvornog toka sve dok prva vrednost iz unutrašnjeg toka ne bude emitovana. U slučajevima kada se iz izvornog toka emituje više vrednosti, a pritom unutrašnji tok nije emitovao vrednost, sve te emitovane vrednosti osim poslednje se ignorišu.

U narednom primeru, klikom na dokument u veb pregledaču se okida DOM događaj koji se emituje u izvornom toku. Međutim, kako je na taj izvorni tok primenjen operator `debounce`, to se taj događaj emituje tek onda kada unutrašnji tok ispali svoj prvi događaj, odnosno, kada tok `rxjs.interval` emituje prvu vrednost nakon jedne sekunde. U slučaju da kliknemo na dokument, pa od tog trenutka nastavljamo kliktanje, pri čemu se svako naredno kliktanje desi u intervalu manjem od jedne sekunde, onda će svi ispaljeni događaji biti ignorisani, osim poslednjeg (tj. osim onog klik-događaja koji nije praćen nekim drugim klik-događajem u intervalu manjem od jedne sekunde).

Kod 4.32: reaktivno-programiranje/rxjs/debounce.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6     <title>RxJS Debounce</title>
7   </head>
8
9   <body>
10    <script src="rxjs.umd.js"></script>

```

```

11     <script src="getSubscriber.js"></script>
12     <script>
13         const click$ = rxjs.fromEvent(document, 'click');
14         const result = click$.pipe(
15             rxjs.operators.debounce(() => rxjs.interval(1000))
16         );
17         result.subscribe(
18             getSubscriber('Debounce', ev => `(${ev.clientX},${ev.clientY})`)
19         );
20     </script>
21 </body>
22 </html>

```

4.3.8 Kombinovanje tokova

Operator `mergeAll` može biti koristan u slučaju da je potrebno da se pretplatimo na tok višeg reda. *Tok višeg reda* (engl. *higher-order observable*) predstavlja tok koji emituje druge tokove. Svaki put kada se emituje naredni tok iz izvornog toka operatora `mergeAll`, vrši se pretplaćivanje na taj tok i sve vrednosti se emituju kroz rezultujući tok. Naredni primer ilustruje situaciju kada se svakim klikom na dokument kreira novi intervalni tok (dakle, tok `click$` je primer toka višeg reda). Ulančavanjem operatora `mergeAll`, sve vrednosti koje su emitovane iz svih intervalnih tokova biće emitovane u toku `firstOrder`.

Kod 4.33: reaktivno-programiranje/rxjs/mergeAll.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6     <title>RxJS MergeAll</title>
7   </head>
8
9   <body>
10    <script src="rxjs.umd.js"></script>
11    <script src="getSubscriber.js"></script>
12    <script>
13        const click$ = rxjs.fromEvent(document, 'click');
14        const higherOrder = click$.pipe(
15            rxjs.operators.map(ev => rxjs.interval(1000))
16        );
17        const firstOrder = higherOrder.pipe(rxjs.operators.mergeAll());
18        firstOrder.subscribe(getSubscriber('MergeAll'));
19    </script>
20  </body>
21 </html>

```

Operator `switchMap` kreira tok koji emituje vrednosti koje se dobijaju primenom navedene funkcije na svaku emitovanu vrednost iz izvornog toka, pri čemu ta funkcija vraća unutrašnji tok. Prilikom svakog pretplaćivanja, izlazni tok započinje emitovanje onih vrednosti koje su emitovane unutrašnjim tokom. Kada se novi unutrašnji tok emituje, operator `switchMap` prestaje sa emitovanjem vrednosti iz prethodnog unutrašnjeg toka i započinje emitovanje vrednosti iz novog toka. Naredni primer ilustruje korišćenje ovog operatora kojim se svakim klikom na dokumentu započinje emitovanje novog intervala.

Kod 4.34: reaktivno-programiranje/rxjs/switchMap.html

```

1 <!DOCTYPE html>
2 <html lang="en">

```

```

3   <head>
4     <meta charset="UTF-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6     <title>RxJS Switchmap</title>
7   </head>
8
9   <body>
10    <script src="rxjs.umd.js"></script>
11    <script src="getSubscriber.js"></script>
12    <script>
13      const clicks$ = rxjs.fromEvent(document, 'click');
14
15      clicks$
16        .pipe(
17          rxjs.operators.switchMap(function(click) {
18            return rxjs.interval(500).pipe(rxjs.operators.take(10));
19          })
20        )
21      .subscribe(getSubscriber('SwitchMap'));
22    </script>
23  </body>
24 </html>

```

4.3.9 Obrada grešaka

Operator `catchError` služi za obradu grešaka koje nastaju u tokovima. Ovaj operator hvata grešku iz toka i, na osnovu definicije funkcije koja mu se prosledi kao argument, ili vraća novi tok ili izbacuje novu grešku. Naredni primer ilustruje obe situacije, redom.

Kod 4.35: reaktivno-programiranje/rxjs/catchError.html

```

1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <meta charset="UTF-8" />
5      <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6      <title>RxJS CatchError</title>
7    </head>
8
9    <body>
10     <script src="rxjs.umd.js"></script>
11     <script src="getSubscriber.js"></script>
12     <script>
13       rxjs
14         .of(1, 2, 3, 4, 5)
15         .pipe(
16           rxjs.operators.map(n => {
17             if (n === 4) {
18               throw 'four!';
19             }
20             return n;
21           }),
22           rxjs.operators.catchError(err => rxjs.of('I', 'II', 'III', 'IV', 'V'))
23         )
24       .subscribe(getSubscriber('CatchError 1'));
25
26     rxjs
27       .of(1, 2, 3, 4, 5)
28       .pipe(
29         rxjs.operators.map(n => {

```

```

30         if (n === 4) {
31             throw 'four!';
32         }
33         return n;
34     }},
35     rxjs.operators.catchError(err => {
36         throw 'An error occured in source. Details: ' + err;
37     })
38 )
39 .subscribe(getSubscriber('CatchError 2'));
40 </script>
41 </body>
42 </html>

```

Funkcija koja se prosleđuje operatoru `catchError` može prihvatati dva argumenta: prvi je greška koja je ispaljena, a drugi je referenca na sam tok iz kojeg je greška ispaljena.

Drugi argument se može koristiti za, na primer, "restartovanje" toka tako što se jednostavno vrati taj tok kao povratna vrednost funkcije. Međutim, za te svrhe se češće koristi operator `retry`. Ovaj operator kreira izlazni tok koji emituje iste vrednosti kao i izvorni tok, sa izuzetkom greške. U slučaju da izvorni tok pozove funkciju `error` nad pretplatiocem, ovaj operator će pokušati da se ponovo pretplati najviše `count` puta (koji se prosleđuje kao parametar operatora) umesto da propagira poziv funkcije `error` nad pretplatiocem. Podrazumevana vrednost parametra `count` je `-1`, što indikuje da će operator `retry` pokušavati sa ponovnim pretplaćivanjem sve dok ne dođe do uspeha.

Kod 4.36: reaktivno-programiranje/rxjs/retry.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6     <title>RxJS Retry</title>
7   </head>
8
9   <body>
10    <script src="rxjs.umd.js"></script>
11    <script src="getSubscriber.js"></script>
12    <script>
13      const source$ = rxjs.interval(1000);
14      source$
15        .pipe(
16          rxjs.operators.mergeMap(val => {
17            if (val > 5) {
18              throw 'Error!';
19            }
20            return rxjs.of(val);
21          }),
22          rxjs.operators.retry(2)
23        )
24      .subscribe(getSubscriber('Retry'));
25    </script>
26  </body>
27 </html>

```


Literatura za ovu oblast

- [doca] MDN web docs. *Array - JavaScript*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array.
- [Hus] Jafar Husain. *Asynchronous Programming: The End of The Loop* from @jhusain on @eggheadio. URL: <https://egghead.io/courses/asynchronous-programming-the-end-of-the-loop>.
- [Man15] S. Mansilla. *Reactive Programming with RxJS: Untangle Your Asynchronous JavaScript Code*. Pragmatic programmers. Pragmatic Bookshelf, 2015. ISBN: 9781680501292. URL: <https://books.google.rs/books?id=tc33sgEACAAJ>.
- [RxJ] RxJS. *RxJS - Reactive Extensions Library for JavaScript*. URL: <https://rxjs-dev.firebaseapp.com/>.
- [Sta] André Staltz. *Introduction to Reactive Programming* from @andrestaltz on @eggheadio. URL: <https://egghead.io/courses/introduction-to-reactive-programming>.
- [Tra] Brad Traversy. *Learn Reactive Programming and ReactiveX from Scratch*. URL: <https://www.eduonix.com/courses/Web-Development/learn-reactivex-from-ground-up>.



Programiranje serverskih aplikacija

5	Razvojno okruženje Node.js	205
5.1	Sistem zasnovan na događajima	
5.2	Kreiranje serverskih aplikacija	
5.3	Razvojno okruženje Express.js	
	Literatura za ovu oblast	
6	Baza podataka MongoDB	237
7	Mongoose ORM	239

5. Razvojno okruženje Node.js

Node.js predstavlja okruženje za izvršavanje JavaScript koda koje se ne oslanja na izvršavanje u veb pregledaču, kao što je to slučaj kod korisničkih aplikacija. Umesto toga, JavaScript kod se izvršava kroz mašinu V8, razvijenu od strane Google-a. V8 je napisan u programskom jeziku C++ i visoko je performantan što ga čini sjajnim izborom za osnovu Node.js sistema. Okruženje Node.js funkcioniše na principu događaja, čime se maksimalno iskoršćava konkurentnost zahteva, pogotovo što gotovo nijedna funkcija direktno ne vrši ulazno-izlazne operacije, te proces nikad nije blokiran. Upravo je to razlog zašto je moguće pisati visoko skalabilne serverske aplikacije u Node.js sistemu.

5.1 Sistem zasnovan na događajima

U ovoj sekciji ćemo se upoznati sa mnogobrojnim osobinama okruženja Node.js. Započecemo tekst diskusijom o blokirajućim i neblokirajućim operacijama, a zatim ćemo preći na "konkurentnost" u Node.js okruženju kroz petlju događaja.

5.1.1 Blokirajuće i neblokirajuće operacije

Blocking is when the execution of additional JavaScript in the Node.js process must wait until a non-JavaScript operation completes. This happens because the event loop is unable to continue running JavaScript while a blocking operation is occurring.

In Node.js, JavaScript that exhibits poor performance due to being CPU intensive rather than waiting on a non-JavaScript operation, such as I/O, isn't typically referred to as blocking. Synchronous methods in the Node.js standard library that use libuv are the most commonly used blocking operations. Native modules may also have blocking methods.

All of the I/O methods in the Node.js standard library provide asynchronous versions, which are non-blocking, and accept callback functions. Some methods also have blocking counterparts, which have names that end with **Sync**.

Blocking methods execute synchronously and non-blocking methods execute asynchronously.

Using the File System module as an example, this is a synchronous file read:

```
1 const fs = require('fs');
2 const data = fs.readFileSync('/file.md');
3 // blocks here until file is read
```

And here is an equivalent asynchronous example:

```
1 const fs = require('fs');
2 fs.readFile('/file.md', (err, data) => {
3   if (err) {
4     throw err;
5   }
6 });
```

The first example appears simpler than the second but has the disadvantage of the second line blocking the execution of any additional JavaScript until the entire file is read. Note that in the synchronous version if an error is thrown it will need to be caught or the process will crash. In the asynchronous version, it is up to the author to decide whether an error should throw as shown.

Let's expand our example a little bit:

```
1 const fs = require('fs');
2 const data = fs.readFileSync('/file.md');
3 // blocks here until file is read
4
5 console.log(data);
6 moreWork(); // will run after console.log
```

And here is a similar, but not equivalent asynchronous example:

```
1 const fs = require('fs');
2 fs.readFile('/file.md', (err, data) => {
3   if (err) {
4     throw err;
5   }
6   console.log(data);
7 });
8
9 moreWork(); // will run before console.log
```

In the first example above, `console.log` will be called before `moreWork()`. In the second example `fs.readFile()` is non-blocking so JavaScript execution can continue and `moreWork()` will be called first. The ability to run `moreWork()` without waiting for the file read to complete is a key design choice that allows for higher throughput.

Konkurentno izvršavanje

JavaScript execution in Node.js is single threaded, so concurrency refers to the event loop's capacity to execute JavaScript callback functions after completing other work. Any code that is expected to run in a concurrent manner must allow the event loop to continue running as non-JavaScript operations, like I/O, are occurring.

As an example, let's consider a case where each request to a web server takes 50ms to complete and 45ms of that 50ms is database I/O that can be done asynchronously. Choosing non-blocking asynchronous operations frees up that 45ms per request to handle other

requests. This is a significant difference in capacity just by choosing to use non-blocking methods instead of blocking methods.

The event loop is different than models in many other languages where additional threads may be created to handle concurrent work.

Problemi sa mešanjem blokirajućeg i neblokirajućeg koda

There are some patterns that should be avoided when dealing with I/O. Let's look at an example:

```
1 const fs = require('fs');
2 fs.readFile('/file.md', (err, data) => {
3   if (err) {
4     throw err;
5   }
6   console.log(data);
7 });
8 fs.unlinkSync('/file.md');
```

In the above example, `fs.unlinkSync()` is likely to be run before `fs.readFile()`, which would delete `file.md` before it is actually read. A better way to write this, which is completely non-blocking and guaranteed to execute in the correct order is:

```
1 const fs = require('fs');
2 fs.readFile('/file.md', (readFileErr, data) => {
3   if (readFileErr) {
4     throw readFileErr;
5   }
6
7   console.log(data);
8   fs.unlink('/file.md', (unlinkErr) => {
9     if (unlinkErr) {
10      throw unlinkErr;
11    }
12  });
13 });
```

The above places a non-blocking call to `fs.unlink()` within the callback of `fs.readFile()` which guarantees the correct order of operations.

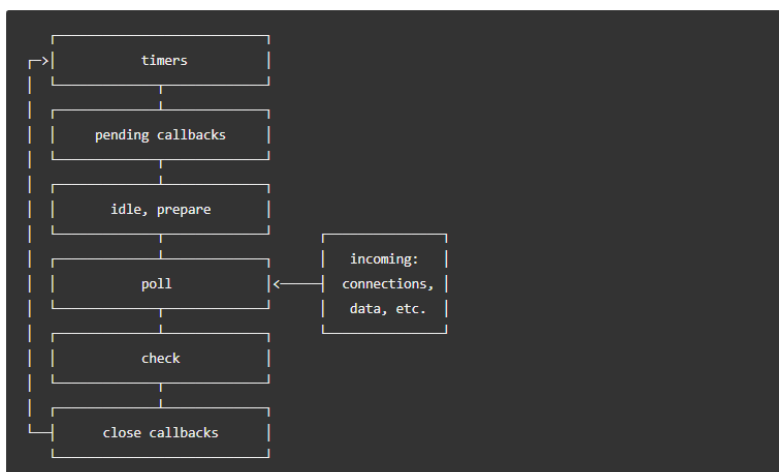
5.1.2 Petlja događaja

The event loop is what allows Node.js to perform non-blocking I/O operations — despite the fact that JavaScript is single-threaded — by offloading operations to the system kernel whenever possible.

Since most modern kernels are multi-threaded, they can handle multiple operations executing in the background. When one of these operations completes, the kernel tells Node.js so that the appropriate callback may be added to the poll queue to eventually be executed. We'll explain this in further detail later in this topic.

When Node.js starts, it initializes the event loop, processes the provided input script (or drops into the REPL, which is not covered in this document) which may make async API calls, schedule timers, or call `process.nextTick()`, then begins processing the event loop.

The following diagram shows a simplified overview of the event loop's order of operations.



Slika 5.1: Prikaz petlje događaja. Svaka "kutija" predstavlja jednu fazu ciklusa u petlji.

Each phase has a FIFO queue of callbacks to execute. While each phase is special in its own way, generally, when the event loop enters a given phase, it will perform any operations specific to that phase, then execute callbacks in that phase's queue until the queue has been exhausted or the maximum number of callbacks has executed. When the queue has been exhausted or the callback limit is reached, the event loop will move to the next phase, and so on.

Since any of these operations may schedule more operations and new events processed in the poll phase are queued by the kernel, poll events can be queued while polling events are being processed. As a result, long running callbacks can allow the poll phase to run much longer than a timer's threshold.

Pregled faza

- **timers:** this phase executes callbacks scheduled by `setTimeout()` and `setInterval()`.
- **pending callbacks:** executes I/O callbacks deferred to the next loop iteration.
- **idle, prepare:** only used internally.
- **poll:** retrieve new I/O events; execute I/O related callbacks (almost all with the exception of close callbacks, the ones scheduled by timers, and `setImmediate()`); node will block here when appropriate.
- **check:** `setImmediate()` callbacks are invoked here.
- **close callbacks:** some close callbacks, e.g. `socket.on('close', ...)`.

Between each run of the event loop, Node.js checks if it is waiting for any asynchronous I/O or timers and shuts down cleanly if there are not any.

5.2 Kreiranje serverskih aplikacija

Osnovni paket koji ćemo koristiti za kreiranje serverskih aplikacija je paket `http`¹. On nam nudi mnogobrojne metode za rad, a mi ćemo koristiti metod `createServer` kojim se kreira nova instanca veb servera. Dakle, da bismo učitali paket `http` i kreirali instancu veb servera, možemo uraditi:

```

1 const http = require('http');
2 const server = http.createServer();

```

¹Dokumentacija

Ono što je potrebno uraditi jeste registrovati funkcije kojima će server odreagovati na određene događaje, kao što je na primer, događaj koji se okine kada pristigne novi HTTP zahtev. Da bismo postavili osluškivač, možemo koristiti metode `on(eventName, listener)`² i `once(eventName, listener)`³. Razlika između ovih metoda je u tome što prvi metod će izvršiti po-potrebi-pozivnu funkciju `listener` svaki put kada je događaj predstavljen niskom `eventName` okinut, a drugi metod će samo jednom. Tako, na primer, za osluškivanje novih zahteva možemo koristiti događaj `'request'`⁴ koji će proslediti po-potrebi-pozivnoj funkciji dva objekta: prvi predstavlja HTTP zahtev od klijenta, a drugi predstavlja HTTP odgovor od servera.

```
1 server.on('request', function(req, res)
2 {
3     // Odgovaramo na zahtev od korisnika ovde...
4 }
```

Da bismo zapravo rekli serveru da treba da se pokrene, moramo pozvati metod `listen([port[, host[, backlog]]][, callback])`⁵, kojoj je dovoljno proslediti broj porta na koji se pokreće serverska aplikacija.

```
1 const port = 3000;
2 server.listen(port);
```

Ukoliko želimo da izvršimo neku akciju kada je server spreman, možemo registrovati po-potrebi-pozivnu funkciju nad događajem `'listening'`⁶:

```
1 server.once('listening', function()
2 {
3     console.log(`Go to http://localhost:${port}`);
4 });
```

Vratimo se na događaj `'request'` i objekte koje on prosleđuje, a sa kojima mi možemo da radimo: HTTP zahtev⁷ i HTTP odgovor⁸. Nad ovim objektima su definisana razna svojstva i metodi koje možemo koristiti. Na primer, nad HTTP zahtevom imamo definisano:

- Svojstvo `headers` sadrži objekat čija su svojstva zaglavlja HTTP zahteva
- Svojstvo `method` sadrži nisku sa HTTP metodom
- Svojstvo `url` sadrži nisku sa url-om u HTTP zahtevu

Nad HTTP odgovorom imamo definisano:

- Metod `setHeader(name, value)` postavlja vrednost `value` za jedno zaglavlje čiji je naziv `name`. Ako zaglavlje već ima definisanu vrednost, biće upotrebljena nova vrednost umesto nje. Potrebno ga je pozvati pre metoda `writeHead`.
- Metod `writeHead(statusCode[, statusMessage][, headers])` šalje HTTP zaglavlja. Možemo mu proslediti statusnu poruku i objekat čija su svojstva zaglavlja odgovora. Ako prosledimo zaglavlja koja su ranije postavljena pomoću metoda `setHeader`, onda će vrednosti prosledene ovom metodu imati prioritet. Metod mora biti pozvan tačno jednom i to pre poziva metoda `write` i `end`.

²Dokumentacija

³Dokumentacija

⁴Dokumentacija

⁵Dokumentacija

⁶Dokumentacija

⁷Dokumentacija

⁸Dokumentacija

- Metod `write(chunk[, encoding][, callback])` postavlja telo odgovora. Metod se može pozvati više puta svaki put dodajući nastavak tela.
- Metod `end([data][, encoding][, callback])` kojim se završava obrada HTTP zahteva i šalje se HTTP odgovor. Metod signalizira da su sva zaglavlja i telo poslata, te se mora pozvati nad svakim odgovorom.

Da bismo pokrenuli našu serversku aplikaciju, potrebno je da se u terminalu pozicioniramo u koreni direktorijum projekta i ukucamo komandu

```
1 $ nodemon server.js
```

Nodemon (<https://nodemon.io/>) predstavlja alat kojom se mogu lakše restartovati pokrenuti serveri prilikom izmena u izvornom kodu. Ovaj alat prati da li je došlo do izmene nekog izvornog fajla i ako jeste, onda će automatski ugasiti server, primeniti izmene i ponovo ga pokrenuti, što je korisno ako testiramo rad servera i pravimo izmene u isto vreme. Da bismo ga instalirali, potrebno je da izvršimo komandu

```
1 $ npm install -g nodemon
```

Primer 5.1 Napisati serversku aplikaciju koja se pokreće na portu 3000 i koja treba da opslužuje naredne HTTP zahteve:

- Bilo koji zahtev nad `http://localhost:3000/*`
 - Potrebno je da se vrati tekst `'Hello world!'` u telu odgovora. Tip odgovora je `text/plain`, a statusni kod je 200.

Rešenje.

Kod 5.1: nodejs/001/server.js

```
1 const http = require("http");
2
3 const server = http.createServer();
4 server.on("request", function(req, res)
5 {
6     res.writeHead(200, {
7         "content-type": "text/plain"
8     });
9
10    res.write("Hello World!");
11    res.end();
12 });
13
14 const port = 3000;
15
16 server.listen(port);
17 server.once("listening", function()
18 {
19     console.log(`Hello World server listening on port ${port}`);
20     console.log(`Go to http://localhost:${port}`);
21 });
```

Prethodni primer ilustruje jednostavnost kreiranja i podizanja serverskih aplikacija. Naravno, što su zahtevi naših zadataka veći, to je i kod kojim se ti zadaci rešavaju veći. Međutim, ono što je pozitivna karakteristika Node.js razvojnog okruženja jeste da se aplikacije jednostavno skaliraju, kao i da je za korišćenje novih paketa dovoljno da razumemo API koji ti paketi nude. Pogledajmo naredni primer.

Primer 5.2 Napisati serversku aplikaciju koja se pokreće na portu 3000 i koja treba da opslužuje naredne HTTP zahteve:

- GET zahtev nad `http://localhost:3000/*`
 - Izračunati skalarni proizvod vektora $x = (x_1, x_2, x_3)$ i $y = (y_1, y_2, y_3)$ koji se prosleđuju serveru preko svojih koordinata.
- OPTIONS zahtev nad `http://localhost:3000/*`
 - Server odgovara da su dostupni metodi OPTIONS i GET.

Ispravno implementirati CORS mehanizam zaštite.

Rešenje. Kao što vidimo, u odnosu na prethodni primer, naši zahtevi su se značajno usložnili. Međutim, nakon ovog primera, razumećemo kako da ispravno povežemo našu klijentsku aplikaciju koja se pokreće iz veb pregledača sa serverskom aplikacijom koja se izvršava kao Node.js aplikacija. Za početak, krenimo od implementiranja serverske strane aplikacije, čiji će kostur biti nešto izmenjeno rešenje iz prethodnog primera:

```
1 const http = require('http');
2
3 const server = http.createServer();
4 server.on('request', function(req, res)
5 {
6     let teloOdgovora = {};
7     let statusniKod = 200;
8
9     // Obrada zahteva ide ovde...
10
11     res.writeHead(statusniKod, {
12         'Access-Control-Allow-Origin': '*',
13         'Access-Control-Allow-Headers': 'Content-Type'
14     });
15     res.write(JSON.stringify(teloOdgovora));
16     res.end();
17 });
18
19 const port = 3000;
20 server.listen(port);
21 server.once('listening', function()
22 {
23     console.log(`http://localhost:${port}`);
24 });
```

Ono što vidimo da je drugačije u odnosu na prethodni primer jeste što smo izdvojili statusni kod i telo odgovora u posebne promenljive, koje ćemo izmeniti u zavisnosti od toga na koji zahtev treba da odgovorimo, kao i na šta je to što ćemo vratiti kao odgovor.

Dodatno, vidimo da u startu postavljamo zaglavlja `Access-Control-Allow-Origin`, kao i `Access-Control-Allow-Headers`. Ovo je u vezi za nečim što zovemo *deljenje resursa među različitim izvorima* (engl. *Cross-Origin Resource Sharing*, skr. *CORS*). U pitanju je mehanizam koji koristi dodatna HTTP zaglavlja u odgovoru da naznači veb pregledaču da dopusti da veb aplikacija koja se izvršava sa jednog izvora (domena) ima pravo da koristi resurse sa servera koji se nalazi na drugom izvoru (domenu). Veb aplikacija kreira *HTTP zahtev među različitim izvorima* (engl. *cross-origin HTTP request*) kada zahteva resurs koji ima drugačiji izvor u odnosu na njegov izvor. Pod "izvorom" podrazumevamo domen, protokol i port. Da bismo ispravno implementirali obradu CORS zahteva, potrebno je da

naša serverska aplikacija naznači sa kojih izvora CORS zahtevi mogu biti prihvaćeni, što se kontroliše zaglavljem `Access-Control-Allow-Origin`. Ukoliko je njegova vrednost `*`, onda su svi izvori dostupni. Inače, mogu se samo neki izvori smatrati kao validni, na primer, `http://foo.example` i tada bismo vratili zaglavlje:

```
1 Access-Control-Allow-Origin: http://foo.example
```

Slično, možemo koristiti zaglavlje `Access-Control-Allow-Headers` da bismo specifikovali koja zaglavlja klijent može da pošalje ka serveru (pored nekih podrazumevanih koje veb pregledač automatski dodaje, kao što su `Accept` i `User-Agent`). U primeru smo postavili da je dozvoljeno zaglavlje još i `Content-Type`. Ovime smo uspešno implementirali CORS mehanizam, te možemo preći na obradu zahteva.

S obzirom da je potrebno da obradimo samo zahteve `GET` i `OPTIONS`, potrebno je da izdvojimo informaciju o metodi iz HTTP zahteva, a zatim da obezbedimo da se samo ti metodi obrađuju:

```
1 let method = req.method;
2
3 if (method === "GET")
4 {
5     // Obrada GET zahteva ide ovde...
6 }
7 else if (method === "OPTIONS")
8 {
9     // Obrada OPTIONS zahteva ide ovde...
10 }
11 else
12 {
13     statusniKod = 405;
14     res.setHeader('Access-Control-Allow-Methods', 'OPTIONS, GET');
15 }
```

Vidimo da se u slučaju slanja nepodržanog metoda u zahtevu vraća statusni kod 405, kao i da se postavlja zaglavlje `Access-Control-Allow-Methods`, čija je vrednost spisak metoda koje server podržava.

Da bi se sprečilo da klijent šalje nepodržane metode, često klijentske aplikacije prvo pošalju `OPTIONS` metod ka serveru pre nego što pošalju sam zahtev. HTTP metod `OPTIONS` služi da se od servera dobije informacija o tome koji su HTTP metodi implementirani na serveru. Potrebno je da server pošalje ovu informaciju kroz zaglavlje `Access-Control-Allow-Methods`:

```
1 else if (method === "OPTIONS")
2 {
3     statusniKod = 200;
4     res.setHeader('Access-Control-Allow-Methods', 'OPTIONS, GET');
5 }
```

Pređimo sada na glavni deo naše serverske aplikacije: implementiranje obrade `GET` zahteva. Kada se podaci od klijenta šalju putem zahteva `GET`, pravilo je da ti podaci budu zapisani kao deo URL-a ka kojem se zahtev šalje, i to u vidu dela URL-a koji se naziva *upit* (engl. *query*). Upit se od ostatka dela URL-a odvaja karakterom `?` iza kojeg slede podaci zapisani u formatu `ime=vrednost` odvojeni karakterom `&`, na primer:

```
1 field1=value1&field2=value2&field3=value3...
```

S obzirom da mi očekujemo koordinate vektora $x = (x_1, x_2, x_3)$ i $y = (y_1, y_2, y_3)$, očekujemo da će klijent poslati **GET** zahtev ka URL-u:

```
1 http://localhost:3000?x1=0&x2=0&x3=0&y1=0&y2=0&y3=0
```

Naravno, nule u gornjem URL-u će biti zamenjene vrednostima koje klijent zaista bude poslao ka serveru. Da bismo dohvatili URL zahteva klijenta, potrebno je da pristupimo svojstvu `url` objekta zahteva `req`. Sada je potrebno da izvučemo informacije iz ovog URL-a, što se može postići upotrebom paketa `url` koji dolazi instaliran uz Node.js. Funkcijom `url.parse`⁹ možemo parsirati URL nisku, čime dobijamo objekat koji sadrži razne informacije o parsiranom URL-u. Nama je korisno svojstvo `query`, čija je vrednost objekat koji ima svojstva koji odgovaraju prosleđenim podacima:

```
1 const url = require('url');
2
3 // ...
4
5 if (method === "GET")
6 {
7     let urlString = req.url;
8     let urlObj = url.parse(urlString, true);
9     let getPodaci = urlObj.query;
10    let x = [getPodaci.x1, getPodaci.x2, getPodaci.x3];
11    let y = [getPodaci.y1, getPodaci.y2, getPodaci.y3];
12
13    // Ostatak obrade GET zahteva ide ovde...
14 }
```

Primetimo da je potrebno da prosledimo kao drugi argument funkcije `url.parse` vrednost `true` da bi se parsirao i upit kao deo URL-a (podrazumevano neće biti parsiran).

Sada kada smo dohvatili podatke, kreirajmo funkciju koja će izračunati skalarni proizvod. Zapravo, kreiraćemo svoj CommonJS modul, da bismo ispoštovali arhitekturu Node.js okruženja:

```
1 // U okviru datoteke 'skalarni-proizvod.js'
2
3 function skalarni_proizvod(x, y)
4 {
5     let sp = 0;
6     for (let i = 0; i < x.length; ++i)
7     {
8         sp += x[i] * y[i];
9     }
10
11     return sp;
12 }
13
14 module.exports.skalarni_proizvod = skalarni_proizvod;
```

Vidimo da modul eksportuje jednu funkciju, koju možemo dohvatiti njegovim uključivanjem u `server.js` datoteku:

```
1 const {skalarni_proizvod} = require('./skalarni-proizvod');
```

Sada možemo da koristimo funkciju `skalarni_proizvod` iz našeg modula:

⁹Dokumentacija

```

1  if (method === "GET")
2  {
3      // ...
4
5      let rezultat = skalarni_proizvod(x, y);
6
7      res.setHeader('Content-Type', 'application/json');
8      teloOdgovora = {rezultat};
9  }

```

Primitimo da smo morali da stavimo vitičaste zagrade oko naziva funkcije. To je zato što ćemo pozivom funkcije `require` dohvatiti objekat koji sadrži tu funkciju. Mogli smo napisati i

```
1  const spPaket = require('./skalarni-proizvod');
```

čime bismo onda umesto poziva `skalarni_proizvod(x, y)` morali da koristimo `spPaket.skalarni_proizvod(x, y)`.

Ceo izvorni kod implementacije na serveru dat je u nastavku:

Kod 5.2: nodejs/002/server.js

```

1  const http = require('http');
2  const url = require('url');
3  const {skalarni_proizvod} = require('./skalarni-proizvod');
4
5  const server = http.createServer();
6  server.on('request', function(req, res)
7  {
8      let teloOdgovora = {};
9      let statusniKod = 200;
10
11      let method = req.method;
12
13      if (method === "GET")
14      {
15          let urlString = req.url;
16          let urlObj = url.parse(urlString, true);
17          let getPodaci = urlObj.query;
18          let x = [getPodaci.x1, getPodaci.x2, getPodaci.x3];
19          let y = [getPodaci.y1, getPodaci.y2, getPodaci.y3];
20
21          let rezultat = skalarni_proizvod(x, y);
22
23          res.setHeader('Content-Type', 'application/json');
24          teloOdgovora = {rezultat};
25      }
26      else if (method === "OPTIONS")
27      {
28          res.setHeader('Access-Control-Allow-Methods', 'OPTIONS, GET');
29      }
30      else
31      {
32          statusniKod = 405;
33          res.setHeader('Access-Control-Allow-Methods', 'OPTIONS, GET');
34      }
35
36      res.writeHead(statusniKod, {
37          'Access-Control-Allow-Origin': '*',
38          'Access-Control-Allow-Headers': 'Content-Type'

```

```

39     });
40     res.write(JSON.stringify(teloOdgovora));
41     res.end();
42 });
43
44 const port = 3000;
45 server.listen(port);
46 server.once('listening', function()
47 {
48     console.log(`http://localhost:${port}`);
49 });

```

Kod 5.3: nodejs/002/skalarni-proizvod.js

```

1 function skalarni_proizvod(x, y)
2 {
3     let sp = 0;
4     for (let i = 0; i < x.length; ++i)
5     {
6         sp += x[i] * y[i];
7     }
8
9     return sp;
10 }
11
12 module.exports.skalarni_proizvod = skalarni_proizvod;

```

Pretpostavimo da nam je na raspolaganju jednostavna klijentska aplikacija koja koristi biblioteku jQuery za validaciju podataka na klijentu i slanje asinhronih zahteva, čija je implementacija data u nastavku:

Kod 5.4: nodejs/002/skalarni_proizvod.html

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <meta name="viewport" content="width=device-width, initial-scale=1.0">
6     <title>Skalarni proizvod</title>
7
8     <link rel="stylesheet" type="text/css" href="bootstrap.css">
9
10    <style>
11        #greska {
12            color: red;
13        }
14    </style>
15 </head>
16 <body>
17     <div class="container text-center">
18         <form id="f" method="GET" action="" novalidate>
19             <fieldset class="form-group">
20                 <legend>Vektor x</legend>
21                 <div class="form-group">
22                     <label for="x1">x:</label>
23                     <input type="text" id="x1" name="x1">
24                     <input type="text" id="x2" name="x2">
25                     <input type="text" id="x3" name="x3">
26                 </div>
27             </fieldset>
28             <fieldset class="form-group">

```

```

29         <legend>Vektor y</legend>
30         <div class="form-group">
31             <label for="y1">y:</label>
32             <input type="text" id="y1" name="y1">
33             <input type="text" id="y2" name="y2">
34             <input type="text" id="y3" name="y3">
35         </div>
36     </fieldset>
37     <div id="greska"></div>
38     <fieldset class="form-group">
39         <legend>Skalarni proizvod</legend>
40         <div class="form-group">
41             <label for="rezultat">Rezultat:</label>
42             <input type="text" id="rezultat" name="rezultat">
43         </div>
44     </fieldset>
45     <div class="form-group">
46         <input type="submit" value="Izračunaj" class="btn btn-primary">
47     </div>
48 </form>
49 </div>
50
51 <script type="text/javascript" src="jquery-3.3.1.min.js"></script>
52 <script type="text/javascript">
53     $("#f").submit(function()
54     {
55         // Brisanje rezultata od proslog izacunavanja
56         $("#rezultat").val("");
57
58         // Dohvatanje vrednosti i validacija
59         let x1 = Number.parseInt($("#x1").val());
60         let x2 = Number.parseInt($("#x2").val());
61         let x3 = Number.parseInt($("#x3").val());
62         let y1 = Number.parseInt($("#y1").val());
63         let y2 = Number.parseInt($("#y2").val());
64         let y3 = Number.parseInt($("#y3").val());
65
66         if (isNaN(x1) || isNaN(x2) || isNaN(x3) || isNaN(y1) || isNaN(y2)
67             || isNaN(y3))
68         {
69             $("#greska").text("Treba da unesete brojeve!");
70             return false;
71         }
72
73         let podaci = {x1, x2, x3, y1, y2, y3};
74
75         // Dohvatanje rezultata.
76         // Primetimo da za slanje podataka koristimo GET metod,
77         // tako da nije potrebno da ih serijalizujemo i postavimo
78         // contentType svojstvo,
79         // zato sto se ti podaci ugnezjdjavaju u URL zahteva.
80         $.ajax("http://localhost:3000", {
81             method: "GET",
82             data: podaci,
83             success: function(data, textStatus, jqXHR)
84             {
85                 console.log(data);
86
87                 if (data.hasOwnProperty("rezultat"))
88                 {
89                     $("#rezultat").val(data.rezultat);
90                 }
91             }
92         });
93     });
94 </script>

```



```
88         }
89     },
90     error: function(jqXHR, textStatus, errorThrown)
91     {
92         $("#greska").text("Postoji greska sa zahtevom: " +
93             textStatus);
94     }
95 });
96 // Cistimo polje za gresku
97 $("#greska").text("");
98
99 // Sprečavanje da veb pregledac prosledi podatke na stranicu zadatu
   u atributu action
100 return false;
101 });
102 </script>
103 </body>
104 </html>
```

Prikaz klijentske aplikacija izgleda kao na narednoj slici:

Vektor x

x:

Vektor y

y:

Skalarni proizvod

Rezultat:

Čitaocu se ostavlja da pokrene serversku aplikaciju i testira je korišćenjem date klijentske aplikacije. Radi kompletnosti, u nastavku rešenja diskutujemo o slanju asinhronog zahteva pomoću biblioteke jQuery.

Videli smo kako je moguće proslediti asinhroni zahtev ka serveru korišćenjem objekta klase `XMLHttpRequest`. Međutim, uverili smo se da je čak i na jednostavnim primerima rad sa ovim objektom veoma neugodan i kod ne izgleda lepo. Na našu sreću, biblioteka jQuery definiše svoju funkciju za slanje asinhronih poziva: `$.ajax(url [, settings])`. Argumenti ove funkcije su:

1. `url` — Niska koja sadrži putanju na koju je potrebno poslati asinhroni zahtev.
2. `settings` — Objekat kojim se vrši konfiguracija asinhronog zahteva. Neke od vrednosti koje možemo podešavati su:
 - `method` — Niska koja sadrži HTTP metod zahteva.

- **data** — Objekat koji definiše podatke koji se šalju u zahtevu. Ukoliko je za **method** izabran **GET** zahtev, podaci će biti prosleđeni kao upit u URL-u. Inače, možemo birati u kom formatu ćemo poslati podatke — na primer, ukoliko želimo da podaci budu poslani u JSON formatu, potrebno ih je serijalizovati tako što se proslede funkciji **JSON.stringify()** i potrebno je postaviti vrednost **contentType** na **'application/json'** (videti ispod).
- **contentType** — Niska kojom se definiše MIME tip podataka koji se šalje.
- **success** — Po-potrebi-pozivna funkcija koja će se izvršiti ukoliko zahtev prođe uspešno. Funkcija može primiti tri argumenta:
 - (a) **data** — Telo odgovora od servera.
 - (b) **textStatus** — Statusna poruka koja je dobijena od servera.
 - (c) **jqXHR** — Specijalan jQuery objekat koji predstavlja nadskup **XMLHttpRequest** klase¹⁰.
- **error** — Po-potrebi-pozivna funkcija koja će se izvršiti ukoliko zahtev prođe neuspešno. Funkcija može primiti tri argumenta:
 - (a) **jqXHR** — Specijalan jQuery objekat koji predstavlja natklasu **XMLHttpRequest** klase¹¹. To znači da možemo da koristimo ista svojstva i metode kao nad običnim **XMLHttpRequest** objektom.
 - (b) **textStatus** — Statusna poruka koja je dobijena od servera. Može biti neka od: **'timeout'**, **'error'**, **'abort'** ili **'parsererror'**.
 - (c) **errorThrown** — Objekat koji predstavlja grešku.

Slanje asinhronog zahteva sada možemo uraditi na sledeći način:

```

1 $.ajax("http://localhost:3000", {
2   method: "GET",
3   data: podaci,
4   success: function(data, textStatus, jqXHR)
5   {
6     if (data.hasOwnProperty("rezultat"))
7     {
8       $("#rezultat").val(data.rezultat);
9     }
10  },
11  error: function(jqXHR, textStatus, errorThrown)
12  {
13    $("#greska").text("Postoji greska sa zahtevom: " + textStatus);
14  }
15 });
```

Ono što treba da zapamtimo kod asinhronih zahteva kreiranih na ovaj način jeste da će uvek biti izvršena ili funkcija **success** ili funkcija **error**, nikada obe.

5.3 Razvojno okruženje Express.js

Sada kada smo naučili kako da parsiramo URL putanju i da odredimo koji HTTP metod je klijent koristio da komunicira sa serverom, u stanju smo da pišemo najrazličitije serverske aplikacije. Ono što bismo brzo shvatili jeste da čim želimo da našoj aplikaciji dodamo novu funkcionalnost, veličina našeg programskog koda značajno raste. Takođe, ništa nas ne sprečava da razmišljamo o arhitekturi izvornog koda, što opet nije dobro.

Ono o čemu nismo diskutovali do sada jeste kako bismo izvršavali parsiranje tela HTTP

¹⁰Dokumentacija

¹¹Dokumentacija

zahteva. Iako sam proces parsiranja u "čistom" Node.js-u nije komplikovan za razumevati, on ipak predstavlja akciju koju je zamorno svaki put pisati. Bilo bi mnogo bolje koristiti pakete trećih lica za akcije slične ovoj.

Kao odgovor na sve ove zahteve, a takođe i da bismo ubrzali razvoj naših serverskih aplikacija, na raspolaganju su nam razna razvojna okruženja, od kojih ćemo se mi upoznati sa Express.js¹². Ovo okruženje predstavlja minimalističko i fleksibilno razvojno okruženje koje je dovoljno moćno da nam olakša razvoj raznih veb aplikacija. Ono za šta ćemo ga mi koristiti jeste za postavljanje arhitekture serverskog koda koja će se jednostavno proširivati svakim novim zahtevom.

Instalacija Express.js razvojnog okruženja je jednostavna:

```
1 $ npm install express
```

5.3.1 Opsluživanje statičkih i dinamičkih sadržaja

U ovoj podsekciji govorićemo o načinima na koje je moguće koristiti Node.js okruženje za kreiranje, na primer, HTML stranica koje su bilo statičkog tipa (u smislu da je njihov sadržaj uvek isti) bilo dinamičkog tipa (u smislu da možemo na dinamički način generisati njihov sadržaj), a koje ćemo zatim vratiti kao odgovor od servera ka klijentu. Ovaj način rada odgovara klasičnom načinu programiranja u jeziku PHP, u kojem se HTML sadržaj generiše izvršavanjem skript datoteka sa ekstenzijom `.php` na veb serveru, a zatim se generisani sadržaj prenosi do klijenta i prikazuje u veb pregledaču.

Za početak ćemo se upoznati sa različitim elementima okruženja za razvoj Express.js, a zatim ćemo ih primeniti na implementaciji nešto ozbiljnije serverske aplikacije za rad sa studentima.

Kao i do sada, kreirajmo `server.js` datoteku i kojoj ćemo instancirati server i pokrenuti ga:

```
1 const http = require('http');
2
3 const app = require('./app');
4
5 const port = process.env.PORT || 3000;
6
7 const server = http.createServer(app);
8 server.listen(port);
9
10 server.once('listening', function() {
11   console.log(`Listening on port ${port}`);
12 });
```

Ono što odmah primećujemo jeste da smo odmah nametnuli modularnost našoj aplikaciji: datoteka `server.js` služi samo za instanciranje i pokretanje serverske aplikacije, dok ćemo u modulu `app.js` smeštati sav kod koji implementira obradu HTTP zahteva od klijenta i vraćanje odgovarajućih HTTP odgovora. Dodatno, taj modul, za koji ćemo ubrzo videti da predstavlja instancu Express.js aplikacije, koristimo pri kreiranju servera umesto da kreiramo funkciju koja će reagovati na svaki događaj pristizanja zahteva (odnosno, događaj `'request'`).

Promenljivama okruženja (engl. *environment variable*) možemo pristupati kroz objekat `process.env` čija svojstva sadrže vrednosti tih promenljivih. U slučaju da nam nije dostup-

¹²<https://expressjs.com/>

na promenljiva `PORT`, koristićemo port 3000, kao i do sada. Ukoliko koristimo aplikaciju `nodemon` za pokretanje servera, onda možemo kreirati datoteku `modemon.json` u kojoj možemo da konfigurišemo pokretanje ovog programa, na primer, možemo postaviti promenljivu okruženja `PORT` na sledeći način:

```
1 {  
2   "env":  
3     {  
4       "PORT": "3000"  
5     }  
6 }
```

Kreiranje Express.js aplikacija

Predimo na kreiranje `app` modula. S obzirom da koristimo Express.js razvojno okruženje, potrebno je da ga imamo uključeno.

```
1 const express = require('express');
```

Kreiranje instance Express.js aplikacije se vrši na jednostavan način — pozivanjem modula kao funkcije, čime će Express.js za nas uraditi veliki broj stvari da bi kreirao objekat koji će rukovati zahtevima:

```
1 const app = express();
```

Ono što izvozimo kao `app` modul jeste upravo instanca Express.js aplikacije, pa ćemo podrazumevati da na kraju datoteke `app.js` postoji naredna linija:

```
1 module.exports = app;
```

Kako sada možemo da registrujemo funkcije koje obrađuju zahteve? Pre odgovora na to pitanje, veoma je važno da razumemo način na koji okruženje za razvoj Express.js funkcioniše, što će nam dati objašnjenje zašto je ona toliko popularna. U Express.js aplikacijama, od trenutka kada HTTP zahtev pristigne do aplikacije, do trenutka kada aplikacija reši da pošalje odgovor (na taj zahtev), HTTP zahtev prolazi kroz niz funkcija koje se nazivaju *srednji slojevi* (engl. *middleware*). Postoji veliki broj biblioteka trećih lica koje implementiraju najrazličitije srednje slojeve za Express.js aplikacije, kao što su parsiranje tela zahteva, zaštitni mehanizmi i mnogi drugi, a koje su nama na raspolaganju za korišćenje. Napomenimo da zahtev ne mora da prođe kroz sve srednje slojeve koje mi definišemo, o čemu ćemo pričati nešto kasnije.

Dakle, ukoliko želimo da definišemo funkciju koja obrađuje zahtev od klijenta, potrebno je da registrujemo svoj srednji sloj (zapravo, možemo registrovati jedan ili više njih) kojim ćemo definisati implementaciju obrade. Registrovanje srednjeg sloja se vrši pozivom metoda `use([path,] callback [, callback...])`¹³ nad instancom Express.js aplikacije. Ovaj metod ima dva osnovna preopterećenja:

1. Ukoliko se pozove sa jednim argumentom, onda se očekuje funkcija srednjeg sloja `callback` koja će biti izvršena nad svakim HTTP zahtevom koji dođe do tog srednjeg sloja.
2. Ukoliko se pozove sa dva argumenta, onda se očekuje putanja zadata niskom `path` i funkcija srednjeg sloja `callback` koja će biti izvršena samo ukoliko prefiks URL-a HTTP zahteva (bez sheme i DNS imena serverske aplikacije) odgovara prosleđenom argumentu `path`. Na primer:

¹³Dokumentacija

- ako registrujemo srednji sloj za vrednost `'/'` argumenta `path`, onda je to ekvivalentno kao da smo registrovali srednji sloj na prvi način.
 - ako registrujemo srednji sloj za vrednost `'/student'` argumenta `path`, onda će zahtev biti obrađen od strane HTTP zahteva čiji su URL-ovi dati sa:
 - `http://localhost:3000/student`,
 - `http://localhost:3000/studenti`,
 - `http://localhost:3000/studentska-sluzba`,
 - ...
- ali ne i URL-ovi dati sa:
- `http://localhost:3000/`,
 - `http://localhost:3000/ispit`,
 - `http://localhost:3000/neka-nasumicna-putanja`,
 - ...

Sama funkcija srednjeg sloja očekuje (barem) tri argumenta: `req`, `res` i `next`. Prvi predstavlja objekat HTTP zahteva¹⁴, drugi predstavlja objekat HTTP odgovora¹⁵, a treći predstavlja naredni srednji sloj koji može biti pozvan ukoliko želimo da se tekući zahtev prosledi na dalju obradu (narednom registrovanom srednjem sloju). Bitno je napomenuti da, pri konstruisanju srednjih slojeva, uzmemo u obzir naredno pravilo:

Funkcija srednjeg sloja mora da ili obradi zahtev i prosledi ga narednom srednjem sloju na obradu ili završi obradu i vrati odgovor klijentu.

Nepoštovanje ovog pravila dovodi do toga da se HTTP zahtev "zaglavi" u srednjem sloju, što najčešće dovodi to *isteka vremena za obradu* (engl. *timeout*) HTTP zahteva na klijentu.

Registrovanje srednjeg sloja koji će obraditi svaki HTTP zahtev koji dođe do njega se može uraditi na sledeći način:

```
1 app.use((req, res, next) => {
2   // Obrada zahteva ide ovde...
3 });
```

Na primer, neka želimo da, kada korisnik učitava početnu stranicu naše serverske aplikacije, u klijentu mu se prikaže naslov "Fakultet" i tekst "Dobrodošli na početnu stranicu fakulteta!":

```
1 app.use((req, res, next) => {
2   const html = `
3     <h1>Fakultet</h1>
4     <p>Dobrodošli na početnu stranicu fakulteta!</p>`;
5
6   res.send(html);
7 });
```

Ono što vidimo jeste da za slanje odgovora od servera možemo koristiti metod `send`¹⁶ nad objektom HTTP odgovora. Taj metod može prihvatiti opcioni argument `body`, što predstavlja telo odgovora, i može biti bilo kojeg tipa. Nije loše napomenuti da ukoliko prosledimo nisku kao telo odgovora, tj. kao parametar metoda `send`, podrazumevaće se da je poslat odgovor u HTML formatu, dok za ostale tipove `boolean`, `number` i `object`,

¹⁴Dokumentacija

¹⁵Dokumentacija

¹⁶Dokumentacija

postavlja se binarni tip kao odgovor. Naravno, mi možemo postaviti svoj tip postavljanjem zaglavlja `Content-Type`, što se vrši metodom `header`¹⁷¹⁸ na sledeći način:

```
1 app.use((req, res, next) => {
2   const html = `
3     <h1>Fakultet</h1>
4     <p>Dobrodošli na početnu stranicu fakulteta!</p>`;
5
6   res.header('Content-Type', 'text/html');
7   res.send(html);
8 });
```

Ono što nam je dodatno važno da znamo kako da upravljamo jeste postavljanje statusnog koda od servera. To možemo izvršiti pozivom metoda `status`¹⁹ kome se prosleđuje celobrojna vrednost koja predstavlja statusni kod:

```
1 app.use((req, res, next) => {
2   const html = `
3     <h1>Fakultet</h1>
4     <p>Dobrodošli na početnu stranicu fakulteta!</p>`;
5
6   res.header('Content-Type', 'text/html');
7   res.status(200);
8   res.send(html);
9 });
```

Ono što je korisno znati jeste da metod `status` vraća HTTP odgovor, pa je moguće nad tim pozivom vršiti ulančavanje metoda:

```
1 app.use((req, res, next) => {
2   const html = `
3     <h1>Fakultet</h1>
4     <p>Dobrodošli na početnu stranicu fakulteta!</p>`;
5
6   res.header('Content-Type', 'text/html');
7   res.status(200).send(html);
8 });
```

Zadatak 5.1 Pokrenuti do sada implementiranu serversku aplikaciju, a zatim u veb pregledaču otvoriti adresu <http://localhost:3000/> i uveriti se da se prikazuje odgovarajući naslov i tekst. Proveriti da li je statusni kod 200²⁰. ■

Sada želimo da implementiramo stranicu za dodavanje novog studenta u bazu podataka. Naravno, u ovom poglavlju nećemo zaista rukovati bazom podataka na serverskom delu (taj zadatak ostavljamo za naredno poglavlje), ali prikazaćemo kako je moguće obrađivati podatke koje dobijamo od klijenta. Neka se ova stranica nalazi na adresi <http://localhost:3000/dodaj-studenta>. Implementaciju ovog zahteva dodajemo ispred prethodno registrovanog srednjeg sloja:

```
1 app.use('/dodaj-studenta', (req, res, next) => {
2   const html = `
```

¹⁷Dokumentacija

¹⁸Ovaj metod predstavlja alias za metod `set` i mogu se koristiti ravnopravno. Autor sugeriše korišćenje aliasa `header` zbog veće izražajnosti u nazivu metoda.

¹⁹Dokumentacija

²⁰Može se desiti da je statusni kod 304 koji označava da nije došlo do promene dokumenta od njegovog prethodnog učitavanja.

```
3     <form action="/registruj-studenta" method="POST">
4         <input type="text" name="indeks">
5         <input type="submit" value="Registruj studenta">
6     </form>`;
7
8     res.status(200).send(html);
9 });
```

Ovo radimo da ne bismo izvršiti srednji sloj za obradu početne stranice, s obzirom da je on registrovan za putanju `'/'`.

Na toj stranici prikazujemo jednostavan formular u kojem je potrebno uneti indeks studenta koji bi trebalo metodom `POST` da bude poslat na adresu `http://localhost:3000/registruj-studenta`. Za sada ćemo samo prikazati podatke u konzoli na veb serveru i preusmeriti korisnika na `http://localhost:3000/uspesna-registracija` gde će mu biti prikazana poruka o uspešnosti akcije.

S obzirom da zahtev šaljemo metodom `POST`, ovo je dobar trenutak da pokažemo kako se parsiraju tela zahteva. Kao što smo rekli na početku, ovo je jedna od akcija koju ne želimo mi da radimo, već želimo da iskoristimo neku biblioteku trećeg lica koja nam to omogućava. Mi ćemo koristiti paket `body-parser`:

```
1 $ npm install body-parser
```

Ovaj paket dolazi uz pregršt funkcija srednjeg sloja koji mogu da parsiraju najrazličitije formate. Mi ćemo se za sada zadržati na URL-kodirane i JSON formate. Registracija srednjih slojeva za ova dva formata se može postići na sledeći način:

```
1 const bodyParser = require('body-parser');
2
3 // ...
4
5 app.use(bodyParser.urlencoded({extended: false}));
6 app.use(bodyParser.json());
```

Svaki od ovih srednjih slojeva pozivaju naredni sloj u nizu, tako da znamo da će HTTP zahtev sigurno doći i do srednjih slojeva koje mi implementiramo. Stoga najviše ima smisla registrovati ove srednje slojeve pre onih koje smo mi implementirali. Više o ovom paketu, kao i o pomenutim metodima i njihovim argumentima se može pronaći na adresi <https://expressjs.com/en/resources/middleware/body-parser.html>.

Ukoliko HTTP zahtev prođe kroz ove srednje slojeve za parsiranje, onda nam je nad objektom HTTP zahteva dostupno svojstvo `body` (dakle, `req.body`) čija je vrednost objekat sa svojstvima koja odgovaraju parovima (ključ, vrednost) u telu zahteva. Sada možemo da implementiramo registraciju studenta:

```
1 app.use('/registruj-studenta', (req, res, next) => {
2     console.log(req.body);
3
4     res.redirect('/uspesna-registracija');
5 });
```

Vidimo da se preusmeravanje izvršava pozivom metoda `redirect`²¹ kome se prosleđuje lokacija. Opciono, možemo specificovati statusni kod kao prvi argument, a podrazumevano se koristi 302.

²¹Dokumentacija

```

1 app.use('/uspesna-registracija', (req, res, next) => {
2   const html = `
3     <p>Uspešno ste registrovali novog studenta</p>
4     <form action="/" method="GET">
5       <input type="submit" value="Početna strana">
6     </form>`;
7
8   res.status(200).send(html);
9 });

```

Zadatak 5.2 Pokrenuti do sada implementiranu serversku aplikaciju, a zatim u veb pregledaču otvoriti adresu <http://localhost:3000/dodaj-studenta> i dodati novog studenta. Pratiti u programerskim alatima pregledača (konkretno, u alatu **Network**) koje sve stranice veb pregledač učitava pri ovoj akciji i podatke koje se tim stranicama šalju. Dodatno, u konzoli serverske aplikacije pratiti ispis tela HTTP zahteva pri učitavanju adrese <http://localhost:3000/registruj-studenta> i uveriti se da je uneseni indeks zaista došao do servera. ■

5.3.2 Postavljanje REST arhitekture

Primer 5.3 Napisati serversku aplikaciju koja se pokreće na portu 3000 i koja treba da opslužuje naredne HTTP zahteve:

- GET zahtev nad <http://localhost:3000/products>
 - Vratiti statusni kod 200 sa porukom **'GET /products'**
- POST zahtev nad <http://localhost:3000/products>
 - Kreirati objekat koji ima svojstva **name** i **price**, čije se vrednosti očitavaju iz tela zahteva. Telo se prosleđuje u JSON formatu. Vratiti statusni kod 201 sa porukom **'POST /products'** i vratiti novokreirani objekat.
- GET zahtev nad <http://localhost:3000/products> sa parametrom **productId**
 - Očitati parametar **productId** i ukoliko je parametar jednak vrednosti **'special'**, vratiti poruku **'Otkrili ste specijalni ID!'**. U ostalim slučajevima vratiti poruku **'Prosledjen je ID: \${productId}'**. Vratiti statusni kod 200 u oba slučaja.
- PATCH zahtev nad <http://localhost:3000/products> sa parametrom **productId**
 - Vratiti statusni kod 200 sa porukom **'Azuriran je proizvod'**
- DELETE zahtev nad <http://localhost:3000/products> sa parametrom **productId**
 - Vratiti statusni kod 200 sa porukom **'Obrisan je proizvod'**
- OPTIONS zahtev nad http://localhost:3000/*
 - Server odgovara da su dostupni metodi OPTIONS, GET, POST, PATCH, DELETE.

Koristiti razvojno okruženje Express.js za rutiranje zahteva. Ispravno implementirati CORS mehanizam zaštite. Zatim, kreirati formulare, stilizovane pomoću biblioteke Bootstrap, kojima se mogu testirati zahtevi na klijentskoj strani i pomoću biblioteke jQuery izvršiti odgovarajuće validacije i poslati asinhroni zahtev ka serverskoj aplikaciji sa odgovarajućim podacima.

Rešenje. Rešenje ovog zadatka ćemo uraditi pristupom odozgo-nadole, odnosno, prvo ćemo pisati najapstraktniji kod, pa ćemo onda ulaziti u detalje. Dakle, prvo što je potrebno

jeste da kreiramo `server.js` datoteku i kojoj ćemo instancirati server i pokrenuti ga:

```
1 const http = require('http');
2
3 const app = require('./app');
4
5 const port = process.env.PORT || 3000;
6
7 const server = http.createServer(app);
8
9 server.listen(port);
10 server.once('listening', function() {
11   console.log(`Listening on port ${port}`);
12 });
```

Odavde već vidimo da ćemo napraviti modul `app` koji će služiti kao oslušivač za zahteve ka našoj serverskoj aplikaciji. Kreiranje zasebnog modula ima smisla s obzirom da imamo veliki broj zahteva koji treba da opslužimo. U ovom slučaju je dovoljno proslediti modul metodu `createServer`.

Promenljivama okruženja (engl. *environment variable*) možemo pristupiti kroz objekat `process.env` čija svojstva sadrže vrednosti tih promenljivih. U slučaju da nam nije dostupna promenljiva `PORT`, koristićemo port 3000, kao i do sada. Ukoliko koristimo aplikaciju `nodemon` za pokretanje servera, onda možemo kreirati datoteku `modemon.json` u kojoj možemo da konfigurišemo pokretanje ovog programa, na primer, možemo postaviti promenljivu okruženja `PORT`:

```
1 {
2   "env":
3     {
4       "PORT": "3000"
5     }
6 }
```

Pređimo na kreiranje `app` modula. S obzirom da koristimo Express.js razvojno okruženje, potrebno je da ga imamo uključeno.

```
1 const express = require('express');
```

Kreiranje instance Express.js aplikacije se vrši na jednostavan način — pozivanjem modula kao funkcije:

```
1 const app = express();
```

S obzirom da će naša aplikacija koristiti tela HTTP zahteva (u nekim zahtevima), bilo bi dobro da naš zahtev prvo "prođe" kroz module za parsiranje tela. Takvih modula ima mnogo, a mi ćemo koristiti paket `body-parser`, koji se jednostavno instalira:

```
1 $ npm install body-parser
```

Da bismo "provukli" HTTP zahtev kroz `body-parser`, potrebno je da nad našom Express.js aplikacijom pozovemo metod `use([path,] callback [, callback...])`²². Ovim metodom se registruje *srednji sloj* (engl. *middleware*) koji će obraditi zahtev kada se putanja zahteva poklopi sa promenljivom `path`. Ukoliko nije specifikovana putanja, onda će svaki zahtev proći kroz registrovani srednji sloj. Srednji sloj ne predstavlja ništa drugo do funkciju koja može da uradi nekakvu vrstu obrade nad zahtevom koji prima. Paket `body-parser` definiše

²²Dokumentacija

više funkcija srednjeg sloja, kao što su `urlencoded`, koji će ispravno kodirati URL za prenos preko interneta i `json`, koji će parsirati telo u formatu JSON. Argument ovih funkcija je objekat sa specifikacijama parsiranja:

```
1 app.use(bodyParser.urlencoded({
2   extended: false
3 }));
4 app.use(bodyParser.json({}));
```

Prva stvar koju je potrebno uraditi jeste napisati srednji sloj koji će ispravno obraditi CORS greške. U Express.js razvojnom okruženju, ovo je jednostavno uraditi:

```
1 app.use(function(req, res, next) {
2   res.header('Access-Control-Allow-Origin', '*');
3   res.header('Access-Control-Allow-Headers', 'Content-Type');
4
5   if (req.method === 'OPTIONS')
6   {
7     res.header('Access-Control-Allow-Methods', 'OPTIONS, GET, POST, PATCH,
8       DELETE');
9     return res.status(200).json({});
10  }
11  next();
12 });
```

Ovde vidimo da smo podesili da naša serverska aplikacija prihvata zahteve sa svih izvora, a da od dodatnih zaglavlja prihvata samo `Content-Type` zaglavlje. Dodatno, već na ovom mestu je sasvim korektno ispitati da li je poslat `OPTIONS` zahtev i odgovoriti na njega. Postavljanje zaglavlja se vrši metodom `header(field [, value])`²³. Postavljanje statusnog koda se može uraditi metodom `status(code)`²⁴ koji je moguće ulančati, na primer, metodom `json([body])`²⁵, koji će postaviti telo odgovora u JSON formatu.

Alternativno, možemo koristiti metod `send` za slanje odgovora od servera. Taj metod može prihvatiti opcioni argument `body`, što predstavlja telo odgovora, i može biti bilo kojeg tipa. Možda nije loše napomenuti da ukoliko prosledimo nisku kao telo odgovora, tj. kao parametar metoda `send`, podrazumevaće se da je poslat odgovor u HTML formatu, dok za ostale tipove `boolean`, `number` i `object`, postavlja se binarni tip kao odgovor. Naravno, mi možemo postaviti svoj tip postavljanjem zaglavlja `Content-Type`.

Ono što vidimo jeste da srednji slojevi mogu biti funkcije od tri argumenta. Prvi predstavlja objekat HTTP zahteva²⁶, drugi predstavlja objekat HTTP odgovora²⁷, a treći predstavlja naredni srednji sloj koji može biti pozvan ukoliko želimo da se tekući zahtev prosledi na dalju obradu (narednom registrovanom srednjem sloju). Bitno je zapamtiti da je potrebno da naši srednji slojevi ili pozivaju naredni srednji sloj ili da izađu iz njega. U slučaju da je poslat `OPTIONS` zahtev, nema smisla dalje obrađivati taj zahtev, te je dovoljno da postavimo zaglavlja i prekinemo dalju obradu (tako što izađemo iz funkcije). Sa druge strane, ukoliko zahtev nije bio `OPTIONS`, to znači da ga je potrebno dodatno obraditi nekim od narednih srednjih slojeva, te zbog toga pozivamo funkciju `next`, koja će za nas uraditi upravo to.

²³Dokumentacija

²⁴Dokumentacija

²⁵Dokumentacija

²⁶Dokumentacija

²⁷Dokumentacija

Sada je potrebno da definišemo srednje slojeve koji će vršiti obradu zahteva. S obzirom da sve preostale zahteve možemo smestiti pod putanjom `/products`, ima smisla napraviti odvojen modul. Pošto je ovaj modul deo našeg API-ja, smestićemo ga u direktorijumu `./api/routes`:

```
1 const productRoutes = require('./api/routes/products');
```

Sada možemo da ga koristimo kao srednji sloj:

```
1 app.use('/products', productRoutes);
```

Međutim, još uvek nismo završili sa `app.js` datotekom. Zašto? Odgovor leži u tome da nismo specifikovali šta će se desiti ako klijent pošalje zahtev na nevalidnu putanju, na primer, `http://localhost:3000/emptyStore`. Drugim rečima, ako nijedan od prethodna dva srednja sloja nije uspeo da obradi zahtev, onda je potrebno napisati srednji sloj kojim ćemo reći da je došlo do greške u zahtevu:

```
1 app.use(function(req, res, next) {  
2   const error = new Error('Zahtev nije podrzan od servera');  
3   error.status = 405;  
4  
5   next(error);  
6 });
```

Pomoću `new Error([message])` možemo napraviti novu grešku sa opcionom porukom koja će biti ispisana. Takođe, dodajemo statusni kod 405 kao deo greške. Iz ovog koda takođe vidimo da se poziva naredni srednji sloj sa greškom kao prvim argumentom.

Zašto je potrebno napraviti još jedan srednji sloj kad smo sve greške obradili? Zapravo, postoje neke greške koje se mogu pojaviti, a da nisu obrađene trenutnim kodom. Na primer, može da se desi da postoji deo koda koji nismo ispravno iskodirali ili, kada budemo koristili bazu podataka, da postoji neki problem u radu sa bazom, što će dovesti do pojavljivanja greške. Za sve takve situacije, definišemo finalni srednji sloj, koji je funkcija sa 4 argumenta, od kojih je prvi greška koja se javila, a preostali argumenti su standardni argumenti srednjih slojeva:

```
1 app.use(function(error, req, res, next) {  
2   res.status(error.status || 500).json({  
3     greska: {  
4       poruka: error.message  
5     }  
6   });  
7 });
```

Konačno, potrebno je da izvezemo našu Express.js aplikaciju kao modul i time smo završili kreiranje modula `app`:

```
1 module.exports = app;
```

Pređimo sada na kreiranje modula za proizvode. Kao i do sada, biće nam potreban `express` modul. Da bismo kreirali ovaj modul, nije potrebno da pravimo celu Express.js aplikaciju kao što smo to radili u modulu `app`. Cilj ovog modula jeste da definiše odgovarajuće funkcije za svaki od zahteva pod putanjom `/products`. Drugim rečima, modul treba da vrši *rutiranje* (engl. *routing*) zahteva na osnovu putanje, odnosno, preusmeravanje zahteva na ispravnu funkciju za obradu. Za rutiranje možemo da koristimo funkciju `express.Router()` koja kreira i vraća novi objekat za rutiranje:

```
1 const express = require('express');
2
3 const router = express.Router();
```

Nad ovim objektom su definisani razni metodi čiji nazivi odgovaraju HTTP metodima, uz napomenu da imena samih metoda sadrže isključivo mala slova. To znači da ako želimo da definišemo funkciju za obradu `GET` zahteva nad putanjom `/products`, dovoljno je da napišemo:

```
1 router.get('/', function(req, res, next) {
2     // Obrada GET zahteva nad /products
3 });
```

Ono što je možda zbunjujuće jeste zašto koristimo kao putanju nisku `"/"` umesto niske `"/products"`. Prisetimo se da smo u modulu `app` koristili modul `products` na sledeći način:

```
1 app.use('/products', productRoutes);
```

Time smo rekli da svi zahtevi koji počinju na putanji `/products` bivaju preusmereni na modul `products`. Zbog toga, kada zahtev stigne do modula `products`, on neće imati prefiks `/products` jer se on na tom mestu podrazumeva.

Implementacija obrade ovog zahteva je jednostavna — potrebno je vratiti statusni kod 200 i odgovarajuću poruku:

```
1 router.get('/', function(req, res, next) {
2     res.status(200).json({
3         poruka: 'GET /products'
4     });
5 });
```

Primetimo da nismo stavili naredbu `return` ispred `res`. To je zato što ukoliko tokom izvršavanja funkcije makar jednom postavimo statusni kod i telo odgovora, onda kada se funkcija završi, biće poslato to što je bilo postavljeno. Naravno, možemo koristiti i naredbu `return` ukoliko to želimo.

Da bismo dobili informaciju o telu HTTP zahteva, na raspolaganju nam je svojstvo `body` objekta HTTP zahteva. Nazivi njegovih svojstava odgovaraju nazivima prosleđenih podataka ka serveru. Napomenimo da ukoliko nismo prvobitno "provukli" HTTP zahtev kroz modul `body-parser`, vrednost ovog svojstva bi bila `null`. Sada smo u stanju da napišemo obradu `POST` zahteva nad putanjom `/products`:

```
1 router.post('/', function(req, res, next) {
2     const kreiraniProizvod = {
3         name: req.body.name,
4         price: req.body.price
5     };
6
7     res.status(201).json({
8         poruka: 'POST /products',
9         kreiraniProizvod
10    });
11 });
```

Ostalo je da još vidimo kako možemo da parametrizujemo zahteve. Da bismo rutirali parametrizovane putanje, dovoljno je da na kraj putanje nadovežemo karakter `:` koji je praćen nazivom parametra koji će nam biti dostupan u kodu. Na primer, ukoliko se putanja

`/:productId` iskoristi u modulu `products` na način kao što su to ostale putanje u tom modulu korišćene, onda ako bismo prosledili zahtev ka URL-u:

```
1 http://localhost:3000/products/2145621
```

u odgovarajućoj funkciji za obradu zahteva bismo imali parametar `productId` čija bi vrednost bila 2145621. Parametri zahteva su dostupni kroz svojstvo `params` objekta HTTP zahteva. Sada smo u stanju da napišemo funkcije za obradu ostalih zahteva:

```
1 router.get('/:productId', function(req, res, next) {
2   const productId = req.params.productId;
3
4   if (productId === 'special')
5   {
6     res.status(200).json({
7       poruka: 'Otkrili ste specijalni ID!',
8       id: productId
9     });
10  }
11  else
12  {
13    res.status(200).json({
14      poruka: 'Prosledjen je ID: ' + productId
15    });
16  }
17 });
18
19 router.patch('/:productId', function(req, res, next) {
20   res.status(200).json({
21     poruka: 'Azuriran je proizvod'
22   });
23 });
24
25 router.delete('/:productId', function(req, res, next) {
26   res.status(200).json({
27     poruka: 'Obrisan je proizvod'
28   });
29 });
```

Konačno, modul je potrebno eksportovati:

```
1 module.exports = router;
```

Ceo izvorni kod implementacije na serveru dat je u nastavku:

Kod 5.5: nodejs/rest1/nodemon.json

```
1 {
2   "env":
3   {
4     "PORT": "3000"
5   }
6 }
```

Kod 5.6: nodejs/rest1/server.js

```
1 const http = require('http');
2
3 const app = require('./app');
4
5 // Mozemo dodeliti port kroz promenljivu okruzenja,
6 // a u slucaju da nije dostupna,
```

```
7 // bice postavljena vrednost 3000
8 const port = process.env.PORT || 3000;
9
10 const server = http.createServer(app);
11
12 server.listen(port);
13 server.once('listening', function() {
14     console.log(`Listening on port ${port}`);
15 });
```

Kod 5.7: nodejs/rest1/app.js

```
1 const express = require('express');
2
3 // Paket za parsiranje tela zahteva
4 const bodyParser = require('body-parser');
5
6 // Kreiramo module za svaku putanju
7 const productRoutes = require('./api/routes/products');
8
9 // Nasa aplikacija ce u osnovi biti express aplikacija
10 const app = express();
11
12 // Dodajemo mogucnosti body-parser paketa
13 app.use(bodyParser.urlencoded({
14     extended: false // Zelimo samo jednostavne url encoded podatke da parsiramo
15 }));
16 app.use(bodyParser.json({
17     // Ovde takodje mozemo konfigurisati
18     // sta sve moze da se parsira
19 }));
20
21 // Da bismo resili CORS greske,
22 // moramo da pre obrade zahteva
23 // dodamo odgovarajuca zaglavlja
24 app.use(function(req, res, next) {
25     res.header('Access-Control-Allow-Origin', '*');
26     res.header('Access-Control-Allow-Headers', 'Content-Type');
27
28     if (req.method === 'OPTIONS')
29     {
30         res.header('Access-Control-Allow-Methods', 'OPTIONS, GET, POST, PATCH, DELETE');
31
32         return res.status(200).json({});
33     }
34
35     // Ovaj poziv ce "proslediti" zahtev narednom middleware-u
36     next();
37 });
38
39 // Za sve putanje koje pocinju sa "/products",
40 // zelimo da koristimo productRoutes middleware
41 // koji smo mi kreirali
42 app.use('/products', productRoutes);
43
44 // Ako je na ovom mestu HTTP zahtev prosao,
45 // to znaci da nije uhvacen od strane gornjih 'use' poziva.
46 // To znaci da je klijent poslao zahtev koji nismo definisali,
47 // pa mozemo da mu vratimo, na primer, 405 statusni kod
48 app.use(function(req, res, next) {
```

```
49     const error = new Error('Zahtev nije podrzan od servera');
50     error.status = 405;
51
52     // Sada moramo da prosledimo nasu kreiranu gresku,
53     // da bi se prikazao umesto podrazumevanog,
54     // pa pozivamo naredni middleware,
55     // sto je u ovom slucaju funkcija ispod
56     next(error);
57 });
58
59 // Dodajemo jos jedan nivo hvatanja gresaka,
60 // za sve ostale greske, tipa:
61 // 1. greske u kodu,
62 // 2. greske u bazi podataka.
63 // 3. ...
64 app.use(function(error, req, res, next) {
65     res.status(error.status || 500).json({
66         greska: {
67             poruka: error.message
68         }
69     });
70 });
71
72 module.exports = app;
```

Kod 5.8: nodejs/rest1/api/routes/products.js

```
1  const express = require('express');
2
3  const router = express.Router();
4
5  // Ne stavljamo /products,
6  // zato sto ce vec imati taj prefiks
7  // kad dodje do ove putanje,
8  // zbog app middleware-a
9  router.get('/', function(req, res, next) {
10     res.status(200).json({
11         poruka: 'GET /products'
12     });
13 });
14
15 router.post('/', function(req, res, next) {
16     const kreiraniProizvod = {
17         name: req.body.name,
18         price: req.body.price
19     };
20
21     res.status(201).json({
22         poruka: 'POST /products',
23         kreiraniProizvod
24     });
25 });
26
27 // Individualni
28
29 router.get('/:productId', function(req, res, next) {
30     const productId = req.params.productId;
31
32     if (productId === 'special')
33     {
34         res.status(200).json({
```

```

35         poruka: 'Otkrili ste specijalni ID!',
36         id: productId
37     });
38 }
39 else
40 {
41     res.status(200).json({
42         poruka: 'Prosledjen je ID: ' + productId
43     });
44 }
45 });
46
47 router.patch('/:productId', function(req, res, next) {
48     res.status(200).json({
49         poruka: 'Azuriran je proizvod'
50     });
51 });
52
53 router.delete('/:productId', function(req, res, next) {
54     res.status(200).json({
55         poruka: 'Obrisan je proizvod'
56     });
57 });
58
59 module.exports = router;

```

Klijentski deo rešenja ne bi trebalo da bude teško uraditi, te ćemo samo dati rešenje. Napomenimo da je neophodno da sve podatke koje šaljemo kao vrednost svojstva `data` prilikom korišćenja jQuery metoda `$.ajax` (osim kad šaljemo GET zahtev) potrebno serijalizovati korišćenjem funkcije `JSON.stringify` i postaviti vrednost `contentType` na `'application/json'`.

Kod 5.9: nodejs/rest1/klijent.html

```

1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta name="viewport" content="width=device-width, initial-scale=1.0">
6      <title>Testiranje rada servera sa proizvodima</title>
7
8      <link rel="stylesheet" type="text/css" href="bootstrap.css">
9  </head>
10 <body>
11     <div class="container">
12         <h1>Testiranje rada servera</h1>
13         <div class="row">
14             <div class="col-md-6 mt-3">
15                 <h3>GET /products[:productId]</h3>
16                 <form id="form1">
17                     <div class="form-group">
18                         <label for="productId1">Identifikator proizvoda:</label>
19                         <input type="text" id="productId1" name="productId1"
20                             class="form-control" title="Unesite identifikator
21                             proizvoda ili ostavite prazno da dohvatite sve
22                             proizvode">

```



```

23         </form>
24         <div id="rezultat1"></div>
25     </div>
26     <div class="col-md-6 mt-3">
27         <h3>POST /products</h3>
28         <form id="form2">
29             <div class="form-group">
30                 <label for="name">Naziv proizvoda:</label>
31                 <input type="text" id="name" name="name" class="form-
32                     control">
33             </div>
34             <div class="form-group">
35                 <label for="price">Cena proizvoda:</label>
36                 <input type="text" id="price" name="price" class="form-
37                     control">
38             </div>
39             <input type="submit" value="Posalji podatke" class="btn btn
40                 -primary">
41         </form>
42         <div id="rezultat2"></div>
43     </div>
44     <div class="row">
45         <div class="col-md-6 mt-3">
46             <h3>PATCH /products:productId</h3>
47             <form id="form3">
48                 <div class="form-group">
49                     <label for="productId3">Identifikator proizvoda:</label>
50                     <input type="text" id="productId3" name="productId3"
51                         class="form-control">
52                 </div>
53                 <input type="submit" value="Posalji podatke" class="btn btn
54                     -primary">
55             </form>
56             <div id="rezultat3"></div>
57         </div>
58         <div class="col-md-6 mt-3">
59             <h3>DELETE /products:productId</h3>
60             <form id="form4">
61                 <div class="form-group">
62                     <label for="productId4">Identifikator proizvoda:</label>
63                     <input type="text" id="productId4" name="productId4"
64                         class="form-control">
65                 </div>
66                 <input type="submit" value="Posalji podatke" class="btn btn
67                     -primary">
68             </form>
69             <div id="rezultat4"></div>
70         </div>
71     </div>
72     <script type="text/javascript" src="jquery-3.3.1.min.js"></script>
73     <script type="text/javascript" src="klijent.js"></script>
74 </body>
</html>

```

Kod 5.10: nodejs/rest1/klijent.js

```
1 $(document).ready(function() {
2     $("#form1").submit(function()
3     {
4         let productId = $("#productId").val();
5
6         if (productId === "")
7         {
8             $.ajax("http://localhost:3000/products", {
9                 method: "GET",
10                success: function(data) {
11                    if(data.hasOwnProperty("poruka")) {
12                        $("#rezultat1").text(data.poruka).css({color: "green"})
13                        ;
14                    }
15                },
16                error: function() {
17                    $("#rezultat1").text("Greska pri dohvatanju proizvoda!").
18                    css({color: "red"});
19                }
20            });
21        }
22        else
23        {
24            $.ajax(`http://localhost:3000/products/${productId}`, {
25                method: "GET",
26                success: function(data) {
27                    if(data.hasOwnProperty("poruka")) {
28                        $("#rezultat1").text(data.poruka).css({color: "green"})
29                        ;
30                    }
31                },
32                error: function() {
33                    $("#rezultat1").text("Greska pri dohvatanju proizvoda!").
34                    css({color: "red"});
35                }
36            });
37        }
38
39        $("#rezultat1").text("");
40        return false;
41    });
42
43    $("#form2").submit(function()
44    {
45        let name = $("#name").val();
46        let price = Number.parseFloat($("#price").val());
47
48        if (name === "")
49        {
50            $("#rezultat2").text("Niste prosledili naziv proizvoda").css({color
51                : "red"});
52            return false;
53        }
54
55        if (Number.isNaN(price))
56        {
57            $("#rezultat2").text("Niste prosledili ispravnu cenu proizvoda").
58            css({color: "red"});
59            return false;
60        }
61    });
62    });
```

```
55     }
56
57     $.ajax("http://localhost:3000/products", {
58         method: "POST",
59         data: JSON.stringify({
60             name, price
61         }),
62         contentType: 'application/json',
63         success: function(data) {
64             if(data.hasOwnProperty("kreiraniProizvod")) {
65                 $("#rezultat2")
66                     .text(`Kreirali ste proizvod:
67                     ${JSON.stringify(data.kreiraniProizvod)}`)
68                     .css({color: "green"});
69             }
70         },
71         error: function() {
72             $("#rezultat2").text("Greska pri postavljanju proizvoda!").css
73                 ({color: "red"});
74         }
75     });
76     return false;
77 });
78
79 $("#form3").submit(function()
80 {
81     let productId = $("#productId3").val();
82
83     if (productId === "")
84     {
85         $("#rezultat3").text("Niste prosledili identifikator proizvoda").
86             css({color: "red"});
87         return false;
88     }
89
90     $.ajax(`http://localhost:3000/products/${productId}`, {
91         method: "PATCH",
92         success: function(data) {
93             if(data.hasOwnProperty("poruka")) {
94                 $("#rezultat3").text(data.poruka).css({color: "green"});
95             }
96         },
97         error: function() {
98             $("#rezultat3").text("Greska pri menjanju proizvoda!").css({
99                 color: "red"});
100         }
101     });
102
103     $("#rezultat3").text("");
104     return false;
105 });
106
107 $("#form4").submit(function()
108 {
109     let productId = $("#productId4").val();
110
111     if (productId === "")
112     {
113         $("#rezultat4").text("Niste prosledili identifikator proizvoda").
114             css({color: "red"});
```

```
112         return false;
113     }
114
115     $.ajax(`http://localhost:3000/products/${productId}`, {
116         method: "DELETE",
117         success: function(data) {
118             if(data.hasOwnProperty("poruka")) {
119                 $("#rezultat4").text(data.poruka).css({color: "green"});
120             }
121         },
122         error: function() {
123             $("#rezultat4").text("Greska pri brisanju proizvoda!").css({
124                 color: "red"});
125         });
126
127     $("#rezultat4").text("");
128     return false;
129 });
130 });
```

Literatura za ovu oblast

[Fou] Node.js Foundation. *Node.js*. URL: <https://nodejs.org/en/>.

6. Baza podataka MongoDB

7. Mongoose ORM



Programiranje klijentskih aplikacija

8	Radni okvir Angular	243
8.1	O verzionisanju	
8.2	Angular CLI i kreiranje novog projekta	
8.3	Osnovni pogled na Angular aplikacije — podizanje aplikacije	
8.4	Komponente i vezivanje podataka	
8.5	Ugrađene Angular direktive	
8.6	Komponente i vezivanje podataka — napredniji koncepti	
8.7	Kreiranje direktiva	
8.8	Mehanizam servisa i ubrizgavanje zavisnosti	
8.9	Rutiranje	
8.10	Filteri	
8.11	Rad sa formularima	
8.12	HTTP komunikacija u Angular aplikacijama	
	Literatura za ovu oblast	

8. Radni okvir Angular

U ovom poglavlju ćemo započeti upoznavanje sa Angular razvojnim okruženjem. Angular predstavlja naprednu biblioteku za kreiranje klijentskih aplikacija, ali je i više od toga — on nam nudi različite elemente za: struktuiranje i stilizovanje aplikacije (komponente, direktive), centralizaciju ili decentralizaciju odgovornosti delova aplikacije (servisi), rad sa formularima, navigaciju kroz aplikaciju (rutiranje), slanje asinhronih HTTP zahteva ka serverskim aplikacijama i još mnogo drugih.

8.1 O verzionisanju

Kada govorimo o razvojnom okruženju koje nosi naziv "Angular", neophodno je da napravimo razliku između prvobitne verzije ovog okruženja, koje se zapravo naziva "AngularJS" ili "Angular 1" (<https://angularjs.org/>), i verzije o kojoj će biti reči u ovom tekstu, a to je "Angular 2", odnosno, neke od narednih verzija "Angular 4", "Angular 5", "Angular 6", "Angular 7", i poslednja verzija u toku pisanja ovog teksta "Angular 8" — jednim imenom "Angular 2+" ili jednostavno "Angular" (<https://angular.io/>). Ukoliko bismo uporedili AngularJS i Angular, videli bismo da postoje razlike između ove dve "verzije" koje su toliko velike da se one mogu smatrati kao dva apsolutno različita razvojna okruženja! Koncepti, imenovanja, tok razvoja, pa čak i programski jezik koje ove dve "verzije" koriste su značajno drugačiji. Zbog toga, dobro je razumeti da postoji razlika između AngularJS i Angular razvojnih okruženja, kao i činjenica da poznavanje jednog okruženja neće nam pomoći da razumemo drugi.

Sa izuzetkom verzije "Angular 3" koja nije javno objavljena, tim koji razvija Angular svakih 6 meseci objavljuje novu verziju ovog razvojnog okruženja. Ove verzije ne predstavljaju fundamentalne promene, već više unapređenja ili uvođenje boljih elemenata za lakši i brži razvoj Angular aplikacija. Svakim objavljivanjem naredne verzije, tim postavlja i neophodne korake koje je potrebno uraditi da bi se projekat koji je razvijen u prethodnoj verziji mogao "prevesti" na narednu verziju i ove promene uglavnom nisu previše zahtevne. Ipak, gotovo svi elementi koji su uvedeni verzijom Angular 2 se i danas koriste, te će većina

koda koja je razvijena u starijim verzijama raditi i danas.

8.2 Angular CLI i kreiranje novog projekta

Angular razvojno okruženje obezbeđuje alat koji se naziva Angular CLI za kreiranje projekata i njihovo upravljanje iz komandne linije. Alat može da se koristi za automatizaciju procesa poput kreiranja projekata, dodavanje novih komponenti, pokretanje servera za lokalno podizanje aplikacije, objavljivanje aplikacije i dr. Dobra je ideja koristiti Angular CLI jer će nam pomoći u kreiranju i održavanju uobičajenih šablonskih operacija tokom razvoja naše aplikacije. Da bismo instalirali Angular CLI, potrebno je pokrenuti narednu komandu:

```
$ npm install -g @angular/cli@latest
```

Jednom kada je instaliran, možemo ga pokrenuti iz komandne linije pomoću komande `ng`. Testiranje instalacije se može izvršiti naredbom

```
$ ng --version
```

kojom će se ispisati trenutna verzija.

Da bismo kreirali novi projekat, koristićemo komandu `ng new <ime projekta>`, na primer:

```
$ ng new angular-hello-world
```

Interaktivna konzola će nas prvo upitati nekoliko pitanja u zavisnosti od verzije Angulara. Na primer, u verziji Angular 8, program nas prvo pita da li želimo da koristimo rutiranje (ukucati `y` i pritisnuti `ENTER` za potvrdu, odnosno samo `ENTER` za odbijanje), a zatim će nas pitati za format stilova (pritisnuti `ENTER` da bi se odabrao podrazumevani CSS). Pogledajmo šta je ova komanda kreirala za nas:

```
$ cd angular-hello-world
$ tree -F -L 1
.
|-- README.md           // Koristan README
|-- angular.json        // Konfiguraciona datoteka za angular-cli
|-- e2e/                // "end-to-end" testovi
|-- node_modules/
|-- package-lock.json
|-- package.json
|-- src/                // Kod nase aplikacije
|-- tsconfig.json       // Konfiguraciona datoteka za TypeScript
`-- tslint.json         // Konfiguraciona datoteka za TSLint
```

Trenutno nas interesuje samo direktorijum `src` jer ćemo tu stavljati kod naše aplikacije:

```
$ cd src
$ tree -F -L 1
.
|-- favicon.ico         // Ikona stranice
|-- index.html          // Pocetna (i jedina) HTML stranica koja se ucitava
|-- app/               // Izvorni kod za elemente nase aplikacije
|-- main.ts            // Ulazna datoteka za TypeScript
|-- styles.css         // Globalni stilovi
|-- environments/      // Podesavanja za razne varijante aplikacije
|-- test.ts            // Glavna datoteka za testiranje
|-- assets/            // Dodatni resursi: slike, datoteke, ...
`-- polyfills.ts       // Glavna datoteka za polifile
```

Za sada je neophodno da obratimo pažnju na narednih pet datoteka/direktorijuma:

- Datoteka `index.html` predstavlja osnovnu i jedinu HTML stranicu od koje se naša Angular aplikacija sastoji. Sve Angular aplikacije predstavljaju tzv. *jednostranične aplikacije* (engl. *single-page application*, skr. *SPA*), odnosno, aplikacija učitava od servera samo jednu HTML stranicu, a svi ostali zahtevi ka drugim "stranicama" (odnosno, delovima aplikacije koje korisniku u veb pregledaču izgledaju kao stranice) se implementiraju tako što se u pozadini koristi JavaScript kod koji menja DOM te jedne stranice. Ovu datoteku uglavnom nema potrebe menjati.
- Direktorijum `app/` sadrži izvorni kod za sve elemente koji čine našu jednostraničnu aplikaciju. Osim strukturnih elemenata koji su predstavljeni komponentama (o čemu će biti reči kasnije), u ovom direktorijumu se sadrže i drugi elementi koji na neki način definišu rad naše aplikacije — menjaju stil elemenata, dohvataju podatke, obrađuju podatke, filtriraju sadržaj i dr.
- Datoteka `main.ts` predstavlja ulaznu datoteku u programerskom smislu — ovo je prva datoteka čiji se kod izvršava i koja je zaslužna za *podizanje* (engl. *bootstrapping*) celokupne aplikacije. Ovu datoteku uglavnom nema potrebe menjati.
- Datoteka `styles.css` sadrži definicije CSS stilova koji važe na nivou celokupne aplikacije. Ovu datoteku možemo menjati ukoliko želimo da definišemo stilove na globalnom nivou, mada, i to je retka operacija. Uglavnom želimo da definišemo stilove na nivou pojedinačnih elemenata.
- Direktorijum `assets/` sadrži razne veb resurse koji su neophodni za ispravno prikazivanje aplikacije, na primer, slike, zvučni zapisi, video zapisi, razne druge datoteke i dr. Prilikom objavljivanja aplikacije, Angular CLI će automatski zapakovati i ovaj direktorijum, zajedno sa drugim potrebnim resursima.

Pokretanje Angular aplikacije se može izvršiti pozivom komande:

```
$ ng serve
```

čime će se podići aplikacija na lokalnom serveru, konkretnije, na adresi `http://localhost:4200` (podrazumevano). Svaki novokreirani Angular projekat dolazi sa nekim već podešenim elementima, sa kojima ćemo se upoznati detaljno u ostatku teksta. Za početak, ukoliko otvorimo veb pregledač na datoj veb adresi, prikazaće nam se prozor kao na narednoj slici:



Slika 8.1: Prikaz aplikacije `angular-hello-world`.

8.3 Osnovni pogled na Angular aplikacije — podizanje aplikacije

Sada prelazimo na nešto detaljnije analiziranje aplikacije `angular-hello-world` koju smo kreirali u prethodnoj sekciji. Cilj ove sekcije je da razumemo na koji način Angular kreira prikaz stranice sa slike 8.1.

Podizanje aplikacije (engl. *bootstrapping*) predstavlja proces sakupljanja svih potrebnih datoteka izvornog koda, konfiguracionih datoteka i resursnih datoteka, zatim procesiranje tih datoteka i kreiranje finalnih datoteka koji čine projekat koji je spreman za izvršavanje. Podizanje Angular aplikacija, između ostalog, podrazumeva i naredne operacije: transpiliranje TypeScript koda u JavaScript kod, minimizaciju i optimizaciju koda, umetanje datoteka koje su nastale ovim procesom u druge datoteke i sl. Sve ove, i mnoge druge operacije, Angular CLI uradi umesto nas, što nam olakšava razvoj aplikacija. Dodatno, prilikom pokretanja servera komandom `ng serve` radi serviranja završnih datoteka, Angular CLI prati izmene u izvornim datotekama i svaki put kada napravimo i sačuvamo izmene, alat ponovo započinje proces podizanja kako bismo u veb pregledaču uvek imali najsvežiju verziju aplikacije. Naravno, ovo je moguće sve dok je proces pokrenut komandom `ng serve` aktivan — jednom kada se proces završi (pritiskom `CTRL + C`), razvojni server se gasi i ne možemo da pristupimo aplikaciji. Hajde da pogledamo kako zapravo Angular vrši ovaj proces.

Rekli smo da je datoteka `main.ts` ta koja je zadužena za podizanje aplikacije. Ako pogledamo njen izvorni kod, pored učitavanja raznih modula i dodatnih naredbi u zavisnosti od varijanti aplikacije, pronaći ćemo i naredne linije koda:

```
1 platformBrowserDynamic().bootstrapModule(AppModule)
2 .catch(err => console.error(err));
```

Očigledno, druga linija služi za hvatanje i prikazivanje izuzetaka, ali više nas interesuje prva linija. Ta linija govori Angular-u koji je to modul naše aplikacije od kojeg je potrebno da se započne podizanje aplikacije, tj. koji modul je *koreni* (engl. *root*). Podrazumevano, aplikacija ima tačno jedan modul — i naše aplikacije koje budemo kreirali će imati tačno jedan modul — ali je moguće razbiti aplikaciju na više modula.

Takođe, podrazumevano se za koreni modul kreira modul `AppModule`, što je sasvim korektan naziv. Primetimo imenovanje ovog modula — naziv modula `App`, praćen rečju `Module` — ovo je konvencija koja se koristi i za mnoge druge elemente o kojima ćemo diskutovati. Naravno, ovakvo imenovanje nije obavezno, ali nije loše držati se konvencije koju razvojni tim Angular-a podržava.

Odakle aplikacija zna za definiciju ovog modula? Ako pogledamo u istoj datoteci, pronaći ćemo narednu liniju:

```
1 import { AppModule } from './app/app.module';
```

Otvaranjem datoteke koja se nalazi na putanji `./app/app.module.ts`, relativnoj u odnosu na direktorijum u kojoj se nalazi `main.ts` datoteka, možemo videti da modul ne predstavlja ništa drugo do (izvezenu) klasu `AppModule`, koja je dekorisana dekoratorom `@NgModule`. U pitanju je dekorator koji je dostupan uz Angular i njime se definišu naredni ključni elementi modula:

- Deklaracije (svojstvo `declarations`) — Reference na klase komponenti (i još nekih drugih elemenata koje ćemo videti) koje ulaze u strukturni sastav naše aplikacije. Svaki put kada kreiramo novu komponentu, potrebno je da njenu referencu smestimo u ovaj niz.

- Uvoženja (svojstvo `imports`) — Reference na Angular module od kojih zavisi modul koji se dekorira (u ovom slučaju `AppModule`). Podrazumevano se uvozi modul `BrowserModule` koji predstavlja jedan od najosnovnijih Angular modula za prikazivanje aplikacije u veb pregledaču.
- Dobavljači (svojstvo `providers`) — Reference na klase servisa (i još nekih drugih elemenata koje ćemo videti) koje ulaze u funkcionalni sastav naše aplikacije. Svaki put kada kreiramo nov servis, potrebno je da njegovu referencu smestimo u ovaj niz.
- Podizanja (svojstvo `bootstrap`) — Reference na module aplikacije koje su neophodne za početno podizanje aplikacije. U najvećem broju slučajeva neće biti potrebe za menjanjem ovog svojstva.

Daljom analizom ove datoteke možemo primetiti da postoji još jedan element koji nije deo jezgra Angular razvojnog okruženja, a to je `AppComponent`, čija referenca stoji u nizu deklaracija i koji je uvezen linijom:

```
1 import { AppComponent } from './app.component';
```

Otvaranjem datoteke `./app.component.ts`, koja se nalazi u istom direktorijumu, primećujemo da ponovo imamo klasu (koja se izvozi), samo što je ovoga puta drugog naziva (ali prati istu konvenciju — naziv komponente + reč **Component**) i ono što je zanimljivije — dekorisanu drugim dekoratorom, **@Component**, koji je takođe dostupan uz Angular. Iako ćemo detaljnije pričati o komponentama kasnije, pogledajmo neke osnovne elemente koje vidimo iz ove datoteke:

- Svojstvo `selector` ima postavljenu vrednost `'app-root'`. Naziv ovog svojstva naš možda asocira na CSS selektore, te ako se prisetimo na koje HTML elemente bi ovaj selektor mogao da se primeni, zaključili bismo da je u pitanju HTML element `<app-root>`. Još uvek nam ovo ništa ne govori, ali zapamtimo ovu činjenicu.
- Naredno svojstvo je `templateUrl` koje deluje da "pokazuje" na datoteku `./app.component.html` koja se nalazi u tekućem direktorijumu. Otvaranjem te datoteke, vidimo naredni sadržaj:

Kod 8.1: angular/angular-hello-world/src/app/app.component.html

```

1 <!--The content below is only a placeholder and can be replaced.-->
2 <div style="text-align:center">
3   <h1>
4     Welcome to {{ title }}!
5   </h1>
6   
7 </div>
8 <h2>Here are some links to help you start:</h2>
9 <ul>
10   <li>
11     <h2><a target="_blank" rel="noopener" href="https://angular.io/
        tutorial">Tour of Heroes</a></h2>
12   </li>
13   <li>
14     <h2><a target="_blank" rel="noopener" href="https://angular.io/cli">
        CLI Documentation</a></h2>
15   </li>

```



```

16   <li>
17     <h2><a target="_blank" rel="noopener" href="https://blog.angular.io/"
      >Angular blog</a></h2>
18   </li>
19 </ul>

```

Primećujemo da se ovaj HTML sadržaj sastoji od poruke `Welcome to {{ title }}`!, koji prati nekakva slika, pa zatim još teksta i lista. Ako se prisetimo, ovo je poprilično ličilo na sadržaj koji smo imali na prikazu 8.1. Da li to znači da taj sadržaj koji smo imali prikazan dolazi na osnovu ovog sadržaja? Odgovor je potvrđan. Kako se ta veza tačno ostvarila videćemo uskoro.

- Vratimo se na datoteku `./app.component.html`. Dalje, primećujemo svojstvo `styleUrls` čija je vrednost niz, čiji su elementi, po svemu sudeći opet putanje. Otvaranjem datoteke na jedinoj putanji u nizu `./app.component.css`, vidimo praznu CSS datoteku. Ako se prisetimo, videli smo da se u korenom direktorijumu izvornog koda aplikacije (tj. u direktorijumu `app/`) nalazi `styles.css` datoteka za koju smo napomenuli da sadrži globalne stilove koji važe na nivou cele aplikacije. Za razliku od te datoteke, stilovi koji se definišu u okviru datoteke `./app.component.css` važe isključivo za elemente koji ulaze u sastav komponente `AppComponent`, koja referiše na ovu datoteku putem svojstva `styleUrls`.
- Konačno, u okviru definicije klase `AppComponent` pronalazimo liniju:

```
1 title = 'angular-hello-world';
```

Gde smo imali prilike da vidimo promenljivu `title`? Ako se ponovo prisetimo, u okviru datoteke `./app.component.html` stajao je sadržaj `Welcome to {{ title }}`!, a na prikazu 8.1 vidimo sadržaj `Welcome to angular-hello-world!`. Odavde možemo zaključiti da Angular ima sposobnost da na neki način koristi podatke iz klase `AppComponent` da bi ih dinamički prikazao u HTML šablonu. Naravno, ovo je samo delić njegove moći.

Ostala je još samo jedna karika u ovom lancu koja povezuje sve. Kako Angular zna da treba da prikaže HTML sadržaj komponente `AppComponent` na stranici? Odgovor leži u `index.html` datoteci:

Kod 8.2: `angular/angular-hello-world/src/index.html`

```

1 <!doctype html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>AngularHelloWorld</title>
6   <base href="/">
7
8   <meta name="viewport" content="width=device-width, initial-scale=1">
9   <link rel="icon" type="image/x-icon" href="favicon.ico">
10 </head>
11 <body>
12   <app-root></app-root>
13 </body>
14 </html>

```

Ova jednostavna datoteka predstavlja osnovnu i jedinu HTML stranicu za koju veb pregledač kreira HTTP zahtev ka razvojnom serveru koji smo pokrenuli komandom `ng serve`. U okviru tela ove HTML datoteke vidimo da se prikazuje jedino element `<app-root>`. Međutim, ako se ponovo (i po poslednji put u ovoj sekciji) prisetimo, videli smo da je ovaj element definisan u dekoratoru `@Component` klase `AppComponent` i to svojstvom `selector`.

Dakle, Angular na osnovu ove datoteke pronalazi koja komponenta deklarise element `<app-root>` i prikazuje njen HTML šablon, zajedno sa eventualnim dinamičkim izmenama koje transpilirani JavaScript kod izvršava.

Međutim, u ovoj datoteci ne postoje reference ni na jednu JavaScript datoteku. Koji je to JavaScript kod koji se izvršava? Angular, takođe, prilikom podizanja aplikacije, nakon što izvrši sve neophodne operacije nad izvornim kodom (transpiliranje, optimizacije, itd.), on umeće završne datoteke u `index.html` datoteku. Ovo možemo da vidimo ako ispitamo izvorni kod u veb pregledaču, koji je prikazan na slici 8.2.

```
1 <!doctype html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>AngularHelloWorld</title>
6   <base href="/">
7
8   <meta name="viewport" content="width=device-width, initial-scale=1">
9   <link rel="icon" type="image/x-icon" href="favicon.ico">
10 </head>
11 <body>
12   <app-root></app-root>
13 <script src="runtime.js"></script><script src="polyfills-es5.js" nomodule></script><script src="polyfills.js">
14 </script><script src="styles.js"></script><script src="vendor.js"></script><script src="main.js"></script></body>
15 </html>
```

Slika 8.2: Procesirani izvorni kod datoteke `index.html` aplikacije `angular-hello-world`.

Naravno, nakon izvršavanja JavaScript koda, DOM stablo stranice izgleda kao na slici 8.3, što definitivno bolje oslikava sadržaj koji se prikazuje:

[illegible]

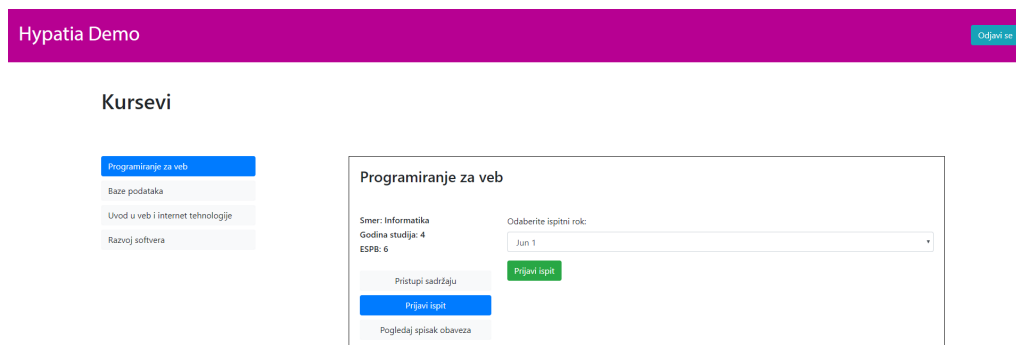
Slika 8.3: DOM stablo aplikacije angular-hello-world.

Ovim smo završili uvodnu diskusiju o tome kako Angular razvojno okruženje dolazi od izvornog koda do finalnog prikaza. U daljem tekstu se bavimo elementima Angulara kojima možemo kreirati najrazličitije bogate klijentske aplikacije.

8.3.1 Uključivanje Bootstrap 4 biblioteke za stilizovanje elemenata

8.4 Komponente i vezivanje podataka

Način na koji se Angular aplikacije razvijaju jeste kreiranjem *komponenti* (engl. *component*). Komponente možemo razumeti kao gradivne elemente naše veb aplikacije. Praktično, cela aplikacija se može podeliti na veliki broj komponenti, koje se najčešće mogu predstaviti hijerarhijski. Prikažimo ovo kroz aplikaciju koja je data na slici 8.4.



Slika 8.4: Primer jedne aplikacije za koju je potrebno izvršiti podelu na komponente.

Ova naizgled jednostavna aplikacija se može podeliti u različite delove. Na slici 8.5 prikazane su dve moguće podele (koje nisu jedine). Prva podela deli aplikaciju na manji broj komponenti koje imaju veliki broj odgovornosti, dok druga podela deli aplikaciju na veliki broj komponenti, ali koje predstavljaju manje logičke celine. Slika 8.6 sadrži nabrojane i imenovane komponente, kao i grafik njihovih hijerarhijskih odnosa za prvu podelu, dok slika 8.7 sadrži iste informacije, samo za drugu podelu.

Podela aplikacije na komponenti igra ključnu ulogu u razvoju Angular aplikacija, te ju je zbog toga moguće izvršiti na razne načine. Najčešće, ukoliko je aplikacija jednostavnija, onda je sasvim korektno i imati manji broj komponenti, dok je kod iole složenijih aplikacija podela na komponente koje imaju manje posla isplativija.

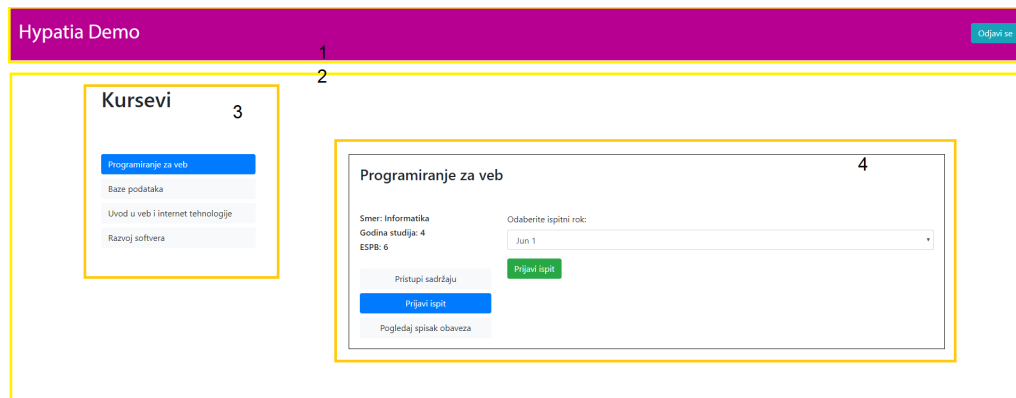
8.4.1 Kreiranje komponenti

Postoje dva načina za kreiranje komponenti: "ručno" i korišćenjem Angular CLI alata. Preporučujemo da prilikom učenja koristite ručni metod, da biste razumeli koji su to elementi koji su važni za kreiranje komponenti, a zatim, kada razumete detalje kreiranja komponenti, koristite Angular CLI alat.

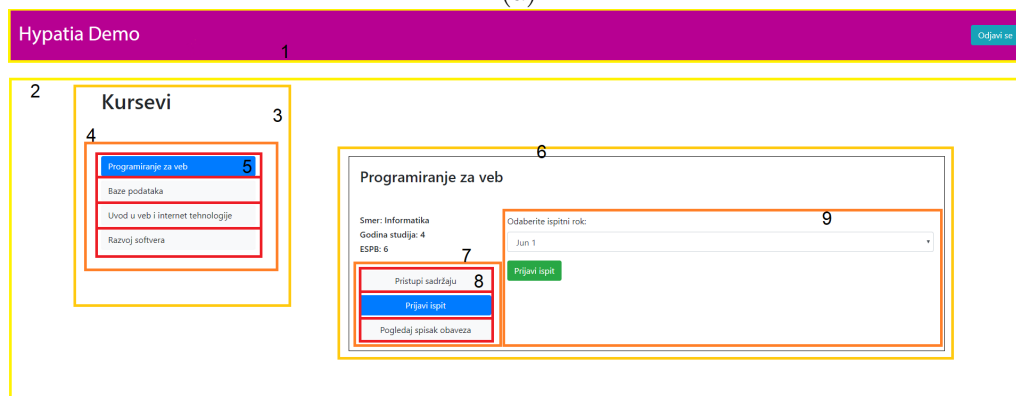
Ručno kreiranje komponenti

Kreirajmo sada komponentu koja će predstavljati jednog studenta. Kao što smo videli ranije, komponente su sačinjene od raznih datoteka. Na primer, pri kreiranju korene komponente `AppComponent`, Angular CLI alat je za nas kreirao odgovarajuće datoteke `app.component.html`, `app.component.css`, `app.component.ts` i `app.component.spec.ts`. Zbog toga, dobra je praksa čuvati svaku komponentu u zasebnom direktorijumu.

Kreirajmo direktorijum `app/student/` i u njemu kreirajmo za sada samo `student.component.ts` datoteku. Obratimo pažnju i na imenovanje ovih datoteka. Sve datoteke koje ulaze u sastav neke komponente c imaju naziv oblika `c.component` praćen odgovarajućom ekstenzijom (`html`, `css`, `ts` ili `spec.ts`). Otvorimo kreiranu datoteku u tekstualnom editoru.



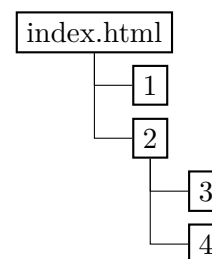
(a)



(b)

Slika 8.5: Podela aplikacije na komponente. Slika (a) ilustruje jednostavniju podelu, dok slika (b) prikazuje složeniju podelu.

1. Zaglavlje stranice
2. Glavni deo
3. Odabir kursa
4. Upravljanje informacijama o kursu



Slika 8.6: Elementi prve podele aplikacije sa slike 8.5 (levo) i njihova hijerarhijska reprezentacija (desno).

Komponente se u Angular-u predstavljaju TypeScript klasama:

```

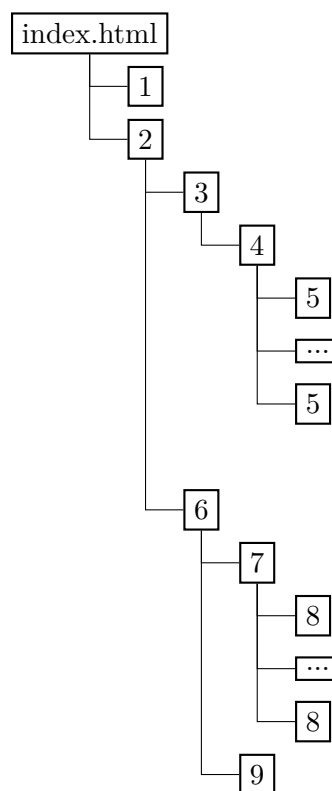
1 export class StudentComponent {
2 }

```

Međutim, da bi Angular znao da ova klasa predstavlja komponentu, potrebno je uraditi dve stvari:

1. Dekorirati klasu dekoratorom `@Component`.
2. Registrovati klasu kao komponentu u `declarations` opciji dekoratora `@NgModule` u modulu u kojem se komponenta koristi.

1. Zaglavlje stranice
2. Glavni deo
3. Upravljanje kursevima
4. Lista kurseva
5. Odabir kursa
6. Upravljanje informacijama o kursu
7. Lista opcija
8. Odabir opcije
9. Detalji opcije



Slika 8.7: Elementi druge podele aplikacije sa slike 8.5 (levo) i njihova hijerarhijska reprezentacija (desno).

Za početak, uključimo potrebni dekorator na vrhu kreirane datoteke, a zatim ga iskoristimo za dekorisanje napisane klase:

```

1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-student',
5   template: '<h3>Ja sam student</h3>',
6   styles: [
7     h3 {
8       color: blue;
9     }
10  ],
11 })
12 export class StudentComponent {
13 }

```

Kao što vidimo, dekorator `@Component` prihvata kao argument objekat sa određenim svojstvima:

- Svojstvo `selector` predstavlja CSS selektor kojim se komponenta referencira u HTML kodu. Najčešći način za selektovanje komponenti jeste preko HTML elemenata. U kodu iznad, ova komponenta će biti korišćena svaki put kada se u HTML kodu koristi element `<app-student></app-student>`. Vrednost ovog svojstva mora biti jedinstveno za sve komponente u aplikaciji.
- Svojstvo `template` predstavlja nisku koja definiše HTML strukturu komponente. Ovaj "kod" će biti iskorišćen na mestu gde je komponenta referencirana. To mo-

že biti jednostavan HTML obeleženi tekst, kao u kodu iznad, ali može imati i neke složenije konstrukte sa kojima ćemo se upoznati kroz različite sekcije ovog poglavlja.

- Svojstvo `styles` predstavlja niz niski koje definišu CSS stilove koji će biti primenjeni nad ovom komponentom. Ovi stilovi su lokalni za ovu komponentu i neće imati uticaja na HTML elemente van ove komponente. U kodu iznad definišemo stil koji boji element `<h3>` u plavu boju. Ovo svojstvo je opciono.

Svojstva `template` i `styles` imaju i svoje alternativne oblike, koji su dati u narednom kodu:

```
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-student',
5   templateUrl: 'student.component.html',
6   styleUrls: ['student.component.css']
7 })
8 export class StudentComponent {
9 }
```

Njihova značenja su sledeća:

- Svojstvo `templateUrl` predstavlja nisku koja sadrži putanju do datoteke koja sadrži HTML strukturu komponente. U kodu iznad je specifikovano da datoteka `student.component.html` sadrži ovu strukturu, što bi značilo da, ukoliko želimo da očuvamo istu strukturu, potrebno je da datoteka `student.component.html` bude sledeće sadržine:

Kod 8.3: angular/komponente1/src/app/student/student.component.html

```
1 <h3>Ja sam student</h3>
```

- Svojstvo `styleUrls` predstavlja niz niski od koje svaka sadrži putanju do datoteke koja sadrži definicije stilova koji će biti primenjeni na komponentu. U kodu iznad je specifikovano da datoteka `student.component.css` sadrži stilove, što bi značilo da, ukoliko želimo da očuvamo istu vizualnu prezentaciju komponente, potrebno je da datoteka `student.component.css` bude sledeće sadržine:

Kod 8.4: angular/komponente1/src/app/student/student.component.css

```
1 h3 {
2   color: blue;
3 }
```

Kao i svojstvo `styles`, i svojstvo `styleUrls` je opciono.

Ono što bi trebalo zapamtiti jeste da je pogrešno specifikovati obe varijante za jednu opciju. Tako, na primer, komponenta će imati: (1) ili `template` ili `templateUrl` (primetimo ekskluzivnu disjunkciju ovde) i (2) `styles` ili `styleUrls` (primetimo "običnu" disjunkciju ovde).

Zbog toga što drugi pristup (korišćenje `templateUrl` i `styleUrls` svojstava) dovodi do lakše proširivosti komponenti, mi ćemo koristiti upravo taj pristup.

Naravno, ne smemo zaboraviti da dodamo zavisnost između novokreirane komponente i korenog modula aplikacije. U datoteci `app.module.ts`, potrebno je uvesti klasu `StudentComponent` i dodati je u niz `declarations` (u kodu su ove dve linije označene `// !`):

```

1 import { BrowserModule } from '@angular/platform-browser';
2 import { NgModule } from '@angular/core';
3
4 import { AppComponent } from './app.component';
5 import { StudentComponent } from './student/student.component'; // !
6
7 @NgModule({
8   declarations: [
9     AppComponent,
10    StudentComponent // !
11  ],
12   imports: [
13     BrowserModule
14  ],
15   providers: [],
16   bootstrap: [AppComponent]
17 })
18 export class AppModule { }

```

Korišćenje Angular CLI alata

Prikažimo i drugi metod za kreiranje komponenti — korišćenjem Angular CLI alata. Recimo da želimo da napravimo komponentu koja se zove `Student2Component`. Onda je potrebno u terminalu, nakon što se pozicioniramo u direktorijum `app/`, da izvršimo narednu komandu:

```
$ ng generate component student2
```

Ili, alternativno, možemo koristiti skraćenu verziju:

```
$ ng g c student2
```

Pogledajmo šta je ova naredba ispisala na standardni izlaz:

```

CREATE src/app/student2/student2.component.html (27 bytes)
CREATE src/app/student2/student2.component.spec.ts (642 bytes)
CREATE src/app/student2/student2.component.ts (277 bytes)
CREATE src/app/student2/student2.component.css (0 bytes)
UPDATE src/app/app.module.ts (490 bytes)

```

Kao što vidimo, naredba je kreirala direktorijum `student2` za komponentu `Student2Component`, u koji je smestila novokreirane datoteke, ispoštovajući imenovanje `naziv.component.*`. Ukoliko pogledamo sadržaje ovih datoteka, videćemo da je Angular CLI alat uradio sav početni posao za nas — kreirao klasu `Student2Component`, dekorisao je, kreirao početne HTML i CSS datoteke, a takođe je i izmenio `app.module.ts` datoteku da uključi ovu klasu u niz `declarations`. Prikazujemo sadržaj ovih datoteka da bismo se uverili (preskačemo datoteku `student2.component.spec.ts` jer nam ona nije važna za sada):

Kod 8.5: `angular/komponente1/src/app/student2/student2.component.html`

```

1 <p>
2   student2 works!
3 </p>

```

Kod 8.6: `angular/komponente1/src/app/student2/student2.component.css`

Primetimo da je datoteka `student2.component.css` prazna jer, podrazumevano, Angular ne definiše nikakve stilove za naše komponente.

Kod 8.7: angular/komponente1/src/app/student2/student2.component.ts

```

1 import { Component, OnInit } from '@angular/core';
2
3 @Component({
4   selector: 'app-student2',
5   templateUrl: './student2.component.html',
6   styleUrls: ['./student2.component.css']
7 })
8 export class Student2Component implements OnInit {
9
10   constructor() { }
11
12   ngOnInit() {
13   }
14
15 }
```

Kod 8.8: angular/komponente1/src/app/app.module.ts

```

1 import { BrowserModule } from '@angular/platform-browser';
2 import { NgModule } from '@angular/core';
3 import { FormsModule } from '@angular/forms';
4
5 import { AppComponent } from './app.component';
6 import { StudentComponent } from './student/student.component';
7 import { Student2Component } from './student2/student2.component';
8 import { Student1ListaComponent } from './student1-lista/student1-lista.
   component';
9 import { Student3Component } from './student3/student3.component';
10 import { Student3ListaComponent } from './student3-lista/student3-lista.
   component';
11 import { Student4Component } from './student4/student4.component';
12 import { Student4ListaComponent } from './student4-lista/student4-lista.
   component';
13 import { Student5Component } from './student5/student5.component';
14 import { Student5ListaComponent } from './student5-lista/student5-lista.
   component';
15 import { Student6Component } from './student6/student6.component';
16 import { Student6ListaComponent } from './student6-lista/student6-lista.
   component';
17 import { Student7Component } from './student7/student7.component';
18 import { Student7ListaComponent } from './student7-lista/student7-lista.
   component';
19
20 @NgModule({
21   declarations: [
22     AppComponent,
23     StudentComponent,
24     Student2Component,
25     Student1ListaComponent,
26     Student3Component,
27     Student3ListaComponent,
28     Student4Component,
29     Student4ListaComponent,
30     Student5Component,
31     Student5ListaComponent,
32     Student6Component,
33     Student6ListaComponent,
34     Student7Component,
35     Student7ListaComponent
36 ],
```

```

37   imports: [
38     BrowserModule,
39     FormsModule
40   ],
41   providers: [],
42   bootstrap: [AppComponent]
43 })
44 export class AppModule { }

```

Da bismo videli da naše komponente zaista funkcionišu, potrebno je da ih iskoristimo u HTML šablonu naše korene komponente `AppComponent`:

```

1 <app-student></app-student>
2 <app-student2></app-student2>

```

Pokrenite razvojni server i otvorite veb pregledač na adresi `http://localhost:4200` i uverite se da se prikazuju obe komponente.

Ono što nam je preostalo da vidimo jeste kako možemo da koristimo već kreirane komponente kao delove drugih komponenti. Na primer, kreirajmo komponentu koja će prikazivati nekoliko `StudentComponent` objekata:

```
$ ng g c student1-lista
```

Otvorimo `student1-lista.component.html` i popunimo narednim kodom:

Kod 8.9: `angular/komponente1/src/app/student1-lista/student1-lista.component.html`

```

1 <h3>Studenti</h3>
2
3 <app-student></app-student>
4 <app-student></app-student>
5 <app-student></app-student>

```

Takođe, izmenimo `app.component.html`, tako da uključi novokreiranu komponentu:

```

1 <h1>Komponente 1</h1>
2
3 <app-student1-lista></app-student1-lista>

```

Šta smo ovim uradili? U HTML šablonu korene komponente `AppComponent` referisali smo na `Student1ListaComponent` preko elementa `<app-student1-lista>`, što smo mogli da uradimo zato što je ta komponenta definisana selektorom `'app-student1-lista'`. Detaljnije ćemo govoriti o tome, ali napomenimo da je u pozadini Angular za nas instancirao objekat klase `Student1ListaComponent` i svaka izmena u toj klasi koja utiče na prikaz komponente, biće vidljiva u veb pregledaču u trenutku izvršavanja te izmene. Ovo ćemo imati prilike da vidimo na delu u podsekciji 8.4.2.

Dalje, u HTML šablonu ove komponente referisali smo na tri komponente `StudentComponent` preko elementa `<app-student>`. Slično kao i za komponentu `Student1ListaComponent`, Angular je u pozadini instancirao tri objekta klase `StudentComponent` i on će biti zadužen za sve eventualne izmene u toj klasi koje utiču na prikaz, ili obrnuto.

Zadatak 8.1 Provežbati kreiranje komponenti:

- Kreirati dve nove komponente: `WarningAlert` i `SuccessAlert`.
- Prikazati ih jednu ispod druge u `AppComponent`.

- Prikazati u prvoj komponenti neku upozoravajuću poruku, a u drugoj komponenti poruku o uspešnosti.
- Stilizovati komponente prikladno (na primer, `WarningAlert` može biti pravougaonik svetlo žute boje, a `SuccessAlert` može biti pravougaonik svetlo zelene boje).

Koristiti oba pristupa kreiranju komponenti. Koristiti oba pristupa kreiranju HTML šablona i stilizovanju komponenti. ■

8.4.2 Vezivanje podataka

U prethodnoj sekciji smo videli kako možemo da kreiramo nove komponente i eventualno ih ugnežđavamo i time kreiramo hijerarhijsku organizaciju između tih komponenti. Međutim, iako je ovo dovoljno za kreiranje statičkih stranica, zašto bismo uopšte koristili ovako složeno okruženje za razvoj za kreiranje statičkih stranica. Zbog toga ćemo se u daljem tekstu upoznavati sa različitim elementima Angulara koji omogućavaju upravljanje dinamičkim sadržajem. Za početak, upoznajmo se sa vezivanjem podataka.

Vezivanje podataka (engl. *data binding*) predstavlja mehanizam kojim se dinamički podaci koji se kreiraju u TypeScript-u koriste za prikazivanje u HTML šablonu, ali i obrnuto, kako izmene u HTML šablonu mogu uticati na menjanje podataka u TypeScript-u. Postoje četiri mehanizma za vezivanje podataka koji se mogu kategorisati u dva nivoa: usmerenost i kardinalnost. Prema usmerenosti, mehanizmi vezivanja podataka se dele na:

- U smeru od TypeScript-a ka HTML šablonu — Izmene u TypeScript objektima utiču na prikaz.
- U smeru od HTML šablona ka TypeScript-u — Izmene u HTML šablonu (na primer, tako što korisnik unese podatak u polje formulara) utiču na sadržaj TypeScript objekata.

Sa druge strane, prema kardinalnosti, mehanizmi vezivanja podataka se dele na:

- U jednom smeru — Mehanizam funkcioniše tako što samo izmene na jednoj strani vezivanja utiču na drugu stranu vezivanja, ali ne i obrnuto.
- U dva smera — Promena bilo na jednoj, bilo na drugoj strani vezivanja utiče na onu drugu stranu vezivanja.

Pre nego što pređemo na vrste vezivanja podataka, uvedimo termine *model* (engl. *model*) da označimo podatke koji se nalaze u memoriji računara (ili koji se mogu dovesti u memoriju računara, na primer, asinhronim HTTP zahtevom ka nekom serveru) i *pogled* (engl. *view*) da označimo prikaz komponente (ili generalno, aplikacije) u veb pregledaču kroz njen HTML šablon.

Interpolacija niski

Interpolacija niski (engl. *string interpolation*) predstavlja mehanizam kojim se podaci koji se nalaze u modelu izračunavaju i prikazuju u pogledu. Dakle, u pitanju je mehanizam u jednom smeru od TypeScript-a ka HTML šablonu.

Recimo da želimo da prikažemo naša tri studenta, ali tako da se prikaže redni broj. U tu svrhu kreirajmo komponentu `Student3Component` koja će predstavljati studenta sa rednim brojem i komponentu `Student3ListaComponent` koja će prikazivati tri ove komponente.

Jedno moguće rešenje jeste da se prilikom konstrukcije svakog `Student3Component` objekta izračuna njegov redni broj. Ovo možemo uraditi statičkim brojačem u istoj klasi, koji će inkrementirati vrednost prilikom svake konstrukcije:

Kod 8.10: angular/komponente1/src/app/student3/student3.component.ts

```

1 import { Component, OnInit } from '@angular/core';
2
3 @Component({
4   selector: 'app-student3',
5   templateUrl: './student3.component.html',
6   styleUrls: ['./student3.component.css']
7 })
8 export class Student3Component implements OnInit {
9
10   static brojac: number = 1;
11
12   redniBroj: number;
13   jmbgStr: string = null;
14
15   constructor() {
16     this.redniBroj = Student3Component.brojac;
17     ++Student3Component.brojac;
18   }
19
20   ngOnInit() {
21   }
22
23   jmbg(): string {
24     if (this.jmbgStr === null) {
25       this.jmbgStr = Array.apply(null, {length: 13})
26         .map(val => Math.floor(Math.random() * 10))
27         .join('');
28     }
29     return this.jmbgStr;
30   }
31
32 }

```

Sada, svaki objekat ima svoju vrednost `this.redniBroj` koja će čuvati njegov redni broj. Ovim smo rešili problem u modelu, ali se postavlja pitanje kako ćemo iskoristiti tu vrednost u pogledu.

Interpolacija niski nam rešava ovaj problem. U HTML šablonu komponente možemo koristiti konstrukt oblika `{{ /* izraz */ }}`, a vrednost koja se zadaje između dvostrukih vitičastih zagrada predstavlja jednolinijski TypeScript izraz koji ili izračunava nisku ili izračunava vrednost koja može da se implicitno konvertuje u nisku. Drugim rečima, možemo iskoristiti naredni HTML šablon:

Kod 8.11: angular/komponente1/src/app/student3/student3.component.html

```

1 <p>{{ redniBroj }}. Ja sam student! JMBG: {{ jmbg() }}</p>

```

Vrednost `redniBroj` unutar interpolacije niski predstavlja broj, koji očito može da se konvertuje u nisku. Angular će prilikom analize ovog šablona naići na interpolaciju niske, izračunati vrednost izraza između zagrada, konvertovati vrednost u nisku i, finalno, zameniti dobijenu vrednost umesto celog izraza u finalnom HTML prikazu. Sada bi čitaocu trebalo da bude jasno zašto smo sve vreme govorili "HTML šablon" umesto **HTML obelevzeni tekst** — u datoteci koja se zadaje kroz `templateUrl` svojstvo komponente mogu se naći neki specijalni konstrukti Angulara koji će biti procesirani i na osnovu kojih će biti dinamički generisan HTML obeležen tekst koji će činiti finalni pogled.

Da bismo videli da zaista bilo koji izraz koji može da izračunava nisku može da se iskoristi u

interpolaciji niske, napisali smo metod `jmbg()` koji izračunava nasumični JMBG, a zatim smo taj metod iskoristili u HTML šablonu za `Student3Component` — zaista, povratna vrednost tog metoda jeste niska čija je vrednost zamenjena na mestu interpolacije niske koja ju je pozvala.

Vezivanje atributa

Kao što znamo, HTML elementi imaju veliki broj atributa koji predstavljaju vrednosti kojima se definiše njihov prikaz ili ponašanje. Na adresi <https://developer.mozilla.org/en-US/docs/Web/API#Interfaces> nalazi se opširan spisak interfejsa, od kojih svaki sadrži spisak atributa kojima se može pristupiti. U primeru koji sledi ćemo koristiti dugme (interfejs `HTMLInputElement`) i njegov atribut `disabled`, ali, naravno, opšti koncepti važe i za druge elemente.

Vezivanje atributa (engl. *property binding*) predstavlja mehanizam kojim se koriste podaci iz modela za definisanje vrednosti atributa elemenata u pogledu. Kao i interpolacija niski, i ovaj mehanizam predstavlja jednosmerni mehanizam u smeru od TypeScript-a ka HTML šablonu.

Za ovaj primer ćemo kreirati komponente `Student4Component` i `Student4ListaComponent`, od kojih druga ponovo ima tri instance prve komponente u šablonu. Komponenti `Student4Component` dodajemo nova svojstva `ime` i `prezime`, a u šablonu koristimo interpolaciju niski za prikazivanje ovih vrednosti. Inicijalno, ove vrednosti su nedefinisane, te zbog toga neće biti prikazane u pogledu. Takođe, u modelu imamo i jednu Bulovu vrednost, inicijalno postavljenu na `false`, koja označava da li je dugme u komponenti uključeno ili isključeno. Kao što vidimo u narednom kodu, dve sekunde nakon što se komponenta kreira, to dugme će biti uključeno:

Kod 8.12: angular/komponente1/src/app/student4/student4.component.ts

```

1 import { Component, OnInit } from '@angular/core';
2
3 @Component({
4   selector: 'app-student4',
5   templateUrl: './student4.component.html',
6   styleUrls: ['./student4.component.css']
7 })
8 export class Student4Component implements OnInit {
9
10   dugmeJeUkljuceno: boolean = false;
11   ime: string;
12   prezime: string;
13
14   constructor() {
15     setTimeout(() => {
16       this.dugmeJeUkljuceno = true;
17     }, 2000);
18   }
19
20   ngOnInit() {
21   }
22
23 }
```

Prikažimo prvo šablon koji koristi ove vrednosti:

Kod 8.13: angular/komponente1/src/app/student4/student4.component.html

```

1 <p>Ime: {{ ime }}</p>
```

```

2 <p>Prezime: {{ prezime }}</p>
3 <input type="button"
4       value="Dohvati informacije"
5       class="btn btn-primary"
6       [disabled]="!dugmeJeUkljuceno">
7 <br><br>

```

Interpolacija niski nam je jasna. Dugme koje se nalazi ispod ima podešene Bootstrap klase `btn` i `btn-primary` kako bi izgledalo lepše. Međutim, ovde vidimo novu sintaksu:

Kod 8.14: angular/komponente1/src/app/student4/student4.component.html (linije 6-6)

```

6 [disabled]="!dugmeJeUkljuceno">

```

Ova sintaksa upravo definiše vezivanje atributa. Njeno značenje je sledeće: izračunaj vrednost izraza `!dugmeJeUkljuceno` iz modela i postavi atribut `disabled` na tu vrednost. Time smo izvršili "vezivanje atributa" za model, te otuda i ime ovog mehanizma vezivanja po-dataka. Ključno je primetiti dve stvari:

1. Vezivanje atributa koristi uglaste zagrade `[]` između kojih se navodi naziv atributa iz DOM interfejsa koji odgovara tom elementu (ili iz nekog njegovog nadinterfejsa).
2. Vrednost koja se navodi između navodnika predstavlja TypeScript izraz koji se izračunava i njegova vrednost se postavlja atributu. Kao i kod interpolacije niski, i ovaj izraz mora biti linijski i može obuhvatati bilo koji izraz (promenljivu, pozive funkcija, kompoziciju operatora i funkcija i dr.).

Posledica ovoga jeste da u zavisnosti od atributa, tip vrednosti pod navodnicima može biti različita. Tako, na primer, atribut `disabled` elementa `<input>` očekuje Bulovu vrednost, dok, na primer, svi HTML elementi imaju svojstvo `innerText`, koji očekuje nisku koja predstavlja prikazani sadržaj u elementu.

S obzirom da je vrednost `!dugmeJeUkljuceno` inicijalno `true`, pri učitavanju komponente, dugme će biti isključeno (`disabled="true"`). Nakon dve sekunde, vrednost `dugmeJeUkljuceno` se postavlja na `true`, odnosno, vrednost izraza `!dugmeJeUkljuceno` je `false`, te se u tom trenutku dugme uključuje (`disabled="false"`).

Vezivanje događaja

Unapredimo sada napisane komponente, tako da se klikom na dugme izračunavaju svojstva `ime` i `prezime` u modelu, koja će zatim biti prikazana u šablonu zbog interpolacije niski. U tu svrhu kreirajmo komponentu `Student5Component` koja ima početnu implementaciju i odgovarajući šablon kao komponenta `Student4Component`, kao i komponentu `Student5ListaComponent` koja će sadržati tri ove komponente u svom šablonu. Ne zaboravimo da ažuriramo šablon korene komponente da prikazuje `Student5ListaComponent`.

Podimo ponovo od modela:

Kod 8.15: angular/komponente1/src/app/student5/student5.component.ts

```

1 import { Component, OnInit } from '@angular/core';
2
3 @Component({
4   selector: 'app-student5',
5   templateUrl: './student5.component.html',
6   styleUrls: ['./student5.component.css']
7 })
8 export class Student5Component implements OnInit {
9

```

```

10     static listaImena: Array<string> = ['Ana', 'Nevena', 'Djordje', 'Milos'];
11     static listaPrezimana: Array<string> = ['Jovanovic', 'Nikolic', 'Stefanovic',
12       'Milovanovic'];
13
14     inicijalizovanStudent: boolean = false;
15     dugmeJeUkljuceno: boolean = false;
16     ime: string;
17     prezime: string;
18
19     constructor() {
20         setTimeout(() => {
21             this.dugmeJeUkljuceno = true;
22         }, 2000);
23     }
24
25     ngOnInit() {
26
27     onDohvatiInformacije(): void {
28         if (this.inicijalizovanStudent) {
29             return;
30         }
31         this.ime = Student5Component.listaImena[Math.floor(Math.random() *
32           Student5Component.listaImena.length)];
33         this.prezime = Student5Component.listaPrezimana[Math.floor(Math.random
34           () * Student5Component.listaPrezimana.length)];
35         this.inicijalizovanStudent = true;
36     }
37 }

```

Na raspolaganju su nam statički nizovi koji sadrže imena i prezimena, kao i metod `onDohvatiInformacije()` koji će komponenti dodeliti ime i prezime iz tih nizova, ali tačno jednom, za šta koristimo svojstvo `inicijalizovanStudent`.

Očigledno, ono što je potrebno uraditi jeste, kada korisnik klikne na dugme iz komponente, pozvati metod `onDohvatiInformacije()` koji će izvršiti ovu inicijalizaciju.

Vezivanje događaja (engl. *event binding*) predstavlja mehanizam kojim se reaguje na promene u HTML šablonu radi ažuriranja modela. Kao i prethodna dva mehanizma, i ovaj mehanizam je jednosmeran, ali za razliku od njih, ovoga puta smer je od HTML šablona ka TypeScript-u.

Prikažimo prvo šablon koji koristi vezivanje događaja, a zatim ćemo detaljnije diskutovati o njemu:

Kod 8.16: angular/komponente1/src/app/student5/student5.component.html

```

1 <p>Ime: {{ ime }}</p>
2 <p>Prezime: {{ prezime }}</p>
3 <input type="button"
4     value="Dohvati informacije"
5     class="btn btn-primary"
6     [disabled]="!dugmeJeUkljuceno"
7     (click)="onDohvatiInformacije()">
8 <br><br>

```

Kao što vidimo, nova sintaksa je data u narednoj liniji:

Kod 8.17: angular/komponente1/src/app/student5/student5.component.html (linije 7-7)

```
7 (click)="onDohvatiInformacije()">
```

Ova sintaksa upravo definiše vezivanje događaja. Njeno značenje je sledeće: kada god se okine događaj čiji je naziv `click`, pozovi metod `onDohvatiInformacije()`. Time smo izvršili "vezivanje događaja" za model, te otuda i ime ovog mehanizma vezivanja podataka. Ključno je primetiti dve stvari:

1. Vezivanje događaja koristi obične zagrade (`()`) između kojih se navodi naziv događaja iz DOM interfejsa koji odgovara tom elementu (ili iz nekog njegovog nadinterfejsa).
2. Vrednost koja se navodi između navodnika predstavlja TypeScript metod koji se poziva onda kada se događaj sa datim imenom ispali nad HTML elementom nad kojim se vezivanje događaja primenjuje.

Konvencija je da se TypeScript izrazi koji se izračunavaju pri vezivanju događaja smeštaju u metode komponente čiji nazivi počinju na `on*`.

Sada ako podignemo aplikaciju i nakon dve sekunde kliknemo na neko dugme, primetićemo da će student na čije je dugme kliknuto dobiti ime i prezime (u modelu) koji će biti automatski prikazani u pogledu zbog interpolacije niski.

Vezivanje događaja — Argument `$event`

Kreirajmo sada komponentu `Student6Component`. Ova komponenta će služiti da korisnik može da unese ime studenta, a zatim da klikne na dugme kojim će se sačuvati izmene (tako što se onemogućuje dalji unos). Kreirajmo i odgovarajuću `Student6ListaComponent` i izmenimo korenu komponentu da je prikazuje.

Pogledajmo prvo model koji ovo implementira:

Kod 8.18: angular/komponente1/src/app/student6/student6.component.ts

```
1 import { Component, OnInit } from '@angular/core';
2
3 @Component({
4   selector: 'app-student6',
5   templateUrl: './student6.component.html',
6   styleUrls: ['./student6.component.css']
7 })
8 export class Student6Component implements OnInit {
9
10   private ime: string;
11   private menjanjeJeDozvoljeno: boolean = true;
12
13   constructor() { }
14
15   ngOnInit() {
16   }
17
18   unosJeOnemogucen(): boolean {
19     return !this.menjanjeJeDozvoljeno;
20   }
21
22   onSacuvajInformacije(): void {
23     this.menjanjeJeDozvoljeno = false;
24   }
25
26   onIzmeniIme(event: Event): void {
27     this.ime = (<HTMLInputElement>event.target).value;
```

```

28     }
29
30 }

```

Pored svojstva `ime` koje će sadržati ime studenta, model sadrži i svojstvo `menjanjeJeDozvoljeno` na osnovu kojeg se određuje da li je unos dozvoljen ili ne. Na raspolaganju su nam i pomoćni metodi `unosJeOnemogucen()` koji će biti korišćen u šablonu za proveru mogućnosti unosa i `onSacuvajInformacije()` koji će onemogućiti dalji unos podataka. Pre nego što prodiskutujemo metod `onIzmeniIme`, pogledajmo šablon komponente:

Kod 8.19: angular/komponente1/src/app/student6/student6.component.html

```

1 <p>Moje ime je {{ ime }}</p>
2 <div class="form-group">
3   <label>Ime:</label>
4   <input type="text"
5         class="form-control"
6         [disabled]="unosJeOnemogucen()"
7         (keyup)="onIzmeniIme($event)">
8 </div>
9 <input type="button"
10       value="Sačuvaj informacije"
11       class="btn btn-primary"
12       [disabled]="unosJeOnemogucen()"
13       (click)="onSacuvajInformacije()">

```

Prikazivanje svojstva `ime` kroz interpolaciju niski i vezivanje atributa `disabled` za polje za unos i dugme smo već diskutovali, kao i vezivanje događaja `click` nad dugmetom. Ono što je novo jeste vezivanje događaja `keyup`¹ nad poljem za unos:

Kod 8.20: angular/komponente1/src/app/student6/student6.component.html (linije 7-7)

```

7       (keyup)="onIzmeniIme($event)">

```

Ono što ovde primećujemo jeste da sada metod `onIzmeniIme` prihvata argument `$event`. U pitanju je specijalna vrednost koja će u TypeScript-u biti vidljiva kao instanca DOM interfejsa `Event` koji je ispaljen nad tim elementom, u ovom slučaju, nad `<input>` elementom. Samim tim, moramo da obezbedimo da metod `onIzmeniIme` upravo prihvata taj argument, što i radimo:

Kod 8.21: angular/komponente1/src/app/student6/student6.component.ts (linije 26-28)

```

26   onIzmeniIme(event: Event): void {
27     this.ime = (<HTMLInputElement>event.target).value;
28   }

```

U telu ovog metoda pristupamo svojstvu `target` nad događajem koji nam izračunava referencu ka elementu nad kojim je događaj ispaljen, čiji je tip `Element`. Kako mi znamo da je u pitanju element `<input>` koji predstavlja instancu DOM interfejsa `HTMLInputElement`, onda ga možemo eksplicitno konvertovati u instancu tog interfejsa, da bismo mogli da pristupimo svojstvu `value` koji predstavlja vrednost tog `<input>` polja². Konačno, dohvaćenu vrednost iz polja koristimo da postavimo vrednost svojstva `ime` u komponenti.

¹Ovaj događaj se okida nakon što korisnik pritisne taster dok je polje za unos u fokusu.

²Setimo se da svako polje `<input>` ima atribut `value=""` koja zapravo predstavlja ono što je upisano u to polje. Upravo na opisani način možemo pristupiti toj vrednosti. Eksplicitno kastovanje je u TypeScript-u neophodno zbog provere tipova, ali naravno, u JavaScript-u ne bismo imali ovakvu proveru, već bismo mogli da pristupimo na jednostavniji način: `event.target.value`.

Podizanjem aplikacije, možemo se uveriti da nakon što otkucamo nekoliko karaktera u polju za unos, svakim karakterom vrednost `ime` će biti izmenjena i to će biti oslikano u prikazu.

Dvosmerno vezivanje

Dvosmerno vezivanje (engl. *two-way databinding*) predstavlja dvosmerni mehanizam povezivanja koji, očigledno, implementira promenu vrednosti u oba smera — ukoliko dođe do promene u modelu, šablon će biti ažuriran, ali važi i obrnuto, ukoliko dođe do promene u šablonu, model će biti ažuriran.

Da bismo demonstrirali dvosmerno vezivanje, kreiraćemo komponentu `Student7Component` koja će imati identičnu implementaciju kao komponenta `Student6Component`, sa jednom razlikom u šablonu, koji prikazujemo u nastavku:

Kod 8.22: angular/komponente1/src/app/student7/student7.component.html

```

1 <p>Moje ime je {{ ime }}</p>
2 <div class="form-group">
3   <label>Ime:</label>
4   <input type="text"
5         class="form-control"
6         [disabled]="unosJeOnemogucen()"
7         (keyup)="onIzmeniIme($event)">
8 </div>
9 <div class="form-group">
10  <label>Dvostruko povezano ime:</label>
11  <input type="text"
12        class="form-control"
13        [disabled]="unosJeOnemogucen()"
14        [(ngModel)]="ime">
15 </div>
16 <input type="button"
17       value="Sačuvaj informacije"
18       class="btn btn-primary"
19       [disabled]="unosJeOnemogucen()"
20       (click)="onSacuvajInformacije()">

```

Kao što vidimo, dodali smo još jedno polje za unos koje smo nazvali "Dvostruko povezano ime", kako bismo ga razlikovali od polja "Ime". Ono što primećujemo jeste nova sintaksa u narednoj liniji:

Kod 8.23: angular/komponente1/src/app/student7/student7.component.html (linije 14-14)

```

14      [(ngModel)]="ime">

```

Atribut `ngModel` predstavlja specijalni atribut koji vezuje model i element. Međutim, on nije HTML atribut, već Angular atribut, tako da je u korenom modulu naše aplikacije neophodno:

1. Uvesti Angular modul `FormsModule`:

Kod 8.24: angular/komponente1/src/app/app.module.ts (linije 3-3)

```

3 import { FormsModule } from '@angular/forms';

```

2. Referisati na njega u `imports` nizu:

Kod 8.25: angular/komponente1/src/app/app.module.ts (linije 37-40)

```
37 imports: [
38   BrowserModule,
39   FormsModule
40 ],
```

Vratimo se nazad na sintaksu dvosmernog vezivanja:

Kod 8.26: angular/komponente1/src/app/student7/student7.component.html (linije 14-14)

```
14 [(ngModel)]="ime">
```

Ako malo bolje pogledamo, vidimo da dvosmerno vezivanje koristi kombinaciju sintakse vezivanja atributa (uglastim zagradama) i vezivanja događaja (običnim zagradama). Zapravo, dvosmerno vezivanje možemo da shvatimo upravo kao uniju ova dva pristupa — vrednost `ime` iz modela utiče na vrednost atributa `ngModel` kao u vezivanju atributa, ali takođe, korisnikove promene u polju utiču na vrednost `ime` u modelu kao u vezivanju događaja.

Sada podignimo aplikaciju da vidimo šta dvosmerno vezivanje praktično znači. Ukoliko krenemo da unosimo ime studenta u polju "Ime", videćemo da se, pored prikazivanja imena interpolacijom niski iznad tog polja, vrednost imena takođe prikazuje i u polju "Dvostruko povezano ime". Međutim, ako krenemo da kucamo ime u polju `Dvostruko povezano ime`, videćemo da polje `Ime` ne menja svoju vrednost — što je i očekivano jer ne postoji veza u smeru od modela ka tom polju.

Smernice za pisanje šablonskih izraza

TypeScript izrazi koji se navode prilikom vezivanja podataka se nazivaju *šablonski izrazi* (engl. *template expression*). Na primer, prilikom interpolacije niski u narednom primeru koda:

```
1 <p>Jedan plus jedan je jednako {{ 1 + 1 }}.</p>
```

šablonski izraz `1 + 1` je okružen dvostrukim vitičastim zagradama. Iako nismo do sada diskutovali o tome, nisu svi TypeScript izrazi validni šablonski izrazi. Šablonski izraz mora da isprati naredne smernice:

- Smernica 1: Bez vidljivih bočnih efekata
 - Šablonski izraz ne sme menjati bilo koje stanje aplikacije osim vrednosti ciljanog svojstva. Ovo pravilo je važno zbog politike jednosmernog protoka podataka (izuzetak od ovog pravila je dvosmerno vezivanje). Ne bi trebalo da brinemo da će čitanje vrednosti podatka promeniti neku od prikazanih vrednosti kroz jedan prolazak u renderovanju.
- Smernica 2: Brza izračunljivost
 - Angular izvršava šablonske izraze nakon svakog ciklusa u kojem su detektovane promene. Ovi ciklusi se aktiviraju prilikom raznih asinhronih aktivnosti kao što su ispunjavanja obećanja, rezultati HTTP zahteva, događaja usled isteka vremena, događaja sa tastature i miša i sl. Zbog toga, izračunavanje šablonskih izraza bi trebalo da bude efikasno. U suprotnom, može doći do vidljivog usporenja aplikacije, posebno na sporijim uređajima. Skuplja izračunavanja bi trebalo keširati.
- Smernica 3: Jednostavnost

- Iako je moguće pisanje složenijih šablonskih izraza, preporučuje se korišćenje svojstava i metoda, uz poneko korišćenje Bulovog operatora `!`. U suprotnom, aplikaciona i poslovna logika bi trebalo biti implementirana kao deo komponente.

8.5 Ugrađene Angular direktive

Direktive (engl. *directive*) predstavljaju instrukcije u okviru DOM stabla. Pod "instrukcije" mislimo operacije nad DOM stablom koje menjaju prikaz stranice, stil elemenata, i sl. Sve Angular direktive se mogu podeliti u tri vrste:

1. Komponente — Da, i komponente predstavljaju direktive jer umetanjem komponente u HTML šablon mi zapravo govorimo Angular-u da je potrebno izvršiti instrukciju dodavanja elementa u DOM stablo. Često se kaže da su komponente u Angular-u direktive sa pogledom.
2. Strukturne direktive — U pitanju su instrukcije koje dodaju ili uklanjaju DOM elemente. Ovaj opis može zvučati sličan opisu za komponente, ali razlika je u tome što strukturne direktive ne prate HTML šabloni koji se umeću.
3. Atributske direktive — U pitanju su instrukcije koje menjaju izgled ili ponašanje elementa, komponente ili druge direktive.

U ovoj sekciji ćemo diskutovati o nekim, već postojećim direktivama koje dolaze uz Angular, dok ćemo u sekciji 8.7 govoriti o tome kako je moguće da mi kreiramo naše direktive.

Krenimo od novog projekta:

```
$ ng new direktive1
```

Kreirajmo i komponentu koja će nam služiti kao početna komponenta za primere u nastavku:

```
$ ng generate component student
```

Kod 8.27: angular/direktive1/src/app/student/student.component.ts

```
1 import { Component, OnInit } from '@angular/core';
2
3 @Component({
4   selector: 'app-student',
5   templateUrl: './student.component.html',
6   styleUrls: ['./student.component.css']
7 })
8 export class StudentComponent implements OnInit {
9
10   private static readonly listaImena: Array<string> = ['Una', 'Milica', '
      Marko', 'Bojan'];
11   private static readonly brojImena: number = StudentComponent.listaImena.
      length;
12
13   private ocene: Array<number> = [];
14   private ime: string = '';
15   private ocenaStr: string = null;
16
17   constructor() {
18     this.ime = StudentComponent.listaImena[Math.floor(Math.random() *
      StudentComponent.brojImena)];
19   }
20 }
```

```

21     ngOnInit() {
22     }
23
24     private prosek(): number {
25         if (this.ocene.length === 0) {
26             return 0.0;
27         }
28         return this.ocene.reduce((prev, next) => prev + next) / this.ocene.
            length;
29     }
30
31     private onDodajOcenu(): void {
32         if (Number.isNaN(Number.parseInt(this.ocenaStr))) {
33             this.ocenaStr = null;
34             return;
35         }
36
37         const ocenaNum: number = Number.parseInt(this.ocenaStr);
38         if (ocenaNum < 5 || ocenaNum > 10) {
39             this.ocenaStr = null;
40             return;
41         }
42
43         this.ocene.push(ocenaNum);
44         this.ocenaStr = null;
45     }
46
47 }

```

Kod 8.28: angular/direktive1/src/app/student/student.component.html

```

1  <div class="alert alert-primary">
2    <h3>Student: {{ ime }}</h3>
3
4    <p>Prosek: {{ prosek() }}</p>
5
6    <hr>
7
8    <div class="form-group">
9      <label for="ocena">Dodaj ocenu:</label>
10     <input type="text"
11       id="ocena"
12       class="form-control"
13       [(ngModel)]="ocenaStr">
14     </div>
15
16     <button class="btn btn-primary"
17       (click)="onDodajOcenu()">Dodaj ocenu</button>
18 </div>

```

Kao što vidimo, `StudentComponent` predstavlja informacije o jednom studentu:

- Ime studenta se bira nasumično prilikom konstrukcije (`ime`).
- Čuvaju se informacije o njegovim ocenama (`ocene`).
- Čuva se informacija o oceni koja se unosi u tekstualno polje (`ocenaStr`).
- Dostupni su metodi koji vrše sledeće akcije:
 - Izračunavanje proseka ocena (`prosek()`).
 - Dodavanje trenutno unete ocene iz tekstualnog polja u niz ocena, praćeno dodatnim proverama unete vrednosti (`onDodajOcenu()`).

Izmenimo početnu postavku `app.component.html` datoteke:

```

1 <div class="container">
2   <div class="row">
3     <div class="col-12">
4       <h1>Direktive 1</h1>
5
6       <!-- Komponente koje budemo kreirali
7           bice smestene ovde, umesto pocetne komponente -->
8       <app-student></app-student>
9     </div>
10  </div>
11 </div>

```

Pre nego što pređemo na opise svake od direktiva, skrenimo pažnju na neke važne napomene. Kroz nastavak teksta, videćemo da se na direktive referiše bilo u *VelikojKamiljojNotaciji* bilo u *malojKamiljojNotaciji*, na primer, `NgIf` i `ngIf`. Postoji vrlo jasan razlog za insistiranje na ovoj razlici — `NgIf` referiše na *klasu* direktive, dok `ngIf` referiše na *ime atributa* direktive. Kada budemo diskutovali o svojstvima direktive i o tome čemu direktiva služi, korišćićemo *klasu* direktive. Sa druge strane, kada budemo opisivali kako se direktiva primenjuje nad HTML elementima, korišćićemo *ime atributa* direktive.

8.5.1 Strukturne direktive

Kao što smo rekli, strukturne direktive se koriste za menjanje strukture DOM stabla. Kao i sa svim direktivama, i one se primenjuju nad elementima nad kojima želimo da izvršimo neku operaciju. Najviše se jedna strukturna direktiva može primeniti nad elementom.

NgIf

Prva direktiva sa kojom se upoznajemo je direktiva `NgIf`. Ova direktiva služi za uklanjanje ili rekreiranje dela DOM stabla na osnovu Bulovog izraza. Pogledajmo rad ove direktive na delu:

Kod 8.29: `angular/direktive1/src/app/ngif-test/ngif-test.component.html`

```

1 <div class="alert alert-primary">
2   <h3>Student: {{ ime }}</h3>
3
4   <div class="form-check">
5     <input class="form-check-input"
6           type="checkbox"
7           id="prosekCheck"
8           (change)="onPromeniPrikazivanjeProseka()">
9     <label class="form-check-label" for="prosekCheck">
10       Prikaži prosek
11     </label>
12   </div>
13
14   <p *ngIf="prikaziProsek">Prosek: {{ prosek() }}</p>
15
16   <hr>
17
18   <div class="form-group">
19     <label for="ocena">Dodaj ocenu:</label>
20     <input type="text"
21           id="ocena"
22           class="form-control"
23           [(ngModel)]="ocenaStr">
24   </div>

```

```

25
26     <button class="btn btn-primary"
27           (click)="onDodajOcenu()">Dodaj ocenu</button>
28 </div>

```

Kao što vidimo, početna komponenta `StudentComponent` dopunjena je jednim *poljem za potvrdu* (engl. *checkbox*) kao i paragrafom nad kojim je upotrebljena direktiva `ngIf`. Prikažimo i dopunu TypeScript koda početne komponente:

Kod 8.30: angular/direktive1/src/app/ngif-test/ngif-test.component.ts (linije 47-51)

```

47     private prikaziProsek: boolean = false;
48
49     private onPromeniPrikazivanjeProseka(): void {
50         this.prikaziProsek = !this.prikaziProsek;
51     }

```

Kao što vidimo u šablonu, nakon svake promene polja za potvrdu, poziva se metod koji će promeniti Bulovu vrednost, a koja određuje da li se paragraf sa prosekom studenta prikazuje ili ne, pomoću korišćenja direktive `ngIf`. Pogledajmo sintaksu:

Kod 8.31: angular/direktive1/src/app/ngif-test/ngif-test.component.html (linije 14-14)

```

14     <p *ngIf="prikaziProsek">Prosek: {{ prosek() }}</p>

```

Kao što vidimo, direktiva `ngIf` se primenjuje nad elementom koji želimo da uklonimo ili dodamo u DOM stablo (u ovom primeru, želimo da uklonimo/rekreiramo paragraf koji sadrži dati tekst). Izraz između navodnika predstavlja šablonski izraz koji uslovljava postojanje elementa nad kojim se direktiva `ngIf` primenjuje. Dodatno, primetimo (i ono što je veoma važno — nikako ne zaboravimo) da sintaksa korišćenja ove direktive, kao i još nekih drugih strukturnih direktiva, koristi karakter `*` ispred naziva direktive. Na ovu napomenu ćemo se osvrnuti ponovo pri kraju ove podsekcije.

NgIf — uslov "inače" i element `<ng-template>`

Direktiva `ngIf` ima i opcionu klauzu `else` koja može poslužiti za rekreiranje prikaza u slučaju da se šablonski izraz izračunava na vrednost `false`. Očigledno, u slučaju nepostojanja ove klauze, neće biti rekreiran nijedan prikaz.

Za početak, pogledajmo primer upotrebe klauze `else`:

Kod 8.32: angular/direktive1/src/app/ngifelse-test/ngifelse-test.component.html (linije 14-17)

```

14     <p *ngIf="prikaziProsek; else prosekJeIskljucen">Prosek: {{ prosek() }}</p>
15     <ng-template #prosekJeIskljucen>
16         <p>Kliknite na checkbox iznad da pogledate prosek</p>
17     </ng-template>

```

Kao što vidimo, klauza `else` se specifikuje u okviru šablonskog izraza direktive `ngIf`, nakon karaktera `;` i njena vrednost je referenca na specijalni Angular element `<ng-template>`, koji vidimo da je upotrebljen u kodu.

`<ng-template>` predstavlja element dostupan uz Angular koji služi za prikazivanje HTML šablona. Međutim, on se ne prikazuje direktno u finalnom prikazu. Zapravo, prilikom procesiranja šablona, Angular će zameniti sva pojavljivanja ovog elementa po jednim komentarom. Međutim, `<ng-template>` ima razne upotrebne vrednosti i sa nekima od njih ćemo se susresti.

Na primer, `<ng-template>` se može iskoristiti za definisanje šablona koji *može* biti prikazan prilikom korišćenja direktive `ngIf`. Kao što vidimo u primeru koda iznad, `<ng-template>` definiše poruku korisniku koja će biti prikazana samo ukoliko je vrednost `prikaziProsek` jednaka `false`. Ovo je omogućeno kroz korišćenje specijalne *referencne promenljive šablona* (engl. *template reference variable*) `#prosekJeIskljucen` primenjene na `<ng-template>`, a zatim iskorišćene u okviru klause `else` direktive `ngIf`.

Očigledno, s obzirom da se sadržaj `<ng-template>` elementa ne prikazuje direktno, već se na njega referiše putem referencne promenljive šablona, nije bilo neophodno da se `<ng-template>` nađe direktno ispod `ngIf` direktive, ali to predstavlja dobru praksu.

NgFor

Često su nam podaci, koje je potrebno da prikažemo, zadati u nizovima. Na primer, s obzirom da naše `StudentComponent` komponente sadrže niz ocena, moglo bi biti korisno prikazati sve te ocene. U ovakve svhre, korišćenje `NgFor` direktive je vrlo praktično. Poput direktive `ngIf`, i direktiva `ngFor` se primenjuje nad elementom koji je potrebno prikazati, samo što se ovoga puta prikazuje kolekcija ovih elemenata. Pogledajmo naredni primer:

Kod 8.33: angular/direktive1/src/app/ngfor-test/ngfor-test.component.html (linije 22-25)

```
22 <h6>Ocene:</h6>
23 <ol>
24   <li *ngFor="let ocenaIzNiza of ocene">{{ ocenaIzNiza }}</li>
25 </ol>
```

Ono što možemo da zaključimo jeste da želimo da prikažemo kolekciju ocena kao numerisanu listu (elementom ``), pri čemu se za prikaz jedne ocene koristi element te liste (element ``). Pogledajmo pažljivo sintaksu `ngFor` direktive:

Kod 8.34: angular/direktive1/src/app/ngfor-test/ngfor-test.component.html (linije 24-24)

```
24 <li *ngFor="let ocenaIzNiza of ocene">{{ ocenaIzNiza }}</li>
```

Poput direktive `ngIf`, i direktiva `ngFor` koristi karakter `*` praćen nazivom direktive, a zatim pod navodnicima izraz. Samo, ovoga puta, izraz koji se koristi veoma liči na iterator *for-of* petlje:

```
1 for (let ocenaIzNiza of ocene) {
2   // Radi nesto sa objektom ocenaIzNiza...
3 }
```

I zaista, primenjena na neki element, direktiva `NgFor` omogućava da element dobije referencu ka elementu iz niza u svakoj iteraciji, kojoj može da pristupi radi dalje obrade. U ovom slučaju, koristimo dobijenu ocenu samo da bismo je prikazali kao sadržaj elementa ``. Alternativno, mogli bismo da je koristimo u međuizračunavanjima, kao argument metoda i dr.

NgFor — korišćenje rangovskih promenljivih

Direktiva `NgFor` ima jedno zanimljivo svojstvo, a to je da nam nudi na raspolaganju razne *rangovske promenljive* (engl. *range variables*). Lista ovih promenljivih, zajedno sa njihovim opisima, data je u tabeli 8.1.

Tabela 8.1: Rangovske promenljive dostupne prilikom korišćenja direktive `NgFor`.

Rang. prom.	Opis
-------------	------

<code>index</code>	Predstavlja brojevu vrednost indeksa dodeljenu elementu iz tekuće iteracije.
<code>odd</code>	Bulova vrednost čija je vrednost jednaka <code>true</code> ukoliko se tekući element nalazi na neparnoj poziciji u nizu.
<code>even</code>	Bulova vrednost čija je vrednost jednaka <code>true</code> ukoliko se tekući element nalazi na parnoj poziciji u nizu.
<code>first</code>	Bulova vrednost čija je vrednost jednaka <code>true</code> ukoliko se tekući element nalazi na prvoj poziciji u nizu.
<code>last</code>	Bulova vrednost čija je vrednost jednaka <code>true</code> ukoliko se tekući element nalazi na poslednjoj poziciji u nizu.

Da bi ove vrednosti bile dostupne, potrebno ih je dodeliti promenljivama u izrazu `ngFor` direktive, nakon `let elem of niz` izraza, a odvojeno karakterom `;` kao u narednom primeru:

Kod 8.35: angular/direktive1/src/app/ngfor-test/ngfor-test.component.html (linije 29-32)

```

29     <h6>Ocene:</h6>
30     <p *ngFor="let ocenaIzNiza of ocene; let indeks = index">
31         {{ indeks+1 }}. ocena: {{ ocenaIzNiza }}
32     </p>

```

NgSwitch

Pored uslovne direktive `NgIf`, na raspolaganju nam je još jedna uslovna direktiva čiji strukturni koncept odgovara naredbi `switch` u programskim jezicima, te je i sama direktiva nazvana `NgSwitch`. Neka je potrebno da prikazemo odgovarajuću poruku u zavisnosti od toga koliko je predmeta student položio (odnosno, koliko ima ocena koje su strogo veće od 5). U tu svrhu, definišimo metod koji će izračunavati broj položenih predmeta:

Kod 8.36: angular/direktive1/src/app/ngswitch-test/ngswitch-test.component.ts (linije 53-55)

```

53     private brojPolozenihPredmeta(): number {
54         return this.ocene.filter(val => val > 5).length;
55     }

```

Pre nego što detaljnije budemo govorili o ponašanju ove direktive, pogledajmo sintaksu kroz naredni primer:

Kod 8.37: angular/direktive1/src/app/ngswitch-test/ngswitch-test.component.html (linije 34-40)

```

34     <div [ngSwitch]="brojPolozenihPredmeta()">
35         <div *ngSwitchCase="0">Nijedan predmet nije položen :(</div>
36         <div *ngSwitchCase="1">Uspeli smo da položimo neki predmet!</div>
37         <div *ngSwitchCase="2">Na dobrom smo putu</div>
38         <div *ngSwitchCase="3">Još samo jedan ispit do 30 ESPB</div>
39         <div *ngSwitchDefault>Očistili smo makar jedan semestar</div>
40     </div>

```

Samo na prvi pogled možemo uočiti klasične elemente naredbe `switch`:

1. Izraz čija se vrednost bude testirala se zadaje sintaksom `[ngSwitch]="brojPolozenihPredmeta()"`.

2. Vrednost izraza se testira nad svakim "slučajem" zadatim sintaksom `*ngSwitchCase="0"`, `*ngSwitchCase="1"`, ...
3. Moguće je koristiti "podrazumevani" test koji će biti iskorišćen u slučaju da nijedan prethodni test nije uspešan sintaksom `*ngSwitchDefault`.

Kao što vidimo, `NgSwitch` direktiva se zapravo koristi kao trojka direktiva `NgSwitch`, `NgSwitchCase` i `NgSwitchDefault`.

Ono što može biti zbunjujuće jeste zašto se `*` ne koristi uz `ngSwitch`, kao što je to bio slučaj sa `ngIf` i `ngFor` direktivama, kada je u pitanju strukturna direktiva? Zapravo, ispostavlja se da sama direktiva `NgSwitch` je atributska, a ne strukturna, jer ne menja DOM stablo direktno, već samo utiče na ponašanje pratećih direktiva.

Za razliku od nje, direktive `NgSwitchCase` i `NgSwitchDefault` jesu strukturne jer dodaju ili uklanjaju elemente iz DOM stabla. Zbog toga se `*` i koristi uz njih, kao u prethodnom primeru.

Ipak, zbog efekta koje ova trojka direktiva finalno proizvodi (menjanje DOM stabla), kada govorimo o direktivi `NgSwitch` u širem smislu (odnosno, njen koncept), govorimo o njoj kao o strukturnoj direktivi. Naravno, i dalje treba imati u vidu da je sama direktiva `NgSwitch` u užem smislu (odnosno, njena implementacija) atributskog tipa.

Čemu služi karakter `*` u strukturnim direktivama?

Sada kada smo verovatno uveli dodatnu konfuziju mešanjem strukturnih i atributskih direktiva, pokušaćemo da se opravdamo davanjem odgovora na pitanje koje predstavlja i naslov ovog dela teksta.

Zvezdasti prefiks (engl. *asterisk prefix*) predstavlja sintaksičku pomoć koju Angular razume. Njegova uloga je da prevede atribut `*mojaDirektiva` u element `<ng-template>` koji će biti obavijen oko elementa nad kojim je direktiva primenjena. Primenjeno na narednom primeru, Angular prevodi naredni kod

```
1 <div *mojaDirektiva="student" class="ime">{{ student.ime }}</div>
```

u naredni kod

```
1 <ng-template [mojaDirektiva]="student">
2   <div class="ime">{{ student.ime }}</div>
3 </ng-template>
```

Dakle, Angular je uradio dve stvari:

- Direktiva `*mojaDirektiva` je izmeštena iz elementa nad kojim je primenjena u element `<ng-template>` u kojem se koristi kao vezivanje atributa.
- Ostatak `<div>` elementa, zajedno sa atributom `class` i sadržajem, izmešten je kao HTML sadržaj elementa `<ng-template>`.

Kao što vidimo, korišćenjem zvezdastog prefiksa Angular nam omogućava da pišemo manje koda, te se preferira njegova upotreba.

8.5.2 Atributske direktive

Kao što smo rekli, atributske direktive se primenjuju radi promene izgleda elementa, komponente ili druge direktive, u prikazu. Moguće je primeniti više atributskih direktiva nad istim elementom.

NgClass

Direktiva **NgClass** služi za dinamičko postavljanje klasa nad elementom nad kojim se primenjuje. Pogledajmo HTML šablon koji predstavlja tek nešto izmenjen HTML šablon komponente **StudentComponent**:

Kod 8.38: angular/direktive1/src/app/ngclass-test/ngclass-test.component.html

```

1 <div [ngClass]="izracunajKlasu()">
2   <h3>Student: {{ ime }}</h3>
3   <p>Prosek: {{ prosek() }}</p>
4
5   <hr>
6
7   <div class="form-group">
8     <label for="ocena">Dodaj ocenu:</label>
9     <input type="text"
10        id="ocena"
11        class="form-control"
12        [(ngModel)]="ocenaStr">
13   </div>
14
15   <button class="btn btn-primary"
16        (click)="onDodajOcenu()">Dodaj ocenu</button>
17 </div>

```

Kao što vidimo, direktiva **NgClass** je iskorišćena nad korenim **<div>** elementom, te je upravo taj element kojem će klasa biti dinamički dodeljena. Vrednost ove direktive je objekat čija su svojstva nazivi klasa, a vrednosti ovih svojstava su dinamički Bulovi uslovi koji određuju da li će klasa sa datim nazivom biti dodeljena elementu ili ne. Na primer, u zavisnosti od proseka, student će biti prikazan drugom bojom:

Kod 8.39: angular/direktive1/src/app/ngclass-test/ngclass-test.component.ts (linije 47-58)

```

47 private izracunajKlasu() {
48   const prosek = this.prosek();
49   return {
50     'alert': true,
51     'alert-success': prosek >= 9,
52     'alert-primary': prosek >= 8 && prosek < 9,
53     'alert-info': prosek >= 7 && prosek < 8,
54     'alert-warning': prosek >= 6 && prosek < 7,
55     'alert-danger': prosek < 6 && prosek !== 0,
56     'alert-light': prosek === 0
57   }
58 }

```

NgStyle

Direktiva **NgStyle** služi za dinamičko postavljanje CSS svojstava nad elementom nad kojim se primenjuje. U narednom HTML šablonu, primetićemo da se nad elementom **<p>** postavljaju CSS svojstva pomoću ove direktive:

Kod 8.40: angular/direktive1/src/app/ngstyle-test/ngstyle-test.component.html

```

1 <div class="alert alert-dark">
2   <h3>Student: {{ ime }}</h3>
3   <p [ngStyle]="izracunajStil()">Prosek: {{ prosek() }}</p>
4
5   <hr>
6

```

```

7     <div class="form-group">
8         <label for="ocena">Dodaj ocenu:</label>
9         <input type="text"
10             id="ocena"
11             class="form-control"
12             [(ngModel)]="ocenaStr">
13     </div>
14
15     <button class="btn btn-primary"
16         (click)="onDodajOcenu()">Dodaj ocenu</button>
17 </div>

```

Slično kao i direktiva `NgClass`, i direktiva `NgStyle` prihvata objekat kao vrednost, sa razlikom da svojstva tog objekta predstavljaju nazive CSS svojstava, dok su vrednosti ovih svojstava one vrednosti koje će biti iskorišćene za stilizovanje elementa. Tako metod `izracunajStil()` čija implementacija je data u nastavku, u zavisnosti od proseka studenta, bira boju i veličinu fonta, i odabrane vrednosti se postavljaju za vrednosti svojstava `color` i `font-size`:

Kod 8.41: angular/direktive1/src/app/ngstyle-test/ngstyle-test.component.ts (linije 47-76)

```

47     private izracunajStil() {
48         let colorChoice: string,
49             fontSizeChoice: string;
50
51         const prosek = this.prosek();
52         if (prosek >= 9) {
53             colorChoice = 'green';
54             fontSizeChoice = '20px';
55         } else if (prosek >= 8) {
56             colorChoice = 'blue';
57             fontSizeChoice = '19px';
58         } else if (prosek >= 7) {
59             colorChoice = 'cyan';
60             fontSizeChoice = '18px';
61         } else if (prosek >= 6) {
62             colorChoice = 'yellow';
63             fontSizeChoice = '17px';
64         } else if (prosek >= 5) {
65             colorChoice = 'red';
66             fontSizeChoice = '16px';
67         } else {
68             colorChoice = 'black';
69             fontSizeChoice = '15px';
70         }
71
72         return {
73             color: colorChoice,
74             'font-size': fontSizeChoice
75         }
76     }

```

8.6 Komponente i vezivanje podataka — napredniji koncepti

U sekciji 8.4 smo detaljno diskutovali o filozofiji razdvajanja aplikacije na komponente na kojoj počiva Angular okruženje i o njihovoj hijerarhijskoj organizaciji, kao i o tome kako se vrši kreiranje komponenti. Takođe, diskutovali smo o različitim Angular mehanizmima

kojima se upravlja podacima u okviru pogleda i modela u okviru jedne komponente. Ono sa čime ćemo se upoznati u ovoj sekciji jeste nastavak ove diskusije, samo u kontekstu više komponenti, odnosno, videćemo kako je moguće slati podatke između komponenti u okviru hijerarhije komponenti.

8.6.1 Tok podataka kroz hijerarhijsku organizaciju komponenti

Kao što smo rekli, Angular aplikacije se dele na veliki broj komponenti koje se mogu povezati u hijerarhijsku organizaciju. Deljenje podataka između ovih komponenti predstavlja jedan od osnovnih mehanizama koje iole korisna aplikacija mora da implementira. Srećom po nas, ukoliko smo razumeli kako vezivanje podataka funkcioniše u okviru jedne komponente, videćemo da tok podataka između dve komponente — konkretno, između roditeljske i dete-komponente — funkcioniše na identičnoj ideologiji, uz naravno, tek nešto složeniju implementaciju.

Da bismo ilustrovali tok podataka kroz roditeljsku i dete-komponentu, korišćićemo vrlo jednostavnu aplikaciju za upisivanje novih studenata. Aplikacija se sastoji od dva logička dela: formular za unos podataka o novom studentu i lista studenata koji su upisani. Samim tim, jedna moguća podela aplikacije na komponente je sledeća:

1. Korena komponenta sadrži komponentu 2 i listu komponenti 3.
2. Komponenta za kreiranje novih studenata.
3. Komponenta koja predstavlja jednog studenta.

Na slici 8.8 dat je prikaz početne aplikacije koja ima tek neke osnovne funkcionalnosti koje su opisane u nastavku. Takođe, na istoj slici su i obeležene komponente od kojih se sastoji aplikacija rednim brojem kojim su navedene iznad.

Upis studenata

1

2

Ime i prezime:

Smer:

Informatika

Upis novog studenta

Upisani studenti

3

TestimePrezime

Student TestimePrezime je upisao smer TestSmer datuma 28.6.2019. u 18:59h

Slika 8.8: Prikaz aplikacije za upisivanje novih studenata.

Inicijalno, korena komponenta sadrži naslovne elemente i po jednu instancu komponente za upisivanje novih studenata i jednu testnu instancu komponente koja predstavlja studenta. Sadržaj početne `app.component.html` datoteke dat u nastavku:

```
1 <div class="container">
2   <div class="row">
3     <div class="col-12">
```

```

4     <h1 class="display-1">Upis studenata</h1>
5     <app-kreator></app-kreator>
6     <hr>
7     <h3>Upisani studenti</h3>
8     <app-student></app-student>
9   </div>
10 </div>
11 </div>

```

Komponenta za upisivanje novih studenata je takođe, za sada, jednostavna i ne sadrži elemente koje do sada nismo videli:

Kod 8.42: angular/komponente2/src/app/kreator/kreator.component.html

```

1 <div>
2   <div class="form-group">
3     <label>Ime i prezime:</label>
4     <input type="text" class="form-control" [(ngModel)]="imePrezime">
5   </div>
6   <div class="form-group">
7     <label>Smer:</label>
8     <select class="form-control" (change)="onSelectedSmer($event)">
9       <option value="I">Informatika</option>
10      <option value="M">Matematika</option>
11      <option value="A">Astronomija</option>
12    </select>
13  </div>
14  <button class="btn btn-primary">Upis novog studenta</button>
15 </div>

```

Kod 8.43: angular/komponente2/src/app/kreator/kreator.component.ts

```

1 import { Component, OnInit } from '@angular/core';
2
3 @Component({
4   selector: 'app-kreator',
5   templateUrl: './kreator.component.html',
6   styleUrls: ['./kreator.component.css']
7 })
8 export class KreatorComponent implements OnInit {
9
10   public imePrezime: string = '';
11   public odabranSmer: string = 'I';
12
13   constructor() { }
14
15   ngOnInit() {
16   }
17
18   onSelectedSmer(event: Event): void {
19     this.odabranSmer = (<HTMLSelectElement>event.target).value;
20   }
21
22 }

```

Komponenta `StudentComponent` u svojoj suštini nije drugačija od onoga sa čime smo se susreli do tada, ali za razliku od prethodnih primera gde smo podatke strukturali *ad-hoc*, ovoga puta smo odlučili da kreiramo klasu `Student` koja će predstavljati *model podataka* (engl. *data model*). U tu svrhu, kreiranu klasu smo smestili na putanju `src/models/student`

`.model.ts`, kako bismo mogli jednostavnije da referišemo na nju iz bilo koje komponente³. Sama klasa je vrlo jednostavna, čija analiza se ostavlja čitaocu:

Kod 8.44: `angular/komponente2/src/models/student.model.ts`

```

1 export class PogresanSmerError extends Error {};
2
3 export class Student {
4     private imePrezime: string;
5     private smer: string;
6     private datumUpisa: string;
7
8     constructor(imePrezime: string, smer: string) {
9         this.imePrezime = imePrezime;
10        this.smer = Student.imeSmera(smer);
11        this.datumUpisa = Student.kreirajDatum();
12    }
13
14    public getImePrezime(): string {
15        return this.imePrezime;
16    }
17
18    public getSmer(): string {
19        return this.smer;
20    }
21
22    public getDatumUpisa(): string {
23        return this.datumUpisa;
24    }
25
26    public get inicijalSmera(): string {
27        return this.getSmer()[0];
28    }
29
30    private static kreirajDatum(): string {
31        const sada: Date = new Date(Date.now());
32        return `${sada.getDate()}.${sada.getMonth()+1}.${sada.getFullYear()}. u
33            ${sada.getHours()}:${sada.getMinutes()}h`;
34    }
35
36    private static imeSmera(skracenica: string): string {
37        switch (skracenica) {
38            case 'I':
39                return 'Informatika';
40            case 'M':
41                return 'Matematika';
42            case 'A':
43                return 'Astronomija';
44            case 'T':
45                return 'TestSmer';
46            default:
47                throw new PogresanSmerError(`Ne postoji skracenica ${skracenica}
48                    za smerove na Matematickom fakultetu!`);
49        }
50    }
51 }

```

³Naravno, samu datoteku koja sadrži definiciju ove klase smo mogli da smestimo bilo gde u okviru Angular projekta, ali činjenica da nam je model podataka odvojen od implementacije komponenti može biti korisna radi logičkog odvajanja implementacije komponenti od implementacije modela podataka.

Komponenta `StudentComponent` koja predstavlja jednog studenta sadrži jedno polje tipa `Student`, koje je na početku inicijalizovano sa testnim studentom, s obzirom da za sada nemamo način da kreiramo nove studente:

Kod 8.45: angular/komponente2/src/app/student/student.component.ts

```

1 import { Component, OnInit } from '@angular/core';
2 import { Student, PogresanSmerError } from '../../models/student.model';
3
4 @Component({
5   selector: 'app-student',
6   templateUrl: './student.component.html',
7   styleUrls: ['./student.component.css']
8 })
9 export class StudentComponent implements OnInit {
10
11   public student: Student = new Student('TestImePrezime', 'T');
12
13   constructor() { }
14
15   ngOnInit() {
16   }
17
18   dohvatiUrlSlike(): string {
19     switch (this.student.inicijalSmera) {
20       case 'I':
21         return 'assets/blue.png';
22       case 'M':
23         return 'assets/red.png';
24       case 'A':
25         return 'assets/green.png';
26       case 'T':
27         return 'assets/gray.png';
28       default:
29         throw new PogresanSmerError(`Nepoznat inicijal smer ${this.student.inicijalSmera}`);
30     }
31   }
32
33 }

```

Šablon ove komponente je takođe jednostavan, a dodatno prikazujemo i stil kojim se postiže izgled komponente kao na slici 8.8.

Kod 8.46: angular/komponente2/src/app/student/student.component.html

```

1 <div class="student mt-2 mb-1">
2   <div class="student-header">
3     <img [src]="dohvatiUrlSlike()" class="rounded mr-2" alt="Blue box">
4     <span class="Ime i prezime">{{ student.getImePrezime() }}</span>
5   </div>
6   <hr>
7   <div class="student-body">
8     Student {{ student.getImePrezime() }} je upisao smer {{ student.getSmer() }} datuma
9     {{ student.getDatumUpisa() }}
10   </div>
11 </div>

```

Kod 8.47: angular/komponente2/src/app/student/student.component.css

```

1 .student {

```

```

2     border: 1px solid rgb(155, 155, 155);
3     box-shadow: 0px 0px 2px 0px rgba(155, 155, 155, 0.75);
4     padding: 10px 20px;
5 }

```

Slanje podataka niz hijerarhiju

Pretpostavimo da smo odabrali da prvo rešimo problem u kojem `StudentComponent` dobija ispravan objekat tipa `Student` koji zatim prikazuje u svom šablonu. Ovo će nam rešiti problem prikazivanja jednog studenta u listi studenta.

Za početak, kreirajmo novu komponentu `Student2Component` koja inicijalno polazi od iste implementacije kao `StudentComponent`. Zatim, potrebno je izmeniti implementaciju `AppComponent` tako da čuva niz studenata koji je upisan do sada:

Kod 8.48: angular/komponente2/src/app/app.component.ts (linije 9-14)

```

9 export class AppComponent {
10     studenti: Array<Student> = [];
11
12     constructor() {
13         this.studenti.push(new Student('TestImePrezime', 'T'));
14     }

```

Zatim, u šablonu `AppComponent` potrebno je izmeniti kod tako da se umesto kreiranja jedne komponente `StudentComponent` koristi niz `studenti` za kreiranje više `Student2Component` objekata:

```

1 <!-- <app-student></app-student> -->
2 <app-student2 *ngFor="let student of studenti"></app-student2>

```

Ovom implementacijom očigledno kreiramo jednu instancu `Student2Component` za svaki element u nizu `studenti`. Takođe, u implementaciji `constructor` metoda `AppComponent` klase, inicijalno dodajemo jednu testnu instancu, kako bismo prikazali makar jednog studenta.

Međutim, i dalje nam ostaje problem što `Student2Component` koristi svoju instancu klase `Student`:

```

1 private student: Student = new Student('TestImePrezime', 'T');

```

Bilo bi korisno kada bismo mogli da povežemo polje `student` u klasi `Student2Component` sa elementom iz niza `studenti` pri svakoj iteraciji `ngFor` direktive. Zapravo, ispostavlja se da možemo, i upravo ovaj mehanizam toka podataka predstavlja slanje podataka niz hijerarhiju komponenti — podatak iz roditeljske komponente (`AppComponent`) se šalje dete-komponenti (`Student2Component`). Ovaj smer slanja podataka se često naziva i *ulazni tok* (engl. *input*), s obzirom da komponenta na ovaj način dobija podatke kojima može da upravlja.



Obratiti pažnju da se prilikom slanja podataka kroz hijerarhiju šalju reference objekata, te svaka izmena poslatog objekta u dete-komponenti biće oslikana i u roditeljskoj komponenti. Prisetimo se da je ovo osobina jezika JavaScript, a ne Angular razvojnog okruženja.

Sada kada smo razumeli šta je potrebno da uradimo, hajde da pogledamo sintaksu kojom se ovo postiže. Da bismo omogućili slanje podataka niz hijerarhiju, potrebno je da uradimo dve stvari:

1. U modelu dete-komponente kreiramo svojstvo odgovarajućeg tipa objekta koji se prenosi iz roditeljske komponente i dekoriramo ga `@Input` dekoratorom.
2. U šablonu roditeljske komponente koristimo sintaksu za vezivanje atributa nad elementom koje predstavlja dete-komponentu kojoj se šalje podatak.

Pogledajmo za svaki korak prvo implementaciju, pa ćemo onda diskutovati detalje. Što se tiče prvog koraka, izmena koja je načinjena data je narednim fragmentom koda:

Kod 8.49: angular/komponente2/src/app/student2/student2.component.ts (linije 9-12)

```

9 export class Student2Component implements OnInit {
10
11     @Input('studentData')
12     public student: Student;
```

Kreirali smo svojstvo `student` klase `Student2Component`, isto kao što je to bio slučaj u klasi `StudentComponent`, ali sa nekoliko značajnih razlika. Prvo, svojstvo je definisano uz javni modifikator pristupa da bi roditeljska komponenta mogla da mu pristupi. Drugo, koristimo dekorator `@Input` (iz paketa `@angular/core`) da bismo rekli Angular-u da roditeljska komponenta može dodeliti vrednost ovom svojstvu. Treće, svojstvo `student` je na početku nedefinisano, što znači da je njegova inicijalna vrednost `undefined`, sve dok mu roditeljska komponenta ne postavi vrednost.

Ono što dodatno primećujemo jeste da dekorator `@Input` prihvata argument. Ovaj argument je opcioni i njime se može definisati naziv ovog svojstva koje roditeljska komponenta vidi. Ukoliko se ne navede, onda će roditeljska komponenta vezivati atribut `student` jer je to naziv svojstva. S obzirom da smo naveli argument `'studentData'`, onda roditeljska komponenta mora da veže atribut `studentData`.

Sada diskutujemo o vezivanju atributa u šablonu roditeljske klase:

```

1 <app-student2 *ngFor="let student of studenti"
2   [studentData]="student"></app-student2>
```

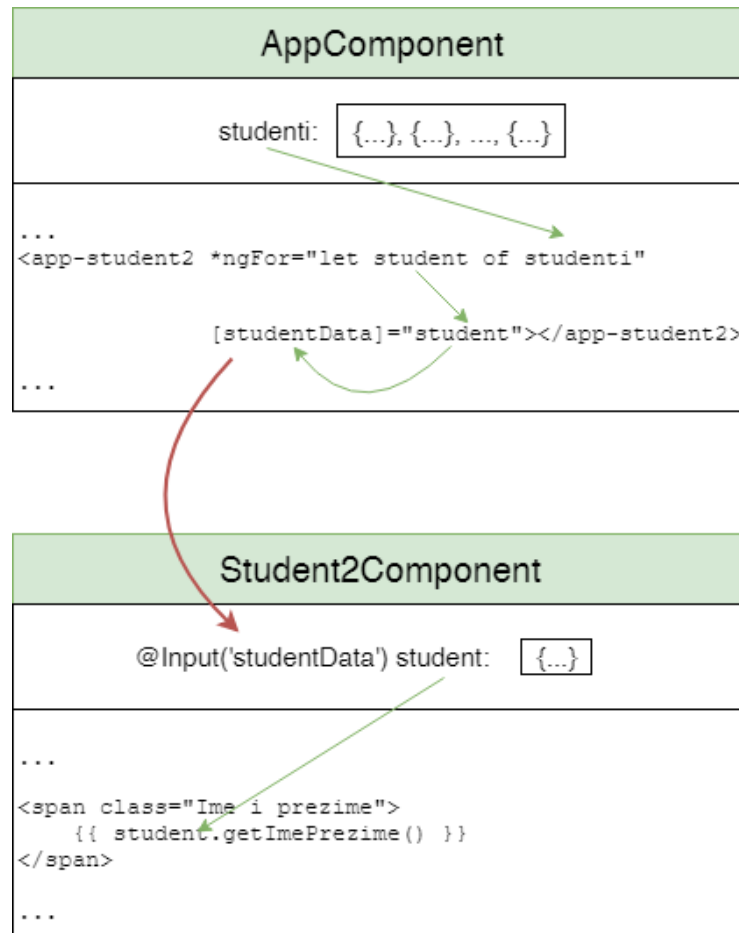
Primetimo da se sintaksa slanja podatka iz roditeljske komponente svodi na vezivanje atributa sa kojom smo se upoznali u sekciji 8.4. Atribut koji se vezuje je svojstvo iz dete-komponente, a njegova vrednost je šablonski izraz koji se izračunava na objekat odgovarajućeg tipa. U ovom slučaju, koristi se element iz niza `studenti` kroz koji se iterira. Primetimo da atribut vezujemo preko imena koje smo dodelili kao argument dekoratora `@Input`.

Ulazni tok se može prikazati grafički i jedan detaljniji prikaz, ilustrovan na opisanom primeru, prikazan je na slici 8.9. Primetimo da zelene strelice predstavljaju tok podataka u okviru komponente, dok je crvenom strelicom prikazan tok podataka između roditeljske i dete-komponente.

Slanje podataka uz hijerarhiju

Sada želimo da kompletiramo aplikaciju tako što ćemo implementirati kreiranje objekta tipa `Student` u `KreatorComponent`, koji će zatim biti prosleđen `AppComponent` kako bi bio smešten u niz `studenti` u okviru te komponente. Ovaj smer slanja podataka se često naziva i *izlazni tok* (engl. *output*), s obzirom da komponenta na ovaj način prosleđuje podatke drugim komponentama.

Kreirajmo komponentu `Kreator2Component` koja polazi od identične implementacije kao i `KreatorComponent`. Ne zaboravimo da izmenimo šablon `AppComponent` tako da koristi ovu komponentu:



Slika 8.9: Prikaz ulaznog toka podataka od roditeljske komponente **AppComponent** ka dete-komponenti **Student2Component**.

```
1 <!-- <app-kreator></app-kreator> -->
2 <app-kreator2></app-kreator2>
```

S obzirom da su polja formulara u **Kreator2Component** povezana sa svojstvima klase u njenom modelu, svaki put kada korisnik izvrši odgovarajuće promene u formularu, ažuriraju se odgovarajuća svojstva u klasi. To znači da je za kreiranje novog studenta dovoljno izvršiti metod koji će se pozvati klikom na dugme "Upis novog studenta":

Kod 8.50: angular/komponente2/src/app/kreator2/kreator2.component.html (linije 14-14)

```
14 <button class="btn btn-primary" (click)="onUpisStudenta()">Upis novog
    studenta</button>
```

Očigledno, potrebno je implementirati metod `onUpisStudenta()` u modelu komponente koji će, na osnovu svojstava, kreirati novi objekat klase **Student** (naredna implementacija metoda je nepotpuna sa razlogom):

Kod 8.51: angular/komponente2/src/app/kreator2/kreator2.component.ts (linije 26-31)

```
26 onUpisStudenta(): void {
27     if (this.imePrezime === '') {
28         window.alert('Molimo unesite ime i prezime novog studenta!');
```

```

29         return;
30     }
31     const noviStudent = new Student(this.imePrezime, this.odabranSmer);

```

Međutim, postavlja se pitanje kako sada da kreiranu instancu `noviStudent` prosledimo komponenti `AppComponent`, da bi ona mogla da je sačuva u nizu `studenti`? Sada nam se javlja potreba za slanjem podataka uz hijerarhiju komponenti, i da bismo to uradili, potrebno je da implementiramo dva koraka:

1. U modelu dete-komponente kreiramo svojstvo generičkog tipa `EventEmitter<T>` (iz paketa `@angular/core`) koje se instancira i dekoriramo ga `@Output` dekoratorom. Kada želimo da pošaljemo podatak tipa `T`, nad kreiranim svojstvom pozivamo metod `emit()` čiji je argument podatak koji se šalje.
2. U šablonu roditeljske komponente koristimo sintaksu za vezivanje događaja nad elementom koji predstavlja dete-komponentu od koje se dobija podatak. Podatak dobijamo tako što implementiramo metod u modelu roditeljske komponente koji reaguje na događaj, a koji kao argument prima specijalni objekat `$event`.

Iako su ovi koraci nešto deskriptivniji u odnosu na slanje podataka niz hijerarhiju, videćemo da nam je sintaksa koja se koristi već poznata, uz tek nešto složeniju implementaciju. Pogledajmo za svaki korak prvo implementaciju, pa ćemo onda diskutovati detalje. Što se tiče prvog koraka, implementacija se sastoji od dva dela. Prvi deo podrazumeva kreiranje svojstva u modelu dete-komponente:

Kod 8.52: angular/komponente2/src/app/kreator2/kreator2.component.ts (linije 14-15)

```

14     @Output('noviStudent')
15     public emitNoviStudent: EventEmitter<Student> = new EventEmitter<Student>()
    ;

```

Kao što vidimo, kreirano svojstvo je definisano javnim modifikatorom pristupa, da bi roditeljska komponenta mogla da mu pristupi. Dodatno, dekorator `@Output` ima slično ponašanje kao i dekorator `@Input`, samo je njegova logika inverzna — svojstvo koje on dekorira se koristi kao izlaz iz komponente umesto kao ulaz. Njegovim opcionim argumentom se može sepcifikovati naziv događaja na koji roditeljska komponenta može da reaguje postavljanjem osluškivača.

Drugi deo podrazumeva slanje podatka roditeljskoj komponenti korišćenjem ovog svojstva (dajemo celu implementaciju metoda `onUpisStudenta()` radi kompletnosti, ali akcenat stavljamo na korišćenje kreiranog svojstva `emitNoviStudent`):

Kod 8.53: angular/komponente2/src/app/kreator2/kreator2.component.ts (linije 26-34)

```

26     onUpisStudenta(): void {
27         if (this.imePrezime === '') {
28             window.alert('Molimo unesite ime i prezime novog studenta!');
29             return;
30         }
31         const noviStudent = new Student(this.imePrezime, this.odabranSmer);
32         this.emitNoviStudent.emit(noviStudent);
33         this.imePrezime = '';
34     }

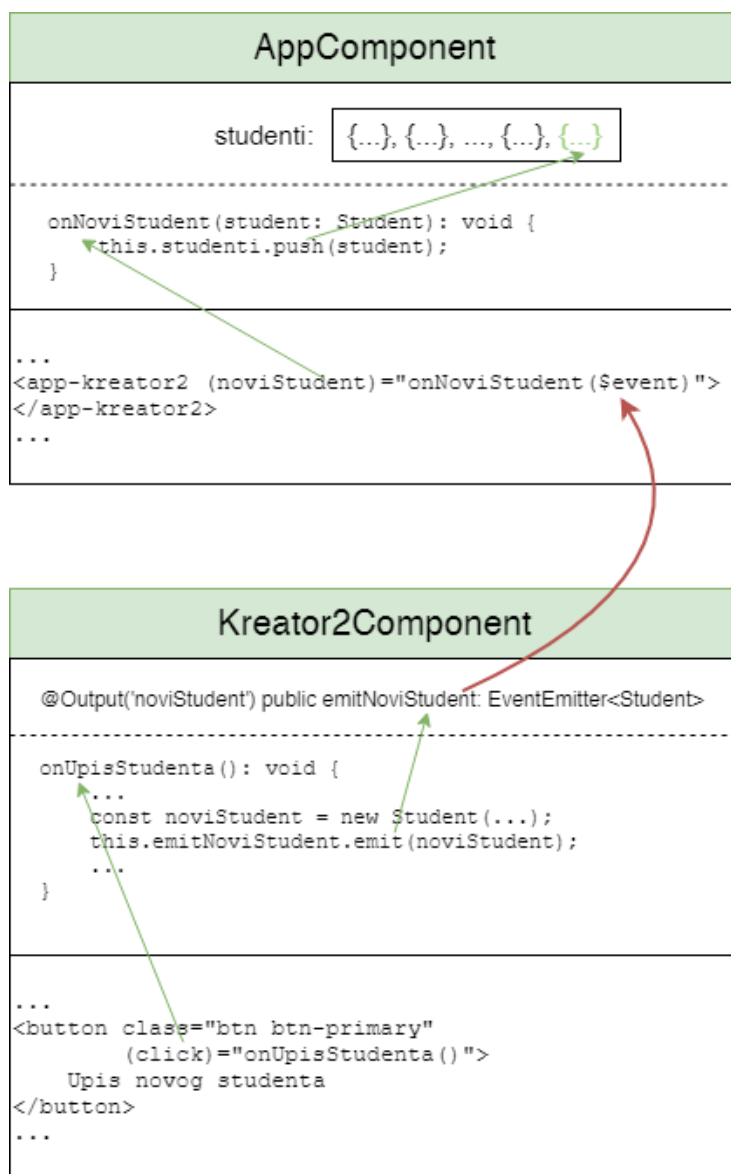
```

Pogledajmo sada implementaciju drugog koraka. Prvo, u šablonu roditeljske komponente postavljamo vezivanje događaja.

```

1 <app-kreator2 (noviStudent)="onNoviStudent($event)"></app-kreator2>

```



Slika 8.10: Prikaz izlaznog toka podataka od dete-komponente **Kreator2Component** ka roditeljskoj komponenti **AppComponent**.

Kao što vidimo, sintaksa koja se koristi je identična sintaksi vezivanja događaja. Naziv događaja potiče od argumenta `@Output` dekoratora u dete-komponenti, odnosno, naziva dekorisanog svojstva ukoliko dekoratoru nije naveden argument. Kao i kod vezivanja događaja, šablonski izraz gotovo uvek predstavlja poziv `on*` metoda čija se implementacija nalazi u modelu komponente. Podatak koji dete-komponenta šalje se u roditeljskoj komponenti vidi kroz specijalni objekat `$event`. Razlika između `$event` objekta u vezivanju događaja DOM stabla i slanju podataka uz hijerarhiju je u tome što je `$event` objekat u ovom slučaju onog tipa koji je tip podatka koji se šalje, što se vidi u potpisu metoda `onNoviStudent()`:

Kod 8.54: angular/komponente2/src/app/app.component.ts (linije 16-18)

```

16  onNoviStudent(student: Student): void {
17    this.studenti.push(student);
18  }

```

Kao što vidimo, njegov prvi argument je tipa `Student`, što je upravo tip podataka koji dete-komponenta šalje. Kao što smo rekli na samom početku ovog dela teksta, cilj nam je bio da smestimo kreiranog studenta u niz `studenti`, otuda je implementacija metoda `onNoviStudent()` jednostavna.



Kao i u slučaju slanja podataka niz hijerarhiju, potrebno je obratiti pažnju da se šalju reference objekata, a ne njihove kopije.

Ulazni tok se može prikazati grafički i jedan detaljniji prikaz, ilustrovan na opisanom primeru, prikazan je na slici 8.10. Primetimo da zelene strelice predstavljaju tok podataka u okviru komponente, dok je crvenom strelicom prikazan tok podataka između dete-komponente i roditeljske komponente.

8.6.2 Referencne promenljive šablona

U delu 8.5.1 kratko smo se osvrnuli na postojanje tzv. referencnih promenljivih šablona u kontekstu `else` klauze `ngIf` direktive. Tamo smo takođe videli da se referencne promenljive šablona uvode znakom `#` koji prati validan JavaScript identifikator, na primer, `#mojaPromenljiva`, dok se njihovo referenciranje izvodi navođenjem samo identifikatora, na primer, `mojaPromenljiva`.

Međutim, ove promenljive imaju i nešto opštiju upotrebnu vrednost. Zapravo, referencnu promenljivu šablona je moguće pridružiti proizvoljnom elementu šablona i nakon toga je moguće referisati na taj element bilo gde u okviru tog šablona. Na primer, moguće je redefinisati metod `onUpisStudenta()` u okviru modela `Kreator3Component` tako da kao argument prima reference na HTML elemente iz formulara, da bi zatim očitao njihove vrednosti:

Kod 8.55: `angular/komponente2/src/app/kreator3/kreator3.component.html`

```

1 <div>
2   <div class="form-group">
3     <label>Ime i prezime:</label>
4     <input type="text" class="form-control" #imePrezimeInput>
5   </div>
6   <div class="form-group">
7     <label>Smer:</label>
8     <select class="form-control" #smerSelect>
9       <option value="I">Informatika</option>
10      <option value="M">Matematika</option>
11      <option value="A">Astronomija</option>
12    </select>
13  </div>
14  <button class="btn btn-primary" (click)="onUpisStudenta(imePrezimeInput,
15    smerSelect)">Upis novog studenta</button>
  </div>

```

Primetimo da se u šablonu više ne koristi dvosmerno vezivanje za `<input>` polje i reagovanje na promene izbora `<select>` polja. Samim tim nisu nam neophodna odgovarajuća svojstva i metodi u modelu komponente s obzirom da ćemo te vrednosti očitavati prilikom upisa studenta, korišćenjem referenci na te elemente:

Kod 8.56: `angular/komponente2/src/app/kreator3/kreator3.component.ts`

```

1 import { Component, OnInit, EventEmitter, Output } from '@angular/core';
2 import { Student } from '../models/student.model';

```

```

3
4 @Component({
5   selector: 'app-kreator3',
6   templateUrl: './kreator3.component.html',
7   styleUrls: ['./kreator3.component.css']
8 })
9 export class Kreator3Component implements OnInit {
10
11   @Output('noviStudent')
12   public emitNoviStudent: EventEmitter<Student> = new EventEmitter<Student>();
13
14   constructor() { }
15
16   ngOnInit() {
17   }
18
19   onUpisStudenta(imePrezimeInput: HTMLInputElement, odabranSmerSelect:
20     HTMLSelectElement): void {
21     const imePrezime: string = imePrezimeInput.value;
22     const odabranSmer: string = odabranSmerSelect.value;
23
24     if (imePrezime === '') {
25       window.alert('Molimo unesite ime i prezime novog studenta!');
26       return;
27     }
28
29     const noviStudent = new Student(imePrezime, odabranSmer);
30     this.emitNoviStudent.emit(noviStudent);
31
32     // Ne preporučuje se menjanje DOM stabla na ovaj način
33     imePrezimeInput.value = '';
34   }
35 }

```

8.6.3 Dekorator @ViewChild

Nekada nam nije dovoljno da na elemente referišemo samo u okviru šablona već i u okviru modela. Ispostavlja se da je moguće referišati na proizvoljan element iz pogleda komponente u okviru implementacije modela te komponente korišćenjem referencnih promenljivih šablona i dekoratora `@ViewChild`.

Dekorator `@ViewChild` dekoriše svojstvo modela komponente čiji je tip `ElementRef` (iz paketa `@angular/core`). U pitanju je specijalni tip podataka koji sadrži svojstvo `nativeElement` koje sadrži referencu na element iz pogleda te komponente.

Dekorator funkcioniše tako što na osnovu prvog argumenta koji predstavlja upit, dohvaća prvi element iz rezultata tog upita. Pod upitom se najčešće smatra naziv referencne promenljive šablona (zadat kao niska) ili tip komponente.

Drugi argument predstavlja objekat kojim se postavljaju dodatne opcije. Obavezna opcija je `'static'` tipa `boolean` čija je vrednost `true` ili `false` ukoliko jeste, odnosno, nije potrebno izvršiti upit pre ciklusa detekcije promene, redom⁴.

U suštini, ukoliko nema potrebe da nam je rezultat upita dostupan prilikom inicijalizacije komponente (odnosno, prilikom izvršavanja `ngOnInit` metoda, videti narednu podsekciju),

⁴U verziji Angular 8, ova opcija mora da se navede, dok će u verziji Angular 9 biti korišćeno automatsko zaključivanje ove opcije na osnovu rezultata upita.

onda možemo koristiti `{static: false}`. U suprotnom, rezultat upita nam je dostupan nakon inicijalizacije pogleda (odnosno, nakon poziva `ngAfterViewInit()` metoda, videti narednu podsekciju).

Kod 8.57: angular/komponente2/src/app/kreator4/kreator4.component.html

```

1 <div>
2   <div class="form-group">
3     <label>Ime i prezime:</label>
4     <input type="text" class="form-control" #imePrezimeInput>
5   </div>
6   <div class="form-group">
7     <label>Smer:</label>
8     <select class="form-control" #smerSelect>
9       <option value="I">Informatika</option>
10      <option value="M">Matematika</option>
11      <option value="A">Astronomija</option>
12    </select>
13  </div>
14  <button class="btn btn-primary" (click)="onUpisStudenta()">Upis novog
    studenta</button>
15 </div>

```

Kod 8.58: angular/komponente2/src/app/kreator4/kreator4.component.ts

```

1 import { Component, OnInit, EventEmitter, Output, ViewChild, ElementRef } from
  '@angular/core';
2 import { Student } from 'src/models/student.model';
3
4 @Component({
5   selector: 'app-kreator4',
6   templateUrl: './kreator4.component.html',
7   styleUrls: ['./kreator4.component.css']
8 })
9 export class Kreator4Component implements OnInit {
10
11   @Output('noviStudent')
12   public emitNoviStudent: EventEmitter<Student> = new EventEmitter<Student>();
13
14   @ViewChild('imePrezimeInput', {static: false})
15   private imePrezimeInput: ElementRef;
16
17   @ViewChild('smerSelect', {static: false})
18   private odabranSmerSelect: ElementRef;
19
20   constructor() { }
21
22   ngOnInit() {
23   }
24
25   onUpisStudenta(): void {
26     const imePrezime: string = (<HTMLInputElement>this.imePrezimeInput.
      nativeElement).value;
27     const odabranSmer: string = (<HTMLSelectElement>this.odabranSmerSelect.
      nativeElement).value;
28
29     if (imePrezime === '') {
30       window.alert('Molimo unesite ime i prezime novog studenta!');
31       return;
32     }
33
34     const noviStudent = new Student(imePrezime, odabranSmer);

```

```

35     this.emitNoviStudent.emit(noviStudent);
36
37     // Ne preporučuje se menjanje DOM stabla na ovaj način
38     (<HTMLInputElement>this.imePrezimeInput.nativeElement).value = '';
39 }
40
41 }

```

8.6.4 Element `<ng-content>` i dekorator `@ContentChild`

8.6.5 Životni tok komponenti i metodi za osluškivanje događaja iz životnog toka

Svaka komponenta ima svoj *životni tok* (engl. *lifecycle*) koji se održava od strane Angulara. U toku ovog životnog toka, Angular vrši kreiranje komponenti, njihovo prikazivanje u okviru DOM stabla, upravlja izmenama i reagovanjima na događaje i uništava ih pre uklanjanja iz DOM stabla.

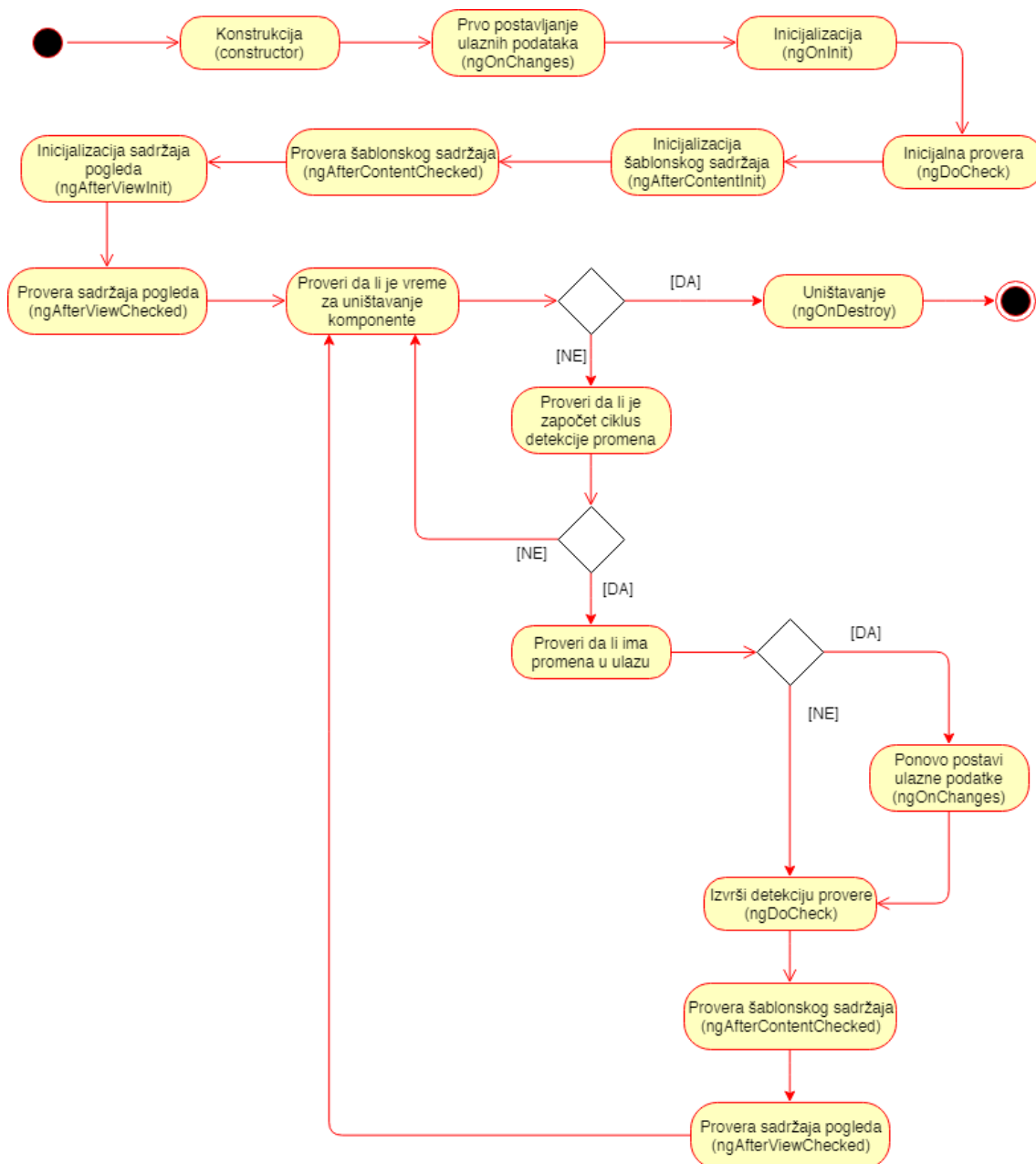
S obzirom da svi ovi događaji (u širem smislu) mogu biti korisni za programera, Angular nam omogućuje da postavimo odgovarajuće osluškivače koji će se izvršiti u specifičnim trenucima. Da bismo mogli da postavimo odgovarajuće osluškivače, potrebno je da klasa iz modela komponente implementira odgovarajuće interfejsa, a zatim da implementira odgovarajuće metode. Na primer, do sada smo nekoliko puta videli interfejs `OnInit` kao i metod `ngOnInit`. Ovaj metod Angular poziva nedugo nakon kreiranja komponente.

Na slici 8.11 detaljno je ilustrovan životni tok jedne komponente dijagramom aktivnosti. Svaka aktivnost za koju je moguće postaviti osluškivač sadrži naziv osluškivača u zagradama.

U nastavku slede opisi za svaki osluškivač koji je dostupan:

- `ngOnChanges()` — Respond when Angular (re)sets data-bound input properties. The method receives a `SimpleChanges` object of current and previous property values. Called before `ngOnInit()` and whenever one or more data-bound input properties change.
- `ngOnInit()` — Initialize the directive/component after Angular first displays the data-bound properties and sets the directive/component's input properties. Called once, after the first `ngOnChanges()`.
- `ngDoCheck()` — Detect and act upon changes that Angular can't or won't detect on its own. Called during every change detection run, immediately after `ngOnChanges()` and `ngOnInit()`.
- `ngAfterContentInit()` — Respond after Angular projects external content into the component's view / the view that a directive is in. Called once after the first `ngDoCheck()`.
- `ngAfterContentChecked()` — Respond after Angular checks the content projected into the directive/component. Called after the `ngAfterContentInit()` and every subsequent `ngDoCheck()`.
- `ngAfterViewInit()` — Respond after Angular initializes the component's views and child views / the view that a directive is in. Called once after the first `ngAfterContentChecked()`.
- `ngAfterViewChecked()` — Respond after Angular checks the component's views and child views / the view that a directive is in. Called after the `ngAfterViewInit()` and every subsequent `ngAfterContentChecked()`.
- `ngOnDestroy()` — Cleanup just before Angular destroys the directive/component. Unsubscribe Observables and detach event handlers to avoid memory leaks. Called

just before Angular destroys the directive/component.



Slika 8.11: Dijagram aktivnosti koji prikazuje životni tok jedne komponente.

Sada ćemo nadograditi prethodnu implementaciju aplikacije tako da se za svakog studenta u konzoli ispisuje informacija prilikom izvršavanja nekih događaja životnog toka. Pre toga, kreirajmo dugme kojim se briše prvi upisani student iz niza `studenti` i implementirajmo odgovarajući metod:

Kod 8.59: angular/komponente2/src/app/app.component.html (linije 10-15)

```

10 <button class="btn btn-danger" (click)="onUkloniPrvogStudenta()">Ukloni
    prvog studenta sa spiska</button>
11 <hr>
12 <h3>Upisani studenti</h3>
  
```



```

13     <!-- <app-student></app-student> -->
14     <!-- <app-student2 *ngFor="let student of studenti" [studentData]="
15         student"></app-student2> -->
16     <app-student3 *ngFor="let student of studenti" [studentData]="student"></
17         app-student3>

```

Kod 8.60: angular/komponente2/src/app/app.component.ts (linije 20-22)

```

20 onUkloniPrvogStudenta(): void {
21     this.studenti.splice(0, 1);
22 }

```

Ovoga puta koristimo komponentu `Student3Component` u kojoj su implementirani neki od osluškivača:

Kod 8.61: angular/komponente2/src/app/student3/student3.component.ts

```

1 import { Component, OnInit, Input, OnChanges, SimpleChanges, DoCheck, OnDestroy
    } from '@angular/core';
2 import { Student, PogresanSmerError } from '../../models/student.model';
3
4 @Component({
5     selector: 'app-student3',
6     templateUrl: './student3.component.html',
7     styleUrls: ['./student3.component.css']
8 })
9 export class Student3Component implements OnInit, OnChanges, DoCheck, OnDestroy
    {
10
11     @Input('studentData')
12     public student: Student;
13
14     constructor() {
15         console.log(`Upisali smo novog studenta na MATF ${this.student ===
16             undefined ? 'čije ime nam je nepoznato još uvek' : this.student.
17             getImePrezime()}! (constructor)`);
18     }
19
20     ngOnInit() {
21         console.log(`Inicijalizacija studenta ${this.student === undefined ? 'čije
22             ime nam je nepoznato još uvek' : this.student.getImePrezime()}! (
23             ngOnInit)`);
24     }
25
26     ngOnChanges(simpleChanges: SimpleChanges): void {
27         console.log('Promene:\n', simpleChanges, '\n', '(ngOnChanges)');
28     }
29
30     ngDoCheck(): void {
31         console.log('Doslo je do ciklusa detekcije promene! (ngDoCheck)');
32     }
33
34     ngOnDestroy(): void {
35         console.log(`Student ${this.student.getImePrezime()} je diplomirao :) (
36             ngOnDestroy)`);
37     }
38
39     dohvatiUrlSlike(): string {
40         switch (this.student.inicijalSmera) {
41             case 'I':
42                 return 'assets/blue.png';
43         }
44     }
45 }

```

```

38     case 'M':
39         return 'assets/red.png';
40     case 'A':
41         return 'assets/green.png';
42     case 'T':
43         return 'assets/gray.png';
44     default:
45         throw new PogresanSmerError(`Nepoznat inicijal smer a ${this.student.
            inicijalSmera}`);
46     }
47 }
48
49 onClickni(): void {
50     console.log('Kliknuto je dugme u Student3Component!');
51 }
52
53 }

```

Tako, na primer, pri kreiranju novog studenta možemo videti da se svi ulazni podaci (oni koji su dekorisani dekoratorom `@Input`) postavljaju prilikom inicijalizacije komponente, a ne prilikom njene konstrukcije:

```

Upisali smo novog studenta na MATF čije ime nam je nepoznato još uvek! (
    constructor)
Promene:
    {student: SimpleChange}
    (ngOnChanges)
Inicijalizacija studenta Miloš! (ngOnInit)
Doslo je do ciklusa detekcije promene! (ngDoCheck)

```

Klikom na dugme "Klikni me!", Angular će odreagovati i započeti ciklus detekcije promena:

```

Kliknuto je dugme u Student3Component!
Doslo je do ciklusa detekcije promene! (ngDoCheck)

```

Klikom na dugme "Ukloni prvog studenta sa spiska", poziva se metod koji uklanja objekat iz niza, te samim tim Angular zaključuje da je potrebno uništiti komponentu koja je bila vezana za njega:

```

Student Miloš je diplomirao :) (ngOnDestroy)

```

Zadatak 8.2 Dopuniti klasu `Student3Component`:

1. Implementirati preostale metoda za osluškivanje, pa analizirati redosled pozivanja ovih metoda prilikom kreiranja nove instance komponente.
2. Dohvatiti dva elementa iz pogleda pomoću referencnih promenljiva šablona i dekoratora `@ViewChild`, pri čemu se jedan element dohvata statički (`{ static: true }`), a drugi ne, pa ispitati u kom trenutku (u kojoj metodi za osluškivanje događaja životnog toka) ove reference bivaju razrešene.
3. Izdvojiti deo pogleda u šablon, pa uraditi prethodnu tačku sa `@ContentChild` dekoratorom.
4. Implementirati metod u klasi `Student` koji nasumično menja smer studenta, pa dodati dugme u komponenti `Student3Component` i implementirati metod koji klikom na to dugme nasumično menja svojstvo `student` tako što mu nasumično menja smer. Analizirati pozive osluškivača događaja životnog toka komponente i vrednosti koje se menjaju (tj. analizirati objekat `SimpleChanges`). Kako se niz

poziva osluškivača u ovom slučaju razlikuje od poziva osluškivača koji se dobija klikom na dugme "Klikni me!"?

8.7 Kreiranje direktiva

8.8 Mehanizam servisa i ubrizgavanje zavisnosti

Elementi Angular radnog okvira koji slede u nastavku biće demonstrirani kroz razvoj jednostavne aplikacije koja sadrži katalog proizvoda, korpu za onlajn kupovinu i formular za naručivanje odabranih proizvoda.

Započnimo sekciju kreiranjem novog projekta:

```
ng new angular-store --skipTests --routing --style=css
```

Kreirajmo komponente koje će predstavljati proizvode:

```
ng generate component product-list
```

Ova komponenta će prikazivati podatke o proizvodima:

```
1 <h2>Products</h2>
2 <hr>
3
4 <div *ngFor="let product of products">
5   <h4>{{ product.name }}</h4>
6   <p>Price: {{ product.price }}</p>
7 </div>
```

Iz ovog koda vidimo da se podaci nalaze u nizu `products`, koji predstavlja članicu odgovarajuće klase za dati šablon, tj. `ProductListComponent`. Kako možemo implementirati ove podatke? Radi jednostavnosti, mi ćemo podatke čuvati na klijentu, mada se u praktičnim primenama podaci, naravno, čuvaju na serveru i potrebno ih je dohvatiti putem nekog veb protokola, na primer, HTTP.

8.8.1 Kreiranje servisa

Da bismo dohvatili podatke, potrebno je da logiku izdvojimo u posebnu klasu koju ćemo zvati *servis* (engl. *service*). Suštinski, postoje dve vrste servisa koji se razlikuju po svojoj svrsi:

- *Servis podataka* ima za cilj upravljanje nad nekim podacima. Na primer: kontaktiranje nekog servera za dohvaćanje, skladištenje, ažuriranje ili brisanje podataka, izračunavanje podataka na osnovu nekakvih ulaza, i sl.
- *Servis usluga* ima za cilj opsluživanje raznih usluga koje su neophodne za rad jedne komponente ili više njih. Na primer: ostvarivanje komunikacije između dve komponente, *upisivanje u dnevnik* (engl. *logging*), pozivanje funkcionalnosti biblioteka trećih lica, i sl.

Suštinski, bez obzira na vrstu servisa, svi oni se implementiraju na isti način u Angular radnom okviru. Servise možemo generisati pomoću komande `ng generate service` koji će kreirati odgovarajuću datoteku sa početnom implementacijom.

Kreirajmo servis koji će služiti za dohvaćanje podataka o proizvodima u našoj prodavnici:

```
ng generate service services/product
```

Rezultat je naredni kod u datoteci `product.service.ts`:

```

1 import { Injectable } from '@angular/core';
2
3 @Injectable({
4   providedIn: 'root'
5 })
6 export class ProductService {
7
8   constructor() {
9   }
10 }
```

8.8.2 Ubrizgavanje zavisnosti i korišćenje servisa u komponentama

Ono što primećujemo jeste da su servisi dekorisani dekoratorom klase `@Injectable`. Ovaj dekorator omogućava da se servisi koriste u našim komponentama pomoću specijalnog koncepta koji se naziva *ubrizgavanje zavisnosti* (engl. *dependency injection*). Dopunimo implementaciju servisa pre nego što diskutujemo više o ovom konceptu:

Kod 8.62: angular/store/src/app/services/product.service.ts

```

1 import { Injectable } from '@angular/core';
2 import { ProductModel } from '../models/product.model';
3
4 @Injectable({
5   providedIn: 'root'
6 })
7 export class ProductService {
8   private products: ProductModel[];
9
10  constructor() {
11    this.products = [
12      new ProductModel(101, 'Phone XL', 799, 'A large phone with one of the
13        best screens'),
14      new ProductModel(102, 'Phone Standard', 699, 'A great phone with one of
15        the best cameras'),
16      new ProductModel(103, 'Phone Mini', 299, ''),
17    ];
18  }
19
20  public getProducts(): ProductModel[] {
21    return this.products;
22  }
23 }
```

Vidimo da servis sadrži instancu niza unapred popunjenih objekata koji predstavljaju proizvode. Kada budemo govorili o komunikaciji između Angular klijentskih aplikacija i serverskih aplikacija, tada ćemo intenzivnije govoriti o raznim strategijama za dohvaćanje podataka i njihovo upravljanje. Radi kompletnosti, prikažimo i klasu `ProductModel` koja predstavlja šablon za proizvode u kodu:

Kod 8.63: angular/store/src/app/models/product.model.ts

```

1 export class ProductModel {
2   constructor(public productId: number,
3     public name: string,
4     public price: number,
5     public description: string) {
6   }
7 }
```

Kako se naš napisani servis za rad sa proizvodima sada koristi? Odgovor je vrlo jednostavan — sve što je potrebno uraditi jeste ubrizgati taj servis u komponentu koja ga koristi kao argument konstruktora, kao što je to urađeno u komponenti koja predstavlja podatke kao listu proizvoda:

Kod 8.64: angular/store/src/app/product-list/product-list.component.ts

```

1 import { Component, OnInit } from '@angular/core';
2 import { ProductService } from '../services/product.service';
3 import { ProductModel } from '../models/product.model';
4
5 @Component({
6   selector: 'app-product-list',
7   templateUrl: './product-list.component.html',
8   styleUrls: ['./product-list.component.css']
9 })
10 export class ProductListComponent implements OnInit {
11
12   public products: ProductModel[];
13
14   constructor(private productService: ProductService) {
15     this.products = this.productService.getProducts();
16   }
17
18   ngOnInit() {
19   }
20
21 }
```

Prilikom instanciranja ove klase, Angular će primetiti da je neophodno da prosledi instancu `ProductService` kao argument konstruktoru date klase. S obzirom da argumentu konstruktora prethodni modifikator pristupa, u ovom slučaju će biti kreirana privatna promenljiva instance `productService` koja će biti inicijalizovata prosleđenom instancom, i samim tim će biti dostupna našoj komponenti bilo gde u njenoj definiciji.

Validno je postaviti pitanje — koliko instanci servisa `ProductService` će Angular kreirati u slučaju da, na primer, kreira više od jedne instance komponente `ProductListComponent`? S obzirom da smo u definiciji servisa naveli da je on dostupan na nivou cele aplikacije (prosleđivanjem opcije `providedIn` čija je vrednost `'root'`), to znači da će Angular kreirati tačno jednu instancu servisa koja će biti deljena svim komponentama koje zahtevaju ovaj servis u aplikaciji, čime se ostvaruje da postoji tačno jedna instanca naših podataka, umesto da svaka komponenta sadrži posebnu instancu tih podataka i time nepotrebno povećava količinu memorije koje aplikacija troši. Upravo ovaj koncept predstavlja prethodno pomenuto *ubrizgavanje zavisnosti* i važno ga je dobro razumeti.

U ovom delu teksta smo prikazali samo osnovne koncepte servisa u Angular radnom okviru. Postoji još veliki broj koncepata o kojima nismo diskutovali, kao što su: detalji koncepta ubrizgavanja zavisnosti, upravljanje instancama servisa, hijerarhijski ubrzgavači servisa, fabrike servisa, itd.

8.9 Rutiranje

Aplikacija koju smo za sada kreirali ima mogućnost samo da prikaže informacije o proizvodima i to se izvršava na početnoj stranici (preciznije, na početnom URL-u, na primer: `http://localhost:4200/`). Želeli bismo da proširimo našu aplikaciju tako da ispunjava još neke mogućnosti:

- Svaki proizvod u listi treba da sadrži dugme koje će otvoriti stranicu na kojoj će biti prikazane ukupne informacije o tom proizvodu i na kojem će on moći da bude kupljen.
- Potrebno je implementirati stranicu na kojoj se prikazuju proizvodi koji su ubačeni u "korpu" i sa koje se može kompletirati kupovina proizvoda.

Iako ćemo govoriti o više "veb stranica" u kontekstu naše aplikacije, svi Angular projekti predstavljaju tzv. *aplikacije sa jednim dokumentom* (engl. *single-page application*, skr. *SPA*). O čemu je ovde reč? Kada se zatraži pristup nekoj stranici Angular veb aplikacije pomoću nekog URL-a, umesto da se šalje novi HTTP zahtev za taj URL, kao što je to slučaj u komunikaciji sa serverskih aplikacijama, Angular je taj koji upravlja implementacijom rutiranja i proverava da li postoji prethodno registrovano *pravilo rutiranja* (engl. *routing rule*) koje govori koju komponentu je potrebno prikazati na stranici. Ova procedura se često naziva *rutiranje na klijentu* kako bi se napravila razlika između te procedure i procedure slanja HTTP zahteva nekoj serverskoj aplikaciji koja se dalje bavi obradom tog zahteva i vraćanjem HTTP odgovora. Pogledajmo šta je neophodno uraditi da bismo implementirali rutiranje.

Za početak, potrebno je da prilikom kreiranja novog Angular projekta odaberemo da želimo da koristimo rutiranje. Angular će nas pitati da li želimo da koristimo rutiranje, ali možemo i specifikovati opciju `--routing` prilikom izvršavanja komande `ng new` kako bismo to uraditi. Nakon inicijalizacije projekta, Angular za nas kreira Angular modul koji ćemo koristiti za rutiranje u datoteci `app-routing.module.ts`:

```
1 import { NgModule } from '@angular/core';
2 import { Routes, RouterModule } from '@angular/router';
3
4
5 const routes: Routes = [
6 ];
7
8 @NgModule({
9   imports: [RouterModule.forRoot(routes)],
10  exports: [RouterModule]
11 })
12 export class AppRoutingModule { }
```

8.9.1 Specifikovanje putanja za rutiranje na klijentu

Kada god želimo da dodamo novi URL za rutiranje, to ćemo uraditi dodavanjem novog objekta u niz `routes`. Evo kako izgleda modul za rutiranje koji implementira zahteve naše aplikacije:

Kod 8.65: `/angular/store/src/app/routes/app-routing.module.ts`

```
1 import { NgModule } from '@angular/core';
2 import { Routes, RouterModule } from '@angular/router';
3 import { ProductListComponent } from '../product-list/product-list.component';
4 import { ProductInfoComponent } from '../product-info/product-info.component';
5 import { CartComponent } from '../cart/cart.component';
6
7
8 const routes: Routes = [
9   { path: '', component: ProductListComponent },
10  { path: 'checkout', component: CartComponent },
11  { path: 'products/:productId', component: ProductInfoComponent }
12 ];
```

```

13
14 @NgModule({
15   imports: [RouterModule.forRoot(routes)],
16   exports: [RouterModule]
17 })
18 export class AppRoutingModule { }

```

Ono što primećujemo jeste da svaki objekat određuje jednu putanju (ili familiju putanja, ukoliko su putanje parametrizovane). Ovi objekti imaju dva važna svojstva:

- Svojstvo `path` definiše nisku koja određuje URL putanje. Prazna niska odgovara početnoj stranici. Ukoliko neka putanja ima parametre, onda se ti parametri navode dvotačkom. Naravno, putanje mogu imati proizvoljan broj parametara. Videćemo kako je moguće dohvatiti vrednosti ovih parametara u samim komponentama u nastavku.
- Svojstvo `component` određuje koja komponenta će biti prikazana u slučaju da korisnik otvori odgovarajuću URL putanju definisanu svojstvom `path`.

Postavlja se pitanje — gde će na stranici ove komponente biti prikazane u slučaju da se URL putanja poklopi sa nekim šablonom iz svojstva `path`? Odgovor se nalazi u specijalnoj direktivi `RouterOutlet` koja se koristi kao komponenta. Ona predstavlja specijalnu komponentu koja označava mesto u HTML šablonu gde će *objekat za rutiranje* (engl. *router*) prikazivati odgovarajuću komponentu za dati URL:

Kod 8.66: angular/store/src/app/app.component.html

```

1 <div class="container">
2   <app-navigation></app-navigation>
3
4   <router-outlet></router-outlet>
5 </div>

```

Vidimo da se naša korena komponenta sastoji iz dve komponente: `NavigationComponent` koja predstavlja navigaciju u okviru naše aplikacije i `RouterOutlet` koja predstavlja sadržaj stranice koji će se dinamički menjati u zavisnosti od URL-a. Prikažimo HTML šablon za navigaciju:

Kod 8.67: angular/store/src/app/navigation/navigation.component.html

```

1 <div class="row">
2   <div class="col-12">
3     <nav class="navbar navbar-expand navbar-dark bg-dark">
4       <a class="navbar-brand" href="#">My Store</a>
5
6       <ul class="navbar-nav ml-auto">
7         <li class="nav-item">
8           <a class="nav-link" [routerLink]="['/']">Home</a>
9         </li>
10        <li class="nav-item">
11          <a class="nav-link" [routerLink]="['/checkout']">Checkout</a>
12        </li>
13      </ul>
14    </nav>
15  </div>
16 </div>

```

Primećujemo da se za veze u okviru naše aplikacije više ne koristi svojstvo `href`, već

direktiva `RouterLink`⁵. Njena vrednost je niz koja predstavlja fragmente URL-a. Prilikom posećivanja date veze, fragmenti će biti nadovezani kako bi se dobila odgovarajuća URL putanja. Naravno, fragmenata može biti proizvoljno mnogo i mogu biti bilo kog tipa, ali će biti konvertovani u nisku pre nadovezivanja.

Sa znanjem do sada, ne bi trebalo da bude teško implementirati dugme za svaki proizvod u listi koje vodi do stranice sa prikazom informacija:

Kod 8.68: angular/store/src/app/product-list/product-list.component.html

```

1 <h2>Products</h2>
2 <hr>
3
4 <div *ngFor="let product of products">
5   <h4>{{ product.name }}</h4>
6   <p>Price: {{ product.price }}</p>
7   <a [routerLink]="['/products', product.productId]" class="btn btn-primary">
     See details</a>
8 </div>

```

Na primer, za proizvod čiji je identifikator 101, aplikacija će kreirati vezu `http://localhost:4200/products/101`. Prikažimo šablon `product-info.component.html` komponente koja prikazuje informacije o proizvodu i dugme za kupovinu:

```

1 <h2>Product details</h2>
2 <hr>
3
4 <div *ngIf="product">
5   <h3>{{ product.name }}</h3>
6   <h4>${{ product.price }}</h4>
7   <p>{{ product.description }}</p>
8
9   <button (click)="addToCart()" class="btn btn-primary">Add to cart</button>
10 </div>

```

Da bismo u komponenti dohvatili informaciju o parametru `productId` koji se nalazi kao deo URL-a, moramo ubrizgati specijalan servis `ActivatedRoute`. Ovaj servis sadrži pregršt korisnih informacija o URL-u koji je trenutno posećen; između ostalog, sadrži informacije o parametrima kroz svojstvo `paramMap` koji predstavlja `Observable<ParamMap>`. Interfejs `ParamMap` sadrži metod `get` kojim se dobija niska sa vrednošću parametra koji se prosleđuje kao argument metoda. Iako je svojstvo `paramMap` tok, nije neophodno ukidati pretplatu nad njim, pošto će Angular to automatski uraditi po uništenju komponente.

Kod 8.69: angular/store/src/app/product-info/product-info.component.ts

```

1 import { Component, OnDestroy } from '@angular/core';
2 import { ActivatedRoute } from '@angular/router';
3 import { ProductService } from '../services/product.service';
4 import { ProductModel } from '../models/product.model';
5 import { Subscription } from 'rxjs';
6 import { CartService } from '../services/cart.service';
7
8 @Component({
9   selector: 'app-product-info',
10  templateUrl: './product-info.component.html',

```

⁵Zapravo, prva veza u navigaciji koristi svojstvo `href`, te se čitalac može uveriti da se klikom na tu vezu zapravo šalje novi HTTP zahtev za početnom stranicom, osim ako se aplikacija već ne nalazi na početnoj stranici. Zbog toga se za potrebe navigacije u okviru aplikacije uvek preferira korišćenje direktive `RouterLink`.


```

11   styleUrls: ['./product-info.component.css']
12 })
13 export class ProductInfoComponent implements OnDestroy {
14
15   public product: ProductModel;
16
17   // Nije neophodno ukidati pretplatu nad paramMap,
18   // posto ce Angular to automatski uraditi po unistenju komponente
19   private paramMapSub: Subscription = null;
20
21   constructor(private route: ActivatedRoute,
22               private productService: ProductService,
23               private cartService: CartService) {
24     this.paramMapSub = this.route.paramMap.subscribe(params => {
25       const pId: number = Number(params.get('productId'));
26
27       this.productService.getProducts().forEach(p => {
28         if (p.productId === pId) {
29           this.product = p;
30         }
31       });
32     });
33   }
34
35   ngOnDestroy() {
36     if (this.paramMapSub !== null) {
37       this.paramMapSub.unsubscribe();
38     }
39   }
40
41   public addToCart() {
42     this.cartService.addToCart(this.product);
43     window.alert('Your product has been added to the cart!');
44   }
45
46 }

```

Primećujemo da smo kreirali još jedan servis — `CartService` — koji nam služi za upravljanje podacima o korpi. Prikažimo i njegovu implementaciju radi kompletnosti:

Kod 8.70: angular/store/src/app/services/cart.service.ts

```

1 import { Injectable } from '@angular/core';
2 import { ProductModel } from '../models/product.model';
3
4 @Injectable({
5   providedIn: 'root'
6 })
7 export class CartService {
8   private items: ProductModel[] = [];
9
10  constructor() { }
11
12  public addToCart(product: ProductModel): void {
13    this.items.push(product);
14  }
15
16  public getItems(): ProductModel[] {
17    return this.items;
18  }
19

```

```

20 public clearCart(): ProductModel[] {
21     this.items = [];
22     return this.items;
23 }
24 }

```

U ovom delu teksta smo prikazali samo osnovne koncepte Angular rutiranja. Postoji još veliki broj koncepata o kojima nismo diskutovali, kao što su: detaljniji uvid u prikazane koncepte, ugneždene rute, programabilna rutiranja, redirekcije, postavljanje autorizacionih protokola nad rutama, animacije rutiranja, itd.

8.10 Filteri

Dobar deo prikazivanja podataka jeste izvršavanje postprocesiranja u svrhe određivanja formata u kojim će podaci biti prikazani. Na primer, trenutno se cene proizvoda prikazuju u dolarima. Međutim, šta ukoliko želimo da omogućimo korisniku da ih prikaže u nekoj drugoj valuti? Ako pogledamo implementaciju šablona `product-info.component.html` koju smo naveli u prethodnoj sekciji, primetićemo da je oznaka valute deo šablona, što bi nam pravilo problem u slučaju da želimo da promenimo valutu. Neki drugi primer sličnog problema bi bio format za prikazivanje datuma — u zavisnosti od zemlje u kojoj korisnik koji pristupa našoj Angular aplikaciji živi, potrebno je prikazati datum u drugačijem formatu. Naravno, sve ovo je moguće implementirati kao metod komponente ili kao servis koji će se baviti time. Međutim, Angular ima još jedan način koji se vrlo elegantno koristi u šablonima. U pitanju su *filteri* (engl. *pipe*).

8.10.1 Ugrađeni filteri

Angular dolazi sa velikim brojem ugrađenih filtera koji su nam na raspolaganju. Puna lista se može pronaći na vezi <https://angular.io/api?type=pipe>, a neki od filtera su ukratko opisani u nastavku:

- **CurrencyPipe** transformiše broj u nisku koja predstavlja zapis sa valutom.
- **DatePipe** transformiše datum u nisku željenog formata.
- **JsonPipe** transformiše bilo koju vrednost u nisku zapisanu u JSON formatu.
- **AsyncPipe** se pretplaćuje na tok ili obećanje i vraća poslednju vrednost koja je emitovana. Kada se nova vrednost emituje, ovaj filter označava tekuću komponentu za proces provere izmena. Kada se komponenta uništi, ovaj filter vrši automatsko uklanjanje pretplate čime se sprečava curenje memorije.

Prikažimo kako se koristi ugrađeni filter **CurrencyPipe** na primeru prikaza komponente **ProductInfoComponent**:

Kod 8.71: `angular/store/src/app/product-info/product-info.component.html`

```

1 <h2>Product details</h2>
2 <hr>
3
4 <div *ngIf="product">
5     <h3>{{ product.name }}</h3>
6     <h4>{{ product.price | currency }}</h4>
7     <p>{{ product.description }}</p>
8
9     <button (click)="addToCart()" class="btn btn-primary">Add to cart</button>
10 </div>

```

Svaki filter ima jedinstveno skraćeno ime koje mu odgovara koje se definiše prilikom definicije tog filtera. Tako, na primer, filter `CurrencyPipe` ima skraćeno ime `currency` koje se koristi u HTML šablonu komponente.

Filteri se mogu parametrizovati i njihovi argumenti se navode dvotačkom nakon naziva filtera. Filteri mogu imati proizvoljan broj parametara i ispred svakog argumenta je neophodno staviti dvotačku. Tako, na primer, ukoliko bismo želeli da prikazemo cenu u evrima, mogli bismo izvršiti filtriranje:

```
1 <h4>{{ product.price | currency: 'EUR' }}</h4>
```

8.10.2 Korisnički-definisani filteri

Za potrebe prikaza informacija iz korpe (na stranici `http://localhost:4200/checkout`), želimo da kreiramo filter koji će računati sumu svih proizvoda iz korpe. Novi filter se može kreirati naredbom:

```
ng generate pipe pipes/sum
```

Neophodno je implementirati metod `transform` kojim se definiše ponašanje filtera. Definicija našeg filtera bi mogla izgledati:

Kod 8.72: `angular/store/src/app/pipes/sum.pipe.ts`

```
1 import { Pipe, PipeTransform } from '@angular/core';
2 import { ProductModel } from '../models/product.model';
3
4 @Pipe({
5   name: 'sum'
6 })
7 export class SumPipe implements PipeTransform {
8
9   transform(products: ProductModel[]): number {
10     return products
11       .map(product => product.price)
12       .reduceRight((acc, next) => acc + next);
13   }
14 }
15 }
```

U šablonu komponente `CartComponent`, kreirani filter se koristi na sledeći način:

Kod 8.73: `angular/store/src/app/cart/cart.component.html` (linije 1-16)

```
1 <h2>Checkout</h2>
2 <hr />
3
4 <ng-template #noItems>
5   The cart is empty
6 </ng-template>
7
8 <div *ngIf="items.length > 0; else noItems">
9   <p>Your current cart includes these items:</p>
10   <div class="alert alert-info" *ngFor="let product of items">
11     <h4>{{ product.name }}</h4>
12     <hr />
13     <h6>{{ product.price | currency }}</h6>
14   </div>
15
16   <h4 class="text-right">Total price: {{ items | sum | currency }}</h4>
```

Ono što takođe primećujemo jeste da je filtere moguće komponovati, čime se procesiranje podataka značajno olakšava.

8.11 Rad sa formularima

U ovoj sekciji ćemo predstaviti osnovni metod za konstrukciju formulara i njihovu obradu. Nadovezaćemo se na prethodnu implementaciju aplikacije i implementiraćemo formular na stranici sa informacijama o odabranim proizvodima koji će zahtevati od korisnika da unese neke osnovne informacije kako bi upotpunio naručivanje proizvoda. Takođe, prikazaćemo elementarnu obradu formulara.

Iako postoje različiti metodi za rad sa formularima u Angular aplikacijama, mi ćemo u ovom tekstu predstaviti samo jedan od njih, a to je reaktivni pristup. Za početak, potrebno je da u korenom modulu `app.module.ts` naše aplikacije uključimo `ReactiveFormsModule` iz paketa `'@angular/forms'`, kao i da uvezemo taj modul u našoj aplikaciji:

```
1 import { ReactiveFormsModule } from '@angular/forms';
2
3 // ...
4
5 imports: [
6   BrowserModule,
7   AppRoutingModule,
8   ReactiveFormsModule // <----
9 ],
10
11 // ...
```

8.11.1 Kreiranje formulara u reaktivnom pristupu

U reaktivnom pristupu postoje dve reprezentacije jednog formulara:

- Skup objekata koji žive u komponenti.
- Vizualni prikaz u HTML šablonu.

Za potrebe kreiranja formulara u komponenti, koristimo servis `FormBuilder` koji je potrebno ubrizgati u `CartComponent`. Takođe, kreirajmo javnu članicu iste komponente `checkoutForm` čiji je tip `FormGroup` i koja će predstavljati koren našeg formulara. Servis `FormBuilder` ima veliki broj korisnih metoda za kreiranje formulara, a jedan od njih je metod `group` koji kreira objekat tipa `FormGroup`. Ovaj metod ima za argument objekat koji određuje nazive kontrola u formularu, njihove početne vrednosti i, opcione informacije kao što je lista funkcija koje predstavljaju validacije svake kontrole. Prikažimo za sada kod koji ovo omogućava, a kasnije ćemo se pozabaviti detaljima:

Kod 8.74: `angular/store/src/app/cart/cart.component.ts` (linije 17-34)

```
17 })
18 export class CartComponent implements OnInit {
19   public items: ProductModel[] = [];
20   public checkoutForm: FormGroup;
21
22   constructor(
23     private cartService: CartService,
24     private formBuilder: FormBuilder
25   ) {
26     this.items = this.cartService.getItems();
27     this.checkoutForm = this.formBuilder.group({
28       name: ['', [Validators.required, this.nameValidator()]],
```

```

29     address: [
30         '',
31         [Validators.required, Validators.pattern('[0-9]+ [a-zA-Z0-9]+')]
32     ],
33     email: ['', [Validators.required, Validators.email]]
34 });

```

Deo formulara koji se implementira u šablonu predstavlja klasičan HTML formular sa odgovarajućim proširenjima:

- Elementu `form` je potrebno pridružiti direktivu `FormGroup` čija je vrednost kreirani objekat iz komponente. Time se ostvaruje veza između formulara i `FormGroup` objekta.
- Svakoј kontroli iz formulara je potrebno pridružiti atribut `formControlName` čija vrednost odgovara jednom od naziva koji su navedeni prilikom kreiranja `FormGroup` objekta u komponenti. Time se ostvaruje veza između pojedinačne kontrole formulara i odgovarajuće reprezentacije u komponenti. Ukoliko je naveden niz validacionih funkcija, onda će te funkcije biti iskorišćene prilikom validacije formulara.
- Elementu `form` se pridružuje osluškivač na događaj `ngSubmit` koji će biti okinut kada se klikne na dugme tipa `'submit'` u okviru formulara. Metod koji se poziva prilikom okidanja ovog događaja mora da implementira logiku kojom se podaci šalju da odredište — ovo odgovara postavljanju atributa `method` i `action` u klasičnom pristupu obrade HTML formulara.

Kod 8.75: angular/store/src/app/cart/cart.component.html (linije 18-57)

```

18 <form [formGroup]="checkoutForm" (ngSubmit)="submitForm(checkoutForm.value)">
19   <div class="form-group">
20     <label for="name">Name</label>
21     <input
22       type="text"
23       name="name"
24       id="name"
25       class="form-control"
26       formControlName="name"
27     />
28   </div>
29   <div class="form-group">
30     <label for="address">Address</label>
31     <input
32       type="text"
33       name="address"
34       id="address"
35       class="form-control"
36       formControlName="address"
37     />
38   </div>
39   <div class="form-group">
40     <label for="email">Email</label>
41     <input
42       type="text"
43       name="email"
44       id="email"
45       class="form-control"
46       formControlName="email"
47     />
48   </div>
49   <button
50

```

```

51     class="btn btn-primary btn-block"
52     type="submit"
53     [disabled]="!checkoutForm.valid"
54   >
55     Purchase
56   </button>
57 </form>

```

8.11.2 Validacija formulara u reaktivnom pristupu

Diskutujemo sada o načinu validiranja formulara. Prema definiciji, formular je *validan* ukoliko je svaka od njegovih kontrola koje korisnik popunjava validna. Sa druge strane, za svaku kontrolu je moguće precizno definisati domen vrednosti koje se smatraju validnim.

Angular nam nudi ugrađene validacione funkcije koje su dostupne kroz klasu `Validators` iz paketa `'@angular/forms'`. Za potpuniju listu ovih funkcija posetiti vezu <https://angular.io/api/forms/Validators>. Kratki opisi neki od ovih funkcija su dati u nastavku:

- `Validators.required` proverava da li je kontroli popunjeno. Napomenimo da se i blanko karakteri smatraju kao validni karakteri.
- `Validators.email` proverava da li unos u kontroli zadovoljava standardan format adrese elektronske pošte⁶.
- `Validators.pattern` proverava da li unos u kontroli zadovoljava regularni izraz koji se prosleđuje kao argument validatorske funkcije.

Sa druge strane, moguće je konstruisati svoju validatorsku funkciju. Prema pravilu, korisnički-definisana validatorska funkcija predstavlja funkciju koja zadovoljava interfejs `ValidatorFn`, tj. funkcija mora da prihvata kao argument objekat klase `AbstractControl` i da vrati jednu od naredne dve vrednosti:

- Ukoliko se validacija kontrole završi uspešno, funkcija vraća `null`.
- Inače, vraća se objekat tipa `ValidationErrors` koji predstavlja klasičan objekat čiji su ključevi niske, a vrednosti proizvoljne (*any*). Ukoliko validacije kontrole prođe neuspešno, onda se ovaj objekat može iskoristiti za dohvaćanje informacija o problemima u validaciji.

Naredna implementacija predstavlja korisnički-definisanu validaciju za ime kupca — svako ime se mora sastojati od barem dva dela (tj. makar jedno ime i jedno prezime):

Kod 8.76: `angular/store/src/app/cart/cart.component.ts` (linije 39-51)

```

39 // This is a factory method – returns a function which serves as a custom
    validator
40 public nameValidator(): ValidatorFn {
41   // The validation function itself must return:
42   // (1) in case of a successful validation:
43   //     null
44   // (2) in case of a failed validation:
45   //     a validation error object
46   return (control: AbstractControl): ValidationErrors | null => {
47     const nameIsCorrect =
48       control.value.split(' ').filter((el: string) => el !== '').length > 1;
49     return nameIsCorrect ? null : { incorrectName: true };
50   };
51 }

```

⁶<https://html.spec.whatwg.org/multipage/input.html#valid-e-mail-address>.

Pre nego što prikazemo kako je moguće koristiti objekat `ValidationErrors` za prikazivanje informacija o validaciji u šablonu, kreirajmo pomoćne očitavače koji će dohvatati objekte iz komponente koji odgovaraju kontrolama formulara u šablonu:

Kod 8.77: angular/store/src/app/cart/cart.component.ts (linije 66-74)

```
66 public get name() {  
67     return this.checkoutForm.get('name');  
68 }  
69 public get address() {  
70     return this.checkoutForm.get('address');  
71 }  
72 public get email() {  
73     return this.checkoutForm.get('email');  
74 }
```

U HTML šablonu komponente ćemo dodati nova polja koja će biti vidljiva samo ukoliko neka validacija kontrola nije uspešna:

Kod 8.78: angular/store/src/app/cart/cart.component.html (linije 59-85)

```
59 <div class="alert alert-danger" *ngIf="name.errors?.required">  
60     Name is required  
61 </div>  
62 <div  
63     class="alert alert-danger"  
64     *ngIf="name.errors?.incorrectName"  
65 >  
66     Name is not correct  
67 </div>  
68 <div  
69     class="alert alert-danger"  
70     *ngIf="address.errors?.required"  
71 >  
72     Address is required  
73 </div>  
74 <div  
75     class="alert alert-danger"  
76     *ngIf="address.errors?.pattern"  
77 >  
78     Address doesn't match the pattern  
79 </div>  
80 <div class="alert alert-danger" *ngIf="email.errors?.required">  
81     Email is required  
82 </div>  
83 <div class="alert alert-danger" *ngIf="email.errors?.email">  
84     Email is not valid  
85 </div>
```

Kao što vidimo, svaki objekat kontrole čuva svojstvo `errors` koji će biti `null` ukoliko je validacija te kontrole uspešna (otuda koristimo operator `?` nakon naziva svojstva). Ugrađene validacione funkcije će postaviti odgovarajuća svojstva ovog objekta ukoliko one ne validiraju uspešno datu kontrolu (na primer, svojstvo `required` odgovara neuspešnoj validaciji `Validators.required`). U liniji 64 primećujemo da se koristi svojstvo `incorrectName` koje smo mi definisali u našoj korisnički-definisanoj funkciji za validaciju.

Ovu sekciju završavamo prikazom implementacije funkcije za slanje podataka iz formulara. Primetimo da ne postoji kod koji zapravo šalje podatke nekoj serverskoj aplikaciji. O ovome ćemo detaljno diskutovati kada budemo govorili o povezivanju klijentskih i serverskih aplikacija u sekcij 8.12.

Kod 8.79: angular/store/src/app/cart/cart.component.ts (linije 53-64)

```

53 public submitForm(data): void {
54     console.log(data);
55     if (!this.checkoutForm.valid) {
56         window.alert('Not valid!');
57         return;
58     }
59
60     // Contact server here...
61
62     this.items = this.cartService.clearCart();
63     this.checkoutForm.reset();
64 }

```

Kao što vidimo, možemo dobiti informaciju da li je ceo formular validan (tj. da li su sve njegove kontrole validne) uvidom u svojstvo `valid` nad objektom `FormGroup` koji je pridružen celom formularu. Ovo svojstvo smo takođe iskoristili u HTML šablonu radi isključivanja dugmeta za slanje podataka u slučaju da formular nije validan.

U ovom delu teksta smo prikazali samo osnovne koncepte rada sa formularima u Angular radnom okviru. Postoji još veliki broj koncepata o kojima nismo diskutovali, kao što su: šablonski pristup radu sa formularima, dinamički-kreirani formulari, validacija kroz više kontrola, razlike između sinhronih i asinhronih validacionih funkcija, itd.

8.12 HTTP komunikacija u Angular aplikacijama

U ovom poglavlju se upoznajemo sa osnovnim klasama i tehnikama koje nam Angular radni okvir omogućava za kreiranje HTTP zahteva i obrade HTTP odgovora. Cilj ovog poglavlja je razumevanje pozicije Angular radnog okvira u klijentsko-serverskoj arhitekturi komunikacije između veb aplikacija. Naravno, kao što znamo, Angular čini klijentsku stranu ove arhitekture.

U ovom poglavlju ćemo se vratiti na aplikaciju koja implementira onlajn prodavnicu iz poglavlja 8 i unaprediti je tako da se mogu kreirati novi ili brisati postojeći proizvodi, naručivati korpe proizvoda, pregledati narudžbine ili poništavati ih.

Naravno, u tu svrhu će nam biti neophodna serverska aplikacija koja komunicira sa nekim SUBP za trajno skladištenje svih ovih informacija i implementira date operacije. Mi smo konstruisali Node.js aplikaciju koja implementira REST API koji implementira date zahteve nad MongoDB SUBP. Izvorni kod za ovu aplikaciju je moguće pronaći na lokaciji `primeri/angular/prodavnica/prodavnica-server`. Napomenimo da se za uspešno izvršavanje koda u ovom poglavlju podrazumeva da je ova serverska aplikacija podignuta na istom računaru i to na portu `3000`, odnosno, da će prihvatati zahteve na korenom URL-u `http://localhost:3000/`. Slanjem GET zahteva na upravo taj koreni URL moguće je dobiti informacije o samom API-ju:

```

[
  { path: '/', children: [] },
  { path: '/product', children: [
    { path: '/', method: 'GET', children: [] },
    { path: '/', method: 'POST', children: [] },
    { path: '/', method: 'GET', parameters: ['productId'], children: [] },
    { path: '/', method: 'PATCH', parameters: ['productId'], children: [] },
    { path: '/', method: 'DELETE', parameters: ['productId'], children: [] }
  ]
},

```



```

    { path: '/order', children: [
      { path: '/', method: 'GET', children: [] },
      { path: '/', method: 'POST', children: [] },
      { path: '/', method: 'GET', parameters: ['orderId'], children: [] },
      { path: '/', method: 'DELETE', parameters: ['productId'], children: [] }
    ]
  },
]

```

8.12.1 Modul `HttpClientModule` i servis `HttpClient`

Da bismo mogli da konstruišemo HTTP zahteve ka serverskim aplikacijama iz Angular klijentskih aplikacija, prva stvar koju je neophodno da uradimo jeste da u korenom modulu naše Angular aplikacije uvezemo modul `HttpClientModule` iz paketa `'@angular/common/http'`:

```
1 import { HttpClientModule } from '@angular/common/http';
```

Zatim, taj modul je potrebno dodati u niz `imports` u dekoratoru `@NgModule` našeg korenog modula:

```

1 @NgModule({
2   ...
3   imports: [
4     ...
5     HttpClientModule
6   ],
7   ...
8 })
9 export class AppModule { }

```

Nakon ove operacije, u našoj Angular aplikaciji će postati dostupan servis `HttpClient` kojeg je neophodno ubrizgati u odgovarajuće servise, komponente, direktive i sve druge elemente naše aplikacije koje implementiraju ostvarivanje komunikacije ka nekim serverskim aplikacijama. Smatra se dobrom praksom da se sav kod koji izvršava ovu komunikaciju smesti u odgovarajući servis, a da se potom taj server koristi kao davalac podataka ili usluga. To je upravo ono što ćemo i mi uraditi.

8.12.2 Slanje HTTP zahteva

Servis `HttpClient` predstavlja klasu koja izvršava HTTP zahteve. Ova klasa je dostupna kao servis koji se ubrizgava u druge klase i sadrži metode koji se koriste radi izvršavanja odgovarajućih HTTP zahteva. U ovoj sekciji ćemo prvo prikazati ove metode, a zatim ćemo ih iskoristiti za implementiranje određenih operacija u našoj aplikaciji.

Svaki od metoda servisa `HttpClient` sadrži veliki broj preopterećenja, a mi ćemo prikazati samo neke od njih:

- Metod `request(method: string, url: string, options: ...)` služi za kreiranje opšteg HTTP zahteva. Njegov prvi argument predstavlja tip HTTP metoda; drugi argument je URL; treći i opciono argument je objekat koji sadrži razne opcije za podešavanje zahteva. Opcije koje su dostupne su:
 - `body?: any` predstavlja telo zahteva;
 - `headers?: HttpHeaders | { [header: string]: string | string[]; }` predstavlja spisak zaglavlja u HTTP zahtevu. Kao što vidimo, to može biti ili objekat klase `HttpHeaders` ili mapa (nizova) niski;

- `observe?: 'body' | 'events' | 'response'` određuje šta je to što će biti vraćeno aplikaciji prilikom obrade HTTP odgovora od servera. Podrazumevana vrednost ove opcije je `'body'`, što znači da će aplikaciji biti prosledeno telo odgovora, bez ostalih informacija. Ovo ponašanje je i najčešće korišćeno, ali ukoliko je potrebno da obradimo neke dodatne informacije, poput zaglavlja u HTTP odgovoru, onda ju je potrebno promeniti. Druge moguće vrednosti su `'events'` čime se ostvaruje da je rezultat tipa `HttpEvent` ili `'response'` čime se ostvaruje da je tipa rezultat `HttpResponse`;
 - `params?: HttpParams | { [param: string]: string | string[]; }` predstavlja spisak parametara u telu HTTP zahteva koji se serijalizuju u MIME formatu `application/x-www-form-urlencoded`;
 - `reportProgress?: boolean` podešava da li se Angular aplikaciji isporučuje informacija o progresu HTTP zahteva;
 - `responseType?: 'json' | 'arraybuffer' | 'blob' | 'text'` određuje koji je MIME tip podataka u telu HTTP odgovora. Podrazumevana vrednost je `'json'`, što znači da će aplikacija deserijalizovati telo HTTP odgovora iz formata `application/json`. Kao što smo često govorili, JSON format je postao najpopularniji izbor za format podataka u veb komunikaciji, tako da je najčešće dovoljno ostaviti ovu opciju kao podrazumevanu. Pored ove vrednosti, moguće je postaviti naredne vrednosti: `'arraybuffer'`, čime se telo odgovora dobija kao `ArrayBuffer`, `'blob'`, čime se telo odgovora dobija kao `Blob` i `'text'`, čime se telo odgovora dobija kao `string`;
 - `withCredentials?: boolean` određuje da li se koriste kredencijali u HTTP zahtevu.
- Metod `get(url: string, options: ...)` služi za kreiranje GET zahteva.
 - Metod `post(url: string, body: any, options: ...)` služi za kreiranje POST zahteva.
 - Metod `patch(url: string, body: any, options: ...)` služi za kreiranje PATCH zahteva.
 - Metod `put(url: string, body: any, options: ...)` služi za kreiranje PUT zahteva.
 - Metod `delete(url: string, options: ...)` služi za kreiranje DELETE zahteva.
 - Metod `head(url: string, options: ...)` služi za kreiranje HEAD zahteva.
 - Metod `options(url: string, options: ...)` služi za kreiranje OPTIONS zahteva.
 - Metod `jsonp(url: string, callbackParam: string)` služi za kreiranje specijalnih HTTP zahteva korišćenjem JSONP obrascu. Može se koristiti u slučaju da serverske aplikacije ne podržavaju CORS protokol. Naše serverske aplikacije su ispravno implementirane, te se nećemo udubljivati u funkcionisanje ovog metoda. Za više informacija o JSONP obrascu, posetite <https://en.wikipedia.org/wiki/JSONP>.

Ono što je važno napomenuti jeste da je povratna vrednost svih ovih metoda RxJS tok, odnosno `Observable<T>`, pri čemu `T` zavisi od preopterećenja ovih metoda kao i od opcija koje smo naveli. Tako, na primer:

- Postavljanje opcije `observe` na vrednost `'body'` rezultuje da je povratna vrednost metoda `Observable<T>`, gde je `T` podatak iz tela odgovora koji zavisi od vrednosti opcije `responseType`. Specijalno, ukoliko je postavljena opcija `responseType` na `'json'`, onda je moguće specifikovati interfejs rezultujućih podataka. Više o tome u podsekciji ??;
- Postavljanje opcije `observe` na vrednost `'events'` rezultuje da je povratna vrednost

metoda `Observable<HttpEvent>`;

- Postavljanje opcije `observe` na vrednost `'response'` rezultuje da je povratna vrednost metoda `Observable<HttpResponse<T>>`, gde parametar `T` zavisi od opcije `responseType` i opcionog tipiziranog parametra (više o tome u podsekciji ??).

8.12.3 Obrada HTTP odgovora

Dakle, da bismo kreirali novi GET zahtev ka našoj serverskoj aplikaciji, možemo iskoristiti metod `get()`. Ovaj, asinhroni metod šalje HTTP zahtev i vraća tok koji emituje odgovarajući podatak kada se dohvati HTTP odgovor od servera.

Hajde da kreiramo GET zahtev ka URL-u `http://localhost:3000/products`. U telu HTTP odgovora od servera biće zapisan niz proizvoda u `application/json` formatu. Da bismo dohvatili ovaj niz, neophodno je da metodu `get()` prosledimo opcije `{observe: 'body', responseType: 'json'}`. Međutim, kao što smo napomenuli, ove vrednosti su podrazumevane, tako da naredni poziv metoda `get()` ciljano nema treći argument:

```

1  ...
2  export class ProductService() {
3    private products;
4    private readonly productsUrl = 'http://localhost:3000/products/';
5
6    constructor(private http: HttpClient, ...) {
7      this.refreshProducts();
8    }
9
10   private refreshProducts() {
11     this.products = this.http.get(this.productsUrl);
12     return this.products;
13   }
14
15   public getProducts() {
16     return this.products;
17   }
18 }
```

Metod `refreshProducts()` izvršava kreiranje GET zahteva i čuva tok u odgovarajući atribut. S obzirom da "posmatramo" telo HTTP odgovora, ovaj tok će emitovati niz proizvoda. Metod `getProducts()` jednostavno dohvata vrednost ovog toka.

U komponenti gde je potrebno iskoristiti dohvaćenu vrednost, na primer, u `ProductListComponent`, ovaj servis se može koristiti na sledeći način:

```

1  ...
2  export class ProductListComponent {
3    public products;
4
5    constructor(private productService: ProductService) {
6      this.productService.getProducts()
7        .subscribe(products => {
8          this.products = products;
9          // Do something with the products...
10        });
11    }
12 }
```

8.12.4 Uvođenje tipiziranosti zahteva

S obzirom da Angular radni okvir koristi jezik TypeScript, vrlo je korisno strukturirati podatke od servera da odgovaraju nekom tipu, kako bismo postigli statičku proveru tipova i za podatke koji se dohvataju izvan Angular aplikacije.

Ono što se često radi jeste da se definiše interfejs koji postavlja ograničenje na podatke koji se dohvataju od servera. U našoj onlajn prodavnici, korisno je da definišemo interfejs koji predstavlja jedan proizvod. Prvo pogledajmo model podataka na serveru:

Kod 8.80: angular/prodavnica/prodavnica-server/components/product/productModel.js

```
1 // U ovom fajlu definisemo model koji ce imati nasi proizvodi
2 const mongoose = require('mongoose');
3
4 const productSchema = mongoose.Schema({
5   _id: mongoose.Schema.Types.ObjectId,
6   name: {
7     type: String,
8     required: true
9   },
10  price: {
11    type: Number,
12    required: true
13  },
14  description: String,
15 });
16
17 module.exports = mongoose.model("Product", productSchema);
```

Dakle, podaci koji se dohvataju sa servera moraju da imaju svoj identifikator, naziv, cenu i opis. Hajde da u našoj klijentskoj aplikaciji kreiramo interfejs koji odgovara ovim podacima:

Kod 8.81: angular/prodavnica/prodavnica-klijent/src/app/product/product.model.ts

```
1 export interface Product {
2   _id: string;
3   name: string;
4   price: number;
5   description: string;
6 }
```

Da bismo označili koji je tip podataka koji se nalaze u telu HTTP odgovora, moguće je koristiti šablonska preopterećenja metoda iz `HttpClient` servisa, na primer, `get<T>()`. Šablonski parametar `T` predstavlja upravo tip podataka iz tela HTTP odgovora. Ova preopterećenja podrazumevaju da su podaci u telu zapisani u `application/json` formatu, odnosno, da je opcija `responseType` postavljena na `'json'`.

Sada možemo refaktorisati kod u servisu `ProductService` da koristi strogu tipiziranost na sledeći način:

```
1 ...
2 export class ProductService() {
3   private products: Observable<Product[]>;
4   private readonly productsUrl = 'http://localhost:3000/products/';
5
6   constructor(private http: HttpClient, ...) {
7     this.refreshProducts();
8   }
```

```

9
10 private refreshProducts(): Observable<Product[]> {
11     this.products = this.http.get<Product[]>(this.productsUrl);
12     return this.products;
13 }
14
15 public getProducts(): Observable<Product[]> {
16     return this.products;
17 }
18 }

```

Primitite poziv metoda `get<Product[]>()`. U komponenti koja koristi ovaj servis, takođe je moguće primeniti strogu tipiziranost:

```

1 ...
2 export class ProductListComponent {
3     public products: Product[] = [];
4
5     constructor(private productService: ProductService) {
6         this.productService.getProducts()
7             .subscribe((products: Product[]) => {
8                 this.products = products;
9                 // Do something with the products...
10            });
11    }
12 }

```

8.12.5 Filter AsyncPipe

Naša aplikacija ne bi bila preterano korisna da se niz proizvoda koji je dohvaćen iz serverske aplikacije negde ne prikazuje. Trebalo bi da komponenta `ProductListComponent` u svom šablonu prikazuje dohvaćene vrednosti na neki način.

Zapravo, deo te implementacije smo već napisali. Ako pogledamo definiciju klase `ProductListComponent`, videćemo da postoji atribut `products` čiji je tip niz proizvoda i koji je inicijalno prazan. U šablonu ove komponente bi se taj niz mogao iskoristiti za prikazivanje informacija o proizvodima na sledeći način:

```

1 <div *ngFor="let product of products">
2     <h4>{{ product.name }}</h4>
3     <p>Price: {{ product.price }}</p>
4     ...
5 </div>

```

Međutim, ono što ovde možda nije očigledno jeste da postoji šansa za curenjem memorije. Naime, nakon pretplaćivanja na tok u modelu komponente, mi nigde ne vodimo računa o ukidanju pretplate. Zbog toga bi trebalo dopuniti implementaciju modela da uzme u obzir i ovu napomenu:

```

1 ...
2 export class ProductListComponent implements OnDestroy {
3     public products: Product[] = [];
4     private activeSubscriptions: Subscriptions[] = [];
5
6     constructor(private productService: ProductService) {
7         const sub = this.productService.getProducts()
8             .subscribe((products: Product[]) => {
9                 this.products = products;
10                // Do something with the products...

```

```

11     });
12     this.activeSubscriptions.push(sub);
13 }
14
15 ngOnDestroy() {
16     this.activeSubscriptions.forEach(sub => {
17         sub.unsubscribe();
18     });
19 }
20 }

```

Umesto svega ovoga, za potrebe prikazivanja podataka iz toka u šablonu, moguće je koristiti ugrađeni Angular filter `AsyncPipe` čiji je naziv `async`. Ovaj filter se pretplaćuje na ulazni podatak, koji može biti tok ili obećanje, i vraća poslednju emitovanu vrednost. Kada se nova vrednost u toku emituje, `async` filter označava komponentu da bude proverena u ciklusu detekcije promena. Kada se komponenta uništava, `async` filter automatski vrši ukidanje pretplate radi izbegavanja potencijalnog curenja memorije.

Dakle, našu komponentu ćemo izmeniti tako da se u njenom modelu čuva samo tok niza proizvoda koji se dobija kao rezultat izvršavanja GET zahteva iz servisa:

```

1 ...
2 export class ProductListComponent {
3     public products: Observable<Product[]>;
4
5     constructor(private productService: ProductService) {
6         this.products = this.productService.getProducts();
7     }
8 }

```

Primitite da je kod skoro tri puta kraći od početnog. Šablon ove komponente se menja samo na jednom mestu i to dodavanjem filtera `async` nakon zadavanja toka `products` u `NgForOf` direktivi:

```

1 <div *ngFor="let product of products | async">
2     <h4>{{ product.name }}</h4>
3     <p>Price: {{ product.price }}</p>
4     ...
5 </div>

```

Šablonski izraz `let product of products | async` ima naredno značenje: prvo se pretplati filterom `async` na tok `products` (čiji je tip `Observable<Product[]>`) i kada bude emitovana vrednost iz tog toka (dakle, kada bude emitovana vrednost tipa `Product[]`), iteriraj kroz emitovani niz i generiši `<div>` element za svaki element iz tog niza.

8.12.6 Dodavanje tela HTTP zahtevu

Neki HTTP metodi podrazumevaju da postoji sadržaj u telu zahteva koji će biti poslat serveru. Takvi metodi su `POST`, `PATCH` i `PUT`. Ako se prisetimo definicije odgovarajućih metoda iz servisa `HttpClient`, primetićemo da svaki od njih kao drugi argument zahteva objekat koji predstavlja telo zahteva.

- Metod `post(url: string, body: any, options: ...)` služi za kreiranje `POST` zahteva.
- Metod `patch(url: string, body: any, options: ...)` služi za kreiranje `PATCH` zahteva.

- Metod `put(url: string, body: any, options: ...)` služi za kreiranje PUT zahteva.

S obzirom da je njegov tip `any`, to znači da možemo proslediti proizvoljne podatke. Ovo ima smisla s obzirom da različiti serveri zahtevaju različite podatke. U našoj aplikaciji, možemo ilustrovati korišćenje metoda `post()` kroz kreiranje nove porudžbine na sledeći način. Ovoga puta krećemo iz suprotnog smera, odnosno, započinjemo od metoda koji vrši obradu formulara:

```

1  ...
2  export class CartComponent {
3      public items: Product[];
4      public checkoutForm: FormGroup;
5      ...
6
7      constructor(
8          private cartService: CartService,
9          private FormBuilder: FormBuilder
10     ) {
11         ...
12     }
13
14     ...
15
16     public submitForm(): void {
17         ...
18
19         this.cartService.createAnOrder(this.checkoutForm.value)
20             .subscribe((order: Order) => {
21                 window.alert(
22                     `Your order (number: ${order._id}) is successfully created!`
23                 );
24                 this.items = this.cartService.clearCart();
25                 this.checkoutForm.reset();
26             });
27
28         ...
29     }
30     ...
31 }

```

Kao što vidimo, u metodu `submitForm()` pozivamo metod `createAnOrder()` servisa `CartService` i prosleđujemo mu podatke iz formulara u vidu objekta sa odgovarajućim nazivima i vrednostima. Nakon toga, pretplaćujemo se na rezultujući tok kako bismo korisniku mogli da prikažemo informaciju o uspešnosti poručivanja, kao i da bismo počistili zastarele podatke, odnosno, da bismo ispraznili korpu i obrisali unete podatke iz formulara zarad kreiranja nove porudžbine.

Pogledajmo sada definiciju metoda `createAnOrder()` servisa `CartService`:

```

1  ...
2  export class CartService {
3      private items: Product[] = [];
4      private readonly ordersUrl = 'http://localhost:3000/orders/';
5
6      constructor(private http: HttpClient) {
7      }
8
9      ...
10

```

```

11 public createAnOrder(formData): Observable<Order> {
12     const body = {
13         ...formData /* destrukcija objekta formData */,
14         products: this.items
15     };
16     return this.http.post<Order>(this.ordersUrl, body);
17 }
18 ...
19 }

```

Kao što vidimo metod `createAnOrder()` prihvata podatke iz formulara. On, dodatno, kreira objekat koji predstavlja telo zahteva koji pored podataka iz formulara sadrži i niz proizvoda koje je korisnik odabrao za kupovinu, a koje ovaj servis pamti u nizu proizvoda `items`. Telo HTTP zahteva se zatim prosleđuje kao drugi argument metodu `post()`. Podrazumevano, telo HTTP zahteva se šalje u formatu `application/json`. Ukoliko je potrebno podatke poslati u drugom formatu, onda je potrebno postaviti zaglavlje `'Content-Type'` na odgovarajući format. Menjanje zaglavlja HTTP zahteva je nešto što nećemo prikazivati, ali daćemo resurse za istraživanje u sekciji 8.12.8.

8.12.7 Obrada grešaka

Ako HTTP zahtev ka serveru ne uspe da se ostvari, metodi iz `HttpClient` će vratiti objekat greške umesto uspešnog odgovora. Zbog toga, isti servis koji izvršava HTTP zahteve trebalo bi i da vrši obradu grešaka. Pogledajmo kako je moguće implementirati obradu grešaka.

Naša aplikacija bi trebalo da implementira neku inteligentnu operaciju ukoliko se pojavi greška prilikom slanja HTTP zahteva. Postoje dve vrste grešaka do kojih može doći:

- Serverska aplikacija može da odbaci HTTP zahtev zbog neke greške, bilo zbog problema na klijentu (familija statusnih kodova 4XX) ili zbog problema na serveru (familija statusnih kodova 5XX). Ova klasa grešaka predstavlja *greške u odgovoru*.
- Do problema može doći i do klijentske strane kao što je greška u mreži koja onemogućava ispunjavanje HTTP zahteva ili izuzetak do kojeg je došlo u nekom RxJS operatoru. Ova klasa grešaka je predstavljena tipom `ErrorEvent`.

`HttpClient` hvata obe vrste grešaka i proizvodi objekat tipa `HttpErrorResponse`. Neka od njegovih svojstava su:

- `message: string` sadrži poruku o grešci;
- `error: any | null` predstavlja objekat greške.

Ova klasa nasleđuje apstraktnu klasu `HttpResponseBody` koja ima neka dodatna korisna svojstva:

- `headers: HttpHeaders` sadrži kolekciju zaglavlja u HTTP odgovoru;
- `status: number` predstavlja statusni kod;
- `statusText: string` predstavlja tekstualni opis statusnog koda;
- `url: string | null` predstavlja URL dohvaćenog resursa ili `null` ukoliko ne postoji.

S obzirom da naša serverska aplikacija u slučaju greške vraća objekat koji sadrži svojstvo `message`, a takođe objekat tipa `HttpErrorResponse` sadrži informaciju o statusnom kodu, onda bi valjalo da u slučaju greške sa servera prikažemo specijalnu stranicu u veb pregledaču koja će ispisati ove informacije.

U tu svrhu, hajde da prvo kreiramo komponentu `ErrorPageComponent`:


```
$ ng generate component routing/error-page
```

Njen model je:

Kod 8.82: angular/prodavnica/prodavnica-klijent/src/app/routing/error-page/error-page.component.ts

```
1 import { ActivatedRoute } from '@angular/router';
2 import { Component, OnInit } from '@angular/core';
3
4 @Component({
5   selector: 'app-error-page',
6   templateUrl: './error-page.component.html',
7   styleUrls: ['./error-page.component.css'],
8 })
9 export class ErrorPageComponent implements OnInit {
10   public message: string;
11   public statusCode: string;
12
13   constructor(private route: ActivatedRoute) {
14     this.route.paramMap.subscribe((params) => {
15       this.message = params.get('message');
16       this.statusCode = params.get('statusCode');
17     });
18   }
19
20   ngOnInit() {}
21 }
```

Njen šablon je:

Kod 8.83: angular/prodavnica/prodavnica-klijent/src/app/routing/error-page/error-page.component.html

```
1 <div class="display-1 text-center">
2   {{ statusCode }}
3   <h1>An error occurred :(</h1>
4 </div>
5
6 <p class="text-center">{{ message }}</p>
```

Neka je potrebno da se ova komponenta prikaže na adresi <http://localhost:4200/error>. Očigledno, neophodno je dodati novo pravilo za rutiranje u našem modulu za rutiranje:

```
1 ...
2 const routes: Routes = [
3   ...
4   { path: 'error', component: ErrorPageComponent }
5 ];
6
7 @NgModule({
8   imports: [RouterModule.forRoot(routes)],
9   exports: [RouterModule]
10 })
11 export class AppRoutingModule { }
```

Postavlja se pitanje kako je moguće prikazati ovu stranicu ukoliko dođe do greške. Da bismo ovo izvršili, neophodno je da implementiramo metod koji će, kada se greška dogodi, preusmeriti našu aplikaciju na <http://localhost:4200/error>. Ovo je moguće ostvariti

jedino putem dinamičkog rutiranja. Na sreću, implementacija ovog dela je poprilično jednostavna. Sve što je potrebno uraditi jeste u odgovarajuću klasu koja vrši obradu grešaka ubrizgati servis `Router` iz paketa `'@angular/router'` i zatim pozvati metod `navigate()` iz tog servisa:

- Prvi argument ovog metoda je niz koji sadrži komande na osnovu kojih se konstruiše URL ka kojoj je potrebno preusmeriti aplikaciju;
- Drugi, opcioni, argument je objekat interfejsa `NavigationExtras` kojim se mogu postaviti neke dodatne komponente URL-a, kao što su parametri upita, heš vrednost, relativna putanja na osnovu koje se URL formira pomoću prvog argumenta, itd.

Za naše potrebe, biće dovoljno da koristimo samo prvi argument. Ono što bismo želeli jeste da preusmerimo aplikaciju na URL `http://localhost:4200/error` i da mu prosledimo parametre `message` i `statusCode` kako bi komponenta `ErrorPageComponent` mogla da im pristupi (videti definiciju iznad). Ovo je moguće uraditi narednim pozivom:

```
1 this.router.navigate([ '/error', { message: ... , statusCode: ... } ]);
```

Naravno, neophodno je i postaviti vrednosti za odgovarajuća svojstva `message` i `statusCode` i to ćemo izvršiti u nastavku.

S obzirom da ćemo obradu grešaka raditi na isti način i za `ProductService` i za `CartService` (o drugom servisu nismo pričali u ovom poglavlju, ali ovaj servis će upravljati informacijama o porudžbinama), bilo bi loše da funkciju za obradu grešaka dupliciramo u oba servisa. Umesto toga, inteligentnije bi bilo da izdvojimo ovu funkciju u zasebnu apstraktnu klasu, a zatim da naši servisi nasleđuju tu klasu i samim tim imaju pristup funkciji za obradu greške. U nastavku dajemo definiciju apstraktne klase `HttpErrorHandler` koja nam služi upravo u te svrhe:

Kod 8.84: angular/prodavnica/prodavnica-klijent/src/app/utils/http-error-handler.model.ts

```
1 import { Observable, throwError } from 'rxjs';
2 import { HttpResponse } from '@angular/common/http';
3 import { Router } from '@angular/router';
4
5 export abstract class HttpErrorHandler {
6   constructor(private router: Router) {}
7
8   protected handleError() {
9     return (error: HttpResponse): Observable<never> => {
10       if (error.error instanceof ErrorEvent) {
11         // A client-side or network error occurred. Handle it accordingly.
12         console.error('An error occurred:', error.error.message);
13       } else {
14         // The backend returned an unsuccessful response code.
15         // The response body may contain clues as to what went wrong,
16         this.router.navigate([
17           '/error',
18           { message: error.error.message, statusCode: error.status },
19         ]);
20       }
21       // return an observable with a user-facing error message
22       return throwError('Something bad happened; please try again later.');
```

Prodiskutujmo sada o definiciji apstraktne klase `HttpErrorHandler`. Ona ima privatni atribut `router` i zaštićeni metod `handleError()`. Ovaj metod je zaštićen da bi mogao da bude korišćen na nivou definicije klase koja nasleđuje apstraktnu klasu `HttpErrorHandler`. Kao što vidimo, dati metod vraća zatvorenje kao povratnu vrednost. Objasnićemo zašto je neophodno da ovaj metod vrati zatvorenje nešto kasnije.

Zatvorenje koje se kreira mora da prati naredni potpis. Ono mora kao argument da uzima objekat tipa `HttpErrorResponse`, o kojem smo već diskutovali i da vrati novi tok. Ako se prisetimo obrade grešaka o RxJS biblioteci, ovo bi trebalo da nas asocira na jedan od operatora, ali ostavićemo diskusiju o korišćenju ovog zatvorenja nakon opisa njegove definicije.

Definicija zatvorenja jeste ono što mi želimo da bude izvršeno kada dođe do greške. Recimo da želimo da se u slučaju bilo kakve greške na klijentu jednostavno ispiše greška u konzolu, dok za svaku grešku koju smo dobili od servera izvrši preusmerenje aplikacije na stranicu za grešku. Dodatno, recimo da želimo da se u oba slučaja vrati tok koji ispaljuje grešku, što je moguće izvršiti pozivom RxJS funkcije `throwError()` i prosleđivanjem greške.

Informacija o tome zbog čega je nastala greška se može izvršiti ispitivanjem tipa svojstva `error` iz `HttpErrorResponse` objekta. Ako je tip ovog svojstva `ErrorEvent`, onda je došlo do greške na klijentu. U suprotnom, greška je potekla sa servera i to svojstvo ima vrednost iz tela servera, što je u našem slučaju objekat koji ima svojstvo `message` koje sadrži poruku sa servera. Ostatak implementacije bi trebalo da bude jasan u skladu sa diskusijom koju smo imali do sada.

Kako se ovako napisani metod koristi? Pomenuli smo da se u slučaju greške u RxJS toku može koristiti specijalni RxJS operator. Ako ste se prisetili da je u pitanju operator `catchError`, onda ste bili u pravu. Da se podsetimo: ovaj operator prihvata funkciju koja kao prvi argument prihvata grešku do koje je došlo i očekuje da ta funkcija vrati tok nad kojim će se izvršiti pretplata. Naše napisano zatvorenje, koje se dobija kao povratna vrednost metoda `handleError()`, tačno se uklapa u tu semantiku. Dakle, prilikom kreiranja HTTP zahteva, potrebno je ulančati operator `catchError()` i proslediti mu zatvorenje iz metoda `handleError()`. Primer je dat u narednom kodu:

Kod 8.85: angular/prodavnica/prodavnica-klijent/src/app/product/product.service.ts (linije 34-38)

```

34 public getProductById(id: string): Observable<Product> {
35     return this.http
36         .get<Product>(this.productsUrl + id)
37         .pipe(catchError(super.handleError()));
38 }

```

Jedina stvar koju smo ostali dužni da objasnimo jeste zašto metod `handleError()` mora da vrati zatvorenje umesto da bude implementiran kao dato zatvorenje. Posmatrajmo definiciju varijantne ovog metoda bez zatvorenja:

```

1 ...
2 export abstract class HttpErrorHandler {
3     constructor(private router: Router) {}
4
5     protected handleError(error: HttpErrorResponse): Observable<never> {
6         if (error.error instanceof ErrorEvent) {
7             console.error('An error occurred:', error.error.message);
8         } else {
9             this.router.navigate([

```

```

10         '/error',
11         { message: error.error.message, statusCode: error.status },
12     });
13 }
14 return throwError('Something bad happened; please try again later.');
```

Ovako definisan metod bi se mogao koristiti na sledeći način, na primer, u `ProductService`:

```

1 ...
2 export class ProductService extends ErrorHandler {
3     ...
4     public getProductById(id: string): Observable<Product> {
5         return this.http
6             .get<Product>(this.productsUrl + id)
7             .pipe(catchError(super.handleError));
8     }
9     ...
10 }
```

Primitimo da, s obzirom da je `handleError()` u ovoj implementaciji upravo ta funkcija koja obrađuje grešku, onda RxJS operatoru `catchError` prosleđujemo baš tu funkciju. Međutim, zbog načina na koji `catchError` funkcioniše, prilikom izvršavanja metoda `handleError()`, u okviru njene definicije, vrednost `this` neće biti referenca na objekat servisa `ProductService` kao što bismo to očekivali. Umesto toga, `this` će biti instanca tipa `CatchSubscriber`, zato što se `catchError` pretplaćuje na izvorni tok i kreira pretplatioca-posrednika između izvornog toka i našeg metoda za obradu grešaka. Zbog toga, metod `handleError()` će biti pozvan u kontekstu `CatchSubscriber` objekta i on neće imati svojstvo naziva `router`, te će pokušaj poziva metoda `this.router.navigate()` u definiciji metoda `handleError()` prijaviti izuzetak tipa `TypeError`.

Zašto zatvorenje rešava ovaj problem u prvoj (i jedinoj ispravnoj) implementaciji metoda `handleError()`? Kao što znamo, zatvorenja pamte kontekst u kojem su konstruisana, što znači da će zatvorenje zapamtiti šta je bila vrednost `this` prilikom njene konstrukcije, a to je upravo referenca na objekat servisa. U kojem god drugom kontekstu da to zatvorenje bude pozvano, ono je zapamtilo na šta je `this` pokazivalo i imaće pristup atributu `router`. Napomenimo da će doći do identičnog problema ukoliko se zatvorenje ne implementira pomoću lambda funkcije već pomoću klasične definicije anonimne funkcije ključnom rečju `function`. Ovo su nam sve poznati koncepti iz diskusije o jeziku JavaScript, ali treba ih se podsetiti, pogotovo u situacijama kada nije naočito razlog za pojavljivanje defekata koje kreiramo u kodu.

8.12.8 Dodatne mogućnosti

U ovom delu teksta smo prikazali samo osnovne koncepte rada sa HTTP zahtevima u Angular radnom okviru. Postoji još veliki broj koncepata o kojima nismo diskutovali, kao što su:

- Dodavanje i ažuriranje zaglavlja HTTP zahteva:
<https://angular.io/guide/http#adding-and-updating-headers>
- Konfigurisanje HTTP URL parametara:
<https://angular.io/guide/http#configuring-http-url-parameters>
- Pregled i prikaz progresu:
<https://angular.io/guide/http#report-progress>

- Presretanje HTTP zahteva i odgovora:
<https://angular.io/guide/http#intercepting-requests-and-responses>
- Automatsko ponovno pokušavanje kreiranja HTTP zahteva u slučaju neuspeha:
<https://angular.io/guide/http#retrying-a-failed-request>
- i mnogi drugi.

Postoji veliki deo implementacije naše klijentske aplikacije o kojima nismo diskutovali, kao što su kreiranje **DELETE** zahteva ili implementacija preostalih **GET** i **POST** zahteva koje server-ski API podržava. Ipak, implementacija klijentske aplikacije obuhvata i metode koje implementiraju ove zahteve, tako da se čitaocu savetuje da pregleda kompletnu implementaciju. Izvorni kod za ovu aplikaciju je moguće pronaći na lokaciji `primeri/angular/prodavnica/prodavnica-klijent`. Jedina stvar koja je ostavljena za vežbu jeste ažuriranje informacije o proizvodu.

Zadatak 8.3 Dopuniti implementaciju projekta `prodavnica-klijent` tako da se vrši ažuriranje informacija o proizvodu. Da bi korisnik mogao da ažurira informacije o proizvodu, potrebno je da se u komponenti `ProductInfoComponent` nalazi dugme koje prikazuje ili sakriva formular za izmenu podataka o tom proizvodu. Formular treba da ima polja kojim se menjaju naredne vrednosti: naziv, cena i opis. Omogućiti da se u tim poljima podrazumevano prikazuju trenutne vrednosti za dati proizvod. Implementirati obradu formulara. Pri pohranjivanju podataka iz formulara, ukoliko je formular uspešno validiran, poslati asinhroni HTTP zahtev tipa **PATCH** na URL `http://localhost:3000/products/productId`, pri čemu je `productId` parametar koji predstavlja identifikator proizvoda čiji se podaci ažuriraju. U telu zahteva smestiti vrednosti iz formulara. Serverska aplikacija već implementira svoj deo u komunikaciji. Omogućiti da se u slučaju uspeha korisniku prikaže poruka da su podaci ažurirani i da je neophodno da osveži prozor veb pregledača, a u slučaju neuspeha prikazati stranicu za grešku. Pratiti diskutovane idiome Angular radnog okvira, kao što su reaktivni formulari, implementacija HTTP zahteva u servisu, i sl. ■

Literatura za ovu oblast

- [Goo] Google. *Angular*. URL: <https://angular.io/>.
- [Mur+18] Nathan Murray i drugi. *Ng-book: The Complete Guide to Angular*. 8th. USA: CreateSpace Independent Publishing Platform, 2018. ISBN: 1985170280, 9781985170285.

