UNIVERZITET U NIŠU
ELEKTRONSKI FAKULTET Katedra za računarstvo

Razvoj biblioteke u Angularu za kreiranje materijala za korišćenje u nastavi

ZAVRŠNI RAD

Zadatak:

Upoznati se sa modernim alatima koji se koriste za razvoj veb-aplikacija. Detaljnije istražiti frejmvork Angular i u njemu implementirati biblioteku za za kreiranje materijala za korišćenje u nastavi.

Mentor: Doc dr. Ivan Petković
Kandidat: Lazar Ljubenović 14722

Komisija:

1. Doc dr. Ivan Petković	Datum prijave: _____
2. Prof dr. Suzana Stojković	Datum predaje: _____
3.	Datum odbrane: _____

Niš, 2018.

Sadržaj

Moderni veb

Internet je odavno postao sastavni deo života ljudi, a prvenstveno mladima. Predstavlja neiscrpan izvor zabave i informacija i povezuje ljude širom sveta. Ujedinjene nacije su 2016. godine proglasile pristup internetu osnovnim ljudskim pravom. Gugl je još 2011. godine pustio u prodaju Hrombuk, lap-top u kome su operativni sistem i brauzer spojeni u jedno. Sve više aplikacija se izvršavaju direktno u brauzeru, umesto klasičnih desktop aplikacija.

Ovo ima brojne prednosti, ali se sledeće dve izdvajaju kao ključne.

Veb-aplikacije od korisnika **ne zahtevaju instalaciju**. Mnogo je veća verovatnoća da će potencijalni klijent koristiti aplikaciju ukoliko može da je koristi čim naide na nju, umesto da mora da je preuzme i instalira na svojoj mašini. Sem toga, danas nije retkost da korisnici poseduju više uredaja koje svakodnevno koriste. Veb-aplikacija je po definiciji dostupna sa svakog od njih, dok bi desktop- ili mobilnu aplikaciju trebalo instalirati na svakom uredaju ponaosob.

Pored ovoga, **ažuriranje** klasičnih desktop aplikacija zahteva neki vid akcije od korisnika. Čak i ako je ažuriranje dobro isplanirano i može da se odvija automatski i bez čitave ponovne instalacije programa, korisnici uvek imaju mogućnost da ipak ostanu na staroj verziji. Developer ne može da bude siguran u to koju verziju aplikacije koriste klijenti, što za posledicu ima razne probleme u kompatibilnosti prilikom, na primer, komunikacije sa serverom, koji bi morao da uvek podržava sve verzije desktop aplikacije. Veb-aplikaciju je trivijalno ažurirati; garantovano je da će svi korisnici istog trenutka moći da pristupe jedino najnovijoj verziji.

1.1 Jednostranične aplikacije

Sadržaj na internetu je prvobitno zamišljen kao skup statičkih dokumenata koji su međusobno povezani. Praćenjem tih veza („klikom na link”), korisnik pomoću brauzera sa svog računara (klijent) izdaje udaljenom i znatno moćnijem računaru

(server) zahtev za pribavljanje nove stranice, koristeći URL kao identifikator. Kao odgovor na zahtev, server klijentu vraća novu stranicu, koju klijent iznova renderuje. Za izgradnju ovakvih statičkih stranica dovoljno je koristiti HTML (*HyperText Markup Language*), kojim se definiše semantička struktura sadržaja stranice, i CSS (*Cascading Style Sheet*), koji služi za deklaraciju stilova, tj. načina prikaza (prezentacije) dokumenata pisanih u HTML-u.

Kada brauzer primi stranicu od servera, on je parsira. Na osnovu HTML-a kreira strukturu podataka u vidu stabla koja se naziva DOM (*Document Object Model*), a na osnovu CSS-a se na sličan način generiše CSSOM (*Cascading Style Sheets Object Model*). Na osnovu ova dva stabla brauzer može da renderuje stranicu.

Statičke veb-strance su vremenom zamenile dinamičke veb-stranice, koje su korisnicima omogućile odredeni vid interakcije. Da bi stranica bila dinamička, osim HTML-a i CSS-a mora se pisati i kod u JavaSkriptu, kojim se programira način interakcije s korisnikom. Brauzeri nude API ka DOM-u koje programer može iskoristiti da osluškuje dogadaje (klik mišem, pritisak dirke na tastaturi, itd), i da programski manipuliše strukturom DOM-a. Ovo znači da stranica može promeniti svoju strukturu i izgled na osnovu akcija korisnika, bez potrebe da se sa servera dobavi cela nova stranica, i bez potrebe da je brauzer iznova parsira i renderuje.

Takav način ponašanja stranice se u početku koristio samo za neke manje izmene na stranici koje je bilo trivijalno promeniti JavaSkriptom, npr. za validaciju formi na klijentu: korisnik odmah nakon unosa lozinke može biti obavešten da ona mora imati više od odredenog broja karaktera (umesto da pošalje formu, da se na serveru detektuje nevalidna lozinka, i da se vrati nova stranica sa upisanom greškom). Ovaj vid funkcionisanja predstavlja preteču veb-aplikacija, kod kojih se sve više funkcionalnosti događa na klijentu umesto na serveru.

Veb-aplikacija je klijentska aplikacija kod koje se sav korisnički interfejs izvršava u okviru brauzera. Ne postoji jasna granica izmedu „dinamičke veb-stranice" i „veb-aplikacije"; za sajtove koji po izgledu više podsećaju na desktop ili mobilnu aplikaciju postoji veća šansa da se proglase veb-aplikacijama. Medu veb-aplikacijama se posebno izdvajaju jednostranične aplikacije (SPA, *Single-Page Application*) jer se potpuno odvajaju od tipične veb paradigme gde se korisnik kreće kroz stranice koje imaju različite URL-ove.

1.1.1 Tehnologije na koje se oslanja SPA

U osnovi svake SPA aplikacije leži Ajaks (AJAX). Striktno govoreći, ne radi se o tehnologiji, već o skupu više tehnologija, od kojih se kao najznačajnija izdvaja `XMLHttpRequest` koja u kombinaciji sa HTML-om, CSS-om, JavaSkriptom, DOM-om i XML-om ili JSON-om omogućava inkrementalno ažuriranje korisničkog interfejsa bez slanja zahteva ka serveru za novu HTML stranicu, i samim tim bez osvežavanja čitave stranice. Na ovaj način je aplikacija znatno responzivnija, u smislu da brže odgovara na zahteve korisnika.

Mada je X u AJAX oznaka za XML, za razmenu podataka se relativno brzo sa XML-a prešlo na JSON, iz razloga što predstavlja kompaktniji zapis podataka, ali i zbog toga što od 2009. godine JavaSkript ima ugradenu funkciju za parsiranje JSON formata u JavaSkript objekat.

Međutim, vremenom se pokazalo da je XMLHttpRequest nezgrapan i previše komplikovan za rad. Dolazak prototipa `Promise` u ES2015 iskorišćen je za definiciju novog standarda za obavljanje Ajaks zahteva pod nazivom `fetch`.

1.1.2 Prednosti

Kao prednosti jednostraničnih aplikacija izdvajaju se tri glavne osobine.

Jednostranične aplikacije su **brze**. Većina resursa se učitava samo jednom (lejaut, stilovi, skripte). Samo se podaci razmenjuju sa serverom. Na primer, u aplikacijama kao što su mejl-klijenti, struktura aplikacije ostaje ista tokom navigacije kroz nju. Umesto da se otvaranjem mejla iznova učitava čitava stranica, dobavlja se samo sadržaj mejla i zatim se JavaSkriptom taj sadržaj ubacuje u odgovarajuću sekciju sajta – ostatak stranice nema potrebe da se ponovo preuzima sa servera.

Iskustvo korisnika je znatno bolje prilikom korišćenja jednostraničnih aplikacija spram klasičnih veb-sajtova. Ovo je direktna posledica goreopisane brzine. Ako se iskoriste neke tehnike za optimistično učitavanje podataka *pre* nego što ih korisnik zatraži, moguće je učiniti da se promene koje korisnik zahteva dogode (naizgled) istog trenutka.

Pošto jednostranične aplikacije konzumiraju **API samo za razmenu podataka**, identičan API se može iskoristiti i za druge aplikacije; na primer, mobilna aplikacija može gadati iste tačke na API-ju, a interno na drugačiji način prikazivati te podatke. Ovo je nemoguće ostvariti u tradicionalnoj paradigmi jer server vraća čitavu HTML stranicu, koja je mobilnom klijentu beskorisna.

1.1.3 Mane

Mane jednostraničnih aplikacija su mnogo suptilnije i uglavnom se tiču činjenice da one ne mogu u potpunosti iskoristiti funkcionalnosti brauzera, već da moraju manuelno iznova implementirati te funkcionalnosti u JavaSkriptu.

Brauzeri čuvaju **istoriju posećenih stranica** što čini povratak na prethodnu stranicu veoma brzim. Kada korisnik pritisne dugme za navigaciju unazad, on očekuje da se promena dogodi uz veoma malo kašnjenje i da stranica bude u sličnom stanju u odnosu na ono kada je poslednji put bila prikazana na ekranu.

Kada je sajt izgraden tradicionalnim modelom, brauzer će biti u stanju da iskoristi keširanu verziju stranice i povezane resurse. Naivna implementacija jednostranične aplikacije će povratak nazad poistovetiti sa bilo kojom drugom akcijom, odnosno biće okinut dogadaj koji će detektovai osluškivač u JavaSkriptu,

što će kao posledicu imati ponovno ispaljivanje zahteva ka serveru za istim podacima; dakle javlja se dodatna latentnost i (verovatno) vizuelna promena.

Da bi korisnici mogli da imaju zadovoljavajuće iskustvo prilikom navigacije kroz jednostraničnu aplikaciju, slična funkcionalnost se mora implementirati u JavaSkriptu. Aplikacija treba da kešira stranice korišćenjem memorije, lokalnog skladišta, baze podataka na strani klijenta ili kolačića. Sem toga, aplikacija mora da odluči i kada da pokupi te stranice i iskoristi ih. Da bi se ovo rešilo, neophodno je da se napravi razlika izmedu klika na link ili kucanja adrese direktno u brauzer, i pritiskom na dugmad back i forward na brauzeru. Medutim, iz bezbednosnih razloga, nemoguće je iz Javaskripta ustanoviti na koji je način korisnik navigirao na drugu stranicu.

Sličan problem postoji i kod čuvanja **pozicije skrola**. Brauzeri pamte poziciju skrola na stranicama koje su ranije posećene. Pošto se kod jednostraničnih aplikacija stranica nikada ne menja, brauzer nije svestan promene „stranica", pa čuva poziciju skrola čak i kada se to ne očekuje. Na primer, ako korisnik klikne na link na dnu stranice, kada mu se otvori „nova" stranica, brauzer neće znati da treba da ažurira poziciju skrola na vrh, pa će korisnik biti odveden na dno.

Ovo je će biti moguće delimično zaobići korišćenjem *Custom Scroll Restoration* API-ja. Međutim, u slučaju da se prethodno dobavljeni podaci ne keširaju, ili da postoji bilo kakvo kašenjenje između otvaranje stranice i prikaza asinhrono podataka, pozicija skrola će i dalje biti pogrešna.

Zatim, brauzeri će korisnike obavestiti da imaju **popunjenu formu** koja još uvek nije poslata kada probaju da navigiraju na neku drugu stranicu. Ovo je implementirano kroz dogadaj `beforeunload` na koji se developer može pretplatiti. Medutim, kod jednostraničnih aplikacija ovaj dogadaj postaje beskoristan jer brauzer ne pravi zahteve ka pravim stranicama. Developer bi morao lično da bude odgovoran da pruži ovakvo iskusvo korisnicima.

1.2 Sazrevanje JavaSkripta

1.2.1 Nastanak

JavaSkript je nastao maja 1995. godine kao „jezik-lepak" za tadašnji dominantni brauzer Netskejp. Prvobitno nazvan Mocha, četiri meseca kasnije biva prekršten u LiveScript da bi tri meseca nakon toga dobio ime po kojem je i danas poznat: JavaScript. Ovaj jezik je nastao iz potrebe za premošćavanjem jaza izmedu brauzera kao korisničkog okruženja za prikaz dokumenata i jezika za opis tih dokumenata, HTML-a. Definisao ga je Brendon Ajk za samo deset dana.

Sa pojavom drugih brauzera, pojavili su se i jezici koji su očigledno bili inspirisani JavaSkriptom, ali njihovi stvaratelji nisu smeli da ga zvanično tretiraju kao

JavaSkript zbog autorskih prava koji je imao San Majkrosistems. Na primer, JavaSkript je poslužio kao podloga Adobiju za jezik ActionScript, i sada predstavlja njegov zvaničan dijalekat. Ipak, najpoznatiji primer kopije JavaSkripta je nesumnjivo JScript koji je došao uz Majkrosoftov Internet Eksplorer u verziji 3, već krajem 1996. Osim u imenu, razlikovao se i u nekim detaljima u implementaciji, ali i u API-ju za komunikaciju sa DOM-om. Iz tog razloga je ubrzo organizovana komisija za standardizaciju JavaSkripta, nazvana ECMA.

1.2.2 Pokušaji popravke

Osnova za moderni JavaSkript definisana je 1999. godine u trećoj verziji standarda, ES3. Počelo se sa radom na četvrtoj verziji, koja je trebalo da bude još radikalnija i da ispravi neke „greške" u JavaSkript standardu. Najavljivane promene su bile toliko velike da su čak dovele do toga da ES4 bude postane poznat kao „JavaSkript 2". Medutim, komitet za standardizaciju je bio podeljen: dok se jedna strana zalagala za poboljšanje jezika zarad bolje budućnosti, druga se bojala da će ovime nastati prevelika pometnja, jer bi sve veb-stranice koje koriste "stari" JavaSkript prestale da rade zbog promene u sintaksi jezika (tzv. „slamanje veba"). Sukob je aktivno trajao do 2003, kada je projekat zvanično napušten.

Dve godine kasnije, osnivač JavaSkripta Ajk u saradnji za Mozilom počinje da radi na projektu E4X. Pridružuje im se i Makromedija (sada Adobi), u nadi da će standardizovati svoj ActionScript3 u saradnji sa ECMA-om, i time ponovo spojiti razdeljene dijalekte jezika. Ipak, razlika je bila previše velika, što su obe strane počele da shvataju krajem 2007. godine. Otprilike u to vreme, Daglas Krokford, koji je tada radio u Jahuu, udružuje snage sa Majkrosoftom kao opozicija promenama koje treba da budu definisane standardom ES4, a svoje stavove iskazuju tako što standard žele da nazovu jednostavno ES3.1, jer unosi samo neznatne promene.

Sve se ovo dešava u vreme kada je JavaSkript zajednica pokrenula revoluciju u mogućnostima koje pruža JavaSkript, što nesumnjivo otpočinje 2005. godine kada je Džesi Džejms Geret skovao termin Ajaks koji je iskoristio za opis skupa tehnologija izgradenim nad JavaSkriptom. Ovo je bio renesansni period za jezik, i tada počinju da se javljaju biblioteke otvorenog koda kao što su jQuery, Mootools i Dojo koje su do pojave SPA rejmvorka bile dominantni alati za izgradnju dinamičkih veb-sajtova i veb-aplikacija.

Iz rata oko konačnog oblika četvrte verzije standarda koji se konačno okončava 2009. godine izlazi Krokford. U JavaSkript je posle dvanaest godina uneto svega nekoliko neznatnih promena, a rezultujući standard je preimenovan u ES5, kako ne bi došlo do kasnije zabune oko eventualne četvrte verzije o kojoj se već godinama govori. U standard tada dolazi takozvani strict mode, koji definiše "strožu" verziju jezika, ali na kompatibilan način, kako ne bi narušio već postojeći kod koji se nalazi na vebu.

Dve godine kasnije usledila je mala promena standarda, editorijalnog karaktera, nazvana ES5.1, koja nije bila od većeg značaja za JavaSkript.

1.2.3 Godišnji ciklusi

Nove osobine JavaSkripta koje nisu ušle u ES5 standard ostale su poznate pod imenom Harmonija (*Harmony*). Tek 2015. godine se ECMA komitet ponovo sastaje radi definisanja standarda šeste edicije standarda, kada se konačno, posle petanest godina nagomilanih ideja za proširene i poboljšanje jezika, sprovode u delo. Šesta edicija standarda je vrlo kratko pre zvaničnog objavljana preimanovana iz ES6 u ES2015, zbog ideje da se u nastavku standard za JavaSkript obnavlja na svakih godinu dana, kako naredne promene ne bi bile toliko velike kao ova.

ES2015 je drastično promenio izgled JavaSkripta, čime je omogućeno znatno jednostavnije pisanje složenih aplikacija. Nesumnjivo najznačanija novina u jeziku su moduli koji omogućuju podelu koda u fajlove, ali je tu i intuitivnija sintaksa za defisanje klasa, iteratori i generatori (koji će poslužiti kao osnova za async/await u ES2017), generatorski izrazi, `for...of` petlje, „streličaste" funkcije za sintaksno jednostavnije pisanje funkcionalnog koda, binarni podaci, tipizirani nizovi, kolekcije (mape, skupovi, slabe mape), promisi, dodatne funkcije na 'Math' objektu, bolja refleksija, kao i posrednici za metaprogramiranje virtuelnih objekata i omotača.

Sedma edicija objavljena 2016. godine uvodi blage promene: operator za stepenovanje `**` i `Array#includes`. Sledeća edicija ponovo uvodi dosta promena u jezik koje omogućavaju da se procesorska moć iskoristi na znatno višem nivou nego ranije. Medu novinama koje ES2017 uvodi izdvaja se mogućnost paralelnog programiranja, izvodenja atomičnih operacija ukoliko ih procesor podržava, nadgledanja tokova podataka (observable streams), definisanja tipova podataka koji omogućuju SIMD programiranje. Sem toga, uvedena je sintaksna integracija promisa i generatora pomoću ključnih reči `async` i `await`, a bolji Simboli predstavljaju korak bliže predefinisanju operatora.

Presek za ES2018 napravljen je juna 2018. godine, a u novine se ubraja korišćenje *rest/spread* sintakse za objekte pored nizova, asinhrona iteracija (`for await...of` petlje), metoda `finally` nad prototipom promisa i imenovane grupe u regeksima.

1.2.4 Odbor TC39

Proces usvajanja nove konstrukcije u jezik nije jednostavan i ne dešava se preko noći. ECMAScript je dizajniran od strane odbora koji se naziva TC39. Sačinjen je od kompanija među kojima su i kreatori svih značajnijih brauzera. Redovnim

sastancima prisustvuju delegati koje biraju kompanije članice, a nekada se pridružuju i spoljni eksperti za određene oblasti. Sažeci sastanka su javno dostupni onlajn.

Odbor je organizovan uz izdanje ES2016, kada je zvanično potvrđeno da će se jezik nadograđivati na godišnjem nivou. Konačnu odluku o uvođenju osobine u JavaSkript donosi većina iz odbora, ali pod uslovom da nijedna članica nije strogo protiv ideje. Kako su mnoge članice kompanije koje rade na razviću brauzera, za njih slaganje sa promenom jezika sa sobom nosi i teret odgovornosti oko njihove implementacije.

Svaki predlog za dopunu jezika prolazi kroz nekoliko *faza zrelosti*, počev od nulte faze. Prelazak u sledeću fazu mora biti odobren od strane odbora. Svaki predlog počinje kao neformalna, tzv. **slamnata** (*strawman*) ideja – ovo je faza 0. Ukoliko TC39 smatra da predlog ima potencijal, on se dodaju na listu slamnatih predloga.

U fazi 1 se za ideju formalno može govoriti kao o **predlogu** (*proposal*). Za predlog se bira jedan *šampion* koji je za njega odgovoran, pri čemu jedan od šampiona i ko-šampiona mora da bude član odbora. U ovoj fazi se daje tekstualni opis ideje, praćen primera, diskusiji o API-ju, semantici koja se uvodi u jezik i algoritmima koji će se koristiti za obavljanje metoda koje se uvode u jezik. Takođe se identifikuju potencijalne prepreke u usvajanju predloga u specifikaciju jezika. Ukoliko predlog to dozvoljava, neophodno je da se implementiraju polifilovi i nekoliko demonstracija koje ih koriste.

Prva faza samo iskazuje da je odbor zainteresovan za predlog. Tek kada predlog pređe u drugu fazu, počinje da se piše **nacrt** (*draft*) – to je prva verzija onoga što treba da postane deo formalne specifikacije jezika.

TajpSkript

TajpSkript (*TypeScript*) je jezik koji podseća na JavaScript i služi kao njegova zamena za pisanje velikih aplikacija. Dodaje opcione tipove i TC39 predloge koji još uvek nisu ušli u standard, ali su barem u fazi 3. Tajpskript se kompajlira (preciznije, „transpajlira") u čitljiv JavaSkript.

2.1 Klasifikcija sistema tipova

Skup pravila na osnovu kojih se svojstvo poznato kao **tip podatka** dodeljuje zaličitim delovima kompjuterskog programa (promenljive, izrazi, funkcije, moduli, itd) naziva se **sistem topiva podataka**. Služe da bi se formalizovali koncepti koje programeri koriste za algebarske vrednosti, strukture podataka i druge komponente programa, kao što su „logička vrednost", „niz stringova" i „funkcija koja vraća broj".

Glavna namera postojanja tipova podataka jeste da minimizuje šanse za pojavu bagova u programima i pruži mogućnost za bolje isksutvo tokom razvoja softvera, ali se može koristiti i kao sredstvo za sprovođenje optimizacije. **Provera tipova** (*type checking*) je proces kojim se utvrđuje da li su ispoštovana pravila i ograničenja koja nameću tipovi podataka u nekom programskom jeziku. Postoji nekoliko osobina na osnovu kojih se može izvršiti podela načina na koje se vrši provera tipova.

2.1.1 Podele

Prvo pitanje koje se nameće jeste *kada* se dešava provera: da li tokom kompilacije ili tokom izvršenja programa. Za jezike koji se interpretiraju (JavaSkript, Pajton, Rubi) ne postoji faza kompilacije pa je jedino mesto gde se može vršiti provera tipova – tokom izvršenja. Za ovakve jezike – u kojima se provera tipova sprovodi tokom izvršenja – kaže se da imaju **dinamičku** proveru tipova. Interpretatori za ovakve jezike obično svakom objektu dodeljuju oznaku tipa u kojoj se nalaze

informacije o tipu. Ukoliko se provera vrši tokom kompjaliranja, radi se o **statičkoj** proveri. Ako program zadovolji uslove koje nameće analizator kod jezika sa statičkom proverom tipova, onda to daje garanciju da program tokom izvršenja neće da naiđe na određeni skup mogućih grešaka u vezi sa tipovima podataka.

Poznajući trenutak izvršenja provere tipova podataka, postavlja se pitanje *kako* se ona sprovodi. Kod jezika sa **jakim** tipovima podataka, nad kodom su dozvoljene samo one operacije koje imaju smisla u skladu sa semantikom jezika i one koje neće dovesti do gubitka informacija. Za jezike koje ne nameću ova ograničenja kaže se da imaju **slabe** tipove podataka. Ne postoji jasna granica između jakih i slabih tipova, pa se ovi termini koriste uglavnom za poređenje jezika (koji ima „jače" a koji „slabije" tipove) ili za određene osobine jezika (za koju se može reći da je „jaka" a za koju „slaba").

JavaSkript nema podršku za statičko definisanje tipova podataka, a uz to je i slabo tipiziran jezik. Promenljiva kojoj je dodeljen string može u sledećem redu da ima referencu na niz brojeva, a zatim da u nju bude upisana logička promenljiva. Kako je JavaSkript jezik koji se interpretira, ni nema prilike da se ovo zabeleži kao greška prilikom kompajliranja; taj proces ne postoji. Međutim, čak i tokom izvršenja programa, ovakvo ponašanje koda se neće tretirati kao greška i interpretator će bez problema upisivati bilo koju vrednost u promenljivu.

Ovakvo ponašanje ne samo da može negativno da utiče na performanse (jer optimizator nije u stanju da predvidi koliko memorije je potrebno da se zauzme za promenljivu), već čini kod nečitljivim i podložnim greškama. Glavna prednost TajpSkripta je uvođenje tipova podataka u JavaSkript, ali samo prilikom kompajliranja.

TajpSkript ni na koji način ne utiče na fazu interpretiranja i izvršenja koda. TajpSkript dolazi uz kompajler koji postojeći TajpSkript kod kompajlira u JavaSkript. Dobijeni JavaSkript kod ne sadrži nikakve informacije i tipovima podataka; ne postoje nikakvi uslovi koji će dovesti do greške ukoliko se promenljivoj dodeli drugačiji tip od navedenog. Umesto toga, TajpSkript ima zadatak da tokom kompajliranja utvrdi tipove i da se postara da developera obavesti o nastaloj nekonzistentnosti tako emituje grešku.

Da bi se ovo postiglo, TajpSkriptov kompajler obavlja statičku analizu koda tokom koje zaključuje tipove i proverava da li su određeni operatori izvodljivi nad njima. Na primer, iako je se JavaSkript interpretator ne buni za pokušaj sabiranja praznog niza i praznog objekta (`[] + {}`), TajpSkript će prijaviti grešku „Operator '+' cannot be applied to types '`undefined[]`' and '`{}`'".

2.1.2 Deklaracije

Kako bi bilo moguće da se koriste postojeće biblioteke pisane u JavaSkriptu, u TajpSkript je uveden pojam *deklaracije*. Na primer, TajpSkipt ne može automatski

da bude svestan postojanja eksterne biblioteke učitane kroz `script` tag u `head` sekciji HTML dokumenta za koju se piše TajpSkript kod. Deklaracija promenljive ili nekog drugog simbola navodi se ili u posebnom fajlu sa ekstenzijom `.d.ts` ili direktno u postojeći fajl, uz prefiks `declare`.

Deklaracije za mnoge postojeće JavaSkript biblioteke se mogu naći na GitHab repozitorijumu Majkrosoftovog projekta *DefinitelyTyped* koji održava zajednica. Svi paketi se mogu preuzeti sa npm repozitorijuma. Radi lakšeg pronalaska deklaracionih fajlova koristi se imenski prostor (*namespace*) `@types`, a za ime deklaracionog paketa se koristi isto ime koje je dodeljeno samom paketu za koji se instaliraju deklaracije. Na primer, za paket `jquery` deklaracije se nalaze na `@types/jquery`.

Deklaracije se mogu dostaviti i zajedno sa samim paketom; ovaj način se preferira u situacijama kada je sam paket već napisan u TajpSkriptu. Koristeći opciju `-declaration`, TajpSkriptov kompajer može, pored `.js` fajlova, generisati i `.d.ts` fajlove, a iz `package.json` vrednosti pod ključem `types` može se pročitati putanja do ovog fajla. Ovako TajpSkriptov kompajler zna gde da pronađe potrebne deklaracije.

2.1.3 Definisanje tipova

Neki stariji jezici sa statičkim tipovima podataka od developera zahtevaju da se tipovi uvek definišu eksplicitno. TajpSkript je specifičan po tome što su se njegovi stvaraoci postarani da održe barijeru prelaska od JavaSkripta na TajkSkript izuzetno niskom. To je postignuto skroz nekoliko osobina.

Uz odgovarajuća podešavanja kompajlera, migracija na TajpSkript se sastoji od samo jednog koraka: promeniti ekstenziju svim `.js` datotekama u `.ts`. Ovo je omogućeno činjenicom da su tipovi podataka **opcioni**. Svaki JavaSkript kod može se formalno prevesti u TajpSkript kod promenom ekstenzije, a TajpSkriptov kompajer će u procesu kompajliranja od `.ts` datoteke generisati kod identičan polaznoj `.js` datoteci. TajpSkript je zbog ovoga s namerom zamišljen kao **nadskup** JavaSkripta.

Za **eksplicitno** definisanje tipova koristi se postfiksna sintaksa koja je popularna među jezicima koji pružaju opcione tipove (npr. *EkšnSkript* i F♯). Ispred tipa se navode dve tačke (`:`).

```
const foo: string = "Hello"
```

Informacije o tipovima često nije neophodno ručno specificirati jer tipovi u TajpSkriptu mogu biti **implicitni**. Analizator će, gde god je to moguće, na osnovu analize koda zaključiti o kom je tipu reč. Na primer, ukoliko se konstanta inicijalizuje brojnom vrednošću, za konstantu se implicitno zaključuje da ima tip `number`. Slično, ako se iz funkcije koja prima dva stringa vrati vrednost dobijenja njihovom konkatenacijom pomoću polimorfnog operatora `+`, povratnoj vrednosti

funkcije se implicitno dodeljuje tip ‘string‘ jer je upravo tog tipa rezultat primene ovog operatora nad dva stringa.

```
const foo = 2103
const concat = (a: string, b: string) => a + b
```

2.1.4 Strukturalni sistem tipova

Tradicionalni objektno-orijentisani jezici kao što su Java i C koriste sistem tipova u kome se podudaranje između tipova podataka vrši na osnovu njihovih eksplicitno navedenih *imena*. Ovakvi sistemi nazivaju se **nominalni**. Na primer, iako su definisane dve strukture istog oblika, njihova međusobna dodela neće biti moguća jer su imena tipova različita.

```
struct Foo { int baz; };
struct Bar { int baz; };

int main () {
  Foo foo;
  Bar bar;
  foo = bar; // error: no viable overloaded '='
}
```

S druge strane su jezici kod kojih je za određivanje privila u sistemu tipova presudna njihova *struktura*. Oni se nazivaju **strukturalni** sistemi, u koje spada TajpSkript.

```
interface Foo { baz: number }
interface Bar { baz: number }

let foo!: Foo
let bar!: Bar
foo = bar // ok
```

2.1.5 Konfiguracija

Skup fajlova koje će biti obrađene od strane kompajlera naziva se **kontekst kompilacije** (*compilation context*). Mada je sva podešavanja moguće proslediti direktno kao argumente CLI aplikacije, obično se, radi preglednosti, za ovo koristi konfiguracioni fajl. U pitanju je JSON5[1] fajl za koji se očekuje da bude imenovan `tsconfig.json` i da se nalazi u korenom direktorijumu projekta. Ukoliko to nije slučaj, putanja do konfiguracione datoteke se može proslediti kao argument opcije `--project` (`-p`).

[1] Nadskup JSON specifikacije u kome se, između ostalog, dopuštaju viseći zarezi i komentari, a omeđivanje ključeva znacima navoda je opciono ukoliko ne sadrži karaktere koji bi doveli do dvosmislenosti; zbog ovoga dobija naziv „JSON za ljude”.

Za uključivanje i isključivanje datoteka iz projekta koriste se tri polja u konfiguracionoj datoteci: `files`, `include` i `exclude`. Svi primaju niz stringova. Ukoliko se ne navede nijedna od ove tri opcije, biće uključene sve `.ts` datoteke koje se nalaze u korenom direktorijumu i poddirektorijumima, rekurzivno.

Da bi se naznačilo da datoteka ne pripada projektu, potrebno je dodati je u niz `exclude`. Pored putanja do datoteka, `exclude` opcija prepoznaje i imena direktorijuma i glob[2] šablone.

Slično, `include` opcija se koristi da bi se ograničio opseg projekta. Takođe razume putanje do datoteka, direktorijuma i blobove. Na primer, sve izvorne datoteke se često nalaze u direktorijumu `src`. Opcija `files` je specifičnija od opcije `include` jer prima samo putanje do datoteka i uglavnom se koristi za manje projekte.

Ponašanje sistema tipa podatka i način na koji će TajpSkript generisati izlaznu datoteku ili datoteke definiše se u sklopu objekta `compilerOptions`. Ovaj ključ prima objekat od preko osamdeset podešavanja. U nastavku će biti iznesene samo najznačajnije; cela lista se može naći na

Izbor ES verzije

Osim što uvodi tipove podataka u JavaSkript, TajpSkript služi i za izbor standarda koji aplikacija treba da podrži. Kod je moguće pisati ne samo u tekućoj verziji standarda, već i koristiti predloge koji još uvek nisu ušli u standard.

Kako ne podržavaju svi brauzeri sve TC39 predloge, developer treba da odluči po kom standardu treba da bude napisan emitovan kod. Ovo se određuje poljem `target`.

Najstariji podržani standard je `es3`, i ovo je podrazumevana vrednost za `target`. Moguće je izabrati i konkretne standarde `es5`, `es2015`, `es2016`, `es2017` i `es2018`. Ukoliko developer ne želi da se kod adaptira ni za jedan standard već da ostane u izvornom obliku (naravno, bez tipova), onda se koristi vrednost `esnext`; ovo znači da će izlazni kod sadržati JavaSkript koji još uvek nije postao standard, pa postoji dobra šansa da kod bude nevalidan u nekim brauzerima.

Moduli

Kao glavni nedostatak JavaSkripta se često navodilo to što nema specifikaciju za podelu koda u manje datoteke. Najraniji alati koji su slućili da nadomeste ovaj nedostatak su radili jednostavnu konkatenaciju. Sa pojavom Noda se javio prvi formalni standard; koristi se funkcija `require` za uvoz datoteka i posebna globalna promenljiva `exports` za izvoz simbola iz njih. Ovaj standard je kasnije

[2] Glob je string kojim se na koncizan i čitljiv način definiše skup datoteka i/ili direktorijuma; koristi specijalne simbole kao `*`, `**` i `?` da bi se zadale određene „komande”. Primeri: `*.txt`, `node_modules/**/*`, `*.tsx?`.

dobio ime **CJS** (*CommonJS*). Iako je ovaj standard bio specifičan za Nod i iako nije bio prihvaćen kao JavaSkript standard, javili su se alati koji dopuštaju developerima da kod namenjen izvršenju u brauzerima (a ne u Nodu) pišu u odvojenim fajlovima i da reference na simbole definišu eksplicitno koristeći `require` i `exports` umesto da se oslanjaju na primitivne metode kao što je konkatenacija. Ovi alati su bili prvi **bandleri**. Kao izlaz su davali isključivo jednu `.js` datoteku, tako što su zavisnosti među izvornim datotekama praćene od jedne početne.

Aplikacije su vremenom rasle a SPA metoda za izgradnju aplikacija se koristila sve češće. Postalo je jasno da se dobar deo koda koji se isporučuje korisnicima nikada ne izvrši, jer korisnik nikada ne dođe do određenog dela aplikacije. Da bi se ovaj problem premostio, zajednica je definisala specifikaciju poznatu kao **AMD** (*Asynchronous Module Definition*). Ovime je omogućeno da se definišu zavisnosti između datoteka u fazi izvršenja programa na klijentu i da se pribave asihrono, kada se za to javi potreba. Pošto su zavinosti eksplicitno definisane, dodatne zavisnosti se pribavljaju automatski, pa se s developera skida teret da mora ručno da definiše koje datoteke se prve moraju učitati da bi se učitala druga.

Osim pisanja JavaSkripta kroz AMD module, moguće je i pisati ih u odvojenim datotekama a prepustiti nekom devop alatu da ih pretvori u AMD module, bilo da su oni u jednoj datoteci ili u više njih.

I CJS i AMD su dosegli dovoljnu popularnost da se smatraju štandardom-- jedan za Node, a drugi za brauzere. Ali šta je sa modulima za koje ima smisla da istovremeno budu dostupni i u Nodu i u brauzeru; na primer, što su biblioteke sa pomoćnim funkcijama kao što je *lodash*?

Kako ne bi postojale dve različite verzije datoteka, osmišljena je nova specifikacija, nazvana **UMD** (*Universal Module Definition*). UMD ne samo da je kompatibilan i sa CJS-om i sa AMD-om, već radi i sa globalno definisanim simbolima.

Paralelno sa ovim, sa porastom popularnosti JavaSkripta i veba kao vodeće platforme za razviće aplikacija, počinje da se intenzivnije radi se na specifikaciji Harmonije, nove verzije EkmaSkript specifikacije koja je danas poznata kao ES2015. Ovime je problem rada sa više fajlova formalno rešen.

Međutim, postoje dva razloga zašto većina developera ne koristi ES2015 module u produkciji. Prvi je to što i dalje postoji određeni procenat ljudi koji koristi brauzere koji nisu implementirali ovaj standard. Drugi je to što se povećava broj zahteva i aplikacija se usporava.

Osim toga što se zahteva više datoteka (što predstavlja više zahteva ka serveru pa samim tim dužu komunikaciju sa istim), ostaje problem minifikacije i drugih tehnika optimizacije koje ili nisu izvodljive ili nisu u dovoljnoj meri izražene. Zbog ovoga se i dalje koriste bandleri koji na složeni način obarđuju fajlove kako bi pronašli njihove međuzavisnosti i od njih stvorile jedan ili više izlaznih datoteka spremnih za produkciju.

Da bi TajpSkript zadovoljio široke potrebe developera, podržava više načina na generiše module. Opcija `module` prima jednu od sledećih vrednosti: `None`, `CommonJS`, `AMD`, `System`, `UMD`, i `ES2015`.

Ovo polje je usko povezano sa poljem `target`; `ES2015` je moguće koristiti samo ako je `target` podešeno na `es5` ili niže. S druge strane, jedino ako se izabere `AMD` ili `System` je moguće definisati opciju `outFile` kojom se podešava ime fajla u kojem će se naći izlazni kod.

Globalni tipovi

Osim jezičkih konstrukcija, novi ES standardi u jezik uvode i nove globalne promenljive ili nove metode i propertije nad postojećim prototipima. Da bi se TajpSkript kompajlirao bez greške i da bi pružio podršku u vidu tipova podataka za sve metode, developer mora da naglasi koje tipove ili koje grupe tipova želi da vidi globalno dostupne, odnosno za šta očekuje da postoji tokom izvršenja koda u okruženju za koje piše softver.

U ovu svrhu se opciji `lib` prosleđuje niz stringova. Pored celih standarda (od `ES5` do `ES2018`), mogu se uključiti i `ESNext` (kojim se pokrivaju deklaracije za predloge koji još uvek nisu postali standard), `WebWorker` (deklaracije za pokretanje koda u okviru veb-vorkera), kao i određene deklaracije iz standardā pojedinačno: na primer, `ES2015.Core` (osnovni skup funkcionalnosti), `ES2015.Promise` (deklaracije za objekte tipa `Promise` iz 2015), `ES2016.Array.Include` (dodata metoda `include` nad prototipom globalnog objekta `Array`), `ES2018.Promise` (deklaracije za novi `Promise` gde je uključena metoda `finally`), itd.

Emitovanje koda

Sem izbora načina na koji se kod parsira i ciljne verzije standarda, moguće je definisati i *šta* treba da se emituje.

Ukoliko je TajpSkript potrebno koristiti samo da bi se proverili da li su zadovoljeni tipovi podataka, postavlja se fleg `noEmit`; ovime kompilator neće generisati niejdan fajl ali će izvršiti sve provere nad kodom o tome da li bi se kompilacija završila uspešno ili ne. Slično, fleg `emitDeclarationOnly` se koristi da bi se generisale samo `.d.ts` datoteke – izlaz neće biti kod koji je moguće pokrenuti već će biti kod kojim je moguće obećati TajpSkriptu da postoji neki kod. Ovaj način kompilacije se koristi kada se `.js` datoteke generišu od strane nekog eksternog alata kao što je Babel.

Ukoliko se u kodu koriste određene konstukcije koje nisu dostupne u EkmaSkript verziji koja je navedena u `target` opciji, TajpSkript može emitovati enke pomoćne funkcije kojim premošćava njihov nedostatak, a koje imaju isti ili dovoljno sličan efekat na kod kao konstrukcija iz novog standarda koju developer koristi u kodu. Na primer, definisanje klase koja nasleđuje neku drugu klasu koristi

ključne reči `class` i `extend`, ali ES5 ne podržava nijednu od njih. Iako je klasu lako imitirati funkcijom, nasleđivanje nije tako trivijalno, pa kompilator dodaje pomoćnu funkciju `__extends` koja prima dva argumenta: izvedenu i osnovnu klasu. Slično, ukoliko se koriste generatorske funkcije, umeće se pomoćna funkcija `__generator`, a ukoliko se koriste ključne reči `async` i `await` onda se pored `__generator` umeće i `__awaiter`.

Potencijalni problem leži u tome što se ove funkcije, ukoliko se koriste, definišu u svakom fajlu ponaosob. Nad velikim projektima ovo može da bude puno bespotrebno ponovljenog koda. Zato je uz verziju 1.5 dodat je fleg `noEmitHelpers`. Kada je ova opcija uključena, TajpSkript i dalje generiše *pozive* ovih pomoćnih funkcija, ali ne i njih same.

Međutim, sada je na developeru da definiše ove pomoćne funkcije – TajpSkript podrazumeva da su globalno definisane. Kako developer ne bi morao ručno da ih definiše, a kako ne bi bile ni generisane u svakoj datoteci, u verziji 2.1 dolazi fleg `importHelpers` sa kojim će generisani kod importovati potrebne module iz biblioteke `tslib`. Ova biblioteka je dostupna na npm repozitorijumu i dovoljno je da je developer instalira u projekat kao zavinosti da bi generisani kod radio.

Greške prilikom kompilacije TajpSkript koda se mogu svrstati u dve grupe. Jedna su greške zbog kojih ni JavaSkript ne bi bio sintaksični ispravan, i zbog njih se kompilacija uvek obustavlja neuspešno, bez emitovanja datoteka. Drugu grupu čine semantičke greške u vidu nelegalnih operacija u vezi s tipovima podataka. Iako je dodela stringa broju nemoguća, ovo je moguće kompajliarti u validan JavaSkript. Postavkom flega `noEmitOnError`, ponašanje prilikom nailaska na drugi tip greške se može promeniti, tj. moguće je zabraniti da se emituju `.js` datoteke sve dok ima bilo kakvih grešaka u `.ts` kodu. Ova opcija se obično ostavlja isključena prilikom migracije sa JavaSkript projekta na TajpSkript, ali i prilikom developmenta – nekada je korisno da se pokrene kod iako nije u potpunosti ispravan, kako bi se nešto brzo testiralo.

Lintanje koda

Iako za to postoji poseban alat

Strogi režim

2.1.6 Tipovi

U JavaSkriptu, podatak koji nije objekat i nema nijednu metodu naziva se **primitivni tip**. Ima ih šest: `string`, `number`, `boolean`, `null`, `undefined` i `symbol`. Za svaki od ovih tipova postoji i odgovarajući statički tip u TajpSkriptu, sa istim imenom.

```
const aNumber: number = 2103
```

```
const aString: string = 'hello'
const aBoolean: boolean = true
```

Kao kontrast primitivnim tipovima, svi neprimitivni tipovi pripadaju tipu `object` (od verzije 2.2).

Osim primtivnih tipova, TajpSkript nudi i neke ambijentalne tipove, u zavinosti od izabranih opcija kroz polje `lib` u konfiguracionom fajlu. Ambijentalni tipovi vezani za postojeće globalne JavaSkript objekte, što znači da postoji tipovi `Object` i `Function`. Na primer, neke od metoda definisanim nad tipom `Object` su `toString (): string` i `hasOwnProperty (v: string): boolean`. Kada se kaže da je u JavaSkritu sve objekat, misli se na to da se u svakom prototipskom lancu na vrhu nalazi `Object`, što znači da su metode kao `toString` i `hasOwnProperty` dostupne svakoj vrednosti, bilo da se radi o primitivnim ili neprimitivnim tipovima.

Prema tome, `Object` opisuje ono što je zajedničko za svaki objekat u JavaSkriptu (uključujući i primitivne vrednosti kao što su brojevi); `object` odgovara užem skupu jer isključuje pritmivne tipove.

Za preciznije definisanje objekata koristi se posebna sintaksa, koja podseća na inicijalizaciju objekata. Unutar para vitičastih zagrada navode se ključevi za koje se očekuje da postoje, a zatim se, posle dvotačke, navodi njihov tip Ova lista se odvaja zarezima.

```
const anObject: { foo: number, bar: string } = {
  foo: 2103,
  bar: 'hello',
}
```

Počev od verzije 1.4, moguće je definisati alijase za tipove. Ovime je omogućeno da se tipovi lakše koriste kroz program i da se međusobno referenciraju.

```
type MyObject = { foo: number, bar: string }
const anObject: MyObject = { foo: 2103, bar: 'hello' }
```

Properti u definicije strukture objekta se može označiti kao opcioni.

Od verzije 2.0 se ispred imena propertija može dodati modifikator `readonly`. Ukoliko on postoji, dodela vrednosti tom propertiju je moguća samo prilikom inicijalizacije.

Mada se za tipiziranje funkcija može koristiti tip `Function`, ovo je retko dovoljno jer ne pruža uvid u to koliko argumenata ima funkcija, kog su oni tipa i koja im je povratna vrednost. Za preciznije definjsanje tipa funkcije koristi se posebna sintaksa koja podseća na streličanstu funkciju u JavaSkriptu. Argumenti funkcije su oblika `arg: Tip`, razdvojeni zarezom i omeđeni oblim zagradama. Desno od ove liste se, nakon debele strelice (`=>`), daje tip povratne vrednosti.

```
type MyFn = (foo: number, bar: string) => boolean
const myFn: MyFn = (foo, bar) => foo > 21 && bar.length < 3
```

Tipovi argumenata i povratne vrednosti se mogu definisati i prilikom same definicije funkcije.

```
const myFn = (foo: number, bar: stirng): boolean => {
  return foo > 21 && bar.length < 3
}
```

U gornjem primeru bi tip `boolean` kao tip povratne vrednosti mogao i sam analizator da zaključi na osnovu poznavanja tipa kojia vraćaju operator &&, > i <, pa ga nije neophodno definisati. Prednost definisanja tipova jeste što predstavlja dokumentaciju čitljivu ljudima, pa čovek može brže ustanoviti koji je povratni tip (ne mora da analizira telo fukcije, već je dovoljno da pročita samo potpis). Eksplicitno navođenje tipa povratne vrednosti je korisno i zbog provere tipa povratne vrednosti u samo celu funkcije.

Osim streličastih funkcija, ovaj način se koristi i kod klasične definicije funkcije pomoću ključne reči `function`.

```
function myFn (foo: number, bar: string): boolean {
  return foo > 21 && bar.length < 3
}
```

Prilikom provere da li je funkcija pozvana ispravno, pored tipova argumenata posmatra se i njihov broj. JavaSkript nema ograničenje po pitanju broja argumenata sa kojima se funkcija može pozvati; ukoliko postoje argumenti viška, oni se igrnoišu, a ukoliko se neki argumenti ne dodele, za njihovu vrednost se uzima `undefined`.

U TajpSkriptu se na broj argumenata može uticati na dva načina. Proizvoljan broj argumenata zdesna se može označiti kao opcioni, dodavanjem simbola `?` posle simboličnog imena argumenta. S druge strane, broj argumenata se može povećati na beskonačnost ukoliko se ispred simboličnog imena poslednjeg argumenta doda token `...`.

Na primer, globalno dostupna funkcija `parseInt` kao prvi parametar prima string koji treba parsirati, a drugi parametar, koji je opcioni, predstavlja brojnu osnovu.

```
declare function parseInt (s: string, radix?: number): number
```

Metoda `push` definisana nad prototipom globalnog objekta `Array` prima proizvoljan broj argumenata – to su elementi koje treba dodati na kraj niza.

```
interface Array<T> {
  push (...items: T[]): number;
}
```

Unija i presek

Čest slučaj je da se nekoj promenljivoj mogu dodeliti može dodeliti vrednost koja može bit različitog tipa. Ovaj slučaj se u TajpSkriptu iskazuje pomoću **unija**.

Mogući tipovi se navode razdvojeni simbolom `|`.

```
const a: number | string = 2103
const b: number | string = 'hello'
```

Kada se pristupa propertijima sa unije, moguće je pristupiti samo onim propertijima koji postoje nad oba tipa. U opštem slučaju je ovo `Object`, ali u zavinosti od tipova koji se nalaze u uniji ih može biti više. Na primer, i nizovi i stringovi imaju svojstvo `length`.

Interfejsi i klase

Drugi način za definisanje oblika objekata su interfejsi. Definišu se ključnom rečju `interface`, nakon koje se navodi ime a potom lista parova oblika `ime: Tip`. Glavna sintaksna razlika liste jeste što se umesto zareza za razdvajanje parova mogu koristiti tačka-zarez ili novi red.

```
interface User {
  name: string
  age: number
}
```

Na sličan način se definišu i klase.

```
class User {
  name: string
  age: number
}
```

Mada se mogu koristiti na sličan način...

Osnovni tipovi

Pored šest tipova izvedenih iz primitivnih tipova koje opisuju JavaSkript objekte i jednog tipa za sve neprimitivne vrednosti, TajpSkript definše još četiri osnovna tipa: `unknown` (od 3.0), `any`, `void` i `never` (od 2.0).

Jedan od glavnih dodataka koji dolaze uz verziju 3.0 jeste **tip `unknown`**. Pomoću `unknown` se opisuju tipovi promenljivih za koje je tip nepoznat, odnosno za promenljive za koje se ne može garntovati kakva će se vrednost naći u njima. Tip `unknown` se najčešće koristi za dinamički učitane podatke, kao što su odgovori sa servera. `unknown` je vršni tip (*top type*). Vršni tip je tip kojem se može dodeliti bilo koji drugi tip, ali se on ne može dodeliti nijednom drugom tipu.

Tip `any` je univerzalni tip. Koristi se za opis tipa promenljivih za koje tip nije od važnosti. Najčešće se koristi tokom migracije ili kada bi pisanje tipa oduzelo previše vremena, a učinak bi bio previše mali. Na primer, ako za neku biblioteku ne postoje deklaracije, programer može najjednostavnije deklarisati globalnu

promenljivu kao `declare const foo: any`. Sa ovime, pokušaji pristupa `foo` neće prouzrokovati TajpSkript grešku; s druge strane, kako TajpSkript ne zna o kom je tipu reč, sve operacije nad njime će bit dozvoljene tokom kompjaliranja – TajpSkriptova statička analiza se u ovmo slučaju ne može koristiti kao garancija da neće doći do greške tokom izvršenja programa.

Svaka funkcija u JavaSkriptu, pod uslovom da se uspešno okonča njeno izvršenje (nema beskonačne petlje i ne dođe do greške pre kraja funkcije), mora da vrati neku vrednost. Povratna vrednost navodi se u istom redu posle ključne reči `return`. Ukoliko se `return` ne navede (ili se kontrolom toka zaobiđe), funkcija se završava kada se izvrši ceo blok koji predstavlja njeno telo. U tom slučaju se implicitno iz funkcije vraća vrednost `undefined`.

Međutim, TajpSkript ipak uvodi **tip `void`** koji se koristi za povratne vrednosti funkcije koje ne vraćaju ništa (izostavljanjem ključne reči `return`). Ovime se naglašava da potrošač ne treba da očekuje nikakvu povratnu vrednost od funkcije ili metode, već da se njenim pozivom očekuje neki sporedni efekat. Na ovaj način je moguće razlikovati slučaj kada je povratna vrednost funkcije eksplicitno `undefined` (na primer, kada se u nizu ne nađe traženi rezulta pozivom `Array#find`) i kada nju treba zanemariti (na primer, `Array#forEach` samo implicitno vraća `undefined`, a sporedni efekat se definiše funkcijom koja se prosleđuje kao argument).

Interfejsi

Klase

Čuvari tipova

Kako je TajpSkript jezik koji se oslanja na JavaSkript, mnoge osobine TajpSkripta su prouzrokovane obrascima i čestim šablonima koje developeri koriste dok pišu JavaScript kod.

Na primer, često se na osnovu nekog propertija utvrđuje o kom tipu objekta je reč.

```
const pointOnPlane = { x: 1, y: 2 }
const pointInSpace = { x: 9, y: 8, z: 7 }

function getHalfPoint (p) {
  if ('z' in p) return { x: p.x / 2, y: p.y / 2 }
  else return { x: p.x / 2, y: p.y / 2, z: p.z / 2 }
}
```

Međutim, utvrđivanje tipova može da bude jako kompleksno, pa u praksi postaje nemoguće utvrditi o kom je tipu reč – barem ne automatskom statičkom analizom koju TajpSkript sprovodi.

```
interface PlanePoint { x: number, y: number }
interface SpacePoint { x: number, y: number, z: number }

function getHalfPoint (p: PlanePoint | SpacePoint): PlanePoint
                                                    | SpacePoint {
  if ('z' in p) return { x: p.x / 2, y: p.y / 2, z: p.z / 2 }
  else return { x: p.x / 2, y: p.y / 2 }
}
```

U prethodnom primeru, verzija 2.6 TajpSkripta prijavljuje grešku na pretposlednjoj liniji prilikom pristupa `p.z`, sa greškom „Property ‘ z’ does not exist on type ‘ PlanePoint’”. Zaista, pošto `p` može da ima bilo koji od dva navedena tipa, u slučaju da se pristupa `p.z` nad tipom `PlanePoint`, dolazi do greške. Kompilator ne može da zaključi da je uslovom `'z' in p` zapravo napravljena razlika između tipova.

Da bi se ovaj čest način pisanja koda u JavaSkriptu podržao, TajpSkript u verziji 1.6 dodaje mogućnost da se developer definiše funkciju koja vraća `true` ili `false`, ali kao tip ima defisan **čuvar tipa** (*type guard*). Čuvari se pišu u formi `x is T`, gde je `x` deklarisan perametar u potpisu funkcije, a `T` je bilo koji tip.

```
function isSpacePoint (p: PlanePoint
                        | SpacePoint): p is SpacePoint {
  return 'z' in p
}
```

Ako se `'z' in p` iz funkcije `getHalfPoint` iz primera zameni pozivom funkcije-čuvara `isSpacePoint(p)`, TajpSkript više ne prijavljuje grešku; sada može da zaključi da je promenljiva `p` u pozitivnoj grani tipa `PlanePoint`, a u negaitvnoj grani ono što ostaje kada se iz `PlanePoint | SpacePoint` odbaci `PlanePoint`, dakle `SpacePoint`.

U određenim delovima koda, TajpSkript može implicitno da zaključi da se radi o čuvaru i da na taj načini poboljša tipove podataka na mestima u kodu gde se dešava granajne. Iz verzije u verziju je ovih tačaka u kodu sve više; najpre samo u uslovima kod `if` konstrukcije nad pozitivnom i negativnom granom i kod ternarnog operatora, zatim u `switch` naredbama, da bi se na kraju došlo do toga da se radi potpuna kontrola toka gde su uključeni i rani izlasci iz funkcije. Sem toga, vremenom se sve više načina pisanja koda prepoznaje kao čuvar.

Dva osnovna i najstarija načina za definisanje čuvara jesu korišćenje `typeof` i `instanceof` operatora.

```
function doSomething (n: string | number): number {
  if (typeof n == 'string') { /* n is of type string here */ }
  else { /* n is of type number here */ }
}
```

Tipovi *null* i *undefined* se iz tipa mogu odstraniti klasičnim poređenjem sa vrednošću `null`.

```
function inc (n: number | null) {
  return n == null ? 0 : n + 1
}
```

Počev of verzije 2.7, operator `in` se takođe može koristiti kao čuvar, pa se sa ovim dodatkom primer koda TODOREF uspešno kompajlira jer se na osnovu `z' in p` može napraviti jasna razlika između dva tipa iz unije.

Tip `never`

Uz verziju 2.0 dolazi tip `never`. U pitanju je primitvni tip koji predstavlja tip vrednosti koja se nikad neće dobiti. `never` je pod-tip svakog tipa, a nijedan tip nije pod-tip tipa `never`, osim samog tipa `never`. Ovaj tip se prirodno javlja u delovima koda koji nisu dosegljivi, bilo zbog TajpSkriptove analize tipova podataka ili zbog same prirode JavaSkripta.

Na primer, funkcije koje se nikad ne završe imaju kao povratni tip `never`. Ovo se može postići beskonačnom petljom ili bezuslovnim bacanjem izuzetka unutar tela funkcije. Sličan tip za povratnu vrednost funkcije je `void`, ali se on koristi za funkcije čije se izvršenje okonča, ali se iz nje ništa ne vrati eksplicitno (odnosno vrati se implicitni `undefined`).

Drugi čest slučaj gde se ovaj tip javlja jeste prilikom ispitivanja tipa simbola pomoću čuvara (bilo implicitnih ili eksplicitnih).

```
function doSomething (x: number | string): any {
  if (typeof x == 'string') return 1
  else if (typeof x == 'number') return 2
  else { /* x is of type never */ }
}
```

Preklapanje funkcija

U jezicima sa jakim tipovima podataka se često dozvoljava da funkcije imaju isto ime, a da se na osnovu broja i tipa argumenata određuje koju od njih treba pozvati. Međutim, kako je JavaSkript jezik sa izuzetno slabim tipovima podataka, ovaj način preklapanja funkcija nije moguć. Kod koji ispituje broj i topve argumenata mora da definiše developer.

Kako TajpSkript nije u stanju da uvek automatski zaključi o kom je tipu reč, i kako bi bilo potrebno previše menjati izvorni kod i izvoditi zaključke o tome koja je bila developerova namera (što se kosi sa ideologijom TajpSkripta), ovakav kod se i dalje mora pisati u telu funkcije, čak i u TajpSkriptu. Ipak, TajpSkript prepoznaje ovaj šablon i omogućuje da se dobro definišu tipovi ovakvih funkcija; ovo se zove **preklapanje funkcije** (*function overloading*, *function overloads*).

Na primer, iako je u telu funckije iz primera u prošlom odeljku jasna razlika između dva tipa, povratna vrednost je dvosmislena. Koji god tip da se prosledi u funkciju `getHalfPoint`, iako je on jasno statički definisan, rezultat će biti unija tipova.

```
function getHalfPoint (p: PlanePoint): PlanePoint
function getHalfPoint (p: SpacePoint): SpacePoint
function getHalfPoint (p) {
  if ('z' in p) return { x: p.x / 2, y: p.y / 2, z: p.z / 2 }
  else return { x: p.x / 2, y: p.y / 2 }
}
```

Ovako definisana funkcija je preklopljena sa dve deklaracije (prva dva reda). Deklaracija u definiciji funkcije se neće koristiti prilikom poziva funkcije, već samo prilikom provere tipova u samom telu funkcije. Na ovaj način je obezbeđeno da tip povratne vrednosti funkcije bude isti kao tip prosleđenog argumenta.

Generički tipovi

Generički tipovi omogućuju da se ista funkcija koristi za više različitih tipova, pri čemu je moguće biti eksplicitan o kome se tipu radi. U prethodnom primeru su deklarisana dva potpisa-preklopa kojima se ovo omogućava, ali je istu funkciju moguće zapisati i kraće, korišćejnjem generičkih tipova prilikom definicije funkcije.

```
function getHalfPoint<T> (p: T): T { /* ... */ }
```

U gornjem isečku koda, dodata je tipska promenljiva `T` koja omogućava da se uhvati tip koji potršač funkcije prosledi (npr. `SpacePoint`) i da se ta informacija iskoristi u ostataku funkcije (bilo u telu ili u deklaraciji). Ovde se `T` koristi kao tip povratne vrednosti.

Međutim, na ovaj način definisana funkcija će zapravo dati grešku prilikom pokušaja pristupa u negativnoj grani uslova iz tela. Naime, iako se čuvarom obezbeđuje da je tip u pozitivnoj grani `SpacePoint`, u negativnoj grani se zna jedino da *nije* u pitanju `SpacePoint`. Ovo je zato što, kako je funkcija trenutno napisana, `T` može biti bilo šta.

Da bi se ograničio skup tipova koji se može koristiti kao `T`, koristi se format `<T extends U>`, pri čemu je `T` generički tip a `U` tip kojim se određuje kojem skupu tipova mora da pripada tip `T`; drugim rečima, `U` je tip takav da je `T` pod-tip `U`.

```
function getHalfPoint<T extends SpacePoint | PlanePoint>
  (p: T): T { /* ... */ }
```

Sada ne samo da je tip unutar tela funkcije ispravno definisan, već se i od potrašača zahteva da se za tip unese isključivo `SpacePoint` ili `PlanePoint` (ili tip koji je njihov podskup).

Kada se poziva funkcija sa generičkim argumentima, posle imena se navodi tip.

```
getHalfPoint<SpacePoint>({ x: 9, y: 8, z: 7 })
```

Međutim, TajpSkipt omogućava i drugačiji način da se funkcija pozove. Ukoliko je moguće zaključiti stvarni tip generičkog tipa na osnovu stvarnih argumenata prosleđenih funckiji, onda se specificiranje generičkog tipa može izostaviti – kompajler će sam popuniti prazninu.

```
getHalfPoint({ x: 9, y: 8, z: 7 })
```

Osim toga, ako se funkciji prosledi tip `SpacePoint | PlanePoint`, povratna vrednost će takođe biti `SpacePoint | PlanePoint`. Ovo je posledica toga što takva unija zadovljava `extends` uslov naveden u deklaraciji generičkog argumenta funkcije.

Pored metoda, i interfejsi i klase mogu biti generički. Na primer, `Array` je generički tip, pa se umesto kraćeg zapisa `T[]` može koristiti pun zapis `Array<T>`.

Uslovni tipovi

Veliki pomak u mogućnostima koje pruža TajpSkript načinjen je verzijom 2.8 uz koju dolaze uslovni tipovi. Uslovni tipovi omogućavaju developeru da bude izabran jedan od dva tipa na osnovu ispunjenja uslova postavljenim kao test veze između uslova sa `extends`. Imaju oblik `T extends U ? X : Y` – kada se `T` može dodeliti `U`, tada je tip `X` a inače je tip `Y`.

Kada je tip koji se proverava „go”, odnosno nije upleten deo nekog tipa-omotača, tada se za uslovni tip kaže da je **distributivan**. Distribucija se vrši automatski nad tipovima koji su delovi unije. Na primer, za `T` se može proslediti `A | B`, pa se rezultat izraza `(A | B) extends U ? X : Y` dobija kao `(A extends U ? X : Y) | (B extends U ? X : Y)`.

Manipulacija tipovima

TajpSkript dolazi uz neke pomoćne tipove koji nemaju nikakav JavaSkript ekvivalent, odnosno ne opisuju postojeće ambijentalne simbole. Oni služe da bi developeru omogućili da transformiše tipove, odnosno da manipuliše njima, kako bi se na osnovu postojećih dobili novi. Zaključno sa verzijom 3.1 ih ima deset.

- Za označavanje da su sva svojstva tipa `T` opcioni, koristi se tip `Partial<T>`.
- Obrnuto, `Required<T>` služi da se označi da su svi propertiji obavezni.
- `Readonly<T>` označava sve propertije kao `readonly`.

Neki generički tipovi definsisani su zahvaljujući uslovnim tipovima.

- `Exclude<T, U>` iz tipa `T` odbacuje tipove koji se mogu dodeliti tipu `U`. Na primer, `Exclude<A | B | C, C | D>` vraća tipa `A | B`. Definisan je kao `T extends U ? never : T`.
- Suprotno od `Exclude`, tip `Extract<T, U>` definisan je kao `T extends U ? T : never` i služi da bi se iz `T` izdvojili samo tipove koje je moguće dodeliti tipu `U`.

Primeri tipova iz `lib`

U ovom odeljku će biti prokomentarisano nekoliko značajnih primera iz `lib.*` deklaracionih datoteka koje TajpSkript koristi za ambijentalne deklaracije.

Metoda `filter` deklarisana je nad prototipom `Array` na sledeći način.

```
interface Array<T> {
  filter<S extends T>(callbackfn: (value: T,
                                   index: number,
                                   array: ReadonlyArray<T>,
                                  ) => value is S,
                      thisArg?: any): S[]

  filter(callbackfn: (value: T,
                      index: number,
                      array: ReadonlyArray<T>,
                     ) => any,
         thisArg?: any): T[]
}
```

Metoda je preklopljena sa dva potpisa; pošto se radi o deklaracionoj `.d.ts` datoteci, nema definicije funkcije pa su oba potpisa vidljiva potrošaču. Prilikom poziva metode, preklopljeni potpisi se obilaze redom koji su navedni, tj. traži se prvo poklapanje.

Zato, ukoliko se kao `callbackfn` prosledi čuvar koji tip `T` svodi na tip `S`, dobijeni rezultat će biti niz čiji su elementi tipa `S`. Ako se pak radi o običnoj funkciji, tip povratne vrednosti neće biti promenjen u odnosu na početni tip koju ima niz nad kojim se metoda poziva.

Osim deklaracija ambijentalnih simbola koji su dostupni tokom izvršenja programa, u `lib` datotekama se nalaze i neki pomoćni tipovi koji mogu poslužiti developeru da izvede jedan tip iz drugog.

Jedan takav tip je `NonNullable`. U pitanju je generički tip koji od prosleđenog argumenta sklanja `undefined` i `null`.

```
type NonNullable<T> = T extends null | undefined ? never : T;
```

Ukoliko je stvarni tip koji se prosledi umesto `T` neki od pod-tipova tipa `null | undefined` (tj. `null | undefined`, `null`, `undefined` ili `never`), onda se dobija tip `never`. Zaista, ako se od nečega što je obavezno *nullable* ukloni ta osobina, dobija se tip koji nikad ne može da ima vrednost. Ako stvarni tip *nije* neki od pod-tipova `null | undefined`, onda se dobija taj isti tip.

Na primer, za `NonNullable<number | string | null>` se radi o distributivnom uslovnom tipu, pa se rezultujući tip dobija na sledeći način.

```
(number | string | null) extends null | undefined ? never : T
  (number extends null | undefined ? never : T) |
  (string extends null | undefined ? never : T) |
  (null   extends null | undefined ? never : T)
    number | string | never
      number | string
```

Uopštenje ovog pomoćnog tipa je `Exclude`, koji prima dva argumenta `T` i `U` i iz `T` odbacuje `U`.

```
type Exclude<T, U> = T extends U ? never : T;
```

Mogućnosti Vejna

Vejn je UI frejmvork koji služi za pisanje jednostraničnih veb-aplikacija u TajpSkriptu. Uz frejmvork dolazi CLI i bandler sa optimizatorom, pa se ne javlja potreba za instaliranjem drugih paketa i podešavanjem sistema za kreiranje produkcionog bilda.

3.1 Šabloni

S obzirom da se o frejmvorku za kreiranje *veb*-aplikacija, način komunikacije sa korisnicima odvija se pomoću DOM-a. Brauzer kreira DOM na osnovu HTML-a poslatog sa servera, ali i pruža programski pristup istom pomoću JavaSkripta. Ideja korišćenja UI frejmvorka je da se zaobiđe ovaj proces Glavni zadatak Vejna je da pogled sa kojim korisnik interaguje održava sinhronizovanim sa modelom, tj. internim stanjem aplikacije. Ovo je omogućeno definisanjem oblika interfejsa kroz *deklarativnu* sintaksu – u pitanju je sintaksa koja podseća na HTML kako bi se developer osećao „kod kuće", ali se zapravo radi o proizvoljnoj sintaksi.

Termin „deklarativna" se odnosi na to da developer ne upravlja tôkom izvršenja koda tokom pisanja šablona. Upravljanje tokom je redosled u kome računar izvršava izraze u skripti [1]. Umesto ovoga, developer samo *deklariše* **šta** treba da bude prikazano na ekranu korisnika na osnovu stanja modela, a ne i **kako** se ovaj proces odvija. Ovo ne samo da olakšava pisanje šablona, već i omogućuje da se vrše interne promene u frejmvorku a da developer toga ne bude svestan – drugim rečima, developer može prostim ažuriranjem verzije frejmvorka koju koristi da unapredi performanse aplikacije.

Šablonska sintaksa je nadskup HTML-a[1] – to znači da svaki validan HTML predstavlja validan Vejn šablon. Ovo za posledicu ima mogućnost da se postojeći

[1] Šabloni predstavljaju isečke korisničkog interfejsa, tj. ne radi se o čitavom HTML dokumentu. Pod „podskupom HTML-a" se misli na podskup sintakse koja je dozvoljena tamo gde se očekuje tzv. *flow content*; v. sekciju 3.2.4.2.2. specifikacije HTML 5.2. U praksi ovi detalji ne predstavljaju prepreku u razumevanju sintakse šablona, pa se koristi termin „podskup HTML-a".

statički sajt vrlo brzo može integrisati u aplikaciju koja se piše u Vejnu, kao i da šablone mogu pisati ljudi koji nemaju iskustva sa pisanjem programa (npr. dizajneri, urednici).

3.1.1 Vezivna sintaksa

Proširenje sintakse HTML-a se ugleda u mogućnosti da se trenutno stanje modela sinhroniše sa pogledom, tj. da se korisnički interfejs osveži svaki put kada dođe do promene modela, u skladu sa pravilima definisanim u šablonima. Ovo je omogućeno vezivnom sintaksom.

Vezivnom sintaksom se mogu definisati izrazi i iskazi. Izrazi su literali i reference na svojstva iz klase, a iskazi su pozivi metoda iz klase.

Literal se definiše pukim navođenjem. Među literale spadaju stringovi (`"foo"`, `'foo'`), brojevi (`21`, `0x3`), logičke konstante (`true` i `false`), `null` i `undefined`. Mogu se referencirati **svojstva** sa klase za koju je šablon vezan ili alijasi koje izlaže direktiva u određenom opsegu, kao i **lanci pristupa** (`item`, `item.name`). Ispred svojstva i lanca pristupa se može dodati znak uzvika (`!`), što znači da će nad podatkom biti primenjen **operator negacije**.

Od iskaza se mogu koristiti jedino **pozivi**. Argumenti se navode u malim zagradama, odvojeni zarezom – kao i u Javaskriptu. Pored izraza, kao argument se može navesti i specijalan simbol # koji označava preuzimanje reference koja se prosleđuje kroz događaj (iz HTML elementa) ili izlaz (iz komponente).

Proizvoljni Javaskript izrazi nisu dozovljeni u vezivnoj sintaksi – operator negacije predstavlja jedini vid manipulacije koji je moguć nad podacima u okviru definicije šablona. Na primer, za šablon se ne može vezati izraz kao `user && user.name` ili `name.length > 0`. Ukoliko developer želi da u šablonu veže kompleksni izraz koji se računa na osnovu postojećih podataka iz modela, treba da koristi getere na klasi.

3.1.2 Interpolacija

Osnovni vid vezivanja modela jeste ispisivanje stringa direktno na korisnički interfejs. Ovaj vid komunikacije modela i pogleda naziva se interpolacija. Vezivna sintaksa se piše između para dvostrukih vitičastih zagrada.

Na primer, na ekran se može ispisati korisničko ime prijavljenog korisnika.

```
Welcome, {{ username }}!
```

3.1.3 HTML elementi

Kao što je već rečeno, Vejn šabloni su nadskup HTML-a, pa se HTML elementi u šablonima mogu koristiti na standardan način. Ovo uključuje definiciju atributa, prazne elemente (*void elements*) i izostavljanje tagova.

```
<form>
  <label for="name">Name</label>
  <input type="text" id="name" name="name">
</form>
```

Slično parserima HTML-a kod modernih brauzera, parser šablona u Vejnu je otporan na greške – na primer, nezatvoreni tagovi (koji moraju biti zatvoreni po specifikaciji) će se automatski zatvoriti, ali nije definisano tačno u kom trenutku (nailazak na sledeći element, kraj šablona, itd), pa se na ovo ponašanje ni u Vejn šablonima ne treba oslanjati.

Navođenjem atributa u HTML-u se definiše koja će svojstva i koje atribute imati odgovarajući čvor u DOM-u. Svojstva i atributi se koriste da finije odrede semantiku i ponašanje elemenata. Na primer, svojstvo `type` na `<input>` elementu određuje kakav se unos očekuje od korisnika.

Svojstva se mogu sinhronizovati sa modelom tako što se ime svojstva navede u uglastim zagradama. U tom slučaju se vrednost koja se navodi sa desne strane imena tumači kao izraz u vezivnoj sintaksi.

```
<p [id]="myParagraph.uniqueId">...</p>
```

Mapiranje između atributa HTML-a i svojstava DOM čvorova nije uvek jedan-na-jedan. Na primer, mada je `<label for="anId»` validna definicija atributa `for`, ova informacija se sa čvora čita (i postavlja) preko svojstva `htmlFor`. Kako bi se napravila eksplicitna razlika između svojstava i atributa čvorova, za definisanje atributa koristi se prefiks `attr`. Ovo je slično razlici između korišćenja direktnog pristupa svojstvu (`element.svojstvo`) i korišćenju metode `setAttribute` nad čvorom.

```
<label [htmlFor]="anId">...</label>
labelEl.htmlFor = anId

<label [attr.for]="anId">...</label>
labelEl.setAttribute('for', anId)
```

U slučaju da ime atributa u šablonu sadrži crticu, ono se automatski tumači kao atribut čvora (a ne njegovo svojstvo) jer više ne postoji dvosmislenost – ime svojstava HTML elemenata su izabrana tako da ne sadrže crtice.

```
<input [aria-label]="aLabel">
```

Interpolacija, atributi i svojstva se koriste sa izrazima vezivne sintakse. Koriste se za vezivanje *podataka* za poglede i očuvanje sinhronizacije. Da bi se delalo na

korisnikovu interakciju sa interfejsom, potrebno je pretplatiti se na događaje koje emituju HTML elementi. Za ovo se koriste iskazi vezivne sintakse; dakle, definišu se pozivi metoda Drugima rečima, za šablon se vezuje *ponašanje* aplikacije.

Za vezivanje metoda koje treba da budu pozvane na određeni događaj, koristi se sličan zapis kao definisanje svojstava i atributa. Naime, ime događaja se omeđuje parom oblih zagrada, a s desne strane se navodi iskaz kojim se poziva metoda.

```
<button (click)="inc()">Increment</button>
```

Za argumente funkcije se, osim iskaza, može navesti i poseban simbol # – *placeholder*. Njime se može pristupiti objektu koji emituje događaj kako bi se informacije koje on nosi sa sobom mogle iskoristiti u telu metode.

3.2 Komponente

Kao i kod svih UI frejmvorka, glavna gradivna jedinica u Vejnu je *komponenta*. Komponente predstavljaju izolovane logičke celine koje se mogu smestiti u korisnički interfejs.

Komponentu treba shvatiti kao prirodnu nadogradnju HTML postojećih HTML elemenata. Na primer, HTML element `HTMLInputElement` ima svoje **ime** na osnovu kojeg se prepoznaje u HTML kodu (string `input`), ima neka **svojstva** na osnovu kojih može da se kontrolišu finese njegovog ponašanja i izgleda koja se mogu specificirati navođenjem HTML atributa ili direktnom manipulacijom kad odgovarajućim propertijima JavaSkript objekta (npr. `type`), i emituje neke **događaje** uz informaciju o tim događajima na koje se potrošači mogu pretplati i reagovati na njih.

Analogno tome, komponente dobijaju **ime** registrovanjem za druge komponente, **ulazi** (*inputs*) koji su sačinjeni od skupa svojstava klase koje je autor komponente proglasio kao delom njenog javnog API-ja, dok su **izlazi** događaji na koje se druge komponente mogu pretplatiti da bi se obavljala komunikacija između njih.

3.2.1 Deklaracija

Komponente se u Vejnu definišu klasom nad kojom je primenjen dekorator `@Template`, uvezen iz ulazne tačke modula `wane`. Prvi i jedini argument dekoratora je string literal kojim se definiše struktura korisničkog interfejsa komponente.

Za stanje komponente koristi se telo klase, gde se standardno mogu definisati njena svojstva i metode. Sva svojstva i metode su dostupne za referenciranje u šablonu.

3.2.2 Pristupna komponenta

Kao što se HTML-om definiše stablo elemenata, u Vejnu se od komponenti pravi *stablo komponenti*. U korenu takvog stabla leži posebna komponetna koja se naziva **pristupna komponenta** (*entry component*).

Pristupnu komponetu treba smestiti u datoteku `index.ts` u folderu `src` i načiniti je podrazumevanim izvozom iz modula – ovako će je Vejn prepoznati kao ulaznu tačku u aplikaciju. Kada se aplikacija pokrene, renderuje se ova komponenta, a ona za sobom povlači sve druge komponente kroz svoj šablon.

3.2.3 Korišćenje

Za razliku od HTML elemenata, koji su uvek globalno dostupni, komponente je potrebno **registrovati** za komponentu kako bi moglo da joj se pristupi iz šablona. Registracija se vrši navođenjem reference na odgovarajuću klasu među argumentima dekoratora `@Register` koji se navodi uz klasu. Ime simbola pod kojim se klasa registruje mora ima veliko početno slovo (tzv. *PascalCase*) kako bi se obezbedilo da neće doći do preklapanja sa imenima postojećih HTML elemenata i potencijalnih veb komponenti koje su registrovane na istoj stranici.

Osim razlike u malim i velikim slovima, postoji još jedna bitna razlika kod korišćenja komponenti u šablonima: simbolička imena za istu klasu (pa samim tim istu komponentu) mogu da se razlikuje među upotrebama u različitim komponenatama. Drugim rečima, ime koje developer da simbolu prilikom registracije je ono pomoću koga komponenta može da se deklariše u šablonu. Na primer, klasa se može preimenovati alijasovanim importom; pored toga, klasa uopšte ne mora da ima ime ukoliko je iz modula izvezena kao podrazumevani simbol, već joj se simboličko ime dodeljuje tek prilikom uvoza.

Registrovana komponenta se u šablon može postaviti kao bilo koji drugi HTML element. Pošto ne može da ima decu, može se koristiti i samo-zatvarajući tag.

```
<Component></Component>
<Component/>
```

3.2.4 Ulazi

Da bi komponentu bilo moguće parametrizovati, uvodi se pojam **ulaza** (*input*) u komponentu. Ovaj pojam je pandam svojstvima koje imaju HTML elementi.

Ulazi u komponentu se definišu deklarisanjem javnog svojstva na klasi.

Pošto se za proveru ispravnosti tipova u izvornom kodu koristi strogi režim TajpSkripta, ulazima za koje se očekuje da uvek dobiju vrednost iz šablona u kome se koriste dodaje se `!` – stoga, ovo označava **obavezni ulaz**, pa Vejn prijavljuje grešku ukoliko mu se vrednost ne prosledi. Za bi se definisala podrazumevana

vrednost ulaza, dovoljno je da se svojstvo jednostavno inicijalizuje. Ulazi sa podrazumevanom vrednošću ne mogu biti obavezni.

```
@Template(‘...‘)
class ExampleComponent {
  public input1!: number
  public input2 = 2
  input3: any
  private prop1: any
}
```

U prethodnom primeru definisani su, redom: eksplicitno definisan obavezni ulaz; eksplicitno definisan opcioni ulaz sa podrazumevanom vrednošću `2`; implicitno definisan ulaz; obično svojstvo na klasi koje se može iskoristiti za čuvanje stanja komponente.

U šablonima, ulazi se koriste kao da je u pitanju bilo koji HTML atribut, s tim što se ime ulaza omeđuje parom uglastih zagrada, a umesto konkretne vrednosti se očekuje **vezivna sintaksa**.

Osim vezivanja za svojstva klase, veza se može ostvariti i sa alijasima, ukoliko se u šablonu koriste direktive koje ih stvaraju. Više reči o tome u odeljku o `w:for` direktivi.

3.2.5 Izlazi

HTML elementi imaju događaje, a Vejn komponente imaju **izlaze** (*output*). Izlaz se deklariše kao metoda bez tela u telu klase. Da bi se izlaz komponente osluškivao, dodaje se HTML atribut, pri čemu je ime izlaza omeđeno parom oblih zagrada, a umesto prosleđivanja konkretne vrednosti se očekuje **poziv funkcije**, tj. metode definisane u klasi.

Da bi se pribavila referenca na vrednosti koje se emituju kroz izlaz, u šablonu se koristi specijalni znak # kao argument funkcije. Pored njega, argumenti funkcije se mogu referencirati na svojstva klase ili mogu biti literali.

3.3 Direktive

Direktive su posebne naredbe koje se umeću u šablone u vidu HTML tagova. Primaju proizvoljan broj čvorova za decu i služe za parametrizaciju načina na koji će se taj sadržaj prikazati u rezultujućem DOM-u.

Sintaksno, direktive se od HTML elemenata i Vejn komponenti razlikuju po prefiksu `w:`. Zbog svoje prirode, parametri se – umesto navođenjem atributa u vidu parov ključ-vrednost – definišu potpuno proizvoljnom sintaksom koja se piše u nastavku imena, a u sklopu otvarajućeg taga.

3.3.1 Uslovi

Prisustvo ili odsustvo dela šablona može se kontrolisati korišćenjem direktive `w:if`. Uslov na osnovu koga se određuje da li će se grupa elemenata naći u DOM-u ili ne se navodi u vidu izraza vezivne sintakse.

```
<w:if isLoggedIn>
  Welcome, {{ user.name }}!
  <img [src]="user.image" [alt]="user.name"/>
</w:if>
```

U prethodnom primeru se, u zavinosti od trenutne vrednosti dodeljene svojsvu klase `isLoggedIn`, isečak šablona unutar direktive `w:if` ili prikazuje ili ne.

3.3.2 Nizovi

Predstavnik struktura podataka za rad sa kolekcijama istorodnih elemenata u JavaSkriptu su nizovi, implementirani pomoću prototipa `Array`. U šablonima se može vršiti iteracija nad nizovima korišćenjem direktive `w:for`.

U nastavku se navodi sintaksa oblika `(item, index) of items; key: id`, gde je

- `items` – svojstvo ili lanac pristupa sa klase,
- `item` – simboličko ime za jedan element iz niza `items`,
- `index` – simboličko ime za indeks elementa `item` iz niza `items`,
- `id` – lanac pristupa kojim se definiše način na koji se može preuzeti „ključ” po kome se semantički razlikuju elementi niza,

dok su `of`, `;` i `key:` ključne reči kojima se poboljšava čitljivost sintakse.

Simbočko ime za indeks elementa nije obavezno navesti; u slučaju da se ono ne navede, zagrade koje omeđuju simboličko ime za vrednost elementa niza nisu obavezne.

Deo šablona koji je naveden unutar direktive čini šablon koji će se koristiti za kreiranje DOM-a. Ovaj deo šablona parametrizovan je vrednošću elementa koji je trenutno posećen tokom iteracije (`item`) i, opciono, njegovom brojnom indeksu (`index`). U direktivi, ova imena postaju deo opsega koji se može koristiti za vezivnu sintaksu u šablonima (pored svojstava i metoda klase).

```
<ol>
  <w:for user of users; key: id>
    <li>{{ user.name }} ({{ user.score }})</li>
  </w:for>
</ol>
```

3.3.3 Skraćeni oblik

Vrlo je čest slučaj da se unutar direktive nađe samo jedan HTML element ili samo jedna komponenta. U tim slučajevima, način zapisa direktive se može skratiti tako što se direktiva navodi kao HTML atribut elementa ili komponente, pri čemu se za ključ koristi ime direktive (sa prefiksom `w:`), a kao vrednost se koristi sintaksa koju ta direktiva razume.

```
<span [w:if]="!isLoggedIn">Welcome, guest!</span>

<ol>
  <li [w:for]="user of users; key: id">
    {{ user.name }} ({{ user.score }})
  </li>
</ol>
```

Dve direktive se ne mogu istovremeno naći na jednom elementu ili jednoj komponenti. Ukoliko se javi potreba za ovakvom strukturom, barem jedna od direktiva se mora zapisati u regularnoj (dužoj) sintaksi.

3.3.4

3.4 Pokretanje

wane binexec iz nodemodules

prod build, dist folder

config fajl

dev server

Bibliography

[1] *Control flow – MDN Web Docs Glossary: Definitions of Web-related terms | MDN*. URL: https://developer.mozilla.org/en-US/docs/Glossary/Control_flow (visited on 08/24/2018).