POGLAVLJE 1

Uvod u programiranje

Dobro došli u knjigu *Naučite JavaScript*.

Deo *Početni koraci* predstavlja uvod u više osnovnih koncepata programiranja – razume se, s posebnim osvrtom na JavaScript (koji ćemo često pisati skraćeno, JS) – i opisuje kako treba razumeti i pristupiti ostalim delovima knjige. Naročito ako tek počinjete da učite programiranje i/ili JavaScript, ovaj deo ukratko opisuje šta treba da znate kako biste napravili svoje *prve korake*.

Knjiga počinje objašnjenjima osnovnih principa programiranja na vrlo visokom nivou. Prvi deo knjige namenjen je onima koji započinju čitanje knjige *Naučite JavaScript* s malo ili nimalo prethodnog iskustva u programiranju, a knjigu su izabrali kao pomoć na putu ka razumevanju programiranja kroz objektiv JavaScripta.

Poglavlje 1 treba shvatiti kao kratak pregled stvari koje ćete morati više da učite i vežbate kako biste zaista *počeli da programirate*. Postoji i mnogo drugih odličnih izvora uvodnih informacija o programiranju koji vam omogućavaju da dalje obrađujete te teme. Preporučujem da učite i iz njih, a ne samo iz ovog poglavlja.

Kada procenite da ste dovoljno dobro savladali opšte osnove programiranja, poglavlje 2 će vam pomoći da savladate stil programiranja kakav je uobičajen u JavaScriptu. Poglavlje 2 vas upoznaje s JavaScriptom, ali ni ono nije sveobuhvatni vodič – čekaju vas i preostali delovi knjige *Naučite JavaScript!*

Ako već prilično dobro poznajete JavaScript, prvo pogledajte poglavlje 3 da biste stekli opštu sliku onoga što možete očekivati od knjige *Naučite JavaScript*, a zatim počnite da je čitate!

Programski kôd

Počnimo od samog početka.

Program, koji se često naziva *izvorni kôd* (engl. *source code*) ili samo *kôd* (engl. *code*) sastoji se od skupa specijalnih naredbi koje nalažu računaru koje poslove treba da obavi. Kôd se najčešće upisuje u tekstualnu datoteku, mada JavaScript omogućava i da kôd kucate direktno u prozor razvojne konzole čitača veba, što ćemo opisati uskoro.

Skup pravila za ispravne formate i kombinacije programskih naredbi zove se *računarski (programski) jezik*, a ponekad i *sintaksa jezika* – slično govornom jeziku čija pravila opisuju kako se reči pravilno pišu i kako se sastavljaju ispravne rečenice pomoću reči i znakova interpunkcije.

Programski iskazi

U programskom jeziku, svaka grupa reči, brojeva i operatora koja obavlja konkretan posao zove se *iskaz* (engl. *statement*). U JavaScriptu, iskaz može izgledati ovako:

```
a = b * 2;
```

Znaci `a` i `b` zovu se *promenljive* (engl. *variables*) (videti odeljak „Promenljive“ na strani 15), koje su nalik običnim kutijama za čuvanje raznih stvari. U programima, promenljive sadrže vrednosti (kao što je broj `42`) s kojima program radi. Zamislite ih kao simbolična mesta rezervisana za čuvanje samih vrednosti.

Nasuprot tome, `2` je samo po sebi vrednost, koja se zove *literal* ili *literalna vrednost* (engl. *literal value*) zato što se pojavljuje samostalno i nije smeštena u promenljivu.

Znaci `=` i `*` su *operatori* (videti odeljak „Operatori“ na strani 10) – oni izvršavaju akcije s vrednostima i promenljivama, kao što je dodeljivanje vrednosti i matematička operacija množenje.

Većina iskaza se u JavaScriptu završava znakom tačka i zarez (`;`).

Iskaz `a = b * 2;` grubo opisano, nalaže računaru da učita tekuću vrednost koja je smeštena u promenljivoj `b`, da zatim tu vrednost pomnoži sa `2`, a onda da dobijeni rezultat uskladišti u drugu promenljivu koju ćemo nazvati `a`.

Programi su samo kolekcije mnogih takvih iskaza, koji svi zajedno opisuju sve korake koje treba izvršiti da bi program obavio ono za šta je namenjen.

Izrazi

Iskazi se sastoje od jednog ili više *izraza* (engl. *expressions*). Izraz čini svaka referenca na promenljivu ili vrednost, ili na skup jedne ili više promenljivih ili literala kombinovanih pomoću operatora.

Na primer, iskaz:

```
a = b * 2;
```

sastoji se od četiri izraza:

- `2` je *izraz s literalnom vrednošću*.
- `b` je *izraz s promenljivom*, što znači da učitava njenu tekuću vrednost.
- `b * 2` je primer *aritmetičkog izraza*, što znači da obavlja operaciju množenja.
- `a = b * 2` je *izraz za dodeljivanje vrednosti*, što znači da rezultat izraza `b * 2` dodeljuje promenljivoj `a` (više reči o izrazima biće u nastavku teksta).

Opšti izraz koji može da se navede samostalno zove se *iskaz izraza*, kao što je sledeći:

```
b * 2;
```

Ta vrsta iskaza nije mnogo uobičajena ni korisna, jer najčešće ne bi uopšte uticala na izvršavanje programa – izraz bi učitao vrednost `b` i pomnožio je sa `2`, ali zatim ne bi ništa učinio s dobijenim rezultatom.

Uobičajeniji iskaz izraza jeste iskaz *pozivnog izraza* (videti odeljak „Funkcije“ na strani 22) jer se ceo iskaz sastoji od izraza pozivanja funkcije:

```
alert( a );
```

Izvršavanje programa

Kako te kolekcije programskih iskaza nalažu programu šta treba da uradi? Programu je potrebno *izvršavanje*, što se zove i *pokretanje* programa.

Iskazi kao što je `a = b * 2` korisni su za programere dok ih čitaju i pišu, ali zapravo nisu u obliku koji računar može direktno da razume. Iz tog razloga, na računaru se koristi specijalna pomoćna alatka (to može biti *interpreter* ili *kompajler*), koja prevodi kôd koji napišete u komande koje računar razume.

Za neke računarske jezike, uobičajeno je da se komande prevode odozgo nadole, red po red, kad god se program izvršava, što se najčešće zove *interpretiranje* koda.

Za druge jezike, prevođenje se obavalja pre pokretanja programa, što se zove *kompajliranje* (ili *prevođenje*) koda; znači, kada se program zatim *pokrene*, izvršavaju se već prevedene i spremne naredbe.

Uobičajena tvrdnja je da je JavaScript *interpretiran* jezik zato što se izvorni JavaScript kôd obrađuje pri svakom izvršavanju. Ali, to nije sasvim tačno. Mašina (engl. *engine*) jezika JavaScript zapravo prvo *kompajlira* a odmah zatim izvršava kompajliran kôd.



Više informacija o kompajliranju JavaScript koda naći ćete u prva dva poglavlja drugog dela knjige – *Opseg vidljivosti i ograde*.

Pokušajte i sami

U ovom poglavlju, svaki programski koncept uvodi se pomoću jednostavnih primera koda, koji su svi napisani na JavaScriptu (razume se!).

Sledeće ne možemo istaći dovoljno jako: dok napredujete kroz ovo poglavlje – a možda će trebati da odvojite vreme kako biste ga prešli i više puta – trebalo bi da uvežbavate sva tri opisana koncepta tako što ćete svojeručno pisati kôd. To ćete najjednostavnije uraditi ako otvorite konzolu s razvojnim alatkama u čitaču veba koji najradije koristite (Firefox, Chrome, IE itd.).



Razvojnu konzolu obično otvarate pomoću odgovarajuće prečice s tastature ili stavke menija. Više informacija o otvaranju i upotrebi razvojne konzole u vašem čitaču veba naći ćete u članku „Upotreba razvojne konzole“ na adresi *http://blog.teamtreehouse.com/mastering-developer-tools-console*.

Da biste u konzolu upisali više redova istovremeno, pritisnite `<shift> + <enter>` kako biste prešli u sledeći prazan red. Čim pritisnete samo `<enter>`, konzola će izvršiti sve što ste upravo otkucali.

Razmotrimo sada postupak izvršavanja koda unutar konzole. Prvo, predlažem da otvorite novu praznu karticu u svom čitaču veba. To najradije radim tako što u polje za adresu upišem `about:blank`. Zatim, otvorite svoju razvojnu konzolu na način koji smo upravo opisali.

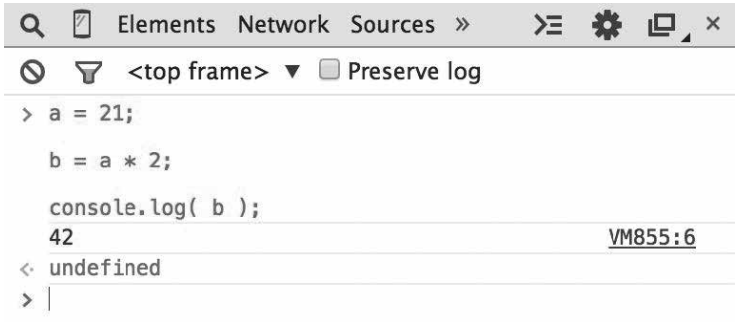
Upišite sledeći kôd i pogledajte kako radi:

```
a = 21;

b = a * 2;

console.log( b );
```

Ako prethodni kôd upišete u konzolu čitača Chrome, trebalo bi da dobijete nešto slično sledećem:



Slobodno pokušajte sami. Najbolji način da naučite programiranje jeste da počnete da pišete kôd!

Rezultati izvršavanja programa

U prethodnom primeru koda, upotrebili smo naredbu `console.log(..)`. Pogledajmo ukratko suštinu tog reda koda.

Možda ste već i sami pogodili, ali to je tačno način na koji ispisujemo tekst (što se zove i *rezultat izvršavanja programa* namenjen korisniku) na razvojnoj konzoli. Taj iskaz ima dve karakteristike koje bi trebalo objasniti.

Prvo, deo `log( b )` predstavlja pozivanje određene funkcije (videti odeljak „Funkcije“ na strani 22). Ono što se tu događa jeste da promenljivu `b` prosleđujemo navedenoj funkciji, čime od nje zahtevamo da preuzme vrednost `b` i ispiše je na konzoli.

Drugo, deo `console.` je referenca na određeni objekat u kojem se nalazi funkcija `log(..)`. Objekte i njihova svojstva detaljnije razmatramo u poglavlju 2.

Drugi način da dobijete rezultate koje možete videti na ekranu jeste da izvršite iskaz `alert(..)`. Na primer:

```
alert( b );
```

Ako ga izvršite, zapazićete da umesto da rezultate ispisuje na konzoli, taj iskaz otvara iskačući „OK“ prozor sa sadržajem promenljive `b`. Međutim, kodiranje i izvršavanje programa na konzoli lakše ćete savladati ako budete koristili iskaz `console.log(..)` a ne iskaz `alert(..)`, zato što tako možete da prikazujete više vrednosti istovremeno bez remećenja interfejsa čitača veba.

U ovoj knjizi prikazivaćemo rezultate programa pomoću iskaza `console.log(..)`.

Unošenje ulaznih podataka

Dok razmatramo rezultate programa, možda se pitate i šta je sa *ulaznim podacima* za programe (na primer, prihvatanjem podataka od korisnika).

Najuobičajeniji način prijema ulaznih podataka jeste da HTML stranica prikaže određene elemente koji se koriste na obrascima (kao što su polja za unošenje teksta) u koje korisnik može nešto da upiše, a zatim se pomoću JS te vrednosti učitavaju u promenljive programa.

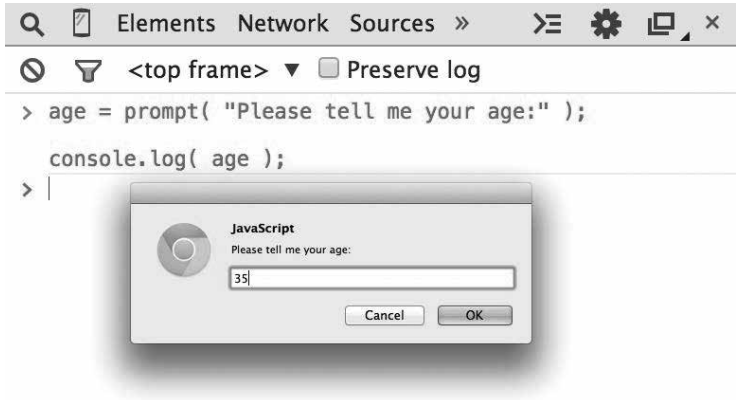
Postoji i jednostavniji način da učitate ulazne podatke kada im je jedina svrha učenje i objašnjavanje nečega, kao u ovoj knjizi. Iskoristite funkciju `prompt(..)`:

```
age = prompt( "Please tell me your age:" );

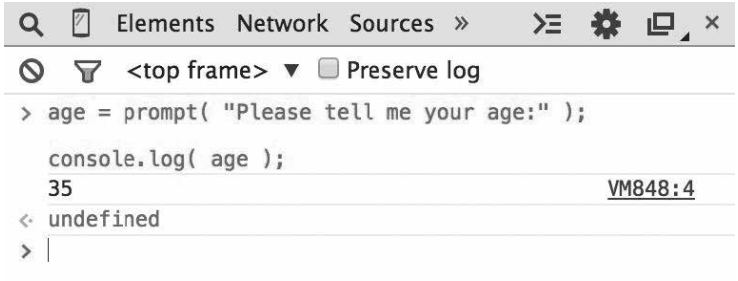
console.log( age );
```

Kao što ste možda i sami pretpostavili, sadržaj poruke koju prosledite funkciji `prompt(..)` – u ovom slučaju, `"Please tell me your age:"` (molimo vas da navedete svoje godine) – ispisuje se u iskačućem prozoru.

Trebalo bi da to bude nešto slično ovome:



Pošto pritiskanjem dugmeta „OK“ prosledite programu ulazni tekst, videćete da se vrednost koju ste otkucali smešta u promenljivu `age`, čiji sadržaj zatim *ispisujemo* pomoću funkcije `console.log(..)`:



Da bismo se bavili samo jednostavnim stvarima dok učimo osnovne koncepte programiranja, primeri u ovom odeljku neće zahtevati ulazne podatke. Ali pošto ste sada videli kako se koristi funkcija `prompt(..)`, ako želite da sebi postavite izazov, možete pokušati da koristite i ulazne podatke dok proučavate primere.

Operatori

Operatori određuju akcije koje izvršavamo s promenljivama i vrednostima. Već smo upoznali dva JavaScript operatora, `=` i `*`.

Operator `*` izvršava matematičko množenje. Sasvim jednostavno, zar ne?

Operator jednakosti `=` koristi se za *dodeljivanje vrednosti* – prvo se izračunava vrednost na *desnoj strani* (izvorna vrednost) operatora `=` a zatim se ona smešta u promenljivu koju zadamo na *levoj strani* (ciljna promenljiva).



Ovo možda liči na čudan obrnut redosled dodeljivanja vrednosti. Umesto `a = 42`, neko bi možda radije obrnuo redosled tako da izvorna vrednost bude na levoj strani a ciljna promenljiva na desnoj, kao u izrazu `42 -> a` (ovo nije ispravno u JavaScriptu!). Nažalost, redosled `a = 42`, kao i slične varijacije, preovlađuje u savremenim programskim jezicima. Ako vam izgleda neprirodan, odvojte malo vremena i vežbajte s njime u mislima da biste se navikli.

Razmotrite sledeće:

```
a = 2;
b = a + 1;
```

U ovom primeru dodeljujemo vrednost `2` promenljivoj `a`. Zatim, učitavamo vrednost promenljive `a` (koja je i dalje `2`), dodajemo joj `1` da bismo dobili rezultujuću vrednost `3`, koju zatim smeštamo u promenljivu `b`.

Mada tehnički gledano to nije operator, rezervisana reč `var` će vam trebati u svakom programu jer je to primarni način na koji *deklarišete* (kaže se i *definišete*) promenljive (videti odeljak „Promenljive“ na strani 15).

Trebalo bi da uvek definišete promenljivu po imenu pre nego što je upotrebite. Ali određenu promenljivu treba da definišete samo jedanput u svakom *opsegu vidljivosti* (videti odeljak „Opseg vidljivosti promenljivih“ na strani 23); posle toga je možete koristiti više puta, prema potrebi. Na primer:

```
var a = 20;

a = a + 1;
a = a * 2;
console.log( a )   // 42
```

Sledi nekoliko najuobičajenijih operatora u JavaScriptu:

Dodeljivanje vrednosti
: `=`, kao u `a = 2`.

Matematički
: `+` (sabiranje), `-` (oduzimanje), `*` (množenje) i `/` (deljenje), kao u `a * 3`.

Kombinovani s dodeljivanjem vrednosti
: `+=`, `-=`, `*=` i `/=` složeni su operatori koji kombinuju određenu matematičku operaciju s dodeljivanjem vrednosti, kao u `a += 2` (isto što i `a = a + 2`).

Inkrementiranje/dekrementiranje
: `++` (inkrementiranje), `--` (dekrementiranje), kao u `a++` (isto što i `a = a + 1`).

Pristupanje svojstvu objekta
: `.` kao u `console.log()`.

  Objekti su vrednosti koje sadrže druge vrednosti na određenim imenovanim mestima koja se zovu svojstva (engl. *properties*). `obj.a` predstavlja vrednost objektnog tipa koja se zove `obj` i ima svojstvo čije je ime `a`. Drugi oblik pristupanja svojstvima je `obj["a"]`. Videti poglavlje 2.

Jednakost
: `==` (labava jednakost), `===` (striktna jednakost), `!=` (labava različitost), `!==` (striktna različitost), kao u `a == b`.

  Videti odeljak „Vrednosti i tipovi“ na strani 12 i poglavlje 2.

Poređenje
: `<` (manje od), `>` (veće od), `<=` (manje od ili jednako), `>=` (veće od ili jednako), kao u `a <= b`.

  Videti odeljak „Vrednosti i tipovi“ na strani 12 i poglavlje 2.

Logički
: `&&` (i, konjukcija), `||` (ili, disjunkcija), kao u `a || b` koji bira `a` *ili* `b`.

  Ovi operatori se koriste za izražavanje kombinovanih uslova (videti odeljak „Uslovni iskazi“ na strani 18), kao `a` *ili* `b` je istinito.



Znatno detaljnija objašnjenja i opise operatora koji ovde nisu pomenuti naći ćete na veb stranici Mozilla Developer Network (MDN), u odeljku „Expressions and Operators“.

Vrednosti i tipovi

Ako pitate nekog zaposlenog u prodavnici mobilnih telefona koliko košta određeni uređaj, a on vam kaže „devedeset i devet, devedeset i devet“ (na primer, 99,99 dolara), on vam zapravo saopštava novčani iznos koji treba da platite (plus dažbine) da biste ga kupili. Ukoliko želite da kupute dva takva telefona, lako možete primeniti mentalnu matematiku da biste udvostručili tu vrednost i dobili 199,98 kao osnovnu cenu.

Ukoliko taj isti prodavac odabere neki sličan telefon i kaže da je „besplatan“ (i možda pri tome oponaša prstima znakove navoda), on vam ne saopštava nikakav iznos, nego samo drugačiji oblik predstavljanja očekivane cene ($0,00) – a to je reč „besplatno“.

Kada zatim pitate da li uz telefon dobijate i punjač, odgovor može biti samo „da“ ili „ne“.

Na vrlo slične načine, kada u programu izražavate određene vrednosti, birate različite oblike predstavljanja tih vrednosti u zavisnosti od toga šta nameravate da radite s njima.

Ti različiti oblici predstavljanja vrednosti zovu se *tipovi* (engl. *types*) u terminologiji programiranja. JavaScript ima ugrađene tipove za svaku od tzv. *primitivnih* (osnovnih) vrednosti:

- Kada treba da obavite neku matematičku operaciju, koristite tip `number`.
- Kada treba da ispišete neku vrednost na ekranu, koristite tip `string` (jedan ili više znakova, reči ili rečenica).
- Kada u programu treba da donesete određenu odluku, koristite tip `boolean` (`true` ili `false`).

Vrednosti koje se umeću direktno u izvorni kôd zovu se *literali*. Literali tipa `string` pišu se između navodnika (`"..."`) ili polunavodnika (`'...'`) – jedina razlika je da li više volite jedan ili drugi stil. Literali tipa `number` i `boolean` samo se navode takvi kakvi su (na primer, `42`, `true` itd.).

Razmotrite sledeće:

```
"I am a string";
'I am also a string';

42;

true;
false;
```

Osim vrednosti tipa `string`/`number`/`boolean`, uobičajeno je da programski jezici stavljaju na raspolaganje i *nizove*, *objekte*, *funkcije* i drugo. O vrednostima i tipovima biće znatno više reči dalje u ovom poglavlju, kao i u sledećem.

Konverzije tipova

Ako imate vrednost tipa `number` (broj) ali treba da je prikažete na ekranu, morate je prvo pretvoriti u tip `string`. U JavaScriptu se to pretvaranje zove *konverzija tipa* (engl. *coercion*). Slično tome, ako neko upiše grupu numeričkih znakova na obrazac veb stranice za e-trgovinu, to je tip `string`; ukoliko vam zatim zatreba da tu vrednost upotrebite u matematičkim operacijama, morate je *konvertovati* u tip `number`.

JavaScript ima više mehanizama za bezuslovnu konverziju *tipova*. Na primer:

```
var a = "42";

var b = Number( a );

console.log( a );  // "42"
console.log( b );  // 42
```

Prikazana upotreba funkcije `Number(..)` (ugrađena funkcija) predstavlja *eksplicitnu* konverziju proizvoljnog izvornog tipa u tip `number`. To bi trebalo da je samo po sebi razumljivo.

Ali, šta se događa kada pokušate da poredite dve vrednosti koje izvorno nisu istog tipa, pa bi bila potrebna *implicitna* konverzija tipa?

Kada se poredi znakovni niz `"99.99"` s brojem `99.99`, većina ljudi bi se složila da su te vrednosti ekvivalentne. Ali one nisu sasvim iste, zar ne? To je ista vrednost predstavljena u dva različita oblika, tj. dva različita *tipa*. Moglo bi se reći da su one „labavo jednake“, zar ne?

Da bi vam olakšao rešavanje ovakvih uobičajenih situacija, JavaScript se u određenim slučajevima sam umeša i *implicitno* pretvara vrednosti u odgovarajuće ciljne tipove.

Tako da ako upotrebite operator labave jednakosti `==` da biste uporedili `"99.99" == 99.99`, JavaScript će konvertovati levu stranu `"99.99"` u njen ekvivalent tipa `number`, što je `99.99`. Poređenje se zatim svodi na `99.99 == 99.99`, čiji rezultat je, razume se, `true`.

Uprkos tome što je osmišljena da pomogne, implicitna konverzija tipova može da vas zbuni ako niste prethodno odvojili vreme i naučili pravila koja pri tome važe. Pošto većina JS programera to ne čini, uobičajeno osećanje je da konverzija tipova zbunjuje i uvodi u programe neočekivane greške, pa bi je zato trebalo izbegavati. Ponekad se čak naziva i greškom u dizajnu jezika.

Međutim, implicitna konverzija je mehanizam koji *se može naučiti*, štaviše, *treba da ga nauči* svako ko namerava da se ozbiljno bavi programiranjem na JavaScriptu. Ne samo da više ne zbunjuje kada savladate njegova pravila, nego može zapravo i poboljšati vaše programe! Uloženi trud je vredan rezultata.



Više informacija o konverziji tipova naći ćete u poglavlju 2 ovog dela knjige i u poglavlju 4 četvrtog dela – *Tipovi i gramatika*.

Komentari u kodu

Prodavac mobilnih telefona može da napiše nekoliko beležaka o odlikama novog modela telefona u ponudi ili novim planovima firme za buduće ponude. Te napomene su namenjene samo zaposlenima – njih ne treba da čitaju kupci. Uprkos tome, te beleške omogućavaju zaposlenima da lakše obavljaju svoje poslove jer je bolje dokumentovano šta i zbog čega treba da pričaju kupcima.

Jedna od najvažnijih lekcija koje možete naučiti u vezi s pisanjem koda jeste da on nije namenjen samo računaru. Programski kôd je u svakom svom deliću namenjen isto toliko, ako ne i više, programeru koliko i kompajleru.

Računaru je bitan samo mašinski kôd, što je niz binarnih jedinica i nula, koji se dobija nakon *kompajliranja* (*prevođenja*) izvornog koda. Postoji gotovo beskonačan broj programa koji mogu rezultovati istim nizom nula i jedinica. Odluke koje donosite u vezi s načinom na koji pišete program imaju svoju težinu – ne samo za vas, nego i za druge članove razvojnog tima, pa čak i za vas ubuduće.

Trebalo bi da pišete programe koji ne samo što rade ispravno, nego i izgledaju razumljivo kada ih neko proučava. Tome možete mnogo doprineti ako birate dobra imena za promenljive (videti odeljak „Promenljive“ na strani 15) i funkcije (videti odeljak „Funkcije“ na strani 22).

Drugi važan deo su komentari u kodu. To su kratki tekstovi koji se dodaju u program isključivo kao objašnjenja za ljude koji čitaju kôd (uključujući i vas, kao programera). Interpreter/kompajler uvek zanemaruje te komentare.

Postoje razna mišljenja o tome šta čini dobro dokumentovan kôd; nije moguće definisati apsolutna i univerzalna pravila. Ipak, pojedine napomene i smernice prilično su korisne:

- Programski kôd bez komentara nikad nije optimalan.
- Previše komentara (na primer, po jedan u svakom redu) verovatno je znak loše napisanog koda.
- Trebalo bi da komentari objašnjavaju *zašto*, a ne *šta*. Mogu da objašnjavaju i *kako* ako je ono što je napisano posebno zbunjujuće.

U JavaScriptu, moguće su dve vrste komentara: jednoredni i višeredni.
Razmotrite sledeće:

```
// Ovo je jednoredni komentar

/* a ovo je
     višeredni
          komentar.
                    */
```

Jednoredni komentar (označen znakovima //) pogodan je kada nameravate da dodate komentar neposredno iznad jednog iskaza ili čak na kraj reda. Sve u redu iza znakova //, do kraja reda, smatra se komentarom (i kompajler ga zato zanemaruje). Nema nikakvog ograničenja po pitanju toga šta možete napisati u sklopu jednorednog komentara.

Razmotrite sledeće:

```
var a = 42;    // 42 je smisao života
```

Višeredni komentar (označen znakovima `/* .. */`) pogodan je ako imate objašnjenje koje zauzima više redova.

Evo uobičajenog oblika upotrebe višerednih komentara:

```
/* Koristi se sledeća vrednost zato što
   je dokazano da ona odgovara na svako
   pitanje postavljeno u univerzumu. */
var a = 42;
```

Višeredni komentar se može umetnuti na proizvoljno mesto u redu, čak i usred reda, zato što se završava znakovima `*/`. Na primer:

```
var a = /* proizvoljna vrednost */ 42;

console.log( a ); // 42
```

Jedini element koji se ne može pojaviti u višerednom komentaru jeste grupa znakova `*/`, zato što bi se to tumačilo kao kraj komentara.

Trebalo bi svakako da svoje učenje programiranja počnete navikom da komentarišete programski kôd. Pošto ćete u celom preostalom delu ovog poglavlja videti da koristim komentare kako bih objasnio određene stvari, činite to isto u svom kodu. Verujte mi, biće vam zahvalan svako ko čita vaš kôd!

Promenljive

U većini programa koji rade nešto korisno, potrebno je pratiti određenu vrednost koja se menja tokom izvršavanja programa jer učestvuje u više operacija potrebnih za poslove koje program obavlja.

U svom programu, to ćete najlakše obezbediti ako tu vrednost dodelite određenom simboličkom kontejneru – koji se zove *promenljiva* (engl. *variable*) – zato što se vrednost u tom kontejneru može tokom vremena *menjati*, prema potrebama.

U nekim programskim jezicima, promenljivu (kontejner) deklarišete tako da uvek sadrži samo određen tip vrednosti – recimo, `number` ili `string`. *Statičko određivanje tipa*, poznato i kao *nametanje tipa*, uglavnom se navodi kao prednost kad je reč o ispravnosti programa, zato što sprečava nenamerne konverzije tipova.

U drugim programskim jezicima, tipovi vrednosti smatraju se važnijim od tipova promenljivih. *Labavo određivanje tipa*, što je poznato i kao *dinamičko određivanje tipa*, omogućava da jedna promenljiva može u svakom datom trenutku sadržati vrednost proizvoljnog tipa. To se uglavnom navodi kao prednost kad je reč o fleksibilnosti programa, jer omogućava da ista promenljiva predstavlja vrednost bez obzira na oblik u kojem se ta vrednost može naći u svakom datom trenutku logičkog toka programa.

JavaScript koristi ovaj drugi pristup, *dinamičko određivanje tipa*, što znači da promenljive mogu da sadrže vrednosti proizvoljnog *tipa*, bez ikakvog nametanja *tipa*.

POGLAVLJE 4

Šta je opseg vidljivosti promenljive?

Jedna od najosnovnijih paradigmi gotovo svih programskih jezika jeste mogućnost smeštanja vrednosti u promenljive i zatim učitavanja ili menjanja tih vrednosti. U stvari, mogućnost čuvanja vrednosti u promenljivama i njihovog učitavanja iz promenljivih jeste ono što određuje tekuće *stanje* programa.

Bez jednog takvog koncepta, program bi mogao da obavlja određene poslove, ali bi bio veoma ograničen i ne baš mnogo zanimljiv.

Uvođenje promenljivih u program nameće najzanimljivija pitanja koja ćemo sada razmotriti: gde te promenljive *žive*? Drugim rečima, gde se one čuvaju? I, što je najvažnije, kako ih naš program pronalazi kad mu zatrebaju?

Ta pitanja dovode do potrebe za dobro definisanim skupom pravila za čuvanje promenljivih na odgovarajućem mestu, kao i za kasnije pronalaženje tih promenljivih. Taj skup pravila nazvaćemo: *opseg vidljivosti* (engl. *scope*) promenljivih.

Ali, gde se i kako definišu pravila tog *opsega vidljivosti*?

Teorija kompajlera

Možda je za nekoga očigledno, a za drugog iznenađujuće – zavisno od vašeg nivoa poznavanja raznih programskih jezika – ali uprkos činjenici da JavaScript pripada opštoj kategoriji „dinamičkih“ ili „interpretiranih“ jezika, to je zapravo kompajliran jezik. *Nije* unapred kompajliran, kao mnogi drugi tradicionalni jezici, niti su rezultati kompajliranja prenosivi na druge distribuirane sisteme.

Ipak, JS mašina obavlja veliki broj istih koraka, mada na složenije načine nego što smo toga uglavnom svesni, kao i svaki drugi tradicionalni kompajler jezika.

U postupku tradicionalnog kompajliranja jezika, uobičajeno je da blok izvornog koda, tj. vaš program, prolazi kroz tri koraka *pre* nego što se izvrši, a sve zajedno je poznato pod opštim imenom „kompajliranje“:

Razlaganje na tokene / leksička analiza

Deljenje grupe znakova na celine koje imaju značenje (u datom jeziku) i koje se zovu tokeni. Na primer, razmotrite program `var a = 2;`. Ovaj program bi verovatno bio podeljen na sledeće tokene: `var`, `a`, `=`, `2` i `;`. Beline između znakova možda će se smatrati tokenima, ali to nije obavezno i zavisi od toga da li one imaju u jeziku neko značenje ili ne.



Razlika između deljenja na tokene i leksičke analize suptilna je i akademskog nivoa, ali se zasniva na tome da li se ti tokeni obrađuju kao da opisuju određeno *stanje* ili ne. Jednostavno rečeno, ako pri tumačenju da li `a` treba smatrati zasebnim tokenom ili samo delom drugog tokena, sistem za razlaganje na tokene primenjuje pravila koja podrazumevaju i prepoznavanje stanja, onda bi *to* bila *leksička analiza*.

Raščlanjivanje

Raščlanjivanje (engl. *parsing*) jeste raspoređivanje niza tokena na stablo sa ugnežđenim elementima, koji uzeti kao celina predstavljaju gramatičku strukturu programa. To stablo se zove „AST“ (od engl. *Abstract Syntax Tree* – apstraktno sintaksno stablo).

Stablo za iskaz `var a = 2;` moglo bi da počne od čvora najvišeg nivoa, koji se zove `VariableDeclaration` i koji ima čvor potomak nazvan `Identifier` (čija vrednost je `a`), kao i još jednog potomka, `AssignmentExpression`, koji pak ima svog potomka po imenu `NumericLiteral` (a vrednost mu je `2`).

Generisanje koda

Postupak pretvaranja AST-a u izvršiv kôd. Ovaj deo se prilično razlikuje u zavisnosti od jezika, ciljne platforme itd.

Zato ćemo, umesto da se izgubimo među detaljima, samo reći da postoji način da se naš prethodno opisani AST za iskaz `var a = 2;` pretvori u skup mašinskih naredaba čiji je posao da *naprave* promenljivu koja se zove `a` (što podrazumeva rezervisanje memorije za nju itd.), i da zatim smeste određenu vrednost u `a`.



Pošto se detalji načina na koji mašina jezika upravlja resursima sistema nalaze dublje nego što ćemo mi kopati, uzećemo zdravo za gotovo da mašina jezika ume da pravi i skladišti promenljive prema potrebama.

Mašina JavaScripta obavlja poslove koji su znatno složeniji od opisana tri koraka – što važi i za većinu drugih kompajlera programskih jezika. Na primer, tokom postupaka raščlanjivanja i generisanja koda, svakako ima koraka za optimizovanje performansi pri izvršavanju koda, kao što je uklanjanje dupliranih elemenata itd.

Dakle, ovde slikam samo širokim potezima četkice. Ali mislim da ćete uskoro shvatiti zbog čega su bitni ti detalji koje *ipak* razmatramo, čak i ako je to samo na visokom nivou.

Prvo, budući da JavaScript mašine nemaju taj luksuz (kao kompajleri drugih jezika) da im je na raspolaganju dovoljno vremena da optimizuju kôd, JavaScript kôd se ne kompajlira unapred, u okviru postupka sklapanja izvršne verzije koda, kao u slučaju drugih jezika.

U mnogim slučajevima, JavaScript kôd se kompajlira jedva nekoliko mikrosekundi (ili još kraće!) pre izvršavanja koda. Da bi obezbedile najbolje performanse, JS mašine primenjuju sve moguće vrste trikova (kao što je JIT, odloženo kompajliranje ili čak kompajliranje „u letu“ itd.) koji su daleko izvan „opsega vidljivosti“ našeg razmatranja.

Jednostavnosti radi, reći ćemo samo da svaki delić JavaScript koda mora da bude kompajliran pre (uglavnom *neposredno* pre!) nego što se izvrši. Tako će program `var a = 2;` JS kompajler *prvo* kompajlirati, a zatim pripremiti za izvršavanje, obično odmah nakon toga.

Opis opsega vidljivosti

Da bismo objasnili opseg vidljivosti, razmišljaćemo o njemu kao o razgovoru između više sagovornika. Ali, *ko* su učesnici tog razgovora?

Podela uloga

Da bismo razumeli razgovor kojem ćemo uskoro prisustvovati, moramo znati uloge likova koji učestvuju u razgovoru povodom kompajliranja programa `var a = 2;`:

Mašina jezika
: Odgovorna je za kompajliranje od početka do kraja našeg JavaScript programa, kao i za njegovo izvršavanje.

Kompajler
: Jedan od Mašininih prijatelja; obavlja prljavi posao raščlanjivanja izvornog koda i generisanja mašinskog koda (videti prethodni odeljak).

Opseg vidljivosti
: Drugi Mašinin prijatelj; formira i održava pretraživu listu svih deklarisanih identifikatora (promenljivih) i stara se o poštovanju skupa striktnih pravila koja određuju kako su ti identifikatori dostupni tekućem kodu koji se izvršava.

Da biste *potpuno razumeli* kako radi JavaScript, morate početi da *razmišljate* na isti način kao što razmišlja Mašina (i njeni prijatelji), postavljati pitanja koja oni postavljaju i odgovarati na ta pitanja na isti način kao oni.

Ja tebi, ti meni

Kada vidite program kao što je `var a = 2;`, najverovatnije pomislite da je to samo jedan iskaz. Međutim, to nije oblik u kojem ga vidi naša nova prijateljica Mašina. Ona zapravo vidi dva jasno razdvojena iskaza – jedan koji će Kompajler obraditi tokom postupka kompajliranja i drugi koji će Mašina obraditi tokom izvršavanja koda.

Sada ćemo opisati način na koji Mašina i njeni prijatelji pristupaju programu `var a = 2;`.

Kompajler će prvo obaviti leksičku analizu ovog programa da bi ga razložio na pojedinačne tokene, koje će zatim raščlaniti i raspodeliti na stablo. Ali, kad Kompajler stigne u fazu generisanja koda, on će naš program obrađivati donekle drugačije nego što možda pretpostavljate.

Razumna pretpostavka bi bila da će Kompajler proizvesti kôd čiji se rad može opisati sledećim pseudokodom: „Rezervisati memoriju za promenljivu, obeležiti je sa `a`, i zatim u tu promenljivu postaviti vrednost `2`“. Nažalost, to nije sasvim tačan opis.

Kompajler će umesto toga uraditi sledeće:

1. Kada naiđe na deklaraciju `var a`, Kompajler pita Opseg vidljivosti da li promenljiva `a` već postoji u njegovoj kolekciji promenljivih. Ako postoji, Kompajler zanemaruje deklaraciju i nastavlja dalje. U suprotnom, Kompajler zahteva da Opseg vidljivosti deklariše u svojoj kolekciji novu promenljivu čije je ime `a`.
2. Kompajler onda generiše kôd koji će Mašina kasnije izvršiti i koji obrađuje iskaz `a = 2`. Kôd koji Mašina izvršava prvo će pitati Mašinu da li je u kolekciji promenljivih tekućeg opsega vidljivosti dostupna neka promenljiva čije je ime `a`. Ako jeste, Mašina koristi tu promenljivu. Ukoliko nije, Mašina traži promenljivu *na drugom mestu* (videti odeljak „Ugnežđeni opsezi vidljivosti“ na strani 69).

Ako Mašina na kraju pronađe promenljivu, dodeljuje joj vrednost `2`. Ukoliko je ne pronađe, Mašina će prijaviti da je došlo do greške.

Ukratko: pri dodeljivanju vrednosti promenljivoj, odvijaju se dve akcije: prvo, Kompajler deklariše promenljivu (ako nije prethodno deklarisana) u tekućem Opsegu vidljivosti; drugo, pri izvršavanju koda, Mašina traži promenljivu unutar Opsega vidljivosti i dodeljuje joj vrednost, ako je tamo nađe.

Terminologija kompajlera

Da bismo nastavili proučavanje, treba nam još malo kompajlerske terminologije.

Kada Mašina izvršava kôd koji je Kompajler pripremio za korak 2, mora da potraži promenljivu `a` kako bi videla da li je promenljiva deklarisana, a ta pretraga se odvija unutar Opsega vidljivosti. Ali, rezultat pretrage zavisi od načina na koji Mašina obavi pretragu.

U našem slučaju, kaže se da bi Mašina obavila LHS pretragu za promenljivu `a`. Druga vrsta pretrage zove se RHS.

Možda ste pogodili šta znače „L“ i „R“. Tim slovima počinju izrazi *lefthand side* (leva strana) i *righthand side* (desna strana).

Strana...čega? *Operacije dodeljivanja vrednosti promenljivoj.*

Drugim rečima, LHS pretraživanje se obavlja kada se promenljiva pojavljuje na levoj strani operacije dodeljivanja vrednosti, a RHS – kada se promenljiva pojavljuje na desnoj strani te operacije.

U stvari, budimo malo precizniji. RHS pretraga se ne razlikuje, u našem primeru, od običnog traženja vrednosti određene promenljive, dok LHS pretraga pokušava da pronađe samu promenljivu, tj. kontejner za vrednost, da bi joj se dodelila vrednost. Stoga, RHS ne znači samo po sebi *zaista* „desna strana operacije dodeljivanja vrednosti“, nego, tačnije rečeno, „suprotno od leve strane“.

Pošto je vaše poznavanje materije zasad još nedovoljno duboko (ali razumete engleski), mogli biste pomisliti da RHS u stvari znači *„retrieve his/her source (value)“* /učitati njegovu/njenu izvornu (vrednost)/ što bi uputilo na to da RHS znači „učitati vrednost promenljive ...“

Razmotrimo to detaljnije.

Kada kažem:

```
console.log( a );
```

referenca `a` je RHS referenca, zato što se ovde promenljivoj `a` ništa ne dodeljuje. Umesto toga, pretražujemo kolekciju promenljivih da bismo učitali vrednost promenljive `a`, koju treba proslediti funkciji `console.log(..)`.

Nasuprot tome:

```
a = 2;
```

U ovom slučaju, referenca `a` je LHS referenca, zato što nam zapravo nije važno koja je tekuća vrednost, nego samo želimo da pronađemo promenljivu kao cilj operacije dodeljivanja vrednosti `= 2`.



Značenje akronima LHS i RHS „leva/desna strana operacije dodeljivanja vrednosti“ ne mora da znači bukvalno „leva/desna strana operatora `=` za dodeljivanje vrednosti“. Pošto se vrednost može dodeliti na još nekoliko načina, bolje je da ga konceptualno zamislite kao: „Ko je odredište u operaciji (LHS)?“ i „Ko je izvor u operaciji (RHS)?“

Razmotrite sledeći program, koji sadrži i LHS i RHS reference:

```
function foo(a) {
    console.log( a ); // 2
}

foo( 2 );
```

Poslednji red – koji poziva funkciju `foo(..)` – predstavlja RHS referencu na `foo`, koja znači „Potraži vrednost `foo` i daj mi je“. Osim toga, pošto `(..)` znači da tu vrednost `foo` treba izvršiti, bilo bi bolje da je to zaista funkcija!

U ovom slučaju imamo neočiglednu, ali važnu operaciju dodeljivanja vrednosti.

Možda vam je u ovom primeru koda promakla implicitna operacija `a = 2`. Ona se izvršava kada se vrednost `2` prosledi kao argument funkciji `foo(..)`, pri čemu se vrednost `2` *dodeljuje* parametru `a`. Da bi se (implicitno) dodelila vrednost parametru `a`, obavlja se LHS pretraga.

Postoji i RHS referenca na vrednost `a`, a ta rezultujuća vrednost prosleđuje se funkciji `console.log(..)`. Funkciji `console.log(..)` potrebna je referenca za izvršavanje. Tu se odvija RHS pretraga radi pronalaženja objekta `console`, a zatim se razrešava njegovo svojstvo radi utvrđivanja da li taj objekat ima metodu koja se zove `log`.

I najzad, ovde možemo konceptualno zamisliti da se odvija LHS/RHS razmena u kojoj se vrednost `2` prosleđuje (putem RHS pretrage promenljive `a`) funkciji `log(..)`. Unutar interne implementacije funkcije `log(..)`, možemo pretpostaviti da ona ima parametre, među kojima se za prvi (možda nazvan `arg1`) obavlja LHS pretraga reference, pre nego što mu se dodeli vrednost `2`.



Možda biste došli u iskušenje da deklaraciju funkcije `function foo(a) {…` konceptualno zamislite kao običnu deklaraciju promenljive s dodeljivanjem vrednosti, kao što je `var foo` i `foo = function(a){….` Kada tako uradite, izgledalo bi privlačno da tu deklaraciju funkcije zamislite kao da podrazumeva LHS pretragu.

Međutim, suptilna ali bitna razllika je to što Kompajler obrađuje i deklarisanje promenljive i definisanje njene vrednosti tokom postupka generisanja koda, tako da kada Mašina izvršava kôd, nije potrebna nikakva obrada da bi se funkciji `foo` „dodelila“ vrednost. Iz tog razloga, deklaracija funkcije se ne može posmatrati kao dodeljivanje vrednosti sa LHS pretragom u obliku u kojem ih ovde razmatramo.

Razgovor između Mašine i Opsega vidljivosti

```
function foo(a) {
    console.log( a ); // 2
}

foo( 2 );
```

Zamislimo prethodnu razmenu (u kojoj se obrađuje navedeni blok koda) kao nekakv razgovor. Taj razgovor bi se odvijao otprilike ovako:

Mašina: Zdravo Opseže, imam jednu RHS referencu `foo`. Da li si nekad čuo za nju?

Opseg vidljivosti: Jesam. Kompajler ju je deklarisao pre samo jedne sekunde. To je funkcija. Evo, izvoli.

Mašina: Odlično, hvala! OK, izvršavam `foo`.

Mašina: Zdravo, Opseže, imam jednu LHS referencu `a`. Da li si nekad čuo za nju?

Opseg vidljivosti: Jesam. Kompajler ju je nedavno deklarisao kao formalni parametar finkcije `foo`. Evo, izvoli.

Mašina: Kao i uvek, ti si mi velika pomoć, Opseže. Hvala još jednom. A sada, vreme je da dodelim `2` promenljivoj `a`.

Mašina: Zdravo, Opseže, izvinjavam se što te opet gnjavim. Treba mi RHS pretraga za `console`. Da li si nekad čuo za to?

Opseg vidljivosti: Nema problema, Mašino, to je ono što radim po ceo dan. Da, imam `console`. To je ugrađena funkcija. Evo, izvoli.

Mašina: Savršeno. Tražim `log(..)`. OK, super, to je funkcija.

Mašina: Zdravo, Opseže. Možeš li mi pomoći s RHS referencom `a`? Mislim da je se sećam, ali ipak želim da proverim još jednom.

Opseg: U pravu si, Mašino. Ista promenljiva, ništa se nije promenilo. Evo, izvoli.

Mašina: Super. Prosleđujem vrednost promenljive `a`, što je `2`, funkciji `log(..)`.

...

Kviz

Proverite znanje koje ste dosad stekli. Igrajte ulogu Mašine i obavite „razgovor“ s Opsegom vidljivosti:

```
function foo(a) {
    var b = a;
    return a + b;
}

var c = foo( 2 );
```

1. Otkrijte sve LHS pretrage (ima ih 3!).
2. Otkrijte sve RHS pretrage (ima ih 4!).



Odgovore na pitanja iz kviza potražite u sažetku poglavlja!

Ugnežđeni opsezi vidljivosti

Rekli smo da je opseg vidljivosti skup pravila za traženje promenljivih po imenima njihovih identifikatora. Međutim, obično ima više opsega vidljivosti koje treba pregledati.

Isto kao što je blok jedne funkcije ugnežđen unutar bloka druge funkcije, tako su i jedni opsezi vidljivosti ugnežđeni unutar drugih. Zbog toga, ako određenu promenljivu ne uspe da pronađe unutar tekućeg opsega vidljivosti, Mašina pretražuje prvi sledeći spoljašnji i tako nastavlja sve dok ne pronađe promenljivu ili dok ne dođe do spoljašnjeg opsega najvišeg nivoa (poznat i kao globalni opseg vidljivosti).

Razmotrite sledeće:

```
function foo(a) {
    console.log( a + b );
}

var b = 2;

foo( 2 ); // 4
```

RHS referenca `b` ne može se razrešiti unutar funkcije `foo`, ali može u opsegu koji okružuje tu funkciju (što je, u ovom slučaju, globalni opseg vidljivosti).

Zato kada bismo se vratili na razgovor između Mašine i Opsega vidljivosti, čuli bismo sledeće:

Mašina: „Zdravo, Opseže vidljivosti funkcije `foo`, da li si čuo za `b`? Imam jednu RHS referencu takve promenljive."

Opseg vidljivosti: „Ne, nikad čuo za to. Traži dalje."

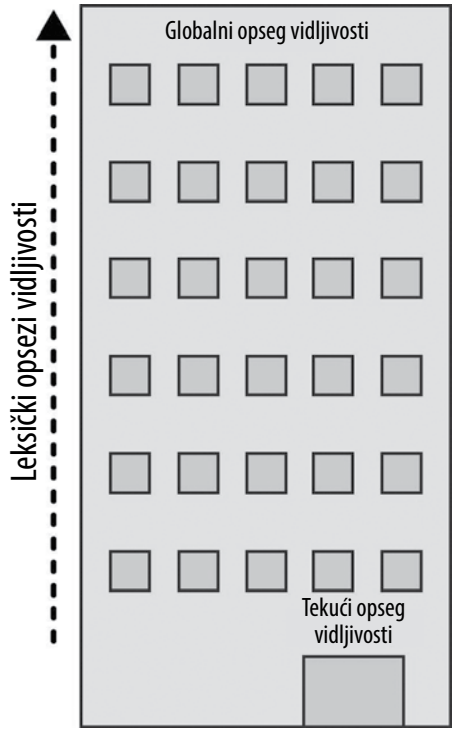
Mašina: „Zdravo, Opseže vidljivosti izvan funkcije `foo`; oh, to je globalni opseg vidljivosti, super. Da li si čuo za `b`? Imam RHS referencu takve promenljive."

Opseg vidljivosti: „Jesam, svakako. Evo, izvoli."

Jednostavna pravila za pretraživanje ugnežđenog opsega vidljivosti: Mašina prvo traži promenljivu unutar opsega vidljivosti tekuće funkcije koja se izvršava; ako je tamo ne nađe, prelazi na prvi sledeći okružujući nivo itd. Kada dođe do spoljašnjeg, globalnog opsega vidljivosti, pretraživanje se završava, bez obzira na to da li je promenljiva nađena ili nije.

Gradnja na metaforama

Da biste sebi vizuelno predstavili pretraživanja ugnežđenih opsega vidljivosti, zamislite sledeću visoku zgradu:



Zgrada predstavlja skup pravila koja važe za pretraživanje ugnežđenih opsega vidljivosti u našem programu. Tekući sprat zgrade, koji god to bio, predstavlja opseg vidljivosti funkcije koja se trenutno izvršava. Najviši sprat zgrade je globalni opseg vidljivosti.

LHS i RHS reference razrešavate tako što referencu prvo tražite na tekućem spratu, pa ako je tamo ne nađete, uzimate lift do sledećeg višeg sprata i tražite tamo, zatim na sledećem višem spratu itd. Kada stignete na najviši sprat (globalni opseg vidljivosti), tamo ili nalazite ili ne nalazite ono što ste tražili. U svakom slučaju, morate prekinuti pretragu.

Greške

Zbog čega je važno da li se obavlja LHS ili RHS pretraga?

Zato što se te dve vrste pretrage ponašaju različito u slučaju kada promenljiva nije deklarisana (nije pronađena ni u jednom pregledanom opsegu vidljivosti).

Razmotrite sledeće:

```
function foo(a) {
    console.log( a + b );
    b = a;
}

foo( 2 );
```

Kada se prvi put izvrši RHS pretraga za `b`, ta promenljiva neće biti pronađena. Tada se ta promenljiva smatra „nedeklarisanom“ jer nije pronađena u opsegu vidljivosti.

Ako RHS pretraga nigde ne pronađe traženu promenljivu, ni u jednom od ugnežđenih opsega vidljivosti, mašina jezika generiše grešku `ReferenceError`. Važno je imati u vidu da je tip te greške `ReferenceError`.

Nasuprot tome, ukoliko mašina obavlja LHS pretragu i dođe do najvišeg sprata (globalnog opsega vidljivosti) a da nije pronašla promenljivu, ako program ne radi u „striktnom režimu“[1], globalni opseg vidljivosti napraviće novu promenljivu s traženim imenom *u globalnom opsegu vidljivosti* i proslediti je Mašini.

„Ne, pre nije bilo takve promenljive, ali hteo sam da pomognem, pa sam ti napravio jednu“.

„Striktni režim“ koji je uveden u verziji ES5, ima više ponašanja različitih od onih u normalnom/opuštenom/lenjom režimu. Jedno od takvih ponašanja je da ne dozvoljava automatsko/implicitno generisanje globalnih promenljivih. U tom slučaju, ne bi bilo nijedne promenljive s globalnim opsegom vidljivosti koji bi LHS pretraga mogla da pronađe, a Mašina bi generisala grešku `ReferenceError`, slično kao pri RHS pretrazi.

Ako je promenljiva pronađena tokom RHS pretrage, ali vi pokušate da s njenom vrednošću uradite nešto što nije moguće – recimo, da izvršite kao funkciju promenljivu koja to nije, ili da referencirate svojstvo vrednosti koja je `null` ili `undefined` – onda Mašina generiše grešku drugačijeg tipa, koji se zove `TypeError`.

`ReferenceError` znači neuspešno pretraživanje opsega vidljivosti, dok `TypeError` znači da je pretraživanje opsega vidljivosti bilo uspešno, ali da ste nad rezultatom pretrage pokušali neku nedozvoljenu/nemoguću akciju.

Sažetak poglavlja

Opseg vidljivosti je skup pravila koja određuju gde i kako se može pronaći data promenljiva (identifikator). Razlog te pretrage može biti dodeljivanje vrednosti promenljivoj, što

1. Videti u MDN-u objašnjenje striktnog režima.

se zove LHS referenca (na levu stranu), ili učitavanje njene vrednosti, što se zove RHS referenca (na desnu stranu).

LHS reference su rezultat operacija dodeljivanja vrednosti. Dodeljivanje vrednosti promenljivoj unutar opsega može se dogoditi zahvaljujući upotrebi operatora `=` ili prosleđivanju (dodeljivanju) vrednosti parametrima funkcija pomoću argumenata.

Mašina JavaScripta prvo kompajlira kôd pre nego što ga izvrši, pri čemu deli iskaze programa, kao što je `var a = 2;`, u dva koraka:

1. Prvo izvršava deo `var a`, da bi deklarisala tu promenljivu u opsegu vidljivosti programa. To se odvija na početku, pre izvršavanja koda.
2. Zatim, izvršava deo `a = 2`, da bi pronašla promenljivu (LHS referencu) i dodelila joj vrednost, ako je nađe.

Obe vrste traženja reference, i LHS i RHS, započinju postupak traženja u opsegu vidljivosti tekuće funkcije u toku izvršavanja, a ukoliko je potrebno (tj. ako tamo ne nađu ono što traže), traže identifikator redom na svim sledećim nivoima ugnežđenih opsega vidljivosti, opseg (sprat) po opseg, sve dok ne dođu do globalnog nivoa (najvišeg sprata) i prekinu pretragu, a tamo ili nađu traženi identifikator ili ga ne nađu.

Ako tražena RHS referenca ne postoji, rezultat je greška tipa `ReferenceError`. Nepostojeća LHS referenca prouzrokuje automatsko generisanje implicitno deklarisane globalne promenljive s traženim imenom (ukoliko nije uključen striktni režim), ili grešku tipa `ReferenceError` (ako je uključen striktni režim).

Odgovori na pitanja iz kviza

```
function foo(a) {
    var b = a;
    return a + b;
}

var c = foo( 2 );
```

1. Otkrijte sve LHS pretrage (ima ih 3!).

   `c = ..;, a = 2 (implicitno dodeljivanje vrednosti parametrima) i b = ..`

2. Otkrijte sve RHS pretrage (ima ih 4!).

   `foo(2.., = a;, a .. i .. b`

POGLAVLJE 4

Šta je opseg vidljivosti promenljive?

Jedna od najosnovnijih paradigmi gotovo svih programskih jezika jeste mogućnost smeštanja vrednosti u promenljive i zatim učitavanja ili menjanja tih vrednosti. U stvari, mogućnost čuvanja vrednosti u promenljivama i njihovog učitavanja iz promenljivih jeste ono što određuje tekuće *stanje* programa.

Bez jednog takvog koncepta, program bi mogao da obavlja određene poslove, ali bi bio veoma ograničen i ne baš mnogo zanimljiv.

Uvođenje promenljivih u program nameće najzanimljivija pitanja koja ćemo sada razmotriti: gde te promenljive *žive*? Drugim rečima, gde se one čuvaju? I, što je najvažnije, kako ih naš program pronalazi kad mu zatrebaju?

Ta pitanja dovode do potrebe za dobro definisanim skupom pravila za čuvanje promenljivih na odgovarajućem mestu, kao i za kasnije pronalaženje tih promenljivih. Taj skup pravila nazvaćemo: *opseg vidljivosti* (engl. *scope*) promenljivih.

Ali, gde se i kako definišu pravila tog *opsega vidljivosti*?

Teorija kompajlera

Možda je za nekoga očigledno, a za drugog iznenađujuće – zavisno od vašeg nivoa poznavanja raznih programskih jezika – ali uprkos činjenici da JavaScript pripada opštoj kategoriji „dinamičkih“ ili „interpretiranih“ jezika, to je zapravo kompajliran jezik. *Nije* unapred kompajliran, kao mnogi drugi tradicionalni jezici, niti su rezultati kompajliranja prenosivi na druge distribuirane sisteme.

Ipak, JS mašina obavlja veliki broj istih koraka, mada na složenije načine nego što smo toga uglavnom svesni, kao i svaki drugi tradicionalni kompajler jezika.

U postupku tradicionalnog kompajliranja jezika, uobičajeno je da blok izvornog koda, tj. vaš program, prolazi kroz tri koraka *pre* nego što se izvrši, a sve zajedno je poznato pod opštim imenom „kompajliranje“:

Razlaganje na tokene / leksička analiza

Deljenje grupe znakova na celine koje imaju značenje (u datom jeziku) i koje se zovu tokeni. Na primer, razmotrite program `var a = 2;`. Ovaj program bi verovatno bio podeljen na sledeće tokene: `var`, `a`, `=`, `2` i `;`. Beline između znakova možda će se smatrati tokenima, ali to nije obavezno i zavisi od toga da li one imaju u jeziku neko značenje ili ne.



Razlika između deljenja na tokene i leksičke analize suptilna je i akademskog nivoa, ali se zasniva na tome da li se ti tokeni obrađuju kao da opisuju određeno *stanje* ili ne. Jednostavno rečeno, ako pri tumačenju da li `a` treba smatrati zasebnim tokenom ili samo delom drugog tokena, sistem za razlaganje na tokene primenjuje pravila koja podrazumevaju i prepoznavanje stanja, onda bi *to* bila *leksička analiza*.

Raščlanjivanje

Raščlanjivanje (engl. *parsing*) jeste raspoređivanje niza tokena na stablo sa ugnežđenim elementima, koji uzeti kao celina predstavljaju gramatičku strukturu programa. To stablo se zove „AST“ (od engl. *Abstract Syntax Tree* – apstraktno sintaksno stablo).

Stablo za iskaz `var a = 2;` moglo bi da počne od čvora najvišeg nivoa, koji se zove `VariableDeclaration` i koji ima čvor potomak nazvan `Identifier` (čija vrednost je `a`), kao i još jednog potomka, `AssignmentExpression`, koji pak ima svog potomka po imenu `NumericLiteral` (a vrednost mu je `2`).

Generisanje koda

Postupak pretvaranja AST-a u izvršiv kôd. Ovaj deo se prilično razlikuje u zavisnosti od jezika, ciljne platforme itd.

Zato ćemo, umesto da se izgubimo među detaljima, samo reći da postoji način da se naš prethodno opisani AST za iskaz `var a = 2;` pretvori u skup mašinskih naredaba čiji je posao da *naprave* promenljivu koja se zove `a` (što podrazumeva rezervisanje memorije za nju itd.), i da zatim smeste određenu vrednost u `a`.



Pošto se detalji načina na koji mašina jezika upravlja resursima sistema nalaze dublje nego što ćemo mi kopati, uzećemo zdravo za gotovo da mašina jezika ume da pravi i skladišti promenljive prema potrebama.

Mašina JavaScripta obavlja poslove koji su znatno složeniji od opisana tri koraka – što važi i za većinu drugih kompajlera programskih jezika. Na primer, tokom postupaka raščlanjivanja i generisanja koda, svakako ima koraka za optimizovanje performansi pri izvršavanju koda, kao što je uklanjanje dupliranih elemenata itd.

Dakle, ovde slikam samo širokim potezima četkice. Ali mislim da ćete uskoro shvatiti zbog čega su bitni ti detalji koje *ipak* razmatramo, čak i ako je to samo na visokom nivou.

Prvo, budući da JavaScript mašine nemaju taj luksuz (kao kompajleri drugih jezika) da im je na raspolaganju dovoljno vremena da optimizuju kôd, JavaScript kôd se ne kompajlira unapred, u okviru postupka sklapanja izvršne verzije koda, kao u slučaju drugih jezika.

U mnogim slučajevima, JavaScript kôd se kompajlira jedva nekoliko mikrosekundi (ili još kraće!) pre izvršavanja koda. Da bi obezbedile najbolje performanse, JS mašine primenjuju sve moguće vrste trikova (kao što je JIT, odloženo kompajliranje ili čak kompajliranje „u letu“ itd.) koji su daleko izvan „opsega vidljivosti“ našeg razmatranja.

Jednostavnosti radi, reći ćemo samo da svaki delić JavaScript koda mora da bude kompajliran pre (uglavnom *neposredno* pre!) nego što se izvrši. Tako će program `var a = 2;` JS kompajler *prvo* kompajlirati, a zatim pripremiti za izvršavanje, obično odmah nakon toga.

Opis opsega vidljivosti

Da bismo objasnili opseg vidljivosti, razmišljaćemo o njemu kao o razgovoru između više sagovornika. Ali, *ko* su učesnici tog razgovora?

Podela uloga

Da bismo razumeli razgovor kojem ćemo uskoro prisustvovati, moramo znati uloge likova koji učestvuju u razgovoru povodom kompajliranja programa `var a = 2;`:

Mašina jezika
: Odgovorna je za kompajliranje od početka do kraja našeg JavaScript programa, kao i za njegovo izvršavanje.

Kompajler
: Jedan od Mašininih prijatelja; obavlja prljavi posao raščlanjivanja izvornog koda i generisanja mašinskog koda (videti prethodni odeljak).

Opseg vidljivosti
: Drugi Mašinin prijatelj; formira i održava pretraživu listu svih deklarisanih identifikatora (promenljivih) i stara se o poštovanju skupa striktnih pravila koja određuju kako su ti identifikatori dostupni tekućem kodu koji se izvršava.

Da biste *potpuno razumeli* kako radi JavaScript, morate početi da *razmišljate* na isti način kao što razmišlja Mašina (i njeni prijatelji), postavljati pitanja koja oni postavljaju i odgovarati na ta pitanja na isti način kao oni.

Ja tebi, ti meni

Kada vidite program kao što je `var a = 2;`, najverovatnije pomislite da je to samo jedan iskaz. Međutim, to nije oblik u kojem ga vidi naša nova prijateljica Mašina. Ona zapravo vidi dva jasno razdvojena iskaza – jedan koji će Kompajler obraditi tokom postupka kompajliranja i drugi koji će Mašina obraditi tokom izvršavanja koda.

Sada ćemo opisati način na koji Mašina i njeni prijatelji pristupaju programu `var a = 2;`.

Kompajler će prvo obaviti leksičku analizu ovog programa da bi ga razložio na pojedinačne tokene, koje će zatim raščlaniti i raspodeliti na stablo. Ali, kad Kompajler stigne u fazu generisanja koda, on će naš program obrađivati donekle drugačije nego što možda pretpostavljate.

Razumna pretpostavka bi bila da će Kompajler proizvesti kôd čiji se rad može opisati sledećim pseudokodom: „Rezervisati memoriju za promenljivu, obeležiti je sa `a`, i zatim u tu promenljivu postaviti vrednost `2`“. Nažalost, to nije sasvim tačan opis.

Kompajler će umesto toga uraditi sledeće:

1. Kada naiđe na deklaraciju `var a`, Kompajler pita Opseg vidljivosti da li promenljiva `a` već postoji u njegovoj kolekciji promenljivih. Ako postoji, Kompajler zanemaruje deklaraciju i nastavlja dalje. U suprotnom, Kompajler zahteva da Opseg vidljivosti deklariše u svojoj kolekciji novu promenljivu čije je ime `a`.
2. Kompajler onda generiše kôd koji će Mašina kasnije izvršiti i koji obrađuje iskaz `a = 2`. Kôd koji Mašina izvršava prvo će pitati Mašinu da li je u kolekciji promenljivih tekućeg opsega vidljivosti dostupna neka promenljiva čije je ime `a`. Ako jeste, Mašina koristi tu promenljivu. Ukoliko nije, Mašina traži promenljivu *na drugom mestu* (videti odeljak „Ugnežđeni opsezi vidljivosti“ na strani 69).

Ako Mašina na kraju pronađe promenljivu, dodeljuje joj vrednost `2`. Ukoliko je ne pronađe, Mašina će prijaviti da je došlo do greške.

Ukratko: pri dodeljivanju vrednosti promenljivoj, odvijaju se dve akcije: prvo, Kompajler deklariše promenljivu (ako nije prethodno deklarisana) u tekućem Opsegu vidljivosti; drugo, pri izvršavanju koda, Mašina traži promenljivu unutar Opsega vidljivosti i dodeljuje joj vrednost, ako je tamo nađe.

Terminologija kompajlera

Da bismo nastavili proučavanje, treba nam još malo kompajlerske terminologije.

Kada Mašina izvršava kôd koji je Kompajler pripremio za korak 2, mora da potraži promenljivu `a` kako bi videla da li je promenljiva deklarisana, a ta pretraga se odvija unutar Opsega vidljivosti. Ali, rezultat pretrage zavisi od načina na koji Mašina obavi pretragu.

U našem slučaju, kaže se da bi Mašina obavila LHS pretragu za promenljivu `a`. Druga vrsta pretrage zove se RHS.

Možda ste pogodili šta znače „L“ i „R“. Tim slovima počinju izrazi *lefthand side* (leva strana) i *righthand side* (desna strana).

Strana...čega? *Operacije dodeljivanja vrednosti promenljivoj.*

Drugim rečima, LHS pretraživanje se obavlja kada se promenljiva pojavljuje na levoj strani operacije dodeljivanja vrednosti, a RHS – kada se promenljiva pojavljuje na desnoj strani te operacije.

U stvari, budimo malo precizniji. RHS pretraga se ne razlikuje, u našem primeru, od običnog traženja vrednosti određene promenljive, dok LHS pretraga pokušava da pronađe samu promenljivu, tj. kontejner za vrednost, da bi joj se dodelila vrednost. Stoga, RHS ne znači samo po sebi *zaista* „desna strana operacije dodeljivanja vrednosti“, nego, tačnije rečeno, „suprotno od leve strane“.

Pošto je vaše poznavanje materije zasad još nedovoljno duboko (ali razumete engleski), mogli biste pomisliti da RHS u stvari znači *„retrieve his/her source (value)“* /učitati njegovu/njenu izvornu (vrednost)/ što bi uputilo na to da RHS znači „učitati vrednost promenljive ...“

Razmotrimo to detaljnije.

Kada kažem:

```
console.log( a );
```

referenca `a` je RHS referenca, zato što se ovde promenljivoj `a` ništa ne dodeljuje. Umesto toga, pretražujemo kolekciju promenljivih da bismo učitali vrednost promenljive `a`, koju treba proslediti funkciji `console.log(..)`.

Nasuprot tome:

```
a = 2;
```

U ovom slučaju, referenca `a` je LHS referenca, zato što nam zapravo nije važno koja je tekuća vrednost, nego samo želimo da pronađemo promenljivu kao cilj operacije dodeljivanja vrednosti `= 2`.



Značenje akronima LHS i RHS „leva/desna strana operacije dodeljivanja vrednosti“ ne mora da znači bukvalno „leva/desna strana operatora `=` za dodeljivanje vrednosti“. Pošto se vrednost može dodeliti na još nekoliko načina, bolje je da ga konceptualno zamislite kao: „Ko je odredište u operaciji (LHS)?“ i „Ko je izvor u operaciji (RHS)?“

Razmotrite sledeći program, koji sadrži i LHS i RHS reference:

```
function foo(a) {
    console.log( a ); // 2
}

foo( 2 );
```

Poslednji red – koji poziva funkciju `foo(..)` – predstavlja RHS referencu na `foo`, koja znači „Potraži vrednost `foo` i daj mi je“. Osim toga, pošto `(..)` znači da tu vrednost `foo` treba izvršiti, bilo bi bolje da je to zaista funkcija!

U ovom slučaju imamo neočiglednu, ali važnu operaciju dodeljivanja vrednosti.

Možda vam je u ovom primeru koda promakla implicitna operacija `a = 2`. Ona se izvršava kada se vrednost `2` prosledi kao argument funkciji `foo(..)`, pri čemu se vrednost `2` *dodeljuje* parametru `a`. Da bi se (implicitno) dodelila vrednost parametru `a`, obavlja se LHS pretraga.

Postoji i RHS referenca na vrednost `a`, a ta rezultujuća vrednost prosleđuje se funkciji `console.log(..)`. Funkciji `console.log(..)` potrebna je referenca za izvršavanje. Tu se odvija RHS pretraga radi pronalaženja objekta `console`, a zatim se razrešava njegovo svojstvo radi utvrđivanja da li taj objekat ima metodu koja se zove `log`.

I najzad, ovde možemo konceptualno zamisliti da se odvija LHS/RHS razmena u kojoj se vrednost `2` prosleđuje (putem RHS pretrage promenljive `a`) funkciji `log(..)`. Unutar interne implementacije funkcije `log(..)`, možemo pretpostaviti da ona ima parametre, među kojima se za prvi (možda nazvan `arg1`) obavlja LHS pretraga reference, pre nego što mu se dodeli vrednost `2`.



Možda biste došli u iskušenje da deklaraciju funkcije `function foo(a) {…` konceptualno zamislite kao običnu deklaraciju promenljive s dodeljivanjem vrednosti, kao što je `var foo` i `foo = function(a){….` Kada tako uradite, izgledalo bi privlačno da tu deklaraciju funkcije zamislite kao da podrazumeva LHS pretragu.

Međutim, suptilna ali bitna razllika je to što Kompajler obrađuje i deklarisanje promenljive i definisanje njene vrednosti tokom postupka generisanja koda, tako da kada Mašina izvršava kôd, nije potrebna nikakva obrada da bi se funkciji `foo` „dodelila“ vrednost. Iz tog razloga, deklaracija funkcije se ne može posmatrati kao dodeljivanje vrednosti sa LHS pretragom u obliku u kojem ih ovde razmatramo.

Razgovor između Mašine i Opsega vidljivosti

```
function foo(a) {
    console.log( a ); // 2
}

foo( 2 );
```

Zamislimo prethodnu razmenu (u kojoj se obrađuje navedeni blok koda) kao nekakv razgovor. Taj razgovor bi se odvijao otprilike ovako:

Mašina: Zdravo Opseže, imam jednu RHS referencu `foo`. Da li si nekad čuo za nju?

Opseg vidljivosti: Jesam. Kompajler ju je deklarisao pre samo jedne sekunde. To je funkcija. Evo, izvoli.

Mašina: Odlično, hvala! OK, izvršavam `foo`.

Mašina: Zdravo, Opseže, imam jednu LHS referencu `a`. Da li si nekad čuo za nju?

Opseg vidljivosti: Jesam. Kompajler ju je nedavno deklarisao kao formalni parametar finkcije `foo`. Evo, izvoli.

Mašina: Kao i uvek, ti si mi velika pomoć, Opseže. Hvala još jednom. A sada, vreme je da dodelim `2` promenljivoj `a`.

Mašina: Zdravo, Opseže, izvinjavam se što te opet gnjavim. Treba mi RHS pretraga za `console`. Da li si nekad čuo za to?

Opseg vidljivosti: Nema problema, Mašino, to je ono što radim po ceo dan. Da, imam `console`. To je ugrađena funkcija. Evo, izvoli.

Mašina: Savršeno. Tražim `log(..)`. OK, super, to je funkcija.

Mašina: Zdravo, Opseže. Možeš li mi pomoći s RHS referencom `a`? Mislim da je se sećam, ali ipak želim da proverim još jednom.

Opseg: U pravu si, Mašino. Ista promenljiva, ništa se nije promenilo. Evo, izvoli.

Mašina: Super. Prosleđujem vrednost promenljive `a`, što je `2`, funkciji `log(..)`.

...

Kviz

Proverite znanje koje ste dosad stekli. Igrajte ulogu Mašine i obavite „razgovor“ s Opsegom vidljivosti:

```
function foo(a) {
    var b = a;
    return a + b;
}

var c = foo( 2 );
```

1. Otkrijte sve LHS pretrage (ima ih 3!).
2. Otkrijte sve RHS pretrage (ima ih 4!).



Odgovore na pitanja iz kviza potražite u sažetku poglavlja!

Ugnežđeni opsezi vidljivosti

Rekli smo da je opseg vidljivosti skup pravila za traženje promenljivih po imenima njihovih identifikatora. Međutim, obično ima više opsega vidljivosti koje treba pregledati.

Isto kao što je blok jedne funkcije ugnežđen unutar bloka druge funkcije, tako su i jedni opsezi vidljivosti ugnežđeni unutar drugih. Zbog toga, ako određenu promenljivu ne uspe da pronađe unutar tekućeg opsega vidljivosti, Mašina pretražuje prvi sledeći spoljašnji i tako nastavlja sve dok ne pronađe promenljivu ili dok ne dođe do spoljašnjeg opsega najvišeg nivoa (poznat i kao globalni opseg vidljivosti).

Razmotrite sledeće:

```
function foo(a) {
    console.log( a + b );
}

var b = 2;

foo( 2 ); // 4
```

RHS referenca `b` ne može se razrešiti unutar funkcije `foo`, ali može u opsegu koji okružuje tu funkciju (što je, u ovom slučaju, globalni opseg vidljivosti).

Zato kada bismo se vratili na razgovor između Mašine i Opsega vidljivosti, čuli bismo sledeće:

Mašina: „Zdravo, Opseže vidljivosti funkcije `foo`, da li si čuo za `b`? Imam jednu RHS referencu takve promenljive."

Opseg vidljivosti: „Ne, nikad čuo za to. Traži dalje."

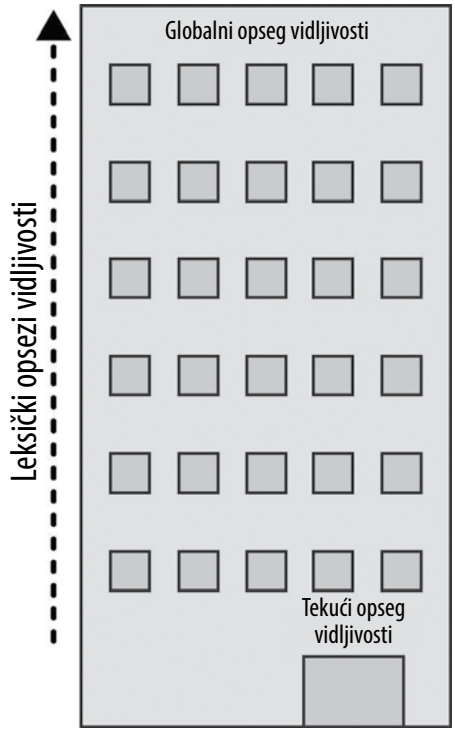
Mašina: „Zdravo, Opseže vidljivosti izvan funkcije `foo`; oh, to je globalni opseg vidljivosti, super. Da li si čuo za `b`? Imam RHS referencu takve promenljive."

Opseg vidljivosti: „Jesam, svakako. Evo, izvoli."

Jednostavna pravila za pretraživanje ugnežđenog opsega vidljivosti: Mašina prvo traži promenljivu unutar opsega vidljivosti tekuće funkcije koja se izvršava; ako je tamo ne nađe, prelazi na prvi sledeći okružujući nivo itd. Kada dođe do spoljašnjeg, globalnog opsega vidljivosti, pretraživanje se završava, bez obzira na to da li je promenljiva nađena ili nije.

Gradnja na metaforama

Da biste sebi vizuelno predstavili pretraživanja ugnežđenih opsega vidljivosti, zamislite sledeću visoku zgradu:



Zgrada predstavlja skup pravila koja važe za pretraživanje ugnežđenih opsega vidljivosti u našem programu. Tekući sprat zgrade, koji god to bio, predstavlja opseg vidljivosti funkcije koja se trenutno izvršava. Najviši sprat zgrade je globalni opseg vidljivosti.

LHS i RHS reference razrešavate tako što referencu prvo tražite na tekućem spratu, pa ako je tamo ne nađete, uzimate lift do sledećeg višeg sprata i tražite tamo, zatim na sledećem višem spratu itd. Kada stignete na najviši sprat (globalni opseg vidljivosti), tamo ili nalazite ili ne nalazite ono što ste tražili. U svakom slučaju, morate prekinuti pretragu.

Greške

Zbog čega je važno da li se obavlja LHS ili RHS pretraga?

Zato što se te dve vrste pretrage ponašaju različito u slučaju kada promenljiva nije deklarisana (nije pronađena ni u jednom pregledanom opsegu vidljivosti).

Razmotrite sledeće:

```
function foo(a) {
    console.log( a + b );
    b = a;
}

foo( 2 );
```

Kada se prvi put izvrši RHS pretraga za `b`, ta promenljiva neće biti pronađena. Tada se ta promenljiva smatra „nedeklarisanom“ jer nije pronađena u opsegu vidljivosti.

Ako RHS pretraga nigde ne pronađe traženu promenljivu, ni u jednom od ugnežđenih opsega vidljivosti, mašina jezika generiše grešku `ReferenceError`. Važno je imati u vidu da je tip te greške `ReferenceError`.

Nasuprot tome, ukoliko mašina obavlja LHS pretragu i dođe do najvišeg sprata (globalnog opsega vidljivosti) a da nije pronašla promenljivu, ako program ne radi u „striktnom režimu“[1], globalni opseg vidljivosti napraviće novu promenljivu s traženim imenom *u globalnom opsegu vidljivosti* i proslediti je Mašini.

„Ne, pre nije bilo takve promenljive, ali hteo sam da pomognem, pa sam ti napravio jednu“.

„Striktni režim“ koji je uveden u verziji ES5, ima više ponašanja različitih od onih u normalnom/opuštenom/lenjom režimu. Jedno od takvih ponašanja je da ne dozvoljava automatsko/implicitno generisanje globalnih promenljivih. U tom slučaju, ne bi bilo nijedne promenljive s globalnim opsegom vidljivosti koji bi LHS pretraga mogla da pronađe, a Mašina bi generisala grešku `ReferenceError`, slično kao pri RHS pretrazi.

Ako je promenljiva pronađena tokom RHS pretrage, ali vi pokušate da s njenom vrednošću uradite nešto što nije moguće – recimo, da izvršite kao funkciju promenljivu koja to nije, ili da referencirate svojstvo vrednosti koja je `null` ili `undefined` – onda Mašina generiše grešku drugačijeg tipa, koji se zove `TypeError`.

`ReferenceError` znači neuspešno pretraživanje opsega vidljivosti, dok `TypeError` znači da je pretraživanje opsega vidljivosti bilo uspešno, ali da ste nad rezultatom pretrage pokušali neku nedozvoljenu/nemoguću akciju.

Sažetak poglavlja

Opseg vidljivosti je skup pravila koja određuju gde i kako se može pronaći data promenljiva (identifikator). Razlog te pretrage može biti dodeljivanje vrednosti promenljivoj, što

1. Videti u MDN-u objašnjenje striktnog režima.

se zove LHS referenca (na levu stranu), ili učitavanje njene vrednosti, što se zove RHS referenca (na desnu stranu).

LHS reference su rezultat operacija dodeljivanja vrednosti. Dodeljivanje vrednosti promenljivoj unutar opsega može se dogoditi zahvaljujući upotrebi operatora `=` ili prosleđivanju (dodeljivanju) vrednosti parametrima funkcija pomoću argumenata.

Mašina JavaScripta prvo kompajlira kôd pre nego što ga izvrši, pri čemu deli iskaze programa, kao što je `var a = 2;`, u dva koraka:

1. Prvo izvršava deo `var a`, da bi deklarisala tu promenljivu u opsegu vidljivosti programa. To se odvija na početku, pre izvršavanja koda.
2. Zatim, izvršava deo `a = 2`, da bi pronašla promenljivu (LHS referencu) i dodelila joj vrednost, ako je nađe.

Obe vrste traženja reference, i LHS i RHS, započinju postupak traženja u opsegu vidljivosti tekuće funkcije u toku izvršavanja, a ukoliko je potrebno (tj. ako tamo ne nađu ono što traže), traže identifikator redom na svim sledećim nivoima ugnežđenih opsega vidljivosti, opseg (sprat) po opseg, sve dok ne dođu do globalnog nivoa (najvišeg sprata) i prekinu pretragu, a tamo ili nađu traženi identifikator ili ga ne nađu.

Ako tražena RHS referenca ne postoji, rezultat je greška tipa `ReferenceError`. Nepostojeća LHS referenca prouzrokuje automatsko generisanje implicitno deklarisane globalne promenljive s traženim imenom (ukoliko nije uključen striktni režim), ili grešku tipa `ReferenceError` (ako je uključen striktni režim).

Odgovori na pitanja iz kviza

```
function foo(a) {
    var b = a;
    return a + b;
}

var c = foo( 2 );
```

1. Otkrijte sve LHS pretrage (ima ih 3!).

   `c = ..;, a = 2 (implicitno dodeljivanje vrednosti parametrima) i b = ..`

2. Otkrijte sve RHS pretrage (ima ih 4!).

   `foo(2.., = a;, a .. i .. b`

POGLAVLJE 10

Sada this postaje jasnije!

U poglavlju 9 raščistili smo s raznim pogrešnim mišljenjima o identifikatoru `this` i saznali da je `this` veza koja se uspostavlja pri svakom pozivanju funkcije i zavisi isključivo od mesta pozivanja funkcije (načina na koji je ona pozvana).

Mesto pozivanja

Da bismo razumeli kako se `this` povezuje (tj. šta u datom trenutku predstavlja), moramo shvatiti *mesto pozivanja* (engl. *call-site*): to je mesto u kodu gde se funkcija poziva (to nije mesto gde je ona deklarisana). Moramo ispitati mesto pozivanja funkcije da bismo odgovorili na pitanje: šta *konkretno* `this` predstavlja?

Pronalaženje mesta pozivanja uglavnom se svodi na „pronaći gde se funkcija poziva“ ali nije uvek tako jednostavno, pošto neki modeli pisanja koda mogu da zamagle *stvarno* mesto pozivanja.

Važno je imati na umu *stablo pozivanja* (engl. *call-stack*) (to je stablo koje čine sve funkcije koje su bile pozvane pre nego što smo stigli do tekućeg trenutka izvršavanja). Mesto pozivanja koje nas zanima je *aktivno pre* izvršavanja tekuće funkcije.

Ovako bismo ilustrovali stablo pozivanja i mesto pozivanja:

```
function baz() {
    // pošto se stablo pozivanja sastoji od: `baz`
    // mesto pozivanja ove funkcije nalazi se u
    // globalnom opsegu vidljivosti

    console.log( "baz" );
    bar(); // <-- mesto pozivanja funkcije `bar`
}

function bar() {
    // pošto se stablo pozivanja sastoji od: `baz` -> `bar`
    // mesto pozivanja ove funkcije nalazi se u `baz`

    console.log( "bar" );
    foo(); // <-- mesto pozivanja funkcije `foo`
}
```

```
function foo() {
    // pošto se stablo pozivanja sastoji od: `baz` -> `bar` -> `foo`
    // mesto pozivanja ove funkcije nalazi se u `bar`

    console.log( "foo" );
}

baz(); // <-- mesto pozivanja funkcije `baz`
```

Kada analizirate kôd, imajte na umu da treba pronaći stvarno mesto pozivanja (na stablu pozivanja), zato što je to jedino što je bitno za povezivanje `this`.



Stablo pozivanja možete vizuelizovati u svom umu ako proučite redosled u lancu pozivanja funkcija, kao što smo to uradili u prethodnom primeru koda. Ali to je sporo i podložno greškama. Drugi način da vidite stablo pozivanja jeste da iskoristite alatku za otkrivanje grešaka u svom čitaču veba. Većina savremenih čitača veba namenjenih upotrebi na stonim računarima ima ugrađene razvojne alatke među kojima je i JS dibager. U prethodnom primeru koda, pomoću tih alatki možete postaviti prekidnu tačku u prvi red funkcije `foo()`, ili samo umetnite u prvi red iskaz `debugger;`. Kada izvršavate stranicu, dibager će se zaustaviti na tom mestu i prikazati listu svih funkcija koje su bile pozvane do tog reda, što će biti stablo pozivanja. Znači, ako pokušavate da otkrijete šta predstavlja `this`, upotrebite alatke za otkrivanje grešaka, a zatim pogledajte drugu stavku odozgo – tako ćete saznati stvarno mesto pozivanja koje ste tražili.

Samo po pravilu

Sada ćemo videti *kako* mesto pozivanja određuje na šta će `this` upućivati tokom izvršavanja funkcije.

Potrebno je da ispitate mesto pozivanja i utvrdite koje od četiri pravila važi. Prvo ćemo pojedinačno objasniti svako od ta četiri pravila, a onda ćemo ilustrovati njihov redosled prioriteta, u slučaju da bi se na dato mesto pozivanja *moglo* primeniti više pravila istovremeno.

Podrazumevano povezivanje

Prvo pravilo koje ćemo razmotriti potiče od najuobičajenijeg slučaja pozivanja funkcije: samostalni poziv funkcije. To pravilo za `this` možete zamisliti kao podrazumevano pravilo koje se primenjuje kada nijedno drugo pravilo nije primenljivo.

Razmotrite sledeći kôd:

```
function foo() {
     console.log( this.a );
}

var a = 2;

foo(); // 2
```

Prvo na šta treba da obratite pažnju, ako još niste, jeste to da su promenljive koje su deklarisane u globalnom opsegu vidljivosti, kao što je `var a = 2`, sinonimi za svojstva globalnog objekta istog imena. To nisu kopije jednih u druge – one *jesu* i jedno i drugo. Zamislite ih kao dve strane istog novčića.

Drugo, vidimo da kada se pozove funkcija `foo()`, u njoj se `this.a` razrešava u globalnu promenljivu `a`. Zašto? Zato što se u ovom slučaju, na mestu pozivanja funkcije primenjuje *podrazumevano povezivanje* za `this`, pa identifikator `this` predstavlja globalni objekat.

Kako znamo da se u ovom slučaju primenjuje pravilo *podrazumevanog povezivanja*? Ispitaćemo mesto pozivanja da bismo utvrdili kako se poziva funkcija `foo()`. U navedenom primeru, `foo()` se poziva u obliku obične i nedekorisane reference same funkcije. Pošto se u tom slučaju ne primenjuje nijedno drugo pravilo, važi *podrazumevano povezivanje*.

Ako je uključen striktni režim (tj. važi `strict mode`), globalni objekat nije upotrebljiv za *podrazumevano povezivanje*, pa zato `this` postaje objekat `undefined`:

```
function foo() {
    "use strict";

    console.log( this.a );
}

var a = 2;

foo(); // TypeError: `this` je `undefined`
```

Suptilan ali važan detalj: uprkos tome što se sva pravila za povezivanje `this` zasnivaju na mestu pozivanja funkcije, globalni objekat je upotrebljiv za *podrazumevano povezivanje* samo ako za izvršavanje sadržaja `foo()` nije uključen striktni režim; za samo mesto pozivanja funkcije `foo()`, nebitno je da li je striktni režim uključen ili nije:

```
function foo() {
    console.log( this.a );
}

var a = 2;

(function(){
    "use strict";

    foo(); // 2
})();
```



Namerno mešanje uključenog i isključenog striktnog režima u kodu uglavnom se ne preporučuje. Trebalo bi da striktni režim važi za ceo program ili ni za jedan njegov deo. Međutim, budući da ponekad u program uključujete biblioteku iz trećeg izvora koja radi u drugačijem režimu od vašeg koda, morate voditi računa i o tim suptilnim detaljima kompatibilnosti.

Implicitno povezivanje

Drugo pravilo koje treba uzeti u obzir jeste da li mesto pozivanja ima objekat konteksta, koji se naziva i vlasnički ili objekat okruženja, mada ti alternativni nazivi navode na donekle pogrešan zaključak.

Razmotrite sledeće:

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2,
    foo: foo
};

obj.foo(); // 2
```

Prvo, obratite pažnju na oblik u kojem je funkcija `foo()` deklarisana a zatim dodata kao referenca svojstvu objekta `obj`. Bez obzira na to da li je funkcija `foo()` inicijalno deklarisana *unutar* objekta `obj`, ili mu je naknadno dodata kao referenca (kao u navedenom primeru koda), ta funkcija nije ni u kom slučaju „vlasništvo“ objekta `obj`, niti je u njemu „sadržana“.

Međutim, pošto se na mestu pozivanja kontekst objekta `obj` *koristi* za referenciranje funkcije, u trenutku pozivanja funkcije *mogli biste* reći da je objekat `obj` „vlasnik“ reference funkcije, tj. da je „sadrži“.

Kako god da nazovete taj model, na mestu pozivanja funkcije `foo()` prethodi joj referenca na objekat `obj`. Kada se za referenciranje funkcije koristi kontekstni objekat, pravilo *implicitnog povezivanja* nalaže da se *taj* objekat povezuje sa `this` dok se funkcija izvršava. Pošto `obj` jeste `this` tokom izvršavanja funkcije `foo()`, `this.a` je sinonim za `obj.a`.

Na mestu pozivanja važan je samo najviši/poslednji nivo lanca referenciranja svojstava. Na primer:

```
function foo() {
    console.log( this.a );
}

var obj2 = {
    a: 42,
    foo: foo
};

var obj1 = {
    a: 2,
    obj2: obj2
};

obj1.obj2.foo(); // 42
```

Gubljenje implicitno uspostavljene veze

Jedan od najuobičajenih razloga za nerviranje čiji je izvor povezivanje identifikatora `this`, pojavljuje se kada neka *implicitno povezana* funkcija izgubi tu vezu, što obično znači automatsko prelaženje na *podrazumevano povezivanje* na globalni objekat ili objekat `undefined` – zavisno od toga da li je isključen striktni režim.

Razmotrite sledeće:

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2,
    foo: foo
};

var bar = obj.foo; // referenca/alijas funkcije!

var a = "uh, globalni"; // `a` je svojstvo globalnog objekta

bar(); // "uh, globalni"
```

Uprkos tome što izgleda kao da je `bar` referenca na `obj.foo`, to je zapravo samo još jedna referenca na samu funkciju `foo`. Osim toga, bitno je jedino šta se događa na mestu pozivanja, a tamo imamo `bar()`, što je običan i nedekorisan poziv funkcije, pa se zato primenjuje *podrazumevano povezivanje*.

Suptilniji, češći i neočekivaniji slučaj nastaje kada pozvanoj funkciji prosleđujemo povratnu funkciju:

```
function foo() {
    console.log( this.a );
}

function doFoo(fn) {
    // `fn` je samo još jedna referenca na `foo`

    fn(); // <-- mesto pozivanja!
}

var obj = {
    a: 2,
    foo: foo
};

var a = "uh, globalni"; // `a` je svojstvo globalnog objekta

doFoo( obj.foo ); // "uh, globalni"
```

Prosleđivanje parametara je samo implicitno dodeljivanje vrednosti, a pošto prosleđujemo funkciju, to je implicitno dodeljivanje reference, pa je zato konačni rezultat isti kao u prethodnom primeru.

Šta ako funkcija kojoj prosleđujete povratnu funkciju nije vaša, nego je ugrađena u jezik? Nema razlike, rezultat je isti:

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2,
    foo: foo
};

var a = "uh, globalni"; // `a` je svojstvo globalnog objekta

setTimeout( obj.foo, 100 ); // "uh, globalni"
```

Zamislite da je u JavaScript okruženje ugrađena sledeća gruba i teorijska pseudoimplementacija funkcije `setTimeout()`:

```
function setTimeout(fn,interval) {
    // sačekati (nekako) `interval` milisekundi
    // a onda pozvati funkciju fn
    fn(); // <-- mesto pozivanja!
}
```

Prilično je uobičajeno da funkcije koje prosleđujete kao povratne funkcije *izgube* povezivanje svog `this`, kao što smo upravo videli. Još jedan način na koji nas `this` može iznenaditi jeste kada funkcija kojoj smo prosledili provratnu funkciju namerno menja `this` u tom pozivu. Funkcije za obradu događaja u popularnim JavaScript bibliotekama prilično „vole“ da menjaju `this` tako da upućuje, na primer, na DOM element koji je izvor događaja. Mada to ponekad može biti korisno, u drugim slučajevima vas može baš razbesneti. Nažalost, te alatke vam retko kad pružaju mogućnost da sami birate.

U svakom slučaju, `this` je neočekivano izmenjen, a pošto vi nemate zaista kontrolu nad time kako će se izvršiti referenca na vašu povratnu funkciju, (još) nemate način da zadate njeno mesto pozivanja kako biste nametnuli povezivanje `this` u obliku koji vama odgovara. U nastavku teksta razmotrićemo način „rešavanja“ tog problema *ispravljanjem* `this`.

Eksplicitno povezivanje

Pri *implicitnom povezivanju*, kao što smo upravo videli, morali smo da izmenimo dati objekat tako da on funkciji prosleđuje referencu na sebe i da zatim referencu na svojstvo objekta upotrebimo za (implicitno) povezivanje `this` s tim objektom.

Ali, šta ako želimo da se – pri pozivanju funkcije – `this` poveže s konkretnim objektom, ali da pritom ne moramo zadavati i referencu na svojstvo tog objekta?

„Sve“ funkcije u jeziku stavljaju na raspolaganje određene metode (kroz svojstvo `[[Prototype]]` objekta funkcije – više informacija o tome naći ćete u nastavku teksta), što može biti korisno za ovu namenu. Konkretno, objekti funkcija imaju metode `call(..)` i `apply(..)`. Tehnički rečeno, radna okruženja JavaScripta ponekad stavljaju na raspolaganje funkcije koje su toliko specijalne (učtivo govoreći!) da ne nude tu funkcionalnost. Ali takve su malobrojne. Većina funkcija koje postoje – a svakako sve funkcije koje ćete vi napisati – imaju metode `call(..)` i `apply(..)`.

Kako te metode rade? Obe prihvataju kao svoj prvi parametar objekat s kojim treba povezati `this`, a zatim pozivaju funkciju s tim zadatim `this`. Budući da direktno zadajete šta želite da `this` bude, to zovemo *eksplicitno povezivanje*.

Razmotrite sledeće

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2
};

foo.call( obj ); // 2
```

Pozivanje funkcije `foo` sa *eksplicitnim povezivanjem* u obliku `foo.call(..)`, omogućava da nametnemo da `this` predstavlja objekat `obj`.

Ako umesto objekta prosledite prostu primitivnu vrednost (tipa `string`, `boolean` ili `number`) za povezivanje sa `this`, ta prosta vrednost konvertuje se u objektni oblik (`new String(..)`, `new Boolean(..)`, odnosno `new Number(..)`). To se često naziva „pakovanje“ (engl. *boxing*) vrednosti.



Obe metode – `call(..)` i `apply(..)` – povezuju `this` na isti način. One se *razlikuju* samo po dodatnim parametrima, ali je to zasad nebitno.

Nažalost, *eksplicitno povezivanje* – samo po sebi – još nam ne pruža nikakvo rešenje za ranije opisani problem funkcije koja „gubi“ već uspostavljeno povezivanje `this`, ili to povezivanje nameće neka spoljašnja biblioteka itd.

Čvrsto povezivanje

Postoji varijanta modela *eksplicitnog povezivanja* koja zapravo obavlja posao. Razmotrite sledeće:

```
function foo() {
    console.log( this.a );
}
```

```
var obj = {
    a: 2
};

var bar = function() {
    foo.call( obj );
 };

bar(); // 2
setTimeout( bar, 100 ); // 2

// čvrsto povezana `bar` više ne dozvoljava menjanje svog `this`
bar.call( window ); // 2
```

Razmotrimo kako radi ova varijanta. Deklarišemo funkciju `bar()` koja, interno i ručno poziva `foo.call(obj)`, što znači da se funkcija `foo` poziva tako da `this` predstavlja promenljivu `obj`. Bez obzira na to kako posle toga pozivate funkciju `bar`, ona će uvek ručno pozivati funkciju `foo` s promenljivom `obj`. Ta veza je i eksplicitna i čvrsta, pa je zato zovemo *čvrsto povezivanje* (engl. *hard binding*).

Najuobičajeniji način umotavanja funkcije u *čvrstu vezu* formira prolaz za sve argumente koje prosledite i povratnu vrednost koju dobijete:

```
function foo(something) {
    console.log( this.a, something );
    return this.a + something;
}

var obj = {
    a: 2
};

var bar = function() {
    return foo.apply( obj, arguments );
};

var b = bar( 3 ); // 2 3
console.log( b ); // 5
```

Drugi način da iskoristite ovaj model jeste da napravite višekratno upotrebljivu pomoćnu funkciju:

```
function foo(something) {
    console.log( this.a, something );
    return this.a + something;
}

// jednostavna pomoćna funkcija za povezivanje this
function bind(fn, obj) {
    return function() {
        return fn.apply( obj, arguments );
```

```
    };
}

var obj = {
    a: 2
};

var bar = bind( foo, obj );

var b = bar( 3 ); // 2 3
console.log( b ); // 5
```

Pošto je *čvrsto povezivanje* tako uobičajen model, u ES5 je na raspolaganju u obliku ugrađene metode, `Function.prototype.bind(..)`, koja se koristi na sledeći način:

```
function foo(something) {
    console.log( this.a, something );
    return this.a + something;
}

var obj = {
    a: 2
};

var bar = foo.bind( obj );

var b = bar( 3 ); // 2 3
console.log( b ); // 5
```

Metoda `bind(..)` vraća novu funkciju koja je generisana tako da poziva izvornu funkciju s kontekstom za `this` koji zadate.

Pozivanje API-ja s parametrom „context“

Funkcije iz mnogih biblioteka, kao i mnoge nove funkcije ugrađene u jezik JavaScript i radno okruženje, imaju neobavezan parametar, čije je ime najčešće „context“, a svrha mu je da vas poštedi upotrebe metode `bind(..)` kako biste obezbedili da vaša povratna funkcija koristi baš određeni `this`.

Na primer:

```
function foo(el) {
    console.log( el, this.id );
}

var obj = {
    id: "awesome"
};

// pri pozivanju `foo(..)` kao `this` koristi se `obj`
[1, 2, 3].forEach( foo, obj );
// 1 awesome  2 awesome  3 awesome
```

Interno, sve funkcije te vrste gotovo uvek koriste *eksplicitno povezivanje* pomoću `call(..)` ili `apply(..)` da vam uštede trud.

Povezivanje funkcija pomoću operatora new

Četvrto i poslednje pravilo za povezivanje `this` zahteva da se vratimo na veoma često pogrešno mišljenje u vezi sa funkcijama i objektima u JavaScriptu.

U tradicionalnim jezicima orijentisanim na klase, „konstruktori" su specijalne metode pridružene klasama. Kada pomoću operatora `new` pravite instancu klase, poziva se konstruktor te klase. To obično izgleda slično ovome:

```
nekaInstanca = new NekaKlasa(..);
```

JavaScript ima operator `new` i model njegove upotrebe u kodu izgleda isto kao u jezicima orijentisanim na klase; većina programera podrazumeva da JS mehanizam radi nešto slično. Međutim, delovanje operatora `new` u JavaScriptu zapravo *nema nikakve sličnosti* s funkcionalnošću kakvu nude jezici orijentisani na klase.

Prvo ćemo redefinisati šta je „konstruktor" u JavaScriptu. U JS-u, konstruktori su samo funkcije koje su pozvane tako da im prethodi operator `new`. One nisu vezane za klase, niti generišu instance klasa. Nisu čak ni specijalne vrste funkcija. To su samo obične funkcije koje, u suštini, rade malo drugačije zato što su pozvane s operatorom `new`.

Na primer, razmotrite funkciju `Number(..)` kada igra ulogu konstruktora, prema citatu iz specifikacija za ES5.1:

> 15.7.2 Konstruktor Number
>
> Kada se unutar izraza funkcija Number pozove s operatorom `new`, ona postaje konstruktor i inicijalizuje novonapravljeni objekat.

Dakle, manje-više svaka obična funkcija, uključujući i metode objekata, kao što je `Number(..)` (videti poglavlje 11), može se pozvati s operatorom `new` ispred, da bi taj poziv funkcije onda postao *konstruktorski poziv*. To je važna suptilna razlika: ne postoji ništa što bi bilo „konstruktorska funkcija" – postoje samo *konstruktorski pozivi funkcija*.

Kada funkciju pozivate s operatorom `new` ispred, što se inače zove konstruktorski poziv, automatski se obavlja sledeće:

1. Pravi se (konstruiše) potpuno nov objekat.
2. Novokonstruisani objekat se povezuje pomoću svog svojstva `[[Prototype]]`.
3. Novokonstruisani objekat se zadaje kao objekat koji `this` predstavlja u ovom pozivu funkcije.
4. Ako ne vraća vlastiti alternativni objekat, funkcija pozvana s operatorom `new` *automatski* vraća novokonstruisani objekat.

Koraci 1, 3 i 4 odnose se na naše tekuće razmatranje. Korak 2 ćemo zasad preskočiti i vratiti se na njega u poglavlju 13.

Razmotrite sledeći kôd:

```
function foo(a) {
    this.a = a;
}

var bar = new foo( 2 );
console.log( bar.a ); // 2
```

Pozivanjem funkcije `foo(..)` s operatorom `new` ispred nje, konstruisali smo nov objekat i zadali da je on `this` u ovom pozivu funkcije `foo(..)`. Dakle, upotreba `new` je poslednji način na koji se pri pozivanju funkcije može povezati identifikator `this`. To ćemo nazvati *povezivanje pomoću new*.

Uvek po redosledu

Sada znamo četiri pravila za povezivanje `this` pri pozivanju funkcija. Samo treba da pronađete mesto pozivanja i ispitate ga kako biste utvrdili koje se pravilo primenjuje. Ali, šta ako je na mestu pozivanja moguća primena više pravila istovremeno? Pošto mora postojati određeni redosled prioriteta tih pravila, sada ćemo ilustrovati redosled kojim se ona primenjuju.

Trebalo bi da bude jasno da je *podrazumevano povezivanje* pravilo s najnižim prioritetom od sva četiri. Zato ćemo ga preskočiti.

Šta ima viši prioritet, *implicitno povezivanje* ili *eksplicitno povezivanje*? Ispitajmo šta se događa:

```
function foo() {
    console.log( this.a );
}

var obj1 = {
    a: 2,
    foo: foo
};

var obj2 = {
    a: 3,
    foo: foo
};

obj1.foo(); // 2
obj2.foo(); // 3

obj1.foo.call( obj2 ); // 3
obj2.foo.call( obj1 ); // 2
foo();   // undefined
```

Dakle, *eksplicitno povezivanje* ima viši prioritet od *implicitnog povezivanja*, što znači da bi prvo trebalo da se zapitate da li je primenjeno *eksplicitno povezivanje* pre nego što ispitate da li je primenjeno *implicitno povezivanje*.

POGLAVLJE 17

Izvorni objekti

U poglavljima 15 i 16 pomenuli smo više ugrađenih objekata, koji se obično zovu „izvorni“ objekti (engl. *natives*), kao što su `String` i `Number`. Sada ćemo ih detaljno razmotriti.

Najčešće se koriste sledeći izvorni objekti:

- `String()`
- `Number()`
- `Boolean()`
- `Array()`
- `Object()`
- `Function()`
- `RegExp()`
- `Date()`
- `Error()`
- `Symbol()` – dodat u ES6!

Kao što vidite, ti izvorni objekti su zapravo ugrađene funkcije.

Ako u JS dolazite iz nekog jezika kao što je Java, JavaScriptov `String()` će vam izgledati kao konstruktor `String(..)` pomoću kojeg ste navikli da konstruišete vrednosti znakovnog tipa. Zato ćete brzo shvatiti da možete raditi ovakve stvari:

```
var s = new String( "Zdravo svete!" );

console.log( s.toString() ); // "Zdravo svete!"
```

Tačno je da se svaka od tih izvornih funkcija može koristiti kao izvorni konstruktor. Ali ono što se tako konstruiše može biti različito od onog što mislite:

```
var a = new String( "abc" );

typeof a;                            // "object" ... nije "String"

a instanceof String;                 // true

Object.prototype.toString.call( a ); // "[object String]"
```

Rezultat generisanja vrednosti pomoću konstruktora (`new String("abc")`) jeste objektni omotač u koji je upakovana primitivna vrednost (`"abc"`).

Važno je to da operator `typeof` pokazuje da ti objekti nemaju vlastite specijalne *tipove*, nego su zapravo samo podtipovi opšteg tipa `object`.

Postojanje objektnog omotača može se prepoznati i pomoću sledećeg iskaza:

```
console.log( a );
```

Rezultat ovog iskaza zavisi od vašeg čitača veba, budući da razvojne konzole mogu slobodno birati najprikladniji oblik u kojem serijalizuju objekat da bi ga prikazale programeru.



U vreme pisanja ove knjige, najnovija verzija čitača Chrome prikazuje nešto slično sledećem: `String {0: "a", 1: "b", 2: "c", length: 3, [[PrimitiveValue]]: "abc"}`. Starije verzije Chromea prikazale bi samo nešto nalik sledećem: `String {0: "a", 1: "b", 2: "c"}`. Poslednja verzija Firefoxa sada prikazuje `String ["a","b","c"]`, ali je ranije prikazivala `"abc"` kurzivnom slovima, što je na pritisak miša otvaralo inspektor objekta. Razume se, opisani rezultati su vrlo podložni promenama, a vaše lično iskustvo može biti drugačije.

Suština je to da iskaz `new String("abc")` konstruiše omotački objekat u koji je upakovan znakovni niz `"abc"`, a ne samu primitivnu vrednost `"abc"`.

Interno svojstvo [[Class]]

Vrednosti za koje operator `typeof` pokazuje `"object"` (kao što je niz) imaju dodatno interno svojstvo `[[Class]]` (koje treba posmatrati više kao internu *klas*ifikaciju nego kao nešto što ima veze s klasama u smislu tradicionalnog koda orijentisanog na klase). Tom svojstvu se ne može pristupati direktno, ali se njegova vrednost može utvrditi indirektno, pozivanjem metode `Object.prototype.toString(..)` za samu vrednost. Na primer:

```
Object.prototype.toString.call( [1,2,3] );
// "[object Array]"

Object.prototype.toString.call( /regex-literal/i );
// "[object RegExp]"
```

Iz tog razloga, za niz u navedenom primeru, vrednost internog svojstva `[[Class]]` je `"Array"`, a za regularni izraz, to je `"RegExp"`. U većini slučajeva, vrednost tog internog svojstva `[[Class]]` odgovara ugrađenom izvornom konstruktoru (videti u nastavku teksta) koji se odnosi na vrednost – ali nije uvek tako.

Šta je s primitivnim vrednostima? Prvo, `null` i `undefined`:

```
Object.prototype.toString.call( null );
// "[object Null]"

Object.prototype.toString.call( undefined );
// "[object Undefined]"
```

Obratite pažnju na to da postoje izvorni konstruktori `Null()` ili `Undefined()`, ali uprkos tome, vrednosti internog svojstva `[[Class]]` koje se prikazuju jesu `"Null"` i `"Undefined"`.

U slučaju drugih prostih primitiva – kao što su `string`, `number` i `boolean` – imamo zapravo drugačije ponašanje, koje se najčešće naziva „pakovanje“ (videti odeljak „Pakovanje primitiva u objektne omotače“ na strani 42):

```
Object.prototype.toString.call( "abc" );
// "[object String]"

Object.prototype.toString.call( 42 );
// "[object Number]"

Object.prototype.toString.call( true );
// "[object Boolean]"
```

U ovom primeru, svaka od tih prostih primitiva automatski se umeće u odgovarajući objektni omotač, što je razlog zbog kojeg se kao vrednosti njihovog internog svojstva `[[Class]]` prikazuju `"String"`, `"Number"` i `"Boolean"`.



Ponašanja metode `toString()` i svojstva `[[Class]]` koja su ovde opisana, malo su se promenila od verzije ES5 do ES6, ali ćemo te detalje razmotriti u šestom delu ove knjige – *ES6 i sledeće verzije*.

Pakovanje primitiva u objektne omotače

Objektni omotači služe vrlo važnoj svrsi. Pošto primitivne vrednosti nemaju ni svojstva ni metode, da biste upotrebili metodu `.length` ili `.toString()` treba vam objektni omotač oko primitivne vrednosti. Srećom, JS će automatski *upakovati* (engl. *box*) (tj. umotati) primitivnu vrednost da bi omogućio takve pristupe:

```
var a = "abc";

a.length; // 3
a.toUpperCase(); // "ABC"
```

Dakle, ako nameravate da u svojim znakovnim nizovima često pristupate tim svojstvima/metodama – kao što je, na primer, uslov `i < a.length` u petlji `for` – možda bi vam izgledalo logično da od samog početka radite isključivo s objektnim oblikom vrednosti, pa zato nema potrebe da ga JS mašina implicitno pravi za vas.

Ali, ispostavlja se da je to loša ideja. Performanse čitača veba odavno su optimizovane za vrlo česte slučajeve kao što je `.length`, što znači da će vaš program *zapravo raditi sporije* ako pokušate da ga „unapred optimizujete“ tako što direktno koristite objektni oblik (koji se ne nalazi na optimizovanoj putanji).

Uglavnom, nema pravog razloga da se direktno koristi objektni oblik. Bolje je da prepustite mašini jezika da implicitno pakuje vrednosti na mestima gde je to neophodno. Drugim rečima, nemojte nikad raditi stvari kao što su `new String("abc")`, `new Number(42)` itd. – uvek radije radite s literalnim primitivnim vrednostima `"abc"` i `42`.

Zamke upotrebe objektnih omotača

Postoje čak i zamke koje se pojavljuju pri direktnoj upotrebi omotača, o kojima bi trebalo da vodite računa ako se ipak *namerno* opredelite za njihovu upotrebu.

Na primer, razmotrite sledeće upakovane vrednosti tipa `Boolean`:

```
var a = new Boolean( false );

if (!a) {
    console.log("Ups" ); // ovo se nikada ne izvršava
}
```

Problem je to što smo vrednost `false` upakovali u objekat, ali pošto se svi objekti po svojoj prirodi ponašaju kao `true` (videti poglavlje 18), ispitivanje objekta daje suprotan rezultat od ispitivanja njegove interne vrednosti `false`, što je sasvim suprotno ponašanje od očekivanog.

Ako želite da ručno upakujete primitivnu vrednost, upotrebite funkciju `Object(..)` (a ne rezervisanu reč `new`):

```
var a = "abc";
var b = new String( a );
var c = Object( a );

typeof a; // "string"
typeof b; // "object"
typeof c; // "object"

b instanceof String; // true
c instanceof String; // true

Object.prototype.toString.call( b ); // "[object String]"
Object.prototype.toString.call( c ); // "[object String]"
```

Ponavljam, direktna upotreba objektnog omotača umesto same vrednosti (kao `b` i `c` u navedenom primeru) uglavnom se ne preporučuje, ali može se dogoditi da naiđete na određene retke slučajeve gde to može biti korisno.

Raspakivanje

Ako imate objektni omotač i želite da iz njega izdvojite njegovu internu primitivnu vrednost, upotrebite metodu `valueOf()`:

```
var a = new String( "abc" );
var b = new Number( 42 );
var c = new Boolean( true );

a.valueOf(); // "abc"
b.valueOf(); // 42
c.valueOf(); // true
```

Raspakivanje se može obaviti i implicitno, ako vrednost unutar objektnog omotača upotrebite na način koji zahteva primitivnu vrednost. Taj postupak (konverzija tipa) detaljnije je opisan u poglavlju 18, ali ukratko rečeno:

```
var a = new String( "abc" );
var b = a + ""; // `b` sadrži raspakovanu primitivnu vrednost "abc"

typeof a;       // "object"
typeof b;       // "string"
```

Izvorne funkcije kao konstruktori

Za vrednosti koje su tipa `array`, `object`, `function` ili regularni izrazi, gotovo je uvek bolje rešenje da ih konstruišete u literalnom obliku, budući da u oba slučaja dobijate objekat, a ne skalarnu vrednost.

Kao što smo već videli u prethodnom delu poglavlja u slučaju drugih izvornih funkcija, te konstruktorske oblike treba uglavnom izbegavati – osim ako ste zaista sigurni da vam trebaju – prvenstveno zato što uvode izuzetke i zamke s kojima verovatno *ne želite* da imate posla.

Konstruktor Array(..)

```
var a = new Array( 1, 2, 3 );
a; // [1, 2, 3]

var b = [1, 2, 3];
b; // [1, 2, 3]
```



Ispred konstruktora `Array(..)` ne morate zadati rezervisanu reč `new`. Ako je izostavite, ponašaće se isto kao da ste je zadali. Zato `Array(1,2,3)` daje isti rezultat kao `new Array(1,2,3)`.

Konstruktor `Array` ima specijalan oblik u kojem ako mu prosledite samo jedan argument tipa `number`, umesto da tu vrednost obrađuje kao *sadržaj* niza, tumači je kao veličinu za koju „unapred priprema“ niz (ili tako nekako).

To je vrlo loša ideja. Prvo, taj oblik možete zadati nenamerno jer se na to lako zaboravlja.

Što je još važnije, ne postoji način da unapred pripremite niz zadate veličine. Umesto toga, ono što zaista dobijate je prazan niz, ali čije svojstvo `length` ima numeričku vrednost koju ste zadali.

Niz čiji elementi nemaju nikakve eksplicitne vrednosti, ali čije je svojstvo `length` takvo da *navodi na pomisao* da ti elementi postoje, jeste vrsta čudne i egzotične strukture za podatke u JavaScriptu, koja se ponaša vrlo neobično i zbunjujuće. Mogućnost stvaranja takvih vrednosti potiče isključivo od danas zastarelih i istorijskih funkcionalnosti („objekti nalik nizovima“ kao što je objekat `arguments`).



Niz koji sadrži barem jedan „prazan element“ često se naziva „proređen niz“.

Situaciju ne poboljšava to što je ovo još jedan primer gde razvojne konzole u čitaču veba mogu slobodno određivati oblik u kojem će predstaviti takav objekat, što dodatno zbunjuje.

Na primer:

```
var a = new Array( 3 );

a.length; // 3
a;
```

Oblik u kojem se `a` serijalizuje u čitaču Chrome (u vreme pisanja ove knjige) izgleda ovako: `[ undefined x 3 ]`. To je zaista nesrećan izbor jer podrazumeva da taj niz ima tri elementa čije su vrednosti `undefined`, dok u stvari ti elementi ni ne postoje (takozvani „prazan niz“ – još jedno loše ime!).

Da biste shvatili razliku, pokušajte sledeće:

```
var a = new Array( 3 );
var b = [ undefined, undefined, undefined ];
var c = [];
c.length = 3;

a;
b;
c;
```



Kao što vidite u slučaju promenljive `c` u ovom primeru, nakon definisanja niza mogu se pojaviti prazni elementi. Kada svojstvo `length` niza izmenite tako da ne pokazuje tačan broj vrednosti u zaista definisanim elementima, implicitno uvodite prazne elemente. U stvari, u prethodnom primeru možete pozvati čak i metodu `delete b[1]`, što uvodi prazan element usred niza `b`.

Promenljiva `b` (zasad u čitaču Chrome) serijalizuje se u oblik `[ undefined, undefined, undefined ]`, za razliku od oblika `[ undefined x 3 ]` za `a` i `c`. Zbunjeni? Da, kao što bi i svako bio.

Što je još gore, u vreme pisanja ove knjige, Firefox prikazuje `[ , , , ]` za `a` i `c`. Da li ste shvatili šta toliko zbunjuje? Pogledajte pažljivije. Tri zareza znače da u nizu postoje četiri elementa, a ne tri kao što bismo očekivali.

Otkud sad to!? Firefox dodaje na kraj svog oblika serijalizovanja jedan zarez jer su – od specifikacije za ES5 – u listama (vrednosti u nizu, svojstava itd.) dozvoljeni završni zarezi (pa se zato izostavljaju i zanemaruju). Znači, ako u svom programu ili na konzoli upišete vrednost `[ , , , ]`, dobićete zapravo vrednost koja je `[ , , ]` (tj. niz s tri prazna elementa). Takvo rešenje – mada zbunjuje ako čitate sadržaj razvojne konzole – brani se time da čini tačnim ponašanje pri kopiranju i umetanju.

Ako u ovom trenutku odmahujete glavom i prevrćete očima, niste jedini! Ali tako stoje stvari.



Izgleda kao da u ovakvim slučajevima Firefox menja svoj rezultat u `Array [ <3 empty slots> ]`, što je svakako veliki korak napred u poređenju sa `[ , , , ]`.

Nažalost, postaje sve gore. Gore od toga da se samo prikazuju u zbunjujućem obliku na razvojnoj konzoli, `a` i `b` iz prethodnog primera koda zapravo se u nekim slučajevima ponašaju isto, a u drugim slučajevima različito:

```
a.join( "-" ); // "--"
b.join( "-" ); // "--"

a.map(function(v,i){ return i; }); // [ undefined x 3 ]
b.map(function(v,i){ return i; }); // [ 0, 1, 2 ]
```

Ajoj.

Metoda `a.map(..)` *greši* iz sledećeg razloga: pošto elementi niza zapravo ne postoje, metoda `map(..)` nema ništa što bi mogla da obrađuje u iteraciji. Metoda `join(..)` radi drugačije. Možemo je zamisliti kao da je implementirana u sledećem obliku:

```
function fakeJoin(arr,connector) {
    var str = "";
    for (var i = 0; i < arr.length; i++) {
        if (i > 0) {
            str += connector;
        }
        if (arr[i] !== undefined) {
            str += arr[i];
        }
    }
    return str;
}

var a = new Array( 3 );
fakeJoin( a, "-" ); // "--"
```

Kao što vidite, metoda `join(..)` radi tako što *pretpostavlja* da elementi postoje i obrađuje ih u petlji koja se izvršava `length` puta. Evo šta metoda `map(..)` interno radi: pošto (verovatno) ne polazi od iste pretpostavke, njen rezultat za čudan niz s „praznim elementima“ neočekivan je i verovatno će prouzrokovati greške.

Dakle, ako *zaista* želite da napravite niz čiji su elementi vrednosti `undefined` (a ne „prazni elementi“), kako biste to uradili (osim ručno)?

```
var a = Array.apply( null, { length: 3 } );
a; // [ undefined, undefined, undefined ]
```

Zbunjeni? Da. Evo kako to otprilike radi.

Metoda `apply(..)` na raspolaganju je u svim funkcijama i poziva funkciju za koju je upotrebite, ali na poseban način.

Njen prvi argument je objekat na koji upućuje `this` (opisan u trećem delu ove knjige – *Identifikator this i prototipovi objekata*), za koji, pošto nam je u ovom slučaju nebitan, zadajemo `null`. Trebalo bi da drugi argument bude niz (ili nešto što se *ponaša kao niz* – tj. „objekat nalik nizu“). Elementi tog „niza“ raspodeljuju se kao argumenti funkcije koja se poziva.

Dakle, `Array.apply(..)` poziva funkciju `Array(..)` kojoj kao argumente prosleđuje raspodeljene vrednosti (iz objektne vrednosti `{ length: 3 }`).

Unutar metode `apply(..)`, možemo pretpostaviti da postoji još jedna petlja `for` (otprilike kao u metodi `join(..)` iz prethodnog dela) koja se izvršava od `0` do – ali ne i uključujući – `length` puta (`3` u našem primeru).

Za svaki indeks, metoda učitava ključ iz objekta. Tako da ako objekat koji prosledite na mestu parametra za niz ima unutar funkcije `apply(..)` interno ime `arr`, svojstvima tog objekta pristupaće se u obliku `arr[0]`, `arr[1]` i `arr[2]`. Razume se, pošto nijedno od tih svojstava ne postoji u objektu `{ length: 3 }`, svako od ta tri pristupanja svojstvu vraća vrednost `undefined`.

Drugim rečima, funkcija `Array(..)` u suštini se poziva u sledećem obliku: `Array(undefined,undefined,undefined)`, što je razlog zbog kojeg dobijamo niz popunjen vrednostima `undefined`, a ne samo one (nezgodne) prazne elemente.

Iako je `Array.apply( null, { length: 3 } )` neobičan i opširan način za izradu niza čiji su elementi vrednosti `undefined`, to je *znatno* bolji i pouzdaniji način od pucanja sebi u nogu s praznim elementima zbog `Array(3)`.

Zaključak: *nikad, ni u kom slučaju*, nemojte namerno praviti ni koristiti egzotične nizove s praznim elementima. To se ne radi. Oni su zlo.

Konstruktori Object(..), Function(..) i RegExp(..)

Konstruktori `Object(..)`/`Function(..)`/`RegExp(..)` najčešće su neobavezni (pa ih stoga treba uglavnom izbegavati, osim kada su zaista neophodni):

```
var c = new Object();
c.foo = "bar";
c; // { foo: "bar" }

var d = { foo: "bar" };
d; // { foo: "bar" }

var e = new Function( "a", "return a * 2;" );
var f = function(a) { return a * 2; }
function g(a) { return a * 2; }

var h = new RegExp( "^a*b+", "g" );
var i = /^a*b+/g;
```

Praktično ne postoji razlog za upotrebu konstruktorskog oblika `new Object()`, a naročito zato što vas to primorava da svojstva ručno dodajete objektu jedno po jedno, umesto da ih sve zadate u jednom iskazu kao u literalnom obliku definicije objekta.

Konstruktor `Function` je koristan samo u izuzetno retkom slučaju kada parametre ili telo funkcije treba da definišete dinamički. Nemojte posmatrati `Function(..)` samo kao alternativni oblik za `eval(..)`. Gotovo nikad vam neće trebati da funkcije definišete dinamički na taj način.

Snažno se preporučuje da se regularni izrazi definišu u literalnom obliku (`/^a*b+/g`) – ne samo zbog jednostavnije sintakse, nego i zbog boljih performansi: JS mašina ih unapred kompajlira i smešta u keš, pre izvršavanja koda. Za razliku od drugih konstruktora koje smo dosad razmatrali, `RegExp(..)` pruža određenu razumnu korist: za dinamičko definisanje šablona regularnog izraza:

```
var name = "Kyle";
var namePattern = new RegExp( "\\b(?:" + name + ")+\\b", "ig" );

var matches = someText.match( namePattern );
```

Pošto se ova vrsta slučaja sasvim normalno ponekad pojavljuje u JS programima, moraćete da upotrebite oblik `new RegExp("pattern","flags")`.

Konstruktori Date(..) i Error(..)

Izvorni konstruktori `Date(..)` i `Error(..)` znatno su korisniji od drugih izvornih konstruktora, zato što ni za jedan od tih objekata ne postoji literalni oblik.

Da biste konstruisali objektnu vrednost koja predstavlja datum, morate upotrebiti oblik `new Date()`. Konstruktor `Date(..)` prihvata opcione argumente pomoću kojih se zadaju datum/vreme, ali ako ih izostavite, uzima se tekući datum/vreme.

Najčešći razlog da konstruišete datumski objekat jeste da dobijete tekuću vrednost Unix vremenske oznake (što je celobrjna vrednost koja predstavlja broj sekundi protekao od 1. januara 1970. godine). To možete postići pomoću metode `getTime()` instance datumskog objekta.

Još lakši način je da samo pozovete statičku pomoćnu funkciju definisanu u ES5: `Date.now()`. Polifil za okruženja starija od ES5 vrlo je jednostavan:

```
if (!Date.now) {
    Date.now = function(){
        return (new Date()).getTime();
    };
}
```



Ako konstruktor `Date()` pozovete bez `new`, dobijate tekstualni oblik tekućeg datuma/vremena. Tačan oblik tog podatka nije zadat u specifikacijama jezika, mada se mnogi čitači veba slažu oko nečega sličnog sledećem: `"Fri Jul 18 2014 00:31:02 GMT-0500 (CDT)"`.

Konstruktor `Error(..)` (vrlo slično prethodnom konstruktoru `Array()`) ponaša se isto, bez obzira na to da li uz njega zadate rezervisanu reč `new`.

Glavni razlog zbog kojeg biste napravili objekat koji predstavlja grešku jeste da biste u taj objekat preuzeli sadržaj tekućeg stabla (steka) pozivanja funkcija (koje je u većini JS mašina na raspolaganju kao vrednost svojstva samo za čitanje `.stack` nakon konstruisanja objekta). Taj kontekst stabla pozivanja uključuje stablo pozivanja funkcija i redni broj reda u kojem je konstruisan objekat greške, što znatno olakšava otkrivanje i otklanjanje grešaka.

Uobičajeno je da se takav objekat greške pravi pomoću operatora `throw`:

```
function foo(x) {
    if (!x) {
        throw new Error( "x nije zadato" );
    }
    // ..
}
```

Instance objekata grešaka uglavnom imaju barem svojstvo `message`, a ponekad i druga svojstva (koja bi trebalo da tumačite kao samo za čitanje), kao što je `type`. Međutim, osim ispitivanja pomenutog svojstva `stack`, uglavnom je najbolje da pozovete metodu `toString()` objekta greške (eksplicitno, ili implicitno kroz konverziju tipa – videti poglavlje 18) da biste dobili čitljivije formatiranu poruku o grešci.



Tehnički govoreći, osim opšteg izvornog konstruktora `Error(..)`, postoji i više drugih izvornih konstruktora koji su uže specijalizovani za pojedine vrste grešaka: `EvalError(..)`, `RangeError(..)`, `ReferenceError(..)`, `SyntaxError(..)`, `TypeError(..)` i `URIError(..)`. Ali se ti konstruktori za specifične greške vrlo retko ručno koriste. Oni se automatski pozivaju ako se u programu javi stvarni izuzetak (kao što je referenciranje nedeklarisane promenljive, što izaziva grešku `ReferenceError`).

Konstruktor Symbol(..)

U verziji ES6 uvedena je novina nazvana „Symbol“. Vrednosti tipa Symbol su specijalne „jedinstvene“ (što nije striktno garantovano!) vrednosti koje se mogu koristiti kao imena svojstava objekata bez bojazni od dupliranja. One su prvenstveno namenjene za specijalna ugrađena ponašanja ES6 konstrukata, ali možete definisati i vlastite simbole.

Simboli se mogu koristiti za imena svojstava, ali konkretno ime simbola ne možete videti, niti mu pristupati u programu, a ni iz razvojne konzole. Ako pokušate da prikažete ime simbola u razvojnoj konzoli, prikazaće se, na primer, `Symbol(Symbol.create)` – ili nešto slično.

U ES6 postoji više unapred definisanih simbola, kojima se pristupa kao statičkim svojstvima funkcijskog objekta `Symbol`, kao što su `Symbol.create`, `Symbol.iterator` itd. Da biste ih upotrebili, zadajte nešto poput:

```
obj[Symbol.iterator] = function(){ /*..*/ };
```

POGLAVLJE 20

Asinhronizam: pojmovi sada i kasnije

Jedan od najvažnijih, ali uprkos tome često slabo shvaćenih delova programskog jezika kao što je JavaScript jeste kako izraziti ponašanje programa koje je promenljivo tokom vremena.

To se ne svodi samo na pitanje šta se događa od početka petlje `for` do kraja petlje `for`, kojoj je, razume se, potrebno izvesno vreme (više mikrosekundi ili milisekundi) da bi se izvršila. To je pitanje šta se događa kada se deo programa izvršava *sada*, a neki njegov drugi deo se izvršava *kasnije* – tokom perioda između *sada* i *kasnije*, kada program zapravo nije aktivan.

Praktično svaki netrivijalan program koji je dosad napisan (naročito na JS-u) morao je da na jedan ili drugi način upravlja tim intervalom, bez obzira na to da li čeka dok korisnik unosi ulazne podatke, ili čeka podatke iz baze podataka ili sistema datoteka, šalje podatke kroz mrežu i čeka odziv iz nje, ili obavlja posao koji se ponavlja u jednakim vremenskim intervalima (kao što je animacija). U svim tim različitim slučajevima, program mora da upravlja svojim stanjem kroz periode neaktivnosti. Kao što kaže poznata preporuka iz londonskog metroa (u vezi s rastojanjem između vrata vagona i ivice perona): „vodite računa o razmaku“.

U stvari, taj odnos između *sada* i *kasnije* delova programa čini srž asinhronog programiranja.

Nesporno je da mogućnost asinhronog programiranja postoji još od samih početaka JavaScripta. Ipak, većina JS programera nije nikad zaista pažljivo razmatrala kako i gde se ono pojavljuje u njihovim programima, niti istraživala razne druge načine da ga primeni. *Dovoljno dobro* rešenje uvek je bila skromna povratna funkcija. Mnogi i danas tvrde da su povratne funkcije više nego dovoljne.

Ali kako JS nastavlja da raste i širi se i po obimu i po složenosti kako bi zadovoljio neprekidno rastuće zahteve za prvoklasnim jezikom koji radi i u čitačima veba, i na serverima i na svakom zamislivom uređaju između njih, napori koje ulažemo da bismo upravljali asinhronizmom softvera postaju sve teži i zahtevaju nalaženje rešenja koja su istovremeno i razumnija i pružaju više mogućnosti.

Mada vam sve ovo možda zasad izgleda prilično apstraktno, znajte da ćemo se time baviti detaljnije i konkretnije u celom ovom delu knjige. U narednih nekoliko poglavlja proučićemo više novih tehnika koje tek počinju da se primenjuju za asinhrono JavaScript programiranje.

Pre nego što pređemo na to, moramo znatno detaljnije razmotriti šta je asinhroni rad i kako se on postiže u JavaScriptu.

Program u delovima

Ceo svoj JS program možete napisati u jednoj *.js* datoteci, ali je on gotovo uvek sastavljen od više delova, od kojih će se samo jedan izvršiti *sada*, dok će se preostali delovi izvršiti *kasnije*. Najuobičajenija jedinica za svaki *delić* programa jeste *funkcija* (`function`).

Problem koji većina JS programera naizgled ima jeste to što se *kasnije* ne događa neposredno i odmah nakon *sada*. Drugim rečima, poslovi koji se ne mogu završiti *sada* završiće se – po definiciji – asinhrono, pa zato neće imati blokirajuće ponašanje kao što biste intuitivno očekivali ili želeli.

Razmotrite sledeće:

```
// ajax(..) je neka funkcija iz biblioteke
var data = ajax( "http://some.url.1" );

console.log( data );
// Uh! `data` najčešće neće sadržati Ajax rezultate
```

Verovatno znate da se standardni Ajax zahtevi ne izvršavaju sinhrono, što znači da funkcija `ajax(..)` još nema nikakvu vrednost da vrati kako bismo je dodelili promenljivoj `data`. Kada bi funkcija `ajax(..)` mogla da blokira izvršavanje programa dok ne stigne odziv, onda bi se operacija dodeljivanja `data = ..` izvršila kako treba.

Međutim, Ajax ne radi tako. Asinhroni Ajax zahtev šaljemo *sada*, ali ćemo njegove rezultate dobiti *kasnije*.

Najjednostavniji (ali svakako ne i jedini, niti obavezno najbolji!) način da „sačekamo“ od *sada* do *kasnije* jeste da upotrebimo funkciju, poznatu pod nazivom *povratna funkcija* (engl. *callback function*):

```
// ajax(..) je neka funkcija iz biblioteke
ajax( "http://some.url.1", function myCallbackFunction(data){

    console.log( data ); // Ura, sad imamo nešto u `data`!

} );
```



Možda ste čuli da je moguće slati sinhrone Ajax zahteve. Mada je to tehnički izvodljivo, nemojte to nikad i ni u kom slučaju raditi, zato što tako blokirate korisnički interfejs čitača veba (dugmad, menije, pomeranje sadržaja prozora itd.) i sprečavate svaku interakciju s korisnikom. To je užasno loša ideja i trebalo bi da je uvek izbegavate.

Pre nego što počnete da negodujete jer se ne slažete s time, reći ćemo da vaša želja da izbegnete zbrku s povratnim funkcijama ne opravdava upotrebu blokirajućeg, sinhronog Ajaxa.

Na primer, razmotrite sledeći kôd:

```
function now() {
    return 21;
}

function later() {
    answer = answer * 2;
    console.log( "Smisao života je:", answer );
}

var answer = now();

setTimeout( later, 1000 ); // Smisao života je: 42
```

Ovaj program se sastoji od dva dela: dela koji se izvršava *sada* i dela koji će se izvršiti *kasnije*. Trebalo bi da bude prilično jasno koji su to delovi, ali bićemo sasvim eksplicitni:

Sada:

```
function now() {
    return 21;
}

function later() { .. }

var answer = now();

setTimeout( later, 1000 );
```

Kasnije:

```
answer = answer * 2;
console.log( "Smisao života je:", answer );
```

Deo programa *sada* izvršava se odmah, čim pokrenete izvršavanje programa. Ali pošto funkcija `setTimeout(..)` definiše događaj (isticanje vremenskog intervala) koji će se odigrati *kasnije*, sadržaj funkcije `later()` izvršiće se u nekom budućem trenutku (1000 milisekundi od sada).

Kad god deo koda upakujete u funkciju i zadate da se ona izvrši kao odziv na određeni događaj (isticanje vremenskog intervala, pritiskanje tastera miša, prijem Ajax odziva itd.), tako formirate element koji se izvršava *kasnije* i time u program uvodite asinhronizam.

Asinhronizam konzole

Ne postoji specifikacija niti skup zahteva koji bi propisivali kako treba da rade metode `console.*` – zato što one zvanično nisu sastavni deo JavaScripta, nego ih u JS dodaje *radno okruženje* (videti četvrti deo ove knjige – *Tipovi i gramatika*).

Iz tog razloga, različiti čitači veba i JS okruženja rade kako njima odgovara, što ponekad dovodi do zbunjujućeg ponašanja.

Konkretno, postoje čitači veba u kojima – pod određenim uslovima – funkcija `console.log(..)` ne prikazuje odmah ono što joj prosledite. Glavni razlog zbog kojeg se to može dogoditi jeste to što su ulazno/izlazne operacije veoma spore i blokiraju delove mnogih programa (ne samo sâm JS). Zato čitač veba može pružiti bolje performanse (ulazno/izlaznih operacija na veb stranici) ako funkcija `console` U/I zahteve obrađuje asinhrono u pozadini, a da vi možda niste ni svesni da se to događa.

Ne mnogo uobičajen, ali moguć slučaj gde bi to moglo da bude *vidljivo* (ne iz samog koda, nego od spolja):

```
var a = {
    index: 1
};

// kasnije
console.log( a ); // ??

// još kasnije
a.index++;
```

Standardno bismo očekivali da vidimo kako se stanje objekta `a` prikazuje onakvo kakvo je bilo tačno u trenutku izvršavanja iskaza `console.log(..)`, tj. da se ispiše nešto poput `{ index: 1 }`, tako da – kad se izvrši sledeći iskaz `a.index++` – on menja stanje kakvo je bilo neposredno nakon prikazivanja stanja `a`.

Prethodni primer koda predstavljaće objekat na vašoj konzoli najčešće baš onako kako biste očekivali. Ali, može se dogoditi da isti kôd radi u situaciji gde čitač veba smatra da ulazno/izlazne operacije na konzoli treba da izvršava u pozadini, kada se može dogoditi da je – u trenutku kad se objekat prikazuje na konzoli čitača veba – iskaz `a.index++` već bio izvršen, pa zato konzola prikazuje `{ index: 2 }`.

Teško je tačno odrediti pod kojim će uslovima U/I operacije na konzoli biti odložene, pa čak i to da li će to biti uočljivo. Imajte samo u vidu taj mogući U/I asinhronizam u slučaju da pri otkrivanju grešaka imate problema s objektima koji se u programu menjaju *nakon* izvršavanja iskaza `console.log(..)`, ali se uprkos tome na konzoli vide neočekivane naknadne izmene.



Ako vam se dogodi takav redak slučaj, najbolje rešenje je da u svom JS dibageru zadate prekidne tačke umesto da podatke prikazujete pomoću iskaza `console`. Drugo najbolje rešenje je da napravite „snimak“ datog objekta tako što ga serijalizujete u promenljivu tipa `string` – na primer, pomoću metode `JSON.stringify(..)`.

Petlja za obradu događaja

Jedna (možda šokantna) tvrdnja: uprkos tome što očigledno omogućava pisanje asinhronog JS koda (kao što je primer obrade događaja isticanja vremenskog intervala koji smo upravo opisali), sve donedavno (ES6), u sam JavaScript nije bilo ugrađeno zapravo ništa što bi direktno omogućavalo asinhronizam koda.

Molim!? To izgleda kao besmislena tvrdnja, zar ne? U stvari, prilično je tačna. Sama mašina jezika JS nije nikad radila ništa drugo osim što je u svakom datom trenutku izvršavala po jedan delić vašeg programa, kad god se to od nje zahtevalo.

„Od nje zahtevalo". Ko je zahtevao? To je onaj najvažniji deo!

JS mašina ne radi u izolaciji, nego unutar svog *radnog okruženja*, koje je za većinu programera tipičan čitač veba. Pomoću softvera kao što je Node.js, poslednjih nekoliko godina (ali ni u kom slučaju samo tokom tog vremena), upotreba JavaScripta proširila se i izvan čitača veba, na druga okruženja – kao što su serveri. U stvari, danas se JavaScript ugrađuje u sve moguće vrste uređaja – od robota do sijalica.

Zajednička „nit" (ovo je ne baš maštovita asinhrona igra reči) svih tih okruženja jeste to da u sebi imaju mehanizam koji upravlja izvršavanjem programa *tokom vremena* i koji u svakom trenutku poziva JS mašinu. Taj mehanizam se zove *petlja za obradu događaja* (engl. *event loop*).

Drugim rečima, JS mašina nema „urođen" osećaj za vreme, već služi kao okruženje za izvršavanje – na (spoljni) zahtev – proizvoljno zadatog dela JS koda. Spoljno okruženje JS mašine oduvek je određivalo *vremenski raspored* „događaja" (izvršavanja delova JS koda).

Tako, na primer, kada vaš JS program pošalje Ajax zahtev za učitavanje određenih podataka sa servera, kôd koji tumači odgovor servera zadajte unutar zasebne funkcije (čije je uobičajeno ime *povratna funkcija* /engl. *callback*/), a JS mašina kaže radnom okruženju: „Sada ću privremeno prekinuti rad, ali čim završiš obradu ovog zahteva koji sam poslala u mrežu i primiš neke podatke, molim te pozovi ovu povratnu funkciju".

Čitač veba onda osluškuje i čeka odziv iz mreže, a kada dobije nešto za vas, stavlja na raspored za izvršavanje povratnu funkciju tako što je umeće u petlju za obradu događaja.

Dakle, šta je petlja za obradu događaja?

Prvo ćemo je konceptualizovati pomoću malo pseudokoda:

```
// `eventLoop` je niz koji igra ulogu reda čekanja
// (obrada po modelu `prvi došao, prvi uslužen`, FIFO)
var eventLoop = [ ];
var event;

// petlja se izvršava "večno"
while (true) {
    // obrada jednog "podeljka"
    if (eventLoop.length > 0) {
        // učitati sledeći događaj iz reda čekanja
        event = eventLoop.shift();

        // izvršiti sledeći događaj
        try {
            event();
        }
        catch (err) {
            reportError(err);
        }
    }
}
```

Ovo je, razume se, veoma pojednostavljen pseudokod, koji ilustruje koncepte. Ali trebalo bi da bude dovoljan za bolje razumevanje suštine.

Kao što vidite, imamo petlju koja se neprekidno izvršava i koju smo predstavili u obliku petlje `while`, a svaka iteracija te petlje zove se *podeljak* (engl. *tick*). U svakom podeljku, ako postoji događaj koji čeka u redu, on se preuzima odatle i izvršava. Ti događaji su povratne funkcije.

Važno je imati na umu da `setTimeout(..)` ne postavlja vašu povratnu funkciju u red čekanja petlje za obradu događaja. Metoda `setTimeout(..)` pokreće odbrojavanje vremenskog intervala; kada taj interval istekne, radno okruženje postavlja vašu povratnu funkciju u petlju za obradu događaja, da bi je neki budući podeljak pokupio i izvršio.

Šta ako u tom trenutku u petlji događaja već ima 20 stavki na čekanju? Onda vaša povratna funkcija čeka. Ona staje u red i čeka iza svih ostalih – standardno ne postoji način za zaobilaženje i preskakanje mesta u redu prema napred. To objašnjava zašto intervali čekanja koje definišete pomoću metode `setTimeout(..)` nisu uvek tačne dužine. Garantovano je (grubo rečeno) da se vaša povratna funkcija neće pokrenuti *pre* kraja intervala čekanja koji zadate, ali se to može dogoditi tačno u tom trenutku ili posle njega, zavisno od stanja petlje za obradu događaja.

Drugim rečima, vaš program je najčešće izdeljen na više malih delova, koji se izvršavaju jedan za drugim u petlji za obradu događaja. Tehnički govoreći, u istom redu čekanja za izvršavanje mogu biti isprepleteni i događaji koji nisu u direktnoj vezi s vašim programom.



Uz tvrdnju da ES6 menja prirodu mesta na kome se upravlja redom čekanja u petlji za obradu događaja, upotrebili smo izraz „donedavno“. To je najvećim delom formalna tehnička napomena, ali pošto ES6 sada tačno propisuje kako radi petlja za obradu događaja, to znači da je, tehnički gledano, to u „nadležnosti“ JS mašine, a ne radnog okruženja JavaScripta. Jedan od najvažnijih razloga za tu promenu bilo je uvođenje ES6 obećanja, koja ćemo razmotriti u poglavlju 22, zato što ona zahtevaju direktnu i preciznu kontrolu nad operacijama raspoređivanja u redu čekanja petlje događaja (videti opis metode `setTimeout(..0)` u odeljku „Saradnja“ na strani 423).

Paralelni višenitni rad

Vrlo je uobičajeno da ljudi brkaju značenje reči „asinhrono“ i „paralelno“ ali one znače zapravo sasvim različite stvari. Ne zaboravite, „asinhrono“ se odnosi na interval između *sada* i *kasnije*, a „paralelno“ opisuje stvari koje se mogu odvijati istovremeno.

Najuobičajenije alatke za paralelan rad koda jesu *procesi* i *niti izvršavanja* (engl. *threads*). Procesi i niti izvršavaju se nezavisno jedni od drugih i mogu se izvršavati istovremeno: na zasebnim procesorima, pa čak i na zasebnim računarima, ali više niti može deliti memoriju istog procesa.

Nasuprot tome, petlja za obradu događaja deli svoje zadatke na poslove koje izvršava serijski (jedan za drugim), što sprečava upotrebu paralelnih procesa i menjanje sadržaja deljene memorije. Paralelizam i „serijalizam“ mogu postojati istovremeno u obliku petlji za obradu događaja koje međusobno sarađuju ali rade u zasebnim nitima.

Do preplitanja paralelnih niti izvršavanja i preplitanja asinhronih događaja može doći na vrlo različitim nivoima granularnosti.

Na primer:

```
function later() {
    answer = answer * 2;
    console.log( "Smisao života je:", answer );
}
```

Ceo sadržaj funkcije `later()` bio bi jedna stavka u redu čekanja petlje za obradu događaja, ali ako razmatramo nit u kojoj bi se ovaj kôd izvršavao, imali bismo zapravo desetak operacija niskog nivoa. Na primer, `answer = answer * 2` zahteva da se prvo učita tekuća vrednost promenjive `answer`, zatim da se negde smesti vrednost `2`, čemu sledi operacija množenja, a onda se rezultat te operacije smešta u `answer`.

U jednonitnom okruženju, nije baš bitno da li su stavke u redu čekanja niti operacije niskog nivoa, zato što ništa ne može da prekine izvršavanje niti. Ali ako imate paralelan sistem, gde dve različite niti rade za isti program, vrlo je moguće nepredvidljivo ponašanje.

Razmotrite sledeće:

```
var a = 20;

function foo() {
    a = a + 1;
}

function bar() {
    a = a * 2;
}

// ajax(..) je neka funkcija iz biblioteke
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

U jednonitnom ponašanju JavaScripta, ako se funkcija `foo()` izvrši pre funkcije `bar()`, rezultat je to da `a` ima vrednost `42`, ali ukoliko se funkcija `bar()` izvrši pre funkcije `foo()`, rezultat u `a` biće `41`.

Međutim, ako se JS događaji koji dele iste podatke izvršavaju paralelno, problemi bi bili znatno suptilniji. Razmotrite sledeće dve liste poslova (izražene pomoću pseudokoda) kao niti koje izvršavaju kôd funkcije `foo()`, odnosno `bar()` i pogledajte šta se događa kada se obe niti izvršavaju tačno u isto vreme:

Nit 1 (`X` i `Y` su privremene lokacije u memoriji):

```
foo():
  a. učitati vrednost `a` u `X`
  b. upisati `1` u `Y`
  c. sabrati `X` i `Y`, upisati rezultat u `X`
  d. vrednost `X` upisati u `a`
```

Nit 2 (`X` i `Y` su privremene lokacije u memoriji):

```
bar():
  a. učitati vrednost `a` u `X`
  b. upisati `2` u `Y`
  c. pomnožiti `X` sa `Y`, upisati rezultat u `X`
  d. vrednost `X` upisati u `a`
```

A sada, recimo da se obe niti izvršavaju potpuno paralelno. Verovatno već možete da uočite problem, zar ne? Obe niti koriste za svoje privremene korake iste deljene memorijske lokacije `X` i `Y`.

Koji je konačan rezultat u `a` ako se koraci izvrše sledećim redosledom?

```
1a  (učitati vrednost `a` u `X` ==> `20`)
2a  (učitati vrednost `a` u `X` ==> `20`)
1b  (upisati `1` u `Y` ==> `1`)
2b  (upisati `2` u `Y` ==> `2`)
1c  (sabrati `X` i `Y`, upisati rezultat u `X` ==> `22`)
1d  (vrednost `X` upisati u `a` ==> `22`)
2c  (pomnožiti `X` sa `Y`, upisati rezultat u `X` ==> `44`)
2d  (vrednost `X` upisati u `a` ==> `44`)
```

Rezultat u `a` biće `44`. Ali šta ako je redosled sledeći?

```
1a  (učitati vrednost `a` u `X` ==> `20`)
2a  (učitati vrednost `a` u `X` ==> `20`)
2b  (upisati `2` u `Y` ==> `2`)
1b  (upisati `1` u `Y` ==> `1`)
2c  (pomnožiti `X` sa `Y`, upisati rezultat u `X` ==> `20`)
1c  (sabrati `X` i `Y`, upisati rezultat u `X` ==> `21`)
1d  (vrednost `X` upisati u `a` ==> `21`)
2d  (vrednost `X` upisati u `a` ==> `21`)
```

Rezultat u `a` biće `21`.

Dakle, programiranje niti može biti vrlo zapetljano, zato što ako ne preduzmete posebne mere da sprečite događanje te vrste prekidanja/preplitanja, možete dobiti veoma iznenađujuće i nedeterminističko ponašanje koje često dovodi do glavobolja.

JavaScript nikada ne deli podatke između niti, što znači da taj nivo nedeterminizma ne pravi problem. Ali to ne znači da je JS uvek deterministički. Već smo videli da različit redosled izvršavanja funkcija `foo()` i `bar()` proizvodi dva različita rezultata (`41` ili `42`).



Možda još nije očigledno, ali nije svaki nedeterminizam loš. Nedeterminizam je ponekad nebitan, a ponekad je nameran. Više primera toga videćemo u ovom i u sledećih nekoliko poglavlja.

Potpuno izvršavanje

Zbog jednonitne prirode JavaScripta, kôd unutar funkcije `foo()` (i `bar()`) čini jednu jedinicu, što znači sledeće: čim funkcija `foo()` počne da se izvršava, ceo njen kôd će se izvršiti pre nego što počne izvršavanje koda funkcije `bar()`, i obrnuto. To ponašanje se zove *potpuno izvršavanje* (engl. *run-to-completion*).

U stvari, semantika potpunog izvršavanja postaje jasnija kada funkcije `foo()` i `bar()` sadrže više koda, na primer:

```
var a = 1;
var b = 2;

function foo() {
    a++;
    b = b * a;
    a = b + 3;
}

function bar() {
    b--;
    a = 8 + b;
    b = a * 2;
}

// ajax(..) je neka Ajax funkcija iz biblioteke
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

Pošto funkcija `bar()` ne može prekinuti izvršavanje funkcije `foo()`, niti obrnuto, ovaj program može dati samo dva moguća rezultata, u zavisnosti od toga koja funkcija prva počne da se izvršava – kada bi bio moguć višenitni rad i kad bi bilo dozvoljeno da se prepliće izvršavanje pojedinačnih iskaza funkcija `foo()` i `bar()`, broj mogućih rezultata programa značajno bi se povećao!

Deo 1 programa je sinhron (događa se *sada*), ali su delovi 2 i 3 asinhroni (događaju se *kasnije*), što znači da će između njihovog izvršavanja proteći određeno vreme.

Deo 1:

```
var a = 1;
var b = 2;
```

Deo 2 (`foo()`):

```
a++;
b = b * a;
a = b + 3;
```

Deo 3 (`bar()`):

```
b--;
a = 8 + b;
b = a * 2;
```

Pošto se od delova 2 i 3 svaki može izvršiti pre onog drugog, ovaj program može imati dva moguća rezultata, kao što je ovde ilustrovano:

Rezultat 1:

```
var a = 1;
var b = 2;

// foo()
a++;
b = b * a;
a = b + 3;

// bar()
b--;
a = 8 + b;
b = a * 2;

a; // 11
b; // 22
```

Rezultat 2:

```
var a = 1;
var b = 2;

// bar()
b--;
a = 8 + b;
b = a * 2;

// foo()
a++;
b = b * a;
a = b + 3;

a; // 183
b; // 180
```

Dva moguća rezultata istog koda znače da i dalje imamo nedeterminizam! Ali to je na nivou redosleda izvršavanja funkcija (događaja), a ne na nivou redosleda izvršavanja iskaza programa (ili, u stvari, na nivou redosleda operacija nad izrazima) kao u slučaju niti. Drugim rečima, determinizam programa je bolji nego u slučaju višenitnog izvršavanja.

Primenjeno na ponašanje JavaScripta, za taj nedeterminizam usled redosleda izvršavanja funkcija postoji uobičajen izraz *stanje utrkivanja* (engl. *race condition*) jer se funkcije `foo()` i `bar()` međusobno trkaju koja će se prva izvršiti. Konkretno, imamo trku zato što nema pouzdanog načina da predvidimo koju će vrednost dobiti `a` i `b`.



Kada bi u JavaScriptu postojala funkcija čije ponašanje nije potpuno izvršavanje, mogli bismo imati mnogo više mogućih rezultata, zar ne? Ispostavlja se da ES6 uvodi jednu baš takvu stvar (videti poglavlje 23), ali zasad ne brinite – i to ćemo razmotriti!

Istovremenost

Zamislite veb lokaciju koja prikazuje listu ažuriranih statusa (kao što je lista vesti na društvenim mrežama) koja se puni dok korisnik pomera sadržaj prozora. Da bi to radilo kako treba, potrebno je da se (barem) dva zasebna „procesa“ izvršavaju *istovremeno* (na primer, tokom istog vremenskog intervala, ali ne obavezno i u istom trenutku).



Reč „proces“ ovde pišemo između navodnika zato što to nisu pravi procesi operativnog sistema kako ih računarska nauka definiše. To su virtuelni procesi, ili poslovi, koje čine logički povezani sekvencijalni nizovi operacija. Reč „proces“ koristimo umesto „posao“ zato što ona terminološki odgovara definiciji koncepata koje ovde razmatramo.

Prvi „proces“ će se odazivati na događaje `onscroll` (tako što će slati Ajax zahteve za nov sadržaj) kako oni nastaju dok korisnik pomera sadržaj veb stranice nadole. Drugi „proces“ će primati Ajax odgovore (radi prikazivanja novog sadržaja na stranici).

Očigledno, ako korisnik pomera sadržaj stranice dovoljno brzo, možda ćete videti dva ili više događaja `onscroll` unutar jednog vremenskog intervala koji je potreban da stigne prvi odgovor i da se on obradi, pa ćete zato imati brzo odigravanje događaja `onscroll` i događaja Ajax, isprepletene jedne s drugim.

Istovremenost (engl. *concurrency*) imamo kada se dva ili više „procesa“ izvršavaju istovremeno tokom istog perioda, bez obzira na to da li se operacije koje ih čine odvijaju *paralelno* (u istom trenutku na zasebnim procesorma ili jezgrima). Istovremenost možete zamisliti kao paralelizam na nivou „procesa“ (ili poslova), nasuprot paralelizmu na nivou operacija (niti koje rade na zasebnim procesorima).



Istovremenost uvodi i opcioni pojam međusobne interakcije tih „procesa“, na koji ćemo se vratiti u nastavku teksta.

Tokom datog vremenskog intervala (nekoliko sekundi tokom kojih korisnik pomera sadržaj prozora), pokušaćemo da vizuelizujemo svaki nezavisan „proces“ kao niz događaja/operacija:

„Proces“ 1 (događaji `onscroll`):

```
onscroll, request 1
onscroll, request 2
onscroll, request 3
onscroll, request 4
```

POGLAVLJE 27

Sintaksa

Ako ste pisali JS kôd barem neko vreme, vrlo je verovatno da vam je sintaksa prilično poznata. Svakako postoje mnoge posebnosti, ali u opštem slučaju, to je prilično smislena i lako razumljiva sintaksa koja ima mnogo sličnosti s drugim jezicima.

Međutim, ES6 uvodi priličan broj novih sintaksnih oblika na koje se treba navići. U ovom poglavlju ćemo ih navesti pojedinačno da bismo razmotrili šta sve postoji.



U vreme pisanja ove knjige, neke od mogućnosti koje razmatramo već su implementirane u raznim čitačima veba (Firefox, Chrome itd.), dok su druge implementirane samo delimično, a mnoge uopšte nisu implementirane. Vaše lično iskustvo koje ćete imati ako isprobate primere iz ove knjige može biti drugačije od onog što je ovde opisano. U takvim slučajevima, isprobajte ih u verziji s transpajlerima, pošto te alatke pokrivaju naveći deo novih ES6 mogućnosti.

ES6Fiddle (*http://www.es6fiddle.net/*) odlično i lako upotrebljivo „igralište“ za isprobavanje ES6 koda, kao što je veb REPL za transpajler Babel (*http://babeljs.io/repl/*).

Deklaracije čiji je opseg vidljivosti blok

Verovatno znate da je u JavaScriptu osnovna jedinica za opseg vidljivosti promenljivih oduvek bila funkcija. Ako vam je trebalo da formirate opseg vidljivosti čija je veličina određeni blok koda, najuobičajeniji način da to uradite – osim korišćenja standardne funkcije – bio je upotreba *funkcijskog izraza za trenutno izvršavanje* (IIFE). Na primer:

```
var a = 2;

(function IIFE(){
    var a = 3;
    console.log( a );    // 3
})();

console.log( a );        // 2
```

Deklarisanje promenljivih pomoću rezervisane reči let

Međutim, sada možemo zadavati deklaracije promenljivih koje su vezane za određeni blok, što se (nimalo iznenađujuće) zove *opseg vidljivosti veličine bloka*. To znači da nam treba samo par { .. } kako bismo definisali opseg vidljivosti. Umesto rezervisane reči `var`, koja uvek deklariše promenljive čiji je opseg vidljivosti okružujuća funkcija (ili globalni opseg, ako su deklarisane na najvišem nivou), zadajte rezervisanu reč `let`:

```
var a = 2;

{
    let a = 3;
    console.log( a );    // 3
}

console.log( a );        // 2
```

U JS-u dosad nije bila mnogo uobičajena upotreba samostalnog bloka { .. }, ali je to oduvek bila ispravna sintaksa. Taj model će odmah prepoznati programeri iz drugih jezika u kojima se može definisati *opseg vidljivosti veličine bloka*.

Verujem da je to najbolji način da se deklarišu promenljive čiji opseg vidljivosti je blok, unutar vlastitog bloka { .. }. Osim toga, trebalo bi da `let` deklaraciju (ili deklaracije) uvek postavite na početak tog bloka. Ako treba da deklarišete više promenljivih, preporučio bih vam da zadate samo jedan `let`.

Što se stila pisanja tiče, ja čak najradije pišem `let` u istom redu gde je početna zagrada {, kako bih jasnije istakao da je svrha tog bloka samo deklarisanje opsega vidljivosti tih promenljivih.

```
{   let a = 2, b, c;
    // ..
}
```

To će izgledati čudno i verovatno se neće slagati s preporukama iz većeg dela literature o ES6. Ali imam dobre razloge za svoju uvrnutost.

Postoji još jedan eksperimentalan (nestandardizovan) oblik `let` deklaracije nazvan `let` blok, koji izgleda ovako:

```
let (a = 2, b, c) {
    // ..
}
```

Taj oblik je ono što zovem *eksplicitni* opseg vidljivosti veličine bloka, dok je oblik `let ..` deklaracije koji preslikava oblik `var` deklaracije više *implicitan*, pošto nije najjasnije vidljivo kojem paru { .. } pripada. Programeri uglavnom smatraju da je upotreba *eksplicitnih* mehanizama ipak preporučljivija od *implicitnih* mehanizama, a ja tvrdim da je ovo jedan od takvih slučajeva.

Ako uporedite prethodna dva oblika, oni su vrlo slični, a po mom mišljenju, oba po stilu pripadaju *eksplicitnom* zadavanju opsega vidljivosti veličine bloka. Nažalost, oblik

`let (..) { .. }`, *najeksplicitnija* opcija, nije prihvaćen u verziji ES6. To će se možda promeniti u nekoj verziji posle ES6, ali mislim da nam je prva opcija zasad najbolja.

Da biste bolje shvatili *implicitnu* prirodu `let ..` deklaracija, razmotrite sledeće slučajeve upotrebe:

```
let a = 2;

if (a > 1) {
    let b = a * 3;
    console.log( b );       // 6

    for (let i = a; i <= b; i++) {
        let j = i + 10;
        console.log( j );
    }
    // 12 13 14 15 16

    let c = a + b;
    console.log( c );       // 8
}
```

Brza provera, bez gledanja u primer: koja promenljiva (ili promenljive) postoji samo unutar iskaza `if`, a koja promenljiva (ili promenljive) postoji samo unutar petlje `for`?

Odgovori: iskaz `if` sadrži promenljive `b` i `c` čiji je opseg vidljivosti blok iskaza `if`, a petlja `for` sadrži promenljive `i` i `j` čiji je opseg vidljivosti blok petlje.

Da li ste morali da razmišljate izvesno vreme? Iznenađuje li vas to što promenljiva `i` nije dodata u opseg vidljivosti bloka svog okružujećeg iskaza `if`? Razlog te mentalne pauze i pitanja – ja to zovem „mentalna taksa“ – potiče od činjenice da taj `let` mehanizam ne samo što je za nas novina, nego je i *implicitan*.

Činjenica da se deklaracija `let c = ..` nalazi tako duboko unutar opsega vidljivosti uvodi i određenu opasnost. Za razliku od promenljivih koje deklarišete na tradicionalan način pomoću `var`, koje važe u celom okružujućem opsegu vidljivosti funkcije u kojem su deklarisane, `let` deklaracije se dodaju u opseg vidljivosti svog bloka, ali se ne inicijalizuju sve do mesta gde se pojavljuju u bloku.

Pristupanje promenljivoj koja je deklarisana pomoću `let` pre njene `let ..` deklaracije/inicijalizacije prouzrokuje grešku, dok je za `var` deklaracije taj redosled nevažan (osim u pogledu stila).

Razmotrite sledeće:

```
{
    console.log( a );    // undefined
    console.log( b );    // ReferenceError!

    var a;
    let b;
}
```



Ta greška `ReferenceError` zbog preranog pristupanja promenljivoj koja je deklarisana pomoću `let` tehnički se zove *TDZ (Temporal Dead Zone – privremena mrtva zona)* greška – pristupate promenljivoj koja je deklarisana ali još nije inicijalizovana. To nije jedini slučaj kada viđamo TDZ greške – one se u ES6 pojavljuju na više mesta. Osim toga, imajte u vidu da „inicijalizovana" ne znači eksplicitno dodeljivanje vrednosti u kodu, pošto je `let b;` potpuno ispravna sintaksa. Budući da se za promenljivu kojoj u trenutku deklarisanja nije dodeljena nikakva vrednost podrazumeva da joj je dodeljena vrednost `undefined`, iskaz `let b;` isto je što i `let b = unde fined;`. Bez obzira na to da li je inicijalizujete eksplicitno ili implicitno, promenljivoj `b` ne možete pristupati dok se ne izvrši iskaz `let b`.

Poslednja zamka: operator `typeof` se sa TDZ promenljivama ponaša drugačije nego s nedeklarisanim (ili deklarisanim!) promenljivama. Na primer:

```
{
    // `a` nije deklarisana
    if (typeof a === "undefined") {
        console.log( "cool" );
    }

    // `b` je deklarisana, ali u svojoj TDZ
    if (typeof b === "undefined") { // ReferenceError!
        // ..
    }

    // ..

    let b;
}
```

Pošto promenljiva `a` nije deklarisana, upotreba operatora `typeof` jedini je bezbedan način da proverimo postoji li ona ili ne. Ali, izraz `typeof b` prouzrokuje TDZ grešku zato što duboko niže u kodu slučajno imamo deklaraciju `let b`. Uh.

Sada bi trebalo da bude jasnije zbog čega insistiram da se sve `let` deklaracije nalaze na početku svojih opsega vidljivosti. Tako se potpuno izbegavaju nenamerne greške zbog preranog pristupanja promenljivoj. Osim toga, tako postaje *eksplicitnije* – kada pogledate početak bloka, svakog bloka – koje promenljive on sadrži.

Vaši blokovi (iskazi `if`, petlje `while` itd.) ne moraju da dele svoje izvorno ponašanje s ponašanjem u vezi sa opsegom vidljivosti.

Ta vaša eksplicitnost, koju ćete samo morati da disciplinovano održavate, na duži rok će vas poštedeti mnogih glavobolja.



Više informacija o deklarisanju promenljivih pomoću `let` i opsega vidljivosti veličine bloka naći ćete u drugom delu ove knjige – *Opseg vidljivosti i ograde*.

Deklaracija let i petlja for

Jedini izuzetak koji bih napravio u vezi s *eksplicitnim* oblikom deklaracije `let` u bloku jeste `let` deklaracija koja se nalazi u zaglavlju petlje `for`. Razlog će vam možda izgledati nejasan, ali lično verujem da je to jedna od najvažnijih ES6 odlika.

Razmotrite sledeće:

```
var funcs = [];

for (let i = 0; i < 5; i++) {
    funcs.push( function(){
        console.log( i );
    } );
}

funcs[3]();     // 3
```

Izraz `let i` u zaglavlju petlje `for` deklariše promenljivu `i` ne samo za petlju `for`, nego ponovo deklariše novu promenljivu `i` u svakoj iteraciji petlje. To znači da ograde koje se formiraju unutar jedne iteracije petlje ograđuju te promenljive po iteraciji, kao što biste i očekivali.

Ako isprobate isti primer, ali s deklaracijom `var i` u zaglavlju petlje `for`, rezultat će biti `5` umesto `3`, zato što bi u spoljašnjem opsegu koji funkcija ograđuje postojala samo jedna promenljiva `i`, umesto nove promenljive `i` u svakoj iteraciji koju bi funkcija ogradila.

Isto možete postići i u neznatno opširnijem obliku:

```
var funcs = [];

for (var i = 0; i < 5; i++) {
    let j = i;
    funcs.push( function(){
        console.log( j );
    } );
}

funcs[3]();     // 3
```

U ovom slučaju, izričito deklarišemo novu promenljivu `j` u svakoj deklaraciji, a nakon toga ograda deluje na isti način. Draže mi je prvo rešenje; zbog dodatne mogućnosti koju ono pruža radije se opredeljujem za oblik `for (let .. ) ...`. Može se opravdano zameriti da je to malo *implicitniji* oblik, ali je za moj ukus ipak dovoljno *eksplicitan* i koristan.

Deklaracija pomoću `let` ponaša se na isti način i u petljama `for..in` *i* `for..of` (videti odeljak „Petlja for..of“ na strani 693).

Deklarisanje konstanti

Postoji još jedna deklaracija čiji opseg važenja je blok: to je deklaracija pomoću rezervisane reči `const`, čime se formiraju *konstante*.

Šta je tačno konstanta? To je promenljiva čija se vrednost nakon inicijalizovanja može samo čitati. Razmotrite sledeće:

```
{
    const a = 2;
    console.log( a );    // 2

    a = 3;               // greška TypeError!
}
```

Nakon inicijalizovanja vrednosti promenljive, u trenutku deklarisanja, više nije dozvoljeno menjati tu vrednost. Deklaracija `const` mora sadržati i eksplicitnu operaciju inicijalizovanja vrednosti promenljive. Ako želite da vrednost *konstante* bude `undefined`, morate deklarisati `const a = undefined`.

Konstante ni po čemu ne ograničavaju samu vrednost, nego samo dodeljivanje vrednosti konstanti. Drugim rečima, sama vrednost nije zbog deklaracije `const` postala ni „zamrznuta“, ni nepromenljiva, nego to važi samo za operaciju dodeljivanja vrednosti. Ako je to neka složena vrednost, kao što je objekat ili niz, sadržaj te vrednosti se i dalje može menjati:

```
{
    const a = [1,2,3];
    a.push( 4 );
    console.log( a );    // [1,2,3,4]

    a = 42;              // greška TypeError!
}
```

Promenljiva `a` ne sadrži zaista konstantan niz, nego konstantnu referencu na niz. Sam niz se slobodno može menjati.



Dodeljivanje objekta ili niza kao vrednost konstante znači da sakupljač smeća neće moći da pokupi tu vrednost sve dok leksički opseg vidljivosti te konstante ne nestane, zato što se nikad ne može izbrisati referenca na tu vrednost. To može biti poželjno, ali vodite računa ako vam to nije bila namera!

U suštini, deklaracija pomoću `const` sada samo nameće ono što smo godinama ugrađivali u kôd kao stilski oblik, gde smo ime promenljive deklarisali velikim slovima i dodeljivali joj neku literalnu vrednost za koju smo vodili računa da je nakon toga nikad ne menjamo. Nema ograničenja za promenljive deklarisane pomoću `var`, ali ima za promenljive deklarisane pomoću `const`, što može da vam pomogne da otkrijete nenamerne izmene vrednosti.

Rezervisana reč `const` *može* se upotrebiti za deklarisanje promenljivih u petljama `for`, `for..in` i `for..of` (videti odeljak „Petlja for..of“ na strani 693). Međutim, svaki pokušaj dodeljivanja nove vrednosti, kao što je u tipičnoj odredbi `i++` petlje `for`, prouzrokovaće grešku.

Da li koristiti const ili ne

Postoje mišljenja da bi u nekim slučajevima JS mašina mogla da optimizuje `const` deklaraciju bolje od `let` ili `var` deklaracije. Teorijski, tada mašina jezika lakše prepoznaje da se vrednost/tip promenljive neće nikad menjati, pa zato može da eliminiše moguć režijski deo posla.

Bez obzira na to da li je `const` deklaracija zaista korisna ili su to samo naše fantazije i intuicije, važnija odluka koju treba doneti je da li namerno želite ponašanje konstante ili ne. Ne zaboravite: jedna od najvažnijih uloga vašeg izvornog koda jeste da jasno pokazuje šta je tačno vaša namera, i to ne samo vama u ovom trenutku, nego i vama ubuduće, kao i drugima koji sarađuju s vama na tom kodu.

Neki programeri radije prvo deklarišu svaku promenljivu kao `const` a zatim ublažavaju deklaraciju u `let` ako se u kodu pojavi potreba da se njena vrednost menja. To je zanimljiva ideja, ali nije jasno da li to zaista poboljšava čitljivost ili razumljivost koda.

To nije nekakva *zaštita*, kao što mnogi veruju, zato što svaki programer koji naknadno poželi da promeni vrednost promenljive deklarisane kao `const` može jednostavno u njenoj deklaraciji `const` izmeniti u `let`. U najboljem slučaju, to štiti od slučajnih izmena. Ali ponovo, izvan naših intuicija i gledišta, izgleda da ne postoji nikakva objektivna i jasna mera šta je tačno „slučajno“ i kako to predupredi. Slična oprečna mišljenja postoje i u vezi s konverzijom tipova.

Moj savet: da biste izbegli potencijalno zbunjujući kôd, koristite `const` isključivo za deklarisanje promenljivih za koje namerno i očigledno želite da istaknete da se njihove vrednosti neće menjati. Drugim rečima, ponašanje koda nemojte *zasnivati* na `const`, nego tu vrstu deklaracije koristite kao alatku da biste signalizirali nameru, kada se ta namera može jasno pokazati.

Funkcije čiji je opseg vidljivosti blok

Počev od ES6, opseg vidljivosti funkcija čije se deklaracije nalaze unutar blokova ograničen je na blok u kojem su deklarisane. Pre ES6, to nije bilo izričito zadato u specifikaciji, ali su mnoge implementacije tako radile, pa je sada specifikacija usklađena sa stvarnošću.

Razmotrite sledeće:

```
{
    foo();                  // radi!

    function foo() {
        // ..
    }
}

foo();                      // ReferenceError
```

Funkcija `foo()` je deklarisana unutar bloka `{ .. }` i za ES6 njen opseg vidljivosti je taj blok. Dakle, ona nije vidljiva izvan tog bloka. Ali obratite pažnju i na to da je deklaracija funkcije „podignuta“ na početak bloka, za razliku od `let` deklaracija, koje uvode već ranije pomenutu zamku TDZ greške.

Deklarisanje funkcija čiji je opseg vidljivosti blok može biti problem ako ste navikli da pišete kôd kao što je sledeći i očekujete ponašanje kao ranije, kad opseg vidljivosti nije bio blok:

```
if (something) {
    function foo() {
        console.log( "1" );
    }
}
else {
    function foo() {
        console.log( "2" );
    }
}

foo();    // ??
```

U okruženjima starijim od ES6, funkcija `foo()` bi uvek ispisala `"2"` bez obzira na vrednost `something`, zato što su se obe deklaracije funkcije izvlačile iz svojih blokova i uvek bi važila samo poslednja deklaracija.

U ES6, poslednji red izaziva grešku `ReferenceError`.

Operator za razdvajanje/grupisanje ostatka vrednosti

ES6 uvodi nov operator `...` koji se obično naziva operator za *razdvajanje* ili operator za *grupisanje ostatka* vrednosti (engl. *spread* ili *rest operator*), zavisno od toga gde/kako je upotrebljen. Pogledajmo kako to izgleda:

```
function foo(x,y,z) {
    console.log( x, y, z );
}

foo( ... [1,2,3] );            // 1 2 3
```

Kada operator `...` zadate ispred nekog niza (zapravo, ispred svakog *iterabilnog objekta*, što ćemo razmotriti u poglavlju 28), on se ponaša tako da „razdvaja“ niz na njegove pojedinačne elemente.

Na slučajeve upotrebe u obliku kao što je prikazan u primeru, nailazićete kada treba razdvojiti zadati niz na skup argumenata u pozivu funkcije. Operator `...` se u tom slučaju ponaša kao jednostavnija sintaksna zamena metode `apply(..)`, koju bismo pre ES6 najčešće upotrebili u sledećem obliku:

```
foo.apply( null, [1,2,3] ); // 1 2 3
```

Ali, operator `...` može se iskoristiti za razdvajanje/izdvajanje vrednosti i u drugim kontekstima – recimo, unutar deklaracije niza:

```
var a = [2,3,4];
var b = [ 1, ...a, 5 ];

console.log( b );    // [1,2,3,4,5]
```