DEO II: PROGRAMSKI JEZIK JavaScript

Sadržaj

Poglavlje 3: OSNOVE PROGRAMSKOG JEZIKA JavaScript

U ovom poglavlju prikazaćemo programski jezik JavaScript. Obuhvatićemo sve bitne mogućnosti samog jezika i nećemo se baviti specifičnostima dva najčešća ambijenta korišćenja, brauzerom i ambijentom van brauzera što je dosta uobičajeno. Iako je cilj ovoga dela knjige da pruži obuhvatan prikaz jezika JavaScript, određeni sadržaji će ostati nepokriveni. Razlog tome je pre svega karakter knjige u kojoj je centralna tema funkcionalno programiranje a JavaScript je „samo" tehnologija implementacije. Drugi razlog je izuzetno obiman materijal koji bi, u slučaju potpunog pokrivanja jezika, trebalo izložiti. Za vrlo kompletan i detaljan uvid u programski jezik JavaScript čitalac se upućuje na brojne knjige i izvore na internetu, kao što su https://javascript.info/ i https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide.

Pre no što "zaronimo" u detalje jezika, još nekoliko reči o njegovoj orijentaciji u odnosu na paradigme programiranja.

JavaScript nije funkcionalni programski jezik, već je to multiparadigmatski jezik koji, pre svega, podržava sledeće paradigme koje su **direktno podržane ugrađenim jezičkim konstruktima**:

- Imperativno programiranje - programiranje usmereno ka detaljnom opisivanju algoritamskih koraka u programu.
- Objektno orijentisano programiranje zasnovano na prototipovima - programiranje zasnovano na prototipskim objektima i njihovim instancama.
- Metaprogramiranje - programiranje koje manipuliše osnovom JavaScript modela izvršavanja i koje je, po mišljenju eksperata, izuzetan kvalitet JavaScript-a.

Druge paradigme, kao što su klasno-orijentisano objektno programiranje, programiranje vođeno događajima, pa i funkcionalno programiranje (do izvesne mere), podržavaju se koristeći jezik JavaScript kao medij za implementaciju; razumljivo i precizno rečeno: za njih se podrška mora posebno programirati i to se može uraditi korišćenjem ugrađenih jezičkih konstrukata JavaScript-a. Posledica ove situacije su brojne biblioteke pisane u JavaScript-u koje značajno proširuju mogućnosti JavaScript-a izvan granica ugrađenih jezičkih konstrukata.

3.1. O programskom kodu, pojmu *vrednost* i pojmu *varijabla*

Programski kod je skup instrukcija koje računaru kažu šta treba da radi prilikom izvršavanja programa. Izvršavanje programa svodi se na manipulaciju entitetima koji su predstavljeni u kodu.

U računarstvu i programiranju softvera, ***vrednost*** je reprezentacija nekog entiteta kojom program može da manipuliše, da nad njima vrši neke operacije.

Varijabla je apstraktna lokacija za skladištenje uparena sa pridruženim simboličkim imenom koja može da skladišti neku poznatu ili nepoznatu vrednost.

Iz aspekta hardvera računara, vrednosti su samo "gomile nula i jedinica" a operacije su specifične za dati hardver.

Iz aspekta programera, vrednosti mogu da budu različite stvari počevši od jednostavnih stvari kao što su brojevi, alfabetski znakovi, oznake koje se dodeljuju da označe istinitost (`true` ili `false`) pa do složenih kao što su objekti i funkcije.

Vrednostima se dodeljuju *tipovi* - skupovi pravila koja vrednost mora da zadovoljava. Ta pravila definišu same vrednosti i operacije koje se nad tim vrednostima mogu izvršavati.

U zavisnosti od sistema tipiziranja programskog jezika, varijable mogu da čuvaju samo određeni unapred deklarisani tip vrednosti (strogo tipiziranje) ili tip može da "proističe" iz trenutno uskladištene vrednosti (slabo tipiziranje) na taj način dozvoljavajući jednoj promenljivoj da skladišti sve vrste vrednosti koje podržava programski jezik. JavaScript je *slabo* tipiziran programski jezik.

3.2. Struktura koda u programskom jeziku JavaScript

Sintaksni konstrukti koji se pojavljuju u programu pisanom u programskom jeziku JavaScript su: ***izrazi, deklaracije***, ***naredbe***, ***komentari*** i ***ostali konstrukti***.

Izrazi su sintaksni konstrukti jezika koji se evaluiraju se na ***vrednost***. Sastoje se od elemenata koji mogu da budu ***doslovne vrednosti*** (na primer: 1, true, 'ABC$'), ***varijable*** (imena poput `_promenljiva`, `mojObjekat`, `mojeIme` kojima se identifikuju vrednosti), ili kombinacija vrednosti i varijabli povezanih operatorima (na primer, `_promenljiva = mojeIme + ' ABC$'`).

Deklaracije su sintaksni konstrukti jezika koji ***izjavljuju određene osobine drugih konstrukata*** (ponekad ih zovu i ***deklarativne naredbe***). Deklaracije u JavaScript-u obuhvataju deklaracije promenljivih, deklaracije funkcija, deklaracije za rad sa modulima i deklaraciju klase. To su sledeći konstrukti:

- Deklaracije varijabli:
  - **`let`** i **`var`** - Deklarišu mutabilnu[1] varijablu opciono je inicijalizujući na vrednost.
  - **`const`** - Deklariše imenovanu konstantu koja je imutabilna (što znači da joj se vrednost može dodeliti samo jednom) i obavezno joj dodeljuje vrednost.
- Deklaracije funkcija:
  - **`function`** - Deklariše funkciju sa specificiranim parametrima.
  - **`function*`** - Deklariše generatorsku funkciju koja olakšava pisanje iteratora.
  - **`async function`** - Deklariše asinhronu funkciju sa specificiranim parametrima.
  - **`async function*`** - Deklariše asinhronu generatorsku funkciju koja olakšava pisanje asinhronih iteratora.
- **`class`** - Deklariše klasu.
- **`export`** - Koristi se za izvoz funkcija kako bi se one učinile raspoloživim za uvoz u eksterne module i druge skriptove. (Napomena: može da se pojavi na najvišem nivou modula)
- **`import`** – Koristi se za uvoz funkcija izvezenih iz drugog modula ili drugog skripta (Napomena: može da se pojavi na najvišem nivou modula)

Naredbe su sintaksni konstrukti koji rezultuju izvršavanjem akcija (često ih zovu ***izvršne naredbe***). Program je, iz aspekta izvršavanja, sekvenca naredbi.
Jedna naredba može se pisati u više linija, ali se obično piše u jednoj liniji. Kraj naredbe označava se terminalnim simbolom tačka-zapeta (`;`). U većini slučajeva može se izostaviti terminator, odnosno u većini slučajeva nova linija implicira terminalni simbol `;`. Ima i situacija kada JavaScript nije u stanju da ispravno pretpostavi terminalni simbol i o tome treba voditi računa (primer na adresi https://javascript.info/structure). Dozvoljeno je i više naredbi u jednoj liniji i tada se one razdvajaju terminalnim simbolom.

Izvršne naredbe u jeziku JavaScript mogu se kategorisati u sledeće grupe:

- Naredbe dodele;

[1] Mutabilna varijabla je varijabla u koju se vrednosti mogu višekratno upisivati (na primer, `x = 12; x = true;`)

- Naredbe kontrole toka
- Naredbe iteriranja;

Naredbe dodele vrše dodelu identifikatora vrednostima. Dodela se vrši pomoću ***operatora dodele***. Operator dodele (koji je binarni operator) dodeljuje svom levom operandu (operandu sa leve strane operatora) vrednost koja se dobija evaluiranjem desnog operanda (operanda sa leve strane operatora).

Osnovni operator dodele, koji se označava znakom jednakosti (**=**), dodeljuje vrednost operanda sa svoje desne strane operandu sa svoje leve strane. Na primer,

```
z = f(a);
```

je naredba dodele koja varijabli `z` dodeljuje vrednost koju vrati poziv funkcije `f(x)`za vrednost argumenta `x = a`.

Postoje i kompozitni operatori koji, pored dodele, sadrže i druge operacije, poput aritmetičkih. Na primer, dodela sabiranja koja se označava sa **+=** ima sintaksu `z += f(a)` sa sledećim značenjem `z = z + f(a)`. Ako se izraz evaluira na objekat, leva strana naredbe dodele može da vrši dodele svojstvima tog objekta.

Lista kompozitnih operatora sa objašnjenjem značenja je na linku https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#assignment_operators.

Naredbe kontrole toka se koriste za upravljanje redosledom izvršavanja naredbi programa. U ovu grupu spadaju:

- **`return`** – Specificira vrednost koju će vratiti funkcija i vraća kontrolu programa onome ko je funkciju pozvao (na naredbu koja sledi poziv funkcije).
- **`break`** – Prekida izvršavanje tekuće petlje, `switch`, ili `label` naredbe i prenosi kontrolu toka programa na naredbu koja sledi terminiranu naredbu.
- **`continue`** – Prekida izvršavanje naredbi u tekućoj iteraciji tekuće ili labelirane petlje i nastavlja izvršavanje sledeće iteracije u petlji.
- **`throw`** - Generiše izuzetak koji je definisao korisnik.
- **`if...else`** – Izvršava deo koda u **`if`** bloku ako je specificirani uslov zadovoljen. Ako uslov nije zadovoljen, izvršava se deo koda u **`else`** bloku (**`else`** blok nije obavezan; ako ga nema, izvršava se deo koda koji neposredno sledi **`if`** blok).
- **`switch`** – Evaluira izraz u odnosu na vrednost `case` klauzule i izvršava naredbu asociranu sa tim `case`.
- **`try...catch`** – Markira blok naredbi koje će se izvršiti (`try`) i specificira odgovor ukoliko treba da se generiše izuzetak.

Naredbe iteriranja su specifična vrsta naredbi kontrole toka koje upravljaju repetitivnim izvršavanjem delova koda. U grupu naredbi iteriranja spadaju:

- **`do...while`** – Kreira petlju koja izvršava specificiranu naredbu/naredbe proveravajući testni uslov dok se testni uslov ne evaluira na `false`. Uslov se proverava **nakon izvršavanja** naredbe/naredbi što znači da će se specificirana naredba/naredbe izvršiti bar jednom.
- **`for`** – Kreira petlju koja se sastoji od tri opciona izraza zatvorena u male zagrade i razdvojena znakom tačka-zapeta iza čega se navodi naredba koja se izvršava u petlji.
- **`for...in`** – Iterira nad nabrojivim svojstvima objekta proizvoljnim redosledom. Naredba se može izvršiti za svako posebno svojstvo.
- **`for...of`** – Iterira nad iterabilnim objektima (uključujući nizove, nizolike objekte, iteratore i generatore), pozivajući prilagođenu „iteracionu kuku“ (deo koda izdvojen u funkciju) sa naredbama koje treba izvršiti za vrednost svakog različitog svojstva.

- **`for await...of`** – Iterira nad asinhronim iterabilnim objektima, nizolikim objektima, iteratorima i generatorima, pozivajući prilagođenu „iteracionu kuku“ sa naredbama koje treba izvršiti za vrednost svakog različitog svojstva.
- **`while`** - Kreira petlju koja izvršava specificiranu naredbu/naredbe proveravajući testni uslov dok se testni uslov ne evaluira na `true`. Uslov se proverava **pre izvršavanja** naredbe/naredbi.

Ostali konstrukti/naredbe su svi konstrukti koje nisu klasifikovani ni u jednu od napred pomenutih grupa. Imaju različite (uglavnom pomoćne) uloge u jeziku kao što su podrška kompatibilnosti jezika, debagovanje, obezbeđivanje dodatnih informacija za naredbe kontrole toka, deklaracija bloka, proširivanje lanca dosezanja i specifične izvršne funkcionalnosti. Ovde spadaju:

`empty` – Prazna naredba (prazan red) je nešto što će se tretirati kao sintaksno ispravno iako JavaScript sintaksa na tom mestu očekuje "normalnu" naredbu.

`block` – Konstrukt koji se koristi da grupiše nula ili više naredbi. Blok je ograničen parom vitičastih zagrada - **`{ }`**.

`Izrazna naredba (Expression statement)` – Izrazna naredba je jedan od “mutnijih” konstrukata u JavaScript-u. Po definiciji, to je **izraz na mestu na kome se u kodu očekuje naredba**. Izraz se evaluira i njegov rezultat se poništava, pa izrazna naredba ima smisla samo za izraze koji proizvode bočne efekte kao što je izvršavanje funkcije ili ažuriranje varijable. Sve naredbe dodele su izrazne naredbe. Više detalja i izraznoj naredbi možete naći adresi https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/Expression_statement. O izraznim naredbama biće još reči u odeljku **4.2.1. Varijable, izrazi, vrednosti**.

`debugger` – Poziva bilo koju raspoloživu funkcionalnost debagovanja. Ako funcionalnost debagovanja ne postoji, naredba ne proizvodi nikakav efekat.

`label` – Omogućuje da se naredbi dodeli identifikator na koji se može referisati pri korišćenju naredbi `break` ili `continue`.

`with`[2] - Proširuje lanac dosezanja naredbe. Ne preporučuje se korišćenje.

`"use strict"` ili **`'use strict'`** – Dugo je jezik JavaScript evoluirao bez obraćanja pažnje na pitanje kompatibilnosti. Jeziku su dodavane nove mogućnosti, a da se stare nisu menjale iako promene nisu uvek bile konzistentne. Godine 2009 pojavila se ECMAScript 5 (ES5) specifikacija koja je dodala nove mogućnosti jeziku i modifikovala neke do tada postojeće mogućnosti. Da bi se obezbedilo da stari kod radi, većina ovih novih mogućnosti je podrazumevano isključena. Da bi se mogle koristiti, treba ih eksplicitno omogućiti posebnom direktivom: "use strict".

Komentari su konstrukti koji ni na koji način ne utiču na izvršavanje i mogu da se nađu bilo gde u kodu. Mogu da budu ***jednolinijski*** ili ***višelinijski***. ***Jednolinijski*** komentari započinju dvostrukim slešom **`//`** i mogu da zauzmu celu liniju ili da slede u istoj liniji iza naredbe. ***Višelinijski*** komentari započinju stringom **`/*`** i završavaju se stringom **`*/`**. Sve što se nalazi između ova dva stringa (može da bude i više linija) je komentar. Ugnježdavanje komentara nije podržano. Postoji i treća vrsta komentara u JavaScript-u – ***hešbeng*** (*hashbang*) komentar koji započinje baš sekvencom **`#!`** (nije dozvoljen prazan prostor ispred). Ponaša se kao jednolinijski komentar – komentarom se smatra sadržaj koji se zauzima prostor do kraja linije. Dozvoljen je samo jedan takav komentar na isključivo na početku skripta ili modula.

U nastavku ćemo još govoriti o konstruktima jezika JavaScript u kontekstima u kojima se oni koriste.

[2] Povučena iz upotrebe.

3.2.1. Varijable, izrazi, vrednosti

U ovom odeljku objasnićemo koncepte varijabla, vrednost i izraz u Jeziku JavaScript.

3.2.1.1. Varijable u jeziku JavaScript

Varijabla je u programskom jeziku JavaScript apstraktna lokacija za skladištenje uparena sa pridruženim ***simboličkim imenom*** koja može da sadrži sve vrste vrednosti koje se javljaju u programu.

Za imenovanje varijabli u JavaScript-u važe sledeća ograničenja[3]:

- Prazna mesta nisu dozvoljena u imenu varijable.
- Hipenacije (znak -) nisu dozvoljene u imenu varijable.
- Ime varijable može da sadrži samo simbole slova, simbole cifara, simbol **$** ili simbol **_**.
- Prvi karakter u imenu može da bude samo slovo, znak **_**, ili znak **$**, odnosno prvi karakter ne sme da bude simbol cifre.
- Imena varijabli su osetljiva na velika i mala slova.
- Rezervisane reči nisu dozvoljene za imena varijabli.

Postupak pridruživanja simboličkog imena varijabli zove se ***vezivanje*** (eng. binding) ili ***deklaracija varijable***.

Postupak skladištenja vrednosti u varijablu zove se ***dodela vrednosti***. Kad se vrednost uskladišti u varijablu, njoj (vrednosti) se može pristupiti putem simboličkog imena pridruženog varijabli.

Varijabla može da postoji u JavaScript programu a da joj se ne dodeli nikakva vrednost.

Za razliku od imperativnih programskih jezika, gde se vrednostima generalno može pristupiti ili se one mogu promeniti u bilo kom trenutku, u (čistim) funkcionalnim jezicima varijable su vezane sa izrazima i zadržavaju jednu vrednost tokom celog svog životnog veka[4] zbog zahteva referencijalne transparentnosti. U programskom jeziku JavaScript podržana su oba pristupa, pri čemu ključna reč kojom se varijabla deklariše određuje koji pristup se na varijablu primenjuje.

Još jedna važna karakteristika varijable u jeziku JavaScript je njena ***oblast važenja*** (eng. scope). Oblast važenja određuje u kom delu koda (teksta) varijabla može da se koristi, popularno rečeno "odakle je vidljiva".

3.2.1.2. Ključne reči `const`, `let`, i `var`

U jeziku JavaScript postoje tri ključne reči kojima se deklaracija varijable i dodela vrednosti mogu izvršiti, i one se međusobno razlikuju.

Prva je ključna reč je **`const`**. Varijable koje su deklarisane pomoću ključne reči u JavaScript-u zovu se ***nepromenljive varijable*** ili ***imutabilne varijable***. Pri tome, precizno značenje termina ***imutabilna (nepromenljiva) varijabla*** je sledeće: IMUTABILNOJ/NEPROMENLJIVOJ varijabli VREDNOST se u programu MOŽE DODELITI SAMO JEDNOM. Evo jednostavnog primera:

```
const zdravo = 'Zdravo';
zdravo; // Zdravo
```

Kada se deklaracija varijable vrši korišćenjem ključne reči `const` obavezno je da se varijabli istovremeno dodeli vrednost. Razlog tome je što se u slučaju deklaracije varijable putem ključne reči

[3] Na adresi https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Lexical_grammar mogu se naći svi detalji o ovim ograničenjima, uključujući i listu rezervisanih reči.

[4] Životni vek varijable opisuje kada u toku izvršavanja programa varijabla sadrži (smislenu) vrednost. Životni vek varijable zove se i ***obim važenja*** (eng. extent) varijable

`const` **jednom dodeljena vrednost ne može menjati** pa je i logično da joj se vrednost dodeli pri deklaraciji[5].

Primer je sledeći kod:

```
const zdravo = 'Zdravo';
zdravo = 'Zdravo ponovo!';/* vraća grešku: Uncaught SyntaxError: Identifier 'zdravo' has
                                already been declared */
```

U delovima ove knjige koji se bave imutabilnošću vrednosti u JavaScript-u biće još dosta reči o deklaratoru `const` i nekim njegovim specifičnostima koje su relevantne za imutabilnost.

JavaScript ima još dve ključne reči za deklarisanje varijabli i dodelu vrednosti: **`var`** i **`let`** . Deklaracija `var` potiče iz ranije specifikacije jezika a deklaracija `let` je, u stvari, "poboljšana" deklaracija `var`.

Ključne reči `var` i `let`, za razliku od ključne reči `const`, obe deklarišu **varijablu koja je mutabilna** (može joj se više puta dodeljivati vrednost). Dakle, sledeća dva snipeta su legalan JavaScript kod:

```
let zdravo = 'Zdravo';
zdravo = 'Zdravo ponovo!';
```

```
var zdravo = 'Zdravo';
zdravo = 'Zdravo ponovo!';
```

Ono po čemu se `var` i `let` razlikuju je *dosezanje*.

Varijable koje su deklarisane ključnom reči `var` imaju doseg funkcije u kojoj su deklarisane ili globalni doseg, dok doseg `let` i `const` varijabli može da bude globalni, funkcijski ili neposredni zatvarajući blok označen sa { }. Tabela 4.1 daje sažet pregled osobina varijabli deklarisanih ključnim rečima const, let i var.

Tabela 4.1 Deklaracija varijabli u JavaScript-u

Ključna reč	Imutabilnost	Globalni doseg	Funkcijski doseg	Blokovski doseg
`var`	ne	da	da	ne
`let`	ne	da	da	da
`const`	da	da	da	da

Evo primera u kome je varijabla `baz` deklarisana u posebnom bloku ključnom reči `let`:

```
function run(){ // ovde počinje blok funkcije run()
var foo ="Foo";
let bar ="Bar";

  console.log(foo, bar);// vraća: Foo Bar

{ // ovde počinje NOVI blok
let baz ="Bazz"; // varijabla baz deklarisana je u NOVOM bloku
    console.log(baz); // vraća: Bazz jer je baz vidljiva u ovom bloku
}// Ovde se završava NOVI blok

  console.log(baz);/* vraća: Reference Error: baz is not defined zato što
```

[5]To ne važi u potpunosti za kompozitne tipove kao što su `Array` i `Object`.

```
                    se varijabla baz više ne vidi – ona je ovde van
                    bloka NOVI u kome je varijabla baz deklarisana i gde
                    je vidljiva */

} // ovde se završava blok funkcije run()

run(); // Ovde se poziva funkcija run()
```

Ovaj kod vraća grešku jer linija referiše varijablu baz koja jeste deklarisana u funkciji run() ali je deklarisana u posebnom bloku i u tom bloku je vidljiva a van tog bloka nije vidljiva.

Kod u kome je varijabla baz deklarisana pomoću ključne reči var ne vraća grešku:

```
function run(){
var foo = "Foo";
let bar = "Bar";

  console.log(foo, bar);// vraća: Foo Bar

{ // Ovde počinje NOVI blok
var baz = "Bazz";
    console.log(baz); // vraća: Bazz
} // Ovde se završava NOVI blok

  console.log(baz);/* vraća: Bazz iako je varijabla baz definisana u
                    posebnom bloku */
}

run(); // ovde se poziva run()
```

Kako je deklaracija var nasleđe prethodnih verzija jezika a deklaracija let može da uradi sve što može var, preporuka je da se u novom kodu deklaracija var ne koristi.

3.2.2. Izrazi

Izraz je svaki deo koda koji se evaluira na vrednost; kao formula - zamenu se "opšti" brojevi "stvarnim" vrednostima i „sračuna" se rezultat, šta god on bio. Sastoji se od dve vrste elemenata: *operatora* i *operanda*.
Operatori se izvršavaju nad operandima i mogu da zahtevaju različit broj operanada (jedan ili više): n.pr., sabiranje je binarni operator sa dva operanda – sabirka, logička negacija je unarni operator sa jednim operandom. Operandi mogu da budu literalne (doslovne) vrednosti ili varijable a operatori koji se na njih mogu primenjivati zavise od tipa operanda/operanada.

Izrazi se mogu pojavljivati u JavaScript programu na svakom mestu na kome se očekuje vrednost, na primer kao delovi drugih izraza, kao argumenti pri pozivu funkcije, itd. Razlikuju se sledeći izrazi u jeziku JavaScript:

- **Aritmetički izrazi** se evaluiraju na brojčanu vrednost:

```
7;          // brojčana vrednost 7
9 + 1;      // brojčana vrednost 10
7 - 2;      // brojčana vrednost 5
```

- **String izrazi** se evaluiraju na string:

```
'Zdravo!';            // String: Zdravo!
'Zdravo,' + 'narode!';// String: Zdravo, narode!
```

- **Logički izrazi** se evaluiraju na jednu od dve vrednosti `true` ili `false`.

```
10 > 9;               // rezultat evaluacije: true
10 < 20;              // rezultat evaluacije: false
true;                 // rezultat evaluacije: true
a === 20 && b === 30; // rezultat evaluacije: true ili false zavisno od
a i b
```

- **Primarni izrazi** su samostalni izrazi kao što su literalne vrednosti, određene ključne reči i vrednosti varijabli:

```
'hello world'; // String literal
23;            // Numerički literal
true;          // Logička vrednost true
sum;           // Vrednost varijable sum
this;          // Ključna reč (pokazivač) koja se evaluira na tekući
               // objekat
```

- **Izrazi dodele** su izrazi koji koriste znak **=** za dodelu vrednosti varijabli:

```
Prosek = 55;

let b = (a = 1); /* Ovde se prvi izraz dodele (a = 1) evaluira na
vrednost 1 koja se drugim izrazom dodele b = (a = 1) dodeljuje
varijabli b. Ključna reč let nije deo izraza.*/
```

- **l-vrednosti** (engl. *lvalues*) su izrazi koji mogu da se pojave na levoj strani izraza dodele i obuhvataju identifikatore (varijable, svojstva objekata, elemente nizova):

```
i = 10;           // varijabla i
total = 0;        // varijabla total

let obj = {};     // obj - prazan objekat bez svojstava
obj.x = 10;       // svojstvo x objekta obj.

array[0] = 20;     // prvi element niza array
array[1] = 'hello';// drugi element niza array
```

- **Izrazi sa bočnim efektima (Izrazne naredbe)** su izrazi čija evaluacija dovodi do promene ili bočnog efekta poput postavljanja ili modifikovanja vrednosti varijable putem operatora dodele **=**, inkrementiranja/dekrementiranja vrednosti varijable, funkcijskog poziva. Sledi nekoliko primera:

```
sum = 20;         // Varijabli sum dodeljuje se vrednost 20

sum++;            // uvećava vrednost sum za 1

let a = 10;      // Varijabli a dodeljuje se vrednost 10


function modify(x){
   return x *= 10;
}

modify(a);      // poziv funkcije modify() modifikuje vrednost a na 100
```

3.2.2.1. Operatori poređenja i ternarni izrazi

Operatori poređenja služe da se utvrdi da li je vrednost[6] izraza iskazanog zadatim operatorom nad zadatim operandima istinita ili nije istinita. Rezultat operacije je uvek vrednost `true` ili `false`.

Za proveru ***jednakosti*** vrednosti se u JavaScript-u koriste dva operatora: operatora ***striktne jednakosti*** (*triple equals*) čiji je simbol trostruki znak jednakosti (**===**) i operatora ***nestriktne jednakosti*** čiji je simbol dvostruki znak jednakosti (**==**).

Operator striktne jednakosti zahteva da oba operanda budu istog tipa. Vraća vrednost `true` samo ako su i tip i vrednost jednaki, a ako to nije zadovoljeno vraća vrednost `false`:

```
3 + 1 === 4; /* true – oba tipa su number i vrednosti su jednake */

3 + 1 === '4'; /* false – 3 i 1 su tipa number (pa je i 3+1 tipa number),
                  a '4' je tipa string */
```

Operator nestriktne jednakosti pre poređenja vrednosti vrši implicitnu konverziju tipa. Vraća vrednost `true` ako su konvertovane vrednosti jednake:

```
3 + 1 == 4; /* true – oba tipa su number i vrednosti su jednake */

3 + 1 == '4'; /* true – 3 i 1 su tipa number (pa je i 3+1 tipa number);
                 pošto je '4' tipa string, vrši se konverzija number u
                 string */
```

Za operator **==** postoje validni slučajevi korišćenja, ali se generalno preporučuje korišćenje operatora **===**.

Drugi operatori poređenja (komparatori) uključuju:

- **>** veće od
- **<** manje od
- **>=** veće ili jednako od
- **<=** manje ili jednako od
- **!=** nije jednako
- **!==** nije striktno jednako
- **&&** Logičko I
- **||** Logičko ILI

Ternarni izraz je izraz koji omogućuje da se postavi pitanje korišćenjem komparatora i evaluira se na različite odgovore u zavisnosti od istinitosti izraza poređenja.

Sintaksa izraza poređenja u JavaScript-u je:

```
uslov ? izraz1 : izraz2
```

Ovde je `uslov` izraz poređenja, `izraz1` je izraz koji se evaluira ako je rezultat poređenja u uslovu `true`, a `izraz2` je izraz koji se evaluira ako je rezultat poređenja u uslovu `false`.

Primeri ternarnih izraza su:

```
14 - 7 === 7 ? 'Jeste!' : 'Nije.'; // vratiće Jeste!
14 - 7 === 53 ? 'Jeste!' : 'Nije.'; // vratiće Nije.
```

Za malo više detalja o poređenjima i ternarnim izrazima čitalac se upućuje na izvor https://javascript.info/comparison. Za kompletan pregled izraza i operatora u JavaScript-u čitalac se upućuje na link https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators.

[6] Kao što se vidi iz primera koji slede, vrednost može da bude i izraz.

3.2.3. Vrednosti u jeziku JavaScript

Kao i u drugim programskim jezicima, i u JavaScript-u vrednosti se svrstavaju u grupe koje se nazivaju ***tipovi***. JavaScript podržava većinu vrsta vrednosti (tipova) koje se očekuju od programskih jezika opšte namene: logičke, numeričke, stringove, specijalne vrednosti. Takođe, kao i drugi programski jezici, JavaScript ima i klasifikaciju tipova.

U jeziku JavaScript, tipovi se klasifikuju kao ***primitivni*** i ***neprimitivni***.

JavaScript ima ukupno **osam tipova** od čega **sedam primitivnih** tipova i **jedan neprimitivni** tip.

U nastavku ovog poglavlja ćemo detaljnije objasniti primitivne tipove podataka u JavaScriptu.

Neprimitivni tip u jeziku JavaScript je tip `object` namenjen za složenije strukture podataka. Tipu `object` biće posvećeno posebno poglavlje zbog izuzetno važne uloge koju ima u jeziku JavaScript.

3.2.3.1. Osnovno o neprimitivnom tipu `object`

Iako može da deluje pomalo naopačke, pre no što počnemo da govorimo o primitivnim tipovima u JavaScript-u, uvešćemo pojam objekta u vrlo pojednostavljenoj formi. Razlog tome je što se i sa najčešće korišćenim primitivnim tipovim radi tako što se oni "zamotaju" u objekat i sa njima se, doduše pojednostavljeno, u nekim situacijama radi kao sa objektima.

U JavaScript-u se objekat može posmatrati kao kolekcija svojstava gde su svojstva objekta parovi (***ključ***, ***vrednost***). Vrednostima svojstava pristupa se putem ključeva koji su ili tipa `string` ili tipa `symbol`. Jednostavno rečeno: ključ je ime svojstva kome se pristupa putem imena kao i svakom drugom imenovanom entitetu. Vrednosti svojstva mogu biti vrednosti bilo kog tipa podržanog u jeziku JavaScript, uključujući i druge objekte i funkcije, što omogućava izgradnju složenih struktura podataka i definisanje podtipova. Posebno, **svojstvo** objekta **čija je vrednost funkcija** naziva se ***metoda*** objekta.

Objekti su kompozitni tipovi podataka koji omogućuju rad sa više vrednosti – kolekcijama vrednosti predstavljenih svojstvima. Objekti predstavljaju ***neuređene*** kolekcije. To znači da nad elementima kolekcije predstavljene objektom ne postoji nikakva relacija uređenja – na primer, ne zna se koje je svojstvo objekta "ispred" nekog drugog svojstva tog istog objekta.

Objekti se u JavaScript-u javljaju u dva sintaktička oblika od kojih je jedan (najčešće korišćen) *deklarativni* (*literalni*) oblik koji izgleda ovako:

```
let imeObj = {
        kljuc1: vrednost1,
        kljuc2: vrednost2,
        // ...
};
```

U ovom primeru, `imeObj` je ime objekta, dok su parovi (`kljuc1`, `vrednost1`) i (`kljuc2`, `vrednost2`) svojstva objekta. Parovi (ključ, vrednost) navode se u vitičastim zagradama međusobno razdvojeni zarezima. Pri tome se ključ i vrednost razdvajaju dvotačkom. Objekat može da ima proizvoljno mnogo svojstava.

Vrednosti svojstva se pristupa putem njoj odgovarajućeg ključa. Za to postoje dve sintakse. Jedna, koju ćemo koristit u ovom odeljku, zove se ***tačkasta sintaksa*** (skraćeno ***dot sintaksa***) i izgleda ovako:

```
imeObj.kljuc1 = vrednost1
```

Dakle, tačkasta sintaksa nalaže da se navede ime objekta iz koga sledi znak tačka (`.`) i iza tačke sledi ključ svojstva. Sve ove komponente su obavezne.

U nastavku je celo poglavlje posvećeno objektima. U njemu su detaljno objašnjeni pojedinačni objekti kao i načini povezivanja objekata.

3.2.3.2. Primitivni tipovi

U ovom odeljku ćemo detaljnije objasniti primitivne tipove podataka u JavaScriptu. Pri tome ćemo se prvo pozabaviti samim vrednostima a zatim operacijama nad tim vrednostima.

Klasa ***primitivnih tipova*** obuhvata sledeće tipove:

- `null`,
- `undefined`,
- `symbol`.
- `number`,
- `bigInt`,
- `string`,
- `boolean`,

Tipovi `null` i `undefined` su "specijalni": tip `null` se koristi da predstavi vrednosti koje se ne znaju, a tip `undefined` da predstavi situaciju u kojoj nije dodeljena nikakva vrednost.

Konačno, tip `symbol` je takođe "specijalan": koristi se za predstavljanje jednoznačnih identifikatora.

Tipovi `number` i `bigint` namenjeni su za rad sa brojevnim vrednostima (razlomljene i celobrojne), tip `string` za rad sa tekstualnim vrednostima (pojedinačni karakter ili sekvenca karaktera), a tip `boolean` za rad sa istinitosnim vrednostima (`true`/`false`).

U nastavku ćemo detaljno prikazati svaki od ovih tipova ali ćemo pre toga objasniti i jednu specifičnost JavaScript-a koja se odnosi na interni način rukovanja vrednostima primitivnih tipova.

Omotački objekti primitivnih tipova

Svi primitivni tipovi u JavaScript-u, osim `null` i `undefined`, imaju odgovarajuće tipove omotačkih objekata. Zarad operativnijeg imenovanja, za imena primitivnih tipova i odgovarajućih tipova omotačkih objekata koriste se iste reči, a da bi se međusobno razlikovali ime primitivnog tipa započinje malim slovom a ime njemu odgovarajućeg tipa omotačkog objekta započinje velikim slovom (na primer, primitivni tip `number` i njemu odgovarajući tip omotačkog objekta `Number`).

Svrha omotačkih objekata je da obezbede korisne metode za rad sa primitivnim vrednostima. Na primer, omotački objekat `Number` obezbeđuje primitivnom tipu `number` metode kao što je `toExponential()` kojom se brojčana vrednost prevodi u eksponencijalni format. Za korišćenje tih metoda JavaScript implementira sledeći mehanizam: kada se pristupi svojstvu primitivne vrednosti sa ciljem primene metode, JavaScript automatski omotava vrednost u odgovarajući omotački objekat u kome su mu na raspolaganju sve ugrađene metode kao svojstva tog objekta i pristupa željenom svojstvu tog objekta (na primer, metodi `toExponential()`). Sledi primer:

```
let a = 123
console.log('ja sam a tipa: ', typeof(a), ' i vrednost mi je: ', a)

console.log('ja sam a tipa: ', typeof(a), ' i vrednost mi je: ',
a.toExponential())
```

koji će na konzoli da ispiše:
ja sam a tipa: number i vrednost mi je: 123
ja sam a tipa: number i vrednost mi je: 1.23e+2

Sintaksa ne dozvoljava da se koriste doslovne primitivne vrednosti, odnosno ne možete pisati `123.toExponential`.

Pokušaj pristupa svojstvu primitivnih vrednosti `null` i `undefined` za koje ne postoje omotački objekti rezultuje izuzetkom `TypeError`.

Spiskovi metoda omotačkih objekat za primitivne tipove dostupni su na linku https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/.

3.2.3.3. Tip `null` i tip `undefined`

U JavaScript-u null nije referenca na nepostojeći objekat (null-pointer) kao u nekim drugim jezicima. To je specijalna vrednost koja ima značenje "ništa", "prazno" ili "nepoznata vrednost". Sledeća linija koda kaže da se identifikator `pojmaNemam` **ne zna**:

```
let pojmaNemam = null;
pojmaNemam; // null
```

U JavaScript-u postoji i poseban tip `undefined`. Značenje mu je "vrednost nije dodeljena". Varijabla koja je deklarisana ali joj nije eksplicitno dodeljena vrednost ima vrednost `undefined`; vrednost `undefined` može se dodeliti i eksplicitno:

```
let nekaVarijabla;
nekaVarijabla; // "undefined"

let nekaVarijabla1 = 100;
nekaVarijabla1; // 100

nekaVarijabla1 = undefined;
nekaVarijabla1; // "undefined"
```

3.2.3.4. Tip `symbol`

Tip `symbol` predstavlja jednoznačni identifikator. Vrednost tipa `symbol` kreira se funkcijom `Symbol()` a opciono mu se može dodeliti i "ime", u stvari opis koji se može koristiti za debagovanje. U sledećem primeru `id1`, `id2` i `id3` su identifikatori vrednosti tipa `symbol`, a string `"imeSimbola"` je opcioni opis. Kao što se iz primera vidi, isti opis ne znači da su u pitanju iste vrednosti tipa `symbol`. U stvari, vrednost tipa `symbol` jednaka je isključivo sama sebi.

```
let id1 = Symbol("imeSimbola");
let id2 = Symbol("imeSimbola");
let id3 = Symbol("imeSimbola");

id1 == id2; // false
id1 == id3; // false
id2 == id3; // false
id1 == id1; // true
```

Ovde ćemo spomenuti još jednu važnu osobinu tipa `symbol`: Za razliku od većine primitivnih tipova u JavaScript-u, tip `symbol` se ne konvertuje implicitno u `string`. Ova karakteristika je proistekla iz činjenice da su u jeziku JavaScript stringovi i simboli fundamentalno različiti do mere da se ne smeju slučajno konvertovati jedan u drugi. Za konverziju u string tip `symbol` ima metodu `toString()` a za prikazivanje opisa metodu `description` koju smo u prethodnom primeru videli. Sledi primer koji ilustruje korišćenje metode `toString()` :

```
let id = Symbol("id");
alert(id.toString()); // Symbol(id)
alert(id.description); // id
```

Susretaćemo se još sa tipom `symbol` kada budemo govorili o objektima i tada ćemo ih detaljnije objasniti u kontekstu primene.

3.2.3.4.1. Brojevni tipovi

Ovde spadaju sve vrste brojeva: celobrojni, razlomljeni decimalni (tekući zarez). Pored "regularnih brojeva", postoje i takozvane "specijalne brojevne vrednosti" koje takođe pripadaju tipu number: `Infinity`, `-Infinity` i `NaN`.

3.2.3.4.1.1. Tip number

Predstavljanje vrednosti tipa number

Vrednosti tipa `number` su 64-bitski binarni brojevi dvostruke preciznosti u skladu sa standardom IEEE 754[7].
Celobrojne vrednosti su ograničene sa $\pm 2^{53-1}$ a razlomljene vrednosti mogu da budu iz intervala $[\pm 2^{-1074}, \pm 2^{1024}]$.

Kao što se može zapaziti, JavaScript ne razlikuje celobrojne vrednosti od razlomljenih – za obe se koristi isti tip. JavaScript, u stvari, ima samo razlomljene brojeve a celi brojevi se od razlomljenih razlikuju u sintaksi.

Sintaksa celih brojeva

Za predstavljanje celih brojeva na raspolaganju su četiri sintakse: **decimalna**, **oktalna**[8], **heksadecimalna** i **binarna**.

Decimalna sintaksa za cele brojeve je "direktna"; brojevi se zapisuju kao što biste ih i "ručno" pisali: niz decimalnih cifara kojima može da prethodi i znak **–** za negativne vrednosti, a dozvoljen je na početku i znak **+**. Ilustracija je sledeći primer:

```
var intNum1 = 105;          //doslovni format za celobrojnu vrednost 105
console.log(intNum1);
var intNum2 = -105;         //doslovni format za celobrojnu vrednost -105
console.log(intNum1);
var intNum3 = +105;         //doslovni format za celobrojnu vrednost 105
console.log(intNum3);
```

Rezultat izvršavanja ovog koda je sledeći ispis na konzoli:
105
-105
105

Oktalna sintaksa nalaže da zapis broja započne znakom 0 i iza njega se navodi niz oktalnih cifara (0, 1, 2, 3, 4, 5, 6, 7).Znak **–**na prvoj poziciji koristi se da označi negativnu vrednost. Sledi primer:

```
var octalNum1 = 070;     // oktalni zapis za vrednost 56₁₀
console.log(octalNum1);  // ispisaće 56

var octalNum2 = 079;    // invalidan oktalni zapis-interpretira se kao 79₁₀
console.log(octalNum2); // ispisaće 79

var octalNum3 = -021;    // oktalni zapis za vrednost  -17₁₀
console.log(octalNum3);  // ispisaće -17
```

Heksadecimalna sintaksa nalaže da zapis broja započne sekvencom **0x** i ili **0X** iza koje se navodi niz heksadecimalnih cifara (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D. E, F). I ovde se znak minus (**–)** na prvoj poziciji koristi da označi negativnu vrednost. Sledi primer:

[7] Osnovne informacije o standardu IEEE 754 mogu se naći na adresi https://en.wikipedia.org/wiki/IEEE_754

[8]Oktalna sintaksa nije legalna za striktni režim. Striktni režim se uspostavlja navođenjem komande 'use strict'

```
var hexNum1 = 0xA;            //heksadecimalni zapis za 10₁₀
console.log(hexNum1);         //ispisaće 10
var hexNum2 = 0x1f;           //heksadecimalni zapis za 31₁₀
console.log(hexNum2);         //ispisaće 31
var hexNum3 = -0X2e;          //heksadecimalni zapis za -46₁₀
console.log(hexNum3);         //ispisaće -46
```

Binarna sintaksa

Za predstavljanje binarnih brojeva koristi se sintaksa koja nalaže da se ispred broja navede sekvenca `0b` ili `0B`. Za reprezentaciju broja koriste se sekvenca karaktera `0` i `1`. Korišćenje bilo kog drugog karaktera usekvenci broja (uključujući i blanko) rezultuje sintaksnom greškom. Evo primera:

```
console.log(0b10000000000000000000000000000000) // ispisaće: 2147483648
console.log(0b01111111100000000000000000000000) // ispisaće: 2139095040
console.log(0B00000000011111111111111111111111) // ispisaće: 8388607

/* Sledeća linija vratiće grešku Uncaught SyntaxError: Invalid or unexpected
   token */
console.log(0b10000000000000000000000000000005)
/* Sledeće linije vratiće grešku Uncaught SyntaxError: missing ) after
argument list */
console.log(0b100  00000000000000000000000000)
console.log(0b1.00000000000000000000000000000)
```

Sintaksa razlomljenih brojeva

Za razlomljene brojeve podržane su dve sintakse: **sintaksa sa decimalnom tačkom** i **eksponencijalna sintaksa**.

Sintaksa sa decimalnom tačkom. Za zapisivanje vrednosti u formatu sa decimalnom tačkom, obavezno je navođenje decimalne tačke i najmanje jedne cifre iza decimalne tačke. Cifra ispred decimalne tačke nije obavezna, ali se preporučuje. I ovde se znak minus na prvoj poziciji koristi da označi negativnu vrednost. Evo primera:

```
var floatNum1 = 1.1;
console.log(floatNum1); // 1.1
var floatNum2 = 0.1;
console.log(floatNum2); // 0.1
var floatNum3 = .1;     /* 0.1 - validno, ali se ne preporučuje (zbog
                           čitljivosti koda) */
console.log(floatNum3);
var floatNum4 = -21.1;
console.log(floatNum4); // -21.1
```

Eksponencijalna sintaksa. Za zapisivanje vrednosti u eksponencijalnom obliku, obavezno je navođenje broja (celog ili razlomljenog u decimalnom zapisu) iza koga sledi znak e ili E i iza tog znaka celobrojna vrednost. Ova celobrojna vrednost predstavlja stepen osnove 10 sa kojim se množi broj naveden ispred znakae ili E. I ovde se znak **–** na prvoj poziciji koristi da označi negativnu vrednost zapisanog broja. Znak **–**može da se navede između znaka e ili E broja koji sledi iza znaka i tada on označava negativan eksponent. Evo i primera:

```
var floatNum = 3.125e7;    // jednako je sa 31250000
console.log(floatNum);     // 31250000
var floatNum = -3.125e-2;  // jednako je sa -0.03125
console.log(floatNum);     // -0.03125
```

Greška zaokruživanja. Pri aritmetičkim operacijama na računaru vrlo često se javljaju greške. Ilustrovaćemo to na primeru:

```
var a = 0.1;
var b = 0.2;
console.log(a + b);
if (a + b == 0.3){
    console.log("Dobili ste 0.3.");
}

a = 0.05;
b = 0.25;
console.log(a + b);
if (a + b == 0.3){
    console.log("Dobili ste 0.3.");
}
```

Očekujemo da rezultat izvršavanja koda bude sledeći ispis
0.3
Dobili ste 0.3.
0.3
Dobili ste 0.3.

Međutim, izvršavanjem koda dobija se ispis:

0.30000000000000004
0.3
Dobili ste 0.3.

To znači da je pri sabiranju vrednosti 0.1 i 0.2 došlo do greške.

Važna napomena: U suštini, gotovo svi brojevi u računaru su "pogrešni". Jedan uzrok tome je činjenica da je memorija računara sačinjena od memorijskih lokacija koje su konačne dužine – mogu da skladište konačan (zadati) broj bita. Pored toga, greške u računanjima mogu da nastanu i zbog samih metoda (na primer sračunavanje vrednosti funkcije korišćenjem konačnih redova za funkcije koje se predstavljaju beskonačnim redom). Računskim greškama (i računarskim i neračunarskim) bavi se posebna oblast matematike i računarstva koja se zove analiza grešaka. U suštini, rezultati koji se dobiju numeričkim sračunavanjima su upotrebljivi samo ako se može proceniti njihova greška.

`Infinity` predstavlja matematički koncept beskonačnosti ∞.To je specijalna vrednost koja je veća od najvećeg dozvoljenog broja u JavaScript-u.

Već smo spomenuli da su i celi i razlomljeni brojevi koji mogu da se predstave u JavaScript-u ograničenog opsega. Opsezi vrednosti identifikovani su u JavaScript-u putem četiri specijalne vrednosti:

- `Number.MIN_VALUE` predstavlja najmanji dozvoljen broj uJavaScript-u.
- `Number.MAX_VALUE` predstavlja najveći dozvoljen broj u JavaScript-u.
- `Number.NEGATIVE_INFINITY` predstavlja -∞.
- `Number.POSITIVE_INFINITY` predstavlja +∞.

NaN (**N**ot**a**N**N**umber) služi da predstavi grešku u računanju. Evo i primera za NaN:

```
console.log ('age:', age);
console.log ('2*age:', 2*age);
```

Rezultat izvršavanja je ispis:

age: undefined
2*age: NaN

3.2.3.4.1.2. Tip bigInt

Predstavljanje vrednosti tipa `bigInt`

Iako je za većinu primena opseg celih brojeva $\pm 2^{53-1}$(9007199254740991) dovoljan, ima situacija (na primer, kriptografija, vremenske oznake preciznosti mikrosekunde, ...) koje zahtevaju veći opseg. Zbog toga je naknadno (ne tako davno) jeziku JavaScript dodat tip `bigInt` za predstavljanje celih brojeva proizvoljne (konačne) dužine.

U stvari, tip `number` formalno može da skladišti i veće cele brojeve (do $1{,}7976931348623157 * 10^{308}$), ali izvan bezbednog opsega celih brojeva $\pm 2^{53-1}$ javiće se greška, jer se ne mogu predstaviti sve cifre u fiksnoj 64-bitnoj memoriji koja se koristi za skladištenje tipa `number`. Rezultat je da se skladišti približna vrednost kao u sledećem primeru gde dodavanje različitih vrednosti daje isti rezultat:

```
console.log(9007199254740991 + 1000000); // 9007199254740992
console.log(9007199254740991 + 1);       // 9007199254740992
```

`bigInt` vrednost se predstavlja tako što se iza poslednje cifre u zapisu broja doda slovo **n**:

```
constbigInt = 12345678901234567890123456789012345678901234567890n;
DbigInt = 2n*bigInt
console.log (DbigInt)
```

Izvršavanjem koda ispisuje se na konzolu:
246913578024691357802469135780246913578024691357802469135780n

Aritmetičke operacije nad `bigInt` vrednostima dozvoljene su samo AKO SU SVI OPERANDI tipa `bigInt`.

Ako se u prethodnom kodu množenje zapiše kao `DbigInt = 2*bigInt` (bez n iza operanda2) , vraća se poruka o grešci: *UncaughtTypeError: Cannot mix BigInt and other types, use explicit conversions*

3.2.3.4.2. Operacije nad brojevnim vrednostima

Operacije koje su uobičajene sa brojevima su aritmetičke operacije i operacije poređenja vrednosti. To su binarne operacije – operacije koje se izvršavaju nad dva operanda između kojih se navodi simbol operatora. JavaScript podržava sledeće aritmetičke operacije:

- Sabiranje (simbol operatora **+**),
- Oduzimanje (simbol operatora **-**),
- Množenje (simbol operatora *****),
- Deljenje (simbol operatora **/**),
- Stepenovanje (simbol operatora ******),
- Ostatak celobrojnog deljenja (simbol operatora **%**).

Operacije poređenja vrednosti su takođe binarne. Kao rezultat uvek vraćaju vrednost `true` ako je uslov zadovoljen a vrednost `false` ako uslov nije zadovoljen. Operacije poređenja brojeva koje podržava JavaScript su:

- Veće od (simbol operatora **>**),
- Manje od (simbol operatora **<**).
- Veće ili jednako od (simbol operatora **>=**),
- Manje ili jednako od (simbol operatora **<=**),
- Jednako (simbol operatora **==**),
- Nije jednako (simbol operatora **!=**) .

Pored navedenih binarnih operacija sa brojevima, JavaScript podržava i određene unarne operacije – operacije koje se izvršavaju nad jednim operandom koji se navodi iza ili ispred operatora. Pri tome,

mesto navođenja operatora i operanda može da utiče na rezultat operacije. JavaScript podržava sledeće unarne operacije nad brojevima:

- Inkrementiranje (simbol operatora **++**, operator se navodi **ispred operanda** ili **iza operanda**)
- Dekrementiranje (simbol operatora **--**, operator se navodi **ispred operanda** ili **iza operanda**)

Sledi fragment koda koji daje neke primere rada sa brojevima.

```
/* u ovom primeru koriste se i različite operacije nad brojevima:
   •  +  operacija sabiranja
   •  !=operacija poređenja vrednosti sa značenjem nije jednako
   •  <= operacija poređenja sa značenjem manje ili jednako
   •  >= operacija poređenja sa značenjem veće ili jednako
   •  Math.sqrt() operacija korenovanja
   •   */
var result = Number.MAX_VALUE + Number.MAX_VALUE;
console.log("Number.MAX_VALUE + Number.MAX_VALUE=", result);

console.log("Number.MAX_VALUE=", Number.MAX_VALUE);
console.log("Number.MIN_VALUE=", Number.MIN_VALUE);
console.log("Number.NEGATIVE_INFINITY=", Number.NEGATIVE_INFINITY);
console.log("Number.POSITIVE_INFINITY =", Number.POSITIVE_INFINITY);

if((Math.sqrt(-2)) !=Number.NEGATIVE_INFINITY_){
    console.log(Math.sqrt(-2)," nije jednako sa NEGATIVE_INFINITY");
}
else {
    console.log(Math.sqrt(-2),"je jednako sa NEGATIVE_INFINITY");
}

if((Math.exp(999)) <=Number.POSITIVE_INFINITY_){
   console.log(Math.exp(999), "nije veće od POSITIVE_INFINITY");
}
else {
   console.log(Math.exp(999), " je manje ili jednako POSITIVE_INFINITY");
}

if((99999*99999) <= Number.MAX_VALUE){
   console.log("Boj 99999*99999 nije veći od maksimalne vrednosti");
}
if((0.0000000001) >= Number.MIN_VALUE){
   console.log("Broj 0.0000000001 nije manji od minimalne vrednosti");
}
```

Rezultat izvršavanja je ispis na konzoli:

Number.MAX_VALUE + Number.MAX_VALUE= Infinity
Number.MAX_VALUE= 1.7976931348623157e+308
Number.MIN_VALUE= 5e-324
Number.NEGATIVE_INFINITY = -Infinity
Number.POSITIVE_INFINITY = Infinity
NaN ' nije jednako sa NEGATIVE_INFINITY'
Infinity ' je manje ili jednako POSITIVE_INFINITY'

Boj 99999*99999 nije veći od maksimalne vrednosti
Broj 0.0000000001 nije manji od minimalne vrednosti

3.2.3.5. Tip **`string`**

Tip `string` u JavaScript-u je jedini tip koji se koristi za tekstualne podatke - podatke koji se sastoje od karaktera. String može da ima nula ili više karaktera, odnosno NE POSTOJI POSEBAN TIP ZA POJEDINAČNI KARAKTER.

3.2.3.5.1. Predstavljanje vrednosti tipa **`string`**

Interno, stringovi se predstavljaju u UTF-16[9] formatu.

Sintaksa za tip `string` u JavaScriptu nalaže da se string vrednost navede između navodnika. Pri tome se kao navodnici mogu koristiti dvostruki navodnici (**""**), jednostruki navodnici (**''**) i ”bektik” navodnici (**``**). Jednostruki i dvostruki navodnici se ravnopravno koriste, dok ”bektik” navodnici omogućuju da se u string ”upakuje” proizvoljan izraz tako što će se izraz obmotati sa **${…}**. Evo primera:

```
let single = 'single-quoted';
let double = "double-quoted";
let backticks = `backticks`;
function sum(a, b) {
return a + b;
}

console.log(single, double, backticks, `1 + 2 = ${sum(1, 2)}.`);
```

Rezultat izvršavanja je ispis:
single-quoted double-quoted backticks 1 + 2 = 3.

3.2.3.5.1.1. Specijalni karakteri

Specijalni karakteri su sekvence karaktera koji u jeziku imaju posebno značenje. Najčešće korišćene su sekvence koje omogućuju da se kao sastavni deo stringa navedu i karakteri kojima se omeđuje sam string (različite vrste navodnika), i sekvence kojima se reguliše horizontalno i vertikalno pozicioniranje (tab i prelazak u novi red). Sintaksa nalaže da se ispred svakog specijalnog karaktera navede karakter obrnuta kosa crta - beksleš (****)[10].

Tabela 1 sadrži listu specijalnih karaktera.

Tabela 1. Specijalni karakteri

Karakter	Opis
\n	Novi red
\r	Takođe novi red; zbog kompatibilnosti sa OS Windows
\', \", \`	Navodnici
\t	Tab
\b, \f, \v	Backspace, FormFeed, Vertical Tab; nasleđe iz prošlosti, ne koristi se više
\\	Obrnuta kosa crta (beksleš)

[9] https://en.wikipedia.org/wiki/UTF-16

[10]Karakter sa ovakvom ulogom naziva se u žargonu **karakter izbegavanja** (engl. escape character)

Na primer, moguće je kreirati nizove sa više redova sa jednostrukim i dvostrukim navodnicima korišćenjem takozvanog „znaka za novi red“ (\n) koji označava prelom reda:

```
let studentiLista = "Studenti:\n * Jovan\n * Petar\n * Marica";
console.log(studentiLista);
```

Rezultat je ispis:
Studenti:
*** Jovan**
*** Petar**
*** Marica**

3.2.3.5.1.2. Regularni izrazi (regeksi) u jeziku JavaScript

Regularni izraz/regeks (***regex*** ili ***regexp***), je niz znakova koji specificira obrazac podudaranja u tekstu. Od samog početka (a pojavili su se 50-tih godina prošlog veka) su takvi obrasci našli primenu u editorima teksta i za leksičku analizu u kompajlerima. Danas su široko podržani u programskim jezicima, programima za obradu teksta (posebno lekserima), naprednim uređivačima teksta i još nekim drugim programima. Podrška za regeks je deo standardne biblioteke mnogih programskih jezika, uključujući Java-u i Python, i ugrađena je u sintaksu drugih, uključujući Perl i ECMAScript. Implementacije funkcionalnosti regularnih izraza se često nazivaju mašinom za regeks, a dostupne su i brojne biblioteke za rad sa regularnim izrazima. Krajem 2010-ih, nekoliko kompanija je počelo da nudi hardverske implementacije regeks endžina kompatibilnih sa PCRE (Perl Compatible Regular Expressions) bibliotekom koje su brže od čisto softverskih implementacija.

Iako regularni izrazi zauzimaju vrlo važno mesto u programskim jezicima i programiranju, mi ćemo ovde samo ukratko opisati JavaScript regularne izraze.

Sintaksa obrazaca regularnog izraza

Za izražavanje/zapisivanje obrasca regularnog izraza postoji standardna tekstualna sintaksa. Svaki znak u regularnom izrazu (to jest, svaki znak u nizu koji opisuje njegov obrazac) je ili metaznak sa posebnim značenjem, ili regularni znak koji ima doslovno značenje. Zajedno, metaznakovi i literalni znakovi mogu se koristiti za identifikaciju teksta datog obrasca ili obradu više njegovih instanci. Podudaranja uzoraka mogu varirati od precizne jednakosti do veoma opšte sličnosti, koju kontrolišu metaznakovi. Na primer, obrazac **.** je veoma opšti obrazac – poklapa se sa bilo kojim doslovnim znakom; **`[a-z]`** je manje uopšten (poklapa se sa svim malim slovima od `'a'` do `'z'`); konačno, **b** je precizan obrazac (odgovara samo znaku `'b'`). Uobičajeni meta znakovi (ovde boldovani i razdvojeni znakom zarez) su: **{}**, **[]**, **()**, **^**, **$**, **.**, **|**, *****, **+**, **?** i ****.

Većina formalizama podržava sledeće operacije za konstruisanje regularnih izraza date u Tabeli 2.

Tabela 2 Operacije za konstruisanje regularnih izraza

Simbol	Značenje	Primer
|	Logičko "ili" (*pipe*) označava podudaranje stringa sa leve strane ili stringa sa desne strane (alternative).	**`plavo|modro`** podudara se sa **`plavo`** ali i sa **`modro`**.
?	Označava podudaranje **nula** ili **jedne** pojave karaktera kome prethodi.	**`colou?r`** se podudara i sa `color` i sa `colour`.
*****	Označava **nula** ili **više** podudaranja karaktera kome prethodi.	**ab*c** se podudara sa **ac**, **abc**, **abbc**, **abbbc**, itd.
+	Označava **jedno** ili **više** podudaranja karaktera kome prethodi.	**ab+c** se podudara sa **abc**, **abbc**, **abbbc**, itd., ali se ne podudara sa **ac**.
{n}	Označava **tačno n** podudaranja karaktera kome prethodi.	**`^[\d]{4}$`** se podudara sa 4 cifarska string , i samo sa četiri cifarska stringa jer je **^** na početku i **$** je na kraju regeksa.

Simbol	Značenje	Primer
{n,}	Označava **minimalno n** podudaranja karaktera komjima prethodi.	**go{2,}gle** podudara se sa **google**, **gooogle**, **goooooogle**, **gooooooogle**, ….
{,n}	Označava **maksimalno n** podudaranja karaktera kojima prethodi.	**go{,2}gle** podudara se sa **google**, **gogle**
{n,m}	Označava **minimalno n** i **maksimalno m** podudaranja karaktera kojima prethodi.	**go{2,4}gle** podudara se sa **google**, **gooogle** i **goooogle**.
.	Džoker znak koji se podudara sa svakim pojedinačnim karakterom izuzev kraja inije (end of line).	**sh.rt** podudara se sa **shirt**, **short** i svakim karakterom između **sh** i **rt**. **a.*b** podudara se sa svakim stringom koji sadrži karakter **a** iza koga na nekoj poziciji sledi karakter **b**.
^	Označava podudaranje sa termom ako se term pojavljuje na početku paragrafa ili linije.	**^Jabuka** podudara se sa paragrafom ili linijom koji započinju sa **Jabuka.**
[^]	Označava **nepodudaranje** sa karakterom ili termom u zagradi	**[^a-e]** podudara se sa svakim karakterom osim **a**, **b**, **c**, **d**, **e**.
$	Označava podudaranje sa termom ako se term pojavljuje na kraju paragrafa ili linije.	**bye$** podudara se sa svakom linijom ili parafrafom koji se završava sa **bye**.
[]	Označava podudaranje sa svakim pojedinačnim karakterom iz liste u zagradi.	**b[aecro]d** podudara se sa **bad**, **bed**, **bcd**, **brd**, i **bod.**
–	Označava opseg slovnih ili brojevnih karaktera; često se koristi u srednjoj zagradi.	**k[a-c2-5]m** podudara se sa **kam**, **kbm**, **kcm**, **k2m**, **k3m**, **k4m** i **k5m**.
( )	Grupiše jedan ili više regularnih izraza.	**codexpedia\.(com|net|org)** podudara se sa **codexpedia.com**, **codexpedia.net**, i **codexpedia.org**.
!	Označava **nepodudaranje** sledećeg karaktera ili regularnog izraza.	q(?![0-9]) podudara se sa karakterom **q** ako karakter iza **q** nije cifra; na primer podudara se sa **q** u stringovima **abdqk**, **quit**, **qeig** ali se ne podudara sa **q** u stringovima **q2kd**, **sdkq8d**.
****	Poništava specijalno značenje prethodećeg karaktera	**a\.b** proglašava džoker znak **.** znakom bez specijalnog značenja, odnosno podudara se samo sa **a.b** .

Kreiranje regularnog izraza u JavaScriptu

Regularni izraz se i JavaScript-u može kreirati na dva načina.

- Prvi je putem literala (doslovnog oblika) regularnog izraza koji se sastoji od obrasca navedenog unutar kosih crta, kao u sledećem primeru:

  ```
  const regIz = /ab+c/;
  ```

  Literalni način kreiranja regularnog izraza obezbeđuju kompilaciju regularnog izraza pri učitavanju skripta. Ako se regularni izraz ne menja, korišćenje ovog načina kreiranja može poboljšati performanse.

- Drugi način je pozivanje konstruktorske funkcije objekta RegExp gde je sam obrazac string pod navodnicima zadat kao argument konstruktora:

```
const regIz = new RegExp ("ab+c");
```

Korišćenje konstruktora obezbeđuje kompilaciju regularnog izraza u vreme izvršenja. Preporučuje se u situacijama kada se zna da će se obrazac regularnog izraza menjati ili se obrazac uopšte ne zna – na primer, dobija se iz korisničkog unosa.

Podrška regularnih izraza u JavaScriptu

JavaScript podržava širok repertoar za rad sa regularnim izrazima počevši od jednostavnih (izrazi bez metakaraktera) do najsloženijih izraza sa metakarakterima. Podrška obuhvata sledeće:

- **Tvrdnje** (*Assertions*), što uključuje indiciranje početaka i završetaka linija i reči i druge obrasce kojima se na neki način indicira moguće podudaranje (unapred, unazad, uslovno).
- **Klasifikovanje karaktera** što omogućuje razlikovanje tipova karaktera (na primer, razlikovanje slova od cifara, podudaranje sa specijalnim karakterima za vertikalnu i horizontalnu kontrolu poput novog reda, taba i sl.)
- **Grupisanje i povratno referenciranje** gde grupe grupišu više obrazaca u celinu, a grupe za „hvatanje“ pružaju dodatne informacije o pod-podudarnosti kada se obrazac regularnog izraza koristi za podudaranje sa stringom. Povratne reference se odnose na prethodno „uhvaćenu“ grupu u istom regularnom izrazu.
- **Kvantifikacija** koja indicira broj karaktera ili izraza za podudaranje.

U JavaScript-u, regularni izrazi su objekti a obrasci se koriste sa metodama `exec()` i `test()` objekta `RegExp` i sa metodama `match()`, `matchAll()`, `replace()`, `replaceAll()`, `search()` i `split()` prototipa `String`.

Detaljan opis podrške koji JavaScript pruža za rad sa regularnim izrazima dostupan je na adresi https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_expressions.

Primeri regularnih izraza

Primer 1:

a) Regularni izraz za proveru ispravnosti e-mail adrese

```
\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}\b
```

1. Regularni izraz za proveru ispravnosti veb linkova sa domenima com, org, edu, gov, rs

```
https?://(www\.)?[A-Za-z0-9]+\.(com|org|edu|gov|rs)/?.*
```

Primer 2: korišćenja regularnog izraza u JavaScriptu

U sledećem jednostavnom primeru ilustrovano je korišćenje regularnih izraza u JavaScriptu kojim se proverava ispravnost unetog broja telefona za koji je propisani format **nnn-nnn-nnnn**.

```
<!DOCTYPE html>
<html>
<body>
<p>
  Unesite Vaš broj telefona (sa pozivnim kodom) pa kliknite na "Check".
  <br />
  Očekivani format je ###-###-####.
</p>
<form id="form">
  <input id="phone" />
  <button type="submit">Check</button>
</form>
<p id="output"></p>

<script>
```

```
const form = document.querySelector("#form");
const input = document.querySelector("#phone");
const output = document.querySelector("#output");

const re = /^(?:\d{3}|\(\d{3}\))([-/.])\d{3}\1\d{4}$/;

function testInfo(phoneInput) {
  const ok = re.exec(phoneInput.value);

  output.textContent = ok
    ? `Hvala, vaš broj telefona ${ok[0]} je u ispravnom formatu`
    : `${phoneInput.value} nije broj telefona sa ispravnim formatom!`;
}

form.addEventListener("submit", (event) => {
  event.preventDefault();
  testInfo(input);
});
</script>

</body>
</html>
```

Ovde je u liniji
`const re = /^(?:\d{3}|\(\d{3}\))([-/.])\d{3}\1\d{4}$/;`

definisan obrazac poklapanja koji kaže da se telefonski broj sastoji od stringa u kome su tri dekadne cifre iza koga sledi znak **-** , zatim opet od stringa u kome su tri dekadne cifre iza koga sledi znak **-** i na kraju stringa u kome su četiri dekadne cifre.

3.2.3.6. Logički tip **`boolean`**

Tip `boolean` se u JavaScript-u koristi za rad sa logičkim vrednostima. Sadrži dve vrednosti `true` (istinito) i `false` (neistinito).

3.2.3.6.1. Konverzija u tip **`boolean`**

Vrednosti svih drugih tipova (`number`, `string`) kada se koriste u kontekstu logičkih vrednosti u JavaScript-u se implicitno (automatski) konvertuju u tip `boolean`. JavaScript je *slabo tipiziran* jezik sa vrlo prisutnom implicitnom konverzijom tipa što zahteva poseban oprez i dodatni napor programera.

U logičkim kontekstima poput uslova, vrednosti određenog tipa se kategorišu u dve grupe: one koje se evaluiraju na `true` (zovu se “thruty“) i one koje se evaluiraju na `false` (zovu se “falsy“).

Postoji šest vrednosti koje se evaluiraju na `false` u JavaScript-u:

- Ključna reč `false`
- Primitivna vrednost `undefined`
- Primitivna vrednost `null`
- Prazan string (`''`, `""`)
- Specijalna vrednost `NaN`
- `number` ili `bigInt` koji predstavlja `0` (`0`, `-0`, `0.0`, `-0.0`, `0n`)

Sve druge vrednosti se evaluiraju na `true`. Ovo pravilo se odnosi na SVE JavaScript vrednosti, pa i one za koje “zdrav razum“ očekuje da se evaluiraju na `false` poput praznog objekta (`{}`) i praznog niza (`[]`). Informacija o logičkoj vrednosti na koju će posmatrana vrednost biti evaluirana može se pribaviti funkcijom `Boolean()` ili navođenjem znaka `!!` ispred vrednosti kao u sledećem primeru:

```
console.log (!! undefined) // ispisaće na konzoli: false
console.log (Boolean(undefined)) // ispisaće na konzoli: false
```

3.2.3.6.2. Sintaksa tipa **boolean**

Sintaksno, tip `boolean` predstavlja se svojom vrednošću `true` ili `false`, izrazima poređenja (n.pr., `10 > 9`) i izrazima koji sadrže logičke operacije o kojima će biti reči odmah iza ovog odeljka. Svi sledeći konstrukti su legalni konstrukti jezika:

```
true // true
false // false
10 > 9 // true
9 > 10 // false
console.log (10 > 9) // ispisaće na konzoli: true
{
    let isGreater = 4 > 1;
    console.log(isGreater); // ispisaće na konzoli: true
}
```

3.2.3.6.3. Logički operatori

U JavaScript-u postoje tri osnovna logička operatora:

- Disjunkcija, logičko ili - simbol (**||**),
- Konjunkcija, logičko i - simbol (**&&**),
- Negacija - simbol (**!**).

Pored njih, podržan je i operator

- Nulto spajanje - simbol (**??**).

Iako se nazivaju „logičkim“, mogu se primeniti na vrednosti bilo kog tipa, ne samo na logičke vrednosti. Njihov rezultat takođe može biti bilo kog tipa. U nastavku ćemo prikazati svaki od ovih operatora.

Disjunkcija

Operator || (disjunkcija, logičko ILI). Ovo je n-arni operator (može se primeniti na n operanada) koji u „normalnim“ jezicima operiše samo sa logičkim tipom i vraća vrednost `true` ako bar jedan operand ima vrednost `true`, ili vraća vrednost `false` ako su svi operandi sa vrednošću `false`.

JavaScript ima “jednu ali vrednu“ specifičnost u implementaciji ovog operatora. Kao prvo, operator može da operiše sa operandima čija vrednost nije tipa `boolean`. Kao drugo, može da vrati vrednost koja nije `true` ili `false`. Algoritam funkcioniše na sledeći način:

- Evaluira operande sa leva na desno.
- Svaki operand konvertuje u tip `Boolean` (sledeći pravila koja su navedena na početku ovoga odeljka). Čim kao rezultat konverzije dobije `true`, zaustavlja se i **vraća originalnu (nekonvertovanu) vrednost tog operanda** (tzv. kratko spajanje).
- Ako su svi operandi evaluirani na `false`, **vraća originalnu (nekonvertovanu) vrednost poslednjeg operanda**.

Evo primera:

```
console.log(( 1 || 0 )); /* vraća vrednost prvog operanda (vrednost 1 tipa
                            number) jer se 1 evaluira na true) */

console.log ( null || 1 ); /* vraća vrednost drugog operanda (vrednost 1
                              tipa number) jer je 1 prva vrednost koja se
                              evaluira na true  */
console.log ( null || 0 || 1 ); /* vraća vrednost trećeg operanda
```

```
                                    (vrednost 1 tipa number) jer se operand
                                    1 prvi evaluira na true) */


console.log ( undefined || null || 0 ); /* vraća vrednost poslednjeg
                                           operanda (vrednost 0 tipa
                                           number) jer se svi operandi
                                           evaluiraju na false) */
```

Konjunkcija

Operator && (konjunkcija, logičko I). Ovo je takođe n-arni operator (može se primeniti na n operanada) koji u „normalnim" jezicima operiše samo sa logičkim tipom i vraća vrednost `true` ako su svi operandi imali vrednost `true`, ili vraća vrednost `false` ako je bio bar jedan operandi sa vrednošću `false`.

JavaScript ima specifičnost i u implementaciji ovog operatora. Kao prvo, operator može da operiše sa operandima čija vrednost nije tipa `boolean`. Kao drugo, može da vrati vrednost koja nije `true` ili `false`. Algoritam funkcioniše na sledeći način:

- Evaluira operande sa leva na desno.
- Svaki operand konvertuje u tip `boolean` (sledeći pravila koja su navedena na početku ovoga odeljka). Čim kao rezultat konverzije dobije `false`, zaustavlja se i **vraća originalnu (nekonvertovanu) vrednost tog operanda** (tzv. kratko spajanje).
- Ako su svi operandi evaluirani na `true`, **vraća originalnu (nekonvertovanu) vrednost poslednjeg operanda**.

Evo primera:

```
/* ako se prvi operand evaluira na true, AND vraća drugi operand. */
console.log ( 1 && 0 ); // ispisaće: 0
console.log ( 1 && 5 ); // ispisaće:5

/* ako se prvi operand evaluira na false, AND vraća njega. Drugi operand
se ignoriše. */
console.log ( null && 5 ); // ispisaće: null
console.log ( 0 && "šta god bilo" ); // ispisaće:0
```

Negacija

Operator ! (negacija). Negacija je unarni operator (može se primeniti samo na jedan operand) koji u „normalnim" jezicima operiše samo sa logičkim tipom i vraća vrednost `false` ako je operand imao vrednost `true`, ili vraća vrednost `true` ako je operand sa vrednošću `false`. Specifičnost JavaScript-a je što operator može da operiše i nad operandom čija vrednost nije tipa `boolean`. Operator radi sledeće:

1. Konvertuje operand u tip `boolean`: `true`/`false`.
2. Vraća inverznu `boolean` vrednost (`false` ako je operand konvertovan u `true`, odnosno `true` ako je operand konvertovan u `false`.

Evo ilustrativnog primera:

```
let a = null
let b = 1
console.log (!a && !b) // ispisaće: false. Zašto?
console.log (!(a&&b)) // ispisaće: true. Zašto?
```

Detaljan opis ovih logičkih operatora je na linku https://javascript.info/logical-operators.

Nulto spajanje

Operator ?? (nulto spajanje). Ovaj operator može se smatrati specijalnim slučajem operatora disjunkcije.

Operator je binarni i vraća vrednost operanda koji mu je sa desne strane ako mu je operator sa leve strane `null` ili `undefined`; u ostalim slučajevima vraća vrednost operanda koji mu je sa leve strane.

Evo jednostavnog primera:

```
const foo = null ?? 'podrazumevani string';
console.log(foo); // Očekivani ispis: " podrazumevani string "

const baz = 0 ?? 42;
console.log(baz); // Očekivani ispis: 0
```

Operator **??** zgodan je za korišćenje u situacijama u kojima treba izdvojiti vrednost koja je definisana (nije `null` i nije `undefined`) kao u sledećem primeru:

```
let Ime = null;
let Prezime = null;
let Nadimak = "Čičak Stravični";

// Prikazuje prvu vrednost koja je definisana:
alert(Ime ?? Prezime ?? Nadimak ?? "Anonimus"); // Čičak Stravični
```

Detaljan opis ovog operatora može se naći na linku https://javascript.info/nullish-coalescing-operator.

Redosled izvršavanja logičkih operatora

Pri izvršavanju logičkih operatora redosled je sledeći: prvo se izvršava operator negacije, zatim operator konjunkcije i na kraju operator disjunkcije, odnosno nultog spajanja[11]. Modifikacija redosleda izvršavanja moguća je korišćenjem malih zagrada u izrazima.

3.3. Sažetak

[11] Redosled izvršavanja svih operatora u JavaScriptu dostupan je na linku https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence#Table

Literatura uz Poglavlje 3

Poglavlje 4: NEPRIMITIVNI TIP `object`

JavaScript ima jedan osnovni neprimitivni tip, tip `object`.

U javaScript-u se objekti mogu posmatrati kao kolekcija svojstava gde su svojstva parovi (ključ, vrednost). Ključevi svojstava su ili tipa `string` ili tipa `symbol`. Vrednosti svojstva mogu biti vrednosti bilo kog tipa podržanog u jeziku, uključujući i objekte, što omogućava izgradnju složenih struktura podataka i definisanje podtipova.

JavaScript razlikuje dve vrste svojstava objekta: vrstu ***podatak*** i vrstu ***pristup***.

Svojstvo vrste ***podatak*** pridružuje ključu vrednost bilo kog tipa podržanog u jeziku. Jednostavno rečeno, svojstvo vrste ***podatak*** u sebi skladišti same vrednosti sa kojima se operiše pri evaluaciji izraza u toku izvršavanja programa. Primeri su vrednost `142.5`, `true`, `"ја сам стринг на ћирилици"`, `undefined`, itd. Te vrednosti su uskladištene u memoriji na lokaciji na koju upućuje ključ.

Svojstvo vrste ***pristup*** pridružuje ključu vrednost koja je specijalna funkcija koja se izvršava pri dobavljanju ili postavljanju vrednosti svojstva. Jednostavno rečeno, svojstvo vrste ***pristup*** ne skladišti samu vrednost, nego ***način dolaženja do vrednosti***.

Svako svojstvo, pored imena/ključa, ima atribute od kojih je jedan njegova vrednost ali ima i drugih atributa. JavaScript mehanizam interno omogućuje pristup svakom atributu. Atributi se mogu konfigurisati/postaviti metodom `Object.defineProperty()` i mogu se pročitati pomoću metode `Object.getOwnPropertyDescriptor()`. Detalji se mogu naći na stranici https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty .

U JavaScript-u, objekti su jedine promenljive vrednosti. Funkcije su, u stvari, takođe objekti sa dodatnom mogućnošću pozivanja. U zajednici onih koji iskazuju interesovanja za JavaScript prisutna je krilatica: "U JavaScript-u sve je objekat!" i živa diskusija o ovom iskazu. Složili se ili ne složili sa ovim tvrđenjem, ne možemo da ne prihvatimo činjenicu da je koncept objekta ključni u JavaScript-u. Naime, JavaScript je objektno-orijentisan jezik u pravom smislu te reči. Preciznije, to je prototipski-baziran objektno-orijentisani jezik koji nasleđivanja (odnosno veze među objektima), umesto putem klasnog obrasca, nativno ostvaruju putem koncepta ***prototipskog objekta***. Prototipski objekat je objekat koji se koristi kao obrazac od koga se dobijaju inicijalna svojstva za novi objekat. Čak i način korišćenja većine primitivnih tipova u JavaScript-u povezuje ih sa konceptom objekta, što smo videli u Poglavlju 4 kada smo govorili o objektnim omotačima primitivnih tipova.

U JavaScript-u, objekti se mogu posmatrati na dva osnovna nivoa: pojedinačni objekat i povezani objekti. Ova dva nivoa biće detaljno objašnjena u odeljcima koji neposredno slede.

Pored toga, od značaja je i mehanizam "proizvođenja" objekata, odnosno koncept i implementacija konstruktora. Nakon izlaganja o povezanim objektima, sledi izlaganje o konstruktorima i nasleđivanju među konstruktorima.

4.1. Pojedinačni objekti

Objekti su kompozitni tipovi podataka koji omogućuju rad sa više vrednosti – kolekcijama vrednosti. Objekat vrednosti skladišti u obliku parova (***ključ***, ***vrednost***) koje nazivamo ***svojstvima***. Objekti predstavljaju ***neuređene*** kolekcije. To znači da nad elementima kolekcije predstavljene objektom ne postoji nikakva relacija uređenja – na primer, ne zna se koje je svojstvo objekta "ispred" nekog drugog svojstva tog istog objekta.
Objekti su generalni gradivni blokovi nad kojima je izgrađen najveći deo JavaScript-a. Oni se u JavaScript-u mogu svrstati u jedan od 6 primarnih tipova:

- `String`
- `Number`
- `Boolean`
- `Null`
- `Undefined`
- `Object`

Zato što ova imena mogu da dovedu do zabune, treba posebno naglasiti da ***jednostavne primitive*** (`string`, `number`, `boolean`, `null`, i `undefined`) same ***nisu objekti***. `null` se ponekad referiše kao objektni tip, ali je to zabluda koja potiče iz baga u jeziku koji uzrokuje da funkcija `typeof null` vrati `"object"` što je nekorektno (i zbunjujuće). U stvari, `null` je poseban primitivni tip. Dakle, pomenuto stanovište : **"sve u JavaScript-u je objekat"** ipak nije potpuno tačno, iako postoje situacije u kojima se jednostavne primitive tretiraju kao objekti kreiranjem omotačkog objekta, a *postoji* i nekoliko specijalnih objektnih podtipova sa specifičnim ponašanjem koji se mogu nazvati *složenim primitivama*. O "omotavanju" u objekte već je bilo reči u odeljku o primitivnim tipovima, a ovde ćemo samo nabrojati ključne podtipove i navesti njihove osnovne karakteristike.

Ključni podtipovi tipa Object u Javascript-u su `Array` i `Function`.

Podtip `Array` je podtip objekta koji podržava rad sa ***uređenim kolekcijama***. To znači da nad elementima kolekcije predstavljene objektom tipa `Array` postoji relacija uređenja – tačno se zna koji je element objekta/niza "ispred" nekog drugog svojstva tog istog objekta/niza. U nastavku ovog odeljka govorićemo detaljno o podtipu `Array`.

Podtip `Function` je objekat koji se ***može pozvati*** (eng. "callable object"). Funkcije, koje u JavaScript-u takođe predstavljaju ključni koncept, su bazično normalni objekti sa pripojenom semantikom mogućnosti pozivanja i sa njima se može postupati kao sa bilo kojim drugim običnim objektom – na primer, funkcijama se mogu dodeljivati svojstva. Funkcijama je u nastavku posvećeno posebno poglavlje.

4.1.1. Sintaksa

Objekti se u JavaScript-u javljaju u dva sintaktička oblika: *deklarativni* (*literalni*) oblik, i *konstruktivni* oblik.

Literalna sintaksa za objekat je:

```
let mojObj = {
        kljuc1: vrednost1,
        // još konačan broj parova kljuc: vrednost
};
```

Konstruktivna sintaksa je:

```
let mojObj = new Object();
mojObj.kljuc = vrednost;
```

Rezultat korišćenja konstruktivnog oblika i literalnog oblika je potpuno ista vrsta objekta. Očigledna razlika je u tome što se u literalnoj deklaraciji u jednoj naredbi objektu može dodati jedno ili više svojstava (parova ključ/vrednost), dok se u konstruktivnoj sintaksi ti parovi dodaju jedan po jedan, posebnim naredbama dodele. Zbog toga se za kreiranje objekata gotovo uvek koristi literalna sintaksa.

4.1.2. Ugrađeni objekti

U JavaScript-u ima nekoliko podtipova objekata koji se zovu ***ugrađeni objekti***. Za neke od njih, imena ukazuju da su direktno povezani sa odgovarajućim parnjacima iz kategorije jednostavnih primitiva, ali

su njihove relacije, u stvari, mnogo komplikovanije što će se i videti u nastavku. Sledeća lista obuhvata ugrađene objekte[12]:

- `String`
- `Number`
- `Boolean`
- `Object`
- `Function`
- `Array`
- `Date`
- `RegExp`
- `Error`

Ovakvi ugrađeni elementi se pojavljuju kao stvarni tipovi ili čak klase u drugim jezicima kao što je, na primer, `String` klasa u jeziku Java.

Međutim, u JavaScript-u su to *samo ugrađene funkcije*. Svaka od tih ugrađenih funkcija može da se koristi kao konstruktor (poziv funkcije sa operatorom new) čiji je rezultat *novo-konstruisani* objekat (instanca) odgovarajućeg podtipa. Sledeći primer ilustruje razliku između primitivnog tipa `string` i objekta `String`:

```
let strPrim = "Ja sam string";
console.log (strPrim, " tipa:", typeof strPrim);/* Ispis: Ja sam string
                                                    tipa:  string */
console.log (strPrim, " instanca objekta String:",
            strPrim instanceof String); /* Ispis: Ja sam string  instanca
                                           objekta String: false */


let strObj = new String( "I ja sam string" );
console.log (strObj, " tipa:", typeof strObj); /* Ispis: I ja sam string
                                                  tipa:  string */

console.log (strObj, " instanca objekta String:",
            strObj instanceof String); /* Ispis: I ja sam string  instanca
                                          objekta String: true */
console.log(strPrim, " tipa:", typeof strPrim, ", ali se kada treba
ponašam kao objekat/niz String – moja dužina je: ", strPrim.length, " a
vrednost mog petog svojstva je: ", strPrim.charAt(5));
```

U ovom fragmentu koda primitivna vrednost "`Ja sam string`" uskladištena u varijabli `strPrim` nije objekat; to je primitivni literal i nepromenljiva (imutabilna) vrednost.

Da bi se nad njom izvršile operacije kao što je provera njene dužine, pristup pojedinačnom karakteru, itd., potreban je `String` objekat[13]. Ovde smo ga dobili pomoću konstruktora (poziv `new String`) kojim smo u varijablu `strObj` uskladištili instancu objekta tipa `String`. Na sreću, jezik automatski konvertuje `string` primitivu u `String` objekat kada je potrebno, što znači da skoro nikada neće biti potrebe da se eksplicitno kreira objekat. To potvrđuja poslednja linija ovog fragmenta koda:

```
console.log(strPrim, " tipa:", typeof strPrim, " ali se kada treba ponašam
kao objekat/niz String – moja dužina je:, strPrim.length, ", a vrednost
mog petog svojstva je: ", strPrim.charAt(5));
```

[12]Zapazite da ova imena počinju velikim slovom.

[13]U JavaScript-u `string` tip je imutabilan

U toj liniji se `strPrim` implicitno konvertuje u objekat koji ima metode `length` i `charAt(n)`, odnosno nema potrebe da se eksplicitno kreira objekat da bi pristup metodi radio. Rezultat je ispis:

Ja sam string tipa: string , ali se kada treba ponašam kao objekat/niz String – moja dužina je: 13 a vrednost mog petog svojstva je: m

Ista vrsta konverzije se dešava nad brojčanom literalnom primitivom. Na primer, u izrazu `42.359.toFixed(2)` kreira se objektni omotač `new Number(42.359)` i poziva se metoda tog objekta (u ovom slučaju metoda `toFixed()`).

U istom su odnosu i `Boolean` objekti i `boolean` primitive.

Tipovi `null` i `undefined` nemaju objektnog omotača, već samo primitivne vrednosti.

Nasuprot tome, `Date` (datumska) vrednosti mogu *samo* da se kreiraju konstruktivnom sintaksom jer nemaju parnjaka u literalnom obliku.

Tipovi `Object`, `Array`, `Function`, i `RegExp` (regularni izrazi) su svi objekti bez obzira da li se koristi literalna ili konstruktivna sintaksa. Konstruktivna sintaksa u nekim slučajevima nudi više opcija kreiranja nego literalna sintaksa. Iako se objekti mogu kreirati na oba načina, jednostavnijoj literalnoj sintaksi daje se gotovo univerzalna prednost. **Konstruktivnu sintaksu treba koristiti samo ako su potrebne dodatne opcije koje ona nudi**.

`Error` objekti se retko kreiraju eksplicitno u kodu, već se obično kreiraju automatski pri generisanju izuzetaka. Moguće ih je kreirati konstruktivnom sintaksom `newError()`.

Može da se pregleda interni podtip tako što će se pozajmiti bazna podrazumevana metoda `toString()` koja u sledećem fragmentu koda otkriva da je `strObj` objekat koji je, u stvari, kreiran putem `String` konstruktora:

```
// inspekcija podtipa objekta
let strObj = new String( "STR" );
console.log (Object.prototype.toString.call(strObj)); // [object String]
console.log (strObj);
```

Rezultat izvršavanja ovog fragmenta koda je sledeći ispis;

[object String]
String {'STR'}
0: "S"
1: "T"
2: "R"
length: 3
[[Prototype]]: String
[[PrimitiveValue]]: "STR"

Iz ovog ispisa se vidi (na dva mesta) da je `strObj` objekat koji je kreiran putem `String` konstruktora. Prvo mesto je ispis **[object String]**, a drugo je **[[Prototype]]: String**.

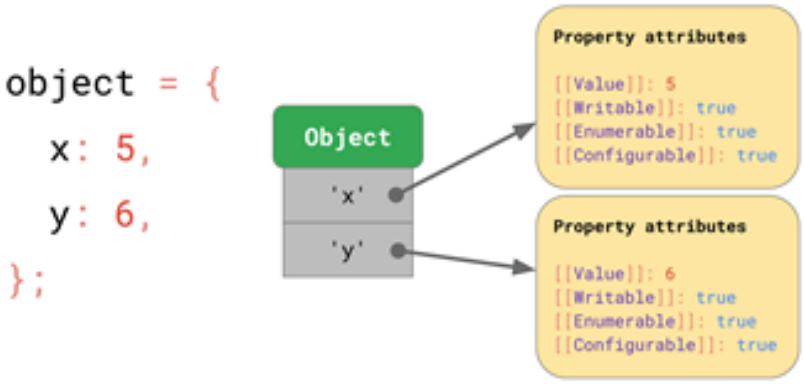
Metoda `toString()` instance `Object` vraća string koji predstavlja taj objekat. Ovaj primer koristi `Object.prototype.toString()` na specifičan način, odnosno ova metoda se može koristiti na mnogo različitih načina. Potpuna informacija o mogućnostima koje `Object.prototype.toString()` pruža i načinima korišćenja može se naći na adresi

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/toString.

4.1.3. Sadržaji objekta

Sadržaj objekta sastoji se od ***vrednosti*** (bilo kog tipa podržanog u jeziku) skladištenih (zapisanih) u specifično *imenovane lokacije* koje nazivamo *ključevima*.

ECMAScript specifikacija u suštini objekte definiše kao rečnike sa ključevima koji pokazuju na atribute (karakteristike) svojstva što je ilustrovano na slici 5.1.

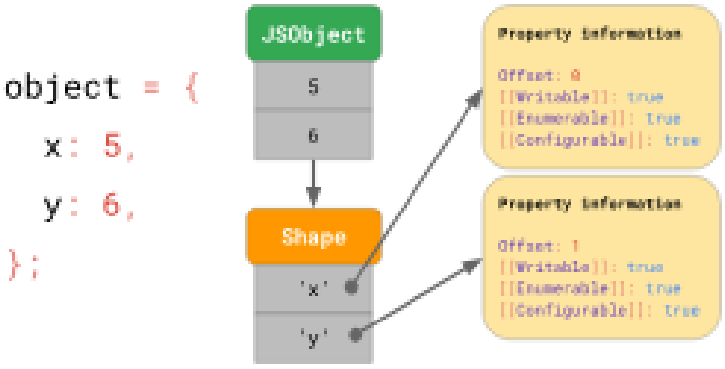


Slika 5.1 ECMAScript model objekta [https://mathiasbynens.be/notes/shapes-ics]

Sa slike se vidi da svojstva, pored ***vrednosti*** (atribut `[[value]]`, poseduju i druge atribute (atributi `[[Writable]]`, `[[Enumerable]]`, i `[[Configurable]]`) o kojima ćemo posebno govoriti u nastavku.

Iako mi kažemo "sadržaji objekta" što implicira da su te vrednosti *stvarno* smeštene unutar nečega što je objektni kontejner, to fizički nije tako.

U stvari, endžini implementiraju model definisan ECMAScript specifikacijom tako da optimizuju rad sa objektima – da ubrzaju pristup svojstvima i da uštede memoriju. V8 a i ostali endžini koriste mehanizam zvani ***Shape*** u kome skladište imena svojstava i atribute svojstava, dok se vrednosti svojstava skladište odvojeno. U strukturi ***Shape*** se, umesto vrednosti, nalaze ofseti do vrednosti svojstva. Mehanizam je ilustrovan slikom 5.2.



Slika 5.2 Implementacija objekta u JavaScript-u [https://mathiasbynens.be/notes/shapes-ics]

4.1.3.1. Rad sa svojstvima

Sa svojstvom objekta koje je, u stvari, vrednost na *nekoj lokaciji* operiše se korišćenjem dve notacije.

Prva notacija zove se ***dot (tačkasta) notacija/sintaksa***. U ovoj notaciji se prvo navodi ime objekta iza koga sledi znak tačka (`.`) i iza tačke ime ključa vrednosti kojoj se pristupa. Evo primera:

```
let mojObjekat = {
       a:2
};

let ispis = mojObjekat.a      /* dot notacija koja varijabli ispis
                                 dodeljuje vrednost svojstva sa imenom a
                                 objekta sa imenom mojObjekat */
console.log(ispis)            /* ispisaće 2 na konzoli */
```

Druga je ***notacija/sintaksa uglaste zagrade***. Ovde se navodi ime objekta iza koga sledi uglasta zagrada u kojoj se navodi ime ključa. Evo primera:

```
let mojObjekat = {};          /* Kreira se prazan objekat (bez svojstava)
                                 sa imenom mojObjekat */

mojObjekat["a"] = 3;          /* Sintaksa uglaste zagrade – dodaje se
                                  svojstvo koje ima ključ/ime a i vrednost
                                  3. */
/* Sintakse se mogu "mešati" */

console.log(mojObjekat["a"])  /* sintaksa uglaste zagrade - ispisaće 3 na
                                 konzoli */
console.log(mojObjekat.a)     /* dot sintaksa - ispisaće 3 na konzoli */
mojObjekat.b = 13;            /* Dot sintaksa – dodaje se svojstvo koje
                                 ima ključ/ime b i vrednost 33. */
console.log(mojObjekat["b"])  /* sintaksa uglaste zagrade - ispisaće 13 na
                                 konzoli */
```

Dot sintaksa se uobičajeno naziva " pristup svojstvu", dok se sintaksa uglaste zagrade obično naziva "pristup ključu". Obe sintakse obezbeđuju pristup istoj *lokaciji* odakle će izvući neku vrednost ili gde će postaviti neku vrednost, (u primerima koji neposredno prethode su to 2, 3 i 13), tako da se termini mogu koristiti ravnopravno. Mi ćemo najčešće da koristimo termin "pristup svojstvu" od sada na dalje.

4.1.3.1.1. Imena (ključevi) svojstava

Prema specifikaciji, samo dva primitivna tipa mogu poslužiti kao ključevi svojstava objekta u jeziku JavaScript:

tip `string`, ili
tip `symbol`.

U nastavku ćemo prvo da objasnimo ključeve svojstava predstavljene primitivnim tipom `string`. Nakon toga upoznaćemo se i sa ključevima tipa `symbol`.

4.1.3.1.1.1. Ključevi tipa **`string`**

Iako rade istu stvar, među dot sintaksom i sintaksom uglaste zagrade postoji razlika kada se radi o Imenima (ključevima) svojstava.

Glavna razlika je što dot sintaksa zahteva ime svojstva koje poštuje pravila leksičke gramatike jezika JavaScript (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Lexical_grammar), koja se odnose na način formiranja imena (recimo, ograničava skup karaktera i sl.), dok sintaksa uglaste zagrade može kao ime svojstva da primi bazično svaki UTF-8/Unicode kompatibilan string.

Na primer, da bi se referenciralo svojstvo sa imenom "SuperSuper-Zabava!", treba da se koristit `["SuperSuper-Zabava!"]` sintaksa zbog toga što `SuperSuper-Zabava!` nije leksički validno ime svojstva (znakovi **-** i **!** su nelegalni).

Takođe, kako sintaksa uglaste zagrade koristi **vrednost** stringa da specificira lokaciju, to znači da sam program može da generiše vrednost tog stringa kao u sledećem primeru:

```
let mojObjekat = {
        a:2,
        b:3,
};

let idx;
let hocuA = true;

if (hocuA) {
        idx ="a";
}
console.log( mojObjekat[idx] ); // 2
idx = "b"
console.log( mojObjekat[idx] ); // 3
```

U objektima, imena svojstava su uvek ili tipa `string` ili tipa `symbol`. Ako se kao ime svojstva koristi bilo koja vrednost drugog tipa, ona će uvek prvo biti konvertovana u `string`. Evo i primera:

```
let mojObjekat                = { };
mojObjekat [true]             = "foo";
mojObjekat [3]                = "bar";
mojObjekat [mojObjekat]       = "baz";

console.log(mojObjekat ["true"]);                   // Vraća: "foo"
console.log(mojObjekat ["3"]);                      // Vraća: "bar"
console.log(mojObjekat ["[object Object]"]);        // Vraća: "baz"
console.log(mojObjekat ["[mojObjekat]"]);           // Vraća: "undefined"
console.log(mojObjekat ["mojObjekat"]);             // Vraća: "undefined"
```

Ova konverzija uključuje čak i brojeve koji se uobičajeno koriste kao indeksi niza, pa treba biti oprezan da se ne napravi zbrka u korišćenju brojeva između objekata i nizova.

4.1.3.1.1.2. Ključevi tipa `symbol`

Najkraće rečeno, simbol je "jedinstvena primitivna vrednost" sa opcionim opisom. Tako koncipirana ona u jeziku JavaScript ima dva osnovna slučaja korišćenja:

- omogućuje "skrivena" svojstva objekta, i
- omogućuje preuređivanje nekih ugrađenih ponašanja.

U ovom odeljku bavićemo se tipom `symbol` iz aspekta prvog slučaja korišćenja. O drugom slučaju korišćenja govorićemo kasnije.

„Skrivena“ svojstva objekta

Simboli omogućuju kreiranje „skrivenih“ svojstava objekta, svojstava kojima nijedan drugi deo koda ne može slučajno pristupiti ili ih prepisati. Primer situacije u kojoj je to potrebno je kada se radi sa korisničkim objektima koji pripadaju kodu neke treće strane i treba da se tim svojstvima doda novo svojstvo kome treća strana neće moći da pristupi. Jedan legalan način je da se iskoristi tip `symbol` (setimo se, ključ svojstva objekta pored tipa string, može još jedino da bude tip `symbol`).

Da pogledamo kako se to radi. Recimo da se radi o objektima `korisnik` koji pripadaju kodu treće strane i da tim objektima, iz nekog razloga u našoj aplikaciji treba dodeliti novo svojstvo **`id`**.

```
// ovaj objekat pripada kodu treće strane i u njemu nema svojstva id
let korisnik = {
  ime: "Jova"
};

/* Ovo je sada naš kod koji treća strana ne vidi. Ovde objektu korisnik
dodajemo svojstvo id i dodeljujemo mu vrednost 1 */
let id = Symbol("id");
korisnik[id] = 1;
alert( korisnik[id] ); /* ispis: 1 - podacima se može pristupiti
                          korišćenjem simbola kao ključa */
alert( korisnik.id);   /* ispis: undefined - podacima se ne može pristupiti
                          korišćenjem imena simbola kao ključa */
```

Na isti način može se rešiti i situacija u kojoj dva skripta u istom objektu imaju isto svojstvo koje hoće da koriste „privatno“: Svako za to svojstvo pravi svoj ključ tipa `symbol` i sa njim radi nezavisno. Sledeći primer to ilustruje:

```
let user = { name: "Jovan" };

// Prvi skript ima svoje "id" svojstvo simbol id sa opisom id
let id = Symbol("id");
user[id] = "Prva id vrednost";

// Drugi skript ima svoje "id" svojstvo simbol id sa istim opisom id
let id = Symbol("id");
user[id] = "Druga id vrednost";
```

Globalni simboli i globalni registar

Mi rekosmo da su simboli jedinstveni - različiti su, čak i ako imaju isto ime. Međutim, to baš i nije sasvim tačno. Ponekad je potrebno da simboli sa istim imenom budu isti entiteti. Na primer, različiti delovi neke aplikacije žele da pristupe simbolu sa imenom `"id"` koji znači potpuno isto svojstvo.

Da bi se to postiglo, JavaScript podržava *globalni registar simbola*. U njemu se mogu kreirati simboli i kasnije im pristupiti uz garanciju da će ponovljeni pristupi istim imenom vraćati potpuno isti simbol.

Da bi se pročitao (kreirao, ako ga nema) simbol iz registra, koristi se metoda `Simbol.for(key)`. Taj poziv proverava globalni registar i, ako postoji simbol sa ključem `key`, metoda ga vraća. Ukoliko simbol sa ključem `key` ne postoji, metoda kreira novi simbol `Symbol(key)` i skladišti ga u registar. Na primer:

```
// Čitanje iz globalnog registra
let id = Symbol.for("id"); // ako simbol ne postoji, on se kreira.

// čitaj ga ponovo (možda iz nekog drugog dela koda)
let idAgain = Symbol.for("id");
```

```
// to je taj isti simbol
alert( id === idAgain ); // true
```

Simboli globalnom registru nazivaju se *globalni simboli*. Ako je potreban simbol za celu aplikaciju, dostupan svuda u kodu – globalni simboli su prava stvar za to.

Za globalne simbole, metoda `Simbol.for(key)` vraća simbol po imenu. Za suprotnu operaciju koja vraća ime po globalnom simbolou postoji metoda `Simbol.keyFor(sym)`. Dakle, može se uraditi nešto ovako:

```
// pribavi simbol po imenu
let simb = Symbol.for("imeSimbola");
let simb2 = Symbol.for("id");

// pribavi ime po simbolu
alert( Symbol.keyFor(simb) ); // imeSimbola
alert( Symbol.keyFor(simb2) ); // id
```

Metoda `Symbol.keyFor()` interno koristi globalni registar simbola da pronađe ključ za simbol. Dakle, ne radi za neglobalne simbole. Ako simbol nije globalan, neće moći da ga pronađe i vratiće vrednost `undefined`.

Međutim, svi simboli imaju svojstvo `description` pomoću koga se može pribaviti ime lokalnog simbola kao u sledećem primeru:

```
let globalniSimbol = Symbol.for("imeSimbola ");
let lokalniSimbol = Symbol("imeSimbola ");

alert( Symbol.keyFor(globalniSimbol) ); // imeSimbola, global simbol
alert( Symbol.keyFor(lokalniSimbol) ); // undefined, nije globalni

alert( lokalniSimbol.description ); // imeSimbola
alert( globalniSimbol.description ); // imeSimbola
```

Postoji mnogo „sistemskih“ simbola koje JavaScript koristi interno i oni se mogu koristiti za fino podešavanje različitih aspekata objekata. Spisak „sistemskih“ simbola postoji na linku https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol.

4.1.3.1.1.3. Sračunata imena svojstava u literalnoj deklaraciji objekta

Sintaksa pristupa svojstvu `mojObjekat[]` koju smo upravo prikazali je korisna ako imate potrebu da koristite vrednost sračunatu iz izraza kao ime ključa, na primer `mojObjekat[prefiks + ime]`. Međutim, to i nije od pomoći pri deklaraciji objekata korišćenjem literalne sintakse.

Specifikacija ES6 uvodi *sračunata imena svojstava*, gde se može specificirati izraz u srednjim zagradama `[ ]` na poziciji imena ključa literalne deklaracije objekta kao u sledećem primeru:

```
let prefiks = "prefiks";

let mojObjekat = {
        [prefiks + "prvaRec"]:"Zdravo",
        [prefiks + "drugaRec"]:"narode",
}

/* dalje se mogu ravnopravno koristiti obe sintakse */
console.log (mojObjekat["prefiksprvaRec"]);   // ispisaće: Zdravo
console.log (mojObjekat.prefiksdrugaRec);     // ispisaće: narode
console.log (mojObjekat.prefiksprvaRec,
```

```
mojObjekat["prefiksdrugaRec"]);// ??
```

4.1.3.1.2. Metode objekta

Kao što je prethodno rečeno, vrednost svojstva objekta može da bude bilo koji tip jezika JavaScript. To znači da vrednost svojstva može da bude i tip `function`, odnosno funkcija. Svojstva objekta koja su tipa `function` nazivaju se ***metode objekta***.

Ukoliko je svojstvo objekta tipa `function`, pristupanje tom svojstvu može da se manifestuje u dva oblika.

Jedan oblik je da se pristupi sadržaju na koji to svojstvo pokazuje, što u JavaScript-u nije ništa drugo nego string koji u sebi skladišti definiciju funkcije – njen izvorni kod.

Drugi oblik je poziv funkcije što znači izvršavanje funkcije. Ovaj drugi oblik je suština onoga što se od te funkcije očekuje – da bi funkcija dala neki rezultat ona se mora izvršiti. Evo trivijalnog primera kojim se ilustruje situacija:

```
let mojObjekat  = {
      kljuc1: "Moja funkcija vraća ono što joj se prosledi.",
      kljuc2: function identitet(x) {return x},
                  }

console.log("Ja sam objekat sa funkcijom:",mojObjekat.kljuc2);
console.log(mojObjekat.kljuc2(mojObjekat.kljuc1));


function identitet(x) {return x}

let mojIdentitet = identitet
let mojObjekat  = {
      kljuc1: "Moja funkcija vraća ono što joj se prosledi.",
      kljuc2: mojIdentitet,
      }

console.log("Ja sam objekat sa funkcijom:",mojObjekat.kljuc2);
console.log(mojObjekat.kljuc2(mojObjekat.kljuc1));
let x = "Ja sam string deklarisan u globalnom dosegu."
console.log("Samostalan poziv funkcije mojIdentitet: ", mojIdentitet(x))
console.log("Poziv metode na koju pokazuje kljuc2 u objektu mojObjekat:  ",
mojObjekat.kljuc2(x))
```

Mi smo takođe dali i našu "definiciju" objekta u kojoj smo kazali da se u JavaScript-u objekat može smatrati **skupom** svojstava. To nameće mentalni model objekta koji naglašava relaciju *pripadanja svojstva objektu* (elementi pripadaju skupu).

To nije neispravan mentalni model, ali se on ne može baš bukvalno primeniti, posebno ako je svojstvo objekta funkcija. Iz ovog iskaza uslediće poduža elaboracija kojom branimo iskazani stav o (ne)pripadanju. Međutim, mislim da se sve što ćemo pričati u nastavku može sažeti u jednu rečenicu: Funkcija je nešto što možemo da definišemo i koristimo a da u svetu u kome to radimo POJAM OBJEKTA UOPŠTE NE POSTOJI. A nije daleko od zdravog razuma ni pretpostavka da svako svojstvo (čak ne nužno funkcija) može da se definiše samo za sebe i da se pridružuje objektu po potrebi. Na primer, potpuno je blisko zdravoj pameti da svojstvo *AUTOR* može da se definiše nezavisno i da zatim da se pridružuje objektima poput objekta KNJIGA, SLIKA, IZLOŽBA. Ili, ako baš hoćemo da govorimo o svojstvu koje opisuje ponašanje što je blisko funkciji, svojstvo DISATI ili svojstvo JESTI može se definisati nezavisno od objekta i kasnije pridruživati objektima koji to svojstvo poseduju – ljudima,

životinjama. U nekim drugim formalizmima koji se koriste u programiranju (RDF i RDF Schema), a imaju sličnosti sa objektnim pristupom utoliko što operišu sa pojmom klasne hijerarhije, to se tako i radi.

Ipak, da iznesemo i elaboraciju jer ona pruža obrazloženje fokusirano na JavaScript a naša knjiga bavi se JavaScript-om.

Oslanjajući se na pomenuti mentalni model i činjenicu da se u drugim objektnim jezicima funkcije koje pripadaju objektima ("klasama") nazivaju "metodama", ljudi neretko koriste termin "pristup metodi" kada se radi o funkciji a termin "pristup svojstvu" samo kada to svojstvo NIJE funkcija.

Tehnički, funkcije nikada ne pripadaju objektima, tako da tvrdnja da je funkcija, kojoj je sticajem okolnosti pristupljeno referencom na objekat, automatski "metoda" deluje kao natezanje semantike.

Jeste tačno da neke funkcije imaju `this` reference u sebi, i da *ponekad* te `this` reference pokazuju na objektnu referencu na mestu poziva. Ali takav način korišćenje zaista ne čini da ta funkcija bude više "metoda" nego bilo koja druga funkcija. Razlog tome je što se `this` dinamički vezuje na mestu poziva u vreme izvršavanja, pa je njegova veza sa objektom u najboljem slučaju indirektna.

Svaki put kada se pristupi svojstvu objekta, to je **pristup svojstvu** bez obzira kakav tip vrednosti se dobije iz tog pristupa. Ako se *desi* da se dobije funkcija, to nije magično "metoda" u toj tački. Nema ništa specijalno (osim mogućeg implicitnog `this` vezivanja) u vezi sa funkcijom koja se poziva pristupom svojstvu objekta u odnosu na funkciju koja se poziva samostalno[14]. Evo i tehnički uverljivog primera:

```
function mojaFunkcija() {
  console.log( "Moja funkcija" );
}

let NekaMojaFunkcija = mojaFunkcija;  /* varijabla NekaMojaFunkcija
pokazuje na mojaFunkcija */

let mojObjekat = {
MojaFunkcija1: mojaFunkcija,
MojaFunkcija2: NekaMojaFunkcija,
};

/* sve linije koda koje slede ispisuju na konzoli isti string:
ƒ mojaFunkcija() {console.log( "Moja funkcija");} */

console.log("mojaFunkcija: ", mojaFunkcija);

console.log ("NekaMojaFunkcija: ", NekaMojaFunkcija);

console.log("mojObjekat.MojaFunkcija1: ", mojObjekat.MojaFunkcija1);
console.log("mojObjekat.MojaFunkcija2: ", mojObjekat.MojaFunkcija2);
```

Ovde su identifikatori `NekaMojaFunkcija`, `mojObjekat.MojaFunkcija1` i `mojObjekat.MojaFunkcija2` samo različito imenovane reference na istu stvar (funkciju) i ni jedna ne implicira ništa specijalno za funkciju, niti implicira da bilo koji objekat "poseduje" tu funkciju. Da je funkcija `mojaFunkcija()` bila definisana tako da u sebi ima `this` referencu

[14] U stvari, u JavaScript-u se funkcija uvek izvršava u nekom objektnom kontekstu. Pitanje je samo koji je to objekat (globalni objekat ili neki drugi objekat; ključna reč `this` kaže koji je to objekat).

na `mojObjekat.MojaFunkcija1` ili na `mojObjekat.MojaFunkcija2`, **jedina razlika** koja bi se mogla opaziti bilo bi *implicitno vezivanje* . Ni jedna referenca nema ništa što bi opravdavalo da se zove "metoda".

Možda bi se moglo tvrditi i da funkcija *postaje ili ne postaje metoda* u toku izvršavanja i samo za taj poziv u zavisnosti od načina pozivanja na mestu poziva (u kontekstu objekta ili samostalno). Čak i ovakva interpretacija bi se mogla shvatiti kao (nepotrebno) razvlačenje semantike.

Čini se da je najkorektniji zaključak iz svega ovoga da su "funkcija" i "metoda" termini koji se u JavaScript-u mogu ravnopravno koristiti.

Čak i kada se funkcija deklariše kao deo literalnog objekta kao što je slučaj sa funkcijom `MojaFunkcija` u sledećem fragmentu koda, ta funkcija *nije u većoj meri vlasništo* objekta – i dalje su to višestruke reference na isti funkcijski objekat koji se može pozivati i kao samostalna funkcija i kao metoda objekta:

```
let mojObjekat = {
MojaFunkcija: function mojaFunkcija() {
                console.log( "Moja funkcija" );
              }
};

let nekaMojaFunkcija = mojObjekat.MojaFunkcija;

console.log(nekaMojaFunkcija);   // Ispis: function mojaFunkcija() {…}
console.log(mojObjekat.MojaFunkcija); /* Ispis: function mojaFunkcija()
{…} */
nekaMojaFunkcija();                    // Samostalan poziv
mojObjekat.MojaFunkcija();             // Poziv metode objekta mojObjekat
```

Napomena: Specifikacija ES6 uvodi referencu `super` koja je namenjena za tipično korišćenje sa mogućnošću `class`. Način na koji se `super` ponaša (statičko `this` vezivanje umesto kasnog vezivanja) daje neku težinu ideji da je funkcija koja je negde `super` vezana više "metoda" nego "funkcija". Ipak, to su samo suptilne semantičke (i mehaničke) nijanse.

4.1.3.1.3. Deskriptori svojstva

Kao što smo videli u prethodnom odeljku, svojstvima objekta u JavaScript-u mogu se predstaviti različite stvari. To onda znači da ta svojstva imaju određene karakteristike o kojima, minimalno, treba imati informacije a i mogućnost njihovog modifikovanja. Pre specifikacije ES5, jezik JavaScript nije pružao direktan način za pribavljanje informacija o karakteristikama svojstava poput osnovne karakteristike da li se svojstvo može samo pribaviti (*read-only*) ili se može i modifikovati. Od ES5 specifikacije sva svojstva su opisana putem **deskriptora svojstva**. JavaScript razlikuje više vrsta deskriptora. Prve tri vrste koje se primenjuju na svojstva podataka objasnićemo u ovom odeljku, a još dve koje se odnose na svojstva pristupa biće opisane u jednom od narednih odeljaka koji se detaljno bavi pristupom svojstvima i postavljanjem svojstava.

Deskriptor svojstva (zvani i *deskriptor podatka*, jer je namenjen samo karakterisanju vrednosti svojstva podataka) za normalno svojstvo objekta, pored same njegove vrednosti, obuhvata i tri dodatne karakteristike: `writable`, `enumerable`, i `configurable`. Vrednosti koje mogu da uzmu ove karakteristike su logičkog tipa: `true` ili `false`. U stvari, ove tri karakteristike opisuju **šta se može raditi sa samim deskriptorima svojstva** (to određuje deskriptor `configurable`) i **šta se može raditi sa svojstvom** (to određuju deskriptori `writable` i `enumerable`).

Kada se kreira objekat, njemu se dodeljuju i svojstva odnosno, vrednosti svojstva i deskriptori svojstava. Podrazumevana vrednost koja se dodeljuje pri kreiranju svojstva korišćenjem dot sintakse

ili sintakse srednje zagrade za sva tri pomenuta deskriptora je `true`. Objektu se mogu dodati nova svojstva i/ili modifikovati vrednosti postojećih svojstava pomoću metode `Object.defineProperty()` pri čemu je modifikacija svojstva moguća SAMO ako deskriptor `configurable` tog svojstva ima vrednost `true`. Ako se svojstvo kreira korišćenjem metode `Object.defineProperty()`, podrazumevana vrednost svojstava je `false`. Slede primeri dodavanja svojstava objektu a u nastavku, kada budemo opisivali pojedinačno deskriptore, daćemo i primere modifikacije postojećeg svojstva.

Posmatrajmo sledeći kod:

```
let mojObjekat = {
        a:2
};

let descriptor = Object.getOwnPropertyDescriptor( mojObjekat, "a" );
console.log("Deskriptori svojstva a: " )
console.log(JSON.stringify(descriptor, null, 2))
```

Metoda (ili funkcija, kako Vam volja) `getOwnPropertyDescriptor()` vratiće sledeće

```
configurable: true
enumerable: true
value: 2
writable: true
[[Prototype]]: Object
```

A metoda `stringify` objekta JSON logovaće deskriptore u JSON formatu pa je rezultat izvršavanja ovde sledeći log:

Deskriptori svojstva a:

```
{
 "value": 2,
 "writable": true,
 "enumerable": true,
 "configurable": true
}
```

Isto će se desiti i pri izvršavanju sledećeg koda :

```
let mojObjekat = {};
mojObjekat.a = 2 // ili mojObjekat["a"] = 2

let descriptor = Object.getOwnPropertyDescriptor( mojObjekat, "a" );
console.log("Deskriptori svojstva a: " )
console.log(JSON.stringify(descriptor, null, 2))
```

Međutim, sledeći kod:

```
let mojObjekat = {};

Object.defineProperty( mojObjekat, "a", {
value:2,
} );
let descriptor = Object.getOwnPropertyDescriptor( mojObjekat, "a" )
console.log("Deskriptori svojstva a: " )
console.log(JSON.stringify(descriptor, null, 2))
```

ispisaće sledeći log:

Deskriptori svojstva a:

```
{
  "value": 2,
  "writable": false,
  "enumerable": false,
  "configurable": false
}
```

Napomena: Ako je objekat prazan, nema svojstava pa nema ni deksriptora tako da metoda `Object.getOwnPropertyDescriptor()` vraća `undefined`.

4.1.3.1.3.1. Deskriptor `Writable`

Mogućnost da se promeni vrednost svojstva objekta kontroliše se njegovim deskriptorom `writable`.

Posmatrajmo sledeći kod:

```
var mojObjekat = {};

Object.defineProperty( mojObjekat, "a", {
value:2,
writable:false, // nije writable!
configurable:true,
enumerable:true
} );

mojObjekat.a = 3;      // pokušaj dodele nove vrednosti svojstvu a
mojObjekat.a;          // 2; nije uspeo jer je svojstvo a nije writable
```

Kao što može da se vidi, modifikacija vrednosti svojstva sa imenom **a** nije se desila, svojstvo **a** zadržalo je vrednost 2. A nije se pojavila ni bilo kakva poruka o grešci. Ako to isto pokušamo u režimu `strict mode`, dobijamo grešku:

```
"use strict";

var mojObjekat = {};

Object.defineProperty( mojObjekat, "a", {
value:2,
writable:false, // not writable!
configurable:true,
enumerable:true
} );

mojObjekat.a = 3; // TypeError
```

Greška `TypeError` kaže nam da ne možemo menajti vrednost svojstva koje je **non-writable**.

Ova karakteristika svojstva ima veze sa geterima/seterima[15]. Naime, `writable:false` znači da se vrednost svojstva ne može menjati što je, na neki način, ekvivalentno sa definisanjem `no-op` setera. U stvari, `no-op` seter trebao bi da, pri pozivu, generiše baš grešku `TypeError` da bi bio u potpunosti saglasan sa `writable:false`.

[15] Geteri/seteri su funkcije koje se izvršavaju pri dobavljanju/postavljanju svojastava.

4.1.3.1.3.2. Deskriptor **`Configurable`**

Sve dok svojstvo ima karakteristiku da je konfigurabilno (vrednost deskriptora `configurable` je `true`), moguće je modifikovati deskriptorsku definiciju (menjati vrednosti deskriptorskih svojstava) korišćenjem funkcije `defineProperty()`. Evo primera:

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

mojObjekat.a = 3;
mojObjekat.a;  // 3

Object.defineProperty( mojObjekat, "a", {
        value:4,
        writable:true,
        configurable:false,    // deksriptor svojstava nije konfigurabilan!
        enumerable:true
} );

mojObjekat.a;                  // 4
mojObjekat.a = 5;
mojObjekat.a;                  // 5 jer je writable:true

Object.defineProperty( mojObjekat, "a", {
        value:6,
        writable:true,
        configurable:true,     // neće moći, jer je configurable:false
        enumerable:false       // neće moći, jer je configurable:false
} ); // TypeError
```

Poslednji poziv `defineProperty()` vraća greškuTypeError, bez obzira na `strict` režim, ako se pokuša izmena deskriptorske definicije nekonfigurabilnog svojstva što je u ovom kodu pukušano:

`configurable:false → configurable:true` i `enumerable:true → enumerable:false`.

Važno upozorenje: **postavljanje `configurable`** na **`false`** je **jednosmerna akcija i ne može se opozvati!** Ipak, postoji izuzetak: **čak i ako je svojstvo već `configurable:false`,** deskriptor **`writable` se uvek može promeniti sa `true` na `false`** bez poruke o grešci, ali se **ne može uraditi obrnuto – promeniti ga na `true` ako je već `false`**

Druga stvar koja sprečava probleme sa jednosmernošću `configurable:false` jeste mogućnost da se koristi `delete` operator za uklanjanje postojećeg svojstva. Sintaksa operatora je:

```
delete(svojstvo)
```

gde je svojstvo izraženo dot notacijom ili nitacijom srednje zagrade. Posmtrajno sledeći primer:

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

console.log('svojstvo a sa configurable:true pre delete: ',mojObjekat.a);
                  /* 2 */
delete mojObjekat.a;
console.log('svojstvo a sa configurable:true nakon delete: ',mojObjekat.a);
                       /* undefined */
```

```
Object.defineProperty( mojObjekat, "a", {
      value:2,
      writable:true,
      configurable:false,
      enumerable:true
} );

console.log('svojstvo a sa configurable:false pre delete: ',mojObjekat.a);
                        /* 2 */
delete mojObjekat.a;
console.log('svojstvo a sa configurable:false nakon delete: ',mojObjekat.a);
                  /* 2 */
```

Kao što se vidi, poslednji (drugi) poziv `delete` rezultovao je otkazom (svojstvo **a** nije izbrisano) bez poruke o grešci (silently), jer je svojstvo **a** deklarisano kao nekonfigurabilno (`configurable:false`).

Operacija `delete()` se koristi samo za uklanjanje objektnih svojstava (kada se mogu ukloniti) direktno iz objekta na koji se operator primenjuje. Ako je svojstvo objekta poslednja preostala *referenca* na neki objekat/funkciju i na njega se primeni `delete`, time se uklanja referenca i nereferencirani objekat/funkcija mogu da se odstrane u procesu oslobađanja memorije. Ali **nije dobro** da se razmišlja o `delete` kao o alatu za oslobađanje alocirane memorije kao što je to u drugim jezicima (n.pr. C/C++). U JavaScript-u, `delete` je samo jedna operacija uklanjanja svojstva – ništa više.

4.1.3.1.3.3. Deskriptor `Enumerable`

Poslednja deskriptorska karakteristika koju ćemo ovde da objasnimo je `enumerable` (još dve ćemo objasniti u odeljku o geterima/seterima).

Ime očigledno ukazuje na njenu prirodu - ova karakteristika kontroliše da li će se svojstvo pojaviti u određenim nabrajanjima, kao što je, na primer, `for..in` petlja. Vrednost svojstva `false` znači da se ono neće pojavljivati u takvim nabrajanjima, iako je samo svojstvo potpuno dostupno. Naravno, vrednost `true` znači da će se pojavljivati u takvim nabrajanjima.

Sva normalna svojstva definisana od strane korisnika su podrazumevano nabrojiva (`enumerable:true`), jer je to najčešće i potrebno. Ako svojstvo treba da bude izostavljeno kod nabrajanja, ono treba da ima deskriptor `enumerable:false`.

Nabrojivost ćemo obraditi sa više detalja u nastavku.

4.1.3.1.4. Nepromenljivost objekata

Na samom početku izlaganja o objektima rečeno je: Tip `object` je jedini tip u JavaScript-u koji nije nepromenljiv (imutabilan).

Iako je, generalno gledano, neprirodno da objekat bude nepromenljiv, ponekad je poželjno da svojstva i objekti budu nepromenljivi. Specifikacija ES5 (pa i JavaScript) podržava ovakvo rukovanje objektima na više različitih nijansiranih načina.

Pri tome, zajedničko za **sve** pristupe je da ***kreiraju plitku nepromenljivost***. To znači da utiču samo na objekat i njegova direktna svojstva. Ako objekat ima referencu na drugi objekat (niz, objekat, funkciju, itd.) ***sadržaji tog drugog objekta*** nisu izloženi uticaju ovih mehanizama i ***ostaju promenljivi***. Evo primera:

```
const mojNepromenljivObjekat = {}
Object.defineProperty( mojNepromenljivObjekat, "a", {
      value: [1,2,3],
      writable:false,
      configurable:false,
```

```
      enumerable:true
} );
mojNepromenljivObjekat.a; // [1,2,3]
mojNepromenljivObjekat.a.push( 4 );
mojNepromenljivObjekat.a; // [1,2,3,4]
```

U ovom snipetu je `mojNepromenljivObjekat` već kreiran i zaštićen kao nepromenljiv. Međutim, da bi se od promena zaštitili i sadržaji `mojNepromenljivObjekat.a` (što je drugi objekat - niz), trebalo bi da se i `a` učini nepromenljivim, recimo, na sledeći način:

```
let mojNiz = [1,2,3]

let mojNepromenljivObjekat = {}
Object.defineProperty( mojNepromenljivObjekat, "a", {
      value: Object.freeze (mojNiz), // Object.freeze () "zamrzava" mojNiz
      writable:false,
      configurable:false,
      enumerable:true
} );
mojNepromenljivObjekat.a; // [1,2,3]
mojNepromenljivObjekat.a.push( 4 ); /* Greška: Uncaught TypeError: Cannot
add property 3, object is not extensible */
```

Napomena: Nije baš jako uobičajeno da se kreiraju duboko usađeni nepromenljivi objekti u JavaScript programima. Mogu da postoje specijalni slučajevi koji to traže, ali je generalni obrazac dizajna da se to ne radi. Dakle, ukoliko Vam se učini da treba da *zapečatite* (*seal)* ili *zamrznete* (*freeze*) sve Vaše objekte, preporuka je da ponovo razmotrite dizajn Vašeg programa iz aspekta robusnosti na potencijalne promene vrednosti objekata; prosto, nije prirodno da se objekti ne menjaju.

4.1.3.1.4.1. Konstantno objektno svojstvo

Kombinovanjem `writable:false` i `configurable:false` može se kreirati *konstantno objektno svojstvo* (vrednost mu se ne može izmeniti, samo svojstvo se ne može redefinisati niti obrisati). Naravno, objektu se mogu dodavati nova svojstva. Sledeći snipet to ilustruje.

```
let mojObjekat = {};

mojObjekat.ime = "Ja se zovem mojObjekat"
console.log(JSON.stringify(mojObjekat))

mojObjekat. OMILJENI_BROJ = 120;
console.log(JSON.stringify(mojObjekat))

Object.defineProperty(mojObjekat,"OMILJENI_BROJ", {
               value:42,
               writable:false,
               configurable:false
       } );
console.log(JSON.stringify(mojObjekat))
let descriptor = Object.getOwnPropertyDescriptor( mojObjekat, "ime" );
console.log(JSON.stringify(descriptor, null, 2 ))
descriptor = Object.getOwnPropertyDescriptor( mojObjekat, "OMILJENI_BROJ"
);
console.log(JSON.stringify(descriptor, null, 2 ))
```

```
console.log(" Moj trenutni omiljeni broj je: ", mojObjekat.OMILJENI_BROJ);

let NOVI_OMILJENI_BROJ = 24
console.log(" Predomislio sam se. Hoću da moj novi omiljeni bude: ",
NOVI_OMILJENI_BROJ);

mojObjekat.OMILJENI_BROJ = NOVI_OMILJENI_BROJ;

console.log(" Nakon promene vrednosti sa 42 na 24, moj novi omiljeni broj
je: ", mojObjekat. OMILJENI_BROJ);

/* Uzaludno pokušavam da izbrišem svojstvo "OMILJENI_BROJ" */
delete mojObjekat.OMILJENI_BROJ
console.log(JSON.stringify(mojObjekat ));

/* Ali uspešno mogu da dodam novo svojstvo "MRSKI_BROJ" sa podrazumevanim
vrednostima deskriptora */
mojObjekat.MRSKI_BROJ = 142
console.log(JSON.stringify(mojObjekat));

descriptor = Object.getOwnPropertyDescriptor( mojObjekat, "MRSKI_BROJ" );
console.log(JSON.stringify(descriptor, null, 2 ));
```

4.1.3.1.4.2. Sprečavanje proširivanja objekta

Ukoliko se želi sprečiti dodavanje novih svojstava objektu, a da to ne utiče na ostala svojstva objekta, treba pozvati metodu `Object.preventExtensions()` kao u sledećem primeru:

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

Object.preventExtensions( mojObjekat );

mojObjekat.b = 3;
mojObjekat.b; // undefined
```

U nestriktnom režimu nema kreiranja svojstva **b**, ali se ne izdaje ni poruka o grešci. U striktnom režimu se izdaje greška **TypeError: Cannot add property b, object is not extensible.**

Pečaćenje (`Seal`)

Funkcija `Object.seal()` kreira "zapečaćen" objekat, što znači da se uzima postojeći objekat i, u suštini, poziva `Object.preventExtensions()` nad tim objektom čime se sva njegova postojeća svojstva označavaju kao `configurable:false`.

```
var mojObjekat = {
        a:2
};
console.log(JSON.stringify(mojObjekat)) // {a:2}

let descriptor = Object.getOwnPropertyDescriptor( mojObjekat, "a" );
console.log(JSON.stringify(descriptor, null, 2))
/*
{
  "value": 2,
```

```
  "writable": true,
  "enumerable": true,
  "configurable": true
} */

Object.seal( mojObjekat );
console.log(JSON.stringify(mojObjekat)) // {a:2}
descriptor = Object.getOwnPropertyDescriptor( mojObjekat, "a" );
console.log(JSON.stringify(descriptor, null, 2))
/*
{
  "value": 2,
  "writable": true,
  "enumerable": true,
  "configurable": false
} */

mojObjekat.b = 3;
console.log(JSON.stringify(mojObjekat)) // {a:2}
descriptor = Object.getOwnPropertyDescriptor( mojObjekat, "b" ); //
undefined
console.log(JSON.stringify(descriptor, null, 2)) // undefined
```

Nakon operacije pečaćenja, ne samo da ne mogu da se dodaju nova svojstva, već ne može ni da se rekonfiguriše ili briše ništa od postojećih svojstava ali još uvek *mogu* da se modifikuju vrednosti svojstva ako im deskriptor `writable` nije imao vrednost `false`.

Zamrzavanje (`Freeze`)

Metoda `Object.freeze()` kreira "zamrznuti" objekat što znači da uzima postojeći objekat ii nad tim objektom poziva `Object.seal()` i dodatno markira sva svojstva "pristupa podacima" kao `writable:false`, tako da im se vrednosti ne mogu menjati.

Ovaj pristup je najviši nivo nepromenljivosti koji se može primeniti na sam objekat jer se njime sprečava svaka promena objekta i svaka promena njegovih direktnih svojstava (iako, kao što je gore pomenuto, sadržaji bilo kojih drugih referenciranih objekata nisu izloženi ovom uticaju).

Moguće je i "duboko zamrznuti" objekat pozivajući `Object.freeze()` nad tim objektom i, zatim, rekurzivno, iterirajući nad svim objektima koje on referencira (koji, inače, ne bi bili izloženi uticaju), pozivati `Object.freeze()`. Pri tome treba biti krajnje oprezan jer se može nenamerno uticati na druge (deljene) objekte i time izazati vrlo neprijatni ukupni efekti.

4.1.3.1.5. Pristup vrednostima svojstava objekta

U ovom odeljku objasnićemo detaljnije mehanizme koje JavaScript nudi za pristupanja vrednosti svojstva objekta sa ciljem preuzimanja/postavljanja te vrednosti. Biće opisane operacije `[[Get]]` i `[[Set]]`, kao i `Getter`-i i `Setter`-i.

Operacija `[[Get]]`

Postoji suptilan ali važan detalj o načinu izvršavanja pristupa svojstvima u jeziku JavaScript. Objasnićemo o čemu se radi posmatrajući sledeći kod:

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

mojObjekat.a; // 2
```

`mojObjekat.a` je sintaksni konstrukt za pristup svojstvu, ali taj pristup se ne svodi *samo na traženje svojstva sa imenom* **a** *u objektu* **`mojObjekat`**, kao što to može da izgleda na prvi pogled.

Prema specifikaciji jezika, gore navedeni kod u stvari izvršava operaciju `[[Get]]` (nešto kao funkcijski poziv: `[[Get]]()`) nad objektom `mojObjekat`. Podrazumevana ugrađena `[[Get]]` operacija za objekat *prvo* traži svojstvo sa zadatim imenom u objektu nad kojim je pozvana i, ako ga nađe, vraća/postavlja vrednost. Pored toga, `[[Get]]` algoritam definiše drugo važno ponašanje koje se izvršava u slučaju da se u objektu *ne pronađe svojstvo sa traženim imenom*. U delu ovoga poglavlja koji se se bavi povezanim objektima u jeziku JavaScript detljano je opisano šta se u takvim slučajevima dešava. A dešava se prolazak kroz `[[Prototype]]` lanac, ako ga ima.

Jedan važan rezultat te `[[Get]]` operacije je da, ako se ni na koji način ne može naći vrednost za zahtevano svojstvo, operacija vraća vrednost `undefined`:

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

myObject.b; // undefined
```

Ovo ponašanje je različito od ponašanja kada se referenciraju *varijable* njihovim imenima. Ako se referencira varijabla koja se ne može razrešiti unutar primenljivog leksičkog dosega, rezultat nije `undefined` kao za objektna svojstva, nego se vraća greška `Reference Error`.

Evo jednog snipeta koji će baciti još malo svetla na ponašanje JavaScript-a pri pristupanju svojstvima objekata i probleme koji zbog toga mogu da nastanu. Posmatramo sledeći kod:

```
var mojObjekat = {
        a:undefined
};

mojObjekat.a; // undefined
mojObjekat.b; // undefined
```

Iz perspektive *vrednosti* ovde nema razlike između dve reference (**a** i **b**) - za obe je vraćen isti rezultat `undefined`. Međutim, druga `[[Get]]` operacija, iako se to baš i ne vidi na prvi pogled, potencijalno je izvršila malo više "rada" da pribavi vrednost za `myObject.b` nego za pribavljanje vrednosti `myObject.a`: svojstvu **a** vrednost `undefined` je dodeljena, i ona je nađena u samom objektu, dok za **b** nije nađeno ništa, pa ni da je `undefined`.

To dovodi do situacije da, gledajući samo vrednosti, ne može da se zaključi da li svojstvo *postoji* i *sadrži eksplicitno* vrednost `undefined`, ili svojstvo *ne postoji* i `undefined` je podrazumevana povratna vrednost nakon što `[[Get]]` nije uspela da vrati nešto eksplicitno. Uskoro (u odeljku Egzistencija svojstva) ćemo pokazati kako *se mogu* razlikovati ta dva slučaja.

Operacija `[[Put]]`

Kako postoji interno definisana operacija `[[Get]]` za pribavljanje vrednosti iz svojstva, za očekivati je da postoji i pretpostavljena `[[Put]]` operacija za postavljanje vrednosti svojstva.

Dosta je prirodna tendencija da se misli da dodela vrednosti svojstvu nekog objekta samo poziva `[[Put]]` za postavljanje vrednosti ili kreiranje tog svojstva objekta. Međutim, situacija je nešto nijansiranija od toga.

Kako se pri pozivu ponaša `[[Put]]` zavisi od brojnih faktora, pri čemu je najuticajniji **da li svojstvo** već **postoji u objektu** ili **ne postoji**.

Ako svojstvo postoji u objektu, `[[Put]]` algoritam će, ugrubo, da uradi sledeće:

1. Proveri da li je svojstvo pristupni (accessor) deskriptor (vidi odeljak "Geteri i seteri") **Ako jeste, pozovi seter ako seter postoji.**
2. Proveri da li je svojstvo podatka deskriptor `writable` sa vrednošću `false`. **Ako jeste, otkaži bez poruke o otkazu u nestriktnom režimu , ili izdaj grešku `TypeError` u striktnom režimu .**
3. Inače, postavi vrednost postojećeg svojstva na normalan način.

U slučaju kada svojstvo ne postoji u objektu, operacija `[[Put]]` je još nijansiranija i složenija. O njoj će biti više govora kada se bude govorilo o konceptu `[[Prototype]]`.

Geteri i seteri

Podrazumevane `[[Put]]` i `[[Get]]` operacije za objekte potpuno kontrolišu način postavljanje vrednosti postojećih ili novih svojstava, odnosno nalaženja vrednosti postojećih svojstava, respektivno.[16]

ES5 specifikacija je uvela način da se pregazi deo tih podrazumevanih operacija, ne na nivou objekta nego na nivou pojedinačnog svojstva, korišćenjem *getera* i *setera*.

Getri su svojstva koja, u stvari, pozivaju skrivene funkcije za nalaženje vrednosti.

Seteri su svojstva koja, u stvari, pozivaju skrivene funkcije za postavljanje vrednosti.

Ako se svojstvo definiše tako da ima ili geter ili seter ili oba, ono po definiciji postaje "pristupno svojstvo" (engl. "accessor property"), nasuprot "svojstvu podatka" (engl. "data property"). Za pristupna svojstva, deskriptori `value` i `writable` se zamenjuju deskriptorima svojstva zvanim `set` i `get` . Deskriptori `configurable` i `enumerable` ostaju. U sledećem primeru ćemo da pokažemo kako se definišu geteri i seteri.

Posmatrajmo sledeći kod:

```
var mojObjekat = {
        /* definicija getera za `a` */
        get a() {
                return 2;
        }
};

Object.defineProperty(
        mojObjekat,     // cilj
        "b",            // ime svojstva
        {  /* deskriptor */
                /* definicija getera za `b` */
                get:function(){
                        return this.a * 2
                },

        /* Osiguravanje da se `b` pojavljuje kao nabrojivo objektno
        svojstvo */
                enumerable:true
        }
);
```

[16] **Napomena:** Korišćenjem budućih/naprednih mogućnosti jezika moguće je pregaziti podrazumevane `[[Get]]` ili `[[Put]]` operacije za celokupni objekat (ne samo za pojedinačno svojstvo/svojstva).

```
mojObjekat.a; // 2
mojObjekat.b; // 4
```

I putem literalne sintakse sa `get a() { }` i kroz eksplicitnu definiciju sa `defineProperty()`, kreira se svojstvo objekta koje, u stvari, ne skladišti vrednost, već pristup njemu automatski rezultuje pozivanjem geterske funkcije i rezultat pristupa svojstvu, odnosno pribavljena vrednost svojstva, je vrednost koju vrati taj poziv geterske funkcije šta god taj povratni rezultat bio. Evo i primera koji to ilustruje:

```
let mojObjekat = {
        // defnicija getera za `a`
        get a() {
                return 2;
        }
};

mojObjekat.a = 3;
console.log(JSON.stringify(mojObjekat) // {"a":2}
```

Kako je ovde definisan samo geter za **a**, ako pokušamo da postavimo vrednost za **a** kasnije, operacija postavljanja neće rezultovati greškom već, prosto, neće izvršiti nikakvu dodelu i to bez ikakve notifikacije. Čak i da je ovde postojao validan seter, naš prilagođeni geter je hard-kodiran da vrati samo vrednost 2, tako da bi ono što je urađeno operacijom postavljanja bilo ignorisana.

Da bi se ovaj scenario "urazumio", svojstva bi trebalo definisati sa seterima koji pregaze podrazumevanu `[[Put]]` operaciju. U realnim uslovima se gotovo uvek deklarišu i geter i seter (postojanje samo jednog od njih često vodi ponašanju koje može da izgleda neočekivano):

```
let mojObjekat = {
        // definiši geter za `a`
        get a() {
                return this._a_;
        },

        // definiši seter za `a`
        set a(val) {
                this._a_ = val *2;
        }
};

mojObjekat.a = 2;

mojObjekat.a; // Vratiće 4
```

Objašnjenje: U ovom primeru se izrazom `mojObjekat.a = 2` u stvari, specificirana vrednost 2 zapisuje dodelom (`[[Put]]` operacija) u varijablu `_a_`. Ime `_a_` ne implicira ništa posebno u vezi sa ponašanjem varijable, to je normalno svojstvo kao i svako drugo a ime je ovde odabrano bez ikakve posebne namere.

4.1.3.1.6. Egzistencija svojstva

Ranije smo pokazali da rezultat pristupa svojstvu kao što je `myObject.a` može da bude vrednost `undefined` ako je `undefined` eksplicitno zapisano u svojstvo **a**, ali i ako svojstvo **a** uopšte ne postoji. Dakle, vrednost je ista u oba slučaja pa se postavlja pitanje: kako se ta dva slučaja mogu razlikovati?

JavaScript omogućuje da se pita neki objekat da li ima određeno svojstvo *bez* da se traži pribavljanje vrednosti tog svojstva. Za to se može koristiti operator **`in`** ili metoda `hasOwnProperty()` :

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

console.log (("a" in mojObjekat));    // true
console.log (("b" in mojObjekat);     // false

console.log (mojObjekat.hasOwnProperty( "a" ));      // true
console.log (mojObjekat.hasOwnProperty( "b" ));      // false
```

Operator **in** će da vrati true ako svojstvo **a** *postoji u objektu* mojObjekat ili *postoji negde na višem nivou prolaza kroz* [[Prototype]] *lanac* objekta mojObjekat. Ukoliko to nije slučaj, vratiće false.

Poziv hasOwnProperty() vraća true *samo ako sam* mojObjekat *ima u sebi svojstvo* **a**. On *neće konsultovati* [[Prototype]] *lanac* i vratiće vrednost false *ako* mojObjekat *nema u sebi svojstvo* **a** čak i ako to svojstvo postoji negde u prototipskom lancu objekta.

Vratićemo se na važne razlike između ta dva operatora kada budemo detaljno ispitivali povezivanje objekata.

Funkcija hasOwnProperty() je dostupna za sve normalne objekte putem delegacije objektu Object.prototype. Međutim, moguće je kreirati objekat koji nije vezan na Object.prototype (putem Object.create(null). U tom slučaju, poziv metode kao što je myObject.hasOwnProperty() će biti neuspešan.

U takvom scenariju, robusniji način za ovu proveru je Object.prototype.hasOwnProperty.call(mojObjekat,"a"), koji pozajmljuje bazni hasOwnProperty() metod i koristi *eksplicitno this vezivanje* da ga primeni na mojObjekat.

Napomena: Ljudi često pogrešno misle da operator **in** proverava *postojanje **vrednosti** unutar objekta*, ali to je netačno: on, u stvari, *proverava da li postoji **ime svojstva***. Rezultat ovakvog pogrešnog razumevanja je izražena sklonost među programerima da pokušavaju provere nalik sledećoj:

```
console.log((4 in [2, 4, 6]))
```

Većina onih koji ovako nešto napišu očekivaće da se ovaj izraz evaluira na true. Međutim, ovde je rezultat false zbog toga što je 4 vrednost svojstva, a ne ime (vrednost ključa svojstva). U objektu [2, 4, 6] nema svojstva sa imenom "4" (imena svojstava u objektu su "0", "1", i "2") pa je rezultat false. U stvari, objekat [2, 4, 6] izgleda ovako:

```
0: 2
1: 4
2: 6
length: 3
[[Prototype]]: Array(0)
```

Zbog toga će obe sledeće linije koda da vrate true:

```
console.log((0 in [2, 4, 6]))
console.log((2 in [2, 4, 6]))
```

4.1.3.1.7. Nabrajanje

Prethodno smo ukratko objasnili ideju nabrojivosti/enumerabilnosti ("enumerability") kada smo objašnjavali enumerable deskriptor svojstva. Hajde da to detaljnije pogledamo.

Posmatramo sledeći kod:

```
let mojObjekat = { };
```

```
Object.defineProperty(
   mojObjekat,
   "a",
// napravi da je "a" nabrojivo, kao što je normalno
        { enumerable:true, value:2 }
);

Object.defineProperty(
        mojObjekat,
        "b",
        // napravi da je "b" nenabrojivo
        { enumerable:false, value:3 }
);

console.log("JSON.stringify(mojObjekat) kaže mojObjekat: ",
JSON.stringify(mojObjekat)) /* JSON.stringify(obj) "posećuje" samo
nabrojiva svojstva     */


console.log("Međutim, mojObjekat ima i nenabrojivo svojstvo b: ",
mojObjekat.hasOwnProperty( "b" ), " čija je vrednost: ", mojObjekat.b);
console.log(" I operator in kaže da je tvrdnja da u objektu mojObjekat
postoji svojstvo b: ", ("b" in mojObjekat));

// .......

console.log("U objektu mojObjekat nabrojiva svojstva su:")
for (let k in mojObjekat) {
        console.log( "Svojstvo sa imenom: ", k, " ima vrednost: ",
mojObjekat [k] );
}
```

Ovaj kod ispisaće na konzoli sledeće:

JSON.stringify(mojObjekat) kaže mojObjekat: {"a":2}
Međutim, mojObjekat ima i nenabrojivo svojstvo b: true čija je vrednost: 3
I operator in kaže da je tvrdnja u objektu mojObjekat postoji svojstvo b: true
U objektu mojObjekat nabrojiva svojstva su:
Svojstvo sa imenom: a ima vrednost: 2

Iz primera se jasno vidi da svojstvo b objekta `mojObjekat` **postoji** i ima dostupnu vrednost 3, ali se ne pojavljuje u `for..in` petlji (iako **jeste** otkriveno pri proveri postojanja putem operatora **`in`** van petlje **`("b" in mojObjekat)`** i putem poziva **`mojObjekat.hasOwnProperty( "b" )`**). To je zbog toga što "nabrojiv" bazično znači "biće uključen ako se iterira kroz svojstva objekta ".

Drugi način za razlikovanje prebrojivih i neprebrojivih svojstava je ilustrovan sledećim primerom:

```
let mojObjekat = { };

Object.defineProperty(
        mojObjekat,
        "a",
        // napravi `a` nabrojivim, kao što je normalno
        { enumerable:true, value:2 }
);
```

```
Object.defineProperty(
        mojObjekat,
        "b",
        // napravi `b` nenabrojivim
        { enumerable:false, value:3 }
);

console.log (" Svojstvo a je nabrojivo: ",
mojObjekat.propertyIsEnumerable( "a" )); // true
console.log (" Svojstvo b je nabrojivo: ",
mojObjekat.propertyIsEnumerable( "b" )); // false


console.log (" Svojstva koja vidi Object.keys() : ", Object.keys(
mojObjekat ));
// ["a"]
console.log (" Svojstva koja vidi getOwnPropertyNames () : ",
Object.getOwnPropertyNames( mojObjekat )); // ["a", "b"]
```

Funkcija `propertyIsEnumerable()` testira da li dato ime svojstva postoji *u samom objektu* i istovremeno je `enumerable` deskriptor tog svojstva postavljen na `true`.

`Object.keys()` vraća niz imena *svih prebrojivih sopstvenih svojstava objekta*, dok `Object.getOwnPropertyNames()` vraća niz imena *svih sopstvenih svojstava objekta*, bez obzira da li su prebrojiva ili nisu.

Dakle, dok se operator `in` razlikuje od `hasOwnProperty()` u tome da li se konsultuje `[[Prototype]]` lanac ili ne (`in` konsultuje prototipski lanac, a `hasOwnProperty()` ne), `Object.keys()` i `Object.getOwnPropertyNames()` su u tom pogledu isti: oba proveravaju *samo svojstva deklarisana u direktno specificiranom objektu*.

Trenutno ne postoji ugrađeni način za dobijanje liste **svih svojstava** koji je ekvivalentan onome što bi test **`in`** operatora konsultovao (prolazeći sva svojstva u celom `[[Prototype]]` lancu). Takav pomoćni program bi se moga aproksimirati tako što bi se rekurzivno prolazilo kroz `[[Prototype]]` lanac objekta, i za svaki nivo uzimala lista `Object.keys()` da se dobiju samo prebrojiva svojstva ili lista `getOwnPropertyNames()` da se dobiju i prebrojiva i neprebrojiva svojstva.

4.1.3.1.8. Iteriranje nad svojstvima i vrednostima svojstava objekta

Petlja `for..in` iterira nad listom prebrojivih svojstava (preciznije, nad imenima ključeva svojstava) objekta (uključujući njegov `[[Prototype]]` lanac). Sintaksa je sledeća:

```
for (key in ime_objekta) { /* Ovde je key varijabla; njeno ime ne mora da
                              bude key */
  /* telo koje se izvršava za svako prebrojivo svojstvo u objektu */
}
```

Evo jednostavnog primera za ilustraciju:

```
let student = {
  ime: "Petar",
  dob: 21,
  jeZaposlen: true
};
```

```
for (let i in student) {
  // ključevi i vrednosti svojstava
  console.log( i, ": ", student[i] );
}
```

Rezultatat izvršavanja je log:

ime : Petar
dob : 21
jeZaposlen : true

Ako se malo preurede deskriptori svojstva `ime` (postavi se deskriptor `enumerable: false`):

```
let student = {
  ime: "Petar",
  dob: 21,
  jeZaposlen: true
};
Object.defineProperty(
        student,
        "ime",
        { enumerable:false}
);

for (let i in student) {
  // ključevi i vrednosti svojstava
  console.log( i, ": ", student[i] );
}
```

Rezultatat izvršavanja je log:

dob : 21
jeZaposlen : true

Ali šta biva ako se želi iterirati nad ***vrednostima*** tih svojstava?

Sve što smo napred kazali u vezi sa iteriranjem odnosi se na svojstva sa ključem tipa `string`. Simbolička svojstva **nisu uključena u iteriranje koje se realizuje `for … in` petljom** kao u sledećem primeru:

```
let id = Symbol("id");
let korisnik = {
  ime: "Jova",
  godine: 30,
  [id]: 123 /* Srednje zagrade su sintaksa za postavljanje simboličkog
               svojstva u literalnoj sintaksi objekta */
};

for (let kljuc in korisnik) alert(kljuc); // ime, godine (nema simboličkog
id)
```

Specifikacija ES6 dodaje sintaksu `for..of` za iteriranje nad vrednostima nizova (i objektima, ako objekat definiše sopstveni, prilagođeni iterator). O korišćenju `for..of` petlje za iteriranje nad nizovima detljano ćemo govoriti u odeljku koji se bavi posebno nizovima.

Iteriranje nad vrednostima svojstava objekta biće detaljno objašnjeno u odeljku koji se posebno bavi temom iteratora. Ovde ćemo samo da napomenemo da su iteratori objekti stvari koje će se iterirati (*iteratorski objekti*), i petlje zatim iteriraju nad sukcesivnim povratnim vrednostima iz poziva `next()`

metode tog iteratorskog objekta, po jednom za svaku iteraciju petlje. Pri tome, iteratorski objekat se dobija od podrazumevane interne funkcije poznate u specifikaciji kao @@iterator. Povratna vrednost iz `next()` poziva iteratora je objekat oblika `{ value: .. , done: .. }`, gde je `value` vrednost tekuće iteracije a `done` je logička vrednost koji kaže da li ima još da se iterira (`false`) ili je iteriranje završeno (`true`).

Napomena: Za razliku od iteriranja nad indeksima niza po numerički uređenom načinu (`for` petlja, ili `for..of` petlja), redosled iteracija nad svojstvima objekta **nije garantovan**[17] i može da varira među različitim JavaScript endžinima. **Nemojte da se oslanjate** na opaženo uređenje za bilo šta što traži konzistentnost među okruženjima, jer je to nepouzdano.

4.1.3.1.9. Pristup svojstvima ugneždenih objekata `?.`

Jedna od stvari od izuzetnog značaja u programiranju je robusnost programa. Robusnost programa znači da je program otporan na greške, odnosno da se terminirajuće greške u programu dešavaju samo kada je to zaista opravdano. A opravdano je kada se greška ne može predvideti i/ili se na nju ne može reagovati tako da program nastavi da se izvršava u izmenjenim uslovima.

Od nedavno JavaScript pruža bezbedan način za pristup svojstvima ugneždenih objekata, čak i ako posredno svojstvo ne postoji.

Problem nepostojećeg svojstva

Započećemo sa primerom koji ilustruje vrlo čestu situaciju u programiranju.

Pretpostavimo da imamo objekat `korisnik` koji skladišti razne informacije o korisniku, među kojima i adresu korisnika. Većina korisnika ima adrese u svojstvu `korisnik.adresa`, sa ulicom `korisnik.adresa.ulica`, ali postoje i korisnici za koje ovih podataka nema. U takvom slučaju pokušaj da se pristupi svojstvu `ulica` rezultuje terminirajućom greškom:

```
let korisnik = {}; // korisnik bez svojstva adresa "adresa"
alert(korisnik.adresa.ulica); // greška! Program prestaje da se izvršava
```

Jedan način kojim bi napred opisana situacija mogla da se reši je da organizujemo rukovanje greškama tako da se umesto terminirajuće greške vrati vrednost `undefined`. To je sasvim prihvatljivo rešenje jer potpuno odgovara semantici – imamo situaciju u kojoj vrednost nije dodeljena.

Tehnički, to se može uraditi na različite načine. Ali ni jedan od tih načina nije baš elegantan, niti rezultuje preterano čitljivim kodom.

JavaScript pruža mogućnost da se to uradi na lepši način korišćenjem operatora **`.?`**.

Operacija zvana *opciono ulančavanje* (**`?.`**) zaustavlja evaluaciju izraza ako se pod-izraz ispred operatora **`?.`** evaluira na `undefined` ili `null` i vraća `undefined`. Drugim rečima, `value?.prop`:

- Radi kao `value.prop`, ako `value` postoji.
- Vraća `undefined` ako `value` ne postoji, odnosno evaluira se na `undefined` ili `null`.

U primeru sa početka to bi izgledalo ovako:

```
let korisnik = {}; // korisnik nema adresu
alert( user?.address?.street ); // undefined (nema greške)
```

Napomena: Operacija **`?.`** se primenjuje samo na deo izraza koji je neposredno ispred operatora. Na primer, u `korisnik?.adresa.ulica.broj` operator `?.` omogućuje da svojstvo `korisnik` bude odsutno (bezbedno `null`/`undefined`) i vraća `undefined` u tom slučaju, ali to je samo za svojstvo

[17] U stvari, postoje neka pravila dobre prakse poput pravila da se svojstva sa ključevima string tipa iteriraju po alfabetskom redosledu (i još neka). Ta pravila endžini implementiraju u većini slučajeva ali ona nisu obavezan deo jezika.

korisnik. Ostalim svojstvima se pristupa na regularan način. Dakle, ako želimo i neka od njih da učinimo bezbednim, mora se iza svojstva navesti ?..

Detaljnije objašnjenje o ovoj mogućnosti možete naći na linku https://javascript.info/optional-chaining.

4.1.3.2. Objektne reference i kopiranje objekta

Jedna od fundamentalnih razlika između objekata i primitiva je u tome što se objekti čuvaju i kopiraju „po referenci“, dok se primitivne vrednosti (brojevi, stringovi logički vrednosti, itd.) uvek kopiraju „kao vrednost“. To znači da će se pri kopiranju primitivne vrednost napraviti nova lokacija u memoriji sa svojim identifikatorom (ime varijable) i u tu lokaciju biće kopirana vrednost. Konačan ishod su dve lokacije koje skladište jednake vrednosti. Na primer:

```
let poruka = "Zdravo!";                          // (1)
console.log(poruka) // Zdravo!
let fraza = poruka;                              // (2)
console.log(fraza) // Zdravo!
let fraza = "Zdravo i Tebi!"
console.log(fraza) // Zdravo i Tebi!
console.log(poruka) // Zdravo!
```

U ovom primeru imamo dve potpuno nezavisne promenljive poruka i fraza, od kojih svaka na početku sadrži string "Zdravo!" pri čemu je taj string promenljiva poruka dobila direktnom dodelom (1) a promenljiva fraza kopiranjem promenljive poruka (2). Promena vrednosti jedne promenljive (u ovom primeru promena vrednosti promenljive fraza sa "Zdravo!" na "Zdravo i Tebi!" se ne odražava na vrednost druge promenljive, promenljive poruka - on i dalje ostaje "Zdravo!".

Podsetimo se da se u objektu čuvaju ključevi svojstava a da se vrednosti i deskriptori svojstava čuvaju negde „napolju“ i da su ključevi reference koje pokazuju na to mesto „napolju“ gde su podaci stvarno smešteni. Baš kao što ključ otključava vrata prostorije u kojoj se nalazi ono što vas interesuje.

Kada se objekti kopiraju, kopiraju se samo reference a ne kopiraju se vrednosti na koje te reference pokazuju.

Ilustrovaćemo to tako što ćemo prethodni primer da transformišemo na sledeći način:

```
let porukaObjekat = {
                     poruka:"Zdravo!",
                    }

let frazaObjekat = porukaObjekat;
console.log(" Ja sam svojstvo poruka iz porukaObjekat: ",
             porukaObjekat.poruka)  // Zdravo!

console.log(" Ja sam svojstvo poruka iz frazaObjekat: ",
             frazaObjekat.poruka)  /* Zdravo! */

frazaObjekat.poruka = "Dobro jutro!" /* Promenjena je vrednost svojstva
                                        poruka u objektu frazaObjekat */

console.log("Ja sam svojstvo poruka iz porukaObjekat: ",
            porukaObjekat.poruka)  /* Dobro jutro! – promena se
                                      reflektuje na vrednost svojstva
                                      poruka u objektu porukaObjekat */
```

U ovom primeru, oba ključa (i ključ `poruka` u objektu `porukaObjekat` i ključ `poruka` u objektu `frazaObjekat` pokazuju na istu lokaciju jer je objekat `frazaObjekat` dobijen kopiranjem reference objekta `porukaObjekat`, odnosno oba imena (`porukaObjekat` i `frazaObjekat`) pokazuju na istu lokaciju.

Jedna od mogućnosti koja mnogo zanima nove JavaScript programere je način na koji mogu da dupliraju (umnožavaju) objekat. Bilo bi lepo da postoji ugrađena `copy()` metoda. Međutim, pokazuje se da je stvar "malo" komplikovanija, jer se takva metoda može koristiti (a i implementirati) samo ako je u potpunosti jasno **šta tačno znači ekvivalencija objekata** (kada su dva objekta **jednaka**) i šta bi trebao da bude podrazumevani algoritam za dupliranje. Jasno je da u svemu tome ključnu ulogu igra to što se objekti skladište i kopiraju po referenci.

4.1.3.2.1. Poređenje objekata

Objekti se u JavaScriptu porede po referenci. U JavaScript-u **dva objekta su jednaka ako imaju istu referencu** (pokazivač na objekat), odnosno samo ako u memoriji imaju lokacije u koje je uskladištena ista referenca na objekat. Najkraće rečeno, dva objekta su jednaka samo ako je to isti objekat. Evo primera koji to ilustruju.

```
/* Jednaki objekti */
let a = {};
let b = a; /* referenca se kopira, i u lokaciju b se postavlja ista
vrednost kao vrednost koja je u a */

console.log( a == b ); /* true, obe varijable referenciraju isti objekat
*/
console.log( a === b ); // takođe true
/* Različiti objekti iako ime je sadržaj isti – oba su prazni objekti*/
let c = {}; // objekt koji ima svoju posebnu referencu
let d = {}; // objekt koji ima svoju posebnu referencu

console.log( c == d ); /* false, različite reference*/
console.log( c === d ); // takođe false
```

Kada su u pitanju poređenja objekata gde relacija nije jednakost (na primer `a>b`) ili poređenje objekta sa primitivnom vrednošću (na primer, `d = 7`) , objekti se pri poređenju konvertuju u primitivne vrednosti. Pravila konverzije objekata u primitivne vrednosti objašnjena su u posebnom odeljku pod naslovom Konverzija tipova u JavaScript-u.

Nama je u daljem izlaganju od interesa samo poređenje po jednakosti.

Važan bočni efekat skladištenja objekata po referenci je da **objekat deklarisan ključnom reči `const` *nije nepromenljiv***. To znači da će da Vam se desi sledeće:

```
const porukaObjekat = {
                    poruka:"Zdravo!",
                   }
console.log ("Kada je objekt porukaObjekat deklarisan sa const, meni je
dodeljena vrednost: ", porukaObjekat.poruka)

porukaObjekat.poruka = "Dobar dan dobri ljudi!"

console.log ("Iako je objekt porukaObjekat deklarisan sa const, mene mogu
da promenu. Zato sam ja sada: ", porukaObjekat.poruka)
```

4.1.3.2.2. Kopiranje objekata

Da bismo bolje sagledali složenost kopiranja objekata, za ilustraciju posmatrajmo sledeću situaciju:

```
function drugaFunkcija() { /*... nešto radi ...*/ }

let drugiObjekat = {
  c:true
};

let drugiNiz = [];

let mojObjekat = {
  a:2,
  b: drugiObjekat,     /* Referenca na objekat drugiObjekat, ne kopija
                          celog objekta drugiObjekat!*/
  c: drugiNiz,         /* Referenca na objekat drugiNiz (podtipa Array),
                             ne kopija celog objekta drugiNiz */
  d: drugaFunkcija,      /* Referenca na objekat drugaFunkcija (podtipa
                             Function), ne kopija celog objekta
                             drugaFunkcija! */

};

drugiNiz.push(drugiObjekat, mojObjekat); // reference
console.log(drugiNiz[0])
console.log(drugiNiz[1])


function nekaFunkcija() { /*..*/ }

let nekiNiz = [];

let tudjiObjekat = {
  c:true
};

let mojObjekat = {
  a: 2,
  b: tudjiObjekat,  /* Referenca na objekat tudjiObjekat, ne kopija
                       celog objekta tudjiObjekat!*/
  c: nekiNiz,       /* Referenca na objekat nekiNiz (podtipa Array), ne
                       kopija celog objekta nekiNiz */
  d: nekaFunkcija,  /* Referenca na objekat nekaFunkcija (podtipa
                       Function),ne kopija celog objekta nekaFunkcija! */

};

nekiNiz.push(tudjiObjekat,  mojObjekat);  /*  U  nekiNiz  uskladištene  su
reference na objekte a ne sami objekti */
console.log(nekiNiz[0]) // Ovde je referenca na objekat tudjiObjekat
console.log(nekiNiz[1])// Ovde je referenca na objekat mojObjekat

console.log(mojObjekat.c)
console.log(mojObjekat.d)

console.log('tudjiObjekat.c kaže: Sada sam ', tudjiObjekat.c)
```

```
tudjiObjekat.c = 'Sam sam sebe promenuo pa sam sada string, a to će da promeni i one u koje sam kopiran'
console.log('tudjiObjekat.c kaže: ', tudjiObjekat.c)
console.log('Zato je nekiNiz[0] sada: ',nekiNiz[0])

mojObjekat.b.c = 'A sada me je promenuo objekat u koji sam kopiran pa sam postao drugačiji string'
console.log('tudjiObjekat.c kaže: ', tudjiObjekat.c)
console.log('Zato je nekiNiz[0] sada: ',nekiNiz[0])
```

Rezultat izvršavanja ovog prilično smušenog koda je sledeći:

```
{c: true}
{a: 2, b: {…}, c: Array(2), d: ƒ}
(2) [{…}, {…}]
ƒ nekaFunkcija() { /*..*/ }

tudjiObjekat.c kaže: Sada sam  true
tudjiObjekat.c kaže:  Sam sam sebe promenuo pa sam sada string, a to će da promeni i one u koje sam kopiran
Zato je nekiNiz[0] sada:
{c: 'Sam sam sebe promenuo pa sam sada string, a to će da promeni i one u koje sam kopiran'}
tudjiObjekat.c kaže:  A sada me je promenuo objekat u koji sam kopiran pa sam postao drugačiji string
Zato je nekiNiz[0] sada:
{c: 'A sada me je promenuo objekat u koji sam kopiran pa sam postao drugačiji string'}
```

Šta bi tačno trebala da bude *kopija* objekta `mojObjekat`?

Da bi se odgovorilo na to pitanje, prvo bi trebalo odgovoriti na pitanje da li bi to bila *plitka* ili *duboka* kopija.

Plitka kopija kopira samo sopstvena svojstva objekta a za svojstva koja nisu definisana u objektu kopira reference. Dakle, u primeru bi rezultat kopiranja bio novi objekat u kome bi bila vrednost 2, a svojstva b, c, d bi bila reference na **ista mesta** na koja pokazuju i odgovarajuća svojstva originalnog objekta. Prethodni kod je plitka kopija – reference su deljene. Svako može da promeni sadržaj na koji pokazuje referenca koju on "vidi".

Duboka kopija bi duplicirala (pravila novi) `mojObjekat`, novi `tudjiObjekat` i novi `nekiNiz`.

Međutim, ovde `nekiNiz` u sebi ima reference na `tudjiObjekat` i `mojObjekat` , pa bi i *te* reference (ono na šta one pokazuju) trebalo duplirati, umesto da se očuvavaju samo reference. Zbog toga bi dupliranje objekata `tudjiObjekat` i `mojObjekat` dovelo bi do problema beskonačnog cirkularnog dupliranja zbog cirkularne reference.

Kopiranje objekata zahteva odgovore i na niz drugih pitanja. Da li će se detektovati cirkularna referenca i prosto prekinuti cirkularni prolaz (ostavljajući duboki element delimično dupliranim)? Da li će se to tretirati kao fatalna greška i neće se duplirati ništa? Da li će se uraditi nešto između?

Takođe, nije potpuno jasno ni šta bi značilo "dupliranje" funkcije ako znamo da referenca na funkciju pokazuje na izvorni kod funkcije. Ima nekih hakerskih rešenja kao što je izvlačenje `toString()` serijalizacije izvornog koda funkcije (što se razlikuje u pojedinim implementacijama i nije pouzdano u svim endžinima zavisno od tipa funkcije).

Situacija je takva da različita JavaScript okruženja imaju svoje interpretacije i odluke. Međutim, koje od njih (ako uopšte postoji) bi trebalo da bude usvojeno kao JavaScript standard? Dugo vremena na to pitanje nema opšte prihvatljivog jednoznačnog odgovora.

4.1.3.2.2.1. Plitko kopiranje objekata

Plitko kopiranje je dovoljno razumljivo i nosi mnogo manje problema, pa je specifikacija ES6 definisala `Object.assign()` za taj zadatak. `Object.assign()` prihvata *ciljni* objekat kao svoj prvi parametar (u primeru čiji kod sledi je to prazan objekat) i jedan ili više *izvornih* objekata kao sledeće parametre (u primeru čiji kod sledi je to jedan objekat sa imenom `mojObjekat`). On iterira nad svim *enumerabilnim sopstvenim ključevima* (enumerabilno svojstvo je svojstvo koje se "uzima u obzir" pri iteriranju; sopstveni su ključevi koji su **deklarisani u samom objektu**) *izvornog* objekta/objekata i kopira ih (samo putem dodele =) u *ciljni objekat*. Vraća *ciljni objekat*.

Prikazaćemo primere korišćenja funkcije `Object.assign()` a pre toga ćemo da napravimo par helpera koji će da nam olakšaju ispisivanje logova:

```
function DefinisiSvojstvoObjekta (Objekat, Svojstvo, Vrednost, Dw, Dc, De)
{
      Object.defineProperty( Objekat, Svojstvo, {
      value: Vrednost,
      writable:Dw,
      configurable:Dc,
      enumerable:De
     } )
}

function hakovanjeZaFunkcije (key, value) {
  // ako je svojstvo funkcija, vrati kod te funkcije
  if (typeof value === 'function') {
    return value.toString();
  }
  return value;
}

function IspisiObjekat (Objekat) {
    let s = JSON.stringify(Objekat, hakovanjeZaFunkcije, 2);
      console.log (s)
}

function PribaviIspisiDeskriptorSvojstva (Objekat, Svojstvo) {
      let descriptor = Object.getOwnPropertyDescriptor( Objekat, Svojstvo
);
      console.log("Deskriptori svojstva ",Svojstvo,": \n",
      JSON.stringify(descriptor, null, 2))
}

function IspisiNaslov (naslov) {
         console.log(naslov)
         }
```

Sada ćemo da prikažemo primere.

Prvi od primera ilustruje kopiranje vrednosti i kopiranje referenci i posebno naglašava da kopira samo nabrojiva svojstva:

```
function drugaFunkcija() { /*..*/ }

let drugiObjekat = {
  c:true
};

let drugiNiz = [];

let mojObjekat = {
  a:2,
  b: drugiObjekat,     /* referenca na objekat drugiObjekat, ne kopija
                          celog objekta drugiObjekat!*/
  c: drugiNiz,         // opet referenca, sada na prazan niz!
  d: drugaFunkcija,    // opet referenca, sada na funkciju!

};

let novObjekat = Object.assign( {}, mojObjekat );
console.log(novObjekat.a);                    // ispisaće: 2
console.log(novObjekat.b === drugiObjekat);   // ispisaće: true
console.log(novObjekat.c === drugiNiz);       // ispisaće: true
console.log(novObjekat.d === drugaFunkcija);  // ispisaće: true

// ispiši originalni objekat mojObjekat
IspisiNaslov("mojObjekat")
IspisiObjekat (mojObjekat)
// Ispiši kopiju novObjekat
IspisiNaslov("novObjekat")
IspisiObjekat (novObjekat)

/* Ako u originalu svojstvu d postavimo deskriptor enumerable na false, to
svojstvo neće biti kopirano u kopiju */
DefinisiSvojstvoObjekta (mojObjekat, "d",mojaFunkcija, true, true, false)
let novinoviObjekat = Object.assign( {}, mojObjekat );
// Ispiši kopiju novinoviObjekat
IspisiNaslov("novinoviObjekat")
IspisiObjekat (novinoviObjekat)
```

Dupliranje koje se ostvaruje pomoću `Object.assign()` je čisti stil dodele putem jednakosti (**==**), pa se **specijalne karakteristike svojstva definisane deskriptorima svojstva** izvornog objekta **ne očuvavaju** na kopiranim svojstvima ciljnog objekta kao u sledećem primeru:

```
// Ilustrativni primeri
let NasloviZaIspis = ["tudjiObjekat","mojObjekat", "novObjekat" ]

// Objekat tudjiObjekat
let tudjiObjekat = {
  c:true
};
IspisiNaslov(NasloviZaIspis[0])
IspisiObjekat (tudjiObjekat)

// Objektu tudjiObjekat svi deskriptori se postavljaju na false
DefinisiSvojstvoObjekta (tudjiObjekat, "c", true, false, false, false)

// Svi deskriptori objekta tudjiObjekat su false
```

```
PribaviIspisiDeskriptorSvojstva (tudjiObjekat, "c")

// Objekat mojObjekat
let mojObjekat = {
  a:2,
  b: tudjiObjekat.c,
  };
IspisiNaslov(NasloviZaIspis[1])
IspisiObjekat (mojObjekat)
PribaviIspisiDeskriptorSvojstva (mojObjekat, "a")

/* U kompozitni objekat novObjekat kopiraju su objekti tudjiObjekat i
mojObjekat */
let novObjekat = Object.assign( {}, tudjiObjekat, mojObjekat );

IspisiNaslov(NasloviZaIspis[2])
IspisiObjekat (novObjekat)

/* Svojstvo novObjekat.b u koje je kopirano svojstvo tudjiObjekat.c ima
sve deskriptore true, a original je imao sve deskriptore false */
PribaviIspisiDeskriptorSvojstva (novObjekat, "b")
```

4.1.3.2.3. Duboko kopiranje objekata

Dobar link: https://javascript.plainenglish.io/shallow-copy-and-deep-copy-in-javascript-a0a04104ab5c

Duboka kopija objekta je kopija čija svojstva ne dele iste reference (upućuju na iste osnovne vrednosti) kao svojstva izvornog objekta od kojeg je napravljena kopija. Rezultat toga je da promena izvora ne izaziva promenu kopije i obrnuto. Pre no što se pozabavimo načinima koje JavaScript koristi za duboko kopiranje objekata, kazaćemo neke opšte stvari koje treba da nam pomognu da bolje razumemo i problem i ponuđena rešenja.

Dva objekta o1 i o2 su **strukturno ekvivalentna** ako su **njihova posmatrana ponašanja ista**. Ova ponašanja uključuju:

- Svojstva objekata o1 i o2 imaju ista imena u istom redosledu.
- Vrednosti njihovih svojstava su strukturno ekvivalentne.
- Njihovi lanci prototipova su strukturno ekvivalentni (mada kada se bavimo strukturnom ekvivalentnošću, ovi objekti su obično jednostavni objekti, što znači da oba nasleđuju od `Object.prototype`).

Strukturno ekvivalentni objekti mogu biti ili isti objekat (`o1 === o2`) ili kopije (`o1 !== o2`). Pošto se ekvivalentne primitivne vrednosti uvek upoređuju operatorom jednakosti, ne mogu se napraviti njihove kopije.

Sada se mogu formalnije definisati duboke kopije na sledeći način:

- Oni nisu isti objekat (`o1 !== o2`).
- Svojstva objekata o1 i o2 imaju ista imena u istom redosledu.
- Vrednosti njihovih svojstava su duboke kopije jedna druge.
- Njihovi prototipski lanci su strukturno ekvivalentni.

Duboke kopije mogu ali ne moraju imati kopirane prototipske lance (a često ih nemaju). Ali dva objekta sa strukturno ne-ekvivalentnim prototipskim lancima (na primer, jedan je niz, a drugi običan objekat) nikada nisu kopije jedan drugog.

Kopija objekta čija sva svojstva imaju samo primitivne vrednosti zadovoljava definicije i duboke i plitke kopije ali takvi objekti nisu od interesa u ovom kontekstu jer nas interesuje kontekst mutirajućih ugneždenih svojstava.

U JavaScript-u, standardne ugrađene operacije kopiranja objekata (spread sintaksa, `Array.prototype.concat()`, `Array.prototype.slice()`, `Array.from()`, `Object.assign()` i `Object.create()`) vrše plitko kopiranje.

Duboko kopiranje pomoću `JSON.parse(JSON.stringify())`

Jedno partikularno rešenje u JavaScript-u je da se objekti koji su **JSON-sigurni** mogu lako *duplirati* sa:

```
var new Obj = JSON.parse( JSON.stringify(someObj) );
```

Naravno, to zahteva da su objekti **JSON-sigurni** što znači **da se mogu serijalizovati u JSON string i zatim rekonstruisati u objekat iste strukture i sa istim vrednostima**. Praksa pokazuje da **većina objekata NISU JSON-sigurni objekti**.

Primer koji sledi ilustruje neke probleme koji se mogu javiti pri korišćenju `JSON.parse(JSON.stringify())`.

```
/* Primer nekih problema koji se javljaju pri korišćenju  JSON.parse(
JSON.stringify() ) */

const sampleObject = { /* originalni objekat */
  string: 'string',
  number: 123,
  boolean: false,
  null: null,
  notANumber: NaN,
  date: new Date('1999-12-31T23:59:59'),
  undefined: undefined,  /* vrednosti undefined se potpuno gube , uključujući
                            i ključ koji sadrži vrednost undefined */
  infinity: Infinity,  /* Infinity vrednosti se gube (vrednost postaje
                          'null') */
  regExp: /.a/, /* RegExp se gubi (vrednost postaje prazan objekat {}) */
}

console.log(sampleObject)
/*
Object { string: "string",
number: 123,
boolean: false,
null: null,
notANumber: NaN,
date: Date Fri Dec 31 1999 23:59:59 GMT-0500 (Eastern Standard Time),
undefined: undefined,
infinity: Infinity,
regExp: /.a/ } */

// svojstvo sa imenom undefined postoji u originalnom objektu
console.log (sampleObject.hasOwnProperty( "undefined" )) // true

console.log(typeof sampleObject.date) // object
```

```
const faultyClone = JSON.parse(JSON.stringify(sampleObject)) /* “duboka“
kopija */

console.log(faultyClone)
/*
Object { string: "string", // OK
         number: 123,  // OK
         boolean: false, // OK
         null: null, // OK
         notANumber: null, // NaN vrednosti izgubljena (postala 'null')
         date: "2000-01-01T04:59:59.000Z", // Date tip stringifikovan
         infinity: null, // Infinity vrednosti izgubljena (postala 'null')
         regExp: {} } // RegExp vrednost izgubljena (postala {}) */

// Date objekat je stringifikovan, rezultat primene .toISOString()
console.log(typeof faultyClone.date) // string

// svojstvo sa imenom undefined je izgubljeno u dubokoj kopiji
console.log (faultyClone.hasOwnProperty( "undefined" )) // false
```

Duboko kopiranje pomoću funkcije `structuredClone()`

Mogućnost koju za duboko kopiranje nudi JavaScript koja je najbliža viziji kompletne duboke kopije je globalna funkcija `structuredClone()`. Ukoliko u prethodnom primeru primenimo funkciju `structuredClone()`:

```
const correctClone = structuredClone(sampleObject) /* “duboka“ kopija */
console.log(correctClone)
```

dobićemo ispravnu duboku kopiju `correctClone` objekta `sampleObject`:

```
Object { string: "string",
number: 123,
boolean: false,
null: null,
notANumber: NaN,
date: Date Fri Dec 31 1999 23:59:59 GMT-0500 (Eastern Standard Time),
undefined: undefined,
infinity: Infinity,
regExp: /.a/ }
```

Funkcija `structuredClone()`omogućava i da se prenosivi objekti prenesu iz originalnog objekta u novi objekat. Preneseni objekti se odvajaju od originalnog objekta i vezuju se na novi objekat; oni više nisu dostupni u originalnom objektu.

Sintaksa je sledeća:

```
structuredClone(value)
structuredClone(value, options)
```

gde je `value` objekat čija se kopija pravi a `options` je niz prenosivih objekata koji će biti premešteni a ne kopirani u vraćeni objekat (duboku kopiju).

Kopiranje putem funkcije `structuredClone()` vrši se primenom algoritma strukturiranog kloniranja koji kloniranje vrši rekirzijom kroz ulazni objekat uz održavanje mape prethodno posećenih referenci da bi izbegao beskonačne cikluse obilaska. Algoritam IMA SLEDEĆA OGRANIČENJA:

- Objekti tipa `Function` ne mogu se duplicirati; pokušaj rezultuje izuzetkom `DataCloneError`.

- Pokušaj kloniranja DOM čvorova takođe rezultuje izuzetkom `DataCloneError`.
- Nek svojstva objekata se ne očuvavaju:
    - Svojstvo `lastIndex` objekta `RegExp`.
    - Ne kopiraju se ni deskriptori svojstva, seteri, geteri i slične meta-karakteristike, nego se zadržavaju podrazumevane vrednosti: na primer, ako je u originalnom objektu vrednost deskriptora svojstva `writable` bila `flase`, u kopiji će biti `true`.
    - Algoritam ne obilazi niti kopira prototipski lanac.

Bibliotečke funkcije za duboko kopiranje

Postoji više rešenja za duboko kopiranje objekata koja su sastavni deo JS biblioteka. Među njima su često korišćena rešenja iz poznatih biblioteka Lodash (metoda `_.cloneDeep()`) i Ramda (metoda `R.clone()`).

Za situacije u kojima su važne performanse preporučuje se biblioteka **Really Fast Deep Clone** (https://github.com/davidmarkclements/rfdc) za koju je objavljeno da je 400% brža od metode biblioteke Lodash.

4.1.3.3. Komponovanje objekata

Objekti se mogu i komponovati u nove objekte. Objekti dobijeni komponovanjem nazivaju se ***kompozitni objekti***. Objekti od kojih se komponuje kompozitni objekat zovu se ***podobjekti***.

Postoji mnogo vrsta objekata i mnoge strukture podataka se mogu kreirati korišćenjem kompozicije objekata. Ipak, postoje tri fundamentalne tehnike koje čine osnovu svih drugih načina komponovanja objekata. To su sledeće tehnike:

- **Konkatenacija** je tehnika kojom se kompozitni objekat formira dodavanjem novih svojstava postojećem objektu. Svojstva mogu se spajiti jedno po jedno ili kopirati iz postojećih objekata. Na ovaj način formira se novi objekat gde podobjekti ne zadržavaju svoj identitet.
- **Agregacija** je tehnika kojom se kompozitni objekat formira iz prebrojive kolekcije podobjekata tako da kompozitni objekat ***sadrži*** podobjekte. Dakle, formira se objekat koji ***sadrži*** druge objekte. Pri tome, svaki podobjekat zadržava svoj identitet, tako da se može destrukturirati iz agregacije bez gubitka informacija.
- **Delegiranje** je tehnika kojom se kompozitni objekat formira tako što objekat delegira/ prosleđuje drugom objektu. Prototipovi JavaScript-a su delegati: instanca niza prosleđuje pozive ugrađenih metoda tipa objektu Array.prototype, objekti prosleđuju na Object.prototype, itd.

4.1.3.3.1. Komponovanje objekata konkatenacijom

Jedan način za konkatenaciju objekata koji smo već videli je korišćenje funkcije `Object.assign()`:

```
let ObjekatD = Object.assign({}, ObjekatA, ObjekatB);
```

`Object.assign()` zahteva da mu se kao prvi argument prosledi odredišni objekat (objekat u koji se svojstva prepisuju). Ukoliko se taj argument izostavi, objekat na mestu prvog argumenta tretira se kao odredišni objekat i **biće mutiran**.

Isto se može uraditi i korišćenjem *operatora rasprostiranja*:

```
let ObjekatD  =  {...ObjekatA, ...ObjekatB};
```

U ovom zapisu tri tačke `...` ispred imena objekata (`ObjekatA` i `ObjekatB`) su oznaka operatora koji se naziva **operator rasprostiranja objekta** (engl. *object spread operator*). To je operator koji iterira nad svojstvima ulaznih objekta (`ObjekatA` i `ObjekatB`) i dodeljuje svojstva iz ulaznih objekata novom objektu (u primeru, objektu `ObjekatC`).

Operator prepisuje postojeća svojstva ulaznih objekata u novi objekat i, ako u ulaznom objektu naiđe na svojstvo sa imenom koje je već prepisano u novi objekat (iz nekog prethodno „obrađenog“ ulaznog

objekta), tekuću vrednost tog svojstva u novom objektu zamenjuje vrednošću svojstva na koje je naišao.

Sledeći snipet ilustruje ove situacije.

```
// Komponovanje objekata konkatenacijom
// komponentni objekti su ObjekatA i ObjekatB
const ObjekatA = {
   a: 'bar',
   b: 'oA-bar',
};

const ObjekatB = {
   b: 'oB-bar'
};

IspisiNaslov(' Kompozicija objekata ')
IspisiNaslov(' Komponentni objekat ObjekatA: ')
IspisiObjekat (ObjekatA)      // {"a": "bar",  "b": "oA-bar"}
IspisiNaslov(' Komponentni objekat ObjekatB: ')
IspisiObjekat (ObjekatB)      // {"b": "oB-bar"}

// Konkatenacija - operator rasprostiranja
let ObjekatD  =  {...ObjekatA, ...ObjekatB};
IspisiNaslov(' Konaktenacija sa operatorom rasprostiranja ')
IspisiNaslov(' Kompozitni objekat ObjekatD ')
IspisiObjekat (ObjekatD)      //{"a": "bar", "b": "oB-bar"}

// Konkatenacija sa Object.assign() sa praznim odredišnim objektom
ObjekatD = Object.assign({}, ObjekatA, ObjekatB);
IspisiNaslov(' Konaktenacija sa praznim odredišnim objektom ')
IspisiNaslov(' Kompozitni objekat ObjekatD ')
IspisiObjekat (ObjekatD)      // {"a": "bar", "b": "oB-bar"}

// Konkatenacija sa Object.assign()sa odredišnim objektom objekatA
ObjekatD = Object.assign(ObjekatA, ObjekatB);
IspisiNaslov(' Konaktenacija sa odredišnim objektom objekatA ')
IspisiNaslov(' Kompozitni objekat ObjekatD ')
IspisiObjekat (ObjekatD)      //{"a": "bar", "b": "oB-bar"}
IspisiNaslov(' Mutirani objekat ObjekatA ')
IspisiObjekat (ObjekatA)      //{"a": "bar", "b": "oB-bar"}
```

Naravno, i ovde vrednosti koje se skladište u svojstvima mogu da budu i funkcije. Evo primera:

```
function zdravoSvima () {
   console.log (' Zdravo svima!')}

function dobarDan () {
   console.log (' Dobar dan!')}

const ObjekatA = {
   a: 'bar',
   b: zdravoSvima // pokazivač na funkciju zdravoSvima
};
```

```
const ObjekatB = {
   c: 'bar',
   d: dobarDan // pokazivač na funkciju dobarDan
};

let ObjekatD = Object.assign({}, ObjekatA, ObjekatB);
IspisiNaslov(' Može i sa funkcijom ')
IspisiNaslov(' Kompozitni objekat ObjekatD ')
IspisiObjekat (ObjekatD)      // {"a": "bar", "b": "oB-bar"}
IspisiNaslov(' Pozivanje funkcija iz kompozitnog objekta ObjekatD ')
objekatD.b()            // Ispis: Zdravo svima!
objekatD.d()            // Ispis: Dobar dan!
```

4.1.3.3.2. Komponovanje objekata agregacijom

Agregacije su odlične za primenu univerzalnih apstrakcija, kao što je primena funkcije na svaki član agregata (npr. `array.map(fn)`), transformacija vektora kao pojedinačnih vrednosti, itd. Može se implementirati na širokom dijapazonu struktura (nizovi, mape, stabla, grafovi, skupovi).

Međutim, ako postoji potencijalno stotine hiljada ili milion podobjekata, obrada strima ili delegiranje su efikasniji.

Slede primeri dve agregacije.

Prva je agregacija niza a druga je agregacija povezane liste po parovima.

```
function kolekcija(a, e) { return a.concat([e])};

const objs = [
 { a: 'a', b: 'ab' },
 { b: 'b' },
 { c: 'c', b: 'cb' }
 ];

IspisiNaslov ('Ulazni objekti')
IspisiObjekat  (objs)// [{"a":  "a","b":  "ab"},{"b":  "b"},{"c":  "c",  "b":
"cb"}]

 // Agregacija niza

IspisiNaslov ('Ulazna matrica')
IspisiObjekat (objs)

const a = objs.reduce(kolekcija, []);
IspisiNaslov ('Agregacija kolekcije')
IspisiObjekat (a)  //[{"a": "a","b": "ab"},{"b": "b"},{"c": "c", "b": "cb"}]
console.log(`nabrojivi ključevi: ${ Object.keys(a) }`) // 0, 1, 2
console.log('element  a[1].b:  ',  a[1].b,' element  a[2].c:  ',  a[2].c)  /*
element a[1].b:  b  element a[2].c:  c */

// Agregacija povezane liste po parovima

function par (a, b) { return [b, a]};

IspisiNaslov ('Ulazni objekti')
IspisiObjekat  (objs) // [{"a":  "a","b":  "ab"},{"b":  "b"},{"c":  "c",  "b":
"cb"}]
```

```
const l = objs.reduceRight(par, []);
IspisiNaslov ('Agregacija povezane liste')
IspisiObjekat (l)/* [{"a": "a","b": "ab"},[{"b": "b"},[{"c": "c","b": "cb"},
                    []]]] */
console.log(`nabrojivi ključevi: ${ Object.keys(l) }`) // 0, 1
```

4.1.3.3.3. Komponovanje objekata delegiranjem

Delegiranje je kada objekat prosleđuje ili delegira drugom objektu. Omogućuje uštedu memorije i dinamičko ažuriranje velikog broja instanci.

1. Štednja memorije: U svakom trenutku može postojati potencijalno mnogo instanci objekta pa bi bilo korisno deliti identična svojstva ili metode među instancama i time smanjiti memorijske zahteve.

2. Dinamičko ažuriranje više instanci: Svaki put kada više instanci objekta treba da dele identično stanje koje će možda morati da se ažurira dinamički i da se promene trenutno odražavaju se u svakoj instanci, na primer, „majstori“ Sketchpad-a ili „pametni objekti“ Photoshop-a.

```
// Agregacija delegiranjem

const delegiraj = function (a, b) { return Object.assign(Object.create(a),
b)};

IspisiNaslov ('Ulazni objekat')
IspisiObjekat (objs) /* [{"a": "a","b": "ab"},{"b": "b"},{"c": "c", "b":
"cb"}] */

const d = objs.reduceRight(delegiraj, {});

IspisiNaslov ('Agregacija delegiranjem - komponovani objekat')
IspisiObjekat (d) //{"a": "a",  "b": "ab"}
console.log(`nabrojivi ključevi: ${ Object.keys(l) }`) // a, b
console.log('d.b: ', d.b,' d.c: ', d.c) // d.b:  ab  d.c:  c
```

4.1.3.4. Konverzija objekata u primitive

Šta se dešava U JavaScript programu kada se objekti saberu (`obj1 + obj2`), oduzmu (`obj1 - obj2`) ili prikažu pomoću `alert(obj)`?

JavaScript ne omogućava da se način rada operatora prilagodi za rad sa objektima za razliku od nekih drugih jezika poput jezika Rubi ili C++. U JavaScriptu ne može se implementirati poseban metod objekta za rukovanje sabiranjem (ili drugim operatorima).

U ovakvim slučajevima u jeziku JavaScript objekti se automatski konvertuju u primitive, a zatim se operacija izvodi nad ovim primitivama i kao rezultat se dobija primitivna vrednost.

Dakle, JavaScript ima jedno važno ograničenje jezika: rezultat `obj1 + obj2` (ili neke druge matematičke operacije) ne može biti drugi objekat! Zbog toga u JavaScriptu ne mogu da se prave objekti koji predstavljaju vektore ili matrice da se dodaju i da se očekuje „zbirni“ objekat kao rezultat. Takve stvari u JavaScriptu prosto nisu moguće - nema matematike sa objektima. Kada se to pojavi u kodu, uz vrlo retke izuzetke (na primer rad sa objektom `Date`), to je zbog greške u kodiranju. Zbog toga se mora razumeti kako se objekat konvertuje u primitivu

U JavaScriptu važe sledeća pravila za konverziju objekata.

- Nema konverzije u `boolean`. Svi objekti su istiniti u logičkom kontekstu.

- Konverzija u tip `numeric` se dešava kada se oduzimaju objekti ili se primenjuju matematičke funkcije. Na primer, datumski objekti (`Date`) mogu se oduzimati, a rezultat je vremenska razlika između dva datuma.
- Što se tiče konverzije u `string`, ona se obično dešava pri ispisu objekta sa `alert(obj)` i u sličnim kontekstima.

Konverzije u `numeric` i `string` mogu se implementirati korišćenjem specijalnih metoda tipa `object`.

Tehnički, u JavaScriptu postoje tri vrste konverzije neprimitivnog tipa u primitivni koje se primenjuju u različitim situacijama. Oslanjaju se na koncept zvani **nagoveštaj** (engl. *hint*) čiji opis postoji na linku https://tc39.github.io/ecma262/#sec-toprimitive. Postoje tri nagoveštaja: *string*, *number* i *default*.

Nagoveštaj *string* se koristi za konverziju objekat-u-string kada se nad objektom izvršava operacija koja očekuje tip `string` kao operand. Na primer, funkcija `alert ()` za ispis objekta, ili korišćenje objekta kao ključa objekta:

```
// Ispis objekta
alert(obj);

// Korišćenje objekta kao ključa svojstva (drugog objekta)
anotherObj[obj] = 123;
```

Nagoveštaj *number* se koristi za konverziju objekat-u-broj kada se nad objektom izvršava operacija koja očekuje tip `number` kao operand. Na primer, aritmetičke operacije, operacije poređenja numeričkih vrednosti, odnosno funkcije koje sadrže izraze navedene vrste:

```
/ eksplicitna konverzija
let num = Number(obj);

// matematičke operacije (izuzev binarnog operatora plus)
let n = +obj; // unarno plus
let delta = date1 - date2;

// poređenja veće/manje
let greater = user1 > user2;
```

Nagoveštaj *default* se koristi za konverziju objekat-u-nešto kada operator "nije siguran" koji primitivni tip očekuje (može da se izvršava nad različitim primitivnim tipovima). Na primer, binarno plus može da se izvršava nad numeričkim vrednostima i tada je rezultat zbir tipa `number`, ali može da se izvršava i nad stringovima i tada je rezultat spojeni string tipa `string`. Isto je i u slučaju poređenja objekta sa stringom, brojem ili simbolom korišćenjem operatora **==**. Evo primera:

```
// Binarno plus koristi "default" nagoveštaj
let total = obj1 + obj2;

// obj == number koristi "default" nagoveštaj
if (user == 1) { ... };
```

Precizno, JavaScript radi konverziju po sledećem algoritmu:

1. Poziva `obj[Symbol.toPrimitive](hint)` – metodu sa simboličkim ključem `Symbol.toPrimitive` (sistem simbol), ako takva metoda postoji,
2. Ukoliko metoda sa simboličkim ključem `Symbol.toPrimitive` ne postoji a nagoveštaj je "string", pokušava se poziv metode `obj.toString()` ili `obj.valueOf()`, koja god da postoji.
3. Opet, Ukoliko metoda sa simboličkim ključem `Symbol.toPrimitive` ne postoji a nagoveštaj je "number" ili "default", pokušava se poziv `obj.valueOf()` ili `obj.toString()`, koja god da postoji.

Sada ćemo malo detaljnije objasniti ovaj algoritam.

Metoda 1. Postoji ugrađeni simbol sa imenom `Symbol.toPrimitive` koji treba koristiti za imenovanje metode konverzije na sledeći način:

```
obj[Symbol.toPrimitive] = function(hint) {
  // Ovde dođe kod koji objekat konvertuje u primitivu
  // on mora da vrati primitivnu vrednost
  // hint = jedna od sledećih vrednosti "string", "number", "default"
};
```

Ako metoda `Symbol.toPrimitive` postoji, ona se koristi za sva tri nagoveštaja i druge metode nisu potrebne.

Evo i primera:

```
let korisnik = {
  ime: "Pera",
  plata: 1000,

  [Symbol.toPrimitive](hint) {
    alert(`hint: ${hint}`);
    return hint == "string" ? `{ime: "${this.ime}"}` : this.plata;
  }
};

// Demo konverzije:
alert(korisnik); // hint: string -> {ime: "Pera"}
alert(+korisnik); // hint: number -> 1000
alert(korisnik + 500); // hint: default -> 1500
```

Dakle, preporučuje se da se u novom kodu koristi metoda `Symbol.toPrimitive`.

Metode 2 i 3 potiču iz vremena kada metoda `Symbol.toPrimitive` nije postojala. Za razliku od metode `Symbol.toPrimitive`, one ne podržavaju rad sa svim nagoveštajima. Metoda 2 namenjena je za rad sa nagoveštajem `string`, a metoda 2 za rad sa nagoveštajem `number` i `default`. Obe mogu da pozovu iste metode objekta (`obj.toString()` ili `obj.valueOf()`), samo im je različita preferenciji metode koju će pozvati u slučaju da obe metode postoje. Ukoliko je nagoveštaj `string`, razumljivo se preferira metoda `obj.toString()`, a ako je nagoveštaj `number` preferira se metoda `obj.valueOf()`.

Sve pomenute metode moraju da vrate primitivnu vrednost. Metode (`obj.toString()` i `obj.valueOf()` su metode koje podrazumevano ima svaki običan objekat. Međutim, te podrazumevane implementacije nisu baš od neke koristi baš zbog povratne vrednosti: metoda `toString` vraća string "[object Object]", dok metoda `valueOf` vraća sam objekat. Da bi se dobilo nešto upotrebljivo, treba napraviti prilagođene metode. Sledeći kod ilustruje kako bi to izgledalo pa da konverzija objekta u string liči na ono što se dobija metodom `Symbol.toPrimitive`:

```
let korisnik = {
  ime: "Jova",
  plata: 1000,

  // za hint="string"
  toString() {
    return `{ime: "${this.ime}"}`;
  },
```

```
  // za hint="number" ili "default"
  valueOf() {
    return this.plata;
  }

};

alert(korisnik); // toString -> {ime: "Jova"}
alert(+korisnik); // valueOf -> 1000
alert(korisnik + 500); // valueOf -> 1500
```

Važne napomene!!!

1. Jedina obavezna stvar u metodama konverzije je da ove metode **moraju da vrate primitivnu vrednost**, **a ne objekat**.
2. Metode primitivne konverzije **ne vraćaju nužno „nagoveštenu“ primitivnu vrednost**, odnosno ne postoji grancija da će `toString` da vrati tačno `string` niti da će metoda `Symbol.toPrimitive` vratii broj za nagoveštaj „broj“.
3. Zbog vrlo slabe podrške za rukovanje greškama u JavaScriptu(danas je bolje, ali ne mnogo bolje), ako `toString` ili `valueOf` vrate objekat, ne prijavljuje se greška, ali se takva vrednost ignoriše (kao da metoda ne postoji). `Simbol.toPrimitive` je stroži: mora da vrati primitivu, inače dolazi do greške.

U realnim uslovima najčešće se javljaju višestruke implicitne konverzije kao što je, na primer, slučaj sa operacijom množenja (*****) koja konvertuje operande u brojeve. Ako se operatoru množenja prosledi objekat kao operand, proračun se odvija u dve faze:

- Objekat se konvertuje u primitivnu vrednost (primenom opisanih pravila).
- Ako je potrebno za dalje proračune, dobijena primitivna vrednost se dalje konvertuje.

Na primer:

```
let obj = {
  // Ovde toString radi sve konverzije objekta, ne koriste se druge metode
  toString() {
    return "2";
  }
};

alert(obj * 3); /* 6;
```

U ovom primitivnom primerčiću dešava se gomila konverzija: Prvo se `obj` konvertuje u `string` primitivu "2"; zatim se `string` primitiva "2" konvertuje u `number` primitivu 2 koja se množi se vrednošću 3 tipa `number` i vraća se vrednost 6 tipa `number`. Na kraju se vrednost 6 tipa `number` konvertuje u `string` primitivu "6" i ona se ispisuje .

U sledećom slučaju operacije sabiranja (**+**) konverzija uopšte nema zato što operacija sabiranja prihvata i tip string kao ulaz. Ako poslednju liniju koda zamenimo sa `alert(obj + "6")`, dobijamo:

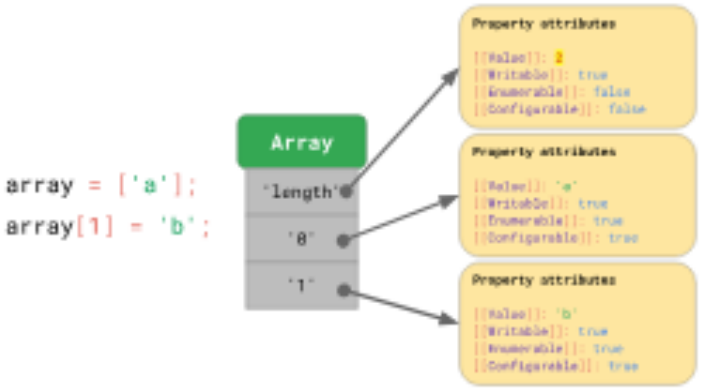
```
alert(obj + 3); // "26".
```

4.1.4. Nizovi

Objekti ne pružaju mogućnost manipulisanja uređenim kolekcijama podataka. Zato za rad sa uređenim kolekcijama podataka u JavaScript-u postoji poseban tip `Array` koji je **uređena** lista vrednosti, niz.

Nizovi predstavljaju ***uređene*** kolekcije elemenata. Nad elementima kolekcije predstavljene nizom postoji relacija uređenja – zna se koji je element niza "ispred" nekog drugog elementa tog istog niza, odnosno zna se "pozicija" svakog elementa u nizu. Pozicija elementa određena je njemu odgovarajućim *indeksom*. Vrednosti indeksa su iz skupa { 0, 1, 2, ... }. Vrednost elementa niza može da bude bilo koji tip Jezika JavaScript – primitivni tipovi i neprimitivni tip `Object` sa svim svojim podtipovima (niz, funkcija). Pojedinačni elementi mogu da budu različitog tipa.

Tip `Array` je podtip tipa `Object`. U stvari, Tip `Array` je objekat sa određenim specifičnostima u odnosu na jednostavan objekat. To se najbolje vidi ako se pogleda način na koji JavaScript implementira tip `Array` što je ilustrovano slikom 5.3.



Slika 5.3 Implementacija tipa `Array` u JavaScript-u [https://mathiasbynens.be/notes/shapes-ics]

4.1.4.1. Sintaksa

Za nizove se takođe koriste dve sintakse, ***konstruktorska sintaksa*** i ***sintaksa srednje zagrade*** **`[]`**.

Konstruktorska sintaksa ima sledeći oblik:

```
decl ime_niza = new Array([e1, e2, e3, . . ., en])
```

`decl = const|let|var`	obavezna ključna reč (deklaracija);
`ime_niza`	obavezan identifikator, treba da zadovoljava sintaksne zahteve jezika;
`new`	obavezna ključna reč;
`Array`	obavezna ključna reč;
`e1, e2, e3, . . ., en`	opciona lista elemenata; bilo koji JavaScript tip; dozvoljeno je "mešanje" tipova.

Primer:

```
let nekiObjekat = {
    k1: "limun",
    k2: 236.12,
    k3: saberi,
}

function saberi(x,y) {
   return x+y
}

let arr1 = new Array();
let arr2 = new Array(1, 2, 3 , 4);
let arr3 = new Array([1, 2], "jabuka" , 3 , 4)
```

```
let arr4 = new Array([1, 2], "jabuka" , 3 , 4, nekiObjekat, saberi)
let arr5 = new Array([1, 2], "jabuka" , 3 , 4, nekiObjekat,
                     saberi(nekiObjekat.k1,nekiObjekat.k2))
```

Ova sintaksa se ređe koristi jer je sintaksa srednje zagrade kondenzovanija. A ima i jednu škakljivu osobinu: Ako se `new Array` pozove sa jednim argumentom koji je broj, kreira se niz u kome stavke nema, a svojstvo `length` je ažurirano na vrednost 2. Evo primera:

```
let arr = new Array(7); // Tebalo bi da se kreira niz [7]
alert( arr[0] ); // undefined! nema elementa.
alert( arr.length ); // length 2
```

Sintaksa srednje zagrade ima sledeći oblik:

```
decl ime_niza = Array([e1, e2, e3, . . ., en])
```

`decl = const|let|var`	obavezna ključna reč (deklaracija);
`ime_niza`	obavezan identifikator, treba da zadovoljava zahteve leksičke sintakse jezika;
`Array`	obavezna ključna reč;
`e1, e2, e3, . . ., en`	opciona lista elemenata; bilo koji JavaScript tip; dozvoljeno je "mešanje" tipova.

`Prethodni primer` "prepisan" u sintaksu srednje zagrade izgleda ovako:

```
let arr1 = [];
let arr2 = [1, 2, 3 , 4];
let arr3 = [[1, 2], "jabuka" , 3 , 4]
let arr4 = [[1, 2], "jabuka" , 3 , 4, nekiObjekat, saberi]
let arr5 = [[1, 2], "jabuka" , 3 , 4, nekiObjekat,
            saberi (nekiObjekat.k1, nekiObjekat.k2)]
```

Kao što je već pomenuto, nizovi imaju malo strukturiraniju organizaciju skladištenja informacija (pri čemu ne postoji restrikcija na tip skladištenih informacija). Rečeno je i da nizovi koriste *numeričko indeksiranje* što znači da se vrednosti zapisuju u lokacije, uobičajeno zvane *indeksi*, koje mi vidimo kao nenegativne celobrojne vrednosti (**0** i pozitivne celobrojne vrednosti). Evo primera koji to ilustruje:

```
let mojNiz = [ "foo", 42, "bar" ];
mojNiz;
```

Ovde se linija **`mojNiz;`** evaluira na objekat

```
0: "foo"
1: 42
2: "bar"
length: 3
[[Prototype]]: Array(0)
```

Ovde svojstva **`0 , 1`** i **`2`** sa imenima nenegativnih celobrojnih vrednosti (string reprezentacije tih celih brojeva) u sebi skladište elemente niza.

U objektu je i ugrađeno svojstvo `length` koje u sebi (treba da) skladišti broj elemenata u nizu. U stvari, to je vrednost indeksa poslednjeg indeksnog elementa u nizu kome je dodeljena vrednost uvećan za 1.

Još jedna stvar u vezi sa svojstvom `length` je da mu se vrednost može manuelno modifikovati. Ako se poveća ručno, neće se desiti ništa opasno. Ali ako se smanji, stvari postaju "zanimljive" - niz se nepovratno skraćuje – gube se elementi čiji je indeks bio veći od nove `length` vrednosti. Najlakši način da "ispraznite" **`mojNiz`** je da napišete sledeću liniju u kodu: **`mojNiz.length=0`**. Kada se ona izvrši, više nema ničega u nizu **`mojNiz`**.

Nizovi ***jesu objekti*** pa, iako je svaki indeks (ključ) nenegativan ceo broj, **mogu se nizu dodati i druga svojstva *sa ključevima koji nisu celobrojne vrednosti*** kao u sledećem fragmentu koda:

```
let mojNiz = [ "foo", 42, "bar" ];
mojNiz.baz = "baz";

mojNiz;
```

Sada je `mojNiz` objekat koji izgleda ovako:

```
    ['foo', 42, 'bar', baz: 'baz']
         0: "foo"
         1: 42
         2: "bar"
         baz: "baz"
         length: 3
         [[Prototype]]: Array(0)
```

Dakle, niz mojNiz je objekat koji ima 4 svojstva **`0` , `1` , `2`** i **`baz`** ali mu je vrednost **`length`** svojstva jednaka **3**. To znači da dodavanje imenovanih svojstava ne menja vrednost svojstva dužine niza (svojstvo `length`); **dužina niza jednaka je broju indeksnih svojstava** i **ne uključuje ne-indeksna svojstva**.

Budite oprezni: Ako pokušate da dodate svojstvo nizu i, pri tome, ime tog svojstva ***izgleda*** *kao broj*, ono će da završi kao numerički indeks (a time će modifikovati i sadržaj niza). Evo primera:

```
let mojNiz = [ "foo", 42, "bar" ];
mojNiz ["3"] ="baz"; // Ovo svojstvo je indeksno, sa indeksom 3

mojNiz.length; // 4 – dodato je indeksno svojstvo
mojNiz;
/* objekat mojNiz sada izgleda ovako:
   ['foo', 42, 'bar', 'baz']
         0: "foo"
         1: 42
         2: "bar"
         3: "baz"
         length: 4
         [[Prototype]]: Array(0) */
```

4.1.4.2. Operacije sa nizovima

4.1.4.2.1. Pribavljanje i postavljanje vrednosti

Pribavljanje i postavljanje vrednosti indeksiranih elementa niza vrši se navođenjem imena niza i indeksa elementa u srednjoj zagradi:

```
let mojNiz = [ "foo", 42, "bar" ];
mojNiz.baz = "baz";
mojNiz[0]      // evaluira se na "foo"
mojNiz[1]      // evaluira se na 42
mojNiz[2]      // evaluira se na "bar"
// sledeće linije modifikuju mojNiz u ["bar-bar",142,"bar-bar"]
mojNiz[0] = "bar-bar"
mojNiz[1] = 142
mojNiz[2] = "bar-bar"
```

```
/* sledeća linija dodaće elemenat sa vrednošću 505 na kraj niza i ostaviće
"rupu"
   na indeksu 3 u kojoj će biti vrednost undefined. length će da bude 5.
*/
mojNiz[4] = 505
/* Sada je mojNiz = ["bar-bar",142,"bar-bar", undedfined, 505] */
```

Poslednji element niza može se pribaviti na osnovu vrednosti svojstva `length` (`ime_niza.[ime_niza.length - 1]`) a postoji i lepša sintaksa `ime_niza.at(-1)`:

```
mojNiz = ["bar-bar",142,"bar-bar", undedfined, 505]
mojNiz.at(-1)  // evaluira se na 505
```

4.1.4.2.1.1. Metode pop/push, shift/unshift

Najčešća praktična primena niza je ***red***. Da se podsetimo: red je skup entiteta koji se održavaju u sekvenci i mogu se modifikovati dodavanjem entiteta na jednom kraju sekvence i uklanjanjem entiteta sa drugog kraja sekvence. To znači da tip koji odgovara nizu treba da podržava dve operacije:

- ***Dodavanje/puš*** (*push*) dodaje element na kraj.
- ***Pomeranje/šift*** (*shift*) pomera element sa početka, preuređujući red tako da drugi element postaje prvi element.

Tip `Array` u JavaScript-u podržava obe operacije.

Još jedna važna primena tipa `Array` je stek gde se elementi dodaju na kraj (vrh) i uklanjaju sa kraja (vrha). Operacija dodavanja se ovde uobičajeno zove ***puš*** a operacija uklanjanja se zove ***pop***.

U stvari, Array tip u JavaScript-u odgovara apstraktnoj strukturi zvanoj ***dek*** (*deque*) gde se operacija dodavanja/uklanjanja može raditi sa oba kraja.

Za rad sa početkom niza na raspolaganju su ugrađene metode `shift()` i `unshift()`, a za rad sa krajem niza na raspolaganju su ugrađene metode `push()` i `pop()`. Evo i ilustrativnih primera:

```
// pimer za push – dodavanje na kraj niza
let voce = ["Jabuka", "Dunja"];
voce.push("Kajsija");
alert( voce ); // Jabuka, Dunja, Kajsija

// pimer za pop – uklanjanje sa kraja niza
let voce = ["Jabuka", "Dunja"," Kajsija" ];
voce.pop();
alert( voce ); // Jabuka, Dunja

// pimer za unshift – dodavanje na početak niza
let voce = ["Jabuka", "Dunja"];
voce.unshift("Kajsija");
alert( voce ); // Kajsija ,Jabuka, Dunja

// pimer za shift – uklanjanje sa početka niza
let voce = ["Jabuka", "Dunja","Kajsija" ];
voce.shift();
alert( voce ); // Dunja, Kajsija

// push i unshift mogu da rade sa više elemenata odjednom
let voce = ["Jabuka"];
voce.push("Dunja", "Kajsija");
voce.unshift("Ananas", "Limun");
```

```
alert( voce ) // Ananas, Limun, Jabuka, Dunja, Kajsija
```

Mogao bi se, naravno, koristiti niz kao kolekcija običnih objekata ključ/vrednost bez numeričkih indeksa, ali to nije dobra ideja zbog toga što nizovi imaju ponašanje i optimizacije specifične za njihovu nameravanu upotrebu, baš kao što je i slučaj sa običnim objektima. Objekte treba koristiti za skladištenje parova ključ/vrednost, a nizove za skladištenje vrednosti po numeričkim indeksima. I još jedna napomena važna za performansu u radu sa nizovima: `push()` i `pop ()` rade brže nego `unshift()` i `shift()`.

4.1.4.2.2. Iteriranje nad nizovima

Za iteriranje nad nizovima u JavaScript-u mogu se tehnički koristiti tri konstrukta: **`for`** petlja nad indeksima, **`for..of`** petlja i **`for..in`** petlja.

`for` petlja nad indeksima se preporučuje ako je potrebno, pored vrednosti elementa, imati i podatak o njegovom indeksu. Kao u sledećem primeru:

```
let gajbeVoca = ["jabuka", "banana", "jagoda"];
for (let i = 0; i < arr.length; i++) {
  console.log( "U gajbi ", i, "je voće ", gajbeVoca [i] );
}
```

`for..of` petlja je optimizovana za rad sa nizovima ali vraća samo vrednost elementa a ne i vrednost indeksa. Primer korišćenja je:

```
let korpaVoca = ["jabuka", "banana", "jagoda"];
for (let vocka of korpaVoca) {
  console.log( "U korpi sa voćem ima  ", vocka);
}
```

Kako je niz objekat, može se koristiti i **`for..in`** petlja kao u sledećem primeru:

```
let studentiGrupa = ["Marko", "Jovan", "Jagoda"];
for (let clan in studentiGrupa) {
  console.log( "Član grupe je student ", studentiGrupa[clan]);
}
```

Međutim, korišćenje **`for..in`** petlje za rad sa nizovima nije dobra ideja iz sledećih razloga:

- Petlja `for..in` "prolazi" kroz sva svojstva, a ne samo indeksna. U pretraživačima i drugim okruženjima postoje takozvani "nizoliki" objekti koji izgledaju kao nizovi, ali nisu pravi nizovi. Oni imaju indeksna svojstva i `length` svojstvo, ali mogu imati i neindeksna svojstva i metode koje obično nisu potrebne za rad sa nizovima. Petlja `for..in` će "proći" i kroz neindeksna svojstva što može izazvati problem.
- Konačno, petlja `for..in` je optimizovana za generičke objekte, a ne za nizove, i stoga je 10-100 puta sporija. To ne mora da bude problem, ali može da bude problem.

Generalno, ne bi trebalo koristiti `for..in` za nizove.

4.1.4.2.2.1. Višedimenzioni nizovi

Elementi niza mogu biti nizovi što se može iskoristiti za predstavljanje matrica:

```
let matrix = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];

console.log( matrix[1][1] ); // 5, prvi element
```

4.1.4.2.3. Konverzija u string

Nizovi imaju sopstvenu implementaciju metode `toString()` koja vraća listu elemenata razdvojnih zarezima:

```
let arr = [1, 2, 3];

alert( arr ); // 1,2,3
alert( String(arr) === '1,2,3' ); // true
```

Tip Array nema metodu `Symbol.toPrimitive`, niti održivo `valueOf`, implementira samo `toString` konverziju tako da se **`[]`** konvertuje u prazan string **`""`**, **`[1]`** postaje **`"1"`** a **`[1,2]`** postaje **`"1,2"`**. Operator binarno plus "**+**" pri radu sa stringovima vrši konverziju drugog "sabirka" u string i zatim spaja prvi i drugi podstring kao u sledećem primeru:

```
alert( "" + 1 ); // "1"
alert( "1" + 1 ); // "11"
alert( "1,2" + 1 ); // "1,21"
```

4.1.4.2.4. Poređenje nizova

Vrednosti tipa Array u JavaScript-u ne bi trebalo porediti korišćenjem operatora **==**.

Podsetimo se: operator **==** u JavaScript-u konvertuje operande na isti tip pre poređenja, osim za tipove **`undefined`** i **`null`** koji su jednaki (**==**) jedan drugom i ničemu više. Dva objekta su jednaka (**==**) samo ako su reference na isti objekat. Ako je jedan argument tipa **`Object`** a drugi je ***primitivni tip***, poređenje **==** radi tako što prvo objekat konvertuje u primitivu pa zatim poredi dve primitivne vrednosti.

U stvari, ako se nizovi porede pomoću operatora **==**, ispada da su dva niza jednaka samo ako su u pitanju dve varijable koje referenciraju tačno isti niz. Ukoliko to nije slučaj, poređenje nizova pomoću operatora može proizvede vrlo "zanimljive" rezultate. Slede karakteristični primeri.

Na primer:

```
alert( [] == [] ); // false
alert( [0] == [0] ); // false
```

Nizovi koji se ovde porede su tehnički različiti objekti. Stoga NISU jednaki. Operator == ne radi poređenje stavku po stavku i njemu ništa ne znači što su ti objekti isti "iznutra", važno mu je samo "ime". Kao kada biste imali dva identična automobila sa različitim serijskim brojevima.

Poređenje sa primitivama je takođe zanimljivo:

```
alert( 0 == [] ); // true
alert('0' == [] ); // false
```

Ovde se u oba slučaja porede primitive sa `Array` objektom. Zbog toga se u obe linije koda prvo objekat `[]` konveruje u primitivu i kao rezultat se dobija prazan string (**`''`**).

Nakon toga se u liniji `alert( 0 == [] );` prazan string konvertuje u `0` zato što je prvi operand numerik. Konačno, **`0 == 0`** se evaluira na `true`.

U drugoj liniji, tipovi operanada su isti (`String`) pa nema konverzije. To znači da se ovde evaluira izraz **`'0' == ''`** što za rezultat da je `false`.

Striktna jednakost (**===**) je čak manje upotrebljiva jer ne konvertuje tipove.

Zaključak je da nizove u celini ne treba porediti korišćenjem operatora poređenja po jednakosti (**==** ili **===**), a ni operatora **>**, **<** i ostalih iz "familije"; šta bi, uopšte značilo da je jedan niz veći (**>**) od drugog? Treba ih porediti u `for ... of` petlji, stavku po stavku.

4.1.5. Destrukturiranje

I nizovi i objekti u JavaScript-u mogu se destrukturirati, odnosno vrednosti koje su u njima uskladištene mogu se ekstrahovati i dodeljivati (drugim) imenovanim varijablama. Ovde ćemo prikazati neke zgodne skraćenice za tu namenu.

Primer za nizove (dodeljujemo vrednosti elemenata niza `['a', 'b']` varijablama `t` i `u`) :

```
const[t, u] = ['a', 'b'];
console.log(t); // Ispis: a
console.log(u); // ispis: b

const[t, u, v] = ['a', 'b'];
console.log(t); // Ispis: a
console.log(u); // ispis: b
console.log(v); // ispis: undefined
```

Primer za objekte:

```
const blep = {
blop: 'blop-vrednost',
};

const { blop } = blep; // ekvivalentno sa: const blop = blep.blop;
blop; // Ispisaće: blop-vrednost
```

Baš kao i u gornjem primeru sa tipom `Array`, moguće je odjednom destrukturirati višestruke dodele za objekte:

```
const blep = {
blop: 'blop-vrednost',
plop: 'plop-vrednost',
};

const { blop, plop } = blep; /* ekvivalentno sa:
const blop = blep.blop;
const blop = blep.plop; */
console.log(blop); // Ispisaće: blop-vrednost
console.log(plop); // Ispisaće: plop-vrednost
```

Ako svojstva nema u objektu, vratiće `undefined`:

```
const blep = {
blop: 'blop-vrednost',
plop: 'plop-vrednost',
};

const { blop, plop, plip } = blep; /* ekvivalentno sa:
const blop = blep.blop;
const plop = blep.plop;
const plip = blep.plip;                          */
console.log('ja sam blop: ', blop); // Ispisaće: ja sam blop:blop-vrednost
console.log('ja sam plop: ', plop); // Ispisaće: ja sam plop:plop-vrednost
console.log('ja sam plip: ', plip); // Ispisaće: ja sam plip:undefined
```

Ako se želi koristiti drugačije ime za novo povezivanje, moguće je dodeliti novo ime na sledeći način:

```
const blep = {
```

```
blop: 'blop-vrednost',
plop: 'plop-vrednost',
};
const{ blop: bloop } = blep;
bloop; // 'blop-vrednost'
```

Značenje: Dodeli **`blep.blop`** kao **`bloop`**.

Za sada toliko o operacijama nad nizovima. Kada budemo govorili o funkcionalnom programiranju u jeziku JavaScript, posebnu pažnju ćemo posvetiti još trima izuzetno značajnim operacijama nad nizovima: `map()`, `reduce()` i `filter()`.

4.1.6. Iteriranje u JavaScriptu: objekat `Iterator`

Iteriranje je jedan od najvažnijih koncepata u programskim jezicima koji omogućuje repetitivno izvršavanje delova koda. U kodu se iteriranje predstavlja različitim jezičkim konstruktima a samo iteriranje je u programiranju standardizovano putem koncepta ***protokol iteriranja***.

Objekat `Iterator` u jeziku JavaScript je objekat koji je u skladu sa ***protokolom iteriranja*** tako što obezbeđuje metodu `next()` koji vraća objekat rezultata iteratora.

All built-in iterators inherit from the Iterator class. The Iterator class provides a @@iterator method that returns the iterator object itself, making the iterator also iterable. It also provides some helper methods for working with iterators.

4.2. Povezani objekti u JavaScript-u

Tema ovoga poglavlja je osnovni mehanizam koji JavaScript pruža za povezivanje objekata. To je veoma bitna tema za razumevanje načina funkcionisanja objekata u JavaScript-u (a u JavaScript-u „skoro sve je objekat“) i zato će biti posebno detaljno obrađena.

4.2.1. Povezivanje objekata

U objektnoj paradigmi objekti su osnovne elementi programa koji svoju funkcionalnost ostvaruje kroz saradnju pojedinačnih objekata. Da bi se takva saradnja ostvarila, neophodno je obezbediti mehanizme kojima će objekti moći međusobno da sarađuju – da komuniciraju i da dele ponašanja i podatke.

Komunikacija među objektima ostvaruje se razmenom poruka a deljenje ponašanja i podataka opisuje se međusobnim vezama objekata. Za opisivanje veza među objektima postoje dva osnovna obrasca, klasni obrazac i prototipski obrazac.

OOP zasnovano na klasama je najrašireniji stil OOP-a u kome je ponašanje objekta u svakoj situaciji njegovog „života“ unapred određeno klasom koji skladišti nacrt ili šablon objekata (njihovu strukturu i međusobne veze). U njemu se međuzavisnosti objekata opisuju hijerarhijskom taksonomijom klasa-podklasa i mehanizmom klasnog nasleđivanja. Zajednička svojstva kodiraju se u klasi sa idejom da se opišu svi scenariji ponašanja koje objekat može da ispolji. Objekat se kreira instanciranjem klase, postupkom u kome svaki objekat dobija svoju kopiju nasleđenih svojstava i, eventualno, neka specifična sopstvena svojstava. U takvom modelu, ponašanje objekta ne može se promeniti bez promene klase.

Drugi stil OOP je programiranje zasnovano na prototipu. U njemu je metodologija programiranja usmerena na ono što objekat radi, a ne na ono što će, možda, raditi u nekoj situaciji. Ovde je

nasleđivanje povezano sa „živim“ radnim objektom koji se zove ***prototip*** kome pristupaju svi objekti koji sa njim dele neko ponašanje i/ili podatke. Objekti ne dobijaju kopije nasleđenog ponašanja, oni dobijaju prototip objekta koji se može i menjati. Ovaj stil ne formira čvrstu i rigidnu taksonomiju. Smatra se da je efikasan koliko i nasleđivanje zasnovano na klasi.

Jezik JavaScript je prototipski baziran objektni jezik.

https://enlear.academy/prototype-vs-class-the-javascript-split-personality-20d984d2fe2

4.2.2. Interno svojstvo `[[Prototype]]` i prototipsko nasleđivanje

Objekti u JavaScript-u imaju interno svojstvo koje je u specifikaciji označeno sa `[[Prototype]]` što je, prosto, referenca na drugi objekat. Svim[18] objektima se pri kreiranju dodeljuje ne-`null` vrednost za to svojstvo.

Posmatrajmo kod koji kreira objekat `mojObjekat`:

```
const mojObjekat = {
        a:2
};
console.log (mojObjekat.a); // 2
mojobjekat;
```

Objekat `mojObjekat` (iz našeg aspekta interesovanja) izgleda ovako:

a: 2
[[Prototype]]: Object

Dakle, objekat `mojObjekat` ima dva svojstva: svojstva **a** sa vrednošću 2 i svojstvo`[[Prototype]]` koje je referenca/ pokazivač na neki objekat.

Za šta se koristi `[[Prototype]]` referenca?

Operacija `[[Get]]` je operacija koja se poziva kada se referencira svojstvo objekta, kao što je `mojObjekat.a`. Prvi korak podrazumevane `[[Get]]` operacije je da **proveri da li sam objekat ima svojstvo a u sebi**, i **ako ga ima ono će da se koristi** (recimo, da se pribavi njegova vrednost).

Dakle, možemo da zaključimo da u situaciji u kojoj sam objekat ima traženo svojstvo, interno svojstvo `[[Prototype]]` ne služi ničemu. Ulogu `[[Prototype]]` svojstvo objekta dobija ako sam objekat (objekat `myObject` u primeru) **nema svojstva a u sebi**, pa ćemo mi sada da se time pozabavimo.

Podrazumevana `[[Get]]` operacija, u slučaju kada objekat nema traženo svojstvo, funkcioniše tako što prelazi na **sledeći objekat** i u tom sledećem objektu traži željeno svojstvo. A **sledeći objekat** je upravo **objekat na koji pokazuje vrednost `[[Prototype]]` svojstva** posmatranog objekta . Sledeći kod to ilustruje:

```
let tudjiObjekat = {
        a:3
};

// kreiranje novog objekta (mojObjekat) povezanog na tudjiObjekat
let mojObjekat = Object.create(tudjiObjekat);
console.log (mojObjekat)
```

[18] Moguće je da objekat ima prazan `[[Prototype]]` pokazivač. Objekat sa praznim `[[Prototype]]` pokazivačem ima prototipski lanac koji je “prazan” , on ništa ni od koga ne nasleđuje. Retko se koristi, a jedna primena mu je jeftina zamena za `map`-e.

```
console.log(mojObjekat.a); // 3
```

Napomena: Objasnićemo uskoro šta tačno radi `Object.create()`. Za sada samo pretpostavite da on kreira objekat (u primeru `mojObjekat`) sa `[[Prototype]]` vezom na specificirani objekat (objekat u listi argumenata funkcije `Object.create()`; u primeru je to `tudjiObjekat`).

Dakle, imamo objekat `mojObjekat` koji je sada svojstvom `[[Prototype]]` povezan na objekat `tudjiObjekat`. Objektu `mojObjekat` nismo dodelili nikakva svojstva pa svojstvo `mojObjekat.a` u stvari ne postoji (naredba `console.log (mojObjekat)` ispisaće prazan objekat). No, bez obzira na to, pristup svojstvu uspeva jer je svojstvo nađeno u objektu `tudjiObjekat` gde ima vrednost 3.

Ako se svojstvo `a` ne nađe ni u objektu `tudjiObjekat`, gleda se svojstvo `[[Prototype]]` objekta `tudjiObjekat` i, ako nije prazno, svojstvo **`a`** se traži u objektu na koji pokazuje svojstvo `[[Prototype]]` objekta `tudjiObjekat`. Taj proces se nastavlja sve dok se u nekom od objekata na koje se „nailaz"i ne pronađe svojstvo sa imenom **`a`**, ili dok se ne dođe do objekta koji nema `[[Prototype]]` svojstvo. Ako se u ovom procesu **ne pronađe traženo svojstvo**, povratni rezultat iz `[[Get]]` operacije je **`undefined`** kao u sledećem primeru:

```
let tudjiObjekat = {
        a:3
};

// kreiranje novog objekta (mojObjekat) povezanog na tudjiObjekat
let mojObjekat = Object.create(tudjiObjekat);
console.log (mojObjekat)
console.log(mojObjekat.b); // undefined jer svojstva b nema ni u prototipu
```

Objekti koji su međusobno povezani svojstvom `[[Prototype]]` čine ***prototipski lanac***.

Ako se koristi **`for..in` petlja** za iteriranje nad objektom, svojstvo do koga se može doći putem prototipskog lanca objekta mora biti nabrojivo (deskriptor `enumerable` mu je `true`) da bi bilo uvršteno u petlju.

Sledeći kod ilustruje slučaj nabrojivih svojstava:

```
let tudjiObjekat = { // sva svojstva su nabrojiva po definiciji
    a:3,
    b:4,
    c:5,
};
// Da se uverimo da su nabrojiva
let descriptor = Object.getOwnPropertyDescriptor( tudjiObjekat, "a" );
console.log("Svojstvo a: \n", JSON.stringify(descriptor, null, 2))
descriptor = Object.getOwnPropertyDescriptor( tudjiObjekat, "b" );
console.log("Svojstvo b: \n", JSON.stringify(descriptor, null, 2))
descriptor = Object.getOwnPropertyDescriptor( tudjiObjekat, "c" );
console.log("Svojstvo c: \n", JSON.stringify(descriptor, null, 2))

// kreiranje objekta povezanog na tudjiObjekat
var mojObjekat = Object.create( tudjiObjekat );

for (var k in mojObjekat) {
        console.log("našao: ", k);
}
// našao: a, b, c
```

```
("a" in mojObjekat); // true
```

Ako se koristi **in operator** za proveru postojanja svojstva u objektu, in će proveriti sva svojstva u objektu – i nabrojiva i nenabrojiva.

Primer nenabrojivih svojstava:

```
let tudjiObjekat = { // sva svojstva su nabrojiva pri kreiranju objekta
    a:3,
    b:4,
    c:5,
};
// da se uverimo da su svojstva nabrojiva
let descriptor = Object.getOwnPropertyDescriptor( tudjiObjekat, "a" );
console.log("Svojstvo a: \n", JSON.stringify(descriptor, null, 2))
descriptor = Object.getOwnPropertyDescriptor( tudjiObjekat, "b" );
console.log("Svojstvo b: \n", JSON.stringify(descriptor, null, 2))
descriptor = Object.getOwnPropertyDescriptor( tudjiObjekat, "c" );
console.log("Svojstvo c: \n", JSON.stringify(descriptor, null, 2))

// Ovde ćemo sva svojstva deklarisati kao nenabrojiva
Object.defineProperty(
        tudjiObjekat,
        "a", { enumerable:false}
);
Object.defineProperty(
        tudjiObjekat,
        "b", { enumerable:false}
);

Object.defineProperty(
        tudjiObjekat,
        "c", { enumerable:false}
);
// da se uverimo da su sada svojstva nenabrojiva
descriptor = Object.getOwnPropertyDescriptor( tudjiObjekat, "a" );
console.log("Svojstvo a: \n", JSON.stringify(descriptor, null, 2))
descriptor = Object.getOwnPropertyDescriptor( tudjiObjekat, "b" );
console.log("Svojstvo b: \n", JSON.stringify(descriptor, null, 2))
descriptor = Object.getOwnPropertyDescriptor( tudjiObjekat, "c" );
console.log("Svojstvo c: \n", JSON.stringify(descriptor, null, 2))

// kreiranje objekta povezanog na tudjiObjekat
var mojObjekat = Object.create( tudjiObjekat );

// nalaženje svojstava u objektu mojObjekat – neće naći ni jedno svojstvo
for (var k in mojObjekat) {
        console.log("našao: ", k);
}

("a" in mojObjekat); // true
("b" in mojObjekat); // true
("c" in mojObjekat); // true
```

Dakle, ovde se konsultuje [[Prototype]] lanac, jedan po jedan link, pri izvršavanju pronalaženja svojstava na različite načine. Traženje se zaustavlja kada se svojstvo nađe, ili kada se stigne do kraja lanca.

4.2.2.1. Ugrađeni objekat `Object.prototype`

Kako je svojstvo [[Prototype]] pokazivač na objekat a objekti imaju metode, tim metodama može da se pristupa pri obilasku lanca. Zato nam je važno da znamo *gde* se tačno "završava" taj [[Prototype]] lanac, odnosno kojim metodama se može pristupiti pri njegovom obilasku.

Vrh svakog *normalnog* [[Prototype]] lanca je ugrađeni objekat **`Object.prototype`**. Taj objekat uključuje mnoštvo različitih pomoćnih programa koji se koriste svuda u JavaScript-u, zbog toga što svi normalni objekti (ugrađeni, ne host-specifična proširenja) u JavaScript-u "potiču od" objekta `Object.prototype`, odnosno imaju na vrhu svog[[Prototype]] lanca objekat `Object.prototype`. Neki od ovih pomoćnih programa su, na primer, `toString()` i `valueOf()`. Pored njih, tu je i `hasOwnProperty()`, kao i još jedna funkcija koju ima `Object.prototype` koje ćemo razmotriti kasnije - funkcija `isPrototypeOf()`.

To znači da svaki objekat može da koristi metode ugrađenog objekta **`Object.prototype`**.

4.2.2.2. Postavljanje i zaklanjanje svojstava

U JavaScript-u je postavljanje svojstava objekta nijansiranije od prostog dodavanja novog svojstva objektu ili menjanja vrednosti postojećeg svojstva. Sada ćemo razmotriti tu situaciju malo detaljnije i potpunije.

Počinjemo sa sledećom linijom koda kojom se svojstvu `foo` objekta `mojObjekat` dodeljuje vrednost `"bar"` :

```
mojObjekat.foo = "bar";
```

Ako objekat `mojObjekat` već ima normalno svojstvo pristupa podacima koje se zove `foo` i koje je prisutno direktno u samom objektu `mojObjekat`, dodela je jednostavna i svodi se na promenu vrednosti postojećeg svojstva u samom objektu.

Ako, pak, `foo` nije direktno prisutno u objektu `mojObjekat`, prolazi se kroz [[Prototype]] lanac, baš kao i kod [[Get]] operacije. Ako se `foo` ne pronađe nigde u lancu, svojstvo `foo` se direktno dodaje objektu `mojObjekat` sa specificiranom vrednošću, kao što se prirodno i očekuje.

Međutim, ako je `foo` već prisutno negde „iznad“ u lancu, dešava se nijansirano (i možda malo iznenađujuće) ponašanje sa dodelom `mojObjekat.foo = "bar"`. Evo kako to, u stvari, izgleda.

Ako se ime svojstva `foo` nađe i u samom objektu `mojObjekat` i negde iznad u [[Prototype]] lancu objekta `mojObjekat`, to se zove ***zaklanjanje***. Svojstvo `foo` koje se pojavljuje direktno u objektu `myObject` *zaklanja* svako drugo `foo` svojstvo koje se pojavljuje negde iznad u lancu, zato što će traženje koje započinje od objekta `mojObjekat` uvek da nađe svojstvo `foo` koje je najniže u lancu, odnosno ono koje je u samom objektu.

Iako izgleda dosta prirodno, kada je u pitanju dodela svojstva/vrednosti kada svojstvo **jeste prisutno** negde na višem nivou [[Prototype]] lanca objekta, to nije baš tako jednostavno.

Sada ćemo da prikažemo tri scenarija za dodelu vrednosti `myObject.foo = "bar"` kada `foo` **nije još direktno prisutno** u objektu `mojObjekat`, ali **jeste prisutno** negde na višem nivou [[Prototype]] lanca objekta `mojObjekat`:

1. Ako se svojstvo normalnog pristupa podacima sa imenom **`foo`** nađe bilo gde na višem nivou lanca [[Prototype]] i **nije označeno kao read-only (`writable:false`)**, tada se novo

svojstvo sa imenom `foo` dodaje direktno objektu `myObject`, rezultujući **zaklonjenim svojstvom**.

2. Ako se svojstvo normalnog pristupa podacima sa imenom **`foo`** nađe na višem nivou `[[Prototype]]` lanca, ali je označeno kao **read-only** (**`writable:false`**), tada se **zabranjuje i kreiranje i postavljanje vrednosti tog postojećeg svojstva u objektu `mojObjekat`**. Ako se kod izvršava u režimu `strict`, biće generisana greška. U protivnom, postavljanje vrednosti svojstva biće ignorisano bez ikakve indikacije. Ni u jednom slučaju **ne dolazi do zaklanjanja**.
3. Ako se **`foo`** svojstvo koje je **seter** nađe na višem nivou lanca `[[Prototype]]` , taj seter će uvek biti pozvan. Nikakvo **`foo`** neće se dodati objektu `mojObjekat` (da zakloni postojeće **`foo`** koje je "iznad" u lancu), a ni `foo` seter neće biti biti redefinisan.

Većina programera pretpostavlja da će dodela svojstva (`[[Put]]`) uvek za rezultat imati zaklanjanje svojstva ako ono već postoji na višem nivou lanca`[[Prototype]]`, ali je, kao što vidite, to tačno samo u prvoj od tri opisane situacije.

Ako se želi zakloniti `foo` u slučajevima #2 i #3, ne može se koristiti dodela putem jednakosti (=) , već se mora izvršiti eksplicitno kreiranje i postavljanje svojstva u objekat `mojObjekat` putem funkcije `Object.defineProperty()` za dodavanje svojstva `foo` objektu `mojObjekat`.

Napomena: Slučaj#2 je možda i najveće iznenađenje od tri navedena. Prisustvo svojstva koje je *read-only* svojstvo sprečava da se implicitno kreira (zakloni) svojstvo sa istim imenom na nižem nivou `[[Prototype]]` lanca. Razlog za ovakvo ograničenje je, pre svega, da bi se održala iluzija klasno nasleđenih svojstava. Ako se razmišlja o svojstvu `foo` na višem nivou lanca kao o svojstvu koje je objekat `mojObjekat` nasledio (kopirano je u njega), ima smisla da se forsira nepromenljiva (*non-writable*) priroda tog `foo` svojstva u objektu `mojObjekat` – u njemu je to svojstvo samo kopija nečega što se ne može menjati. Ako se, pak, uvaži činjenicu da se ***takvo kopiranje stvarno nije desilo***, malo je neprirodno da se sprečava da objekat `mojObjekat` ima svojstvo `foo` samo zbog toga što neki drugi objekat ima u sebi svojstvo `foo` koje se ne može menjati (ono je non-writable). Još je čudnije što ovo ograničenje važi samo za dodelu putem jednakosti (**=**), a ne i za korišćenje `Object.defineProperty()`. Sve u svemu, malo neobična logika pa je valja imati na umu.

Zaklanjanje **metoda** vodi ka ružnom *eksplicitnom pseudo-polimorfizmu* ako treba vršiti delegiranje među ovim svojstvima. Obično je zaklanjanje komplikovanije i nijansiranije od vrednosti rezultata koji se postiže **pa ga izbegavajte ako možete**. Postoji alternativni obrazac dizajna - OLOO stil delegiranja[19] - koji, između ostalog, obeshrabruje korišćenje zaklanjanja i prednost daje čistijim alternativama.

Zaklanjanje može da se "desi" greškom pa valja paziti da se to izbegne. Posmatrajmo sledeći kod:

```
let tudjiObjekat = {
    a:3,
};
let mojObjekat = Object.create(tudjiObjekat);

console.log(tudjiObjekat.a); // 3
console.log(mojObjekat.a); // 3

console.log(tudjiObjekat.hasOwnProperty( "a" )); // true
console.log(mojObjekat.hasOwnProperty( "a" )); // false
```

[19] Stil delegiranja koji delegiranje vrši između objekata, ne između klasa. Termin OLOO (objects-linked-to-other-objects) nije zvaničan termin, već kovanica koju je smislio i koristi Kyle Simpson. Vrlo razumljivo objašnjenje obrasca OLOO možete naći na linku https://karistobias.medium.com/part-1-the-javascript-oloo-pattern-explained-with-pictures-34be175b7908

```
// Evo ga - implicitno zaklanjanje!

mojObjekat.a++;
console.log(tudjiObjekat.a); // 3
console.log(mojObjekat.a); // 4
console.log(mojObjekat.hasOwnProperty( "a" )); // true
```

U slučaju izvršavanja linije `mojObjekat.a++` može da se desi da `mojObjekat.a++` nađe (putem delegacije) i prosto inkrementira svojstvo `tudjiObjekat.a` sam *na licu mesta* kao u primeru.

A može da dođe i do toga da, umesto da ++ operacija bude `mojObjekat.a = mojObjekat.a + 1`, ona u stvari bude `[[Get]]` koji traži svojstvo **a** putem `[[Prototype]]` lanca i dobije tekuću vrednost 2 iz `tudjiObjekat.a`, poveća tu vrednost za jedan, i da zatim `[[Put]]` dodeli vrednost 3 novom zaklonjenom svojstvu **a** objekta `mojObjekat`. Oops!

Budite veoma pažljivi kada radite modifikaciju sa delegiranim svojstvima. Ako želite da inkrementirate `drugiObjekat.a`, jedini ispravan način je da to bude `tudjiObjekat.a++`.

4.2.3. Klase u JavaScript-u

U ovom trenutku, Vi se možda pitate: *Zašto* jedan objekat treba da ima vezu sa drugim objektom? Šta će nam, uopšte, toliki objekti? Šta je tu stvarna dobit? Pitanje je na mestu, ali prvo moramo da razumemo šta `[[Prototype]]` **nije**, da bismo mogli da potpuno razumemo šta on **jeste** i (o)cenimo koja je korist od njega.

U JavaScript-u nema u nativnom implementirane apstraktne obrasce za klase kakvi postoje u klasno-baziranim jezicima. JavaScript **nativno ima samo objekte**.

U stvari, JavaScript je gotovo jedinstven utoliko što sa punim pravom može da nosi atribut "objektni " jer je on jedan od retkih objektnih jezika u kome se objekat može kreirati direktno, bez ikakvog pominjanja klase.

U klasno-baziranom objektnom programiranju klasa, preciznije dijagram klasa, služi da opiše šta pojedinačni objekat koji je instanca neke klase može da radi. Ona opisuje (do određene mere) kako će objekat da izgleda i u kakvoj je vezi sa objektima iste klase i drugih klasa. Dakle, u pitanju je proširiv programski obrazac za kreiranje objekata koji obezbeđuje inicijalna stanja (atributi i vrednosti) i inicijalno ponašanje (metode). Figurativno rečeno, klasa je “nedovršen” objekat, odnosno skica objekta a dijagram klasa je skica (ali mnogo čvršća i manje “nedovršena”) veza među objektima.

Ponovićemo: u jeziku JavaScript, **NEMA NATIVNE KLASE**, **POSTOJI *SAMO* OBJEKAT**. Zbog toga, objekat svoje ponašanje definiše direktno. To ne znači da ne postoji implementacija klasno-baziranih koncepata, samo su ti koncepti napravljeni uz oslonac na nativni mehanizam za povezivanje objekata koji podržava JavaScript.

U nastavku ovog odeljka bavićemo se podrškom za klase koju obezbeđuje JavaScript. Prikazaćemo šta taj mehanizam pruža i na koji način se koristi, te kako se oslanja na pristup zasnovan na prototipu koji je nativan za JavaScript.

4.2.3.1. Podrška za klase u JavaScript-u

Podrška za klase u javaScriptu uvedena je u specifikaciji ECMAScript 2015 (ES6) a najnovija specifikacija ECMAScript 2020 uvela je dodatne fleksibilnosti koja olakšavaju korišćenje klasa.

4.2.3.1.1. Sintaksa klase

Osnovna sintaksa JavaScript klase je sledeća:

```
class ImeKlase {
```

```
  // telo klase
}
```

U ovoj sintaksi, `ImeKlase` je identifikator klase a u telu klase mogu da se nađu sledeće stvari:

```
class ImeKlase {
  // Konstuktor
  constructor() {
    // telo konstruktora
  }

  // Polje instance
  mojePolje = "foo";

  // metoda instance
  mojaMetoda() {
    // telo mojaMetoda
  }

  // Statičko polje
  static mojeStatičkoPolje = "bar";

  // Statička metoda
  static mojaStatickaMetoda() {
    // telo mojaStatickaMetoda
  }

  // Statički blok
  static {
    // inicijalizacioni kod
  }

  // Polje, metoda, statičko polje i statička metoda imaju "private" oblik
(#)
  #myPrivateField = "bar";
}
```

Napomena: Ključna reč `static` definiše statički metod ili polje za klasu ili statički blok inicijalizacije. Statičkim svojstvima se ne može direktno pristupiti iz instance klase, već im se pristupa iz same klase. Statičke metode su često pomoćne funkcije, kao što su funkcije za kreiranje ili kloniranje objekata, dok su statička polja korisna za keš memorije, fiksnu konfiguraciju ili bilo koje druge podatke koji se ne moraju replicirati na instance.

Evo veoma jednostavnog primera koji će nam ilustrovati kako se koriste neki elementi sintakse i otkriti šta je, stvarno, klasa u JavaScript-u.

U našem primeru klasa ima konstruktor koji je funkcija koja prima jedan argument (`ime`) i definiše jednu metodu `kaziZdravo()` koja na konzoli ispisuje vrednost primljenog argumenta:

```
class Student {
  constructor(ime) { this.ime = ime; }
  kaziZdravo() { console.log(this.name); }
}

// Pa da vidimo šta je, u stvari, klasa
console.log ("class Student je tipa: ", typeof Student); // function
```

```
Student; /* Vraća sledeće:

class Student {
  constructor(ime) { this.ime = ime; }
  kaziZdravo() { console.log(this.name); }
} */
```

Dakle, u JavaScript-u **klasa je samo funkcija** koja radi sledeće:

- Kreira funkciju sa zadatim imenom klase (u primeru to je `Student`) koja postaje rezultat deklaracije klase. Kod te funkcije preuzima se od `constructor` metode. Ukoliko se takva metoda ne napiše, podrazumeva se funkcija sa praznim telom.
- Uskladišti metode klase (u primeru to je `kaziZdravo()`) u objekat sa imenom koje odgovara imenu klase.

U prethodnom snipetu smo deklarisali klasu i videli smo da je ona **samo vrsta funkcije** a u sledećem fragmentu koda videćemo kako se klasa koristi da "proizvede" konkretan objekat.

Pošto je klasa vrsta funkcije, jedino što ona može (i treba) da uradi je da se izvrši. Shodno tome, konkretan objekat koji je baziran na datoj klasi kreira se pozivanjem funkcije `Student(ime)`. Pa da pokušamo:

```
class Student {
  constructor(ime) { this.ime = ime; }
  kaziZdravo() { console.log("Zdravo ", this.ime); }
}

mojStudent = Student("Marko"); // poziv kojim se kreira objekat - instanca
mojStudent.kaziZdravo(); // pozivanje instance
```

Ovo ne radi, vraća grešku **Uncaught TypeError: Class constructor Student cannot be invoked without 'new'**

Poruka o grešci nas upućuje da treba **drugačije** da pozovemo funkciju `Student()`, preciznije da treba da je pozovemo **sa ključnom reči new**. Ako to uradimo (navedemo ključnu reč **new** ispred poziva funkcije), dobijamo kod koji radi:

```
class Student {
  constructor(ime) { this.ime = ime; }
  kaziZdravo() { console.log("Zdravo ", this.ime); }
}

mojStudent = new Student("Marko"); // poziv klase sa ključnom reči new
mojStudent.kaziZdravo(); // Zdravo Marko
// da vidmo da li je mojStudent stvarno objekat
console.log("mojStudent je tipa:  ", typeof mojStudent ); /* ispis:
                                                            mojStudent je
                                                            tipa: object
                                                         */
// I da vidimo šta ima u tom objektu
console.log(mojStudent); // izgleda ovako:
/*
   Student {ime: 'Marko'}
            ime: "Marko"
            [[Prototype]]: Object */
// Šta je ovde prototip ?
```

```
Object.getPrototypeOf(mojStudent) === Student.prototype // true
console.log (Student.prototype)
/*
      {constructor: ƒ, kaziZdravo: ƒ}
            constructor: class Student
            kaziZdravo: ƒ kaziZdravo()
            [[Prototype]]: Object */
```

Sada se može napraviti više (koliko god želimo) objekata po šablonu klase `Student` – na primer, objekat za svakog studenta iz grupe pozivanjem klase sa ključnom reči `new`.

4.2.3.2. JavaScript klase "ispod haube"

Ovaj odeljak namenjen je čitaocima koji žele da steknu dublji uvid u način na koji je u JavaScript-u implementirana podrška za klase. Naravno da je korisno da ga pročitate, ali nije neophodno ukoliko samo želite da koristite klase bez dubljeg zalaženja u sam mehanizam jezika kojim je koncept klase ovde implementiran.

Kyle Simpson u svojoj popularnoj seriji "You don't know JavaScript" iznosi i analizira pojavu u JavaScript-u koja za cilj ima, kako on kaže, *besramno hakovanja nečega što izgleda kao "klase"*. Prikazaćemo ovde njegovu analizu u kojoj on ovu pojavu naziva "čudno klasno ponašanje".

To "čudno klasno ponašanje" oslonjeno je na čudnu karakteristiku funkcija u JavaScript-u: sve funkcije podrazumevano dobijaju javno nenabrojivo svojstvo sa imenom `prototype` koje pokazuje na potpuno proizvoljan objekat:

```
function Foo() {
       // ...
}
```

`Foo.prototype; // { constructor: ƒ}`

Taj se objekat često zove "prototip Foo-a", jer mu se pristupa putem reference svojstva nesrećno nazvane `Foo.prototype`. Po Simpsonu, ova termin je predodređen za zbunjivanje i on se šali predlaganjem različitih naziva za taj objekat (na primer, "objekat ranije poznat kao prototip Foo-a" i "objekat proizvoljno označen kao `Foo.prototype`".

Najdirektniji način da se on objasni je da će svaki objekat kreiran pozivanjem `new Foo()` da završi (na neki način proizvoljno) kao `[[Prototype]]`-povezan na taj " `Foo.prototype` " objekat.

Evo ilustracije:

```
function Foo() {
       // ...
}
var a = new Foo();
Object.getPrototypeOf( a ) === Foo.prototype; // true
```

Zašto je to tako? Kada se u JavaScript-u funkcija pozove sa ključnom reči `new` ispred – *kostruktorski poziv* – sledeće stvari se rade automatski:

1. Kreira (konstruiše) se potpuno nov objekat iz ničega.
2. *Novokreirani objekat* dobija javno nenabrojivo svojstvo sa imenom `prototype` .
3. Pokazivač `this` za taj funkcijski poziv postavlja se na novokonstruisani objekat.
4. Ukoliko funkcija ne vraća sopstveni alternativni objekat, funkcijski poziv sa `new` će automatski vratiti novokonstruisani objekat.

Kada se kreira a pozivanjem `new Foo()`, jedna od stvari (korak 2) koja se dešava je da a dobija interni `[[Prototype]]` link na objekat na koji `Foo.prototype` pokazuje.

Šta je, u stvari, posledica ovoga?

U klasno-baziranim jezicima se pravi više **kopija** ("instanci") klase poput izlivanja nečega iz kalupa. To se dešava zato što proces inistanciranja (ili nasleđivanja od) klase znači: "kopiraj plan ponašanja iz te klase u fizički objekat ", i to se ponavlja za svaku novu instancu.

Međutim, u JavaScript-u **nema takvih akcija kopiranja**. Ne kreiraju se višestruke instance klase. Mogu se kreirati višestruki objekti koje `[[Prototype]]` *povezuje* u zajednički objekat. Podrazumeva se da se ne dešava se kopiranje i zbog toga te instance ne završavaju kao potpuno odvojeni objekti, već su oni baš ***povezani***.

`new Foo()` rezultuje novim objektom (ovde je on nazvan **a**), i **taj novi objekat a** se interno povezuje pomoću `[[Prototype]]` na `Foo.prototype` objekat.

Dakle, završava se sa dva objekta povezana jedan sa drugim. To je *to*. Ovde objekat nije instancirana klasa (napravljen objekat u koji je klasa kopirana) pa sigurno nije kopirano nikakvo ponašanje iz "klase" u konkretan objekat. Samo je ostvareno da dva objekta budu povezani jedan sa drugim.

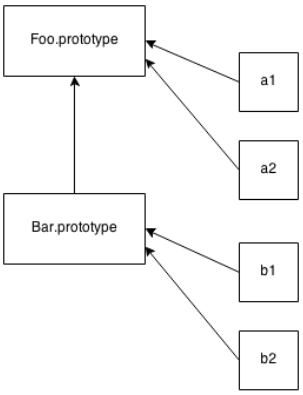
U stvari, ono što izmiče većini JavaScript programera je da poziv funkcije `new Foo()` nije imao skoro ništa *direktno* sa procesom kreiranja veze. **To je bila vrsta slučajnog bočnog efekta**. `new Foo()` je indirektan, zaobilazni način da se dobije ono što se želi: **novi objekat povezan sa drugim objektom**.

A može li to što se želi dobiti na *direktan* način? **Da!** To omogućuje ugrađena metoda **`Object.create()`** o čemu ćemo malo kasnije.

4.2.3.2.1.1. Šta je u nazivu "prototipsko nasleđivanje"?

U JavaScript-u se **ne prave *kopije* iz jednog objekta** ("klasa") **u drugi** ("instanca"). **Prave se *linkovi*** između objekata (slika 4.1). Za mehanizam `[[Prototype]]` , strelice su usmerene sa desna na levo i od dole na gore.

Ovaj mehanizam se često naziva "prototipsko nasleđivanje" (kod ćemo da istražimo u detalje ubrzo), za koje se obično kaže da je verzija "klasičnog nasleđivanja" za dinamičke jezike. To je pokušaj da se iskoristi uobičajeno razumevanje značenja "nasleđivanja" u klasno-baziranom svetu, ali uz unapređenje shvaćene semantike tako da odgovara dinamičkom skriptingu.



Slika 4.1 Nasleđivanje u JavaScript-u nije kopiranje, nego povezivanje

Reč "nasleđivanje" ima vrlo jako značenje sa mnogo mentalnog prethođenja i mogućih presedana. Dodavanje samo atributa "prototipsko" na početku, da bi se odvojilo *u stvari gotovo suprotno*

ponašanje u JavaScript-u od onoga što mi standardno razumemo kao nasleđivanje, rezultovalo je konfuzijom dugom skoro dve decenije.

Simpson to duhovito ilustruje na sledeći način: "Želim da kažem da je postavljanje reči ***prototipsko*** ispred reči ***nasleđivanje*** da bi se označilo u velikoj meri različito značenje kao držanje pomorandže u jednoj ruci i jabuke u drugoj, uz insistiranje da se jabuka zove ***crvena pomorandža***. Ni jedna zbunjujuća oznaka koju stavljam ispred nje, ne može da izmeni *činjenicu* da je jedno voće jabuka a drugo pomorandža. Bolji je pristup da jabuku zovemo jabukom - da se koristi najtačnija i direktna terminologija. To olakšava razumevanje i njihovih sličnosti i **puno razlika**, jer svi imamo isto jasno razumevanja značenja reči ***jabuka***".

Na kraju, zaključuje: "Zbog zabune i mešanja termina, verujem da je sama oznaka ***prototipsko nasleđivanje*** (i pokušaj da se primeni klasno-bazirana terminologija poput ***klasa***, k***onstruktor***, ***instanca***, ***polimorfizam***, itd.) donela **više štete nego dobra** u objašnjavanju *stvarnog* rada objektnog mehanizma JavaScript-a. "

Da sažmemo. Termin "nasleđivanje" u klasno-baziranom OOP-u (a ono je danas dominantno) podrazumeva operaciju *kopiranja* , a JavaScript ne kopira svojstva objekta (nativno, po pretpostavci). Umesto toga, JavaScript kreira link između dva objekta, gde jedan objekat može drugom objektu suštinski da *delegira* svojstvo/pristup funkciji, da ga ZADUŽI za ostvarivanje nekog ponašanja. Zbog toga je "delegiranje" mnogo bolji termin za označavanje mehanizma povezivanja objekata JavaScript-a.

Drugi termin koji možete ponekad sresti u JavaScript-u je "diferencijalno nasleđivanje". Ovde je ideja da ponašanje objekta opisujemo pomoću onoga što je *različito* od opštijeg deskriptora. Na primer, možete da kažete da je auto vrsta vozila, ali takvog da ima tačno 4 točka, umesto da ponovo navodite sve ono od čega je sačinjeno generalno vozilo (motor, itd.) i da tome dodajete specifičnosti.

Ali, baš kao i "prototipsko nasleđivanje", i "diferencijalno nasleđivanje" pretvara se da je mentalni model važniji od onoga što se fizički dešava u jeziku. Ono previđa činjenicu da objekat B u stvari nije diferencijalno konstruisan, već je izgrađen sa definisanim specifičnim karakteristikama, zajedno sa "rupama" (nedostaci delova ili celih definicija) u kojima ništa nije definisano. Te "rupe" valja "popuniti" što delegiranje *može* da preuzme i da ih, u letu, "popuni" delegiranim ponašanjem.

4.2.3.2.1.2. "Konstruktori"

Vratimo se na kod od ranije:

```
function Foo() {
        // ...
}

var a = new Foo();
```

Šta nas, tačno, navodi da o Foo mislimo kao o "klasi"?

Kao prvo, vidimo korišćenje ključne reči `new` , iste one koja se u klasno-orijentisanim jezicima koristi pri konstruisanju klasnih instanci. Kao drugo, ispostavlja se da mi, u stvari, izvršavamo *konstruktorsku metodu* klase jer je `Foo()` u stvari metoda koja se poziva baš kao što se poziva i pravi klasni konstruktor kada se klasa instancira.

Da bi se nastavila konfuzija semantike "konstruktora", proizvoljno označen objekat `Foo.prototype` ima još jedan trik u rukavu. Posmatrajmo sledeći kod:

```
function Foo() {
  // ...
}
```

```
Foo.prototype.constructor === Foo; // true

var a = new Foo();
a.constructor === Foo; // true
```

Objekat `Foo.prototype` podrazumevano (u vreme deklaracije u liniji koda 1 ovoga snipeta!) dobija javno, nenabrojivo svojstvo zvano `constructor`, i to svojstvo je pokazivač (natrag) na funkciju (`Foo` u ovom slučaju) sa kojim je objekat asociran. Štaviše, ***izgleda*** kao da objekat `a` kreiran pozivom "konstruktora" `new Foo()` i sam ima svojstvo zvano `constructor` koje slično pokazuje na "funkciju koja ga je kreirala". A to, u stvari, nije tačno. Objekat `a` NEMA `constructor` svojstvo u sebi i, iako se `a.constructor` u stvari razrešava na `Foo` funkciji, "konstruktor" **u stvari ne znači "konstruisan od strane"**, kao što se čini. Ubrzo ćemo da objasnimo ovu neobičnost.

I još jedna "pikanterija": po konvenciji, u JavaScript svetu, "klase" imaju imena koja počinju velikim slovom pa bi činjenica da je u pitanju `Foo` a ne `foo` ovde trebala da predstavlja jasan znak da je namera da to bude "klasa". Ova konvencija je toliko jaka da mnogi JavaScript linteri javljaju grešku ukoliko se pozove `new` sa metodom čije ime počinje malim slovom, ili ako se ne pozove `new` nad funkcijom koja počinje velikim slovom. Svaka diskusija o smislu ovoga postaje bespredmetna ako se zna da velika slova JavaScript endžinu ***ne znače ama baš ništa***.

4.2.3.2.1.3. Konstruktor ili poziv?

U gornjem snipetu navodi se na mišljenje da je `Foo` "konstruktor", zato što ga pozivamo sa `new` i opažamo da on "konstruiše" objekat.

U stvarnosti, `Foo` nije ništa više "konstruktor" od bilo koje druge funkcije u nekom programu. Funkcije same po sebi **nisu konstruktori**. U stvari, **stavljanje ključne reči `new` ispred normalnog poziva funkcije čini taj poziv "konstruktorskim pozivom"**. Može se reći figurativno da `new` na neki način "kidnapuje" bilo koju normalnu funkciju i **poziva je na način koji dovodi do konstruisanja objekta, uz sve ono što je ta funkcija inače radila**.

Na primer:

```
function NistaNarocito() {
       console.log( "Baš me briga!" );
}
var b = NistaNarocito (); // Samo je ispisano je "Baš me briga!"
console.log ("b: ", b); // undefined
console.log ("tip od b: ", typeof b) // undefined

var a = new NistaNarocito (); // Ispisano je "Baš me briga!"
console.log ("a: ", a); // NistaNarocito {}
console.log ("tip od a: ", typeof a) // object
```

`NistaNarocito ()` je samo normalna funkcija i kada se **"normalno pozove"** ona samo uradi svoj posao - ispiše na konzoli `Baš me briga!`. Međutim, kada se pozove sa `new`, ona, pored toga što uradi svoj posao, **kao bočni efekat *konstruiše* objekat** koji je dodeljen promenljivoj **`a`**. Taj **poziv** sa `new` nazvan je *konstruktorski poziv*, ali `NistaNarocito ()` nije, ni u sebi, ni sama po sebi, *konstruktor*.

Drugim rečima, u JavaScript-u je najprikladnije reći da je "konstruktor" **bilo koja funkcija pozvana sa ključnom reči `new`** ispred.

Funkcije same po sebi nisu konstruktori, ali su f**unkcijski pozivi "konstruktorski pozivi" ako i samo ako se u pozivu koristi ključna reč `new`**.

4.2.3.2.1.4. Mehanika

Da li su *to* jedini uobičajeni okidači za diskusiju o zlosrećnoj "klasi" u JavaScript-u? **Ne baš.** JavaScript programeri nastoje da simuliraju što je moguće više klasne orijentacije:

```
function Foo(ime) {
        this.ime = ime;
}

Foo.prototype.mojeIme = function() {
        return this.ime;
};

let a = new Foo( "JaSamObjekat_a" );
let b = new Foo( "JaSamObjekat_b" );

/* a i b su dva različita objekta i izgleda kao da svaki od njih ima svoje
svojstvo mojeIme sa različitim vrednostima svojstva */
console.log(a.mojeIme ()); // "JaSamObjekat_a"
console.log(b.mojeIme ()); // "JaSamObjekat_b"
```

Ovaj snipet prikazuje još dva trika "klasne orijentacije" koji su u igri:

1. Linija `this.ime = ime`; pridodaje svojstvo `ime` svakom objektu (**a** i **b**, respektivno), slično načinu na koji instance klase enkapsuliraju vrednosti podataka.
2. `Foo.prototype.mojeIme = function() {return this.ime;};` kojim se dodaje svojstvo (funkcija) `mojeIme` objektu `Foo.prototype` . Sada a`.mojeIme()` radi, a šta radi ?

U gornjem snipetu se baš jako navodi na mišljenje da se, kada se objekti **a** i **b** kreiraju, svojstva/funkcije objekta `Foo.prototype` *kopiraju* u svaki od objekata **a** i **b**. **Međutim, to nije ono što se događa.**

Na početku ovoga Poglavlja je objašnjen `[[Prototype]]` link i kako on obezbeđuje korake traženja ako referenca svojstva nije direktno nađena u objektu kao deo podrazumevanog `[[Get]]` algoritma.

Prema načinu kako su kreirani, **a** i **b** završavaju sa internim `[[Prototype]]` povezivanjem na objekat `Foo.prototype`. Kada se `mojeIme` ne pronađe u **a** i u **b** respektivno, ono se pronalazi (putem delegiranja) u `Foo.prototype`. Evo i koda koji to potvrđuje:

```
function Foo(ime) {
        this.ime = ime;
}

let a = new Foo( "JaSamObjekat_a" );
let b = new Foo( "JaSamObjekat_b" );

/* a i b su dva različita objekta i izgleda kao da svaki od njih ima svoje
svojstvo mojeIme sa različitim vrednostima svojstva */
console.log(a.mojeIme ()); // Vraća: Uncaught TypeError: a.mojeIme is not a function
console.log(b.mojeIme ()); // Vraća: Uncaught TypeError: b.mojeIme is not a function
```

Razlog za ovakvo ponašanje je što svojstvu `mojeIme` u nije dodeljena vrednost u objektu `Foo.prototype`:

```
console.log (Foo.prototype.mojeIme) // undefined
```

a objekti a i b ga takođe nemaju, nego ga traže u objektu `Foo.prototype`.

4.2.3.2.1.5. "Konstruktor" Redux

Podsetimo se ranije diskusije o svojstvu `constructor` i o tome kako **nije ispravan** zaključak da evaluacija izraza `a.constructor === Foo` na `true` znači da **a** ima u sebi `constructor` svojstvo koje pokazuje na `Foo`.

To je samo nesrećna zabuna. U stvarnosti, `constructor` referenca se takođe *delegira* na razrešavanje `Foo.prototype`-u kome se **desilo** da podrazumevano ima `constructor` koji pokazuje na `Foo`.

Izgleda kao da bi bilo veoma logično da objekat **a** "konstruisan od strane" `Foo` ima pristup svojstvu `constructor` koje pokazuje na `Foo`. Ali to nije ništa više do lažni osećaj sigurnosti. To je srećan događaj, gotovo tangencijalan, da se **desilo** da `a.constructor` pokazuje na `Foo` putem te podrazumevane `[[Prototype]]` delegacije. Ima nekoliko načina da se ta zlosrećna pretpostavka o značenju `constructor`–a "konstruisan od strane" potuljeno okrene i da Vas ujede.

Kao prvo, svojstvo `constructor` objekta `Foo.prototype` je tu samo po pretpostavci o objektu kreiranom kada je funkcija `Foo` deklarisana. Ako kreirate novi objekat i zamenite pretpostavljenu `prototype` referencu objekta, novi objekat neće po pretpostavci magično dobiti na sebi `constructor`.

Posmatrajmo:

```
function Foo() { /* .. */ }
Foo.prototype= { /* .. */ }; // kreiraj novi prototipski objekat
var a1 = new Foo();
a1.constructor === Foo; // false!
a1.constructor === Object; // true!
```

`Object()` nije "konstruisao" `a1` zar ne? Linja `var a1 = new Foo();` kaže izgleda da ga je `Foo()` "konstruisao". Mnogi programeri misle da `Foo()` vrši konstruisanje, ali mesto na kome se sve raspada je kada mislite da "konstruktor" znači "konstruisan od strane", jer bi, na osnovu takvog rezonovanja, `a1.constructor` trebalo da bude `Foo`, ali to nije tako!

Šta se dešava? `a1` nema svojstvo `constructor` pa delegira uz `[[Prototype]]` lanac objektu `Foo.prototype`. Međutim, taj objekat takođe nema `constructor` (kao što bi ga imao podrazumevani `Foo.prototype` objekat!), pa nastavlja sa delegiranjem, ovoga puta objektu `Object.prototype`, vrhu lanca delegiranja. *Taj* objekat uistinu ima na sebi `constructor` koji pokazuje na ugrađenu `Object()` funkciju.

4.2.3.2.1.6. Zabuna, razbijena

Naravno, može da se doda `constructor` objektu `Foo.prototype`, ali to zahteva manuelni rad, posebno ako se hoće da se postigne slaganje sa nativnim ponašanjem i da svojstvo bude nenabrojivo.

Na primer:

```
function Foo() { /* .. */ }

Foo.prototype = { /* .. */ }; // kreiraj novi prototipski objekat

/* Potrebno je prikladno "popraviti" nedostajuće `constructor`
   svojstvo na novom objektu koji služi kao `Foo.prototype`. */

Object.defineProperty( Foo.prototype, "constructor" , {
        enumerable:false,
        writable:true,
        configurable:true,
        value: Foo    // uputi `constructor` na `Foo`
} );
```

To je puno manuelnog rada da se popravi `constructor`. Štaviše, ono što stvarno radimo je da održavamo u životu zabludu da "konstruktor" znači "konstruisan od strane". To je *skupa* iluzija.

Činjenica je da `constructor` u objektu podrazumevano proizvoljno pokazuje na funkciju koja, za uzvrat, ima referencu natrag na objekat – referencu koju naziva `.prototype`. Reči "constructor" i

"prototype" imaju samo labavo podrazumevano značenje koje kasnije može i ne mora da bude istinito. Najbolje pravilo za ponašanje u ovakvim situacijama je da čovek stalno ima na umu da "konstruktor ne znači konstruisan od".

`constructor` nije magično imutabilno svojstvo. Ono *jeste* nenabrojivo (vidi snipet iznad), ali mu `writable` ima vrednost `true` što znači da njegovo svojstvo `value` može da se menja; štaviše, moguće je dodati ili prepisati (namerno ili slučajno) svojstvo sa imenom `constructor` svakom objektu u svakom `[[Prototype]]` lancu, sa svakom vrednošću koja se smatra pogodnom.

Uzimajući u obzir način prolaska `[[Get]]` algoritma kroz `[[Prototype]]` lanac, jasno je da referenca `constructor` svojstva, bilo gde da je nađena, može da se razreši vrlo različito od onoga što bi se očekivalo.

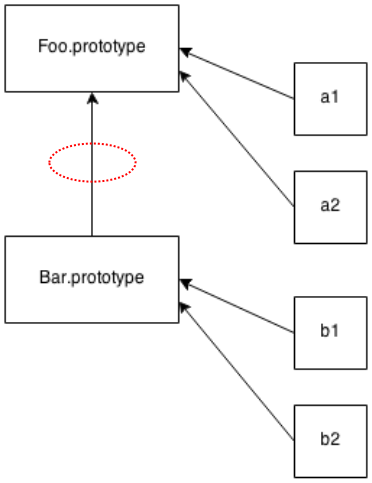
A koliko je stvarno proizvoljno njeno značenje? Za neku proizvoljnu referencu svojstva objekta kao što je `a1.constructor` ne može se *imati poverenja* da je pretpostavljena referenca na funkciju. Štaviše, kao što ćemo ubrzo da vidimo, samo jednostavnim izostavljanjem, `a1.constructor` može da završi time što će pokazivati na nešto sasvim iznenađujuće i neosetljivo.

Sve u svemu, `constructor` je ekstremno nepouzdana i nesigurna referenca za oslanjanje u kodu. **Generalno, takve reference treba izbegavati kad god je moguće.**

4.2.3.3. Prototipsko "nasleđivanje"

Videli smo neke aproksimacije mehanike "klasa" koje se tipično hakuju u JavaScript programe. Naravno, JavaScript "klase" bile bi prilično "rupičaste" kada ne bismo imali i aproksimaciju za "nasleđivanje".

U stvari, mi smo već videli na delu mehanizam koji se uobičajeno zove "prototipsko nasleđivanje" kada smo imali situaciju u kojoj je svojstvo a moglo da "nasledi od" `Foo.prototype`, i na taj način dobije pristup funkciji `mojeIme()`. Međutim, mi tradicionalno razmišljamo o "nasleđivanju" kao o relaciji između dve "klase", a ne kao o relaciji između "klase" i "instance". Tako je to u klasno-baziranom OOP. Podsetimo se slike koju smo videli ranije (slika 4.1):



Ova slika prikazuje delegiranje ne samo od objekta ("instance") `a1` objektu `Foo.prototype`, već i delegiranje od `Bar.prototype` ka `Foo.prototype`, što na neki način *podseća* na koncept klasnog nasleđivanja roditelj-dete. *Podseća*, izuzev strelica koji pokazuju **da su to linkovi delegiranja a ne operacije kopiranja**.

A evo i tipičnog "prototype style" koda koji kreira takve linkove:

```
function Foo(name) {
        this.name = name;
```

```
}

Foo.prototype.myName = function() {
        return this.name;
};

function Bar(name,label) {
        Foo.call( this, name );
        this.label = label;
}

/* ovde se pravi novi `Bar.prototype`
   povezan na `Foo.prototype` */
Bar.prototype = Object.create( Foo.prototype );

/* Pazite! Sada je `Bar.prototype.constructor` otišao,
   i možda ga treba ručno "popraviti" ako ste
   navikli da se oslanjate na takva svojstva! */

Bar.prototype.myLabel=function() {
        return this.label;
};

var a = new Bar( "a", "obj a" );

a.myName(); // "a"
a.myLabel(); // "obj a"
```

Napomena: Da biste razumeli zašto `this` pokazuje na `a` u gornjem kodu, pogledajte deo koji se bavi `this` vezivanjem.

Važan deo ovde je linija koda `Bar.prototype = Object.create(Foo.prototype )`. Ovde `Object.create()` *kreira* "novi" objekat „iz ničega“, i povezuje interni `[[Prototype]]` tog novog objekta sa objektom koji se specificira (`Foo.prototype` u ovom slučaju).

Drugim rečima, ta linija koda kaže: "napravi *novi* 'Bar dot prototype' objekat koji je povezan na 'Foo dot prototype'."

Kada je deklarisana `function Bar() { .. }`, `Bar`, kao i svaka druga funkcija, ima `.prototype` link na svoj podrazumevani objekat. Ali *taj* objekat nije povezan na `Foo.prototype` kao što mi želimo. Stoga kreiramo *novi* objekat koji *jeste* povezan kako mi želimo, efektivno odbacujući originalni nekorektno povezani objekat.

Napomena: Uobičajena zabluda ovde je da će bilo koji od sledećih pristupa takođe da radi, ali oni ne rade kao što biste Vi očekivali:

```
// ne radi kao što Vi želite!
Bar.prototype = Foo.prototype;
```

Naredba `Bar.prototype = Foo.prototype` ne kreira novi objekat za `Bar.prototype` na koji će se povezati. Ova naredba samo čini da `Bar.prototype` bude druga referenca na `Foo.prototype`, koja efektivno povezuje `Bar` direktno na **isti objekat na koji povezuje** `Foo` : `Foo.prototype`. To znači da se, kada počnete da vršite dodeljivanje, poput `Bar.prototype.myLabel = ...`, **ne modifikuje odvojeni objekat** već se modifikuje sam *deljeni* `Foo.prototype` objekat, što bi uticalo na svaki objekat povezan na `Foo.prototype`. To skoro sigurno nije ono što želite. Ako to *jeste* ono

što želite, tada Vam verovatno uopšte ne treba `Bar`, i trebali biste da koristite samo `Foo` i pojednostavite Vaš kod.

```
// radi kao što biste želeli, ali sa
// bočnim efektima koje Vi, verovatno, ne želite :(
Bar.prototype = new Foo();
```

Naredba `Bar.prototype = new Foo()` **kreira u stvari** novi objekat koji je propisno povezan na `Foo.prototype` kao što bismo želeli. Ali, ona koristi `Foo(..)` "konstruktorski poziv" da bi to uradila. Ako ta funkcija ima bilo kakve bočne efekte (kao što su logovanje, promena stanja, registrovanje na drugim objketima, **dodavanje svojstava podataka na `this`**, itd.), ti bočni efekti se dešavaju u vreme tog povezivanja (i verovatno nad pogrešnim objektima!) umesto da se to desi samo kada se kreiraju konačni `Bar()` "potomci" što bismo mi, verovatno, očekivali.

Dakle, ostalo nam je da koristimo `Object.create(..)` za pravljenje novog objekta koji je ispravno povezan i nema bočnih efekata koji potiču od pozivanja `Foo(..)`. Mali nedostatak je što moramo da kreiramo novi objekat, odbacujući stari, umesto da modifikujemo postojeći pretpostavljeni objekat koji već imamo.

Bilo bi *lepo* da postoji standardizovan i pouzdan način za modifikovanje povezivanja postojećeg objekta. Pre specifikacije ES6 postoji nestandardan način (nije podržan u svim brauzerima) preko `.__proto__` svojstva, koje se može postaviti. ES6 dodaje `Object.setPrototypeOf(..)` helper, koji trik realizuje na standardan i predvidiv način.

Evo kako izgledaju pre-ES6 i ES6-standardizovana tehnika za povezivanje `Bar.prototype` sa `Foo.prototype`:

```
// pre-ES6
// odbacuje pretpostavljeni postojeći `Bar.prototype`
Bar.prototype = Object.create(Foo.prototype);

// ES6+
// modifikuje postojeći `Bar.prototype`
Object.setPrototypeOf(Bar.prototype, Foo.prototype);
```

Ukoliko se zanemari lagani pad preformanse (odbacivanje objekta koji se kasnije uklanja) pristup `Object.create(..)` je malo kraći i, možda, čitljiviji nego ES6+ pristup. Međutim, to je verovatno sintaktičko „ispiranje“ u oba slučaja.

4.2.3.4. Inspekcija "klasnih" veza

Šta ako imate objekat poput `a` i želite da vidite kom objektu (ako upšte to radi) on delegira? Inspekcija instance (samo objekat u JavaScript-u) kojom se utvrđuje njeno poreklo nasleđivanja (povezivanje delegiranja u JavaScript-u) se zove *introspekcija* (ili *refleksija*) u tradicionalnim klasno-orijentisanim okruženjima.

Posmatrajmo sledeći kod:

```
function Foo() {
        // ...
}

Foo.prototype.blah = ...;

var a = new Foo();
```

Kako izvršiti introspekciju objekta `a` da bismo našli njegovo "nasleđivanje" (povezivanje delegiranja)? Prvi pristup usvaja "klasnu" konfuziju:

```
a instanceof Foo; // true
```

Operator instanceof uzima čist objekat kao svoj levi operand i **funkciju** kao svoj desni operand. Pitanje na koje odgovara instanceof je: **Da li se u celokupnom [[Prototype]] lancu objekta a, IKADA pojavljuje objekat na koji proizvoljno upućuje Foo.prototype?**

Na žalost, to znači da upit o "nasleđivanju" nekog objekta (a) može da se postavi samo ako imate neku **funkciju** (Foo, sa pridruženom prototype referencom) za testiranje. Ako imate dva proizvoljna objekta, recimo a i b, i želite da utvrdite da li su *objekti* međusobno povezani jedan sa drugim putem [[Prototype]] lanca, sama instanceof ne pomaže.

Napomena: Ako koristite ugrađeni pomoćni program bind(..) da napravite čvrsto vezanu (hard-bound) funkciju, kreirana funkcija neće imati svojstvo prototype. Korišćenje instanceof sa takvom funkcijom transparentno supstituiše prototype *ciljne funkcije* iz koje je kreirana hard-bound funkcija.

Prilično je neuobičajeno da se hard-bound funkcije koriste kao "konstruktorski pozivi", ali ako to radite oni će da se ponašaju kao da je pozvana originalna *ciljna funkcija* što znači da se korišćenje instanceof sa hard-bound funkcijom ponaša u skladu sa originalnom funkcijom.

Ovaj snipet ilustruje koliko može da bude smešan pokušaj da se rezonuje o relacijama između **dva objekta** koristeći "klasnu" semantiku i instanceof:

```
/* helper da se vidi da li je `o1`
    povezano sa `o2`(`o1` delegira `o2`-u) */
function isRelatedTo(o1, o2) {
       function F(){}
          F.prototype = o2;
          return o1 instanceof F;
}

var a = {};
var b = Object.create( a );

isRelatedTo( b, a ); // true
```

Unutar isRelatedTo(), mi pozajmljujemo funkciju F(), reasajniramo njenu vrednost prototype tako da proizvoljno pokazuje na neki objekat o2, zatim pitamo da li je o1 "instanca od" F. Očigledno je da o1 nije *u stvari* nasleđena, niti vodi poreklo pa čak nije ni konstruisana iz F, te bi trebalo da bude jasno zašto je ovakav primer luckast i zbunjujući. Problem se svodi na nespretnost prisilne primene klasne semantike na JavaScript, u ovom slučaju kao otkrivanje indirektne semantike funkcije instanceof.

Drugi, i mnogo čistiji pristup [[Prototype]] refleksiji je:

```
Foo.prototype.isPrototypeOf( a ); // true
```

Zapazite da u ovom slučaju mi uopšte ne marimo za (ili nam čak nije ni potrebno) Foo, nama samo treba **objekat** (u našem slučaju proizvoljno označen kao Foo.prototype) da bismo ga uporedili sa drugim **objektom**. Pitanje na koje odgovor daje isPrototypeOf() je: **u celom [[Prototype]] lancu od a, da li se ikada pojavljuje Foo.prototype?**

Isto pitanje i potpuno isti odgovor. Ali u ovom drugom pristupu ne treba nam indirekcija referenciranja **funkcije** (Foo) čije prototype svojstvo će automatski biti konsultovano. Trebaju nam *samo dva* **objekta** da izvršimo inspekciju njihove međusobne veze. Na primer:

```
/* Jednostavno: da li se `b` pojavljuje bilo gde u
```

```
   [[Prototype]] lancu `c`-a? */
b.isPrototypeOf( c );
```

Zapazite da ovaj pristup uopšte ne traži funkciju ("klasa"). On samo koristi reference objekta direktno na `b` i `c`, i ispituje njihovu vezu. Drugim rečima, naš `isRelatedTo()` pomoćni program je ugrađen u jezik i zove se `isPrototypeOf()`.

Može se direktno pretraživati i `[[Prototype]]` objekta. Od ES5, standardan način za to je:

```
Object.getPrototypeOf( a );
```

I videćete da je referenca objekta baš ono što smo očekivali:

```
Object.getPrototypeOf( a ) === Foo.prototype; // true
```

Većina brauzera (ne svi!) ima i dugo podržan nestandardan alternativni način za pristup internom `[[Prototype]]`:

```
a.__proto__ === Foo.prototype; // true
```

Neobično svojstvo `__proto__` (nestandardizovano do ES6!) "magično" pretražuje interni `[[Prototype]]` objekta kao reference, što je vrlo korisno ako želite da direktno pregledate (ili čak prolazite kroz: `__proto__.__proto__...`) taj lanac.

Baš kao što smo ranije videli sa svojstvom `constructor`, svojstvo `__proto__` u stvari ne postoji u objektu koji pregledate (`a` u našem primeru). U stvari, ono postoji (nenabrojivo) u ugrađenom `Object.prototype`, zajedno sa drugim uobičajenim metodama (`.toString()`, `.isPrototypeOf(..)`, itd.).

Štaviše, `__proto__` izgleda kao svojstvo, ali je bolje o njemu razmišljati kao da je u pitanju geter/seter.

Ugrubo, možemo da zamislimo da je `__proto__` implementiran na sledeći način:

```
Object.defineProperty( Object.prototype, "__proto__", {
        get:function() {
                return Object.getPrototypeOf( this );
        },
        set:function(o) {
                // setPrototypeOf() kao u ES6
                Object.setPrototypeOf( this, o );
                return o;
        }
} );
```

Dakle, kada pristupimo (tražimo vrednost od) `a.__proto__`, to je kao pozivanje `a.__proto__()` (pozivanje getter funkcije). *Taj* funkcijski poziv ima `a` kao svoj `this` iako geterska gunkcija postoji u `Object.prototype` objektu (vidi deo za `this` pravila povezivanja), pa je to isto kao da kažete `Object.getPrototypeOf( a )`.

Svojstvo `__proto__` je i svojstvo koje se može postaviti, baš kao korišćenje ES6 `Object.setPrototypeOf()` prikazano ranije. U svakom slučaju, generalno **ne biste trebali da menjate `[[Prototype]]` postojećeg objekta**.

Postoje vrlo složene tehnike koje se koriste duboko u nekim okruženjima koja dozvoljavaju trikove kao što je "subklasiranje" tipa `Array`, ali se to obično izbegava u generalnoj programerskoj praksi jer vodi ka *mnogo* težem razumevanju/održavanju koda.

Napomena: Od specifikacije ES6, ključna reč `class` omogućuje nešto što aproksimira "subklasiranje" ugrađenih tipova kao što je `Array`.

Jedini drugi uski izuzetak (kao što je ranije pomenuto) bio bi postavljanje `[[Prototype]]` za pretpostavljeni funkcijski `prototype` objekat tako da referencira neki drugi objekat (osim `Object.prototype`). Time bi se izbegla potpuna zamena podrazumevanog objekta sa novim povezanim objektom. Inače, **najbolje je tretirati objekt `[[Prototype]]` povezivanje kao read-only karakteristiku** zarad lakšeg kasnijeg čitanja koda.

Napomena: JavaScript zajednica je neslužbeno iskovala termin "dunder" za dvostruku donju crtu, posebno vodeću u svojstvima kao `__proto__`. Dakle, "trendseteri" u JavaScript-u generalno izgovaraju `__proto__` kao "dunder proto".

4.2.3.5. Objektni linkovi

Kao što smo videli, `[[Prototype]]` mehanizam je interni link koji postoji u jednom objektu koji referencira neki drugi objekat.

To povezivanje se (primarno) manifestuje kada se od prvog objekta zahteva neko svojstvo/metoda, a u tom objektu takvo svojstvo/metoda ne postoji. U tom slučaju `[[Prototype]]` povezivanje kaže *endžinu* da to svojstvo/metodu traži u povezanom objektu. Ako ni taj povezani objekat ne može da zadovolji pretragu, sledi se njegov `[[Prototype]]`, itd. Ta serija linkova između objekata formira ono što se zove "prototipski lanac".

4.2.3.5.1. `Create()`ing Links

Ojasnili smo zašto mehanizam JavaScript-a zvani `[[Prototype]]` **nije** kao *klase* i videli smo kako on, umesto implementacije klasa (kopiranja), kreira **linkove** između odgovarajućih objekata.

Šta je poenta `[[Prototype]]` mehanizma? Zašto je toliko rasprostranjeno među JavaScript programerima da ulažu enorman napor (emulacijom klasa) u svoj kod da bi nekako "objedinili" ta povezivanja?

Setite se da smo dosta ranije u ovom poglavlju kazali da će `Object.create()` biti heroj? Došlo je vreme da vidimo i kako.

```
var foo = {
        something:function() {
                console.log( "Kaži mi, konačno, i nešto dobro ..." );
        }
};

var bar = Object.create( foo );
bar.something(); // Ispis: Kaži mi, konačno, i nešto dobro ...
```

`Object.create()` kreira novi objekat (`bar`) povezan na objekat koji smo specificirali (`foo`), što nam daje svu moć (delegaciju) mehanizma `[[Prototype]]`, ali bez nepotrebnih komplikacija funkcija `new` koje deluju kao klase i konstruktorski pozivi, brkajući `.prototype` i `.constructor` reference, i ostale te dodatne stvari.

Napomena: `Object.create(null)` kreira objekat koji ima prazno (`null`) `[[Prototype]]` povezivanje i zbog toga objekat ne može nigde da delegira. Kako takav objekat nema prototipski lanac, operator `instanceof` (objašnjen ranije) nema ništa da proverava i uvek će da vrati `false`. Ti specijalni, `[[Prototype]]`-prazni objekti se često zovu "rečnici". Ovo ime im potiče od toga što se tipično koriste samo za skladištenje podataka u svojstvima, pre svega zato što nemaju mogućih iznenađujućih efekata koji potiču od delegiranih svojstava/funkcija iz `[[Prototype]]` lanca, i kao takvi predstavljaju čisto skladište podataka.

Nama *ne trebaju klase* za kreiranje smislenih relacija između dva objekta. Jedina stvar o kojoj treba da vodimo računa su objekti povezani za delegiranje, a `Object.create()` daje to povezivanje bez opterećujućeg, nečitkog koda za rad sa klasama.

4.2.3.5.1.1. Polifiling[20] za `Object.create()`

`Object.create()` je dodat u ES5. Može da Vam se pojavi potreba da podržite pre-ES5 okruženja (kao stariji IE), pa da pogledamo jednostavan **parcijalni polifiling za** `Object.create()` koji čak i u tim starijim JavaScript okruženjima daje mogućnost koja nam je potrebna:

```
if (!Object.create) {
        Object.create = function(o) {
                function F(){}
                F.prototype= o;
                return new F();
        };
}
```

Ovaj polifil radi koristeći "throw-away" funkciju `F` i prepisujući njeno `.prototype` svojstvo tako da ono pokazuje na objekat na koji želimo da povezujemo. Zatim koristimo `new F()` konstrukciju da bismo napravili objekat koji će biti povezan onako kako smo specificirali.

Ovakvo korišćenje `Object.create()` je daleko najuobičajenije korišćenje, jer je to deo koji *može* da bude podvrgnut polifilingu. Postoji dodatni skup funkcionalnosti koje obezbeđuje standardni ES5 ugrađeni `Object.create()` koji ne može biti podvrgnut polifilingu za pre-ES5. Kao takav, on se i mnogo manje koristi. Zarad kompletnosti, pogledajmo i tu dodatnu funkcionalnost:

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

var myObject = Object.create( anotherObject, {
        b: {
                enumerable:false,
                writable:true,
                configurable:false,
                value:3
        },
        c: {
                enumerable:true,
                writable:false,
                configurable:false,
                value:4
        }
} );

myObject.hasOwnProperty( "a" ); // false
myObject.hasOwnProperty( "b" ); // true
myObject.hasOwnProperty( "c" ); // true

myObject.a; // 2
myObject.b; // 3
myObject.c; // 4
```

[20] Polifil (**polyfill**) je parče koda (obično baš **JavaScript** na Vebu) koje se koristi da obezbedi modernu funkcionalnost u starijim brauzerima koji je nativno ne podržavaju.

Drugi argument u `Object.create()` specificira imena svojstava koja se dodaju novokreiranom objektu putem deklaracije *deskriptora svojstva* za svako novo svojstvo. Zbog toga što polifiling deskriptora svojstva u pre-ES5 nije moguć, ni ta dodatna funkcionalnost u `Object.create()` ne može se obezbediti polifilingom.

Velika većina primena `Object.create()` koristi polifil-bezbedan podskup funkcionalnosti, pa većina programera koristi **parcijalni polifiling** u pre-ES5 okruženjima.

Neki programeri zastupaju mnogo stroži pristup koji se sastoji u tome da se ni jedna funkcija ne može podvrgnuti polifilingu ukoliko se ne može *potpuno* podvrgnuti polifilingu. Kako je `Object.create()` jedan od pomoćnih programa koji se mogu parcijalno podvrgnuti polifilingu, ta strožija perspektiva kaže da, ako u pre-ES5 okruženju koristite bilo koju funkcionalnost koju pruža `Object.create()`, treba da koristite kastomizovan pomoćni program umesto polifilinga i da se držite podalje od korišćenja imena `Object.create` u celosti. Možete, umesto `Object.create()` , da definišete spostveni pomoćni program koji bi mogao da izgleda ovako:

```
function createAndLinkObject(o) {
	function F(){}
	F.prototype= o;
	return new F();
}

var anotherObject = {
	a:2
};

var myObject = createAndLinkObject( anotherObject );

myObject.a; // 2
```

Vama, naravno, ostaje da sami donesete Vašu odluku.

4.2.3.5.1.2. Primarna funkcionalnost linkova

Na osnovu onoga što se naglašava pri opisivanju funkcionalnosti linkova među objektima (ni ovaj materijal nije izuzetak), lako se dolazi do ideje da linkovi između objekata *primarno* obezbeđuju neku vrstu "propadanja" za "nedostajuća" svojstva i metode. Iako to stvarno može da bude čest ishod, to nije baš pravi način da se razmišlja o funkcionalnosti `[[Prototype]]`. Posmatrajmo sledeći kod:

```
var drugiObjekat = {
	bolje:function() {
		console.log( "ovo je bolje!" );
	}
};

var mojObjekat = Object.create( drugiObjekat );

mojObjekat.bolje(); // "ovo je bolje!"
```

Taj kod će da radi zahvaljujući mehanizmu `[[Prototype]]`, ali ako ste ga napisali tako da `drugiObjekat` dejstvuje kao "propadanje" **samo u slučaju** da `mojObjekat` ne bi mogao da rukuje svojstvima/metodama koje bi neki drugi programer mogao da pokuša da pozove, to ozbiljno uvećava šanse da Vaš softver bude "magičniji" i teži za razumevanje i održavanje.

To ne znači da nema slučajeva kada su "propadanja" dobar dizajnerski obrazac, ali on nije vrlo uobičajen u JavaScript-u pa ako uhvatite sebe da to radite, temeljno razmotrite da li je to zaista prikladan i razuman dizajn.

Napomena: U ES6 je uvedena napredna funkcionalnost zvana `Proxy` koja može da obezbedi nešto od ponašanja tipa "method not found".

Dizajn softvera, na primer, pozivanje `mojObjekat.bolje()` koje će da radi čak i ako objekat `mojObjekat` nema metode `bolje()` uvodi određenu "magiju" u API dizajn koja može da bude neprijatno iznenađujuća za druge programere koji će da održavaju taj softver a, i za autora ako taj kod bude čitao nakon malo dužeg vremena.

Naravno, možete se razvijati API sa manje "magije" u njemu, a da se ipak iskoristi prednost snage `[[Prototype]]` povezivanja. Evo primera:

```
var drugiObjekat = {
        bolje:function() {
                console.log( "ovo je bolje!" );
        }
};

var mojObjekat = Object.create( drugiObjekat );

mojObjekat.uradiBolje = function() {
        this.bolje(); // interno delegiranje!
};

mojObjekat.uradiBolje(); // "ovo je bolje!"
```

Ovde se poziva `mojObjekat.uradiBolje()`, što je metod koji *zaista postoji* u objektu `mojObjekat`, čineći APi dizajn eksplicitnijim (manje "magičnim"). *Interno*, ova implementacija sledi **dizajnerski obrazac delegiranja**, koristeći prednost `[[Prototype]]` delegiranja svojstvu `drugiObjekat.bolje()`.

Drugim rečima, delegiranje će da bude manje iznenađujuće/zbunjujuće ako je interni implementacioni detalj nego ako je jasno izloženo u dizajnu API-a.

Da zaključimo: Nikada nemojte da izgubite važnu poentu "Programi moraju biti napisani da bi ih ljudi mogli čitati, a samo slučajno da bi ih izvršile mašine."

4.3. Sažetak

Kada se pokuša pristup svojstvu objekta koji to svojstvo nema, interno `[[Prototype]]` povezivanje objekta definiše gde će `[[Get]]` operacija dalje da gleda (sledeće mesto). To kaskadno povezivanje od objekta do objekta u suštini definiše "prototipski lanac" (engl. "prototype chain") - nešto slično lancu ugnježdenih dosega - objekata kroz koje treba prolaziti u razrešavanju svojstva.

Svi normalni objekti imaju ugrađen `Object.prototype` kao vrh prototipskog lanca (nešto kao globalni doseg u pretraživanju dosega), gde se zaustavlja rezolviranje svojstva ako ono nije nađeno nigde prethodno u lancu. U tom `Object.prototype` objektu su `toString()`, `valueOf()`, i nekoliko drugih uobičajenih pomoćnih programa koji objašnjavaju kako svi objekti u jeziku mogu da im pristupe.

Najuobičajeniji način da se dva objekta međusobno povežu je korišćenje ključne reči `new` sa funkcijskim pozivom, što u okviru svoja četiri koraka, kreira novi objekat povazan sa drugim objektom.

"Drugi objekat" sa kojim je novi objekat povezan je objekat referenciran proizvoljno nazvanim `.prototype` svojstvom funkcijskog poziva sa `new`. Funkcije pozvane sa `new` često se zovu "konstruktori", uprkos činjenici da one, u stvari, ne instanciraju klasu kao što to rade *konstruktori* u tradicionalnim klasno-orijentisanim jezicima.

Iako **može da izgleda** da ti JavaScript mehanizmi **predstavljaju "instanciranje klase" i "klasno nasleđivanje"** iz tradicionalnih klasno-orijentisanih jezika, **ključna razlika** je što se u JavaScript-u **ne radi kopiranje**. U stvari, **objekti se međusobno povezuju putem internog `[[Prototype]]` lanca**.

Iz različitih razloga, od kojih oni terminološkog porekla nisu manjina, "nasleđivanje" (i "prototipsko nasleđivanje") i svi drugi OO termini prosto nemaju smisla kada se gleda kako JavaScript *stvarno* radi jer je to drugačije od našeg forsiranog mentalnog modela nasleđivanja koji je, u stvari, inspirisan biološkim nasleđivanjem gde se neke stvari kopiraju iz jednog entiteta u drugi.

Ovde je "delegiranje" prikladniji termin zato što te **veze nisu *kopije*** već **linkovi delegiranja**.

Literatura uz poglavlje 4

https://blog.logrocket.com/copy-objects-in-javascript-complete-guide/

Poglavlje 5: OSNOVNO O FUNKCIJAMA

U ovom poglavlju izloženi su prvo uvodni sadržaji o funkcijama u matematici i funkcijama u programiranju, a zatim su izloženi sadržaji koji se bave funkcijama u jeziku JavaScript.

5.1. Funkcije u matematici i funkcije u programiranju

Termin funkcija može da znači različite stvari u zavisnosti od konteksta u kome se koristi. Za nas je ovde od prvenstvenog interesa kontekst programiranja i programskih jezika. Sa druge strane, u domen programiranja koncept funkcije je dospeo, pre svega, iz matematike. Zbog toga ćemo prvo ovde izložiti koncept funkcije u matematici (u meri u kojoj su funkcije iz matematike od interesa za programiranje). Nakon toga izložićemo pogled koji programiranje ima na koncept funkcije da bismo do kraja ovoga poglavlja detaljno razjasnili ulogu i način korišćenja funkcija u programskom jeziku JavaScript.

5.1.1. Funkcije u matematici

U ovom odeljku podsetićemo se koncepta funkcije kako se on pojavljuje u matematici zbog toga što se on u takvom obliku pojavljuje i u programskom jeziku JavaScript. Pri tome, usvojićemo najopštiju definiciju funkcije bez zalaženja u bogatstvo i raznovrsnost koncepta funkcije u matematici, jer je ta opšta definicija sasvim dovoljna da zadovolji naše potrebe za konceptom funkcije u programiranju. Definicija koju mi usvajamo za pojam funkcije je sledeća.

U matematici, ***funkcija*** je **relacija (preslikavanje) između skupova** koja **svakom elementu jednog skupa** (**domen funkcije**) ***pridružuje*** **tačno jedan element drugog skupa** (**kodomen funkcije**).

Ono što funkciju razlikuje od drugih preslikavanja je da jednom elementu domena može da se pridruži SAMO JEDAN element kodomena. Sa druge strane, VIŠE RAZLIČITIH elemenata domena može se preslika na ISTI element kodomena.

Funkcija (f), njen domen (X) i kodomen (Y) označavaju se notacijom

$$f : \mathrm{X} \rightarrow \mathrm{Y}$$

Vrednost funkcije f za element x iz X, označena sa $f(x)$ se naziva **slika x pod f**, ili **vrednost od f primenjene na x**.

Spomenućemo i pojam ***multivarijetetna funkcija*** (funkcija više promenljivih) koji je vrlo čest u praksi (matematičkoj i programerskoj).

Multivarijetetna funkcija ili **funkcija više promenljivih** je funkcija koja zavisi od više argumenata.

Funkcija više promenljivih definisana je na sledeći način.

Dekartov proizvod $[X_1 \times \cdots \times X_n]$ od n skupova $X_1, \cdots, X_n$ je skup svih n-torki $(x_1, \ldots, x_n)$ takvih da $x_i \in X_i$, $1 \leq i \leq n$. Tada je funkcija od n promenljivih funkcija

$$f: U \rightarrow Y$$

gde je domen $U \subseteq X_1 \times \cdots \times X_n$.

Na kraju, spomenućemo i pojam **kompozicije funkcija** koji ima izuzetno važnu ulogu u programiranju.

Ako su zadate dve funkcije $f: X \rightarrow Y$ i $g: Y \rightarrow Z$ takve da je domen funkcije g kodomen funkcije f, njihova kompozicija je funkcija $g \circ f: X \rightarrow Z$ definisana kao

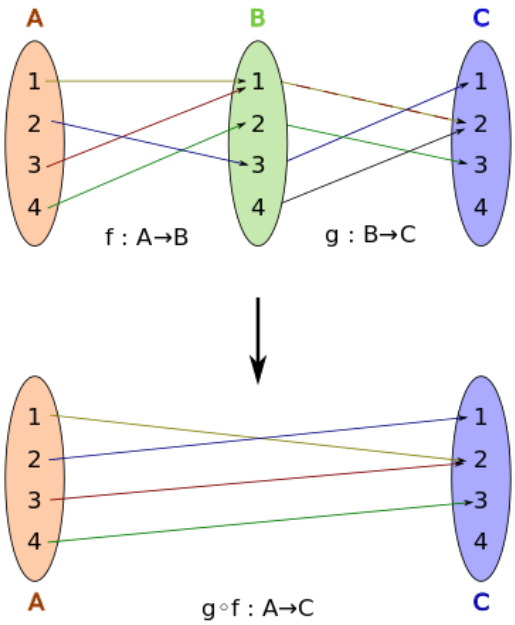
$$(g \circ f)(x) = g\big(f(x)\big)$$

Operacija kompozicije dveju funkcija definisana je samo ako je kodomen prve funkcije domen druge funkcije. Kompozicija funkcija je asocijativna, odnosno važi:

$$h \circ g \circ f = h \circ (g \circ f) = (h \circ g) \circ f$$

Funkcije identiteta id_X i id_Y (levi i desni identitet respektivno) za funkcije koje mapiraju X na Y su funkcije za koje važi: $f \circ id_X = id_Y \circ f = f$.

Vizualizacija kompozicije funkcija prikazana je na slici 6.1.



Slika 6.1 Kompozicija funkcija

5.1.2. Funkcije u programiranju

Iz programerske tačke gledišta, funkcija je blok organizovanog, po mogućstvu višekratno upotrebljivog koda koji se koristi da izvrši jedan, najčešće jednostavan zadatak.

Sadržaj funkcije je njeno telo, koje je deo programskog koda koji se izvršava kada se funkcija pozove. Funkcija može biti napisana tako da očekuje da dobije jednu ili više vrednosti podataka od pozivajućeg programa kojima će da zameni svoje parametre. Program koji poziva funkciju daje za ove parametre stvarne vrednosti koje se nazivaju argumenti. Za prosleđivanje argumenata koriste se mehanizmi sumirani u Tabeli 5.1:

Tabela 5.1 Načini prosleđivanja argumenata funkciji

Način prosleđivanja argumenata	Značenje
Po vrednosti	Argument se evaluira i kopija dobijene vrednosti se prosleđuje funkciji.
Po referenci	Funkciji se prosleđuje referenca na argument (tipično adresa memorijske lokacije argumenta.
Po rezultatu	Vrednost koju je sračunala funkcija za dati argument kopira se natrag u taj argument pri povratku iz funkcije.

Način prosleđivanja argumenata	Značenje
Po vrednosti-rezultatu	Kopija vrednosti argumenta prosleđuje se funkciji i vrednost koju je za taj argument sračunala funkcija kopira se natrag u argument pri povratku iz funkcije.
Po imenu	Kao makro - parametri se zamenjuju neevaluiranim izrazima koji predstavljaju argumente, zatim se argumenti evaluiraju u kontekstu onog ko je funkciju pozvao svaki put kada rutina koristi parametar.
Po konstantnoj vrednosti	Kao poziv po vrednosti, osim što se parametar tretira kao konstanta.

Osnovni **efekat** koji **proizvodi poziv funkcije** je da **vrati vrednost** – ono što je funkcija sračunala.

Pored vraćene vrednosti, poziv funkcije može imati i **sporedne efekte** (željene ili neželjene) kao što su čitanje sa perifernog uređaja ili ispis na njega, kreiranje datoteke, zaustavljanje ili odlaganje izvršavanja programa, nenamerna ili namerna modifikovanja podataka u memoriji računara. Funkcija sa sporednim efektima može da vrati različite rezultate kada se pozove, čak i ako je pozvana sa istim argumentima. Široka upotreba funkcija sa sporednim efektima je karakteristika imperativnih programskih jezika, ali se oni koriste i u drugim paradigmama programiranja jer nije moguće napraviti upotrebljiv program bez sporednih efekata. Pri tome, svi programski jezici nastoje da kontrolišu sporedne efekte sa osnovnim ciljem da izbegnu one neželjene.

Postoje i funkcije koje ne vraćaju nikakvu vrednost ili vraćaju `null-vrednost` se uobičajeno nazivaju **procedura**. Procedure obično modifikuju svoje argumente što je takođe sporedni efekat i suštinski su deo proceduralnog programiranja.

Funkcija se može kodirati tako da se može rekurzivno[21] pozivati, što omogućava direktnu implementaciju funkcija definisanih matematičkom indukcijom i rekurzivnim algoritmima.

Funkcija čija je svrha da izračuna i vrati jednu logičku vrednost (`true` ili `false`) ponekad se naziva **predikat**.

5.2. Funkcije u JavaScript-u

U jeziku JavaScript funkcije imaju status "građana prvog reda". To znači da se sa funkcijama može raditi sve ono što se može raditi i sa drugim tipovima, uključujući i primitivne tipove.

Formalno, funkcije se mogu dodeljivati imenima kao vrednosti, mogu se porediti, mogu biti operandi u izrazima, mogu se prosleđivati kao argumenti i mogu se vraćati kao povratne vrednosti. Naravno, nije svaka takva radnja smislena – šta bi, na primer, značila operacija poređenja kojom se utvrđuje da li je jedna funkcija veća od druge funkcije[22]? U stvari, ključne karakteristike funkcije u JavaScript-u su sledeće.

Da može da se prosleđuje kao argument

```
function a() { ... }
function b(a) { ... }
b(a); // ovde se poziva funkcija b sa argumentom koji je funkcija a.
```

Da može da se vraća kao rezultat druge funkcije

[21] Kolokvijalno, rekurzivni poziv funkcije je kada funkcija “poziva samu sebe”.

[22] U matematici postoji pojam jednakih (ekvivalentnih) funkcija: dve funkcije f i g su jednake ako imaju isti domen i isti kodomen i za svako x iz domena važi f(x) = g(x).

```
function a() {
    function b() { ... }
    return b; // povratna vrednost funkcije a je funkcija b
}
```

Da može da se dodeli varijabli kao podatak

```
var a = function() { ... };
```

Da može da poseduje svojstva i metode

```
function a() { ... }
var obj = {};
a.call(obj);
```

Da može da se definiše bilo gde u izrazu što omogućuje ugneždavanje

```
function a() {
    function b() {
        function c() {
            ...
        }
        c();
    }
    b();
}
a();
```

Svaki programski jezik, pa i JavaScript, pored ugrađenih funkcija koje su sastavni deo jezika, omogućuje programeru da definiše sopstvene funkcije.

5.2.1. Kreiranje i pozivanje funkcije

Za korišćenje funkcija potrebno je uraditi dve stvari: (1) ***kreirati/definisati funkciju***, i (2) „zatražiti" od funkcije da uradi ono za šta je definisana – ***pozvati funkciju***. Nakon što završi svoje izvršavanje, funkcija treba da vrati kontrolu programa onome ko ju je pozvao.

Za kreiranje funkcije koristi se deklaracija funkcije. Za definisanje funkcija u JavaScript-u postoje dve bazične sintakse. Jedna koristi samostalnu deklaracionu naredbu `function`, a druga je ***funkcijski izraz***. Pored toga, postoji i ***streličasta*** sintaksa.

5.2.1.1. Bazične sintakse

5.2.1.1.1. Sintaksa samostalne naredbe

Deklaracija funkcije u jeziku JavaScript korišćenjem samostalne deklaracione naredbe izgleda ovako:

```
function imeFunkcije(parametri) {
        // ovo što sledi je telo funkcije
        console.log("Poziv ImeFunkcije sa argumentom: ", parametri);
        return parametri + parametri // Ovo je naredba koja vraća vrednost
}
```

Ovde je `function` obavezna ključna reč, `imeFunkcije` je identifikator funkcije – ime funkcije, a (`parametri`) je lista ulaznih parametara. Lista parametara sadrži imena parametara razdvojena zarezima a može da bude i prazna. Iza liste ulaznih parametara sledi telo funkcije zatvoreno u vitičaste zagrade. U telu funkcije je kod koji opisuje ono što funkcija radi. Naredba `return` u telu funkcije vraća rezultat izvršavanja funkcije (u primeru je to udvostručena vrednost argumenta) i vraća kontrolu toka izvršavanja onome ko je funkciju pozvao.

Ključna reč `function` kreira varijablu/objekat tipa `function` sa imenom `imeFunkcije` u koju se smešta string koji sadrži izvorni kod definicije funkcije. Sledeća linija koda to ilustruje:

```
console.log(' imeFunkcije \n', typeof imeFunkcije, '\n', imeFunkcije );
```

Izvršavanje ove linije koda rezultuje sledećim ispisom:

imeFunkcije
function
ƒ imeFunkcije(parametri) {
// ovo što sledi je telo funkcije
console.log("Poziv ImeFunkcije sa argumentom: ", parametri);
return parametri + parametri // Ovo je naredba koja vraća vrednost
}

Sada se sa varijablom mogu raditi stvari koje su legalne za bilo koju drugu varijablu. Na primer, može se njena vrednost dodeliti drugoj varijabli

```
let mojaFunkcija = imeFunkcije
```

što za rezultat ima sledeće:

mojaFunkcija
function
ƒ imeFunkcije(parametri) {
// ovo što sledi je telo funkcije
console.log("Poziv ImeFunkcije sa argumentom: ", parametri);
return parametri + parametri // Ovo je naredba koja vraća vrednost
}

Pozivanje funkcije vrši se **navođenjem imena** iza koga sledi **par malih zagrada** u kojima se navode **konkretne vrednosti argumenata** za poziv. Sledi primer:

```
console.log (imeFunkcije (1))
console.log ('Ja sam završila ovaj poziv i vratila sam kontrolu onome ko me je pozvao')
console.log (imeFunkcije('Dobar vam dan!'))
console.log ('Ja sam završila ovaj poziv i vratila sam kontrolu onome ko me je pozvao')
```

Rezultat izvršavanja ovih linija koda biće sledeći ispis na konzoli:

Poziv ImeFunkcije sa argumentom: 1
2
Ja sam završila ovaj poziv i vratila sam kontrolu onome ko me je pozvao
Poziv ImeFunkcije sa argumentom: Dobar vam dan!
Dobar vam dan!Dobar vam dan!
Ja sam završila ovaj poziv i vratila sam kontrolu onome ko me je pozvao

5.2.1.1.2. Sintaksa funkcijskog izraza

Funkcijski izraz je sintaksa koja kreira funkciju i već pri kreiranju eksplicitno dodeljuje funkciju varijabli kao bilo koju drugu vrednost. Ovakva funkcija, šta god ona bila, predstavlja vrednost uskladištenu u varijablu kojoj je dodeljena. Na dalje, ova funkcija je samo vrednost kao i bilo koja druga vrednost. Evo primera:

```
let kaziZdravo = function() {
console.log( "Zdravo!" );
};
```

Ovde se varijabla kojoj se dodeljuje funkcija zove `kaziZdravo` a sama funkcije nema drugo posebno ime.

U JavaScriptu, vrednost funkcijskog izraza je *definicija (kod) same funkcija* a *ne rezultat njenog izvršavanja* što znači da će sledeća linija koda

```
console.log("kaziZdravo: ",  kaziZdravo)
```

da prikaže na konzoli string:

let kaziZdravo = function() {console.log("Zdravo!");};

Funkcija se poziva navođenjem imena varijable kojoj je dodeljena iza koga sledi par malih zagrada u kojima su navedeni argumenti. U primeru to izgleda ovako:

```
kaziZdravo();
```

Rezultat izvršavanja ove linije je sledeći ispis na konzoli:

```
Zdravo!
```

Evo i primera u kome se u JavaScript-u definiše funkcija koja ima isto značenje kao i matematička funkcija $f(x) = 2x$. Ova definicija funkcije interpretira string `f(x) = 2x` na sledeći način: `f od x je` jednako je `2x` i kada se izvrši treba da vrati udvostručenu vrednost onoga što je primljeno kao `x`. Na primer, `f(2)`, ima isto značenje kao `4.`

Prvi način je da je definišemo kanonički, kao samostalnu naredbu:

```
function double (x){
                  return  x * 2;
         }
console.log(double); // Ispisaće na konzoli: "ƒ double (x){return  x * 2;}"
console.log(double(2));       // Ispisaće na konzoli: 4
```

Drugi način je da je definišemo kao funkcijski izraz:

```
const double = function (x){
                  return  x * 2;
               }

console.log(double); // Ispisaće na konzoli: "ƒ (x){return  x * 2;}"
console.log(double(2));       // Ispisaće na konzoli: 4
```

Sintaksa dozvoljava da se pri definisanju funkcije funkcijskim izrazom funkciji dodeli ime različito od imena varijable kojoj se dodeljuje funkcija pri definisanju , odnosno sledeći kod je sintaksno ispravan:

```
let kaziZdravo = function zdravoSvima() {
console.log( "Zdravo!" );

};
```

Međutim, ovako kreirano ime funkcije može se koristiti samo u telu funkcije. Pokušaj korišćenja tog imena izvan tela funkcije generiše grešku ako isto ime nije negde drugo deklarisano ili se uzima druga vrednost ako je negde deklarisana sa imenom funkcije.

Evo primera u kome se funkcija poziva svojim deklarisanim imenom koje nigde drugo nije deklarisano:

```
let kaziZdravo = function zdravoSvima() {
console.log( "Zdravo!" );
};
kaziZdravo()           // Ispis: Zdravo!
```

```
zdravoSvima()           /* Vraća grešku Uncaught ReferenceError: zdravoSvima
is not
                        defined */
```

A evo i primera u kome je isto ime deklarisano kao varijabla kojoj se dodeljuje izvorni kod druge funkcije (funkcija koja ispisuje drugačiji string na konzolu) takođe definisane funkcijskim izrazom u kome NEMA posebnog imena funkcije:

```
let kaziZdravo = function zdravoSvima() { // ZdravoSvima je posebno ime
funkcije
console.log( "Zdravo!" );
};

let zdravoSvima = function() { // Ova funkcija nema posebno ime
    console.log( "Zdravo svima!" );
};

kaziZdravo()            // ispis: Zdravo!
zdravoSvima()           // ispis: Zdravo svima!
```

U slučaju poziva u telu funkcije referenciranjem na deklarisano ime može se napraviti rekurzija. Sledeći kod će tri puta ispisati string `"Zdravo!"`:

```
const poziv = {
      pozivIt: function kaziZdravo (n) {
            console.log( "Zdravo!" );
            if (n <= 1) {
                  return;
            }
            return kaziZdravo(n - 1);
  },
};
poziv.pozivIt(3);
```

Ključna razlika između funkcije deklarisane ključnom reči `function` i funkcijskog izraza je u sledećem:

1. **Sintaksa: kako ih razlikovati u kodu.**
   - *Kanonička deklaracija:* funkcija je definisana **posebnom naredbom i glavnom toku koda**

     ```
     // kanonička function deklaracija
     function sum(a, b) {
     return a + b;
     }
     ```

   - *Funkcijski izraz:* funkcija definisana **unutar izraza ili drugog sintaksnog konstrukta**. Ovde je funkcija sa desne strane izraza dodele (**=**):

     ```
     // funkcijski izraz
     let sumFE = function(a, b) {
     return a + b;
     };
     ```

2. **Trenutak kreiranja funkcije**
   1. **Funkcija deklarisana kanoničkom deklaracijom** može se pozvati u kodu pre nego što je navedena njena definicija. Na primer:

      ```
      console.log(sum(2,3));     // prvo poziv

      function sum(a, b) { //zatim definicija funkcije
      ```

```
return a + b;
}
```

- **Funkcijski izraz** može se pozivati u kodu tek nakon dela koda koji ga definiše. Na primer:

```
//Nije ispravno
console.log(sumFE(2,3));
let sumFE = function(a, b) {
return a + b;
};
//Ispravno je
let sumFE = function(a, b) {
return a + b;
};
console.log(sumFE(2,3));
//Ispravno je i ovo
let sumFE = function(a, b) {
return a + b;
}(2,3);
```

- **Blok dosezanja:** Ako se deklaracija funkcije pomoću funkcijskog izraza nađe u bloku koda, funkcija je vidljiva na svakom mestu u tom bloku, ali nije vidljiva izvan tog bloka. Dugim rečima, funkcija definisana u globalnom dosegu može da pristupi svim varijablama definisanim u globalnom dosegu. Funkcija definisana unutar druge funkcije može da pristupi i varijablama unutar roditeljske funkcije i svim drugim varijablama kojima može da pristupi roditeljska funkcija Slede primeri:

```
// Samostalna naredba
const num1 = 20;
const num2 = 3;
const name = "Mikica";

// Ova funkcija je definisana u globalnom dosegu
function pomnozi() {
  return num1 * num2;
}

console.log(pomnozi ()); // 60

// Primer ugnježdene funkcije
function dobaviSkor() {
  const num1 = 2;
  const num2 = 3;

  function saberi () {
    return `${name} ima skor ${num1 + num2}`;
  }

  return saberi();
}
```

```
console.log(dobaviSkor ());   // "Chamakh scored 5"
console.log(saberi(3,3))      // funkcija saberi() nije definisana

// Funkcijski izraz
const num1 = 20;
const num2 = 3;
const name = "Mikica";

// funkcija je definisana u globalnom dosegu
  return num1 * num2;
}

console.log(pomnozi()); // 60

// primer ugnježdene fukncije
function dobaviSkor() {
  const num1 = 2;
  const num2 = 3;

  let saberi = function add() {
    return `${name} ima skor ${num1 + num2}`;
  }

  return saberi();
}

console.log(dobaviSkor());    // "Mikica ima skor 5"
console.log(saberi(3,3))      // funkcija saberi() nije definisana
```

Za razliku od nekih drugih funkcionalnih jezika kao što je Haskel, u JavaScript-u **male zagrade ovde imaju suštinski smisao** – one označavaju da je u pitanju **poziv funkcije**. Naime, bez njih funkcija ne bi bila pozvana:

```
double;     // Vraća pokazivač na funkciju
double 2;   // Vraća grešku: SyntaxError: Unexpected number
```

5.2.1.2. Streličasta sintaksa

Postoji i treća jednostavna i koncizna sintaksa koja je često bolja od funkcijskog izraza i vrlo je popularna danas pa ćemo je i mi najčešće koristiti u ovoj knjizi. To je ***streličasta sintaksa***. Ona izgleda ovako:

```
const identifikator = ( arg1, arg2, ...argN) => izraz
```

Ovim se kreira funkcija **`identifikator`** koja prima argumente **`arg1 ... argN`**, vrši evaluaciju izraza **`izaz`** sa desne strane strelice na bazi primljenih argumenata i vraća rezultat evaluacije.

U slučaju višelinijskog tela funkcije, ono se stavlja u vitičaste zagrade i mora obavezno da ima naredbu `return`:

```
let sum = (a, b) => {  // vitičasta zagrada otvara višelinijsku funkciju
let result = a + b;
      return result; // ako se koristi vitičasta zagrada, obavezana je
    }                // eksplicitna "return" naredba
```

Naš JavaScript funkcijski izraz `double(x)` u streličastoj sintaksi izgleda ovako:

```
const double = x => x * 2;
```

Definicija funkcije (izvorni kod) može da se prikaže navođenjem samo imena funkcije ili korišćenjem metode `toString()`:

```
const double = x => x * 2;
double;            // Vratiće: x => x * 2
double.toString(); // Vratiće:  'x => x * 2'
```

Pozivanje je isto kao i za kanonički deklarisane funkcije. Na primer:

```
const double = x => x * 2
console.log ('Ja sam poziv sa argumentom 2 i vraćam rezultat: ', double(2))
console.log ('Ja sam varijabla u koju je upisan izvorni kod i vraćam: ',
double)
```

5.2.2. Anonimne funkcije

Anonimna funkcija je koncept koji se pojavljuje u većini programskih jezika. Kada se kreira anonimna funkcija, ona je deklarisana bez ikakvog identifikatora. To je razlika između normalne funkcije i anonimne funkcije.

I u JavaScript-u, anonimna funkcija je vrsta funkcije koja nema deklarisano ime. Shodno tome, anonimna funkcija se ne može definisati korišćenjem sintakse samostalne naredbe `function`. Može se deklarisati korišćenjem sintakse funkcijskog izraza i streličaste sintakse.

Primer deklarisanja anonimne funkcije putem funkcijskog izraza sledi (izraz je označen zatvaranjem u par malih zagrada):

```
(function () {
    console.log('Ovo je anonimna funkcija');
});
```

Ako se koristi streličasta sintaksa, definicija izgleda ovako:

```
() => console.log('Ovo je anonimna funkcija');
```

Kao i u drugim jezicima, anonimne funkcije se i u JavaScript-u koriste za prosleđivanje funkcija drugim funkcijama kroz listu argumenata. Primer je:

```
setTimeout(function () {
    console.log('Izvrši nakon jedne sekunde')
}, 1000);
```

U ovom primeru se anonimna funkcija `function () {console.log('Izvrši nakon jedne sekunde')}` prosleđuje funkciji `setTimeout()` da funkcija `setTimeout()` pokrene ispis na konzolu nakon 1 sekunde.

5.2.3. Upravljanje ulazom u funkciju

Iako je u JavaScript-u moguće da funkcija i nema nikakav ulaz ili da se funkciji ulazi prosljediti putem deljenog stanja, mi se time ovde nećemo baviti jer to nije u duhu dobre prakse programiranja. Dobra praksa programiranja je da se ulazi funkciji saopštavaju ***isključivo putem ulaznih parametara***. To znači da se pri pozivu funkcije ulazne vrednosti navode onako kako to (uz određeni stepen slobode) propisuje lista parametara u deklaraciji funkcije U nastavku ćemo prikazati mogućnosti upravljanja ulazom u funkciju koje pruža JavaScript.

5.2.3.1. Podrazumevane vrednosti parametara

Postoje situacije u programiranju u kojima je zgodno da se vrednosti nekih parametara (argumenti) zadaju unapred uz mogućnost da se, pri pozivu funkcije, izmenu. Argumenti koji se na ovaj način mogu zadavati zovu se ***podrazumevane vrednosti parametara***. Podrazumevana vrednost se koristi ukoliko za parametar nije eksplicitno zadata (u pozivu navedena) njegova vrednost (argument). U protivnom koristi se eksplicitno zadata vrednost.

JavaScript podržava podrazumevane vrednosti parametara.

Na primer, funkcija `iliNula()`, čija definicija sledi, radi kao funkcija identiteta (vraća prosleđenu vrednost) osim ako se pozove sa argumentom `undefined`, ili joj se uopšte ne prosledi argument. U tim slučajevima, funkcija će da vrati nulu:

```
const iliNula = (n = 0) => n;
console.log('iliNula(2): ', iliNula(2))   // iliNula (2): 2
console.log('iliNula(false):', iliNula(false))  // iliNula(false): false
console.log('iliNula(null): ', iliNula(null))   // iliNula(null): null
console.log('iliNula(undefined): ', iliNula(undefined))/* iliNula
                                                    (undefined): 0 */
console.log('iliNula(): ', iliNula())     // iliNula(): 0
```

Podrazumevana vrednost se postavlja tako što se u signaturi (deklaraciji) funkcije iza odgovarajućeg parametra navede znak `=` i iza znaka navede se ta vrednost (kao što je `n = 0` u deklaraciji funkcije `iliNula`). Ako se na taj način dodeli pretpostavljena vrednost, alati za zaključivanje tipa kao što su **Tern.js, Flow**, ili **TypeScript** mogu automatski da zaključe signaturu tipa povratne vrednosti funkcije (u primeru to je tip vraćene vrednosti n) i ako se anotacije tipa ne deklarišu eksplicitno. Dakle, HM signatura funkcije mogla bi da se zapiše ovako:

```
// iliNula :: a -> a
```

U praksi je ovo pomoć programeru. Sa dobrim signaturama integrisana okruženja ili drugi pomoćni programi mogu da obezbede prikazivanje signatura pri unosu poziva funkcija (vrlo raširena praksa) i da doprinesu boljem razumevanju načina korišćenja funkcija. Podrazumevane dodele pomažu pisanju samodokumentujućeg koda.

Napomena: Podrazumevani parametri se ne uzimaju u obzir pri radu sa svojstvom funkcije `length` koja u sebi skladišti informaciju o deklarisanom broju parametara (arnosti funkcije). To može da dovede do otkazivanja pomoćnih programa kao što je auto-kuriranje koji zavise od `length` vrednosti. Neki pomoćni programi za kuriranje (n.pr `lodash/curry`) dopuštaju prosleđivanje prilagođene arnosti, ali to treba pažljivo raditi. Ovde samo da napomenemo da je *kuriranje* specifičnost načina izvršavanja funkcija u funkcionalnom programiranju i svodi se na to da se funkcija sa više parametra izvršava tako što obrađuje "jedan-po-jedan" parametar u svakom pozivu vraćajući funkciju koja prima sledeći parametar. O kuriranju ćemo još detaljno pričati u delu knjige pod naslovom Funkcionalno programiranje u JavaScript-u.

5.2.3.2. Imenovani argumenti

JavaScript funkcije mogu da primaju literale objekata kao argumente i da koriste destrukturirajuće dodele u signaturi parametra sa ciljem postizanja ekvivalenta imenovanim argumentima. Moguće je dodeljivati pretpostavljene vrednosti parametrima korišćenjem sintakse podrazumevanog parametra kao u sledećem primeru:

```
const kreirajKorisnika = ({
ime = 'Anonimni korisnik',
avatarSlicica = '/avatars/anonymous.png'
}) => ({
```

```
        ime,
        avatarSlicica,
       }
      );

/* U ovom pozivu funkcije kreirajKorisnika nije zadat argument pa se
koristi podrazumevana vrednost (objekat koji je naveden u signaturi
funkcije */

const podrazumevaniKorisnik = kreirajKorisnika({})
console.log (podrazumevaniKorisnik)

/* Vraća:
{ime: 'Anonimni korisnik', avatarSlicica: 'avatars/anonymous.png'}
*/

/* U ovom pozivu funkcije kreirajKorisnika je eksplicitno navedena
vrednost parametra, pa se koristi navedena vrednost (objekat koji je
naveden u pozivu funkcije */

const peraKorinsik = kreirajKorisnika({
ime: 'Pera',
avatarSlicica: 'avatars/shades-emoji.png'
});

console.log (peraKorinsik)

/* Vraća:
{ime: 'Pera', avatarSlicica: 'avatars/shades-emoji.png'}
*/
```

5.2.3.3. `rest` i `spread` sintaksa

Još jedna mogućnost funkcija u JavaScript-u je da objedine grupu (preostalih) argumenata u signaturama funkcija koristeći `rest` *operator*. Simbol za `rest` operator je niz od tri uzastopne tačke:**`...`**

`Rest` operator (**`...`**) smešta pojedinačne argumente u niz (smešta ih u članove niza).

Evo i primera. Sledeća funkcija prosto 'preskače' prvi argument a preostale argumente (`rest`) vraća u obliku niza:

```
const aTail = (head, ...tail) => tail;
aTail(1, 2, 3); // Vraća: [2, 3]
```

“Inverzni” operator operatoru `rest` je operator `spread`. On raspoređuje (rasprostire) elemente niza na pojedinačne argumente. Evo primera:

```
const shiftToLast = (head, ...tail) => [...tail, head];

shiftToLast(1, 2, 3); // Vraća: [2, 3, 1]
```

Nizovi u JavaScript-u imaju iterator koji se poziva pri korišćenju operatora `spread`. Iterator isporučuje vrednost za svaku stavku niza. U izrazu `[...tail, head]` iterator kopira svaki element u redosledu iz niza `tail` u novi niz koji kreira okružujuća literalna notacija. Kako je zaglavlje već individualni element, ono se samo "bućne" na kraj niza i posao je završen.

5.2.4. Ugnježđene funkcije

Funkcija se naziva „ugnježđena“ kada je kreirana unutar druge funkcije. U JavaScript-u se to može uraditi korišćenjem bilo koje od tri navedene sintakse za definisanje funkcije . Sledi primer u kome je ugnneždena funkcija deklarisana sintaksom samostalne naredbe:

```
// spoljna funkcija
function kaziZdravoDovidjenja(ime, prezime) {

// pomoćna ugnježđena funkcija – sintaksa samostalne naredbe
  function pribaviPunoIme() {
    return ime + " " + prezime;
  }
/* Definicija putem funkcijskog izraza izgledala bi ovako:
   let pribaviPunoIme = function () {
     return ime + " " + prezime;
  } */
/* Definicija putem streličaste sintakse izgledala bi ovako:
   const pribaviPunoIme  = () => ime + " " + prezime; */

  console.log( "Zdravo, " + pribaviPunoIme () );
  console.log ( "Doviđenja, " + pribaviPunoIme () );
}

// poziv spoljne funkcije
kaziZdravoDovidjenja("Marija", "Zdravković")
```

Rezultat je log:
Zdravo, Marija Zdravković
Doviđenja, Marija Zdravković

U ovom primeru, funkcija `pribaviPunoIme()` koja je deklarisana unutar funkcije ima pristup spoljnim varijablama `ime` i `prezime` deklarisanim u funkciji `kaziZdravoDovidjenja()`. Mehanizam koji to omogućuje je pokazivač na leksički doseg spoljašnje funkcije (Poglavlje 3, odeljak 3.3.1. Dosezanje).

Ugnježđena funkcija može biti vraćena ili kao svojstvo novog objekta ili kao rezultat sama po sebi. Zatim se može koristiti negde drugde. Bez obzira gde se koristi, ona i dalje ima pristup istim spoljnim varijablama. Primer u kome funkcija `napraviBrojac()` kreira i vraća kao svoj rezultat funkciju `brojac` koja vraća sledeći broj pri svakom pozivanju to ilustruje:

```
function napraviBrojac() {
  let broj = 0;

  return function() {
    return broj++;
  };
}

let brojac = napraviBrojac(); // poziv spoljne funkcije

console.log (brojac) // ƒ () { return broj++;} – vraćena ugnježđena funkcija

// izvršavanje vraćene funkcije
console.log( brojac() ); // 0
```

```
console.log( brojac() ); // 1
console.log( brojac() ); // 2
```

Ovaj primer demonstrira još jednu mogućnost funkcija u JavaScript-u koja se zove ***zatvaranje*** o kome ćemo posebno govoriti u odeljku koji neposredno sledi.

5.2.5. Zatvaranje

U prethodnim odeljcima upoznali smo se sa glavnim osobinama funkcija i mogućnostima JavaScript-a koje se odnose na rad sa funkcijama. Videli smo kako se kreiraju i kako se pozivaju funkcije, videli smo i neke načine prosleđivanja ulaznih vrednosti funkciji i videli smo, konačno, da funkcija može pristupiti promenljivim van nje („spoljne“ promenljive).

Pristupanje funkcije spoljnim promenljivim je pitanje koje treba posebno razjasniti zato što, za razliku od unutrašnjih promenljivih koje se u potpunosti kontrolišu od strane/iz same funkcije, spoljne promenljive podležu uticajima koji su kreirani van same funkcije. Dakle, pitanje na koje želimo da odgovorimo je: Šta se dešava ako se spoljne promenljive promene od kreiranja funkcije i šta se dešava ako se funkcija prosledi kao argument i pozove sa drugog mesta u kodu?

Funkcije formiraju opseg doseznja[23] za promenljive—to znači da se varijablama definisanim unutar funkcije ne može pristupiti nigde izvan funkcije. Opseg funkcije nasleđuje se od svih gornjih opsega. Na primer, funkcija definisana u globalnom opsegu može pristupiti svim varijablama definisanim u globalnom opsegu. Funkcija definisana unutar druge funkcije takođe može pristupiti svim varijablama definisanim u njenoj roditeljskoj funkciji, kao i svim drugim varijablama kojima roditeljska funkcija ima pristup. S druge strane, roditeljska funkcija (i bilo koji drugi roditeljski opseg) nema pristup varijablama i funkcijama definisanim u unutrašnjoj funkciji. Ovo obezbeđuje neku vrstu enkapsulacije za promenljive u unutrašnjoj funkciji.

U nastavku ćemo ovo ukratko da objasnimo a detaljno objašnjenje možete naći u poglavlju koje se bavi JavaScript okruženjem.

5.2.5.1. Leksičko okruženje

U JavaScript-u, svaka funkcija, blok koda `{...}` i skript u celini imaju interni (skriveni) povezani objekat koji se zove **Leksičko okruženje** koji se sastoji od dva dela:

1. **Zapis okruženja** (Environment Record) koje je objekat u čijim svojstvima se skladište sve lokalne varijable (uključujući i vrednosti).
2. **Referenca na spoljašnje leksičko okruženje** koje pokazuje na leksičko kruženje spoljašnjeg koda (koda unutar koga se nalazi ta funkcija, blok ili skript).

Upravo referenca na spoljašnje leksičko okruženje je mehanizam koji funkcijama u JavaScript-u omogućuje da pristupi spoljnim varijablama. Kada kod se želi pristup varijabli ona se prvo traži u unutrašnjem leksičkom okruženju, zatim spoljašnjem, zatim spoljašnjem od tog spoljašnjeg, i tako dalje do globalnog. Ako se varijabla ne pronađe ni u jednom od ovih okruženja, imamo dva ponašanja:

- U striktnom režimu se generiše terminirajuća greška.
- U ne-striktnom režimu se kreira varijabla sa datim imenom u globalnom leksičkom okruženju.

Varijabla se ažurira u leksičkom okruženju u kome ‘živi’, u kome je kreirana

[23] Terminom “dosezanje” označava se vidljivost varijabli. Dosezanje može da bude globalno kada svi delovi nekog programa mogu da “vide” varijable u tom opsegu, ili može da bude lokalno kada varijable mogu da se vide samo iz nekih delova programa (na primer, iz funkcije).

5.2.5.2. Zatvaranje

Koncept zvani ***Zatvaranje*** igra vrlo važnu ulogu u jeziku JavaScript. ***Zatvaranje*** je funkcija koja pamti svoje spoljašnje promenljive i može im pristupiti i nakon što spoljašnja funkcija završi sa izvršavanjem. U nekim jezicima to nije ni moguće, u nekim funkcija treba da bude napisana na poseban način da bi se to ostvarilo.

U JavaScript-u, sve funkcije su prirodno zatvaranja, izuzev funkcija koje se kreiraju putem konstruktora (ključna reč new).

Primer brojača koji smo prikazali kao ilustraciju ugneždavanja funkcija je dobar i za ilustrovanje zatvaranja. Pa da ga pogledamo.

```
function napraviBrojac() {
  let broj = 0;

  return function() {
    return broj++;
  };
}

let brojac = napraviBrojac(); // poziv spoljne funkcije

console.log (brojac) // ƒ () { return broj++;} – vraćena ugnježđena funkcija

// izvršavanje vraćene funkcije
console.log( brojac() ); // 0
console.log( brojac() ); // 1
console.log( brojac() ); // 2
```

U ovom primeru, vraćena ugneždena funkcija `brojac` je zatvaranje.

Svaki put kada se pozove funkcija `brojac`, ona ima pristup leksičkom okruženju funkcije `napraviBrojac()`. U tom okruženju deklarisana je varijabla `broj`. Kada se pozove funkcija `napraviBrojac()`, ona će u svom leksičkom okruženju da napravi varijablu `broj` sa vrednošću `0` i vratiće unutrašnju funkciju „dopunjenu“ leksičkim okruženjem funkcije `napraviBrojac()`. Pri pozivanju, vraćena funkcija uvek nalazi varijablu `broj` u toj svojoj „dopuni“ i na tom mestu, dakle u leksičkom okruženju funkcije `napraviBrojac()` je povećava za 1. Kada se vraćena funkcija pozove prvi put ona će da naiđe na varijablu `broj` sa vrednošću 0, vratiće tu vrednost (0) i uvećaće je za 1. Kada se pozove drugi put, naići će na varijablu `broj` sa vrednošću 1, vratiće je i uvećaće je za 1. Treći poziv nailazi na varijablu `broj` sa vrednošću 2, vraća vrednost 2 i uvećava je za 1. Na kraju će vrednost varijable `broj` u leksičkom okruženju funkcije `napraviBrojac()` biti 3.

U slučaju kada je funkcija kreirana putem konstruktora (ključna reč new) njen `[[Environment]]` pokazivač postavlja se da pokazuje na globalno leksičko okruženje a ne na leksičko okruženje svoje spoljašnje funkcije. Zbog toga funkcija kreirana pomoću konstruktora nema zatvaranje nad funkcijom u kojoj je definisana. Evo i primera prerađene funkcije `napraviBrojac()`:

```
function napraviBrojac() {
  let broj = 0;
  function Brojac (broj) {
       return broj++;
  };
    return new Function (Brojac)
}
```

```
let noviBrojac = napraviBrojac()
console.log (noviBrojac) // vraćena funkcija
console.log (noviBrojac()); // poziv – vraća undefined
console.log (noviBrojac()); // poziv – vraća undefined
console.log (noviBrojac()); // poziv – vraća undefined
```

Sledeći jednostavan primer očiglednije ilustruje ponašanje funkcija kreiranih konstruktorom u odnosu na zatvaranje.

```
// Unutrašnja funkcija kreirana konstruktorom
function spoljnaFunkcija() {
  let vrednost = "neki tekst";

  let func = new Function('console.log(vrednost)');
  return func;
}

spoljnaFunkcija ()(); // error: vrednost is not defined
```

Ako se ne koristi konstruktor, zatvaranje se kreira:

```
// Unutrašnja funkcija nije kreirana konstruktorom
function spoljnaFunkcija() {
  let vrednost = "neki tekst";

  let func = function () {console.log(vrednost)};
  return func;
}

spoljnaFunkcija ()(); // neki tekst
```

5.2.6. Funkcije višeg reda i kompozicija funkcija

U matematici i računarstvu, ***funkcija višeg reda*** je funkcija koja je u stanju da uradi najmanje jedno od sledećeg:

- primi jednu ili više funkcija kao argumente,
- vrati funkciju kao rezultat.

Sve ostale funkcije su funkcije prvog reda. U matematici funkcije višeg reda se nazivaju i operatori ili funkcionali. Diferencijalni operator (izvod) je uobičajen primer, on preslikava funkciju na njen izvod koji je takođe funkcija.

Kompozicija funkcija je proces prosleđivanja povratne vrednosti jedne funkcije na ulaz drugoj funkciji. U matematičkoj notaciji (a i u jeziku Haskel) to se piše ovako:

$$f \circ g$$

a semantika (i prevod u JavaScript) je sledeća:

$$f(g(x))$$

Evaluacija se vrši "iznutra":

1. Evaluira se `x`.
2. `g()` se primenjuje na `x`.
3. `f()` se primenjuje na povratnu vrednost funkcije `g(x)`.

Primer:

```
const f = n => n + 1;
const g = n => n*2;
f(g(2)); // 5
```

Ovde imamo kompoziciju dve funkcije: funkcije `f()` i funkcije `g()`. Vrednost 2 se prosleđuje funkciji `g()` koja vraća izlaz 4. Zatim se vrednost 4 prosleđuje funkciji `f()` koja vraća 5.

Funkciji se može proslediti proizvoljan izraz kao argument. Prosleđeni izraz se evaluira pre nego što se primeni funkcija. Na primer:

```
f(g(2+1) * g(2)); // vratiće: 6*4+1 = 25
```

Šta se tu dešava?

1. Prvo se evaluira izraz 2+1 na vrednost 3
2. Zatim se na vrednost 3 primenjuje funkcija `double()` koja vraća vrednost 6
3. Na vrednost 2 se primenjuje funkcija `double()` koja vraća vrednost 4.
4. Zatim se evaluira izraz 6*4 na vrednost 24.
5. Konačno, funkcija `inc()` se primenjuje na vrednost 24 i ta funkcija vraća vrednost 24+1=25

Ako se malo bolje pogledaju ovi primeri, nije teško zaključiti da za kompozicija funkcija programski jezik treba da podrži mogućnost da funkcija primi kao argument drugu funkciju, odnosno **da podrži koncept funkcije višeg reda**.

U stvari, mehanizam komponovanja funkcija u JavaScript-u je isti kao u matematici pa možemo da napravimo funkciju koja će da implementira kompoziciju funkcija.

```
const komponovanaFunkcija = (f,g,x) => f(g(x));
const f = n => n + 1;
const g = n => n*2;

// poziv

rezultatKompozicije = komponovanaFunkcija(f, g, 2) // 5
```

Na kraju jedno **vrlo važno podsećanje**: Baš kao i u matematici, i u programiranju važi pravilo: pri kompoziciji funkcija **izlaz prve funkcije koja se primenjuje mora da odgovara ulazu druge funkcije koja se primenjuje**. Sledeći jednostavan primer to ilustruje (u njemu funkcija `g` vraća objekat, dok funkcija `f` očekuje primitivnu numeričku vrednost):

```
const komponovanaFunkcija = (f,g,x) => f(g(x));
const f = n => n + 1;
const g = n => {n*2};
rezultatKompozicije = komponovanaFunkcija(f, g, 2) // NaN
```

5.2.7. Povratni poziv

U kompjuterskom programiranju, funkcija povratnog poziva je svaka referenca na izvršni kod koja se prosleđuje kao argument drugom delu koda (pozivaocu) od koga se očekuje da pozove (izvrši) funkciju povratnog poziva kao deo svog posla.

Izvršenje može biti trenutno (sinhroni/blokirajući povratni poziv), ili se može desiti vremenski odloženo (asinhroni/neblokirajući povratni poziv).

Pri sinhronom pozivu, funkcija povratnog poziva se poziva pre no što pozivalac završi svoje izvršavanje i pozivaocu se zaustavlja izvršavanje do završetka izvršavanja prosleđene funkcije povratnog poziva.

Pri asinhronom povratnom pozivu funkcija povratnog poziva se može pozvati i nakon što pozivalac završi svoje izvršavanje - pozivalac nastavlja sa radom , ne zaustavlja mu se izvršavanje.

U JavaScript-u, funkcija povratnog poziva je funkcija koja se prosleđuje drugoj funkciji kao argument, koja se zatim poziva unutar te druge funkcije da bi se završila neka vrsta rutine ili radnje.

Potrošač API-ja zasnovanog na povratnom pozivu piše funkciju koja se prosleđuje u API. Dobavljač API-ja (pozivalac) preuzima funkciju i u nekom trenutku vraća (ili izvršava) funkciju unutar tela pozivaoca. Pozivalac je odgovoran za prosleđivanje ispravnih parametara u funkciju povratnog poziva. Pozivalac takođe može da očekuje određenu povratnu vrednost od funkcije povratnog poziva i da tu povratnu vrednost koristi u svom daljem ponašanju.

JavaScript je jezik koji je fundamentalno oslonjen na mehanizam povratnog poziva i ima razvijenu podršku za povratne pozive. Kako je nama cilj da objasnimo suštinu mehanizma, ovde ćemo da navedemo jednostavan, čak primitivan primer povratnog poziva sa jedinim ciljem da ilustrujemo princip. Sledi primer koda:

```
function sracunaj(num1, num2, callbackFunction) { // funkcija pozivalac
    return callbackFunction(num1, num2);
}

function sracunajProizvod(num1, num2) { // funkcija povratnog poziva
    return num1 * num2;
}

function sracunajZbir(num1, num2) { // funkcija povratnog poziva
    return num1 + num2;
}

console.log(sracunaj(10, 15, sracunajProizvod)); // 150
console.log(sracunaj(5, 15, sracunajZbir)); // 20
```

U ovom primeru se prvo se definiše funkcija `sracunaj()` koja je ovde pozivalac i ima parametar `callbackFunction` kojim se prosleđuje funkcija za povratni poziv. Zatim se definišu funkcije povratnog poziva `sracunajProizvod()` i `sracunajZbir()`. U ovom primeru, `sracunaj()` se poziva dva puta, jednom sa `sracunajProizvod()` kao povratnim pozivom i jednom sa `sracunajZbir()` kao povratnim pozivom. Funkcije povratnog poziva vraćaju proizvod i zbir. U ovom primitivnom primeru, upotreba povratnog poziva je prvenstveno demonstracija principa. Potpuno isti rezultat se postiže ako se jednostavno pozovu povratni pozivi kao regularne funkcije, na primer `sracunajZbir(num1, num2)`.

I sledeći primer ilustruje kako se prave i kako se koriste povratni pozivi:

```
function sracunajSumuNakon2Sec(a, b, callback) {
/* setTimeout je JS funkcija koja poziva unutrašnju funkciju nakon zadatog
   vremena */
/* Sledeći poziv će kreirati odlaganje od 2000 ms i nakon toga će da pozove
   callback sa argumentom rezultat. Na kraju će da prosledi pozivajući
   anonimnu funkciju sa argumentom rezultat */

    setTimeout(function () {
        let rezultat = a + b;
        callback(rezultat)
    }, 2000)
}

/* Jedan način da se koristi funkcija sracunajSumuNakon2Sec je prosleđivanje
   anonimne funkcije kao parametra callback funkcije sracunajSumuNakon2Sec
   (boldovan deo koda)*/
```

```
sracunajSumuNakon2Sec (2, 3, function (rezultat) {
    console.log(„Iz anonimne funkcije suma je „ + rezultat)
})

/* Drugi način je da se odvojeno definiše funkcija koja će biti povratno
pozvana i da se ona prosledi kao callback parametar */

// Posebno definisana funkcija
function print(a){
    console.log(„Iz funkcije sa imenom print suma je „ + a);
}
// poziv funkcije sracunajSumuNakon2Sec

sracunajSumuNakon2Sec (2, 3, print);
```

Povratni pozivi se generalno koriste kada pozivalac treba da izvrši događaje pre nego što se povratni poziv završi, ili kada funkcija nema (ili ne može) da ima relevantne povratne vrednosti na koje bi delovala, kao što je slučaj sa asinhronim JavaScript funkcionalnostima (zasnovanim na tajmerima) ili XMLHttpRequest zahtevima . Korisni primeri se mogu naći u JavaScript bibliotekama kao što je jQuery gde metoda `.each()` iterira nad nizolikim objektom, a prvi argument je povratni poziv koji se izvršava pri svakoj iteraciji.

5.2.8. Funkcije i objekti

Metod je funkcija pridružena objektu, obično kao svojstvo objekta:

```
const mojNiz = [1, 2, 3];
mojNiz.map(double); // Vratiće:[2, 4, 6]
```

Ovaj primer pokazuje da, baš kao i običan objekat, i tip podataka `Array` može da ima pridružene metode. U ovom slučaju, `mojNiz` je objekat a `map()` je svojstvo objekta koje za vrednost ima funkciju.

Kada se pozove, ova funkcija se primenjuje na argumente i na specijalni prametar koji se zove `this` koji se automatski postavlja kada se metod pozove. Parametar `this` je pokazivač na tekući objekat, u ovom slučaju mehanizam kojim `map()`dobija pristup sadržaju niza. Zapazite da se funkcija `double` ne poziva, već se prosleđuje kao argument metodi `map()`. U stvari, `map()` prima funkciju (u primeru funkciju `double()`) kao argument i primenjuje je na svaki član niza. Kao rezultat vraća novi niz sa modifikovanim vrednostima koje je vratila funkcija `double()`. Originalni niz ostaje nepromenjen što se vidi iz sledećeg primera:

```
const arr = [1, 2, 3];
console.log(' Ulazni niz pre modifikacije:' + arr)   // [1, 2, 3]
console.log (' Modifikovani niz:' + arr.map(double); // [2, 4, 6]
console.log(' Ulazni niz nakon modifikacije:' + arr) // [1, 2, 3]
```

U drugom primeru ilustrujemo korišćenje metode `filter()`. Metoda `filter()` koristi `predicate` funkciju koja vraća logičke vrednosti `true` ili `false`. Metoda `filter()` prima funkciju `predicate` kao argument i vraća novu listu koja sadrži samo one elemente originalne liste za koje funkcija `predicate` vrati vrednost `true`. U sledećem kodu

```
const gte = granica => n => n >= granica;
[1, 2, 3, 4, 5, 6].filter(gte(4)); // [4, 5, 6]
```

iz niza `[1, 2, 3, 4, 5, 6]` izdvajaju se vrednosti koje su veće od 4. Funkcija predicate ovde je funkcija koju vraća funkcija `gte()` pozvana sa vrednošću argumenta 4 . Toj vraćenoj funkciji se kroz

parametar n prosleđuju vrednosti elemenata ulaznog niza/liste `[1, 2, 3, 4, 5, 6]`. U vraćenu listu se prepisuju svi elementi ulazne liste koji su veći ili jednaki 4. Dakle, vraćena lista je [4, 5, 6].

5.2.8.1. Ulančavanje metoda

Funkcije se mogu komponovati, pa se i metode mogu ulančavati. **Ulančavanje metoda** je proces direktnog pozivanja metode nad povratnom vrednošću funkcije, bez potrebe da se na povratnu vrednost referiše njenim imenom kao u sledećem primeru:

```
const mojNiz = [1, 2, 3];
mojNiz.map(double).map(double); // Vraća: [4, 8, 12]
```

U ovom primeru se ulančavaju dva poziva metode `map()`. Rezultat je da se funkcija `double()` prvo primenjuje na originalne vrednosti niza `mojNiz` a zatim na vrednosti dobijene prvom primenom funkcije `double()`, odnosno vrednosti niza `mojNiz` se učetvorostručuju.

Česte su situacije kada je potrebno selektovati stavke iz liste, te selektovane stavke modifikovati na definisani način, i smestiti ih u novu listu. To se može ostvariti ulančavanjem metoda `filter()` i `map()` gde se metodi `filter()` prosleđuje funkcija koja opisuje kriterijum selekcije a metodi `map()` prosleđuje se funkcija koja opisuje način modifikacije selektovanih stavki.

Sledi primer u kome su ulančane metode `filter()` i `map()` tako da vrate listu koja u sebi sadrži sve elemente koji su veći od 4 pomnožene a 2:

```
const gte = cutoff => n => n >= cutoff;
[2, 4, 6].filter(gte(4)).map(double); // Vraća: [8, 12]
```

5.2.8.2. `Function()` konstruktor

Konstruktor `Function()` kreira objekte tipa `function`. Direktno pozivanje konstruktora može da kreira funkcije dinamički, ali pati od bezbednosnih problema i drugih daleko manje značajnih problema sa performansom poput funkcije `eval()`[24]. Za razliku od funkcije `eval()` koji može imati pristup lokalnom dosegu, konstruktor funkcije kreira funkcije koje se izvršavaju samo u globalnom dosegu.

Sintaksa za kreiranje funkcije putem konstruktora je sledeća:

```
let func = new Function ([arg1, arg2, ...argN], teloFunkcije);
```

Ovde su `new` i `Function` obavezne ključne reči, `arg1` … `argN` su argumenti funkcije, a `teloFunkcije` je string koji sadrži kod funkcije.

Osnovna razlika između konstruktora i ostalih načina za kreiranje funkcije je što se ovde funkcija kreira doslovno iz stringa koji se prosleđuje u vreme izvršavanja koda. To znači da se funkcija može kreirati dinamički, u toku izvršavanja programa. Sledeće linije koda to ilustruju:

```
let suma = new Function('a', 'b', 'return a + b');
console.log( suma(1, 2) ); // 3
```

Ovaj način kreiranja funkcije koristi se u vrlo specifičnim slučajevima kao što je primanje koda sa servera ili dinamičko kompiliranje funkcije iz obrasca u složenim veb aplikacijama.

U nastavku ćemo detaljno objasniti mehanizam kreiranja funkcije putem konstruktora.

5.2.8.2.1. Mehanizam kreiranja funkcije putem konstruktora

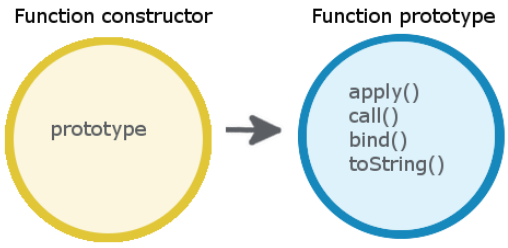
Pošto je funkcija u JavaScriptu specijalna vrsta objekta, svaka funkcija ima svojstvo `prototype` povezano sa internim objektom prototipa. Kada se funkcija pozove kao konstruktor za kreiranje

[24] Funkcija `eval()` evaluira JavaScript kod predstavljen kao string i vraća rezultat njegovog izvršavanja.

instanci objekata, ti objekti instance imaju vezu na isti prototipski objekat koji im daje zajedničko ponašanje – sve instance dele ponašanje tog prototipa. Ključ za razumevanje konstruktora funkcija je u odnosu između sledećih objekata:

- `Function()` objekat,
- Prototipski objekat objekta `Function()`,
- Objekti instance funkcije.

Ugrađeni objekat `Function` konstruktora funkcije je sam po sebi funkcija i stoga ima svojstvo `prototype` koje upućuje na njegov prototipski objekat putem `Function.prototype`. Ova dva objekta su ugrađena u JavaScript izvršno okruženje pre nego što bilo koji kod počne da se izvršava. Vizuelna ilustracija data je na slici 6.2.



Slika 6.2 Konstruktorski i prototipski objekat tipa `Function`

Videćemo sada detaljnije kako se kreiraju instance funkcija. Neka imamo dve definicije funkcija:

```
function Alpha() { ... }
function Bravo() { ... }
```

Važna napomena: U JS-u imena funkcija u definiciji za funkcije konstruktore **moraju da počinju velikim slovom**.

Kada su definisane ove dve funkcije dešava se sledeće:

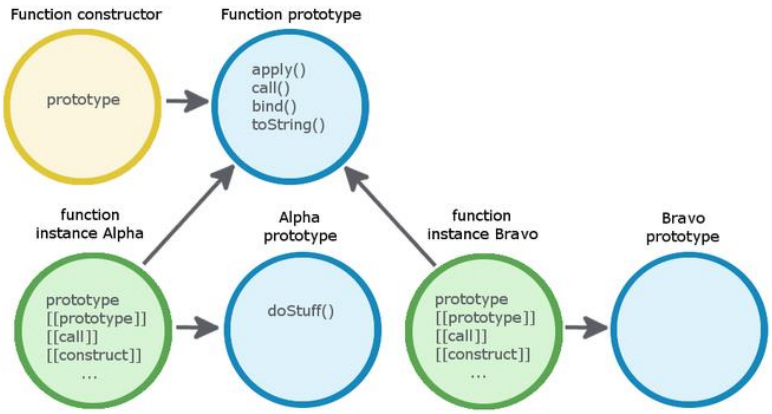
1. Ključna reč `function` kreira novi objekat i prosleđuje ga u konstruktor `Function()`.
2. Novom objektu se dodeljuju interna svojstva među kojima i svojstvo `[[prototype]]`. Svojstvo `prototype` konstruktora `Function` koje referencira prototipski objekat konstruktora `Function` kopira se u svojstvo `[[prototype]]` tog novog objekta. Dakle, konstruktor i novi objekat pokazuju na isti prototipski objekat.
3. Novi objekat je takođe funkcija pa i on ima svojstvo `prototype` koje pokazuje na prototipski objekat koji se takođe automatski kreira čime je omogućeno da se i on koristi kao konstruktor.
4. Novi objekat se vraća i dodeljuje se svojoj referenci (imenu).

Funkcije se mogu proširivati dodavanjem novih svojstava, odnosno novih metoda njihovom prototipu tako što će se referencirati `prototype` svojstvo konstruktorske funkcije kao u sledećem primeru:

```
Alpha.prototype.doStuff = function() { ... };
```

Sada instanca `Alpha` i sve instance kreirane konstruktorom `Alpha` imaju pristup deljenim metodama prototipa `Alpha`.

Situacija nakon definisanja dva konstruktora `Alpha()` i `Bravo()` i proširivanja prototipa konstruktora `Alpha()` prikazana je na slici 6.3.



Slika 6.3 Konstruktorski i prototipski objekti funkcija `Alpha()` i `Bravo()`

Rezultat su dve instance objekata tipa `function` koji se referenciraju kao `Alpha` i `Bravo`. Oni su povezani na isti `Function.prototype` putem svog `[[prototype]]` svojstva što im daje pristup metodama prototipa kao što su `.call()`, `.bind()`, `.apply()`, itd.

5.2.8.2.2. Ključna reč **new**

Pozivom konstruktorske funkcije kreira se instanca funkcije. U našem primeru to izgleda ovako. Poziva se funkcijski objekat (recimo `Alpha`) sa operatorom (ključnom reči) new što objekat `Alpha` tretira kao konstruktorsku funkciju.

```
var alpha1 = new Alpha();
var alpha2 = new Alpha();
```

Prethodne dve linije koda rezultuju kreiranjem instanci funkcija (objekata) `alpha1` i `alpha2` u procesu koji se sastoji u sledećem.

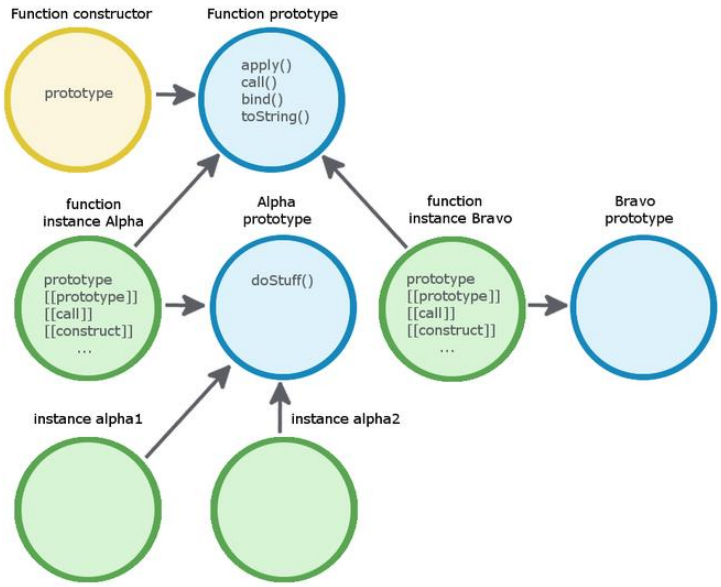
Kreira se nova prazna instanca objekta i prosleđuje se konstruktoru `Alpha` kao `this` pokazivač.

Novoj instanci objekta se implicitno (bez intervencije programera) dodeljuje više internih svojstava zajedničkih svim objektima. Među tim svojstvima je i svojstvo `[[prototype]]`. Prototipsko svojstvo konstruktora, koje pokazuje na prototipski objekat konstruktora, kopara se u svojstvo `[[prototype]]` novog objekta.

Vraća se novi objekat i dodeljuje se svojim referencama (identifikatorima) `alpha1` i `alpha2`.

Ukratko, iskorišćena je ključna reč new za pozivanje `Alpha` kao konstruktorske funkcije koja kreira dve instance objekta `alpha1` i `alpha2`. Ključna razlika u odnosu na situaciju kada je kreiran objekat `Alpha` je što objekti alpha1 i alpha2 nisu prototipovi.

Vizualizacija ove situacije prikazana je na slici 6.4.



Slika 6.4 Prototipski lanac funkcija

5.2.8.2.3. Svojstva objekata instanci funkcije

Kao što smo videli, instanca funkcije je posebna vrsta objekta. Kako je u pitanju posebna vrsta objekta, ona ima specifična svojstva. U Tabeli 6.2 su sistematizovana svojstva objekata instanci funkcije.

Tabela 6.2 Svojstva objekta instance funkcije

Svojstvo	Opis svojstva
`prototype`	Automatski se kreira za svaku funkciju čime se obezbeđuje da funkcija može da se koristi kao konstruktor.
`[[Code]]`	Kod tela funkcije - ono što se nalazi između zagrada `{}`
`[[FormalParameters]]`	Parametri funkcije
`[[BoundThis]]`	`this` parametar koji referencira pozivajući objekat
`[[Call]]`	Ovo svojstvo imaju samo funkcije, jer su funkcije jedini objekti koji se mogu pozivati. Poziva se putem `()` da se izvrši kod tela funkcije.
`[[Scope]]`	Lanac dosezanja. Koristi se za zatvaranje.
`[[Class]]`	Identifikuje interni tip objekta. Ima vrednost "Function"
`[[Construct]]`	Kreira objekat. Implementiran u funkciji koja se može pozvati operatorom new (sve korisničke funkcije i mnoge ugrađene funkcije).

5.2.9. Signatura funkcije

Signatura funkcije je neka vrsta sažetka o funkciji. Signatura služi da se sažeto iskažu opšte informacije o funkciji kao što su ime, doseg i parametri. Ne sadrži nikakav opis onoga što funkcija radi.

Veoma je korisna ljudima za lakše čitanje koda, ali njena glavna uloga je u programiranju jer smanjuje potrebu eksplicitnog deklarisanja tipa i omogućuje dinamičko (u fazi izvršavanja programa) zaključivanje tipova i druge manipulacije kodom. Ovo je od posebnog značaja za JavaScript koji slabo tipiziran jezik sa vrlo redukovanim mogućnostima eksplicitnog deklarisanja u izvornom kodu. Očigledan primer je preopterećenje funkcija gde se funkciji sa istim imenom mogu pridružiti različita značenja u zavisnosti od parametara. Na primer, operacija sabiranja (**+**) koja u slučaju bojevnih operanada vrši aritmetičko sabiranje i vraća tip `number`, a u slučaju operanada `string` tipa vrši

konkatenaciju stringova i vraća tip `string` pa se signatura koristi da se, u toku izvršavanja programa, na osnovu tipa izlaza odabere ono što će funkcija da uradi (numeričko sabiranje ili konkatenaciju).

Mi ćemo se ovde ograničiti samo na sintaksu signature funkcije i nećemo zalaziti ni u teoriju signature tipa, niti dublje u njenu primenu u programiranju. Prosto, cilj nam je da naučimo da "čitamo" signature, odnosno da naučimo da iz signatura razumemo koliko god možemo o funkciji koju signatura opisuje.

Uobičajeno, signatura funkcije se sastoji od:

1. Imena funkcije.
2. Liste tipova parametara. Parametri mogu a ne moraju da budu imenovani i mogu a ne moraju da budu zatvoreni u male zagrade.
3. Tipa povratne vrednosti.

U JavaScript-u nije obavezna specifikacija signatura tipa, pa ni signature funkcije. JavaScript endžin određuje tipove u fazi izvršavanja programa. Ako se obezbedi dovoljno informacija, analizom toka podataka signaturu mogu da zaključe razvojni alati kao što su integrisana razvojna okruženja i Tern.js[25] analizator koda.

Shodno činjenici da ne nameće obavezu signature tipa, JavaScript nema ni sopstvenu notaciju signatura funkcije, ali ima nekoliko "standarda" koji su u opticaju: JSDoc[26] koji je istorijski značajan ali se ne koristi mnogo, TypeScript i Flow kao najveći konkurenti trenutno. Ima i drugih poput Rtype[27] koji preporučuje Erik Eliot za dokumentovanje. Konačno, to je i sistem koji koristi jezik Haskell, kurirani Hindley-Milner tipovi[28]. Prosto, nema standardizovane notacije za JavaScript pa se koriste različite stvari. I mi ćemo tako da radimo – radi ilustracije pokazaćemo Rtype signaturu, ali ćemo pretežno da koristimo Hindley-Milner signaturu.

5.2.9.1. Rtype signatura tipa

Iako JavaScript ne zahteva anotaciju signatura, znajući šta signature ***jesu*** i šta one ***znače***, jasno je da su važne za efikasnu komunikaciju o načinu korišćenja i načinu kompozicije funkcija. Većina ponovno upotrebljivih pomoćnih programa za kompoziciju funkcija zahteva da im se proslede funkcije koje imaju zadatu signaturu tipa. Dakle, potreban nam je način da izrazimo te signature. **Rtype** signature su smišljene kao mešavina (najboljih?) mogućnosti Haskell i TypeScript notacije tipova sa idejom da se pokrije ekspresivna snaga samog jezika JavaScript jer, po oceni Erika Eliota, ni Haskell ni TypeScript notacija to ne uspevaju. Rtype signature ugrubo izgledaju ovako:

```
functionName(param1: Type, param2: Type) => ReturnType
```

Signatura započinje imenom funkcije `functionName`, iza koga sledi otvorena mala zagrada, iza otvorene zagrade se navode opciona imena parametara sa dvotačkom na kraju (na primer, `param1:`), iza čega sledi tip (`Type`). Tip može da bude tip varijable (po konvenciji jedno malo slovo) ili konkretan tip (potpuna reč koja počinje velikim slovom). Tip povratne vrednosti se navodi iza debele strelice (**`=>`**) koja ukazuje na njega (`ReturnType`). `ReturnType` može, kao i parametarski tip, da bude par `name: Type`.

Na primer, signatura za funkciju `double` je:

```
double(x: Number) => Number
```

[25] http://ternjs.net/

[26] **JSDoc** je markerskijezik za anotiranje JavaScript izvornog koda.

[27] https://github.com/ericelliott/rtype

[28] https://en.wikipedia.org/wiki/Hindley%E2%80%93Milner_type_system
http://web.cs.wpi.edu/~cs4536/c12/milner-damas_principal_types.pdf

Ova signatura kaže sledeće: funkcija `double` prima kao ulaz jednu vrednost tipa `Number` i vraća vrednost tipa `Number`.

Šta biva ukoliko se želi da ulazni argumenti mogu da budu i objekti? Evo primera:

```
const one = {
   name: 'One',
   valueOf: () => 1,
};

double(one); // Vratiće 2
```

U tom slučaju treba promeniti signaturu tako da koristi varijablu za označavanje tipa, na primer **n**:

```
double(x: n) => Number
```

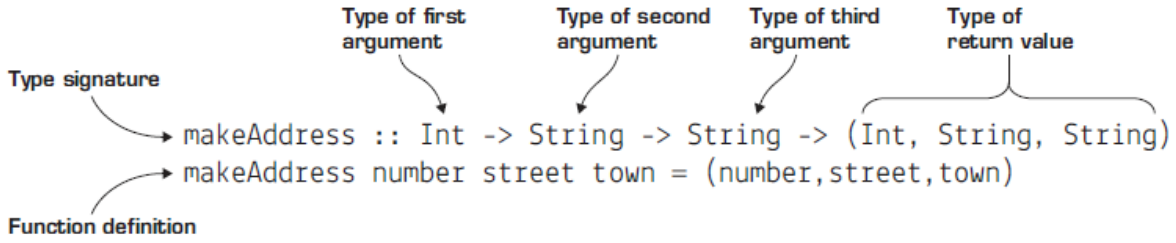
Ako ime parametra nije važno, može se izostaviti:

```
double(n) => Number
```

5.2.9.2. Hindley-Milner signatura tipa

Sistem tipa Hindley–Milner (HM) je klasični sistem tipa za Lambda račun sa parametarskim polimorfizmom[29]. Među značajnijim osobinama HM-a su njegova kompletnost i sposobnost da zaključi najopštiji tip datog programa, bez napomena o tipu koje daje programer ili drugih nagoveštaja. Kao metod za zaključivanje tipa, Hindley-Milner je u stanju da izvede tipove promenljivih, izraza i funkcija iz programa napisanih potpuno netipiziranim stilom. HM sistemu tipa se daje prednost u jezicima funkcionalnog programiranja. Implementiran je u programskom jeziku Haskell koji je referentni jezik funkcionalnog programiranja pri čemu se smatra da mu je baš sistem tipiziranja poseban kvalitet.

Nas prvenstveno interesuju signature funkcija i to funkcija više promenljivih pa da vidimo kako to izgleda kod HM signature. Pokazaćemo to na jednostavnom primeru „sklapanja adrese“:



Slika 6.1 HM signatura funkcije više promenljivih [Will Kurt, Get Programming with HASKELL]

Ako prevedemo definiciju funkcije u JavaScript, možemo je zapisati na tri načina (streličasta sintaksa, funkcijski izraz ili samostalna funkcija):

```
let makeAddress = (number, street, town) => [number, street, town]
let makeAddress = function (number, street, town) { return [number, street,
town]}
function makeAddress(number, street, town) { return [number, street, town]}
```

Ako, pak, pogledamo signaturu sa slike 6.1, videćemo da se ona „uklapa“ sa svakom od korišćenih sintaksi i da je to uvek ista stvar:

`makeAddress`	ime funkcije, odnosno signatura tipa
`number, street i town`	ulazni parametri (tipovi)

[29] U programskim jezicima i teoriji tipova, parametarski polimorfizam omogućava da se jednom delu koda pridruži „generički“ tip koristeći promenljive umesto stvarnih tipova, a zatim da se instancira sa određenim tipovima po potrebi.

`[number, street, town]` tip izlaza koji je niz

HM signatura ove funkcije „upodobljena“ sintaksi JavaScript-a mogla bi da izgleda ovako:

`makeAddress :: (Number, String, String) → [Number, String, String]`

Šta nam, dakle, o funkciji kaže njena signatura, a šta nam ne kaže?

Kaže nam kako se funkcija identifikuje, kakvi su joj ulazi i kakav joj je izlaz.

Ne kaže nam ništa o načinu na koji funkcija transformiše ulaze u izlaz. Ali to nas to i ne interesuje kada gledamo program kao celinu. Ako nas, pak, to interesuje, treba da pročitamo telo funkcije. A telo je jedino što je još ostalo da se kaže o funkciji – sve ostalo rečeno je u signaturi.

I to vam je suština signature funkcije. Ostalo su tehnički detalji koji nisu nevažni u praksi. Zato ćemo u nastavku i o njima da govorimo.

Čim hoćete nešto da izrazite, treba vam jezik. I HM signatura može se posmatrati kao jednostavan jezik koji na neki način izražava funkciju. U tom jeziku funkcije se izražavaju na sledeći način:

a → b

gde su a i b varijable proizvoljnog tipa. To znači da se signatura funkcije `makeAddress` može prirodnim jezikom predstaviti sledećom rečenicom: `makeAddress` *je funkcija koja kao ulaz prima tri argumenta i kao izlaz vraća niz.*

Evo još nekoliko primera [Lonsdorf] da ih pročitamo:

Prvi primer je sasvim jednostavan i lako se čita: **`strLength` je funkcija koja prima nešto tipa `String` i vraća nešto tipa `Number`:**

```
// strLength :: String -> Number
const strLength = s => s.length;
```

Drugi primer je malo komplikovaniji:

```
// join :: String -> [String] -> String
const join = curry((what, xs) => xs.join(what));
```

Ovde se u signaturi pojavljuju dve strelice pa nije na prvi pogled jasno kako sada razlikovati argumente od vraćene vrednosti. Ako se setimo da funkcija uvek vraća samo jednu povratnu vrednost[30] odgovor je jednostavan - *tip vraćene vrednosti je **uvek** ono što sledi iza poslednje strelice*. Dakle, u primeru funkcija `join` vraća vrednost tipa `String`. Jasno je da je `String` iza prve strelice tip prvog argumenta a kako postoji još i `[String]` iza druge strelice i iza njega nova strelica, znači da je to takođe tip argumenta. Dakle u signaturi funkcije `join` piše: **Join je funkcija koja prima dva argumenta od kojih je prvi nešto tipa `String` a drugi nešto tipa `[String]` i vraća nešto tipa `String`.**

Treći primer koji čitamo je:

```
// match :: Regex -> (String -> [String])
const match = curry((reg, s) => s.match(reg));
```

Ovde se u signaturi takođe pojavljuju dve strelice, ali se pojavljuje i grupisanje u malu zagradu koje dodatno opisuje kakva je, u stvari, funkcija `match`. Čitamo je na sledeći način: funkcija match je funkcija koja prima nešto tipa Regex i vraća nešto što prima nešto tipa String i vraća nešto tipa `[String]`. A šta je to nešto što može da primi neki tip ulaza i da vrati neki tip izlaza? Naravno – funkcija.

Dakle u signaturi funkcije mach piše**: `match` je funkcija koja prima nešto tipa `Regex` i vraća funkciju koja prima nešto tipa String i vraća nešto tipa `[String]`.**

[30] Ovo se može obrazložiti i kuriranjem o kome ćemo posebno govoriti.

Četvrti primer signature koju čitamo je:

```
// replace :: Regex -> String -> String -> String
const replace = curry((reg, sub, s) => s.replace(reg, sub));
```

Ovako kako je napisan znači sledeće: **`replace` je funkcija koja prima tri argumenta i vraća nešto tipa `String`. Pri tome, prvi argument je nešto tipa `Regex` , a drugi i treći su nešto tipa `String`.**

Da se autor signature potrudio i dodao zagrade na sledeći način:

```
// replace :: Regex -> (String -> (String -> String))
```

mogli bismo da pročitamo sledeće: **`replace` je funkcija koja prima nešto tipa `Regex` i vraća funkciju** (spoljašnji par zagrada) **koja prima nešto tipa `String` i vraća funkciju** (unutrašnji par zagrada) **koja prima nešto tipa `String` i vraća nešto tipa `String`.**

Ima ljudi koji smatraju da velika količina zagrada u signaturi zahteva dodatnu koncentraciju pri čitanju pa je pitanje mere njihovog korišćenja opravdano. Međutim, nesporno je da one omogućuju jasnije i preciznije izražavanje.

Sledeća stvar koju ćemo da objasnimo je kako se korišćenjem HM signature može uopštiti izražavanje tipa u slučaju funkcija koje dozvoljavaju različite tipove (na primer dozvoljeni tip je string ali objekat). Taj problem HM signatura tipa razrešava tako što uvodi varijable za označavanje tipa. Imena ovih varijabli su konvencija a suština je da ime varijable označava tip, odnosno da **varijable sa istim imenom ne mogu da označavaju različite tipove** i da **varijable sa različitim imenima mogu a ne moraju da označavaju različite tipove**. Ilustracija su sledeći primeri.

Prvi primer je signatura funkcije identiteta koja prima ulaz nekog tipa i vraća taj isti ulaz, što znači i isti tip (ovde označen varijablom **`a`**):

```
// id :: a -> a
const id = x => x;
```

U drugom primeru je signatura funkcije **`map`** u kojoj se pojavljuju dva tipa, tip **`a`** i tip **`b`** koji mogu, a ne moraju da budu različiti tipovi:

```
// map :: (a -> b) -> [a] -> [b]
const map = curry((f, xs) => xs.map(f));
```

Konačno, pogledaćemo i HM signaturu funkcije **`reduce`** da pokušamo da objasnimo šta se može pročitati iz signature a šta, ipak, ostaje nedorečeno. Ovako izgledaju signatura i definicija funkcije reduce:

```
// reduce :: ((b, a) -> b) -> b -> [a] -> b
const reduce = curry((f, x, xs) => xs.reduce(f, x));
```

Gledajući signaturu, vidimo da funkcija prima tri argumenta. Prvi argument je funkcija koja očekuje tip **`b`** i tip **`a`** i vraća tip **`b`**. Drugi argument je tip **`b`** a treći je niz **`a`** tipova. Povratna vrednost je tip **`b`.** Toliko iz signature može da se pročita. Konačno „mađijanje“ o prosleđenoj funkciji biće na osnovu izlazne vrednosti. Znajući šta redukcija radi – svodi niz vrednosti na jednu vrednost (na primer sabiranje niza brojeva), gornje čitanje moglo bi se smatrati ispravnim.

Sposobnost rasuđivanja o tipovima i njihovim implikacijama je veoma važna za kvalitetno programiranje.

5.3. Sažetak

Ovde treba dodati sažetak poglavlja.

Literatura uz poglavlje 5

Poglavlje 6: ASINHRONO PROGRAMIRANJE I JavaScript

https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Introducing

Centralna tema ovog poglavlje je asinhrono programiranje u jeziku JavaScript. U njemu ćemo prvo objasniti pojmove sinhronost i asinhronost u programiranju, obrazložiti potrebu za asinhronim programiranjem, navesti glavne probleme koji se pri asinhronom programiranju javljaju i opisati podršku koju JavaScript pruža za asinhrono programiranje. Govorićemo o povratnim pozivima, generatorskim funkcijama (specijalna vrsta funkcija u JavaScriptu koje omogućuju vraćanje višestrukih povratnih vrednosti), o iteriranju i iteratorima (sinhronim i asinhronim). Na kraju ćemo objasniti mehanizam **`Promise`** koji JavaScript pruža za asinhrono programiranje.

6.1. Sinhronost i asinhronost u programiranju

Pri izvršavanju računarskog programa angažuju se različiti resursi računara: centralni procesor, memorija, periferijski uređaji. Vrlo grubo rečeno, najprimitivniji način rada bio bi da određeni program unapred "rezerviše" sve resurse koji su mu potrebni i da drugi programi nemaju pristup tim resursima sve dok se taj određeni program ne završi. Model sinhronog programiranja odgovara baš takvom načinu rada, u modelu sinhronog programiranja se stvari dešavaju jedna za drugom. Na primer, kada se pozove funkcija koja vrši dugotrajnu radnju, ona vraća rezultat tek kada je radnja završena a ostatak programa "čeka" dok se izvršavanje funkcije ne okonča.

Nasuprot sinhronom modelu, asinhroni model omogućava da se više stvari dešava istovremeno (ili da to bar tako izgleda korisniku). Kada se započne radnja koja dugo traje, program nastavlja da radi druge stvari koje ne zavise od rezultata te radnje.

Asinhroni model se na računaru implementira uvođenjem dodatnih niti[31] (*thread*) kontrole toka programa. Pošto većina ozbiljnijih savremenih računara sadrži više procesora, može više niti čak i da rad doslovno istovremeno, na različitim procesorima. U najjednostavnijem slučaju sa dve niti prva nit pokreće neki program u kome se u nekom trenutku zahteva pribavljanje nekog resursa sa mreže. Zahtev za pribavljanje resursa sa mreže bi kreirao i pokrenuo drugu nit, a zatim da obe niti čekaju da dođu traženi resursi, nakon čega se ponovo sinhronizovati da kombinuju svoje rezultate.

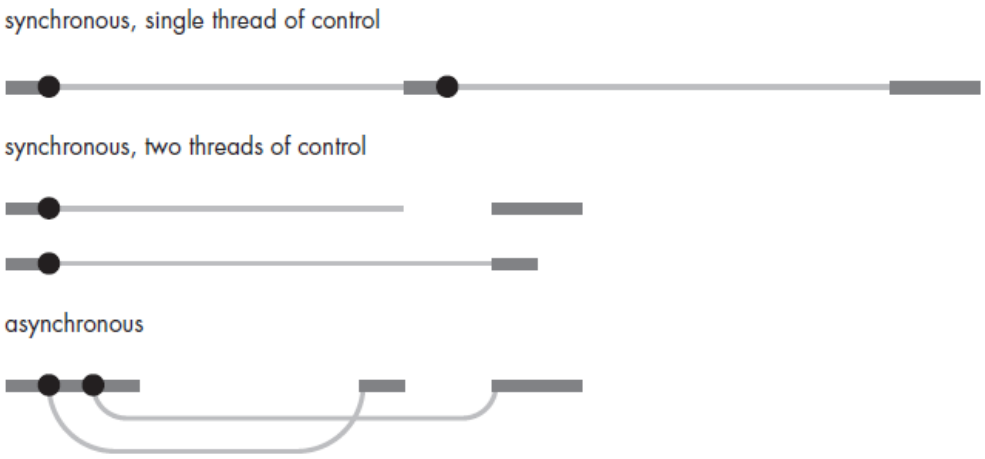
Na slici 7.1 [Haverbeke] koja ilustruje neke jednostavne situacije izvršavanja programa u sinhronom i asinhronom režimu, deblje linije predstavljaju vreme koje program provodi u izvršavanju, a tanje linije predstavljaju vreme provedeno u čekanju na mrežu.

U sinhronom modelu, vreme potrebno za mrežu je sastavni deo vremenske linije za datu nit kontrole.

U asinhronom model, pokretanje mrežne akcije konceptualno izaziva podelu vremenske linije. Program koji je pokrenuo akciju nastavlja da radi, a akcija se dešava pored njega i obaveštava program kada je završena.

Iz ove ilustracije lako se može videti prednost asinhronog modela. Kod **asinhronog modela**, vreme koje program provede u čekanju može, **u najgorem slučaju**, da bude **jednako čekanju niti sa dužim čekanjem** što je **najbolji slučaj** za **sinhroni model**. U realnosti, vreme čekanja za asinhroni model je, po pravilu, kraće od najgoreg slučaja.

[31] Da se podsetimo: Nit je pokrenuti program čije izvršavanje operativni sistem može da prepliće sa drugim programima



Slika 7.1 [Haverbeke] Jednostavne situacije izvršavanja programa u sinhronom i asinhronom režimu

Računari su "asinhrone mašine" koje nastoje da postignu visoku performansu (recimo, brzinu rada) tako što optimalno koriste svoje resurse izbegavajući "prazan hod" koliko god je moguće. To je osnovni princip operativnih sistema računara koji upravljaju resursima, kako u slučaju pojedinačnog računara tako i u slučaju povezanih računara (recimo, u klaudu).

Sa druge strane, ljudi su po prirodi "sinhrone mašine": rade stvari jednu za drugom i teško im je da do detalja sagledaju kako treba da izgleda asinhrona mašina (računarski program) koja će implementirati paralelno[32] ili konkurentno[33] obavljanje operacija sa ciljem podizanja performansi programa za koji pišu kod. Poseban problem za ljude je sinhronizovanje procesa koje može da bude veoma komplikovano i podložno greškama.

Zbog toga se programski jezici prave tako da obezbede jezičke konstrukte koji će da omoguće programerima da pišu izvorni kod u kome će stvari "izgledati sekvencijalno" što je više moguće, a da o asinhronosti računa vode ti namenski konstrukti.

U nastavku ovoga poglavlja videćemo šta od tih konstrukata obezbeđuje JavaScript i kako se oni koriste.

6.2. Asinhrono programiranje u JavaScriptu

Obe platforme za JavaScript programiranje, pretraživači i serverska okruženja, podržavaju asinhrone operacije ali se ne zahteva da se one implementiraju kao niti. Pošto je poznato da je programiranje sa nitima teško (razumevanje šta program radi je mnogo teže kada je raditi više stvari odjednom), ovaj pristup se generalno smatra dobrim.

6.3. Povratni pozivi

Mehanizam povratnog poziva koji smo već pomenuli i objasnili ga u osnovnim crtama je jedan od osnovnih koncepata na koji se oslanja JavaScript endžin. U stvari, JavaScript kod se izvršava u petlji događaja, na jednoj niti. Realnost je da se sav JavaScript izvršava sinhrono — to je petlja događaja koja omogućava da se u red čekanja postavi funkcija koja se neće izvršiti sve dok petlja ne bude dostupna, ponekad i nakon što je kod koji je funkciju postavio u red završio sa izvršavanjem.

[32] Paralelno računanje je tip računanja u kome se više kalkulacija ili procesa odvijaju istovremeno.

[33] Konkurentno računanje je tip računanja u kome se više računanja odvija konkurentno tokom vremenskih perioda koji se preklapaju, a ne sekvencijalno čekajući da se jedno računanje završi pre no što drugo može da započne.

Da pogledamo prvo primer pribavljanja podataka sa mreže koji će nam poslužiti da razjasnimo sinhroni i asinhroni povratni poziv.

Pseudokod koji ilustruje sinhrono pribavljanje mogao bi da izgleda ovako:

```
response = request.get 'http://www.google.co.in'
print response
```

Tok izvršavanja ovog koda je:

- Izvršava se metoda `get` i nit izvršavanja čeka dok se ne dobije odgovor.
- Google vraća odgovor pozivaocu koji ga skladišti u neku varijablu.
- Vrednost uskladištena u varijablu (ovde se varijabla zove `response`) se ispisuje na konzolu.

Sledeći snipet je asinhrono izvršavanje istog zadatka u javaScript-u:

```
request('http://www.google.co.in', function(error, response, body) {
  console.log(body);
});
console.log('Gotovo!');
```

Ovde kontrola toka izgleda drugačije:

- Izvršava se funkcija `request` koja prosleđuje anonimnu funkciju (drugi argument u pozivu funkcije `request`) da se izvrši u budućnosti kao povratni poziv kada funkcija request dobije traženi odgovor.
- Dok poziv funkcije `request` čeka željeni odgovor, nastavlja se rad programa i odmah se na konzoli ispisuje poruka "Gotovo!" (linija `console.log('Gotovo!');`).
- U nekom budućem trenutku kada funkcija `request` dobije traženi odgovor, izvršava se povratni poziv funkcije `console.log(body)`.

JavaScript je, dakle, jednonitni konkurentni jezik. Ima jedan stek poziva što znači da u jednom trenutku može da se izvršava samo jedna stvar. Kad god se neki događaj pokrene, on se stavlja u red poruka. Petlja događaja stalno proverava da li postoji događaj prisutan u redu čekanja, obrađuje ga i uklanja događaj iz reda. Dakle, sve dok postoje događaji u redu čekanja, petlja događaja je aktivna.

Šta se dešava kada već postoji nekoliko događaja u redu čekanja? Događaj se postavlja u red i čeka na izvršavanje. Ali kako deo koda zna da je događaj obrađen u određenom trenutku? Sada dolazi koncept povratnih poziva. U stvari, u red za poruke ne stavlja se događaj, nego funkcija koja se poziva kada se događaj obradi. Ova funkcija se zove **povratni poziv**.

Da vidimo sada kako se kreiraju funkcije povratnog poziva. Za to će nam poslužiti sledeće parče koda.

```
function pribaviResurse(callback){
//
//    ... ovde se dešava pribavljanje resursa
//

//kada su konačno pristigli resursi, izvršavamo povratni poziv funkcije koja
nam je prosleđena putem argumenta "callback" i prosleđujemo pribavljene
resurse putem argumenta "resources" funkcije povratnog poziva
      callback("resources")
}

// Kako se koristi funkcija pribaviResurse:
// Jednostavno prosleđujemo anonimnu funkciju koju će funkcija
// pribaviResurse da pozove nakon što pribavi resurse. U primeru je to
// funkcija
// function(podaci){
```

```
//    console.log(podaci);
// }

pribaviResurse (function(podaci){
      console.log(podaci);
})
```

Ovi primeri su nam pokazali da je mehanizam povratnog poziva je po svojoj prirodi upotrebljiv za implementaciju asinhronog stila programiranja jer omogućuje da se prosleđena funkcija pozove na asinhroni način – u nekom trenutku nakon što se asinhrona operacija koja je u toku završi.

Problemi asinhronog programiranja koje je direktno oslonjeno na povratni poziv nastaju kada kod ne radi pojedinačan zadatak, nego je u pitanju situacija u kojoj se upućuju višestruki pozivi pri čemu svaki od njih zavisi od prethodnog poziva. U suštini dolazi se u situaciju da se radi sa ugnježdenim funkcijama. Ta situacija se, sa vrlo dobrim razlogom, kolokvijalno naziva "pakao povratnog poziva" ili "piramida propasti".

U kompjuterskom programiranju, piramida propasti je uobičajen problem koji se javlja kada program koristi mnogo ugnježdenih nivoa da kontroliše pristup funkciji. Obično se javlja pri proveri postojanja nultih pokazivača ili obradi povratnih poziva. Vizualizuje se u izvornom kodu sa uvlačenjima kao piramida rotirana za 90° odakle potiče i ime. Na stranu estetika, ovakav stil programiranja rezultuje generalno nečitljivim kodom i izuzetno je podložan greškama.

Primeri koji slede ilustruju termin koji je nama interesantan a odnosi se na određeni stil programiranja u JavaScript-u.

Prvi primer ilustruje problem ugnježdenih `if` naredbi koje proizvode prilično nečitljiv kod:

```
function login(){
   if(korisnik == null){
     //Neki kod ovde
     if(korisnikIme != null){
       //pa neki kod ovde
          if(lozinkaDobra == true){
             //pa neki kod ovde
             if(returnedval != 'no_match'){
               // pa neki kod ovde
                if(returnedval != 'nekorektna lozinka'){
                  // pa neki kod ovde
                } else{
                  // pa neki kod ovde
                }
             } else {
                 // pa neki kod ovde
             }
           } else {
                // pa neki kod ovde
          }
       } else {
             // konačno neki kod ovde
       }
    }
}
```

Drugi primer ilustruje obradu povratnih poziva. U JavaScript-u koji koristi AJAX i druge asinhrone metode, sve metode primaju funkciju kao argument povratnog poziva. Uobičajen, ali ružan stil za ovo

kodiranje je korišćenje anonimnih funkcija (u primeru se funkciji `setTimeout` prosleđuje anonimna funkcija koja ispisuje string `"Hello World!"`):

```
setTimeout(function() {
  alert("Hello World!");
}, 1000);
```

Iako je brz i lak za implementaciju, ovaj stil rezultuje izuzetno nečitkim kodom koji vizuelno podseća na piramidu kao u sledećem primeru:

```
// Kod koristi jQuery za ilustrovanje “piramide propasti”
(function($) {
  $(function(){
    $("button").click(function(e) {
      $.get("/test.json", function(data, textStatus, jqXHR) {
        $(".list").each(function() {
          $(this).click(function(e) {
            setTimeout(function() {
              alert("Hello World!");
            }, 1000);
          });
        });
      });
    });
  });
})(jQuery);
```

U ovom primeru vrlo je teško pratiti šta kod stvarno radi i, posebno, šta jeste a šta nije u dosegu funkcije. Treba dobro da se skoncentrišete da shvatite da će na klik biti ispisivana lista nekih podataka i, nakon 1 sekunde čekanja, tekst `Hello World!`. Jedan način da se kod učini malo čitljivijim je da se koriste imenovane funkcije kao u sledećem primeru, ali je rezultat zaista skroman:

```
// Kod koristi jQuery ali su funkcije imenovane
(function($) {

  function init() {
    // Add onClick to buttons
    $("button").click(getData);
  }

  function getData() {
    $.get("/test.json", onSuccess);
  }

  function onSuccess(data, textStatus, jqXHR) {
    $(".list").each(addListOnClick);
  }

  function addListOnClick(e) {
    $(this).click(helloWorldTimeout);
  }

  function helloWorldTimeout() {
    setTimeout(helloWorldAlert, 1000);
  }
```

```
  function helloWorldAlert() {
    alert("Hello World!");
  }

  $(init);

})(jQuery);
```

Zbog ovoga što smo upravo imali priliku da vidimo, u jeziku JavaScript razvijeni su posebni konstrukti za asinhrono programiranje koje ćemo u nastavku izložiti.

Konstrukt koji programeri najlakše prepoznaju kao sredstvo za asinhrono programiranje u JavaScriptu je tip `Promise`. Međutim, priroda problema asinhronog programiranja je tesno vezana kontrolom toka programa, posebno sa iteriranjem i iterativnima strukturama pa je za dublje razumevanje tip `Promise` potrebno razumeti i način implementacije iteriranja i koncept specijalne vrste funkcija koje se zovu generatorske funkcije. U stvari, generatori obezbeđuju gotovo sve što je potrebno za asinhrone operacije unutar postojećih struktura kontrole toka kao što su petlje, uslovne naredbe i try/catch blokovi. Ono što generatori ne obezbeđuju je način za predstavljanje rezultata asinhrone operacije i za to se koristi tip `Promise`.

U nastavku ćemo da objasnimo pojmove generatorska funkcija i tip `Promise` i način njihovog korišćenja za asinhrono programiranje u JavaScriptu.

https://www.promisejs.org/generators/

6.4. Generatori i iteratori

Iteratori i generatori uvode koncept iteracije direktno u jezgro jezika i obezbeđuju mehanizam za prilagođavanje ponašanja `for...of` petlji.

6.4.1. Generatorske funkcije

Koncept generatora je u JavaScript uveden u verziji ES6. Primarni slučaj korišćenja generatora je predstavljanje lenjih (moguće beskonačnih) sekvenci. Objasnićemo o čemu se radi na sledećem primeru.

Posmatramo sledeći kod koji omogućuje da funkcija vrati prvih n prirodnih brojeva:

```
function count(n){
  var res = []
  for (var x = 0; x < n; x++) {
    res.push(x)
  }
  return res
}

for (var x of count(5)) {
  console.log(x)
}
```

Ovaj kod vraća prvih 5 prirodnih brojeva u petlji koja se bez prekida izvršava dok se ne dostigne izlazni kriterijum i nema načina da se njeni koraci odvijaju asinhrono: na primer, da se izvrše dva koraka pa da se sledeća tri izvrše nakon nekog vremena.

Drugi način da se uradi vraćanje prvih n prirodnih brojeva je sledeći kod:

```
function* count(){
  for (var x = 0; true; x++) {
```

```
    yield x
  }
}

for (var x of count()) {
  console.log(x)
}
```

Na prvi pogled, ne vidi se neka razlika između ova dva koda.

Naravno, razlika postoji i sastoji se u sledećem.
Ono što se u drugom snipetu zapravo dešava je da se funkcija `count()` lenjo evaluira tako da se pauzira na svakoj `yield` naredbi i čeka dok se ne zatraži sledeća vrednost. To znači da se for/of petlja izvršava zauvek, neprestano generišući sledeći ceo broj u beskonačnoj listi.
Ovakvo pauziranje funkcije omogućuje pisanje asinhronog koda koji radi asinhrone stvari unutar postojećih struktura kontrole toka kao što su petlje, uslovi i blokovi `try/catch`.

Ono što generatori ne omogućuju je način da se predstavi rezultat asinhrone operacije. Za to u JavaScriptu služi tip `Promise`.

Generatori i `Promise` zajedno omogućuju da se napiše asinhroni kod koji spolja izgleda sinhrono kao u sledećem primeru:

```
var login = async(function* (username, password, session) {
  var user = yield getUser(username);
  var hash = yield crypto.hashAsync(password + user.salt);
  if (user.hash !== hash) {
    throw new Error('Incorrect password');
  }
  session.setUser(user);
});
```

U ovom kodu sve izgleda sinhrono, a on u stvari radi asinhrono na svakoj ključnoj reči `yield`. Rezultat svakog poziva funkcije `login` je tipa `Promise`, tipa koji ima sopstvene operacije prilagođene potrebama aplikacija koje koriste rezultate asinhronih operacija.

Snipeti koji slede treba da objasne kako se, u stvari, kombinuju generatori i tip `Promise` da bi se dobio kod kojim se kontroliše prijavljivanje korisnika koje može da se dešava asinhrono.

Kao što je pokazano u prethodnom snipetu, može se pauzirati izvršavanje funkcije da bi se sačekao rezultat tipa `Promise` koristeći ključnu reč `yield` (*prinos*). Ono što je još potrebno je fina kontrola nad generatorskom funkcijom kako bi se ona ponovo pokrenula nakon pauziranja. Takav mehanizam postoji i zove se metoda `next()`. To može da izgleda ovako:

```
function assert(condition) {
    let returnValue = true
      if (!condition) {
          returnValue = false;
      }
    return returnValue
}

function* demo() {
  var rezultat = yield 10;
    if (assert(rezultat === 32))
      return 42;
}
```

```
var d = demo();
var resA = d.next();
console.log (resA) // => {value: 10, done: false}
var resB = d.next(32);
console.log (resB) // => {value: 42, done: true}
// => {value: 42, done: true}
//ako se ponovo pozove d.next() biće vraćena vrednost undefined
var resC = d.next(17)
console.log (resC) // => {value: undefined, done: true}
```

U ovom snipetu ima linija komentara (`// => {value: 10, done: false}` , `// => {value: 42, done: true}` , `{value: undefined, done: true}`) koje će vam biti potpuno jasne kada u nastavku opišemo način implementacije iteratora u JavaScriptu; za sada samo da kažemo da je `value` vraćena vrednost.

Ono što se ovde dešava je sledeće. Prvi poziv `d.next()` bez argumenta izvršava prvu naredbu `yield` koja će vratiti rezultat izraza prinosa (u primeru je ta vrednost 10). Ključna reč `yield` pauzira funkciju d (odnosno `demo()`) , a kada se pozove `d.next(32)` drugi put, prosleđuje joj se vrednost koja je rezultat izraza prinosa (u primeru to je vrednost 32). Funkcija nastavlja sa izvršavanjem linije koje slede iza naredbe `yield` a to je, prvo, poziv funkcije `assert(rezultat === 32)` i zatim naredba `return` kojom funkcija vraća konačan rezultat (u primeru je konačan rezultat 42).

6.5. Iteriranje i iteratori

6.5.1. Protokoli iteriranja

Protokoli iteriranja u JavaScriptu nisu novi ugrađeni elementi ili sintaksa, nego su to protokoli koji se mogu **implementirati bilo kojim objektom** poštujući neke konvencije. U JavaScriptu postoje dva protokola iteriranja:

- Protokol iterabilnosti;
- Protokol iteratora.

Protokol iterabilnosti propisuje način na koji JavaScript objekti mogu da definišu ili prilagode svoje ponašanje pri iteriranju, poput odluke nad kojim vrednostima će se iterirati u `for...of` konstruktu.

Da bi bio iterabilan, objekat mora da implementira metodu `@@iterator` što znači da sam objekat (ili neki od objekata u njegovom prototipskom lancu) mora da ima svojstvo sa ključem `@@iterator` dostupno putem konstante `Symbol.iterator`:

`[Symbol.iterator]`

koje je funkcije bez argumenata koja vraća objekat saglasan sa **protokolom iteratora**.

Neki ugrađeni tipovi (na primer, `Array` i `Map`) su već implementirani kao iterabilni, ali to nije slučaj sa tipom `Object`.

Kad god objekat treba da se ponovi (kao što je na početku `for...of` petlje), njegov `@@iterator` metod se poziva bez argumenata, a vraćeni iterator se koristi za dobijanje vrednosti koje treba ponoviti, odnosno on kontroliše izvršavanje iteracije.

Važno je zapaziti da, kada se pozove bez argumenata, funkcija `Symbol.iterator` se poziva kao metod objekta koji se može ponavljati. Stoga se unutar funkcije ključna reč `this` može koristiti za pristup svojstvima iterabilnog objekta da bi se odlučilo šta da se obezbedi tokom iteracije.

Ova funkcija može biti obična funkcija, ili može biti generatorska funkcija (generatorske funkcije su specijalna vrsta funkcija koja se odlikuje sposobnošću **da vrati više povratnih vrednosti** i nju ćemo

detaljno proučiti u nastavku kojima će biti reči kada se bude govorilo o asinhronom programiranju) koja pri prvom pozivu vraća objekat iteratora.

Protokol iteratora definiše standardan način za generisanje sekvence (konačne ili beskonačne) vrednosti i potencijalno vraćanje vrednosti nakon što su sve vrednosti generisane.

Objekat je ***iterator*** ako implementira metodu `next()` sa sledećom semantikom:

`next()` je funkcija koja prihvata nula ili jedan argument i **vraća objekat** koji je u skladu sa interfejsom `IteratorResult`. Ako `next()` pri korišćenju od strane ugrađenog jezičkog konstrukta (na primer, `for...of`) vrati vrednost koja nije objekat (na primer, `false` ili `undefined`), izdaje se poruka o grešci: **TypeError ("iterator.next() returned a non-object value")**.

Očekuje se da sve metode protokola iteratora (`next()`, `return()` i `throw()`) vrate objekat koji implementira interfejs `IteratorResult` . Taj interfejs mora da ima sledeća svojstva:

- **`done`** (opciono) – tip `boolean` sa vrednošću `false` ako je iterator mogao da proizvede sledeću vrednost u sekvenci, odnosno vrednost `true` ako je iterator završio sekvencu i u tom slučaju svojstvo `value` opciono specificira povratnu vrednost iteratora. (Ovo je ekvivalentno tome da se uopšte ne navede svojstvo `done`.)
- **`value`** (opciono) - bilo koja JavaScript vrednost koju je vratio iterator. Može se izostaviti kada je vrednost svojstva `done` jednaka `true`.

U praksi, ni jedno svojstvo nije striktno obavezno - vraćeni objekat može biti i prazan. Ako je vraćen objekat bez bilo kojeg svojstva, to je efektivno ekvivalentno sa `{ done: false, value: undefined }`. Ako iterator vrati rezultat sa `done: true`, očekuje se da će svi naredni pozivi `next()` takođe vratiti `done: true`, iako se to ne primenjuje na nivou jezika.

Metoda `next()` može da primi vrednost koja će biti dostupna telu metode. Nijedna ugrađena funkcija ne prosleđuje nikakvu vrednost. Vrednost posleđena sledećoj metodi generatora će postati vrednost odgovarajućeg izraza prinosa.

Opciono, iterator takođe može da implementira metode `return(value)` i `throw(exception)` koje, kada se pozovu, govore iteratoru da je pozivalac završio sa iteriranjem i da može da izvrši bilo koje neophodno čišćenje (kao što je, na primer, zatvaranje veze sa bazom podataka). Ovde je semantika sledeća:

- `return(value)` - funkcija bez argumenata ili sa jednim argumentom koja vraća objekat koji je u skladu sa interfejsom `IteratorResult`, obično sa vrednošću koja je jednaka vrednosti koja joj je prosleđena i vrednošću svojstva `done` jednakoj `true`. Pozivanje ove metode govori iteratoru da pozivalac nema nameru da više upućuje `next()` pozive i da može da izvrši bilo koju radnju čišćenja.
- `throw(exception)` - funkcija bez argumenata ili sa jednim argumentom koja vraća objekat koji je u skladu sa interfejsom `IteratorResult`, obično sa vrednošću koja je jednaka vrednosti koja joj je prosleđena i vrednošću svojstva `done` jednakoj `true`. Pozivanje ovog metoda govori iteratoru da pozivalac otkriva stanje greške, a `exception` je tipično instanca greške.

6.6. Tip `Promise`

Literatura iz Poglavlje 6

Poglavlje 7: JavaScript OKRUŽENJE

Da bi se osiguralo potpunije razumevanje jezika JavaScript, na početku ovoga poglavlja biće izloženi i sadržaji koji opisuju JavaScript okruženje i način izvršavanja JavaScript programa. U njemu se objašnjavaju JavaScript izvršno okruženje (JavaScript Runtime - JRE) i endžin V8[34] koji se trenutno najviše koristi.

7.1. JavaScript model izvršavanja i izvršno okruženje

JavaScript je skriptni jezik prvenstveno namenjen veb aplikacijama.

Kao i svaki drugi skriptni jezik, JavaScript se izvršava u određenom okruženju. Na početku, jedino okruženje u kome su se izvršavali JavaScript programi bilo je okruženje brauzera. Danas se JavaScript izvršava i u drugim, ne-brauzerskim okruženjima od kojih je, verovatno, najpoznatije okruženje Node.js.

JavaScript izvršno okruženje može se konceptualno predstaviti kontejnerom. U tom kontejneru su drugi, manji kontejneri svaki sa svojim specifičnim funkcionalnostima, i proces koji se zove *petlja događaja* (Event Loop) . Na primer, za brauzersko okruženje manji kontejneri su:

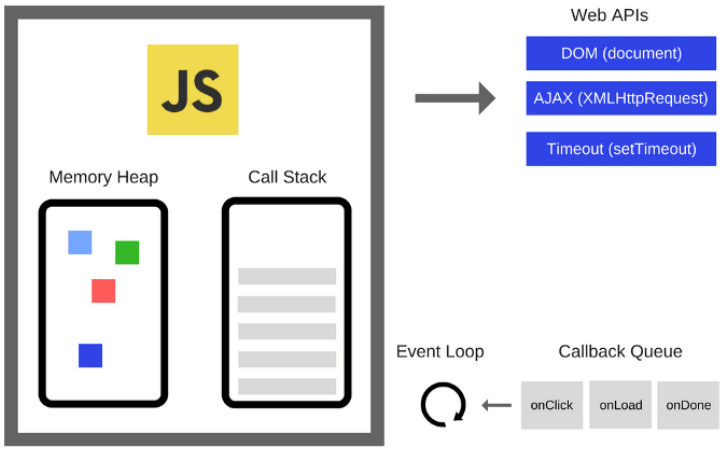
- Endžin,
- Web API kontejner
- Red čekanja povratnih poziva (Callback Queue)

Nebrauzerska okruženja takođe imaju endžin, petlju događaja i mehanizam povratnih poziva, a ono što ih razlikuje od brauzerskih okruženja je komponenta koja zauzima mesto Web API kontejnera u brauzerskim okruženjima. Na primer, Node.js na tom mestu ima I/O API niskog nivoa i biblioteku Node.js Library sa širokim dijapazonom funkcija (praktično, program se izlažu sve funkcionalnosti operativnog sistema). Posebno, Node.js ima i komponentu zvanu Node Package Manager (NPM) koja obuhvata registar i funkcije obezbeđenja konzistentnog rukovanja softverskim paketima koji su zavisnosti lokalnog projekta kao i globalno instalirani JavaScript alati.

JavaScript je jedno-nitni, ne-blokirajući asinhroni programski jezik. Ima model izvršavanja zasnovan na petlji događaja, koja je odgovorna za izvršavanje koda, prikupljanje i obradu događaja i izvršavanje pod-zadataka u redu čekanja. Centralni koncept koji omogućuje asinhronost u JavaScript-u je povratni poziv funkcije (*callback*) koji omogućuje da se nekoj funkciji delegira zadatak da pozove drugu funkciju kada to bude potrebno (recimo, neki uslovi budu zadovoljeni).

Na slici 7.1 dat je grub grafički prikaz JRE-a za okruženje brauzera.

[34] https://en.wikipedia.org/wiki/V8_(JavaScript_engine)



Slika 7.1 JRE za okruženje brauzera

Dakle, JRE je kontejner koji sadrži tri kontejnera: JS endžin, Veb API kontejner i red čekanja kao i komponentu zvanu petlju događaja. JS endžin ćemo detaljno opisati u odeljku koji sledi, sada ćemo vrlo sažeto objasniti preostale komponente.

7.1.1.1. Veb API kontejner

Veb API-ji su, u suštini, niti koje mogu samo da se pozivaju. Generalno, ovo su delovi koji su ugrađeni u samo okruženje. Na primer, u okruženju brauzera, to bi bili API-ji kao što su `document`, `XMLHttpRequest` ili `setTimeout` koji su, uglavnom, asinhrone funkcije. U Node.js-u to bi bili C++ API-ji.

Kada se na steku naiđe na ovakav poziv, on se prosleđuje u Veb API kontejner i ostaje tamo dok se ne izazove akcija: desi se 'klik' mišem, HTTP zahtev pribavi podatke iz izvora, ili tajmer dostigne zadato vreme. Izazivanje akcije dovodi do slanja funkcije povratnog poziva u treći kontejner koji se zove 'red čekanja povratnih poziva' (Callback Que).

7.1.1.2. Petlja događaja

Petlja događaja (Event Loop) je komponenta unutar JavaScript izvršnog okruženja. Njen zadatak je da STALNO nadgleda stek izvršavanja i red čekanja. Ako "vidi" da je stek izvršenja prazan, ona obaveštava red čekanja da treba da pošalje sledeću funkciju povratnog poziva. Funkcija povratnog poziva može se dodati u red čekanja svaki put kada se izazove akcija, a petlja događaja nikada ne prestaje da proverava stek izvršavanja i red čekanja.

7.1.1.3. Red čekanja povratnih poziva

Red čekanja povratnih poziva (Callback Que) skladišti po redosledu pristizanja funkcije koje u njega stignu. On čeka dok se stek izvršenja potpuno ne isprazni. Kada je stek izvršenja prazan, red čekanja šalje funkciju sa svog početka u stek izvršavanja gde se ona procesira po algoritmu steka izvršavanja. Sledeću funkciju red čekanja će poslati kada se stek izvršavanja ponovo isprazni, i tako redom. Red čekanja je struktura podataka koja izvršava FIFO (*first in first out*) algoritam. I to ponašanje je upravo ono na šta se misli kada se kaže da JavaScript može da se izvršava **asinhrono**. JavaScript **ne radi asinhrono**, ali se asinhronost simulira pomoću reda čekanja i steka izvršavanja.

U trenutku kada se isprazne stek izvršavanja i red čekanja, petlja događaja terminira proces izvršavanja programa.

7.2. JavaScript endžin

JavaScript endžin je komponenta koja preuzima izvorni kod, parsira ga i upravlja njegovim izvršavanjem. Postoji više JavaScript endžina, od kojih je V8 endžin najrasprostranjeniji i koristi se, na primer, u brauzeru Chrome i okruženju Node.js.

Na najvišem nivou JavaScript endžin sastoji se od dve osnovne komponente:

- Hip memeorije (Memory Heap) koji je odgovoran za alokaciju memorije, i
- Steka poziva (Call Stack) koji je odgovoran za okvire steka pri izvršavanju poziva (funkcija).

7.2.1. Hip memorija

Kontejner JavaScript endžina koji se zove hip memorija je deo memorije u koji se smeštaju praktično svi podaci pa i deklaracije varijabli i funkcija kroz koje endžin prolazi u toku izvršavanja programa.

7.2.2. Kontekst i stek izvršavanja

Jednostavno rečeno, kontekst izvršavanja je apstraktni koncept okruženja u kome se JavaScript kod evaluira i izvršava. Svaki kod koji se izvršava u JavaScript-u izvršava se unutar nekog konteksta izvršavanja. Postoje dva[35] osnovna tipa konteksta izvršavanja u JavaScript-u

- **Globalni kontekst izvršavanja.** To je pretpostavljeni ili bazni kontekst izvršavanja. Kod koji nije ni u jednoj funkciji nalazi se u globalnom kontekstu izvršavanja. On radi dve stvari: kreira *globalni objekat* koji je u slučaju baruzera `window` objekat i postavlja vrednost pokazivača `this` da pokazuje na globalni objekat. Može da postoji **samo jedan globalni kontekst** u programu.
- **Funkcijski kontekst izvršavanja.** Svaki put kada se funkcija pozove, kreira se potpuno nov kontekst izvršavanja za pozvanu funkciju. Svaka funkcija ima sopstveni kontekst izvršavanja koji se poziva kada se funkcija pozove a ne kada se deklariše. Pri kreiranju konteksta izvršavanja funkcije, prolazi se kroz niz koraka po definisanom redosledu. Taj proces biće kasnije detaljno opisan.

Svi konteksti izvršavanja kreirani u toku izvršavanja koda skladište se u **Stek izvršavanja** (zovu ga i **stek poziva**) koji je stek sa LIFO (Last in, First out) strukturom.

7.2.2.1. Stek izvršavanja

Dakle, da ponovimo: U računarskoj nauci, **stek izvršavanja** (zovu ga i **stek poziva**) je stek struktura podataka koja skladišti informacije o aktivnim delovima koda (funkcijama, podprogramima) računarskog programa. Koristi se za različite, međusobno povezane svrhe, ali mu je glavna namena čuvanje traga o tački kojoj aktivni podprogram treba da vrati kontrolu kada okonča svoje izvršavanje. Aktivni kod je deo koda (podprogram, funkcija) koji je pozvan i koji treba da završi svoje izvršavanje i nakon toga da preda kontrolu onome koji ga je pozvao tački iz koje je pozvan u programu.

Drugim rečima, stek poziva je struktura podataka koja beleži gde se nalazimo u programu. Pri ulasku u funkciju, relevantne informacije (kontekst aktivnog koda) se postavljaju na vrh steka. Pri povratku iz funkcije, uklanja se postavljeni sadržaj.

[35] Pored ova dva konteksta postoji i treći: **kontekst izvršavanja `eval` funkcije**. Naime, kod koji se izvršava u `eval` funkciji takođe ima svoj kontekst izvršavanja ali ga JS programeri retko koriste.

Drugi kontejner u JavaScript endžinu je upravo **stek izvršavanja**. Kada JavaScript endžin naiđe na izvršivu liniju koda poput poziva funkcije, on tu liniju dodaje na stek izvršavanja. Nakon dodavanja funkcije na stek izvršavanja, JavaScript endžin ulazi u tu funkciju i počinje da parsira njen kod pri čemu dodaje varijable na hip, nove funkcijske pozive smešta na vrh steka, ili ih šalje trećem kontejneru u koji idu Web API pozivi.
Kada funkcija vrati vrednost (završi se njeno izvršavanje) ili bude poslata, na primer, u **Web API kontejner**, ona se skida sa steka i prelazi se na sledeću funkciju na steku. Ako JavaScript endžin stigne do kraja funkcije a nije eksplicitno vraćena izlazna vrednost, endžin vraća vrednost `undefined` i skida funkciju sa steka. Upravo taj proces parsiranja funkcije i njenog skidanja sa steka je ono što se misli kada se kaže da se JavaScript izvršava **sinhrono**. Endžin radi jednu stvar u trenutku na jednoj niti.

Stek je struktura podataka koja izvršava LIFO (last in first out) algoritam. Samo funkcija na vrhu steka je u fokusu, i endžin nikada neće preći na sledeću funkciju ukoliko ona koja je iznad nije uklonjena sa steka.

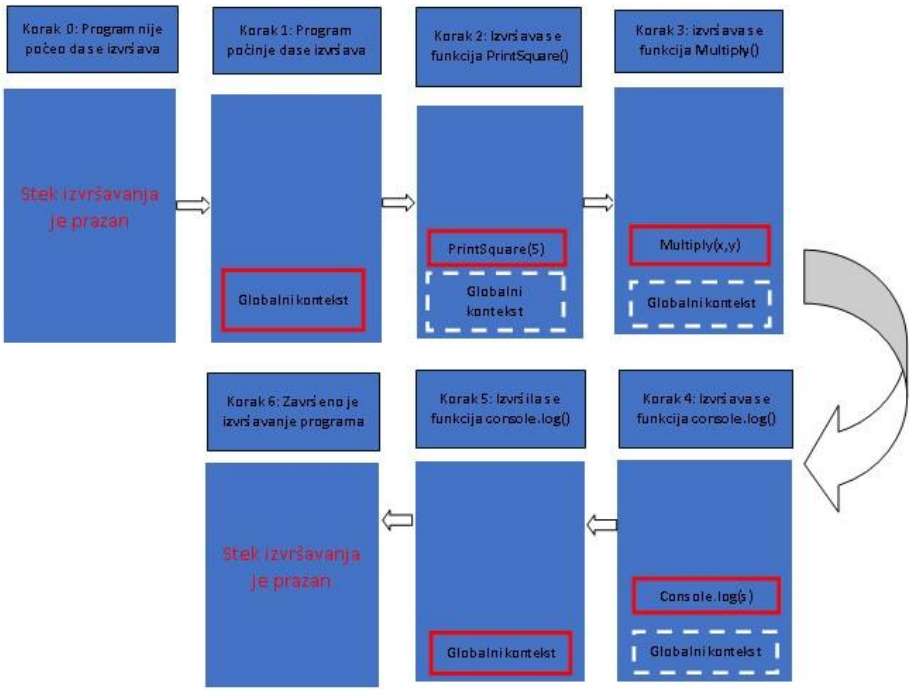
Da vidimo i primer preuzet iz izvora https://blog.sessionstack.com/tagged/tutorial. Dat je kod:

```
function multiply(x, y) {
    return x * y;
}

function printSquare(x) {
    var s = multiply(x, x);
    console.log(s);
}

printSquare(5);
```

Kada endžin započne sa izvršavanjem ovoga koda, prvo će na stek postaviti globalni kontekst. Zatim će na stek postavljati redom kontekste poziva funkcija. Završetak poziva uklanja tekući funkcijski kontekst sa steka. Kada se završi izvršavanje programa, uklanja se i globalni kontekst. Na slici 7.2 prikazan je izgled steka poziva pri izvršavanju ovoga koda:



Slika 7.2 Stek izvršavanja za primer 7.1

Napomena: Svaki ulaz u stek poziva (pravougaonici sa crvenom ivicom na slici 7.2) naziva se stek frejm (**Stack Frame**).
A to je i format stek trejsa koji će Vam trebati pri pojavi greške u programu. Evo i primera.

Neka Vam je dat sledeći kod (u njemu se namerno u funkciji `foo()` generiše greška):

```
function foo() {
    throw new Error('SessionStack will help you resolve crashes :)');
}

function bar() {
    foo();
}

function start() {
    bar();
}

start();
```

Ako ga izvršite u Chrome brauzeru dobićete sledeći rezultat:

```
VM6558:2 Uncaught Error: SessionStack will help you resolve crashes :)
    at foo (<anonymous>:2:11)
    at bar (<anonymous>:5:5)
    at start (<anonymous>:8:5)
    at <anonymous>:10:1
foo @ VM6558:2
bar @ VM6558:5
start @ VM6558:8
(anonymous) @ VM6558:10
```

Kao i svaki drugi (memorijski) resurs računara, stek je ograničen. Zbog toga može da dođe do greške prekoračenja steka. To je situacija u kojoj se stek "prepuni" frejmovima. U funkcionalnom programiranju (i ne samo funkcionalnom) ova vrsta greške ima veze sa rekurzijom. Evo i primera.

Ako u Chrome brauzeru pokrenete sledeći kod:

```
function fRec() {
    fRec();
}

fRec();
```

dobićete rezultat

```
VM6572:2 Uncaught RangeError: Maximum call stack size exceeded
    at fRec (<anonymous>:2:5)
    at fRec (<anonymous>:2:5)
    at fRec (<anonymous>:2:5)
    at fRec (<anonymous>:2:5)
    at fRec (<anonymous>:2:5)
    at fRec (<anonymous>:2:5)
    at fRec (<anonymous>:2:5)
    at fRec (<anonymous>:2:5)
    at fRec (<anonymous>:2:5)
    at fRec (<anonymous>:2:5)
```

7.2.2.2. Kontekst izvršavanja

U prethodnom odeljku smo videli kako JavaScript upravlja stekom izvršavanja. Sada nam je zadatak da objasnimo kako JavaScript endžin rukuje kontekstom izvršavanja. Kontekst izvršavanja obuhvata dve faze:

1. Faza **kreiranja**, i
2. Faza **izvršavanja**.

7.2.2.2.1. Faza kreiranja

Ovo je faza u kojoj se kreira kontekst izvršavanja. U njoj se dešavaju sledeće stvari:

1. Kreira se komponenta **LexicalEnvironment** - **Leksičko okruženje**.
2. Kreira se komponenta **VariableEnvironment** - **var leksičko okruženje** (leksičko okruženje entiteta deklarisanih `var` deklaracijom) .

Konceptualno se kontekst izvršavanja može predstaviti na sledeći način:

```
ExecutionContext = {
  LexicalEnvironment = <referenca na LexicalEnvironment u memoriji>,
  VariableEnvironment = <referenca na VariableEnvironment u memoriji>,
}
```

Leksičko okruženje (Lexical Environment)

Zvanična ES6 dokumentacija definiše leksičko okruženje na sledeći način:

Leksičko okruženje (Lexical Environment) *je tip specifikacije koji definiše pridruživanje* identifikatora *specifičnim varijablama i funkcijama bazirano na leksičkoj strukturi ugnježdavanja ECMAScript koda. Leksičko okruženje sastoji se od Zapisa Okruženja (Environment Record) i moguće nula reference na spoljašnje leksičko okruženje.*

Konkretnije rečeno, *leksičko okruženje* je struktura koja sadrži **mapiranje identifikator-varijabla**. Pri tome, termin **identifikator** odnosi se na imena varijabli/funkcija a **varijabla** je referenca na stvarni objekat (uključujući funkcijski objekat i nizovni (`Array`) objekat) ili primitivnu vrednost.

Za sledeći snipet:

```
var a = 20;
var b = 40;

function foo() {
  console.log('bar');
}
```

leksičko okruženje izgleda ovako:

```
lexicalEnvironment = {
  a: 20,
  b: 40,
  foo: <referenca na funkciju foo>
}
```

Svako leksičko okruženje ima tri komponente:

1. Zapis okruženja (Environment Record);
2. Referencu na spoljašnje okruženje;
3. `this` vezivanje.

Zapis okruženja. Predstavlja mesto gde su deklaracije varijabli i funkcija smeštene unutar leksičkog okruženja. Postoje i dva tipa Zapisa okruženja :

- **Deklaracioni Zapis okruženja (Declarative environment record)** u kome se skladište deklaracije varijabli i funkcija. Leksičko okruženje za kod funkcije sadrži deklaracioni zapis okruženja.
- **Objektni zapis okruženja (Object environment record)** je leksičko okruženje za globalni kod. Odvojeno od deklaracija varijabli i funkcija, Objektni zapis okruženja skladišti i globalni objekat vezivanja (`window` objekat u brauzerima, `global` objekat u node.js). Za svako objektno svojstvo vezivanja (u slučaju brauzera, tu su sadržana svojstva i metode koje brauzer obezbeđuje objektu `window`), kreira se nova stavka u Zapisu.

Važna napomena: Za funkcijski kod, Zapis okruženja sadrži i objekat `arguments` koji sadrži mapiranja između indeksa (svojstva objekta `arguments`) i vrednosti argumenata prosleđenih funkciji i svojstvo `length` koje predstavlja broj argumenata prosleđenih funkciji). Evo primera:

```
function foo(a, b) {
  var c = a + b;
  console.log (arguments)
}
foo(2, 3);


// Log je
Arguments(2) [2, 3, callee: ƒ, Symbol(Symbol.iterator): ƒ]
0: 2
1: 3
callee: ƒ foo(a, b)
length: 2
Symbol(Symbol.iterator): ƒ values()
[[Prototype]]: Object
```

Referenca na spoljašnje okruženje

Referenca na spoljašnje okruženje znači da postoji pristup spoljašnjem leksičkom okruženju. To znači da JavaScript endžin može da traži varijable unutar spoljašnjeg okruženja ako tih varijabli nema u tekućem leksičkom okruženju.

`this` vezivanje

U ovoj komponenti se određuje ili postavlja vrednost ključne reči `this`. U globalnom kontekstu izvršavanja vrednost `this` pokazuje na globalni objekat (u brauzeru, `this` pokazuje na Window Object, dok u ne-brauzerskom okruženju pokazuje na `Global Object`).

U kontekstu izvršavanja funkcije vrednost `this` zavisi od načina pozivanja funkcije. Ako je poziv putem objektne reference (funkcija se poziva kao metoda objekta), vrednost ključne reči `this` postavlja se da pokazuje na taj objekat – objekat metode, u drugim slučajevima vrednost `this` pokazuje na globalni objekat ili je `undefined` (u `strict` režimu).

Evo primera:

```
const person = {
  name: 'peter',
  birthYear: 1994,
  calcAge: function() {
    console.log(2023 - this.birthYear);
  }
}
person.calcAge(); // 29 = 2023 -1994
// 'this' pokazuje na 'person', jer je 'calcAge' pozvano referencom
// na 'person' objekat
```

```
const calculateAge = person.calcAge; /* calculateAge je nezavisna funkcija
                                        (nije metoda objekta) */
calculateAge(); /* NaN jer 'this' pokazuje na globalni window objekat u kome
                   nema vrednosti birthYear */
```

Leksičko okruženje predstavljeno pseudokodom izgleda ovako:

```
GlobalExectionContext = {
  LexicalEnvironment: {
    EnvironmentRecord: {
      Type: "Object",
      // Ovde dolaze vezivanja identifikatora
    }
    outer: <null>, // jer za njega nema spoljašnjeg
    this: <global object>
  }
}
```

```
FunctionExectionContext = {
  LexicalEnvironment: {
    EnvironmentRecord: {
      Type: "Declarative",
      // Ovde dolaze vezivanja identifikatora
    }
    outer: <referenca na globalno ili okruženje spoljašnje funkcije >,
    this: <zavisi od načina pozivanja funkcije>
  }
}
```

var Leksičko okruženje (Variable Environment)

To je takođe leksičko okruženje čiji Zapis okruženja (`EnvironmentRecord`) sadrži vezivanja kreirana deklaracijom `var` unutar tog konteksta izvršavanja.

Pošto je u pitanju leksičko okruženje, ono ima sva svojstva i komponente Leksičkog okruženja. Od ES6, razlika između Leksičkog okruženja i `var` Leksičkog okruženja je da se Leksičko okruženje koristi za skladištenje vezivanja funkcija i varijabli (`let` i `const` ključne reči), dok se `var` Leksičko okruženje koristi samo za skladištenje vezivanja varijabli (ključna reč `var`).

7.2.2.2.2. Faza izvršavanja

U fazi izvršavanja vrše se dodele svim varijablama i izvršava se kod. Faza izvršavanja biće objašnjena na primeru.

Primer. Posmatra se sledeći kod:

```
let a = 20;
const b = 30;
var c;

function multiply(e, f) {
  var g = 20;
  return e * f * g;
}

c = multiply(20, 30);
```

Kada se izvršava ovaj kod JavaScript endžin kreira globalni kontekst za izvršavanje celokupnog programa - globalnog koda. Globalni kontekst u fazi kreiranja izgleda ovako:

```
GlobalExectionContext = {  LexicalEnvironment: {
  EnvironmentRecord: {
      Type: "Object",
      // Ovde idu vezivanja identifikatora
      a: < uninitialized >,
      b: < uninitialized >,
      multiply: < func >
    }
    outer: <null>,
    ThisBinding: <Global Object>
  },
  VariableEnvironment: {
    EnvironmentRecord: {
      Type: "Object",
      // Ovde idu vezivanja identifikatora
      c: undefined,
    }
    outer: <null>,
    ThisBinding: <Global Object>
  }
}
```

U toku faze izvršavanja vrše se dodele varijabli. Dakle, u fazi izvršavanja globalni kontekst izvršavanja će da izgleda ovako:

```
GlobalExectionContext = {LexicalEnvironment: {
    EnvironmentRecord: {
      Type: "Object",
      // Ovde idu vezivanja identifikatora
      a: 20,
      b: 30,
      multiply: < func >
    }
    outer: <null>,
    ThisBinding: <Global Object>
  },
VariableEnvironment: {
  EnvironmentRecord: {
       Type: "Object",
       // Ovde idu vezivanja identifikatora
       c: undefined,
  }
    outer: <null>,
    ThisBinding: <Global Object>
  }
}
```

Kada endžin naiđe na poziv funkcije `multiply(20, 30)` , kreira se novi kontekst izvršavanja funkcije koji izvršava kod funkcije. U fazi kreiranja taj kontekst izgleda ovako:

```
FunctionExectionContext = {LexicalEnvironment: {
    EnvironmentRecord: {
      Type: "Declarative",
      // Ovde idu vezivanja identifikatora
      Arguments: {0: 20, 1: 30, length: 2},
```

```
    },
    outer: <GlobalLexicalEnvironment>,
    ThisBinding: <Global Object or undefined>,
  },
    VariableEnvironment: {
    EnvironmentRecord: {
      Type: "Declarative",
      // Ovde idu vezivanja identifikatora
      g: undefined
    },
    outer: <GlobalLexicalEnvironment>,
    ThisBinding: <Global Object or undefined>
  }
}
```

Nakon toga, kontekst izvršavanja prolazi kroz fazu izvršavanja u kojoj se vrše dodele varijablama unutar funkcije. Sada kontekst izvršenja izgleda ovako:

```
FunctionExectionContext = {LexicalEnvironment: {
    EnvironmentRecord: {
      Type: "Declarative",
      // Ovde idu vezivanja identifikatora
      Arguments: {0: 20, 1: 30, length: 2},
    },
    outer: <GlobalLexicalEnvironment>,
    ThisBinding: <Global Object or undefined>,
  },
  VariableEnvironment: {
    EnvironmentRecord: {
      Type: "Declarative",
      // Ovde idu vezivanja identifikatora
      g: 20
    },
    outer: <GlobalLexicalEnvironment>,
    ThisBinding: <Global Object or undefined>
  }
}
```

Po završetki izvršavanja funkcije vraćena vrednost se skladišti u `c`. Dakle, ažurira se globalno leksičko okruženje. Nakon toga završava se izvršavanje programa.

Napomena: Varijable deklarisane pomoću `let` i `const` u ovom primeru nemaju pridruženu vrednost u toku faze kreiranja, za razliku od varijabli deklarisanih putem `var` koje se postavljaju na `undefined`. To je zbog toga što se u fazi kreiranja kod skenira u potrazi za deklaracijama funkcija i varijabli i, dok se deklaracija funkcije u celosti skladišti u okruženje, varijable se inicijalno postavljaju na `undefined` (za deklaraciju `var`) ili ostaju neinicijalizovane (u slučaju `let` i `const`).

To je razlog što se može pristupiti varijablama deklarisanim pomoću `var` pre naredbe deklaracije (iako je vrednost `undefined`) a dobija se fatalna greška kada se pokuša pristup varijablama deklarisanim putem `let` i `const` pre navođenja naredbe deklaracije.

U JavaScript-u to se zove ***podizanje*** (hoisting).

Napomena: Da JavaScript endžin nije mogao da nađe vrednost `let` varijable (recimo, nije dodeljena), on bi joj dodelio vrednost `undefined`.

7.2.3. Detalji rada JavaScript endžina V8

JavaScript endžin izvršava i kompajlira JavaScript program u nativni mašinski kod. Ovde ćemo objasniti kako radi endžin V8 zato što se on koristi i u Chromium-u i u Node.js , odnosno Electron-u, mada su i drugi endžini napravljeni na isti način. U suštini, V8 endžin radi na sledeći način:

1. Započinje pribavljanjem izvornog JavaScript koda (recimo sa mreže).
2. Parsira izvorni kod i transformiše ga u apstraktno sintaksno stablo (AST).
3. Na bazi AST, **Ignition** interpreter može da počne da radi svoj posao i da pravi bajt-kod.
4. U ovoj tački, endžin počinje da izvršava kod i da prikuplja povratne informacije o tipu.
5. Da bi se program ubrzao, bajt-kod se može proslediti optimizujućem kompajleru zajedno sa povratnim informacijama o tipu. Optimizujući kompajler na bazi primljenog pravi neke pretpostavke (o tipu) i proizvodi visoko optimizovan mašinski kod.
6. Ako se u nekoj tački izvršavanja ispostavi da je neka od pretpostavki kompajlera postala nekorektna, kompajler vrši deoptimizaciju i vraća se na interpreter.

U nastavku su malo detaljnije opisani ovi koraci.

7.2.3.1. Priprema izvornog koda

Prva stvar koju radi V8 endžin je pribavljanje izvornog koda. Kada se kod pribavi, treba ga modifikovati u oblik koji kompajler može da razume. Taj proces naziva se parsiranje i sastoji se iz dva dela: *skenera* i *parsera*.

Skener prima JavaScript fajl sa izvornim kodom i konvertuje ga u listu poznatih tokena. Na adresi https://github.com/v8/v8/blob/master/src/parsing/keywords.txt dostupna je lista JavaScript tokena.

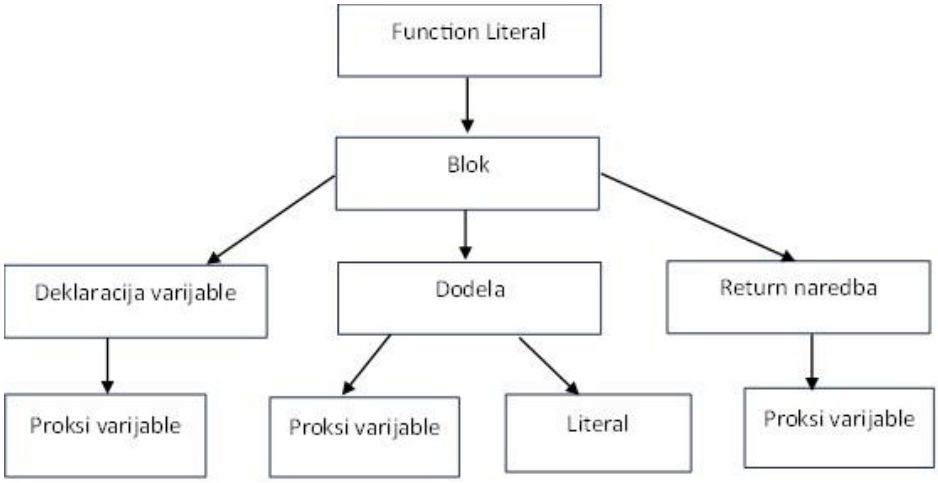
Parser prima rezultat skeniranja i kreira apstraktno sintaksno stablo (*Abstract Syntax Tree*, AST) što je reprezentacija izvornog koda u obliku stabla. Svaki čvor u tom stablu predstavlja jezički konstrukt koji se pojavljuje u kodu (ali se u stablu ne predstavljaju svi detalji sintakse, nego samo strukturni i vrednosti – zato se stablo zove apstraktno).

Za sledeći jednostavan primer to izgleda ovako.

Posmatra se kod:

```
function foo() {
  let bar = 1;
  return bar;
}
```

Taj kod rezultuje sledećom strukturom stabla (Slika 7.3):



Slika 7.3 Primer AST stabla

Na slici se pojavljuje i čvor Proksi varijable (`VariableProxy`) - element koji povezuje apstrakciju varijable sa lokacijom u memoriji. Proces razrešavanja `VariableProxy` čvorova zove se ***analiza dosezanja*** (engl. *Scope Analysis*). U ovom primeru rezultat tog procesa će biti da svi čvorovi Proksi varijable pokazuju na istu varijablu sa imenom `bar` : postoji samo ta deklarisana varijabla, jedino toj varijabli je dodeljena vrednost, jedino tu varijablu vraća funkcija `foo` .

Kod se može izvršiti putem predureženog obilaska stabla (koren, levi, desni):

1. Definisanje `foo` funkcije.
2. Deklarisanje `bar` varijable.
3. Dodela vrednosti `1` varijabli `bar`.
4. Vraćanje `bar` iz funkcije.

7.2.3.2. Transformisanje u mašinski kod

Da bi se izvorni kod izvršio, on mora da bude transformisan u mašinski kod. Ima više načina da se to uradi.

Najčešći način je **transformisanje koda kompilacijom**. U tom modelu **izvorni kod se transformiše u mašinski kod pre započinjanja izvršenja programa**. Ovaj pristup omogućuje bolju optimizaciju i performantniji kod.

Drugi često korišćen način je **interpretacija** gde se **svaka linija koda prevodi i odmah izvršava**. Taj način koriste dinamički tipizirani jezici (jezici gde se tip zaključuje iz izvornog koda i konteksta izvršavanja i ne može se tačno znati pre izvršenja). Ovaj pristup je lakši za implementiranje, ali je obično sporiji.

Naravno, danas se često koristi kombinacija ova dva pristupa koja se zove **Just-in-Time (JIT) kompilacija**.

Endžin **V8** koristi **interpretaciju kao bazni metod**, ali ima i mogućnost da **detektuje funkcije koje se koriste češće** i da ih **kompajlira koristeći informacije o tipu iz prethodnih izvršavanja**. Naravno, pri tome postoji i mogućnost da se tip u međuvremenu promeni. U tom slučaju mora se kompajlirani kod deoptimizirati i vratiti se na interpretaciju. Nakon toga, odnosno pribavljanja povratne informacije o novom tipu, funkcija se može rekompajlirati. U nastavku sledi malo više detalja o JIT kompilaciji.

7.2.3.2.1. Interpreter

V8 koristi interpreter koji se zove **Ignition**[36]. On inicijalno prima AST i generiše bajt-kod[37]. Bajt-kod instrukcije imaju i meta-podatke poput pozicije izvornih linija za kasnije debagovanje. Generalno, bajt-kod instrukcije odgovaraju JavaScript apstrakcijama.

Bajt kod za primer izgleda ovako:

```
LdaSmi #1 // Upiši 1 u akumulator
Star r0   // Upiši u r0 (r0 registar) ono što se nalazi u akumulatoru (bar)
Ldar r0   // Učitaj ono što se nalazi u r0 u akumulator
Return    // Vrati ono što je u akumulatoru
```

Ignition ima nešto što se zove *akumulator* - mesto u koje se mogu skladištiti i iz koga se mogu čitati vrednosti. Akumulator je mehanizam kojim se izbegavaju puš i pop operacije na steku. On je i implicitni (nema ga u listi) argument za mnoge konkretne bajt-kodove i obično u sebi čuva rezultat operacije (podsetite se asemblera). Instrukcija `Return` implicitno vraća akumulator. Izvorni kod za V8 bajt-kod dostupan je na adresi https://github.com/v8/v8/blob/master/src/interpreter/bytecodes.h. Reprezentacija drugih JavaScript konstrukata (petlje, async/await) u bajt kodu može se naći na adresi https://github.com/v8/v8/tree/master/test/cctest/interpreter/bytecode_expectations.

7.2.3.2.2. Kompajler

Ignition "dobacuje" do ove tačke. Ako funkcija postane dovoljno interesantna biće optimizovana u kompajleru **Turbofan** da bi se ubrzala.

Turbofan preuzima bajt-kod koji je napravio **Ignition** i povratne informacije o tipu (**Feedback Vector**) funkcije, primenjuje skup redukcija na bazi toga što je preuzeo i pravi mašinski kod. Međutim, niko ne garantuje da se informacije o tipu neće u budućnosti promeniti. Ukoliko do promena dođe, radi se proces zvani deoptimizacija koji odbacuje optimizovani kod, vraća se na interpretirani kod, nastavlja izvršavanje i ažurira povratnu informaciju o tipu.

7.2.3.2.3. Izvršenje

Nakon generisanja bajt-koda, **Ignition** interpretira instrukcije koristeći tabelu rukovalaca u kojoj se rukovaocu pristupa pomoću bajt-kod ključeva. On nalazi odgovarajuću funkciju rukovaoca i izvršava je za zadate argumente.

Važna stvar u ovom procesu je prikupljanje povratnih informacija o tipu i upravljanje tim informacijama. Pokušaćemo da dočaramo (makar grubu) sliku o tome.

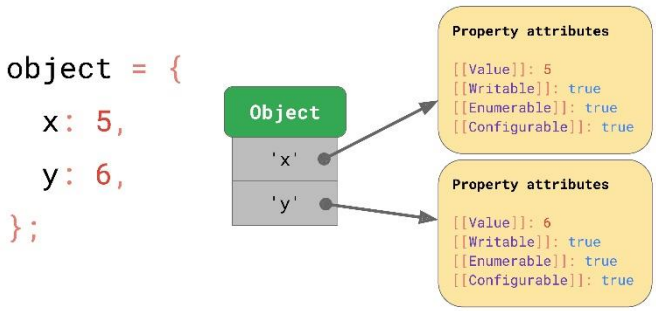
Kao prvo, treba da se podsetimo kako se JavaScript objekti mogu reprezentovati u memoriji.

ECMAScript specifikacija u suštini objekte definiše kao rečnike sa ključevima koji pokazuju na atribute (karakteristike) svojstva. (Slika 7.4).

---

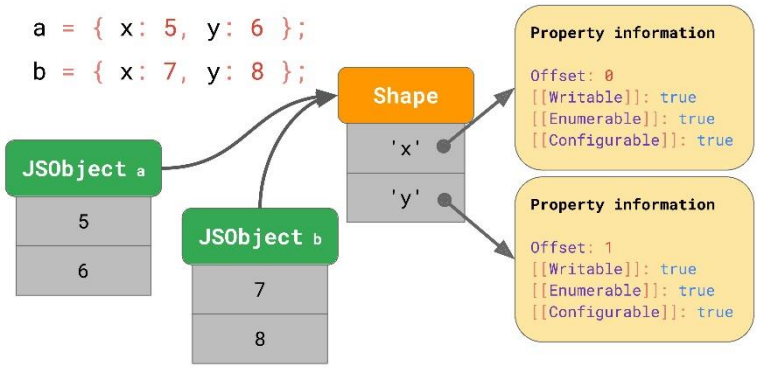
[36] https://github.com/v8/v8/blob/master/src/interpreter/interpreter.h

[37] **Bytecode**, kaže se i **portabilan kod** ili **p-kod**, je oblik seta instrukcija dizajniran za efikasno izvršavanje putem softverskog interpretera. Bajt-kod program može se izvršavati parsiranjem i direktnim izvršavanjem instrukcija, jedna po jedna što čini tu vrstu bajt-kod interpretera veoma portabilnim.



Slika 7.4 ECMAScript model objekta [https://mathiasbynens.be/notes/shapes-ics]

U JavaScript programima je vrlo uobičajeno prisustvo više objekata sa istim ključevima svojstava. Za takve objekte kaže se da imaju isti *oblik* (engl. *Shape*). Takođe je i vrlo uobičajeno pristupanje istom svojstvu objekata istog oblika. Ako bi se za reprezentaciju svakog objekta koristio model iz ECMAScript specifikacije, to bi bilo nepotrebno trošenje memorije i usporavanje rada programa. Zbog toga brauzeri, pa i V8, razdvajaju strukturu objekta od samih vrednosti pomoću mehanizma zvanog **Object Shape** (u V8 endžinu se to naziva **Map**) i vektora vrednosti u memoriji. Vizuelna reprezentacija mehanizma **Object Shape** za slučaj dva objekta **a** i **b** sa istim svojstvima (**x** i **y**) prikazana je na slici 7.5.

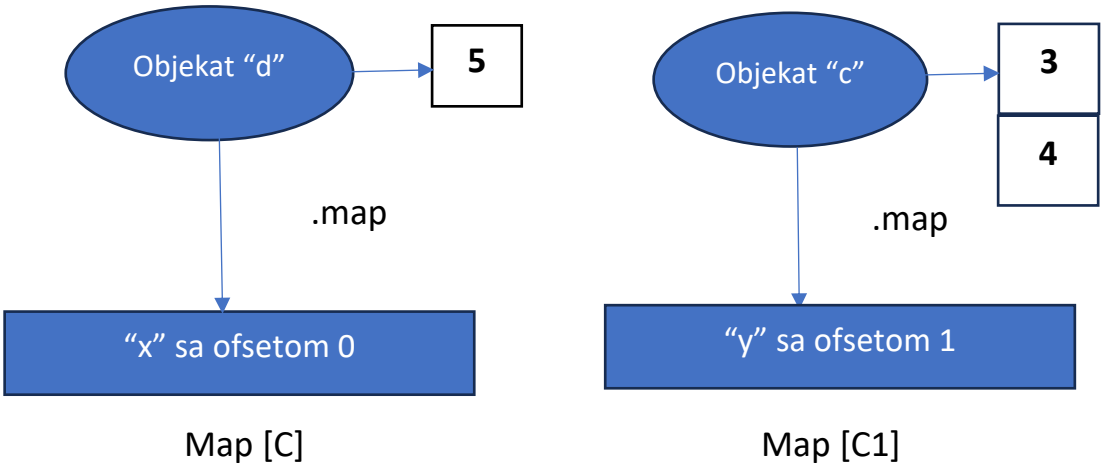


Slika 7.5 Mehanizam Object Shape [https://mathiasbynens.be/notes/shapes-ics]

Evo i primera koda u kome dva objekta (c i d) imaju svojstva sa istim imenima ključeva (x i y):

```
let c = { x: 3 }
let d = { x: 5 }
c.y = 4
```

Prva linija koda rezultuje pravljenjem "shape-a" **`Map[c]`** koji ima svojstvo **x** sa ofsetom `0`. Za drugu liniju, V8 će da ponovno iskoristi isti "shape" za novu varijablu. Nakon treće linije, kreiraće novi "shape" `Map[c1]` za svojstvo **y** sa ofsetom `1` i kreiraće link na prethodni "shape" **`Map[c]`**. Vizuelzacija je prikazana na slici 7.6.



Slika 7.6 Primer mehanizma `Object Shape`

Iz ovog primera se može videti da svaki objekat može da ima link na strukturu ***Shape*** u kojoj V8 može da, za svako ime svojstva, nađe ofset do njegove vrednosti u memoriji (ofset u vektoru vrednosti).

Jasno je da su **`Object Shape`** u suštini povezane liste. Dakle, rezultat izvršavanja naredbe `c.x` je da V8 ode na zaglavlje liste, tamo nađe y, ode na povezani **Shape**, pribavi **x** i učita iz njega ofset. Zatim ide na memorijski vektor i vraća prvi element iz njega.

Naravno, u velikim aplikacijama je ogroman broj povezanih **`Object Shape`**-ova. Vreme pretrage povezane liste je linearno pa traženje svojstava postaje veoma skupa operacija. Za ublažavanje ovog problema V8 nudi **Inline Cache (IC)**[38] (kod V8 IC dostupan je na https://github.com/v8/v8/tree/master/src/ic). On pamti informacije o tome gde se nalaze svojstva objekata i na taj način redukuje pretragu. On prati sve **CALL**, **STORE**, i **LOAD** događaje unutar funkcije i beleži sve **Shape**-ove koji prođu. Struktura podataka u kojoj se čuva IC zove se **`Feedback Vector`**[39]. To je prosto jedan niz koji u sebi čuva sve **IC**-ove za funkciju. Evo primera:

```
function load(a) {
  return a.key;
}
```

Za gornju funkciju Feedback vektor izgleda ovako:

```
[{ slot: 0, icType: LOAD, value: UNINIT }]
```

To je jednostavna funkcija sa smo jednim **IC** koji je tipa `LOAD` i ima value `UNINIT`. To znači da nije inicijalizovan i da se ne zna šta će sledeće da se desi.

Naravno, pri pozivanju funkcije menja se **Inline Cache**. Evo primera:

```
let first = { key: 'first' }  // shape A
let fast  = { key: 'fast' }   // isti shape A
let slow  = { foo: 'slow' }   // novi shape B
load(first)
load(fast)
load(slow)
```

Nakon prvog poziva **`load`** funkcije, inline cache će da dobije ažuriran **`value`**:

[38] https://github.com/v8/v8/tree/master/src/ic

[39] https://github.com/v8/v8/blob/master/src/objects/feedback-vector.h

`[{ slot: 0, icType: LOAD, value: MONO(A) }]`

Svojstvo **`value`** dobija vrednost monomorfičan (`MONO(A)`) , što znači da se taj **`cache`** može rezolvirati samo na **Shape A**.

Nakon drugog poziva, V8 će da proveri vrednost svojstva **`value`** za **IC**-ove i videće da je ono monomorfično i da ima isti **Shape** kao i varijabla `fast`. Zato će da vrati ofset i da ga rezolvira.

Treći put, **Shape** je različit od uskladištenog. Zbog toga će V8 da ga manuelno rezolvira i da ažurira `value` na polimorfično stanje predstavljeno nizom od dva moguća **Shape** -a:

`[{ slot: 0, icType: LOAD, value: POLY[A,B] }]`

Od sada na dalje, svaki put kada se pozove funkcija **`load`**, V8 mora da radi proveru iterirajući nad više mogućnosti.

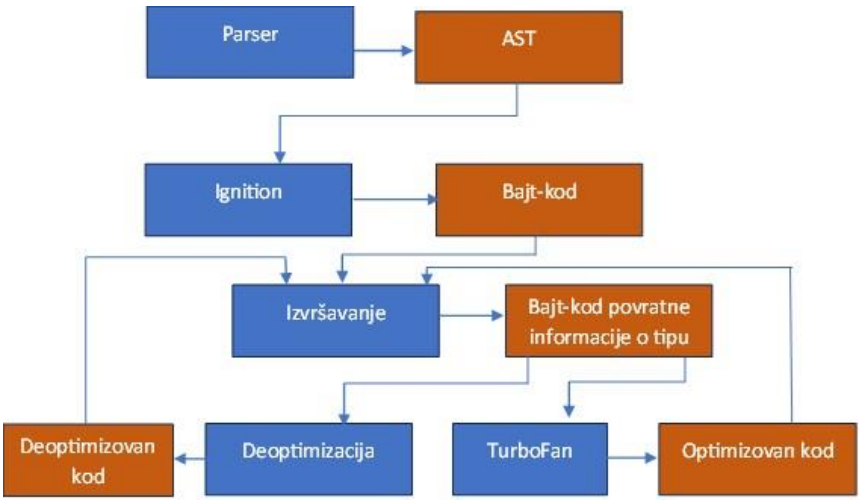
Kod se može ubrzati tako što će se inicijalizovati objekti istim tipom i njihova struktura se neće previše menjati[40].

Ovaj mehanizam vodi i evidenciju o tome koliko puta je pozvan da se odluči da li je u pitanju dobar kandidat za optimizujući kompajler **Turbofan** (https://v8.dev/docs/turbofan).

Vrlo dobar izvor opisa mehanizma Object Shape i inlajn keširanja u brauzerima dostupan je na adresi https://mathiasbynens.be/notes/shapes-ics.

7.2.3.2.4. Rezime

Globalni tok kompilacije JavaScript programa endžina V8 može se prikazati slikom 7.7.



Slika 7.7 Globalni tok kompilacije JavaScript programa endžina V8

Dakle, V8 endžin radi sledeće:

a) Započinje pribavljanjem izvornog JavaScript koda (recimo sa mreže).
b) Parsira izvorni kod i transformiše ga u apstraktno sintaksno stablo (AST).
c) Na bazi AST, **Ignition** interpreter može da počne da radi svoj posao i da pravi bajt-kod.
d) U ovoj tački, endžin počinje da izvršava kod i da prikuplja povratne informacije o tipu.
e) Da bi se program ubrzao, bajt-kod se može proslediti optimizujućem kompajleru zajedno sa povratnim podacima. Optimizujući kompajler na bazi primljenog pravi neke pretpostavke (o tipu) i proizvodi visoko optimizovan mašinski kod.

[40] **Napomena: To ne treba raditi ako se time multiplicira ili poskupljuje kod.**

f) Ako se, u nekoj tački izvršavanja, ispostavi da je neka od pretpostavki kompajlera postala nekorektna, kompajler vrši de-optimizaciju i vraća se na interpreter.

7.3. Dosezanje i zatvaranje

Neophodno je objasniti još dva koncepta da bi se u potpunosti razumelo šta se dešava pri izvršavanju JavaScript programa. To su koncepti ***dosezanje*** (engl. scope) koji ćete sresti i pod nazivom ***opseg vidljivosti*** i koncept ***zatvaranje*** (engl. closure). Oba koncepta odnose se na način na koji JavaScript radi sa varijablama u programu.

7.3.1. Dosezanje

Jedan od osnovnih mehanizama na kome se zasnivaju računarski programi je mehanizam identifikatora. U jezicima računarskog programiranja, identifikator (zove se i simboličko ime) je leksički token[41] koji imenuje entitete jezika. Neke od vrsta entiteta koje identifikator može da označi su varijable, tipovi podataka, oznake, podprogrami i moduli.

Varijabla je u programskom jeziku JavaScript apstraktna lokacija za skladištenje uparena sa pridruženim simboličkim imenom (identifikatorom) koja može da skladišti (sadrži) sve vrste vrednosti koje se javljaju u programu. Uparivanje imena i lokacije skladištenja naziva se ***vezivanje*** (eng. binding). Kada se izvrši vezivanje, sadržaju lokacije skladištenja može se pristupiti putem (navođenjem) imena u kodu programa.

U računarskom programiranju, *opseg vezivanja imena/dosezanje/doseg* (asocijacija imena na entitet kao što je promenljiva) je **deo programa gde je vezivanje imena važeće**, odnosno gde se ime može koristiti za upućivanje na entitet. U praksi se, za većinu programskih jezika, termin „deo programa“ koristi da označi deo izvornog koda (oblast teksta) i naziva se ***leksičko dosezanje***. U nekim jezicima, međutim, termin „deo programa“ označava deo vremena izvršavanja (vremenski period tokom izvršavanja) i poznat je kao ***dinamičko dosezanje***. U delovima programa koji su van dosega, ime se može odnositi na drugi entitet (može imati drugačije vezivanje) ili ni na šta (može biti nevezano). Doseg pomaže u sprečavanju kolizije imena dozvoljavajući da se isto ime odnosi na različite entitete sve dok imena imaju odvojene dosege. Mi ćemo se ovde baviti leksičkim dosezanjem.

U svakom slučaju, koncept dosezanja je u tesnoj vezi sa kontekstom izvršavanja. U delu u kome smo opisivali kontekst izvršavanja kazali smo da JavaScript ima dve osnovne vrste konteksta izvršavanja, globalni i funkcijski i da UVEK POSTOJI JEDAN GLOBALNI KONTEKST a MOŽE DA POSTOJI VIŠE FUNKCIJSKIH KONTEKSTA .

Slično (ali ne baš isto) UVEK POSTOJI **JEDAN GLOBALNI DOSEG** a MOŽE DA POSTOJI **VIŠE DOSEGA KOJI NISU GLOBALNI**.

Trenutna verzija JavaScript-a razlikuje tri[42] vrste dosega:

- Globalni doseg,
- Funkcijski doseg, i
- Blokovski doseg.

Blokovski doseg je relativno nova mogućnost JavaScript-a (naknadno je dodat u jezik) pa ćemo, pre nego što pređemo na detaljno objašnjenje pomenutih dosega, kazati šta je ***blok*** u JavaScript-u.

[41] U računarskom programskom jeziku, termin ***leksički token*** označava stringove (sekvence karaktera) kojima je dodeljeno značenje. Dakle, identifikator je sekvenca karaktera koja imenuje neki entitet jezika.

[42] Moglo bi se reći da, u suštini, postoje **dve** vrste dosega, globalni i blokovski jer sintaksa nalaže da se telo funkcije takođe omeđi vitičastim zagradama

Blok u JavaScript-u je svaki deo koda koji je omeđen vitičastim zagradama `{}`. Primeri su, recimo, `if` naredbe i deklaracije funkcija.

Najkraće rečeno (ima tu još finesa koje su diktirane ključnom reči kojom se entitet deklariše o čemu će kasnije biti reči) tri vrste dosega JavaScript-a mogu se karakterisati na sledeći način.

Globalni doseg je deo memorije u kome se nalaze entiteti (n.pr., varijable, funkcije) koji su deklarisani izvan tela bloka i izvan tela funkcije. Dostupan je iz bilo kog dela JavaScript programa, što znači da se entitetima koji su u njemu može pristupiti iz dela koda koji je izvan bilo kog bloka, ali i iz delova koda koji su u nekom bloku/funkciji.

Funkcijski doseg je deo memorije u kome se nalaze entiteti deklarisani unutar funkcije. Dostupan je iz te funkcije, što znači da se entitetima koji su u njemu može pristupiti iz dela koda koji je unutar (u telu) te funkcije.

Blokovski doseg je deo memorije u kome se nalaze entiteti deklarisani unutar bloka. Dostupan je iz tog bloka, što znači da se entitetima koji su u njemu može pristupiti iz dela koda koji je unutar (u telu) tog bloka.

Među ovim dosezima postoji sledeće uređenje.

Globalni doseg „sadrži“ sve funkcijske i/ili blokovske dosege i ne postoji ni jedan doseg u kome je globalni doseg „sadržan“.

Funkcijski i/ili blokovski dosezi mogu da sadrže druge funkcijske/blokovske dosege, odnosno podržavaju „ugneždavanje“. To znači da, na primer, doseg neke funkcije f u kojoj je deklarisana druga funkcija g, „sadrži“ doseg te funkcije g.

Ova činjenica je ključna za izvršavanje JavaScript koda. Naime, JavaScript okruženje pri izvršavanju dela koda (izraza) u kome se pojavljuje neki entitet (n.pr., varijabla) mora da evaluira taj izraz. Evaluacija izraza zahteva da se pristupi sadržaju na koji pokazuje identifikator entiteta (n.pr., ime varijable). U tom poslu, JavaScript okruženje traži tu varijablu prvo u „najnižem“ lokalnom dosegu. Ako ga tu pronađe, koristi nađeni sadržaj i nastavlja sa izvršavanjem koda. Ako ga ne pronađe u tom dosegu, ide u doseg koji ga „sadrži“ i tamo traži željeni sadržaj. Ovo traženje se nastavlja sve dok se sadržaj ne pronađe ili dok se ne stigne do „najvišeg“ dosega a to je **Globalni doseg**. Ako sadržaja nema ni u Globalnom dosegu, izdaje se greška i izvršavanje koda se zaustavlja.

Ovde namerno nismo navodili primere koda koji bi ilustrovali koncept. Biće ih puno u nastavku kada se budemo bavili sintaksom jezika.

7.3.2. Zatvaranje

Zatvaranje (engl. closure), a srešćete i termine *Closed over Variable Environment* (C.O.V.E), *Persistent Lexical Scope Referenced Data* (P.L.S.R.D) i *backpack*, je izuzetno važno svojstvo jezika JavaScript. To je mehanizam na koji se oslanjaju mnoge važne mogućnosti jezika pa razumevanje JavaScript-a nije moguće bez razumevanja mehanizma zatvaranja.

Kao što je ranije rečeno, JavaScript endžin kontroliše izvršavanja programa putem steka izvršavanja.

Čim se program pokrene, kreira se globalni kontekst izvršavanja i postavlja se na stek. Taj globalni kontekst ima svoju memoriju u koju će da smešta vrednosti nad kojima se operiše. Ta memorija zove se Globalna memorija. Globalni kontekst izvršavanja ostaje na steku sve dok se ne završi izvršavanje celog programa, što znači da je i globalna memorija dostupna do kraja izvršavanja programa.

Pošto je JavaScript jedno-nitna mašina (u jednom trenutku može da izvršava samo jednu sekvencu instrukcija koja se zove *nit izvršavanja*), u tom prvom trenutku nit izvršavanja je globalni kontekst izvršavanja. **Svaki put** kada u kodu **naiđe na poziv funkcije** (ne deklaraciju, nego baš poziv), JavaScript endžin kreira za taj poziv novi kontekst izvršavanja i to funkcijski kontekst sa svojom pripadajućom

lokalnom memorijom i prenosi kontrolu toka (nit izvršavanja) tom novom kontekstu. Kada se završi izvršavanje funkcije (onoga što piše u njenom telu), kreirani funkcijski kontekst zajedno sa pripadajućom memorijom se „briše“ a novi kontekst se postavlja na vrh steka zajedno sa svojom lokalnom memorijom.

Međutim, ima situacija u kojima stanje lokalne memorije pozivajuće funkcije, odnosno funkcije unutar koje je deklarisana pozvana funkcija može biti interesantno i nakon završetka izvršavanja pozvane funkcije. Tu stupa na scenu mehanizam zatvaranja.

Mehanizam zatvaranja u jeziku JavaScript je, u stvari, specifičan način vraćanja rezultata poziva funkcije. Specifičnost se ogleda u tome što se, pri izvršavanju funkcije deklarisane unutar druge funkcije, pored njene povratne vrednosti vraća i sadržaj lokalne memorije pozivajuće funkcije. Zbog toga se kolokvijalno kaže da ***funkcija u JavaScript-u “pamti mesto na kome je rođena“***.

Iskustva kažu da je koncept zatvaranja jedan od koncepata JavaScript-a najtežih za razumevanje programerima. Ovde ćemo navesti samo jedan primer koda kojim se ilustruje zatvaranje a biće ih još puno u nastavku jer se mnogo stvari u JavaScript-u oslanja baš na zatvaranje. Evo primera:

```
spoljna(); // Poziva se funkcija spoljna(), t.j. na stek se postavlja
           // kontekst funkcija spoljna()
function spoljna() {
    let ime = "Ime iz roditeljske funkcije"; /* ime je lokalna varijabla
    funkcije spoljna */
      function ispisiIme() { // Ovo je unutrašnja funkcija koja pravi
                             // zatvaranje

        console.log(ime); // koristi varijablu ime deklarisanu u
                          // roditeljskoj (spoljašnjoj) funkciji
      }
    ispisiIme(); //Uklanja kontekst spoljna() i postavlja svoj kontekst
}
```

U ovom primeru funkcija `ispisiIme()` ispisuje na konzolu vrednost dodeljenu varijabli `ime`, iako je u trenutku pozivanja funkcije `ispisiIme()` lokalna memorija funkcije funkcije `spoljna()` u kojoj varijabla `ime` uklonjena sa steka izvršavanja i na njemu se nalazi lokalna memorija funkcije `ispisiIme()` u kojoj nema varijable `ime`. To je moguće zato što funkcija `ispisiIme()` „zatvara“ nad funkcijom `spoljna()`.

7.4. Stanje programa i deljeno stanje

Računarski program skladišti podatke u varijablama, koje predstavljaju lokacije za skladištenje u memoriji računara. Sadržaj ovih memorijskih lokacija u datom trenutku izvršavanja programa naziva se ***stanje programa***.

Imperativno programiranje je programska paradigma koja opisuje računanje putem stanja programa i naredbi koji menjaju stanje programa. Promene stanja su implicitne (kako su definisane naredbama), njima upravlja mehanizam izvršavanja programa (*runtime*) tako da i izdvojene jedinice izvršavanja kao što su podprogrami, odnosno funkcije imaju pristup promenama stanja koje su izvršili drugi delovi programa, što se u žargonu nazva bočni (željeni ili neželjeni) efekat. Na taj način suočavamo se sa situacijama u kojima se izvan podprograma/funkcija mogu modifikovati neki memorijski sadržaji koje podprogram/funkcija koristi u svom izvršavanju i na taj način izazvati kreiranje neočekivanog izlaznog rezultata (podprogram/funkcija očekuje jednu vrednost a „neko spolja“, bez njenog znanja joj „podmetne“ drugu vrednost)

U deklarativnim programskim jezicima, program opisuje željene rezultate i ne specificira promene stanja direktno. U funkcionalnom programiranju, stanje se obično predstavlja eksplicitno, putem

promenljive stanja (objekta) u svakom koraku izvršavanja programa. Ta promenljiva stanja se kao ulaz prosleđuje funkciji koja transformiše stanje i vraća ažurirano stanje kao deo svoje povratne vrednosti (to je, u stvari, zatvaranje). Čista funkcija "vidi" samo promena stanja u svom dosegu, odnosno dosezima kojima može da pristupi pa se na taj način izbegava nekontrolisano pristupanje sadržaju memorije u toku izvršavanja programa.

Deljeno stanje je svaka promenljiva, objekat ili memorijski prostor koji postoji u deljenom opsegu ili kao svojstvo objekta koji se prenosi između opsega. Deljeni opseg može uključivati globalni opseg ili opsege za zatvaranje.

U objektno orijentisanom programiranju objekti se često dele između dosega putem svojstva dodatog u objekat. Na primer, računarska igra može imati glavni objekat igre, sa objektima likova i predmeta igre uskladištenim kao svojstva glavnog objekta igre. Još jedan izrazit primer je prototipsko povezivanje objekata gde specijalno svojstvo objekta pokazuje na drugi objekat koji je njegov prototip.

Problem sa deljenim stanjem je u tome što se za razumevanje efekata funkcije mora znati cela istorija svake deljene promenljive koju funkcija koristi ili na koju utiče. Navešćemo dva primera da bi smo ovo ilustrovali.

Prvi primer je fenomen "utrkivanja" — veoma česte greške povezane sa zajedničkim stanjem. Zamislite da ste administrator koji ima korisnički objekat koji treba sačuvati. Vaša funkcija `saveUser()` postavlja zahtev API-u na serveru da sačuva taj objekat. Dok se to dešava, korisnik menja svoju sliku na profilu pozivajući funkciju `updateAvatar()` i pokreće drugi zahtev za `saveUser()`. Prilikom sačuvavanja, server vraća kanonski korisnički objekat koji treba da zameni sve što je u memoriji da bi se sinhronizovao sa promenama koje se dešavaju na serveru ili kao odgovor na druge API pozive. Ako se (a to je moguće) drugi odgovor dobije pre prvog odgovora, kada se vrati prvi (sada zastareli) odgovor, nova slika profila se briše u memoriji i zamenjuje starom.

Drugi uobičajeni problem povezan sa deljenim stanjem je što promene u redosledu pozivanja funkcija mogu izazvati kaskadu otkaza ako su funkcije koje deluju na deljeno stanje takve da utiču jedna drugoj na rezultat – imaju međusobno zavisan redosled pozivanja. Evo jednostavnog primera koji to ilustruje:

```
// Deljeno stanje - ovde je ponašanje "spontano"
const x = {
   vrednost: 2
};

// Menja deljeno stanje – modifikuje svojstvo vrednost u objektu x
const x1 = () => x.vrednost += 1;

// Takođe menja deljeno stanje– modifikuje svojstvo vrednost u objektu x
const x2 = () => x.vrednost *= 2;

x1();
x2();
console.log(x.vrednost); // 6
// ako funkcije pozovete ponovo, obrnutim redosledom -
// dobijate ono što ste tražili iako to, možda, niste želeli
x2();
x1();
console.log(x.vrednost); // 13


// Nedeljeno stanje - ovde je ponašanje "disciplinovano"
const x = {
   vrednost: 2
```

```
};

/* Ove funkcije ne vrše mutaciju ulaza. Funkcije inc() i double() vraćaju
NOVE objekte koji imaju svojstvo vrednost */
const inc = x => ({...x, vrednost: x.vrednost + 1});
const double = x => ({...x, vrednost: x.vrednost * 2});

/* Zbog toga što funkcije ne vrše mutaciju, možete ih pozivati koliko god
hoćete puta, bilo kojim redosledom a da ne promenite rezultate drugih poziva
funkcije. */
// Ni jedan poziv ne “dira“ objekat x:
console.log(x);
console.log(inc(double(x)).vrednost); // 5
console.log(x);
console.log(inc(x));
console.log(x);
console.log(double(x));
console.log(x);
console.log(inc(double(x)).vrednost); // 5
```

Funkcionalno programiranje izbegava deljeno stanje — umesto toga ono se oslanja na nepromenljive strukture podataka i “čista računanja“ kojima izvodi nove podatke iz postojećih podataka.

7.5. Sažetak

Ovde treba dodati sažetak poglavlja.

Literatura uz poglavlje 7

[2] Ori Baram, JavaScript Under the Hood: Mastering the Inner Workings, 2017, https://medium.com/@obrm770/javascript-under-the-hood-8cec84bbfd64

[3] Mathias Bynens, Benedict Meurer, JavaScript engine fundamentals: Shapes and Inline Caches, 2018, https://mathiasbynens.be/notes/shapes-ics

[4] Closures, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures