Univerzitet u Sarajevu
Elektrotehnički fakultet Sarajevo
Računarstvo i informatika
Predmet: Web tehnologije

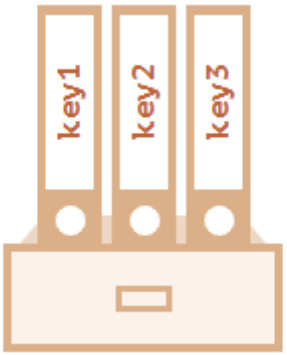
Pripremni materijal za laboratorijsku vježbu br. 4: JavaScript (objekti, nizovi)

1. Objekti

Kao što je rečeno u poglavlju “Tipovi podataka”, JavaScript posjeduje osam tipova podataka. Sedam od njih pripadaju skupini „primitivnih“ tipova podataka, jer njihove vrijednosti sadrže samo jednu “stvar” (bilo da je to string ili broj ili bilo šta drugo). Nasuprot tome, objekti se koriste za pohranjivanje kolekcija različitih podataka i složenijih entiteta. U JavaScript-u objekti predstavljaju jako bitan tip podataka i mogu se pronaći u skoro svakom aspektu ovog programskog jezika.

Objekt se može kreirati korištenjem vitičastih zagrada `{...}` sa opcionalnom listom “svojstava” (eng. *properties*). Svojstvo je par "ključ: vrijednost", gdje je ključ (eng. *key*) string (koji zovemo i "naziv svojstva"), a vrijednost može biti bilo šta.

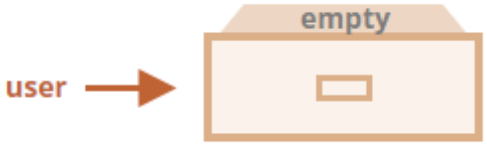
Objekt se može zamisliti kao ormarić sa potpisanim datotekama. Svaki podatak se pohranjuje u svoju datoteku pomoću ključa. Na ovaj način je lako pronaći datoteku po njenom imenu ili je dodati/ukloniti.



Slika 1.1. Ilustracija objekta kao “ormarića” sa datotekama

Prazan objekt (tj. “prazan ormarić”) se može kreirati korištenjem jedne od sljedeće dvije sintakse:

```
let user = new Object(); // sintaksa "konstruktora objekta"
let user = {};  // sintaksa koja koristi "objektni literal"
```



Slika 1.2. Ilustracija praznog objekta (tj. "praznog ormarića") user

Najčešće se koriste prazne vitičaste zagrade `{...}` za kreiranje praznog objekta.

1.1. Literali i svojstva

Moguće je u vitičastim zagradama navesti i određena svojstva u vidu "ključ:vrijednost" parova:

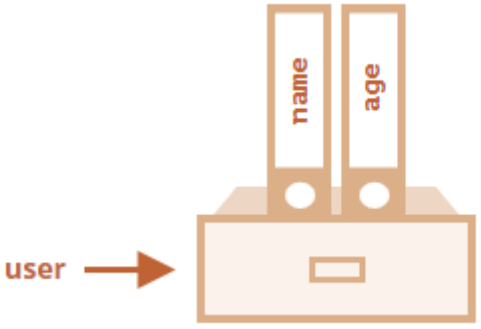
```
let user = {     // objekt
  name: "John",  // kljuc "name" pohranjuje vrijednost "John"
  age: 30        // kljuc "age" pohranjuje vrijednost 30
};
```

Svojstvo ima ključ (također poznat kao "ime" ili "identifikator") koje se navodi prije dvotačke `":"` i vrijednost desno od njega.

U objektu `user` postoje dva svojstva:

1. Prvo svojstvo ima naziv "name" i vrijednost "John".
2. Drugo svojstvo ima naziv "age" i vrijednost 30.

Rezultirajući objekt `user` se može zamisliti kao ormarić sa dvije datoteke koje su označene sa "name" i "age".



Slika 1.3. Ilustracija objekta user kao "ormarića" sa dvije datoteke ("name" i "age")

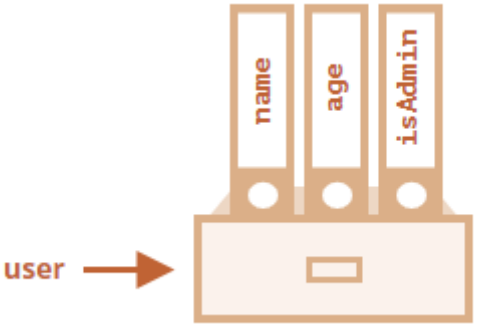
U "ormarić" je moguće dodavati datoteke, brisati ih te čitati njihov sadržaj.

Vrijednostima svojstava se može pristupiti korištenjem operatora tačke (tzv. *dot notation*).

```
// preuzimanje vrijednosti svojstava objekta user
alert( user.name ); // John
alert( user.age ); // 30
```

Vrijednost može biti bilo kojeg tipa. Dodajmo novu vrijednost tipa *boolean*.

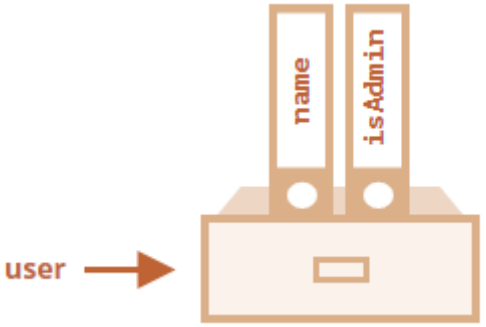
```
user.isAdmin = true;
```



Slika 1.4. Ilustracija dodavanja svojstva "isAdmin" u objekt user kao dodavanje datoteke "isAdmin" u "ormarić"

Za brisanje svojstva, može se koristiti operator `delete`:

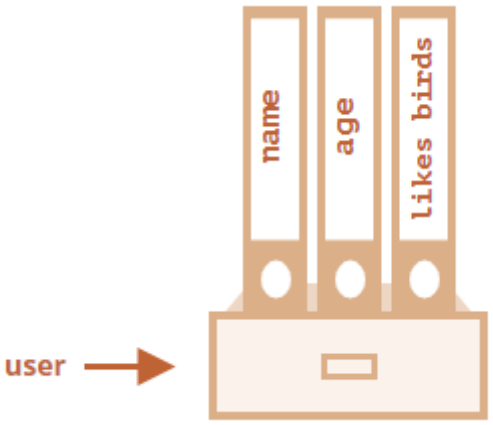
```
delete user.age;
```



Slika 1.5. Ilustracija brisanja svojstva "age" iz objekta user kao brisanje datoteke "age" iz "ormarića"

Moguće je koristiti i nazive svojstava sa više riječi, ali tada se moraju svojstva moraju navesti u sklopu znaka navodnika:

```
let user = {
  name: "John",
  age: 30,
  "likes birds": true  // svojstvo koje sadrzi vise rijeci
};
```



Slika 1.6. Ilustracija dodavanja svojstva "likes birds" u objekt user kao dodavanje datoteke "likes birds" u "ormarić"

Posljednje svojstvo u listi može završiti znakom zareza:

```
let user = {
  name: "John",
  age: 30,
}
```

Ovo se naziva "viseći" zarez (tzv. *trailing* ili *hanging comma*). Olakšava dodavanje/uklanjanje/promjenu svojstava, jer sve linije postaju slične jedna drugoj.

1.2. Uglaste zagrade

Za svojstva koja sadrže više riječi, pristup svojstvima upotrebom operatora tačke ne radi:

```
// rezultuje greskom sintakse
user.likes birds = true
```

JavaScript ne razumije prethodni isječak koda na način na koji bi neki to očekivali. JavaScript misli da se obraćamo `user.likes`, nakon čega javlja sintaksičku grešku kada naiđe na neočekivanu riječ `birds`.

Operator tačka zahtijeva da ključ bude validni identifikator varijable. To znači da: ne sadrži razmake, ne počinje cifrom i ne uključuje posebne znakove (dozvoljeni su jedino znakovi `$` i `_`).

Postoji alternativna notacija koja koristi uglaste zagrade i radi nad bilo kojim stringom:

```
let user = {};

// postavljanje svojstva
```

```
user["likes birds"] = true;

// citanje svojstva
alert(user["likes birds"]); // true

// brisanje svojstva
delete user["likes birds"];
```

Prethodni isječak koda radi ispravno. Obratite pažnju da ispravno navedete string unutar zagrada (bilo koja vrsta navodnika će biti ispravna).

Uglaste zagrade također pružaju način da se dobije naziv svojstva kao rezultat bilo kog izraza (za razliku od sintakse koja koristi string literala). Npr. naziv svojstva se može dobiti iz varijable na sljedeći način:

```
let key = "likes birds";

// isto kao da je pisalo user["likes birds"] = true;
user[key] = true;
```

Ovdje se varijabla `key` može izračunati u vrijeme izvršenja programa ili može zavisiti od korisničkog unosa. Ta varijabla se onda koristi za pristup svojstvu. Ovo kreira veliku dozu fleksibilnosti.

Primjer:

```
let user = {
  name: "John",
  age: 30
};

let key = prompt("What do you want to know about user?", "name");

// pristup putem varijable
alert(user[key]); // John (ako se unese “name”)
```

Notacija koja koristi operator tačke se ne može koristiti na isti način:

```
let user = {
  name: "John",
  age: 30
};
```

```
let key = "name";
alert( user.key ) // undefined
```

1.3. Izračunata svojstva (eng. *computed properties*)

Uglaste zagrade se mogu koristiti u literalu objekta, kada se kreira objekat. Ovo se naziva "izračunata svojstva" (tj. *computed properties*).

Primjer:

```
let fruit = prompt("Which fruit to buy?", "apple");

let bag = {
  [fruit]: 5, // naziv svojstva je preuzet iz varijable fruit
};

alert( bag.apple ); // 5 ako je fruit="apple"
```

Značenje izračunatog svojstva je jednostavno: `[fruit]` znači da naziv svojstva treba uzeti iz varijable `fruit`.

Dakle, ako korisnik unese "apple", `bag` će postati `{apple: 5}`.

U suštini, ovo funkcionira na isti način kao i:

```
let fruit = prompt("Which fruit to buy?", "apple");
let bag = {};

// naziv svojstva je preuzet iz varijable fruit
bag[fruit] = 5;
```

Unutar uglastih zagrada moguće je koristiti i složenije izraze:

```
let fruit = 'apple';
let bag = {
  [fruit + 'Computers']: 5 // bag.appleComputers = 5
};
```

Uglaste zagrade su mnogo moćnije od operatora tačka. One dozvoljavaju bilo koje nazive svojstava i varijabli. Naravno, opširnije su za pisanje.

Zbog svega navedenog, u većini situacija u kojima su nazivi svojstava poznati i jednostavni, koristi se operator tačka. Kada je potrebno nešto složenije, prelazi se na uglaste zagrade.

1.4. Skraćeni način pisanja vrijednosti svojstava

U stvarnom kodu se često koriste postojeće varijable kao vrijednosti za nazive svojstava.

Primjer:

```
function makeUser(name, age) {
  return {
    name: name,
    age: age,
    // ...ostala svojstva (properties)
  };
}

let user = makeUser("John", 30);
alert(user.name); // John
```

U prethodnom primjeru, svojstva imaju ista imena kao varijable. Često se svojstva objekta prave od varijabli zbog čega postoji poseban skraćeni način pisanja vrijednosti svojstva.

Umjesto korištenja `name:name`, može se jednostavno napisati samo varijabla `name` na sljedeći način:

```
function makeUser(name, age) {
  return {
    name, // isto kao da pise name: name
    age,  // isto kao da pise age: age
    // ...
  };
}
```

U istom objektu se mogu koristiti i standardni i skraćeni način pisanja svojstava:

```
let user = {
  name,  // isto kao name:name
  age: 30
};
```

1.5. Ograničenja naziva svojstava

Poznato je da varijabla u JavaScript-u ne može imati naziv koji je isti kao neka od rezervisanih riječi kao što su “for”, “let”, “return” itd. Međutim, kada je riječ o objektima, ne postoji ovo ograničenje:

```
// nazivi svojstava su ispravni
let obj = {
  for: 1,
  let: 2,
  return: 3
};

alert( obj.for + obj.let + obj.return );  // 6
```

Ukratko, nema ograničenja na nazive svojstava. Nazivi mogu biti bilo koji stringovi ili simboli (poseban tip identifikatora, koji u vježbi neće biti detaljnije analiziran). Ostali tipovi se automatski pretvaraju u stringove. Na primjer, broj `0` postaje string `"0"` kada se koristi kao ključ svojstva:

```
let obj = {
  0: "test" // isto kao da pise "0": "test"
};

// oba alerta pristupaju istom svojstvu
// jer se broj 0 pretvara u string "0"
alert( obj["0"] ); // test
alert( obj[0] ); // test (isto svojstvo)
```

Postoji manji problem sa posebnim svojstvom koji ima naziv `__proto__`. Naime, ovo svojstvo se ne može postaviti na vrijednost koja nije objekt:

```
let obj = {};
obj.__proto__ = 5; // dodjela brojcane vrijednosti

// alert ispisuje [object Object]
// vrijednost je objekt, rezultat nije ocekivan
alert(obj.__proto__);
```

Iz prethodnog isječka koda je očito da je ignorirana dodjela tzv. primitivne vrijednosti 5. Inače, `__proto__` svojstvo ima posebnu prirodu koja neće biti detaljnije analizirana u ovoj vježbi.

1.6. Provjera postojanja svojstva, “in” operator

Značajna karakteristika objekata u JavaScript-u, u poređenju sa mnogim drugim jezicima, jeste da je moguć pristup bilo kom svojstvu. Neće se javiti greška ako svojstvo ne postoji! Čitanje nepostojećeg svojstva samo vraća `undefined`. Zbog toga se lako može testirati da li postoji neko svojstvo:

```
let user = {};

// true znaci da ne postoji “noSuchProperty”
alert( user.noSuchProperty === undefined );
```

Postoji i poseban operator `in` kojim se može vršiti provjera da li postoji neko svojstvo objekta. Sintaksa ima sljedeći oblik:

```
"key" in object
```

Primjer:

```
let user = { name: "John", age: 30 };

alert( "age" in user ); // true, user.age postoji
alert( "blabla" in user ); // false, user.blabla ne postoji
```

Obratite pažnju da na lijevoj strani mora biti naziv svojstva. To je obično string unutar navodnika. Ako se izostave navodnici, to znači da varijabla treba da sadrži stvarni naziv koji se testira. Na primjer:

```
let user = { age: 30 };

let key = "age";
alert( key in user ); // true, svojstvo “age” postoji
```

Zašto postoji `in` operator? Zar nije dovoljno uraditi poređenje sa `undefined`? Generalno, u većini situacija poređenje sa `undefined` funkcionira dobro. Ipak, postoji specijalan slučaj kada ovo poređenje ne radi ispravno a operator `in` radi ispravno. Riječ je o situaciji u kojoj svojstvo objekta postoji, ali pohranjuje vrijednost `undefined`:

```
let obj = {
  test: undefined
};

alert( obj.test ); // vrijednost je undefined, dakle nema svojstva?
alert( "test" in obj ); // true, svojstvo postoji!
```

U prethodnom isječku kodu, svojstvo `obj.test` tehnički postoji i operator `in` radi ispravno. Ovakve situacije se dešavaju veoma rijetko, jer `undefined` ne treba eksplicitno dodjeljivati. Praksa je da se koristi `null` za "nepoznate" ili "prazne" vrijednosti. Dakle, `in` operator je egzotičan dodatak u kodu.

1.7. for..in petlja

Za prolazak kroz sve ključeve objekta postoji poseban oblik petlje: `for..in`. Riječ je o potpuno drugačijoj stvari od `for(;;)` konstrukcije koju smo ranije spominjali. Sintaksa je sljedeća:

```
for (key in object) {
  // izvrsava se tijelo petlje za svaki kljuc koji se nalazi u
  // svojstvima objekta
}
```

Na primjer, sljedeći isječak koda ispisuje sva svojstva objekta `user`:

```
let user = {
  name: "John",
  age: 30,
  isAdmin: true
};

for (let key in user) {
  // kljucevi
  alert( key );  // name, age, isAdmin
  // vrijednosti pridruzene svakom kljucu
  alert( user[key] ); // John, 30, true
}
```

Obratite pažnju da sve "for" konstrukcije dozvoljavaju deklariranje varijable petlje unutar petlje, kao što je to urađeno u prethodnom isječku koda sa `let key`. Naravno, može se koristiti i neki drugi naziv varijable umjesto naziva `key`. Na primjer, `for (let prop in obj)` se također često koristi.

1.8. Sortiranost objekata

Jesu li objekti sortirani? Drugim riječima, ako petljom prolazimo kroz objekt, dobivamo li sva svojstva istim redoslijedom kojim su dodana? Možemo li se osloniti na ovo? Kratak odgovor je: "objekti su sortirani na poseban način" pri čemu su cjelobrojna svojstva sortirana, dok se ostala svojstva pojavljuju u onom redoslijedu u kojem su kreirana. U nastavku će biti opisani detalji.

Kao primjer, razmotrimo objekt sa telefonskim kodovima za pojedine države:

```
let codes = {
  "49": "Germany",
  "41": "Switzerland",
  "44": "Great Britain",
  // ..,
  "1": "USA"
};

for (let code in codes) {
  alert(code); // 1, 41, 44, 49
}
```

Objekt se može koristiti da korisniku ponudi popis opcija. Ako razvijamo web stranicu koja je namijenjena većinom za korisnike iz Njemačke, onda vjerojatno želimo da vrijednost `49` bude prva. Međutim, ako pokrenemo kod, vidjet ćemo potpuno drugačiji rezultat:

- USA (1) ide prvi
- zatim Switzerland (41) i tako dalje.

Telefonski kodovi su sortirani u rastućem redoslijedu jer su cijeli brojevi (eng. *integers*). Dakle, u ispisu vidimo `1, 41, 44, 49`.

Cjelobrojna svojstva? O čemu je riječ?

Izraz "cjelobrojna svojstva" u ovom kontekstu označava string koji se može pretvoriti u/iz cijelog broja bez promjene. Dakle, `"49"` je naziv cjelobrojnog svojstva, jer kada se pretvori u cijeli broj i nazad, i dalje ostaje isti. Međutim, `"+49"` i `"1.2"` nisu cjelobrojna svojstva:

```
// Number(...) eksplicitno pretvara vrijednost u broj
// Math.trunc je ugrađena funkcija koja uklanja decimalni dio

// ispisuje "49", isto, cjelobrojno svojstvo
alert( String(Math.trunc(Number("49"))) );

// "49", nije isto kao "+49" ⇒ nije cjelobrojno svojstvo
alert( String(Math.trunc(Number("+49"))) );

// "1", nije isto kao "1.2" ⇒ nije cjelobrojno svojstvo
alert( String(Math.trunc(Number("1.2"))) );
```

S druge strane, ako ključevi nisu cijeli brojevi, tada su navedeni u onom redoslijedu u kojem su kreirani, na primjer:

```
let user = {
  name: "John",
  surname: "Smith"
```

```
};
user.age = 25; // dodavanje novog svojstva

// necjelobrojna svojstva se javljaju u redoslijedu
// u kojem su kreirana
for (let prop in user) {
  alert( prop ); // name, surname, age
}
```

Dakle, kako bismo riješili problem sa telefonskim kodovima, možemo koristiti trik da kodove učinimo necijelim brojevima. Dovoljno je dodati znak plus `"+"` prije svakog koda:

```
let codes = {
  "+49": "Germany",
  "+41": "Switzerland",
  "+44": "Great Britain",
  // ..,
  "+1": "USA"
};

for (let code in codes) {
  alert( +code ); // 49, 41, 44, 1
}
```

2. Nizovi

Objekti omogućavaju pohranjivanje kolekcija podataka u obliku ključ-vrijednost. Ali vrlo često je potrebna neka uređena kolekcija, gdje postoje prvi, drugi, treći element itd. Na primjer, ovo je potrebno za pohranu popisa nečega poput popisa korisnika, robe, HTML elemenata itd.

U ovim situacijama nije zgodno koristiti objekt jer ne nudi metode za upravljanje redoslijedom elemenata. Ne možemo umetnuti novo svojstvo "između" postojećih. Objekti jednostavno nisu namijenjeni za takvu upotrebu.

Postoji posebna struktura podataka pod nazivom `Array`, koja se koristi za pohranjivanje uređenih kolekcija.

2.1. Deklaracija nizova

Postoje dva načina za kreiranje praznog niza:

```
let arr = new Array();
let arr = [];
```

Skoro pa uvijek se koristi drugi način. U okviru uglastih zagrada je moguće navesti inicijalne elemente koje niz posjeduje:

```
let fruits = ["Apple", "Orange", "Plum"];
```

Elementi niza su numerisani, počevši od nule. Element se može dobiti na osnovu broja koji navodimo u uglastim zagradama:

```
let fruits = ["Apple", "Orange", "Plum"];

alert( fruits[0] ); // Apple
alert( fruits[1] ); // Orange
alert( fruits[2] ); // Plum
```

Moguće je promijeniti vrijednost elementa na sljedeći način:

```
fruits[2] = 'Pear';
// sada fruits ima izgled ["Apple", "Orange", "Pear"]
```

Naravno, moguće je dodati i novi element u niz:

```
fruits[3] = 'Lemon';
// sada fruits ima izgled ["Apple", "Orange", "Pear", "Lemon"]
```

Ukupan broj elemenata niza predstavlja njegovu dužinu, tj. `length`:

```
let fruits = ["Apple", "Orange", "Plum"];

alert( fruits.length ); // 3
```

Može se koristiti i `alert` za prikaz cijelog niza:

```
let fruits = ["Apple", "Orange", "Plum"];

alert( fruits ); // Apple,Orange,Plum
```

Niz može pohraniti elemente bilo kojeg tipa. Na primjer:

```
// elementi razlicitih tipova su u istom nizu
let arr = [ 'Apple', { name: 'John' }, true, function() {
alert('hello'); } ];

// izdvajanje objekta koji se u nizu nalazi na indeksu 1
// i prikaz svojstva name tog objekta
alert( arr[1].name ); // John

// izdvajanje funkcije koja se u nizu nalazi na indeksu 3
// i njeno izvršavanje
arr[3](); // hello
```

Zarez na kraju

Niz, baš kao i objekt, može završiti sa znakom zarez:

```
let fruits = [
  "Apple",
  "Orange",
  "Plum",
];
```

Ovaj stil pisanja nizova olakšava dodavanje/brisanje elemenata, jer sve linije koda imaju sličan izgled.

2.2. Metode pop/push, shift/unshift

Red (eng. *queue*) je jedna od najčešćih upotreba niza. U računarstvu, red označava uređenu kolekciju elemenata koja podržava dvije operacije:

- `push` dodaje element na kraj niza.

- `shift` uzima element sa početka i pomjera ga tako da 2. element postaje 1.

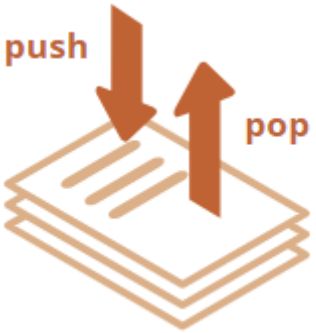
Slika 2.1. Ilustracija metoda push i shift

Nizovi podržavaju obje operacije. U praksi je ovo vrlo često potrebno. Na primjer, red poruka koje se moraju prikazati na ekranu.

Postoji još jedan slučaj upotrebe nizova – struktura podataka pod nazivom stek (eng. *stack*). Podržava dvije operacije:

- `push` dodaje element na kraj.
- `pop` skida element sa kraja.

Dakle, novi elementi se dodaju ili uzimaju uvijek sa "kraja". Stek se obično ilustruje kao špil karata. Nove karte se dodaju na vrh ili uzimaju sa vrha:



Slika 2.2. Ilustracija metoda push i pop

Za stakove, najnovije dodana stavka se uzima prva, što se također naziva LIFO (Last-In-First-Out) princip. Za redove imamo FIFO (First-In-First-Out) pristup.

Nizovi u JavaScript-u mogu raditi i kao red i kao stek. Oni omogućavaju dodavanje/uklanjanje elemenata, kako na/s početka tako i na/s kraja. U računarstvu, struktura podataka koja to dozvoljava naziva se dek (eng. *deque*).

2.2.1. Metode koje rade sa krajem niza

`pop`

Ova metoda uzima posljednji element i vraća njegovu vrijednost:

```
let fruits = ["Apple", "Orange", "Pear"];

alert( fruits.pop() ); // uklanjanje vrijednosti "Pear" i ispis

alert( fruits ); // Apple, Orange
```

`push`

Ova metoda dodaje element na kraj niza:

```
let fruits = ["Apple", "Orange"];

fruits.push("Pear");

alert( fruits ); // Apple, Orange, Pear
```

2.2.2. Metode koje rade sa početkom niza

`shift`

Uzima prvi element niza i vraća njegovu vrijednost:

```
let fruits = ["Apple", "Orange", "Pear"];

alert( fruits.shift() ); // uklanjanje vrijednosti "Apple" i ispis

alert( fruits ); // Orange, Pear
```

`unshift`

Ova metoda dodaje element na početak niza:

```
let fruits = ["Orange", "Pear"];

fruits.unshift('Apple');

alert( fruits ); // Apple, Orange, Pear
```

Metode `push` i `unshift` mogu istovremeno dodati više elemenata:

```
let fruits = ["Apple"];

fruits.push("Orange", "Peach");
fruits.unshift("Pineapple", "Lemon");
```

```
// ispis je ["Pineapple", "Lemon", "Apple", "Orange", "Peach"]
alert( fruits );
```

Generalno, metode `push`/`pop` su brze dok su metode `shift`/`unshift` spore.

2.3. Petlje

Jedan od najstarijih načina za prolazak kroz elemente niza je korištenje brojačke `for` petlje:

```
let arr = ["Apple", "Orange", "Pear"];

for (let i = 0; i < arr.length; i++) {
  alert( arr[i] );
}
```

Međutim, za nizove je moguće koristiti posebnu vrstu petlje, `for..of`:

```
let fruits = ["Apple", "Orange", "Plum"];

// iteracija kroz elemente niza
for (let fruit of fruits) {
  alert( fruit );
}
```

Petlja `for..of` ne daje pristup indeksu trenutnog elementa, već omogućava pristup samo njegovoj vrijednosti, što je sasvim dovoljno u većini slučajeva. Tehnički, pošto su nizovi objekti, moguće je koristiti i `for..in` petlju:

```
let arr = ["Apple", "Orange", "Pear"];

for (let key in arr) {
  alert( arr[key] ); // Apple, Orange, Pear
}
```

Međutim, ovo je zapravo loša ideja. Postoje potencijalni problemi sa ovim pristupom:

1. Petlja `for..in` iterira kroz sva svojstva, ne samo ona numerička. U web preglednicima i drugim okruženjima postoje objekti koji izgledaju kao nizovi (tzv. *array-like* objekti). Oni imaju `length` i indekse kao svojstva, ali mogu imati i druga nenumerička svojstva i metode, koje nam obično nisu potrebne. Međutim, `for..in` petlja će ih izlistati. Dakle, ako treba da raditi sa objektima sličnim nizu, onda ova „dodatna“ svojstva mogu postati problem.
2. Petlja `for..in` je optimizirana za generičke objekte, a ne za nizove, i stoga je 10-100 puta sporija. Naravno, i dalje je veoma brza. Ubrzanje može biti važno samo u uskim grlima. Ali ipak morate biti svjesni razlike.

Generalno, za nizove se ne bi trebala koristiti `for..in` petlja.

2.4. Detaljnije o “length”

Svojstvo `length` se automatski ažurira kada se dese izmjene niza. Preciznije, ovo zapravo nije broj vrijednosti u nizu, već najveći numerički indeks plus jedan. Na primjer, jedan element sa velikim indeksom daje veliku dužinu:

```
let fruits = [];
fruits[123] = "Apple";

alert( fruits.length ); // 124
```

Obratite pažnju da se obično ne koriste ovakvi nizovi.
Još jedna zanimljiva stvar u vezi svojstva `length` je da mu ručno možemo mijenjati vrijednost. Ako ručno povećamo vrijednost svojstva `length`, neće se desiti ništa zanimljivo. Međutim, ako smanjimo vrijednost ovoj svojstva, niz se skraćuje. Proces je nepovratan. Primjer je prikazan u nastavku:

```
let arr = [1, 2, 3, 4, 5];

arr.length = 2; // smanji duzinu niza na vrijednost 2
alert( arr ); // [1, 2]

arr.length = 5; // vracanje duzine na pocetnu
alert( arr[3] ); // undefined: vrijednosti se nisu vratile
```

Zbog ovoga, najlakši način za brisanje vrijednosti niza je korištenjem `arr.length = 0;`.

2.5. new Array()

Postoji i druga sintaksa za kreiranje nizova koja je prethodno spomenuta:

```
let arr = new Array("Apple", "Pear", "etc");
```

Rijetko se koristi, jer je sintaksa koja koristi uglaste zagrade [] kraća. Također, postoji nezgodna karakteristika ove sintakse.

Ako `new Array` pozovemo sa jednim argumentom koji je broj, onda se kreira niz bez elemenata, ali sa datom dužinom:

```
let arr = new Array(2); // da li ce se kreirati niz [2]

alert( arr[0] ); // undefined! nema elemenata
alert( arr.length ); // duzina je 2
```

Kako bi se izbjegla ovakva iznenađenja, obično se koriste uglaste zagrade, osim kada stvarno znamo šta radimo.

2.6. Višedimenzionalni nizovi

Nizovi mogu imati elemente koji su također nizovi. Ovo se može koristiti za pohranjivanje višedimenzionalnih nizova. Na primjer, na ovaj način je moguće pohraniti matrice:

```
let matrix = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];

alert( matrix[1][1] ); // 5, centralni element matrice
```

2.7. Nizove ne treba porediti operatorom ==

Nizovi u JavaScript-u, za razliku od nekih drugih programskih jezika, se ne bi trebali porediti operatorom `==`. Ovaj operator nema poseban tretman za nizove, već radi sa njima kao sa bilo kojim objektima. Pravila su sljedeća:

- Dva objekta su jednaka (u smislu operatora `==`) samo ako su reference na isti objekt.
- Ako je jedan od operanada operatora `==` objekt, a drugi operand je primitivnog tipa, tada se objekt pretvara u primitivni tip (u detalje ove pretvorbe nećemo detaljnije ulaziti).
- …Sa izuzetkom `null` i `undefined` koji su jednaki (u smislu operatora `==`) jedno drugom i ničemu više.

Strogo poređenje `===` je još jednostavnije, jer ne vrši pretvorbu tipova.

Dakle, ako poredimo nizove sa `==`, oni nikada nisu isti, osim ako ne uporedimo dvije varijable koje upućuju na potpuno isti niz.

```
alert( [] == [] ); // false
alert( [0] == [0] ); // false
```

Ovi nizovi su tehnički različiti objekti. Dakle, nisu jednaki. Operator `==` ne vrši poređenje element po element.

Poređenje sa primitivnim tipovima može dati i naizgled čudne rezultate:

```
alert( 0 == [] ); // true

alert('0' == [] ); // false
```

Ovdje se, u oba slučaja, vrši poređenje primitvne vrijednosti sa nizom. Tako se niz `[]`

pretvara u primitivan tip u svrhu poređenja i postaje prazan string `''`. Dalje se proces poređenja vrši na način koji je opisan u prošloj vježbi:

```
// nakon sto se [] pretvori u ''

// ispisuje se true, jer '' biva pretvoreno u broj 0
alert( 0 == '' );

// ispisuje se false, jer nema pretvorbe tipova,
// tj. stringovi su razliciti
alert('0' == '' );
```

Kako onda porediti nizove?
To je jednostavno: nemojte koristiti `==` operator. Umjesto toga, uporedite ih element po element u sklopu petlje ili koristeći neke druge metode za iteraciju kroz elemente niza.