17.03.2025.

Šta je svojstvo (property) u objektu?

Svojstvo u objektu u JavaScript-u predstavlja **par ključa i vrednosti** koji opisuje karakteristike tog objekta. Primer?

```
let osoba = {
    ime: "Marko",      // Svojstvo (key: "ime", value: "Marko")
}
```

JS eksplicitno razlikuje dve vrste svojstava objekta

- ***Sopstvena svojstva*** (svojstva koja su definisana u samom objektu), i
- ***Nasleđena svojstva*** (svojstva koja je objekat nasledio od svojih predaka u prototipskom lancu objekata).

- Ove dve vrste svojstava se tretiraju na različite načine...

Operacije sa objektima

- Operacije sa **objektom kao celinom-Šta se sa objektom može raditi?**
  1. **Kreiranje** objekta: dve sintakse
  2. **Kopiranje** objekta: dva načina: plitko i duboko
- Operacije nad **svojstvima objekta. Koje mogu biti?**
  A. **Pristup** *svojstvu*: dve sintakse, geteri i seteri
  B. **Brisanje/dodavanja** *svojstva*
  C. **Provera postojanja** *svojstva*
  D. **Iteriranje nad** *svojstvima* u objektu

1. Operacije sa objektima: kreiranje objekta$_1$

- Može se kreirati “prazan” i “neprazan” objekat
  - “prazan” objekat : ?

```
{}
```

  - “neprazan” objekat ?:

```
ime_objekta {
    kljuc1 = vrednost1,
    kljuc2 = vrednost2,
    ……
    kljucn = vrednostn,
}
```

- Postoje dve sintakse
  - Literalna
  - Konstruktorska

1.Operacije sa objektima: kreiranje objekta2

- Kreiranje (“praznog”) objekta:
  - **Literalna sintaksa**

```
let korisnik = {};  // "literalna" sintaksa koja se najčešće koristi
```

  - Konstruktorska sintaksa

```
korisnik = new Object(); // „konstruktorska" sintaksa
```

- Kreiranje “nepraznog” objekta
  - literalno

```
let korisnik = { // objekat sa imenom korisnik
  ime: "Petar", // ključ "ime" sa vrednošću „Petar"
  starost: 30 // ključ "starost" sa vrednošću 30
  };
alert(korisnik.ime); //Petar
alert(korisnik.starost); //30
```

- **Kreiranje “nepraznog”objekta : Konstruktorski**

```
let korisnik = new Object(); // objekat imena korisnik
   korisnik.ime;// svojstvo"ime" bez dodeljene vrednosti
   korisnik.starost;  // svojstvo"starost" bez dodeljene vrednosti
   //kako kreirati svojstvo sa dodeljenom vrednošću?
   korisnik.ime = "Petar";// svojstvu"ime" dodeljuje se „Petar"
   korisnik.starost = 30;// svojstvu"starost" dodeljuje se vrednost 30
alert(korisnik.ime);//Petar
alert(korisnik.starost);//30
```

Da li može bez dodeljene vrednosti:

```
korisnik.ime;// svojstvo"ime" bez dodeljene vrednosti
korisnik.starost;  // svojstvo"starost" bez dodeljene vrednosti
```

Može!

A. Objekat: Sadržaj i pristup sadržaju$_1$

- Pricnip rada: U objektu se skladište samo ključevi/imena svojstava, a sami sadržaji se skladište negde “izvan”.
- Pristup sadržaju: opet dve sintakse
  - “**tačkasta**„ – `ime_objekta.ime svojstva`
  - Sintaksa "**uglaste zagrade**" - `ime_objekta["ime_svojstva"]`

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

alert(myObject.a);  // 2 – tačkasta sintaksa
alert(myObject["a"]);  // 2 – Sintaksa "uglaste zagrade"
```

Objekat: Sadržaj i pristup sadržaju$_2$

- Sintaksa [".."] koristi **vrednost** stringa da specificira lokaciju, što znači da sam program može da generiše vrednost tog stringa:

```
var myObject= {a:2,b:33};
var idx;
let wantA= prompt('Želite  li svojstvo  a?',  true)
if (wantA) {
    idx="a"}
else{
     idx= "b"
}
alert(idx);// Ispis?
alert( myObject[idx] ); // Ispis?
```

a ili b

2 ili 33

Zašto je **a** ako ostavimo **true** u promptu?
If (wantA) znači radi u slučaju da je wantA jednako true
Kada je false?
Jedino kada se sve izbriše sa tastaure.
Inače bilo koji tekst (string, broj) je true

Objekat: Sadržaj i pristup sadržaju$_3$

- Ime se može i "sračunati":

```
var prefix = "foo";

var myObject = {
    [prefix +"bar"]:"hello",
    [prefix +"baz"]:"world"
};

alert(myObject["foobar"]);//Ispis?
// hello
alert(myObject["foobaz"]); //Ispis?
// world
```

geteri

- U JavaScriptu, **getteri su** specijalne **metode** objekta koje **omogućavaju čitanje vrednosti svojstava objekta** kao da su obična svojstva, ali omogućavaju dodatnu logiku ili obradu pri čitanju.
- Da biste definisali getter za određeno svojstvo objekta, koristi se

get ključna reč

zajedno sa imenom metode koja predstavlja getter.

- Kada se svojstvo čita, JavaScript automatski poziva getter metodu.

```
Var  person = {
    firstName: 'John',
    lastName: 'Doe',
   // Definicija getter metode za ime
   get fullName() {
        return this.firstName+ ' ' + this.lastName;
    }
};
// Korišćenje gettera
alert(person.fullName); //Ispis?
alert(person.firstName); //Ispis?
alert(person.lastName); //Ispis?
```

John Doe

John

Doe

Objekat: Sadržaj i pristup sadržaju - geteri

Vrlo često je potrebno pribavljanje različitih sadržaja objekta:

```
const obj1 = {
   log: ['a', 'b', 'c'],
   get poslednji() {
       return this.log[this.log.length-1];
   }
};
alert(obj1.poslednji); // Ispis: ?
// Prvi
const obj2 = {
    log: ['a', 'b', 'c'],
    get prvi() {
          return this.log[this.log.length-3];
    }
};
alert(obj2.prvi); // Ispis:?
```

c

a

JS `get` funkcija

- Ponekad je poželjno da se omogući pristup svojstvu koje vraća dinamički sračunatu vrednost.
  - U JS to se postiže sa getters.

    Sintaksa:

    ```
    {get prop() { ... } }
    {get [expression]() { ... } }
    ```

  - `prop` – ime svojstva koje će se vezati za zadatu funkciju koja će sračunati njegovu vrednost.
  - `expression` – Od ECMAScript 2015 mogu se koristiti i izrazi za sračunato ime svojstva za vezivanje za zadatu funkciju.
- Geteri daju mogućnost da se svojstvo *definiše* ali njegovu vrednost *ne sračunavaju* dok se svojstvu ne *pristupi*.
  - Ako se svojstvu nikada ne pristupi, njegova vrednost se ni ne sračunava.

Definisanje getera: novi objekat

- Za nove objekte, u inicijalizatorima objekata

```
const obj = {
  log: ['example','test'],
  get latest() {
    if (this.log.length === 0) return  undefined;
    return this.log[this.log.length - 1];
  }
}
alert(obj.latest); // Ispis?
```

Kako ćemo ispisati dužinu niza LOG?

test

alert(obj.log.length) //ispis?

2

Latest: **ime svojstva koje će se vezati za zadatu funkciju koja će sračunati njegovu vrednost**

Sada sam premestio get van objekta. Znači objekat je definisan i probao sam da uradim get

```
const obj = {
  log: ['example','test']
}
  get latest() {
    if (this.log.length === 0) return  undefined;
    return this.log[this.log.length - 1];
  }

alert(obj.latest); // Ispis?
alert(obj.log.length)
```

Ovo ne radi!!!

Definisanje getera: postojeći objekat

- Znači za postojeće (već definisane) objekte, mora drugačije da se radi. Evo kako:
- pomoću metode `Object.defineProperty()`:

```
const o = {a: 0};  //završeno definisanje objekta

Object.defineProperty(o, 'b', { get: function() {
return this.a + 1; } });

alert(o.b) // pokreće geter koji
      //vraća a + 1 (što je 1)
```

Postavljanje sadržaja objekta: seteri

- Mehanizam **setera** povezuje svojstvo objekta sa funkcijom koja će biti pozvana svaki put kada se pokuša postavljanje tog svojstva.
- JS sintaksa

```
{set prop(val) { . . . }}
{set [expression](val) { . . . }}
```

  Parametri:

  `prop` – ime svojstva za koje se vezuje data funkcija.

  `val` - alias za varijablu koja sadrži vrednost koja se dodeljuje svojstvu `prop`.

  `expression` – Od ECMAScript 2015, moguće je korišćenje izraza za sračunavanje imena svojstva za koje će se vezati data funkcija.

- Seteri se najčešće koriste zajedno sa geterima za kreiranje neke vrste pseoudo-svojstva - svojstva koje nastaje i može mu se pristupiti tek kada je potrebno.

Definisanje setera: nov objekat

```
const language = {
  set current(name) {
    this.log.push(name);
  },
  log: []
}

language.current = 'EN';
alert(language.log); // ['EN']

language.current = 'FA';
alert(language.log); // ['EN', 'FA']
```

Koje je ime setera?
Current

Šta se radi sa :
language.current?
Postavlja vrednost u niz:log

Šta čuva niz :log?
Niz log – čuva istorijat postavljenih vrednosti.

Definisanje setera: postojeći objekat

- Koristi se **defineProperty()**:

```
const o = {a: 0};

Object.defineProperty(o, 'b', {
  set: function(x) { this.a = x / 2; }
});

o.b = 10; //  Pokreće seter
alert(o.a) // Ispis?
```

5

Seter: sračunato ime svojstva

```
const expr = 'foo';

const obj = {
  baz: 'bar',
  set [expr](v) { this.baz = v; }
};

alert(obj.baz); //  Ispis?
"bar„
obj.foo = 'baz'; //  Šta ovo znači?
pokretanje setera
alert(obj.baz); //  Ispis?
"baz"
```

OVO JE BIO GETER:

```
const expr= 'foo';
const obj = {
   get [expr]() { return 'bar'; }
};
alert(obj.foo); //bar
```

OVO JE BIO SETER:

```
const expr = 'foo';
const obj = {
  baz: 'bar',
  set [expr](v) { this.baz = v; }
};
```

Šta se desi u seteru:

1. Expr postaje string :“foo“
2. U objektu obj, se formira baz kao string: „bar“
3. **Setter metodom** čiji naziv je određen dinamički pomoću [expr] se:
4. set [expr](v) { this.baz = v; } je u stvari:
   set foo(v) { this.baz = v; }
5. Drugim rečima, setter foo(v) će promeniti vrednost baz.

```
var o = {
  a: 7,           // ovo je obična vrednost
  get b() {    // ovo je geter
      return this.a+ 1;
  },
   set c(x) {    // ovo je seter
        this.a= x / 2;
   }
};
alert(o.a);     //
o.b                 // pokreće se get b()metodu
alert(o.b);    //
o.c= 50;       //pokreće se funkcija set c(x)
alert(o.a);   //
```

Ispis: 7

Ispis: 8

Ispis: 25

2.čas

B. Brisanje getera i setera

- Koristi se operator `delete`:

```
delete language.current; // briše seter
delete obj.latest; // briše geter
```

Operacije sa objektom: dodavanje i brisanje svojstva$_1$

- **"Tačkasta" sintaksa**

```
// Postavljanje svojstva i njegove vrednosti
    let user = {      // objekat
      name: "John",   // svojstvo "name" vrednost "John"
      age: 30         // svojstvo "age" vrednost 30
    };
    alert( user.name);
    alert( user.age);
//  Brisanje svojstava (operator delete):
    delete user.name
    alert( user.name ); // Ispis?
    undefined
    alert( user.age ); // Ispis?
    30
```

Operacije sa objektom: dodavanje i brisanje svojstva$_2$

- Sintaksa “srednje zagrade”

```
let user1 = {};
alert(user1);  //Ispis:Object
// postavi svojstvo i njegovu vrednost
user1["likes birds"] = true;
// pribavi vrednost svojstva
alert(user1["likes birds"]); // Ispis?
true
// naredba delete briše svojstvo
delete user1["likes birds"];
alert(user1["likes birds"]); // Ispis?
undefined
```

C. Provera postojanja svojstva$_1$

- JS omogućuje da se proveri postojanje svojstva u objektu na tri načina:

1. **`hasOwnProperty()`** metod
2. **`in`** operator
3. Poređenje sa **`undefined`**

Provera postojanja svojstva 2

```
// 1. način - hasOwnProperty() metod
let hero = {
  name: 'Batman'
};
alert(hero.hasOwnProperty('name'));     //Ispis?  true
alert(hero.hasOwnProperty('realName')); //Ispis?  false

// 2. način - in operator
hero = {
  name: 'Batman'
};
alert('name' in hero);     // => true
alert('realName' in hero); // => false

// 3. način - Poređenje sa undefined
hero = {
  name: 'Batman'
};
alert(hero.name);     // Ispis?          Batman
alert(hero.realName); // Ispis?          underfined
```

D.Operacije sa objektom: iteriranje nad svojstvima objekta

- Za prolazak kroz sva svojstva objekta postoji specijalni oblik petlje **`for..in`**.
- Sintaksa

```
for (key in object) {
  // ovo se izvršava za svako svojstvo objekta
}
```

- Primer:

```
let korisnik = {
  ime: "Petar",
  godine: 30,
  jeAdmin: true
};
for (let key in korisnik) {
  alert( key );  // Ispis?
ime, godine, jeAdmin    // Ispiši imena svojstava
alert( korisnik[key] ); // Ispis?
Petar, 30, true   // Ispiši vrednosti svojstava
}
```

2.Operacije sa objektima: kopiranje objekta

- Dve vrste kopiranja: *plitko* i *duboko*
  - ***Plitko kopiranje***: Ne pravi se potpuno nov objekat, već se potpuno kopiraju samo svojstva primitivnog tipa, a za ugnježdene objekte se u kopiju kopiraju reference na postojeći objekat. Operacije nad ugnježdenim objektom se, u stvari, izvršavaju nad originalnim ugnježdenim objektom.
  - ***Duboko kopiranje***: "Pravo" duboko kopiranje (trebalo bi da) pravi kompletnu kopiju svakog objekta na koji naiđe.
  - Implementacija dubokog kopiranja ima raznih i nisu baš sve potpuno dosledne definiciji.
- Jedan dobar izvor informacija o JS kopiranju objekata je https://www.digitalocean.com/community/tutorials/copying-objects-in-javascript

JS plitko kopiranje objekata

- **`Object.assign()`** kopira vrednosti svih **nabrojivih sopstvenih svojstava** jednog ili više izvornih objekata u ciljni objekat. Sintaksa:

```
Object.assign(target, source1);
```

Primer:

```
let obj = {
  a: 1,
  b: 2,
};
let objCopy = Object.assign({}, obj);

alert(`a: ${obj.a}, b: ${obj.b}`); // rezultat?
objCopy.b = 89;
alert(`a: ${objCopy.a}, b: ${objCopy.b}`); // rezultat?
alert(`a: ${obj.a}, b: ${obj.b}`);    // rezultat ?
obj.b = 6;
alert(`a: ${obj.a}, b: ${obj.b}`);    // rezultat ?
alert(`a: ${objCopy.a}, b: ${objCopy.b}`); // rezultat?
```

Zašto nisam naveo ovako?

alert(obj.a + obj.b) – šta je ispis?

alert(obj.a , obj.b) – šta je ispis?

{ a: 1, b: 2 }

{ a: 1, b: 89 }

{ a: 1, b: 2 }

{ a: 1, b: 6 }

{ a: 1, b: 89 }

3 odnosno 1

JS plitko kopiranje: problem

```
let obj = {
  a: 1,
  b: {
  c: 2,
  },
}
let newObj = Object.assign({}, obj); // obj plitko kopiran u newObj
alert(newObj); //Ispis?
obj.a = 10; // a promenjeno u obj, ali nije dodeljeno newObj-u
alert(obj); // Ispis?
alert(newObj); //Ispis?

newObj.a = 20;// a promenjeno u newObj, ali nije dodeljeno obj-u
alert(obj); // Ispis?
alertg(newObj); // Ispis?

let wantC = prompt('Želite li da izmenite svojstvo c u kopiji objekta?',  true)
if (wantC)
newObj.b.c = 30; // c promenjeno u newObj
alert(obj); // Ispis?
alert(newObj); // Ispis?

// Dakle, newObj.b.c = 30 i obj.b.c = 30; Zašto?
```

```
{ a: 1, b: { c: 2} }

{ a: 10, b: { c: 2} }

{ a: 1, b: { c: 2} }
```

```
{ a: 10, b: { c: 2} }

{ a: 20, b: { c: 2} }
```

```
{ a: 10, b: { c: 30} }

{ a: 20, b: { c: 30} }
```

Pojašnjenje:

Kada se koristi Object.assign({}, obj), pravi se **plitka kopija** obj.
To znači:

- Svojstva **prvog nivoa** (a) se kopiraju kao **vrednosti**.
- Svojstva **unutrašnjih objekata** (b) se kopiraju kao **reference**, a ne kao novi objekti.

let newObj = Object.assign({}, obj);

- newObj sada sadrži kopiju obj, ali **b** i dalje pokazuje na **isti objekat** u memoriji

newObj.b.c = 30;

- ❗ ❗ Ovo menja i obj.b.c i newObj.b.c!
- b je bio referenca u oba objekta (obj.b === newObj.b).
- Menjanjem c u newObj.b.c, menjaš isti objekat u memoriji koji koristi obj.
- Finalni rezultat: Zato na kraju oba objekta imaju b.c = 30, jer su obj.b i newObj.b zapravo isti objekat u memoriji.

- Plitko kopiranje –proces kopiranja objekta u novi objekat, pri čemu se samo **prvi nivo osobina kopira, dok se unutrašnje osobine dele između originalnog i kopiranog objekta.**
- Ovo znači da ako originalni objekat sadrži druge objekte kao svoje osobine, kopirani objekat će sadržati referencu na iste objekte, umesto da kreira kopije tih unutrašnjih objekata.

`Object.assign()` kopiranje pristupnog svojstva

```
const obj = {
  foo: 1,
  get bar() {
    return 2;
  },
};

let copy = Object.assign({}, obj);
alert(`foo: ${copy.foo}, bar: ${copy.bar}`)
// Ispis?
// vrednost copy.bar je povratna vrednost getera.
```

{ foo: 1, bar: 2 }

JS duboko kopiranje

- Duboko kopiranje (engl.deep copying) u JavaScriptu se odnosi na proces **stvaranja potpuno nezavisne kopije objekta**, uključujući sve unutrašnje objekte i njihove vrednosti.
- **Ova kopija omogućava da se promene na jednom objektu ne reflektuju na drugi.**
- Razmatrajmo primer gde imamo objekat koji sadrži druge objekte kao svoje osobine. Kod plitkog kopiranja, kada se kopira glavni objekat, unutrašnji objekti ostaju deljeni. **To znači da promene na unutrašnjim objektima u kopiji utiču na original.**
- **Sa dubokim kopiranjem, svaki objekat u strukturi se kopira rekurzivno, što rezultira potpuno nezavisnim kopijama svih unutrašnjih objekata.**

JS duboko kopiranje$_1$

- Korišćenje **JSON.parse(JSON.stringify(object))**

```
let obj = {
  a: 1,
  b: {
    c: 2,
  },
}

let newObj = JSON.parse(JSON.stringify(obj));

obj.b.c = 20;
alert(obj); // Ispis?
alert(newObj); // Ispis?
                          //(newObj netaknut!)
```

{ a: 1, b: { c: 20 } }

{ a: 1, b: { c: 2 } }

Objekat i funkcija

- Svojstva objekta u JS-u mogu da uzmu vrednost bilo kog tipa, pa i tipa `function`.
- Svojstva tipa function odgovaraju metodi objekta.
- Primer:

```
var mojObjekat = {
    a:2,
    b:33,
    c: function double(x){return 2*x},
}
alert(mojObjekat.c(8)) // Ispis?
alert(mojObjekat.c(mojObjekat.b))// Ispis?
```

16

66

JS konverzija objekta u primitive: pravila konverzije

- **Nema** konverzije objekta **u logičku primitivu**.
- Postoje samo konverzije u numerik i string.
- Konverzija u numerik dešava se pri oduzimanju objekata ili primeni matematičkih funkcija.
  - Na primer, `Date` (datumski) objekti mogu se oduzimati; rezultat izraza `date1 - date2` je vreme između dva datuma.
- Konverzija u string dešava se pri ispisivanju objekta funkcijom `alert(obj)` i sličnim kontekstima.

Sun Mar 17 2024 16:28:16 GMT+01

Primer: Sun Nov 11 1962 01:00:00 GMT+01

```
var date1 = new Date(); //    tekući datum
var date2 = new Date('1962-11-11'); //11.11.1962.
// Oduzimanje datuma
var razlika_u_ms= date1 -date2; // broj u msec.
// Pretvaranje rezultata u dana
var razlika_dan= razlika_u_ms/ (1000 * 60 * 60 * 24);
alert(date1);              //Ispis?
alert(date2);              //Ispis?
alert('Razlika u ms između datuma:' + " " + razlika_u_ms);
// Razlika u ms između datuma:   1936020496921
alert('Razlika u danima:' + " " + razlika_dan.toFixed(0)); //22273
```

Podsetnik:

Primitivni tipovi podataka su osnovne (neobjektne) vrednosti koje se koriste za čuvanje jednostavnih podataka. Ima 7 primitivnih tipova pod.:

1. **String**: označava tekstualne podatke.
2. **Number**: Numerička vrednost
3. **Boolean**: Logička vrednost, može biti **true** ili **false**.
4. **Undefined**: označava da promenljiva nije definisana.
5. **Null**: označava odsustvo vrednosti.
6. **Symbol**: koristi se kao ključ u objektima.
7. **BigInt:** brojevi proizvoljne veličine

Primitivni tipovi podataka se ne mogu direktno menjati. Kada i promenite, originalna vrednost ostajene promenjena.

```
var x=10; // Primitivna vrednost tipa Number
var y=x; // Vrednost x se kopira u y
x=20; //Promena x
alert(x); // Output: 20
alert(y); // Output: 10
```

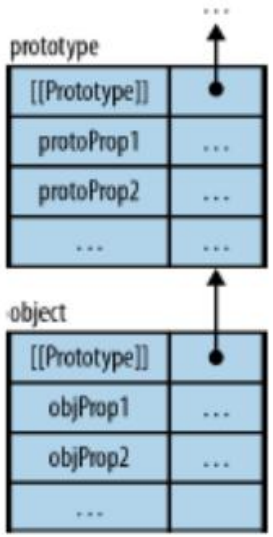
JS konverzija objekta u primitive: kako se odabira konverzija

- Postoje tri varijante konverzije tipa (zovu se “hintovi”) koje se dešavaju u različitim situacijama:
  - **`"string"`** – kada operator očekuje string (n.pr., `alert()`)
  - **`"number"`** – kada operator očekuje numeričku vrednost (n.pr., aritmetičke operacije , poređenja)
  - **`"default"`** – u situacijama kada operator “nije siguran” koji tip očekuje numeričku vrednost (n.pr., binarno + koje može da se primeni na numerik i na string, ili poređenje objekta sa brojem ili simbolom)

Sloj 2: Povezani objekti

Sloj 2: Prototipske relacije među objektima

- Prototipska relacija među objektima bavi se nasleđivanjem: **svaki objekat može da ima drugi objekat kao svoj prototip**.
  - Ako je objekat A prototip objekta B, to znači da objekat B "nasleđuje" **sva svojstva** objekta A.



Objekat A

Objekat B

Sloj 2: svojstvo `[[Prototype]]` i prototipski lanac

- Objekat specificira svoj prototip putem internog svojstva `[[Prototype]]`.
- Svaki objekat ima [[Prototype]] svojstvo koje, naravno može da bude i nula.
- Lanac objekata povezanih svojstvom [[Prototype]] zove se ***prototipski lanac***
- To znači da objekat B i objekat A iz prethodnog objašnjenja formiraju prototipski lanac i da će objekat B specificirati objekat A kao svoj prototip.

Definisanje prototipa i kreiranje objekta na bazi prototipa

- Kreiranje objekta sa zadatim prototipom

  `Object.create(proto, propDescObj?)`

  - `proto` - prototip (neki objekat) objekta koji se kreira
  - `propDescObj` - opcioni parametar; omogućuje dodavanje svojstava kreiranom objektnu putem deskriptora

- Primer

```
// definisanje prototipa
var PersonProto = {
    describe: function () {
        return 'Person named '+this.name;
    }
};
//kreiranje objekta sa definisanim prototipom
var persa = Object.create(PersonProto, {
name: { value: 'Persa', writable: true }
});
alert(PersonProto.describe());// Person name undefined
alert(persa.describe());// Person name Persa
```

Objekat **persa** kreira se iz prototipa **PersonProto**. To znači da će objekat **persa** da nasledi svojstvo **describe** koje ima prototipski objekat **PersonProto**. Pored toga, objektu **persa** se dodaje svojstvo **name** sa vrednošću **'Persa'**koje je i promenljivo (što je specificirano deskriptorom **writable: true**).

Dobavljanje i provera prototipa$_1$

`Object.getPrototypeOf(obj)`

- vraća prototip objekta `obj`

`Object.prototype.isPrototypeOf(obj)`

- proverava da li je primalac metode (direktan ili indirektan) prototip objekta `obj`.
- Preciznije: da li su primalac metode i `obj` u istom prototipskom lancu i da li je `obj` ispred primaoca:

```
var A = {};
var B = Object.create(A);
var C = Object.create(B);
alert(B.isPrototypeOf(C));
alert(C.isPrototypeOf(A));
alert(B.isPrototypeOf(A));
alert(B.isPrototypeOf(C));
alert(C.isPrototypeOf(B));
```

True
False
False
True
false

```
// Dodati alerti za `Object.getPrototypeOf()`
alert(Object.getPrototypeOf(C) === B);
alert(Object.getPrototypeOf(B) === A);
```

Ispis:?

```
// true  jer prototip C je B
// true jer prototip B je A
```

Postavljanje/brisanje svojstva i prototipski lanac

- Postavljanje/brisanje **ignoriše nasleđivanje** i **deluju samo nad sopstvenim (nenasleđenim) svojstvima**.
- Postavljanje svojstva kreira sopstveno svojstvo čak i kada postoji nasleđeno svojstvo sa istim ključem.

```
var proto = { foo: 'a' };
var obj = Object.create(proto);
// obj nasleđuje foo od proto:
alert(obj.foo)
alert(obj.hasOwnProperty('foo'))
// Postavljanje foo daje željeni rezultat:
obj.foo = 'b';
alert(obj.foo)
// Međutim, ovde je kreirano sopstveno svojstvo i nije izmenjen proto.foo:
alert(obj.hasOwnProperty('foo'))
alert(proto.foo)
```

a

False

b

True

a

False: Jer foo nije direktno svojstvo od obj

Deljenje podataka među objektima

- Česte su situacije da više objekata imaju neke zajedničke stvari (svojstva)
- Prototip je zgodan mehanizam da se izbegne situacija u kojoj se iste stvari navode više puta, odnosno za deljenje podataka među objektima.
- To se radi tako što se napravi se da više objekata imaju isti prototip koji u sebi sadrži zajednička svojstva (podatke).

Deljenje podataka: Primer

- **Bez prototipa**

```
var jane = {
  name: 'Jane',
  describe: function () {
       return 'Person named  ' +          this.name;
  }
};
var tarzan = {
  name: 'Tarzan',
  describe: function () {
    return 'Person named  ' +     this.name;
  }
};
alert(jane.describe());
alert(tarzan.describe());
```

Person named Jane

Person named Tarzan

Sa prototipom

```
var PersonProto = {
  describe: function () {
    return 'Person named ’ + this.name;
  }
};
var jane = Object.create(PersonProto, {
name: { value: 'Jane', writable: true }
});

var tarzan = Object.create(PersonProto, {
name: { value: 'Tarzan', writable: true }
});

// Rezultat je sledeći:
alert(jane.describe())
alert(tarzan.describe())
```

Person named Jane

Person named Tarzan

Sloj 3: Konstruktor objekta i nasleđivanje među konstruktorima

Sloj 3: Konstruktor

- Konstruktorska funkcija (konstruktor) je pomoć pri kreiranju **objekata koji su slični**.
- Objekti koje konstruktor kreira zovu se **instance**.
- Konstruktor je, u stvari, normalna funkcija, ali se zbog svoje prirode imenuje, postavlja i poziva na drugačiji način.
  - Po konvenciji, imena konstruktora započinju velikim slovom, dok imena normalnih funkcija započinju malim slovom.
  - Prvo se postavlja ponašanje (prototip) a zatim podaci (objekat)
  - Konstruktori se pozivaju putem ključne reči new
- Konstruktori u JavaScript-u odgovaraju klasama u drugim jezicima.
- Iskoristićemo primer deljenja podataka da ilustrujemo konstruktor

Konstruktor: primer

- Sama funkcija (**Person**) će postavljati podatke (deo 1):

```
function Person(name) {
    this.name = name;
}
```

Deklariše funkciju Person koja se koristi kao **konstruktor** za kreiranje novih objekata tipa Person.

- Objekat `describe` u `Person.prototype` postaje prototip svih instanci `Person`. On će postavljati ponašanje (deo 2):

```
Person.prototype.describe = function () {
    return 'Person named ' + this.name;
};
```

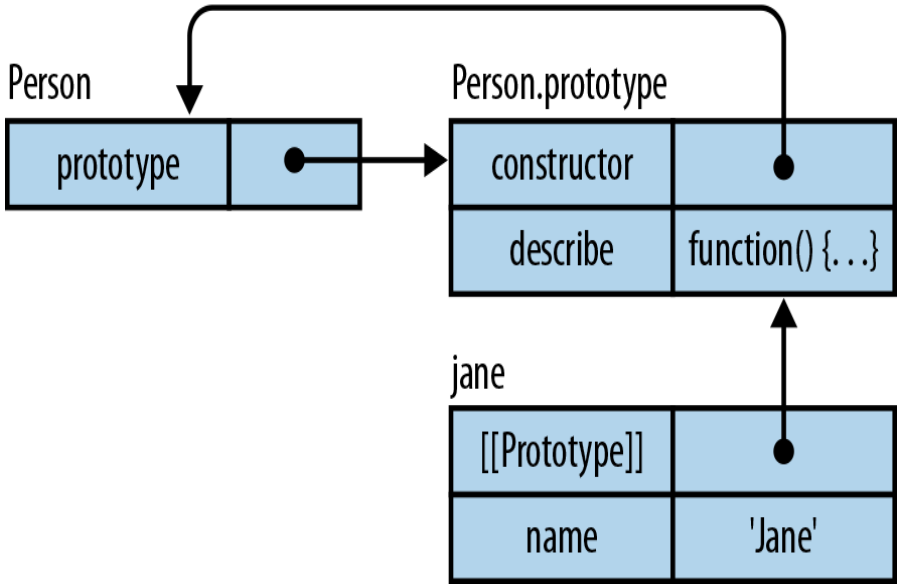
- Primer koji kreira i koristi instancu Person:

```
var jane = new Person('Jane');
jane.describe()
alert(jane.describe()) //Ispis?
'Person named Jane'
```

Ovaj kod **dodaje metodu describe u prototip Person**, što znači da će sve instance Person moći da koriste ovu metodu.

Kreira novu instancu objekta Person sa imenom "Jane" i onda poziva metodu describe()

Kako izgleda instanca `jane`



- **`Jane`** je instanca konstruktora **`Person`** čiji je prototip objekat **`Person.prototype`**
- Svojstvo **`constructor`** objekta **`Person.prototype`** pokazuje nazad na konstruktor

Implementiranje konstruktora: koristiti striktni režim

- Striktni režim: zaštita od zaboravljanja ključne reči new
- Ako se, pri korišćenju konstruktora, ne navede ključna reč new, poziva se obična funkcija, a ne konstruktor.
- Ne kreira se instanca, već se kreira globalna varijabla. Naravno, bez ikakvog upozorenja.

Implementiranje konstruktora: primer

```
// Primer zaboravljenog new:
function SloppyColor(name) {
  this.name = name;
}
//ne-striktni režim
var c = SloppyColor('green');
// nema upozorenja da je izostavljena ključna reč new!
// Ne kreira se instanca:
alert(c); // Ispis?
// Kreira se globalna varijabla:
  alert(name); // Ispis?
```

Undefined- jer `SloppyColor` ne vraća ništa)

```
// Striktni režim generiše izuzetak:
function StrictColor(name) {
'use strict';
  this.name = name;
}
var c = StrictColor('green');          // a trebalo je  var c = new StrictColor('green');
// TypeError: Cannot set property 'name' of undefined
```

green

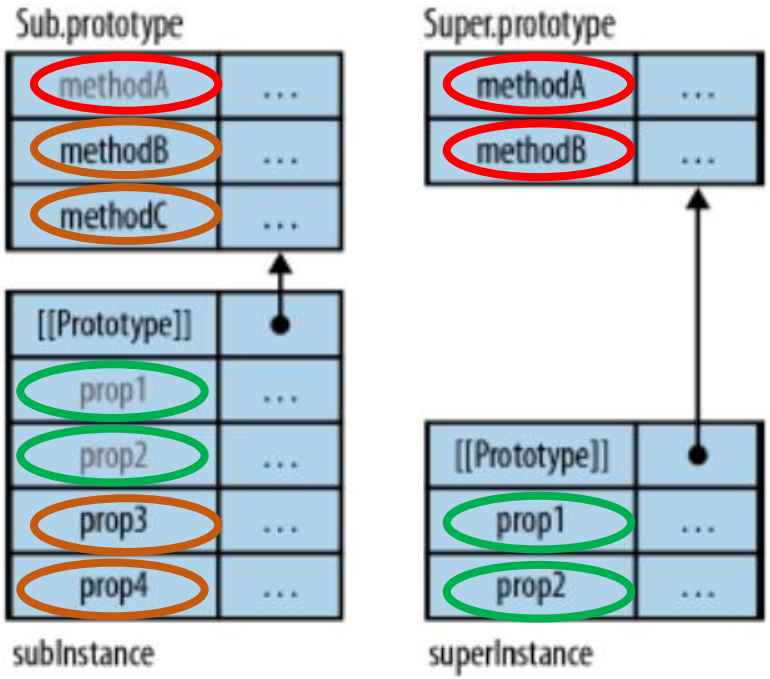
U **strict mode-u**, ako zaboraviš new, **this postaje undefined**, pa this.name = name; baca **TypeError**.

Sloj 3: Nasleđivanje među konstruktorima

- Zadatak: za zadati konstruktor `Super` napisati novi konstruktor Sub, takav da ima sve mogućnosti koje pruža `Super`, plus neke sopstvene mogućnosti.
- Nas žalost, JavaScript nema za ovo ugrađeni mehanizam.
- Naravno, to se može realizovati manuelno.

Nasleđivanje među konstruktorima : Osnovna ideja

- Subkonstruktor Sub bi trebao da ima sva svojstva koja ima Super (i svojstva prototipa i svojstva instance) i da tome doda sopstvena svojstva:



Podsetnik:

- **Instanca je konkretan objekat koji je kreiran iz neke klase**

```
function Osoba(ime) {
   this.ime = ime;
}
```

- const osoba1 = new Osoba("Marko"); // osoba1 je instanca klase Osoba
- **Prototip je objekat koji sadrži zajednička svojstva i metode koje instance mogu koristiti**

```
Osoba.prototype.pozdravi = function() {
   return `Zdravo, ja sam ${this.ime}`;
};
alert(osoba1.pozdravi()); // "Zdravo, ja sam Marko„
```

- **Metoda pozdravi se ne nalazi direktno u osoba1**, već se pretražuje u **prototipu** (Osoba.prototype).

Nasleđivanje svojstava instance

- Svojstva instance se postavljaju u samom konstruktoru pa nasleđivanje svojstava instance super-konstruktora zahteva pozivanje tog konstruktora:

```
function Sub(prop1, prop2, prop3, prop4) {
    Super.call(this, prop1, prop2); // (1) Poziva konstruktor nadklase (Super) i dodeljuje prop1 i prop2 instanci Sub.
    this.prop3 = prop3; // (2) Dodaje prop3 direktno instanci Sub
    this.prop4 = prop4; // (3)
}
```

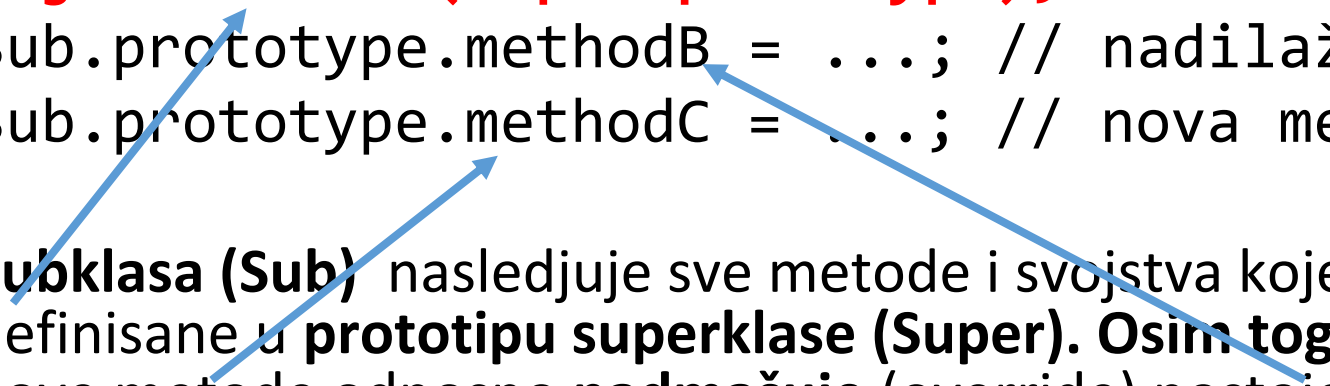
- Ovaj kod pokazuje **nasleđivanje u JavaScript-u** koristeći konstruktorsku funkciju Sub, koja poziva **nadklasu (superklasu) Super** pomoću Super.call(this, prop1, prop2).

3.čas

Nasleđivanje svojstava prototipa

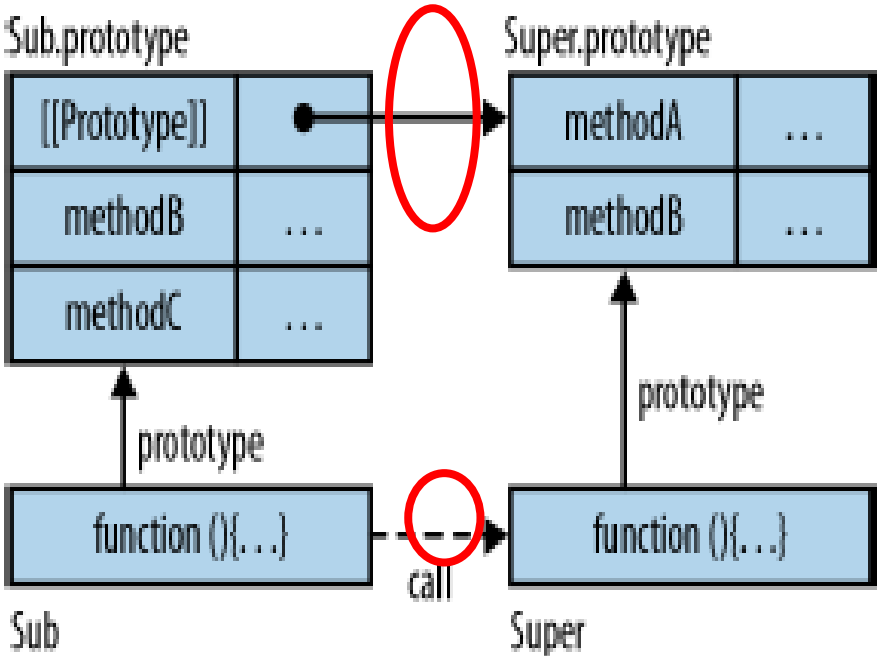
- Kako da **`Sub.prototype`** nasledi sva svojstva od **`Super.prototype`**?
- **Rešenje**: da se objektu **`Sub.prototype`** dodeli kao prototip objekat **`Super.prototype`**:

```
Sub.prototype =
Object.create(Super.prototype);
Sub.prototype.methodB = ...; // nadilaženje
Sub.prototype.methodC = ...; // nova metoda
```



Subklasa (Sub) nasledjuje sve metode i svojstva koje su definisane u **prototipu superklase (Super). Osim toga** dodaje i nove metode odnosno **nadmašuje** (override) postojeće metode (ako metoda methodB postoji i u Super prototipu.

Kako su povezani **Sub** i **Super**



- Konstruktor **Sub** nasleđuje konstruktor **Super** tako što ga poziva i što podesi da **Super.prototype** bude prototip za **Sub.prototype** .

Nadilaženje metode

- Metod u `Super.prototype` se nadilazi tako što se **dodaje metod sa istim imenom** u `Sub.prototype`.
- `methodB` je primer:
  - pretraga za metodom **`methodB`** započinje u objektu **`subInstance`** i u njemu pronalazi **`Sub.prototype.methodB`** pre nego što naiđe na **`Super.prototype.methodB`**.

Super-poziv: ideja

- Super-poziv treba da omogući da se pozove redefinisana metoda iz roditeljske klase.
- Super-poziv metode (recimo da se zove **foo**) sastoji se iz tri koraka :

1. Započeti pretragu “nakon” (u prototipu) osnovnog objekta tekuće metode:
   - **JavaScript prvo traži metodu foo u prototipu podklase (subklase).**
   - Ako metoda nije pronađena u podklasi, JavaScript traži metodu u **prototipu superklase** (roditeljske klase).

2. Naći metodu sa imenom **foo**.
   - Ako je metoda foo definisana u prototipu superklase, JavaScript će je pronaći i pozvati.
   - Ako nije, JavaScript će baciti grešku, jer metoda nije pronađena.

3. Pozvati metodu sa tekućim **this**.

   Ovo omogućava da **metoda podklase pozove metodu roditeljske klase**, čak i ako je metoda u podklasi **nadišla** (overridden) metodu iz roditeljskog objekta.

Primer nasleđivanja konstruktora

- Sada ćemo pokazati nasleđivanje konstruktora na primeru.
- Primer je sledeći:
  - Imamo neki konstruktor `Osoba`
  - Želimo da kreiramo konstruktor `Zaposleni` kao subkonstruktor konstruktora `Osoba`.

Konstruktor **Osoba**

```
function Osoba(ime) {
    this.ime = ime;
}
Osoba.prototype.opisi = function () {
    return ' Osoba zvana '+this.ime;
};
```

Konstruktor Zaposleni

```
function Zaposleni(ime, titula) {
Osoba.call(this, ime);//svojstvo ime od osobe se dodaje
   this.titula = titula;//novo svojstvo za objekat Zaposleni
}
Zaposleni.prototype = Object.create(Osoba.prototype);
                //omogućava nasledjivanje metoda iz osobe u Zaposleni
Zaposleni.prototype.constructor = Zaposleni;
                // zaposleni je konstruktor
Zaposleni.prototype.opisi = function () {
   return Osoba.prototype.opisi.call(this)+
   '('+this.titula+')';
};       //pozivamo metodu Opisi iz Super klase Osoba
// Evo interkacije:
var pera = new Zaposleni('Pera', 'Ložač');
alert(pera.opisi()) // Ispis?:
alert(pera instanceof Zaposleni) //Ispis?:
alert(pera instanceof Osoba) //Ispis?:
```

Osoba zvana Pera (Ložač)

true

true

Niz

Struktura podataka: niz (`Array`)

- Tip **`Array`** je specijalni tip objekta koji služi za skladištenje **uređenih kolekcija vrednosti** gde se pojedinačnim elementima kolekcije može pristupati putem pozicije (indeks).
- **Početna vrednost indeksa je 0**.
- **Ukupan broj elemenata u nizu čuva se u svojstvu `length`** što predstavlja vrednost poslednjeg numeričkog indeksa uvećanu za 1
- Ograničenja na indekse i svojstvo `length`
  - Indeksi su brojevi iz intervala $0 \leq i < 2^{32}-1$.
  - Maksimalna vrednost za `length` je $2^{32}-1$.
- Deklarisanje – da bi se nešto smatralo nizom, potrebno je uraditi deklaraciju/proglašavanje. Postoje dve sintakse za deklarisanje/kreiranje praznog niza:

  `let arr = [];` - sintaksa uglaste zagrade najčešće se koristi

  `let arr = new Array();` - ključna reč **`new`** retko se koristi

Deklarisanje i dodela: sintaksa uglaste zagrade

- Dodela vrednosti elementima može da se radi u deklaraciji:

```
let fruits = ["Apple", "Orange", "Plum"];
alert (fruits.length)// Ispis?
alert (fruits) // Ispis?
```

3

Apple Orange Plum

- Dodela vrednosti elementima može da se radi i van deklaracije

```
let fruits = ["Apple", "Orange", "Plum"];
fruits [3] = "Grape"
alert (fruits.length) // Ispis?
alert( fruits); // Ispis?
fruits [9] = "Pear"
alert (fruits.length) // ?
alert( fruits);
// Apple Orange Plum Grape ,,,,,,Pear – niz sa
// “rupama”
```

4

Apple Orange Plum Grape

10

Deklarisanje niza: ključna reč **new**

- Sintaksa je:

```
let arr = new Array("Apple", "Pear", "etc");
alert( arr[0] ); // Ispis?
alert( arr.length ); // Ispis?
```

Apple

3

- Sintaksa se retko koristi, jer ima "škakljivo" svojstvo:

```
let arr = new Array(2); // da li će kreirati [2] ?
alert( arr[0] ); // undefined! Nema elemenata.
alert( arr.length ); // a dužina je 2
```

- Ako nije number, radi uredno:

```
let arr = new Array("2"); // da li će kreirati ["2"] ?
alert( arr[0] ); // 2! Ima element koji treba.
alert( arr.length ); // a i dužina je dobra 1
```

Struktura podataka **Array** : operacije$_1$

- Kraj niza

```
let fruits = ["Apple", "Orange", "Pear"];
alert( fruits ); // Apple, Orange, Pear
  fruits.pop();
  alert( fruits ); // Ispis?
```

```
Apple, Orange
uklonio krajnji: ("Pear")
```

- **push(...stavke)** dodaj na kraj.
- **pop()** ukloni sa kraja.

```
  fruits.push ("Plum")
     alert( fruits ); // Ispis?
```

```
Apple, Orange, Plum
dodao na kraj ("Plum")
```

Struktura podataka **Array** : operacije$_2$

- Početak niza

```
let fruits = ["Orange", "Pear"];
alert( fruits ); // Orange, Pear
fruits.unshift('Apple');
alert( fruits ); // Ispis?
Apple, Orange, Pear
dodao na početak (Apple)
```

unshift(...stavke) dodaj na početak

```
let fruits = ["Apple", "Orange", "Pear"];
alert( fruits ); // Apple, Orange, Pear
fruits.shift();
alert( fruits ); // Ispis?
Orange, Pear
ukloni početni (Apple)
```

shift() ukloni sa početka:

Struktura podataka **`Array`** : operacije$_3$

- Brisanje elementa: **`splice`** i **`slice`**
- Spajanje nizova: **`concat`**
- Iteriranje nad nizom: **`forEach`**
- Pretraživanje u nizu:
  - **`arr.indexOf`**, **`arr.lastIndexOf`** , **`arr.includes`** , **`arr.filter(fn)`**.
- Transformisanje niza:
  - **`arr.map()`**, **`sort(fn)`**, **`arr.reverse()`**, **`split`** i **`join`**, **`arr.reduce`** i **`arr.reduceRight`**
- Identifikacija tipa: **`Array.isArray()`**

Array : metoda **splice**$_1$

- umetanje, uklanjanje i zamena elemenata.
- Sintaksa:

```
arr.splice(index[, deleteCount, elem1, ..., elemN])
```

- Semantika:

arr je ime niza na koji se primenjuje metoda

Počinjući od pozicije **index** uklanja **deleteCount** elemenata (ako je **deleteCount = 0**, ne uklanja ništa) i zatim umeće elemente **elem1, ..., elemN** na njihovo mesto. Vraća niz uklonjenih elemenata.

Array : metoda splice$_2$

Primer 1 (uklanjanje i zamena elemenata niza):

```
let arr = ["Ja", "učim", "JavaScript", "baš", "sada"];
arr.splice(0, 3, "Hajde", "da igramo");
alert( arr ) // Ispis?
["Hajde", "da igramo", "baš", "sada"]
// uklanja tri prva elementa i zamenjuje sa druga dva
```

Primer 2 (uklanjanje i prikaz niza uklonjenih):

```
let arr = ["Ja", "učim", "JavaScript", "baš", "sada"];
let removed = arr.splice(0, 2);
alert( removed ); // Ispis?
["Ja","učim"] <-- niz uklonjenih
// uklanja 2 prva elementa
```

concat

- Primer:

```
const niz1 = [1, 2, 3];
const niz2 = [4, 5, 6];

const spojeniNiz = niz1.concat(niz2);
alert(spojeniNiz); // Ispis?
```

[1, 2, 3, 4, 5, 6]

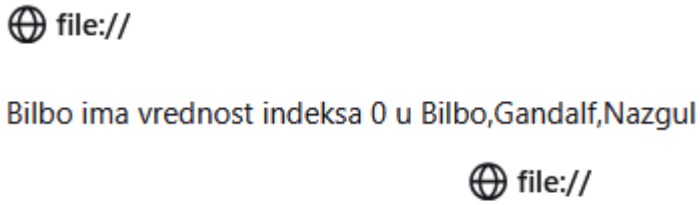
Iteriranje nad nizom: metoda `arr.forEach`

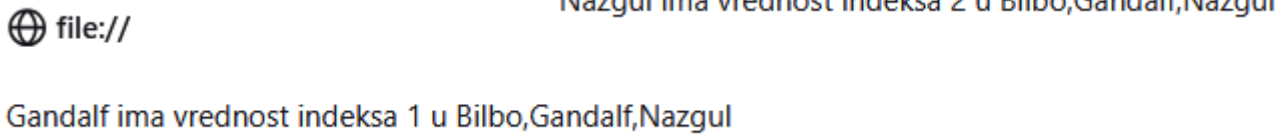
- Omogućuje pokretanje funkcije `function` nad svakim elementom niza.
- Sintaksa:

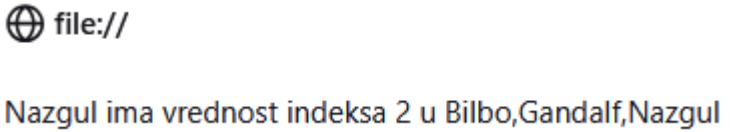
```
arr.forEach(function(item, index, array) {
  // ... Uradi nešto sa item
});
```

- Primer:

```
["Bilbo", "Gandalf", "Nazgul"].forEach((item,
index, array) => {
  alert(`${item} ima vrednost indeksa ${index} u
${array}`);
});
// Šta će da ispiše u alert boksu?
```







Transformisanje niza: metoda `arr.map`

- Sintaksa

```
let result = arr.map(function(item, index, array) {
  // vraća novu vrednost umesto item
});
```

- Semantika: Funkcija `function` se primenjuje na elemente niza `arr` redom i ta vrednost se smešta u odgovarajući element novog niza `array`.
- Primer (transformiše elemente na broj karaktera u elementu):

```
let lengths = ["Bilbo", "Gandalf", "Nazgul"].map(item =>
item.length);
alert(lengths); // Ispis?
```

5,7,6

Kako prebrojati broj slova u nizu?

```
arr=["Bilbo"]
alert(arr.length); // Ispis?
```

0

```
alert(arr[0].length); // Ispis?
```

5

Transformisanje niza: metoda `arr.reduce/arr.reduceRight`

- Sintaksa

```
let value = arr.reduce(function(accumulator, item,
index, array) {
  // ...
}, [initial]);
```

- Semantika: Funkcija `function` se primenjuje na sve elemente niza redom, "noseći" svoj rezultat u sledeći poziv putem parametra `accumulator`.
- Primer (sabira elemente):

```
let arr = [1, 2, 3, 4, 5];
let result = arr.reduce((sum, current) => sum + current,
0);
alert(result); // Ispis?
15
```

- **(sum, current) => sum + current** je funkcija koja dodaje tekući broj (current) na akumuliranu sumu (sum).
- **Početna vrednost (0)** se koristi kao inicijalna suma.

`arr.reduce`: Izvršavanje

	sum	current	result
Prvi poziv	0	1	1
Drugi poziv	1	2	3
Treći poziv	3	3	6
Četvrti poziv	6	4	10
Peti poziv	10	5	15

Detalje o `array` metodama možete naći na https://javascript.info/array-methods

Asocijativni niz (Map)

Asocijativni niz i primene

U računarstvu, ***asocijativni niz*** (**mapa**) je apstraktni tip podataka koji skladišti kolekciju parova (ključ, vrednost) tako da se svaki ključ pojavljuje najviše jednom u kolekciji.

```
// Kreiranje prazne Map kolekcije
let mapa = new Map();
// Dodavanje novih parova (ključ, vrednost)
mapa.set("ime", "Marko");
mapa.set("prezime", "Petrović");
mapa.set("godine", 30);
// Ispis mape nakon dodavanja parova
alert(mapa.get("ime"));    // Ispis?
// Dodavanje postojećeg ključa - prepisuje vrednost
mapa.set("ime", "Nikola");
// Provera nove vrednosti
alert(mapa.get("ime")); // Ispis?
// Ispis svih ključeva i vrednosti
mapa.forEach((vrednost, kljuc) => {
   alert(`${kljuc}: ${vrednost}`);
});
```

Marko

Nikola

ime:Nikola

prezime:Petrović

godine:30

JS: Ugrađeni objekat Map

- JS nema tip `map`, već se za to koristi poseban objekat `Map`.
- U JS-u **`Map` objekti su kolekcije parova ključ/vrednost**.
- **Ključevi u `Map` objektu moraju biti jedinstveni**
- Kroz parove ključ-vrednost objekta Map se iterira pomoću **`forEach` (a može i `for...of`) petlje pri čemu se vraća dvočlani niz `[ključ, vrednost]` za svaku iteraciju.**
- Iteracija se dešava po redosledu umetanja koji odgovara redosledu kojim je svaki par ključ-vrednost prvi put umetnut u mapu metodom `set()`.

- Primer:

```
// Kreiranje prazne mape
const myMap = new Map();

// Dodavanje elemenata u mapu
myMap.set("ime", "Marko");
myMap.set("godine", 30);
myMap.set(1, "Broj kao ključ");
myMap.set(true, "Boolean kao ključ");
```

- **Primer:**

```
// Dohvatanje vrednosti iz mape
alert(myMap.get("ime")); // Ispis?
alert(myMap.get("godine")); // Ispis?
alert(myMap.get(1)); // Ispis?
alert(myMap.get(true)); // Ispis?
// Provera da li ključ postoji
alert(myMap.has("ime")); // Ispis?
alert(myMap.has("prezime")); // Ispis?

// Brisanje ključa iz mape
myMap.delete("godine");

// Veličina mape (broj elemenata)
alert(myMap.size); // Ispis?

// Iteracija kroz mapu
myMap.forEach((value, key) => {
   alert(`${key}: ${value}`);
});
//ispis?
```

Marko
30
Broj kao ključ
Boolean kao ključ

True
False

3

Ime:Marko
1:Broj kao ključ
True: Boolean kao ključ

Iteriranje pomoću forEach

```
const mojaMapa = new Map();
mojaMapa.set(0, 'nula');
mojaMapa.set(1, 'jedan');
mojaMapa.forEach((value, key) => {alert(`${key} =
${value}`);});
// 0 = nula
// 1 = jedan
mojaMapa.set(1, ' bilo šta na poziciji 1');
mojaMapa.set(0, ' bilo šta na poziciji 0');
mojaMapa.forEach((value, key) => {alert(`${key} =
${value}`);});
// 0 = bilo šta na poziciji 0
// 1 = bilo šta na poziciji 1
```

Iteriranje pomoću `for of`

```
const mojaMapa = new Map();
mojaMapa.set(0, 'nula');
mojaMapa.set(1, 'jedan');
for (const [key, value] of mojaMapa) {
     alert(`${key} = ${value}`);
   }
// 0 = nula
// 1 = jedan
mojaMapa.set(1, ' bilo šta na poziciji 1');
mojaMapa.set(0, ' bilo šta na poziciji 0');
for (const [key, value] of mojaMapa) {
     alert(`${key} = ${value}`);
   }
// 0 = bilo šta na poziciji 0
// 1 = bilo šta na poziciji 1
```

Poredjenje **for of** sa **for.Each**:

```
mojaMapa.forEach((value, key)
=> {alert(`${key} =
${value}`);});
```

```
mojaMapa.forEach((value, key)
=> {alert(`${key} =
${value}`);});
```

Kloniranje i spajanje

```
const original = new Map([
  [1, 'jedan'],
]);

const clone = new Map(original);

alert(clone.get(1)); // ispis?
```

jedan

```
alert(original.get(1)); // ispis?
```

jedan

```
alert(original === clone); // ispis?
```

false

original i clone nisu isti objekti u memoriji – oni su dve odvojene instance Map objekta, iako imaju iste vrednosti.

Skup (Set)

Set tip

- Set (skup) je apstraktni tip podataka koji može da skladišti jedinstvene vrednosti, bez ikakvog posebnog redosleda.
  - Za razliku od većine drugih tipova kolekcija, umesto preuzimanja određenog elementa iz skupa, **obično se proverava da li je vrednost u skupu** (članstvo u skupu).
- **Statički skupovi dozvoljavaju samo operacije upita nad svojim elementima** — kao što je provera da li je data vrednost u skupu ili nabrajanje vrednosti u nekom proizvoljnom redosledu.
- **Dinamički dozvoljavaju umetanje i brisanje elemenata iz skupa.**

Operacije algebre skupova

- **`union(S,T)`**: vraća uniju skupova S i T.
- **`intersection(S,T)`**: vraća presek skupova S i T.
- **`difference(S,T)`**: vraća razliku skupova S i T.
- **`subset(S,T)`**: proverava da li je skup S podskup skupa T.

```
// Primeri skupova
const S = new Set([1, 2, 3, 4]);
const T = new Set([3, 4, 5, 6]);

// Testiranje funkcija
console.log("Unija:", union(S, T)); // {1, 2, 3, 4, 5, 6}
console.log("Presek:", intersection(S, T)); // {3, 4}
console.log("Razlika (S - T):", difference(S, T)); // {1, 2}
console.log("Razlika (T - S):", difference(T, S)); // {5, 6}
console.log("Da li je S podskup T?:", subset(S, T)); // false
```

- Primer1:

```
const S = new Set([1, 2, 3, 4]);
const T = new Set([3, 4, 5, 6]);

// Kreiranje unije skupova
const union = new Set([...S, ...T]);
```

file://

Unija: 1, 2, 3, 4, 5, 6

Zašto tri tačkice?

[...S] – **Raspakuje** elemente skupa S u niz. Dakle, umesto da radimo direktno sa skupom, koristimo ... da bismo dobili niz [1, 2, 3, 4]

```
// Konvertujemo Set u string i ispisujemo ga pomoću alert
alert(`Unija: ${[...union].join(", ")}`);
```

Metoda **join()** u JavaScript-u se koristi za **spajanje** svih elemenata niza u jedan string, koristeći određeni separator.

Operacije nad statičkim skupovima

- **`is_element_of(x,S)`**: proverava da li je vrednost x u skupu S.
- **`is_empty(S)`**: proverava da li je skup S prazan.
- **`size(S)`** ili **`cardinality(S)`**: vraća broj elemenata u skupu S.
- **`iterate(S)`**: vraća funkciju koja, pro svakom pozivu, vraća jednu ili više proizvoljno uređenih vrednosti skupa S.
- **`enumerate(S)`**: vraća listu koja sadrži proizvoljno uređene elemente skupa S.
- **`build(`x_1`,`x_2`,…,`x_n`,)`**: kreira Set strukturu sa vrednostima (elementima) $x_1, x_2, \ldots, x_n$.
- **`create_from(collection)`**: kreira novu Set strukturu koja sadrži sve elemente date kolekcije ili sve elemente koje je vratio dati iterator.
- Primer 1(is_element_of) i 2(is_empty):

```
function is_element_of(x, S) {
  return S.has(x);
}
const S = new Set([1, 2, 3, 4]);
alert(is_element_of(3, S));  // Ispis: true
alert(is_element_of(5, S));  // Ispis: false
```

```
function is_empty(S) {
  return S.size === 0;
}

const S1 = new Set([1, 2, 3]);
const S2 = new Set();

alert(is_empty(S1));  // Ispis: ?
alert(is_empty(S2));  // Ispis: ?
```

False

true

Primer 3. size

```
function size(S) {
  return S.size;
}

const S = new Set([1, 2, 3, 4, 5]);
alert(size(S));  // Ispis: ?
```

5

Operacije nad dinamičkim skupovima

- **`create()`**: kreira novu, inicijalno praznu Set strukturu.
- **`create_with_capacity(n)`**: kreira novu, inicijalno praznu Set strukturu koja je u stanju da skladišti do n elemenata.
- **`add(S,x)`**: dodaje element x u S, ako ga već nema u S.
- **`remove(S, x)`**: uklanja element x iz S, ako ga ima u S.
- **`capacity(S)`**: vraća maksimalan broj vrednosti koje S može da skladišti.

- Primer1: **create**

```
function create() {
  return new Set();
}

const set1 = create();
alert("Kreiran prazan set: " + [...set1]);  // Ispis: ?
```

Kreiran prazan set:

Primer 2 : **add**

```
function add(S, x) {
  S.add(x);  // Dodaje element u skup (ako već ne postoji, neće biti duplikata)
  alert("Trenutni set: " + [...S]);  // Ispis trenutnog skupa
}

// Kreiranje skupa
const set = new Set([1, 2, 3]);

// Testiranje funkcije
add(set, 4);  // Ispis: Trenutni set: 1,2,3,4
add(set, 2);  // Ispis: ?
```

Trenutni set: 1,2,3,4 (nije promenjeno)

```
add(set, 5);  // Ispis:?
```

Trenutni set: 1,2,3,4,5

Dodatne operacije

- **`pop(S)`**: vraća proizvoljan element iz S, brišući ga iz S.
- **`pick(S)`**: vraća proizvoljan element iz S. Functionally, the mutator pop can be interpreted as the pair of selectors (`pick`, `rest`), where `rest` returns the set consisting of all elements except for the arbitrary element. Can be interpreted in terms of `iterate`.
- **`map(F,S)`**: returns the set of distinct values resulting from applying function F to each element of S.
- **`filter(P,S)`**: returns the subset containing all elements of S that satisfy a given predicate P.
- **`fold(A0,F,S)`**: returns the value A|S| after applying $A_{i+1} := F(A_i, e)$ for each element e of S, for some binary operation F. F must be associative and commutative for this to be well-defined.
- **`clear(S)`**: delete all elements of S.
- **`equal(S1', S2')`**: checks whether the two given sets are equal (i.e. contain all and only the same elements).
- **`hash(S)`**: returns a hash value for the static set S such that if `equal(S1, S2)` then `hash(S1) = hash(S2)`

- Primer 1:pop()

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

let lastElement = arr.pop();
alert(lastElement);  // Ispis: ?
```

5

```
alert(arr);  // Ispis: ?
```

[1, 2, 3, 4]

Primer 2 : filter(P,S)

```
function filter(P, S) {
  return S.filter(P);  // Filter koristi predikat P
da filtrira niz S
}

// Predikat koji proverava da li je broj veći od
10
function isGreaterThanTen(num) {
  return num > 10;
}

// Primer skupa
let S = [5, 8, 12, 18, 25, 4];

// Pozivanje filter funkcije
let filteredArray = filter(isGreaterThanTen, S);

// Ispis rezultata
alert("Filtrirani niz: " + filteredArray);  // Ispis:?
```

Filtrirani niz: 12,18,25