

# Programiranje u skriptnim jezicima (PJS)

**Nositelj:** doc. dr. sc. Nikola Tanković

**Asistenti:**

- Luka Blašković, univ. bacc. inf.
- Alesandro Žužić, univ. bacc. inf.

**Ustanova:** Sveučilište Jurja Dobrile u Puli, Fakultet informatike u Puli



Fakultet informatike u Puli

## [3] Strukture podataka - Objekti i Polja

#3

JS

Strukture podataka su specijalizirani formati podataka namijenjeni efikasnijoj pohrani, organizaciji, dohvat i obradi podataka. U JavaScriptu, objekti i polja predstavljaju glavne gradivne elemente. **Objekti** su kontejneri koji omogućuju pohranu podataka u obliku proizvoljnog broja parova "ključ:vrijednost", dok **polja** predstavljaju kolekciju različitih elemenata organiziranih u linearni niz. Kombinacija ovih struktura omogućuje efikasno manipuliranje i pristup podacima u JavaScriptu.

**Posljednje ažurirano: 21.5.2024.**

### Sadržaj

- [Programiranje u skriptnim jezicima \(PJS\)](#)
- [3. Strukture podataka - Objekti i Polja](#)
  - [Sadržaj](#)
- [1. Objekti \(eng. \*objects\*\)](#)
  - [1.1 Osnovna sintaksa objekata](#)
    - [1.1.1 Svojstva objekta](#)
    - [1.1.2 Metode objekta](#)
  - [1.2 Ključna riječ `this`](#)
  - [1.3 Ažuriranje objekta](#)
  - [1.4 Konstruktori](#)
    - [Primjer 1 - Stvaranje objekta pomoću funkcije](#)
    - [Primjer 2 - Stvaranje objekta pomoću konstruktora](#)
  - [Vježba 1](#)
- [2. Standardni ugrađeni objekti \(eng. \*built-in objects\*\)](#)

- [2.1 String objekt](#)
- [Vježba 2](#)
  - [2.1.1 Escape znakovi \(eng. \*escape characters\*\) \(DODATNO\)](#)
- [2.2 Number objekt](#)
  - [2.2.1 NaN \(Not a Number\)](#)
  - [2.2.2 Infinity i -Infinity](#)
- [2.3 Math objekt](#)
- [Vježba 3](#)
- [2.4 Date objekt](#)
- [Vježba 4](#)
- [Vježba 5](#)
- [2.4 Usporedba JavaScript objekata](#)
  - [2.4.1 instanceof operator](#)
- [Samostalni zadatak za vježbu 4](#)
- [3. Polja \(eng. \*\*Arrays\*\*\)](#)
  - [3.1 Sintaksa polja](#)
    - [3.1.1 Pristup elementima polja](#)
    - [3.1.2 Veličina polja](#)
    - [3.1.3 Izmjene u polju](#)
    - [3.1.4 Array objekt sintaksa](#)
    - [3.2 Zašto Array objekt?](#)
      - [Primjer 1 - dodavanje, brisanje i pretraživanje koristeći obične uglate zagrade](#)
      - [Primjer 2 - dodavanje, brisanje i pretraživanje koristeći Array objekt](#)
  - [Vježba 6](#)
  - [3.2 Iteracije kroz polja](#)
    - [3.2.1 Tradicionalna for petlja](#)
    - [3.2.2 for...of petlja](#)
    - [3.2.3 for...in petlja](#)
    - [3.2.4 Array.forEach metoda](#)
  - [3.3 Objekti unutar polja](#)
    - [Primjer 3 - iteracija kroz polje objekata](#)
  - [Vježba 7](#)
  - [3.4 Osnovne metode Array objekta](#)
    - [3.4.1 Metode dodavanja, brisanja i stvaranja novih polja](#)
      - [Primjer 4 - paginate funkcija koristeći slice metodu](#)
    - [3.4.2 Metode pretraživanja polja](#)

- [Primjer 5 - Funkcija za brisanje korisnika iz polja](#)
- [Primjer 6 - Implementacija `removeDuplicates` funkcije](#)
- [Samostalni zadatak za vježbu 5](#)

# 1. Objekti (eng. *objects*)

Objekti su osnovna struktura podataka koja omogućavaju organizaciju i pohranu informacija. Objekt je skup povezanih podataka i/ili funkcionalnosti. Obično se sastoje od nekoliko varijabli i funkcija (koji se nazivaju **svojstvima** (eng. *property*) i **metodama** (eng. *methods*) kada se nalaze unutar definicije objekata).

Objekti se koriste za modeliranje stvarnih stvari, kao što su automobili, uloge, ljudi, hrana, knjige, itd.

Prije nego definiramo objekte, važno je razumijeti što su primitivni tipovi podataka u JavaScriptu.

Najjednostavnije rečeno, primitivni tipovi podataka, ili primitivi, su jednostavni podaci koji **nemaju svojstva i metode**, za razliku od objekata. JavaScript ima 7 primitivnih tipova podataka:

- `string`,
- `number`,
- `boolean`,
- `null`,
- `undefined`,
- `symbol` i
- `bigint`.

Primitivne vrijednosti su nepromjenjive (eng. *immutable*).

Na primjer, ako imamo `x = 3.14`, mi možemo promijeniti vrijednost varijable `x` u što god hoćemo, ali ne možemo promijeniti vrijednost `3.14`. `3.14` je uvijek `3.14`, kao što je i `2` uvijek `2`.

Drugi primjer, boolean vrijednosti `true` i `false` su uvijek `true` i `false`, isto vrijedi i za `null` i `undefined`. Takve vrijednosti su nepromjenjive!

Objekte stvaramo koristeći objektne literale, koji se sastoje od parova `ključ:vrijednost` (eng. *key-value*) odvojenih zarezima `,` i okruženih vitičastim zagradama `{}`. Svaki par `ključ:vrijednost` može biti svojstvo ili metoda objekta.

Možemo reći da je JavaScript objekt ustvari varijabla koja se sastoji od jednog ili više `ključ:vrijednost` parova.

Definirajmo prazan objekt `auto`. Postoji praksa da se objekti definiraju pomoću konstante `const`.

```
const auto = {};
```

Ovime smo stvorili prazan objekt `auto` koji ne sadrži nikakve podatke. Možemo ga ispisati u konzolu koristeći `console.log(auto)` i dobiti ćemo prazan objekt `{}`.

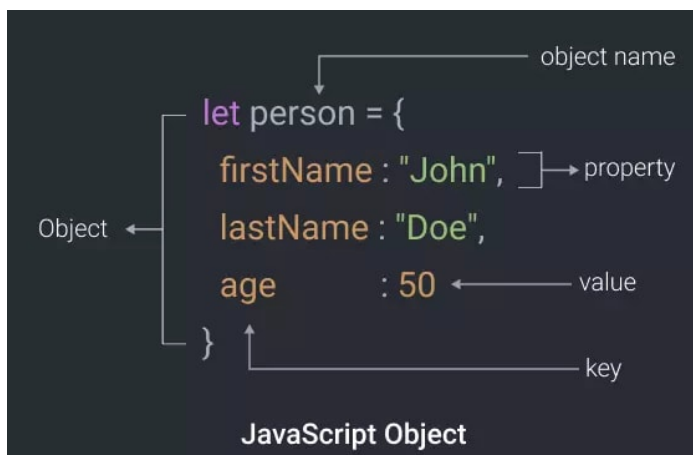
```
console.log(auto); // {}
```

# 1.1 Osnovna sintaksa objekata

U JavaScriptu, objekt se sastoji od više članova, od kojih svaki ima **ključ** (npr. *godina\_proizvodnje* i *boja*) i **vrijednost** (npr. *2020* i *"Crna"*). Svaki par `ključ:vrijednost` mora biti odvojen zarezom `,`, dok su ključ i vrijednost u svakom slučaju odvojeni dvotočjem `:`

Sintaksa uvijek slijedi uzorak:

```
const imeObjekta = {  
  ključ_1: vrijednost_1,  
  ključ_2: vrijednost_2,  
  ključ_3: vrijednost_3,  
};
```



Izvor: <https://dev.to/himanshudevgupta/javascript-most-important-thing-object-2hm1>

## 1.1.1 Svojstva objekta

Primjer objekta `auto` s 4 svojstva (`marka`, `model`, `godina_proizvodnje` i `boja`):

```
const auto = {  
  marka: "Ford",  
  model: "Mustang",  
  godina_proizvodnje: 2020,  
  boja: "Crna",  
};
```

Što će sada vratiti `console.log(auto)`?

```
console.log(auto); // { marka: "Ford", model: "Mustang", godina_proizvodnje: 2020, boja: "Crna" }
```

Možemo pristupiti svojstvima objekta koristeći notaciju točke `.`:

```
console.log(auto.godina_proizvodnje); // 2020  
console.log(auto.marka); // Ford  
console.log(auto.boja); // Crna
```

Moramo paziti da je ključ objekta jedinstven. Ako pokušamo dodati isti ključ više puta, JavaScript će zadržati samo posljednju vrijednost.

```
const auto = {
  marka: "Ford",
  model: "Mustang",
  boja: "Crna",
  godina_proizvodnje: 2020, // ⚠️
  godina_proizvodnje: 2021, // ⚠️ JavaScript će zadržati samo posljednju vrijednost
};
```

Par `ključ:vrijednost` može se deklarirati i na način da se `ključ` stavi unutar navodnika `" "`:

```
const auto = {
  "godina_proizvodnje": 2020,
};
console.log(auto.godina_proizvodnje); // 2020
```

Ovaj način deklariranja također omogućuje dodavanje `ključa` s **razmacima** što nije preporučljivo jer se tim svojstvima može pristupiti samo pomoću notacije uglatih zagrada `[]`:

```
const auto = {
  "godina proizvodnje": 2020,
};
console.log(auto["godina proizvodnje"]); // Dohvaća vrijednost ključa "godina proizvodnje"
koristeći notaciju uglatih zagrada
```

Možemo li dodati broj kao ključ objekta? Odgovor je **da**. Međutim, JavaScript će automatski pretvoriti broj u string.

U tom slučaju za pristupanje svojstvu koristimo notaciju uglatih zagrada `[]`:

```
const auto = {
  1: "Ford", // JavaScript će automatski pretvoriti broj 1 u string "1"
};
console.log(auto[1]); // Ford
console.log(auto.1); // SyntaxError: Unexpected number
```

**Zaključak:** svojstvima objekata možemo pristupiti koristeći notaciju točke `.` ili notaciju uglatih zagrada `[]`. Notacija točke je češće korištena i preporučljiva jer je jednostavnija i čitljivija. Notacija uglatih zagrada koristi se kada ključ sadrži razmake ili kada se ključ sastoji od varijable.

## 1.1.2 Metode objekta

Već smo spomenuli da objekti mogu sadržavati i **funkcije**. Funkcije unutar objekta nazivaju se **metode**. Metode su funkcije koje su vezane uz objekt kojemu pripadaju i koriste definirana svojstva unutar objekta za izvršavanje određenih zadataka.

Primjerice, kako bismo izračunali starost automobila, možemo dodati metodu `izracunajStarost` u objekt `auto`.

Funkcije ćemo definirati unutar objekta koristeći već poznatu sintaksu:

```
const auto = {  
  marka: "Ford",  
  model: "Mustang",  
  godina_proizvodnje: 2020,  
  boja: "Crna",  
  izracunajStarost: function () {  
    return new Date().getFullYear() - this.godina_proizvodnje; // 'this' se odnosi na  
    trenutni objekt u kojem je metoda definirana  
  },  
};
```


Međutim postoji i kraći način - jednostavnim izostavljanjem ključne riječ `function`:

```
const auto = {  
  marka: "Ford",  
  model: "Mustang",  
  godina_proizvodnje: 2020,  
  boja: "Crna",  
  izracunajStarost() {  
    return new Date().getFullYear() - this.godina_proizvodnje;  
  },  
};
```

Metodu `izracunajStarost` možemo pozvati koristeći notaciju točke:

```
console.log(auto.izracunajStarost()); // 4
```

U tablici su navedene metode i svojstva objekta `auto`:

| Objekt  | Svojstva  | Metode                         |
|---|---|--------------------------------|
| <p>auto</p>  | <p>auto.marka = "Ford"<br/>auto.model = "Mustang"<br/>auto.godina_proizvodnje = 2020<br/>auto.boja = "Crna"</p> | <p>auto.izracunajStarost()</p> |

✓ Zapamti! Kada pričamo o objektima, **svojstva** su varijable koje pripadaju objektu, a **metode** su funkcije koje pripadaju objektu.

## 1.2 Ključna riječ `this`

Ključna riječ `this` odnosi se na trenutni objekt u kojem se koristi. U kontekstu metoda objekta, `this` se odnosi na objekt koji sadrži metodu. U gornjem primjeru, `this` se odnosi na objekt `auto` jer je metoda `izracunajStarost` definirana unutar objekta `auto`.

`this` se koristi za pristup svojstvima i metodama objekta unutar samog objekta. Na primjer, u metodi `izracunajStarost`, `this.godina_proizvodnje` koristi se za pristup svojstvu `godina_proizvodnje` objekta `auto`.

Idemo dodati novu metodu `opisiAuto` u objekt `auto` koja će ispisati sve informacije o automobilu u jednoj rečenici, koristeći svojstva objekta `auto`.

Primjetite da se u metodi `opisiAuto` koristi ključna riječ `this` za pristup svojstvima objekta `auto`.

```
const auto = {
  marka: "Ford",
  model: "Mustang",
  godina_proizvodnje: 2020,
  boja: "Crna",
  izracunajStarost: function () {
    return new Date().getFullYear() - this.godina_proizvodnje;
  },
  opisiAuto: function () {
    return `Auto je ${this.marka} ${this.model} boje ${this.boja} iz
    ${this.godina_proizvodnje}.`;
  },
};
```

Sada možemo pozvati metodu `opisiAuto` koristeći notaciju točke:

```
console.log(auto.opisiAuto()); // Auto je Ford Mustang boje Crna iz 2020.
```

## 1.3 Ažuriranje objekta

Što ako želimo dodati, izbrisati ili ažurirati svojstva objekta? To možemo učiniti na nekoliko načina, u ovoj lekciji ćemo proći kroz najjednostavniji. Definirajmo objekt `grad` s nekoliko svojstava:

- `ime`: Pula
- `velicina`: 51.65 km<sup>2</sup>

```
const grad = {
  ime: "Pula",
  velicina: 51.65, // km2
};
```

Recimo da hoćemo ažurirati svojstvo `velicina` na `52` i dodati novo svojstvo `broj_stanovnika` s vrijednošću `56540`. To možemo učiniti na sljedeći način koristeći notaciju točke:

```
grad.velicina = 50;
grad.broj_stanovnika = 56540;
```

Isto je moguće postići koristeći notaciju uglatih zagrada `[]`:

```
grad["velicina"] = 50;
grad["broj_stanovnika"] = 56540;
```

Sada možemo ispisati objekt `grad` koristeći `console.log(grad)` i dobiti ćemo ažurirani objekt. Na jednak način objektu možemo dodati i metode. Primjerice, hoćemo dodati novu metodu `gustocaNaseljenosti()` kojom želimo prikazati broj stanovnika po kvadratnom kilometru.

```
grad.gustocaNaseljenosti = function () {  
  return this.broj_stanovnika / this.velicina;  
};  
  
console.log(grad.gustocaNaseljenosti()); // 1130.8 stanovnika/km²
```

Postoji još jedna korist upotrebe notacije uglatih zagrada `[]` - omogućuje nam pristup svojstvima objekta koristeći varijable. Na primjer, ako imamo varijablu `svojstvo` koja sadrži ime svojstva objekta, možemo koristiti tu varijablu za pristupanje svojstvu objekta:

```
const svojstvo = "ime";  
console.log(grad[svojstvo]); // Pula
```

Navedeno je korisno kada imamo dinamički generirane ključeve. Međutim, isto nije moguće napraviti koristeći notaciju točke `.`.

```
const svojstvo = "ime";  
console.log(grad.svojstvo); // undefined - neće raditi
```

Kako možemo izbrisati svojstvo objekta? Ključnom riječi `delete`! Recimo da hoćemo izbrisati svojstvo `velicina` iz objekta `grad`:

```
delete grad.velicina;  
console.log(grad); // { ime: "Pula", broj_stanovnika: 56540, gustocaNaseljenosti:  
  [Function: gustocaNaseljenosti] }
```

Ako upišete `delete grad.velicina` u konzolu primjetit ćete da će konzola vratiti `true` što znači da je svojstvo uspješno obrisano.

## 1.4 Konstruktori

Objekte smo do sad stvarali ručno, što je u redu ako ih trebamo stvoriti samo nekoliko. Međutim, što ako trebamo stvoriti stotine ili čak tisuće objekata? U tom slučaju, ručno stvaranje objekata postaje nepraktično i vremenski zahtjevno. Do sad smo naučili koristiti funkcije, zašto ne bi koristili funkciju za stvaranje novog objekta?

### Primjer 1 - Stvaranje objekta pomoću funkcije

Želimo stvoriti objekt `korisnik` s tri svojstva: `ime`, `prezime` i `godina_rođenja`.



```
const korisnik = {  
  ime: "Ana",  
  prezime: "Anić",  
  godina_rođenja: 1990,  
};
```

Što ako želimo stvoriti još jednog korisnika? Moramo ponoviti cijeli postupak:

```
const korisnik2 = {  
  ime: "Marko",  
  prezime: "Marić",  
  godina_rođenja: 1985,  
};
```

Kako možemo automatizirati proces? Idemo pokušati stvoriti funkciju `stvariKorisnika()` koja će stvoriti novog korisnika svaki put kada je pozovemo:

```
function stvariKorisnika(ime, prezime, godina_rođenja) {  
  const obj = {}; // stvari prazan objekt  
  obj.ime = ime; // dodaj svojstvo ime  
  obj.prezime = prezime; // dodaj svojstvo prezime  
  obj.godina_rođenja = godina_rođenja; // dodaj svojstvo godina_rođenja  
  
  obj.predstaviSe = function () {  
    console.log(  
      `Bok! Ja sam ${this.ime} ${this.prezime}. Rođen/a sam ${this.godina_rođenja} godine.`  
    );  
  };  
  
  return obj; // vrati objekt  
}
```

Sada možemo jednostavnije stvoriti nove korisnike koristeći novu funkciju `stvariKorisnika()`:

```
const ana = stvariKorisnika("Ana", "Anić", 1990);  
const marko = stvariKorisnika("Marko", "Marić", 1985);  
const petar = stvariKorisnika("Petar", "Petrović", 2001);  
  
ana.predstaviSe(); // "Bok! Ja sam Ana Anić. Rođen/a sam 1990 godine."  
marko.predstaviSe(); // "Bok! Ja sam Marko Marić. Rođen/a sam 1985 godine."  
petar.predstaviSe(); // "Bok! Ja sam Petar Petrović. Rođen/a sam 2001 godine."
```

## Primjer 2 - Stvaranje objekta pomoću konstruktora

Ovo radi dobro, ali zašto moramo svaki put stvarati novi objekt `obj` i vraćati ga na kraju funkcije? U JavaScriptu postoji posebna vrsta funkcije koja se zove **konstruktor** (eng. **constructor**). Konstruktori su posebne funkcije koje se koriste za stvaranje novih objekata. Konstruktori rade na sljedeći način:

1. Stvaraju prazan objekt

2. Dodaju svojstva i metode objektu
3. Automatski vraćaju objekt

Konstruktori, po konvenciji, se pišu velikim početnim slovom i nazivaju po objektu kojeg stvaraju. Dakle, prijašnju funkciju `stvoriKorisnika` možemo preoblikovati u konstruktor `Korisnik`. Kako ne stvaramo prazan objekt, već ga automatski vraćamo, ne moramo koristiti ključnu riječ `return`. Također, za dodavanje svojstava i metoda objektu koristimo ključnu riječ `this`, gdje se `this` odnosi na novi objekt koji se stvara.

```
function Korisnik(ime, prezime, godina_rodenja) {
  this.ime = ime;
  this.prezime = prezime;
  this.godina_rodenja = godina_rodenja;
  this.predstaviSe = function () {
    console.log(
      `Bok! Ja sam ${this.ime} ${this.prezime}. Rođen/a sam ${this.godina_rodenja} godine.`
    );
  };
}
```

Kako bi JavaScript znao da je funkcija `Korisnik` konstruktor, moramo koristiti ključnu riječ `new` prije poziva konstruktora.

```
const ana = new Korisnik("Ana", "Anić", 1990);
const marko = new Korisnik("Marko", "Marić", 1985);
const petar = new Korisnik("Petar", "Petrović", 2001);

ana.predstaviSe(); // "Bok! Ja sam Ana Anić. Rođen/a sam 1990 godine."
marko.predstaviSe(); // "Bok! Ja sam Marko Marić. Rođen/a sam 1985 godine."
petar.predstaviSe(); // "Bok! Ja sam Petar Petrović. Rođen/a sam 2001 godine."
```

Na ovaj način definiramo i stvaramo nove objekte koristeći konstruktor.

## Vježba 1

EduCoder šifra: `Automobil`

1. Definirajte konstruktor `Automobil`. U konstruktor postavite sljedeća svojstva automobilu: `marka`, `model`, `godina_proizvodnje`, `boja` i `cijena`. Kada to napravite, izradite nekoliko objekata tipa `Automobil` koristeći vaš konstruktor.
2. Dodajte metodu `azurirajCijenu(novaCijena)` u konstruktor `Automobil` koja će ažurirati cijenu automobila.
3. Dodajte metodu `detalji()` u konstruktor `Automobil` koja će u jednoj rečenici ispisati sva svojstva automobila.
4. Pozovite za svaki automobil metodu `detalji()` i metodu `azurirajCijenu()`.

Primjer rezultata:

Marka: Toyota  
Model: Corolla  
Godina proizvodnje: 2019  
Boja: siva  
Cijena: 15000

Marka: Volkswagen  
Model: Golf  
Godina proizvodnje: 2015  
Boja: crna  
Cijena: 11500

## 2. Standardni ugrađeni objekti (eng. *built-in objects*)

JavaScript nudi mnoštvo ugrađenih (eng. *built-in*) objekata koji modeliraju koncepte iz stvarnog svijeta, ali i obogaćuju primitivne tipove podataka brojnim korisnim metodama. Ugrađeni objekti pružaju razne metode i svojstva za rad s podacima, poput manipulacije nizovima znakova `String`, rad s datumima `Date`, matematičke operacije `Math`, itd.

Do sad smo se već susreli s nekoliko ugrađenih objekata, poput `Date` i `Math`, a u narednim poglavljima upoznat ćemo se detaljnije s ugrađenim objektima: `String`, `Number`, `Math` i `Date`.

### 2.1 `String` objekt

`String` objekt predstavlja tekstualne podatke, odnosno niz znakova (`string`). Nudi razne korisne metode za manipulaciju i analizu nizova znakova.

Ako postoji ugrađeni `String` objekt, to znači da možemo pozvati i njegov konstruktor `String()` kako bismo stvorili novi `String` objekt. Međutim, to rijetko radimo jer je moguće stvoriti `String` objekt koristeći objektne literale, tj. navodnike `"` ili apostrofe `'`.

**Važno je naglasiti** da kod svih primitivnih tipova podataka (npr. `string`, `number`, `boolean`) možemo koristiti metode i svojstva kao da su objekti. JavaScript automatski za nas pretvara primitivne tipove u objekte kada koristimo metode i svojstva nad njima.

```
const ime = "Ana"; // stvara se primitivni tip podataka string
const prezime = new String("Anić"); // stvara se objekt String pozivanjem konstruktora

console.log(typeof ime); // string
console.log(typeof prezime); // object - stvoren je objekt String

console.log(prezime); // [String: 'Anić']
```

Uočite da se primitivni tipovi podataka pišu malim početnim slovom, a objekti velikim početnim slovom.

Pitanje? Što će vratiti `===` operator za `x` i `y`?

```
let x = "Pas";
let y = new String("Pas");
console.log(x === y); ?
```

► Spoiler!

```
let x = "Pas";
let y = new String("Pas");
console.log(x === y); false
console.log(typeof x); // string
console.log(typeof y); // object
console.log(x == y); true
```

Ispod su navedene neke od najčešće korištenih metoda `string` objekta. Ima ih još [mnogo](#), ali ove su najpoznatije.

| Metoda                     | Objašnjenje   | Sintaksa  | Primjer  | Output                          |
|----------------------------|---|---|--|---------------------------------|
| <code>charAt()</code>      | Vraća znak na određenom indeksu u nizu znakova. Indeks prvog znaka je <code>0</code> .  | <code>string.charAt(index)</code>                                 | <code>'hello'.charAt(1)</code>                   | <code>'e'</code>                |
| <code>concat()</code>      | Spaja dva ili više nizova znakova te vraća novi niz, slično kao operator <code>+</code> nad nizovima.   | <code>string.concat(substring1, substring2 ... substringN)</code> | <code>'hello'.concat(' world')</code>            | <code>'hello world'</code>      |
| <code>indexOf()</code>     | Vraća indeks prvog pojavljivanja podniza (eng. <b>substring</b> ) u nizu  | <code>string.indexOf(substring)</code>                            | <code>'hello'.indexOf('lo')</code>               | <code>3</code>                  |
| <code>lastIndexOf()</code> | Vraća indeks zadnjeg pojavljivanja podniza u nizu   | <code>string.lastIndexOf(substring)</code>                        | <code>'hello'.lastIndexOf('l')</code>            | <code>3</code>                  |
| <code>toUpperCase()</code> | Pretvara cijeli niz znakova u velika slova  | <code>string.toUpperCase()</code>                                 | <code>'hello'.toUpperCase()</code>               | <code>'HELLO'</code>            |
| <code>toLowerCase()</code> | Pretvara cijeli niz znakova u mala slova  | <code>string.toLowerCase()</code>                                 | <code>'HELLO'.toLowerCase()</code>               | <code>'hello'</code>            |
| <code>substring()</code>   | Izdvađa podskup niza znakova i vraća novi niz bez izmjene originalnog niza. Metoda će izdvojiti podskup <code>[indexStart, indexEnd)</code> , dakle <code>indexEnd</code> neće biti uključen. Ako je <code>indexStart &gt; indexEnd</code> , <code>substring()</code> će ih zamijeniti. Ako su indeksi negativni brojevi, interpretirat će se kao <code>0</code> .  | <code>string.substring(indexStart, indexEnd)</code>               | <pre>let novi = 'Novigrad'.substring(0, 4)</pre> | <code>novi === 'Novi'</code>    |
| <code>slice()</code>       | Izdvađa podskup niza znakova i vraća novi niz bez izmjene originalnog niza. Metoda će izdvojiti podskup <code>[indexStart, indexEnd)</code> , dakle <code>indexEnd</code> neće biti uključen. Za razliku od <code>substring()</code> metode, ako je <code>indexStart &gt; indexEnd</code> , <code>slice()</code> će vratiti prazan string <code>""</code> . Ako su indeksi negativni brojevi, brojat brojat će mjesta počevši od kraja. | <code>string.slice(indexStart, indexEnd)</code>                   | <pre>let noviNiz = 'Novigrad'.slice(-4)</pre>    | <code>noviNiz === 'grad'</code> |
|                            | Metoda prvo pretražuje zadani   |   |  |                                 |

|                           |  |  |  |   |
|---------------------------|--|--|--|---|
| <code>replace()</code>    | <code>pattern</code> u stringu koji može biti drugi niz znakova ili <code>RegExp</code> . Ako ga pronade, zamjenjuje prvi <code>pattern</code> podskup s <code>replacement</code> . Metoda vraća novi uređeni znakovni niz bez izmjene originalnog.  | <code>string.replace(pattern, replacement)</code>                  | <code>'Hello, world!'.replace('world', 'JavaScript')</code>  | <code>'Hello, JavaScript!'</code>   |
| <code>split()</code>      | Razdvaja znakovni niz prema danom <code>separator</code> argumentu i dobivene podnizove prema u polje. Vraća polje podnizova bez izmjene originalnog znakovnog niza. Metoda ima i opcionalni separator <code>limit</code> koji označava limit broja podnizova koji se mogu spremiti u polje.                         | <code>string.split(separator, limit)</code>                        | <code>'The quick brown fox jumps over the lazy dog.'.split('')</code>  | <code>['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog.']</code> |
| <code>trim()</code>       | Uklanja razmake s početka i kraja niza. Vraća novi niz bez izmjene originalnog.  | <code>string.trim()</code>   | <code>' hello '.trim()</code>  | <code>'hello'</code>  |
| <code>match()</code>      | Pronalazi podudaranja u znakovnom nizu uz pomoć regularnih izraza ( <code>RegExp</code> ). Vraća polje podskupa niza koji odgovaraju <code>RegExp</code> izrazu.   | <code>string.match(RegExp)</code>                                  | Hoćemo pronaći sve brojeve u rečenici: <code>'Godina je 2024 i mjesec je 3'.</code><br><code>match(/\d+/g)</code> . <code>\d</code> - broj <code>[0-9]</code> , <code>\d+</code> - traži poklapanje jednog ili više broja <code>g</code> - regex oznaka za globalno pretraživanje. | <code>['2024', '3']</code>  |
| <code>repeat()</code>     | Ponavlja niz određeni broj ( <code>count</code> ) puta.  | <code>string.repeat(count)</code>                                  | <code>'hello'.repeat(3)</code>   | <code>'hellohellohello'</code>  |
| <code>startsWith()</code> | Provjerava počinje li niz nekim podnizom. Opcionalno ima parametar <code>position</code> koji definira poziciju gdje se provjerava podniz, <code>0</code> po defaultu. Vraća <code>boolean</code> vrijednost ovisno o pronalasku.  | <code>string.startsWith(substring, position=0)</code>              | <code>'To be, or not to be, that is the question.'.startsWith('To be')</code>  | <code>true</code>   |
| <code>endsWith()</code>   | Provjerava završava li niz nekim podnizom. Opcionalno ima parametar <code>endPosition</code> koji definira krajnju poziciju gdje se očekuje substring, <code>string.length</code> tj. zadnji indeks u stringu po defaultu. Vraća <code>boolean</code> vrijednost ovisno o pronalasku.                                | <code>string.endsWith(substring, endPosition=string.length)</code> | <code>'Cats are the best!'.endsWith('best!')</code>  | <code>true</code>   |
| <code>includes()</code>   | Provjerava postoji li određeni podniz u nizu. Metoda je case-sensitive te vraća <code>boolean</code> vrijednost ovisno o tome postoji li podniz. Dodatno, tu je opcionalni <code>position</code> argument koji započinje pretragu na određenoj poziciji, <code>0</code> po defaultu - dakle pretraživanje od početka | <code>string.includes(substring)</code>                            | <code>'The quick brown fox jumps over the lazy dog.'.includes('fox')</code>  | <code>true</code>   |

Iz tablice možete iščitati razlike između metoda `substring()` i `slice()`. Obe metode vraćaju podniz niza, ali se razlikuju u načinu rada s negativnim indeksima i indeksima koji su izvan granica niza. Preporuka je koristiti `slice()` jer je fleksibilniji i ima jasnije ponašanje, osim ako nemate koristi od specifičnog ponašanja `substring()` - najčešće je to zamjena index argumenata.

**Zašto je dobro naučiti koristiti ove metode?**

Većina ovih metoda koristi se svakodnevno u programiranju. Na primjer, `split()` metoda koristi se za razdvajanje niza znakova na riječi, `toUpperCase()` i `toLowerCase()` metode koriste se za normalizaciju teksta, `replace()` metoda koristi se za zamjenu dijelova teksta, itd. Ne želimo gubiti vrijeme i ručno raditi stvari nad znakovnim nizovima, za koje već postoje gotove metode.

Primjerice, imamo potrebu izvući sve riječi iz neke rečenice. Ispod je primjer kako bismo to ručno napravili:

```
const recenica = "Pula je grad u Istri.";
const rijeci = []; // prazno polje za spremanje riječi (nismo još prošli polja)
let trenutnaRijec = ""; // prazan string za spremanje trenutne riječi
for (let i = 0; i < recenica.length; i++) {
  // prolazimo kroz svaki znak u rečenici
  if (recenica[i] !== " ") {
    // ako trenutni znak nije razmak
    trenutnaRijec += recenica[i]; // dodaj trenutni znak u trenutnu riječ
  } else {
    rijeci.push(trenutnaRijec); // dodaj trenutnu riječ u polje riječi
    trenutnaRijec = ""; // resetiraj trenutnu riječ
  }
}
rijeci.push(trenutnaRijec); // dodaj zadnju riječ u polje riječi
console.log(rijeci); // ["Pula", "je", "grad", "u", "Istri."]
```

To je 10-tak linija kôda za vrlo učestalu radnju 🤖 Isto možemo postići koristeći `String.split()` metodu:

```
const recenica = "Pula je grad u Istri.";
const rijeci = recenica.split(" ");
console.log(rijeci); // ["Pula", "je", "grad", "u", "Istri."]
```

## Vježba 2

EduCoder šifra: `vowels`

1. Napišite funkciju `brojSamoglasnikaISuglasnika` koja prima ulazni string i vraća objekt s dva svojstva:

`samoglasnici: broj_samoglasnika` i `suglasnici: broj_suglasnika`.

- Koristi metodu `match()` za pronalaženje samoglasnika (`regex = /[aeiou]/g`) i suglasnika (`regex = /^[^aeiou\W]/g`) u ulaznom stringu ili koristi `indexOf()` metodu za provjeru podudaranja svakog znaka s nizom samoglasnika.
- Koristi `toUpperCase()` ili `toLowerCase()` za normalizaciju slova.
- Na primjer:

```
console.log(brojSamoglasnikaISuglasnika("Hello World"));
// { samoglasnici: 3, suglasnici: 7 }
```

2. Napišite funkciju `duljinaRijeci` koja prima rečenicu te ispisuje sve riječi iz rečenice i njihovu duljinu. Funkcija ne mora vraćati ništa.

- Na primjer:

```
duljinaRijeci("    JavaScript je zabavan    ");  
// JavaScript: 10  
// je: 2  
// zabavan: 7
```

## 2.1.1 Escape znakovi (eng. *escape characters*) (DODATNO)

Escape znakovi su posebni znakovi koji se koriste za označavanje posebnih znakova u nizovima znakova. Na primjer, ako želimo koristiti znak navodnika " unutar niza znakova, moramo ga označiti escape znakom \. Primjerice, kako bismo pokušali na ovaj način pohraniti sljedeći tekst, naišli bi na problem:

```
const tekst = "We are the so-called \"Vikings\" from the north."; // SyntaxError: Unexpected  
identifier
```

JavaScript će ovaj string presjeći na "We are the so-called".

Ovaj problem možemo riješiti pisanjem jednostrukih navodnika ' umjesto dvostrukih ":

```
const tekst = 'We are the so-called "Vikings" from the north.';
```

No escape znakovi nam omogućavaju rješavanje ovog, i još brojnih sličnih problema s nizovima znakova. Možemo ubaciti escape znak \ prije svakog znaka navodnika ":

```
const tekst = "We are the so-called \"Vikings\" from the north."; // We are the so-called  
"Vikings" from the north.
```

Kako možemo jednostavno ispisati znak \ u nizu znakova? Koristimo dva escape znaka \\\:

```
console.log("C:\\Users\\Ana\\Desktop\\file.txt"); // C:\Users\Ana\Desktop\file.txt
```

ili ako želimo tekst ispisivati u više linije, koristimo escape znakove \n:

```
console.log("Prva linija\nDruga linija\nTreća linija");  
// Prva linija  
// Druga linija  
// Treća linija
```

Tablica escape znakova:

| Code | Result               |
|------|----------------------|
| \"   | "                    |
| \`   | `                    |
| \\   | \                    |
| \b   | Backspace            |
| \f   | Form Feed            |
| \n   | New line             |
| \r   | Carriage return      |
| \t   | Horizontal Tabulator |
| \v   | Vertical Tabulator   |

Ne morate ih sve znati napamet, ali je dobro znati da postoje. Ovi tabulatori nastali su u doba pisačkih strojeva, teleprinter a i fax uređaja. U HTML-u ih nema potrebe koristiti jer se tekst formatira pomoću CSS-a.

## 2.2 Number objekt

`Number` objekt predstavlja numeričke podatke, odnosno brojeve. Nudi razne korisne metode za rad s brojevima u JavaScriptu. Isto kao i `string` objekt, `Number` objekt ima svoj konstruktor `Number()` koji se rijetko koristi jer je moguće stvoriti `Number` objekt koristeći objektne literale odnosno same brojeve.

```
const broj = 5; // primitivni broj
const brojObjekt = new Number(5); // objekt broj - nemojte ovo raditi (samo komplicira kôd)

console.log(typeof broj); // number - uočite malo početno slovo
console.log(typeof brojObjekt); // object - Number objekt
```

Prisjetimo se kratko gradiva iz prve skripte. JavaScript će pokušati evaluirati "string brojeve", npr. `5` u primitivni tip `number`.

```
console.log(5 + 5); // 10
let x = "10";
let y = "2";
console.log(x - y); // 8
console.log(x * y); // 20
console.log(x / y); // 5
```

Ali...

```
console.log(x + y); // "102" - konkatencija stringova
```



U primjeru `x+y` JavaScript neće koristiti matematičku logiku operatora `+` već će spojiti dva stringa u jedan, jer je `+` operator nad stringovima -> operator konkatencije.

Iz ovog razloga, poželjno je izbjegavati spajanje stringova `+` operatorom, već koristiti metodu `String.concat()` s kojom smo se upoznali u prethodnom poglavlju.

Ispod se nalazi tablica s nekoliko najčešće korištenih metoda `Number` objekta:

| Metoda                     | Objašnjenje   | Sintaksa                                    | Primjer   | Output  |
|----------------------------|---|---|---|---|
| <code>toFixed()</code>     | Zakružuje broj na zadani broj ( <code>digits</code> ) decimalnih mjesta. Vraća string (zbog decimalne točke).   | <code>number.toFixed(digits)</code>         | <code>(5.56789).toFixed(2)</code>                                     | <code>"5.57"</code>                           |
| <code>toPrecision()</code> | Za dani broj metoda vraća njegovu string reprezentaciju na zadani broj <b>značajnih znamenki</b> : <code>precision</code> parametar mora biti između <code>1</code> i <code>100</code> .  | <code>number.toPrecision(2)</code>          | <code>(5.123456).toPrecision(2)</code>                                | <code>"5.1"</code>                            |
| <code>toString()</code>    | Vraća string reprezentaciju broja. Opcionalni <code>radix</code> parametar, može biti između <code>2</code> i <code>36</code> i specificira bazu koja se koristi za reprezentaciju broja. Default je <code>10</code> (dekadski zapis) | <code>number.toString(radix=10)</code>      | <code>(123).toString();</code><br><code>(100).toString(2)</code>      | <code>"123";</code><br><code>"1100100"</code> |
| <code>parseInt()</code>    | Metoda pretvara dani string u cjelobrojni ekvivalent. Kao i kod <code>toString()</code> , sadrži opcionalni <code>radix</code> parametar.   | <code>Number.parseInt(string, radix)</code> | <code>parseInt("10.456");</code><br><code>parseInt("40 years")</code> | <code>10; 40</code>                           |
| <code>parseFloat()</code>  | Metoda pretvara dani string u floating-point ekvivalent.  | <code>Number.parseFloat(string)</code>      | <code>parseFloat("10.456")</code>                                     | <code>10.456</code>                           |
| <code>isInteger()</code>   | Provjerava je li dana vrijednost <code>value</code> integer. Vraća <code>boolean</code> vrijednost ovisno o tome.   | <code>Number.isInteger(value)</code>        | <code>isInteger(5.2)</code>   | <code>false</code>                            |
| <code>isNaN()</code>       | Provjerava je li dana vrijednost <code>NaN</code> (Not a Number). Vraća <code>boolean</code> vrijednost ovisno o tome.  | <code>Number.isNaN(value)</code>            | <code>isNaN("string")</code>  | <code>true</code>                             |

## 2.2.1 NaN (Not a Number)

`NaN` je rezervirana riječ u JavaScriptu koja označava "Not a Number". `NaN` je povratna vrijednost nakon evaluacije neuspješnog matematičkog izraza. Na primjer, ako želimo podijeliti broj 100 s jabukom?

```
let x = 100 / "jabuka";
console.log(x); // NaN
```

Naravno, ako se radi o numeričkom stringu, rezultat će biti broj.

```
let y = 100 / "10";
console.log(y); // 10
```

Ironično, `typeof(NaN)` vraća `number`! 😊

## 2.2.2 Infinity i -Infinity

`Infinity` je rezervirana riječ u JavaScriptu koja označava beskonačnost. `Infinity` je povratna vrijednost nakon evaluacije matematičkog izraza koji rezultira beskonačnošću. Na primjer, ako podijelimo bilo koji broj s nulom, rezultat će biti `Infinity`.

```
let x = 100 / 0;
console.log(x); // Infinity
```

`typeof(Infinity)` također vraća `number`.

## 2.3 Math objekt

`Math` objekt sadrži matematičke konstante i funkcije. Ovaj objekt je statički, što znači da se ne može instancirati. Sve metode i konstante `Math` objekta su statičke (eng. **static**), što znači da se pozivaju direktno na objektu, a ne na instanci objekta. `Math` objekt sadrži mnoge korisne metode i konstante za rad s brojevima. Mogu se koristiti samo s `Number` tipom, s `BigInt` tipom neće raditi.

Ispod su navedene neke od najčešće korištenih konstanti i statičnih metoda `Math` objekta.

| Metoda                  | Objašnjenje   | Rezultat           |
|-------------------------|---|--------------------|
| <code>Math.PI</code>    | Vraća vrijednost konstante $\pi$ (pi)                           | 3.141592653589793  |
| <code>Math.E</code>     | Vraća vrijednost konstante $e$ (Eulerov broj)                   | 2.718281828459045  |
| <code>Math.SQRT2</code> | Vraća vrijednost $\sqrt{2}$                                     | 1.4142135623730951 |
| <code>Math.LN2</code>   | Vraća vrijednost prirodnog (ln) logaritma broja <code>2</code>  | 0.6931471805599453 |
| <code>Math.LN10</code>  | Vraća vrijednost prirodnog (ln) logaritma broja <code>10</code> | 2.302585092994046  |

| Metoda                      | Objašnjenje  | Sintaksa                              | Primjer                            | Output  |
|-----------------------------|--|---------------------------------------|------------------------------------|---|
| <code>Math.abs(x)</code>    | Vraća apsolutnu vrijednost broja <code>x</code> .  | <code>Math.abs(x)</code>              | <code>Math.abs(-4.5)</code>        | <code>4.5</code>  |
| <code>Math.ceil(x)</code>   | Metoda zaokružuje i vraća najmanji cijeli broj veći ili jednak zadanom ( <code>x</code> ) broju.   | <code>Math.ceil(x)</code>             | <code>Math.ceil(4.3)</code>        | <code>5</code>  |
| <code>Math.floor(x)</code>  | Metoda zaokružuje prema dolje i vraća najveći cijeli broj manji ili jednak zadanom ( <code>x</code> ) broju.   | <code>Math.floor(x)</code>            | <code>Math.floor(4.9)</code>       | <code>4</code>  |
| <code>Math.max(x, y)</code> | Vraća veći od dva broja <code>x</code> i <code>y</code> . Moguće je navesti i više od 2 parametara, metoda će uvijek vratiti najveći.  | <code>Math.max(x, y, .. N)</code>     | <code>Math.max(5, 10)</code>       | <code>10</code>   |
| <code>Math.min(x, y)</code> | Vraća manji od dva broja <code>x</code> i <code>y</code> . Moguće je navesti i više od 2 parametara, metoda će uvijek vratiti najmanji.  | <code>Math.min(x, y, .. N)</code>     | <code>Math.min(5, 10)</code>       | <code>5</code>  |
| <code>Math.pow(x, y)</code> | Vraća rezultat potenciranja broja <code>x</code> na potenciju <code>y</code> .   | <code>Math.pow(base, exponent)</code> | <code>Math.pow(2, 3)</code>        | <code>8</code>  |
| <code>Math.sqrt(x)</code>   | Računa kvadratni korijen broja <code>x</code> .  | <code>Math.sqrt(x)</code>             | <code>Math.sqrt(9)</code>          | <code>3</code>  |
| <code>Math.round(x)</code>  | Zaokružuje broj <code>x</code> na najbliži cijeli broj.  | <code>Math.round(x)</code>            | <code>Math.round(4.3)</code>       | <code>4</code>  |
| <code>Math.random()</code>  | Generira pseudoslučajan broj između <code>0</code> i <code>1</code> . Funkcija koristi približno uniformnu distribuciju. Ne pruža kriptografski sigurne slučajne brojeve pa se za te svrhe ne koristi. | <code>Math.random()</code>            | <code>Math.random()</code>         | (slučajni broj između <code>0</code> i <code>1</code> ) |
| <code>Math.log(x)</code>    | Računa prirodni logaritam (po bazi $e$ ) broja <code>x</code> .  | <code>Math.log(x)</code>              | <code>Math.log(Math.E)</code>      | <code>1</code>  |
| <code>Math.exp(x)</code>    | Računa $e$ na potenciju <code>x</code> .   | <code>Math.exp(x)</code>              | <code>Math.exp(1)</code>           | <code>2.718281828459045</code>                          |
| <code>Math.sin(x)</code>    | Računa sinus broja <code>x</code> (u radijanima).  | <code>Math.sin(x)</code>              | <code>Math.sin(Math.PI / 2)</code> | <code>1</code>  |
| <code>Math.cos(x)</code>    | Računa kosinus broja <code>x</code> (u radijanima).  | <code>Math.cos(x)</code>              | <code>Math.cos(Math.PI)</code>     | <code>-1</code>   |
| <code>Math.tan(x)</code>    | Računa tangens broja <code>x</code> (u radijanima).  | <code>Math.tan(x)</code>              | <code>Math.tan(Math.PI / 4)</code> | <code>1</code>  |

## Vježba 3

EduCoder šifra: `matematika`

1. Napišite funkciju `hipotenuza(duzinaA, duzinaB)` koja prima dužine dvije katete pravokutnog trokuta. Funkcija treba izračunati i vratiti dužinu hipotenuze primjenjujući Pitagorin poučak, koji glasi:  $c = \sqrt{a^2 + b^2}$ , gdje su `a` i `b` dužine kateta, a `c` dužina hipotenuze. Ispiši rezultat u formatu "Dužina hipotenuze je: `[hipotenuza]`". Za implementaciju koristite metode iz `Math` objekta.

✓ Rezultat:

```
console.log(hipotenuza(3, 4)); // Output: Dužina hipotenuze je: 5.00
```

1. Napišite funkciju proizvoljnog naziva koja prima broj `n`. Funkcija provjerava je li `n` broj, ako nije vraća poruku "Nije broj!". Ako je broj, funkcija vraća 10 brojeva većih od `n` u formatu: "Broj 1: `[n+1]`, Broj 2: `[n+2]`, ..., Broj 10: `[n+10]`". Ako su `[Broj 1 - Broj 10]` decimalni brojevi, zaokružite ih na dvije decimale i ispišite ih u tom formatu u konzolu. Ako su `[Broj 1 - Broj 10]` cijeli brojevi, pretvorite ih u binarni oblik i ispišite u konzolu.

✓ Rezultat:

```
console.log(fun(5));  
// Output: 110, 111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111  
  
console.log(fun(5.5));  
// Output: 6.5, 7.5, 8.5, 9.5, 10.5, 11.5, 12.5, 13.5, 14.5, 15.5
```

3. Napišite funkciju `izracunajSinKos()` koja računa sinus i kosinus kuta `d` (u stupnjevima) te vraća objekt s 2 svojstva: `sinus: sinusVrijednost`, `kosinus: kosinusVrijednost`. Za implementaciju koristite metode iz `Math` objekta. Stupnjeve pretvorite u radijane koristeći formulu: `radijani = stupnjevi * (π / 180)`. Dobivene vrijednosti zaokružite na 2 decimale.

✓ Rezultat:

```
console.log(izracunajSinKos(30));  
// Output: { sinus: 0.5, kosinus: 0.87 }
```

## 2.4 Date objekt

`Date` objekt reprezentira trenutak u vremenu. Ovaj objekt koristi se za rad s datumima i vremenom. `Date` objekt može se koristiti za stvaranje datuma i vremena, te za njihovu manipulaciju i prikaz. `Date` objekt enkapsulira broj milisekundi od 1. siječnja 1970. godine, poznat kao UNIX vremenska oznaka (eng. **UNIX timestamp**).

Generalno, u JavaScriptu postoje 3 načina definiranja datuma:

| Tip        | Primjer                                   |
|------------|---|
| ISO Date   | "2015-03-25" (The International Standard) |
| Short Date | "03/25/2015"                              |
| Long Date  | "Mar 25 2015" ili "25 Mar 2015"           |

Od ovih standarda, ISO format je najčešće korišten i preporučuje se. [ISO 8601](#) sintaksa izgleda ovako: `YYYY-MM-DDTHH:mm:ss.sssZ`, gdje `YYYY` predstavlja godinu, `MM` mjesec, `DD` dan, `T` literal koji odvaja datum i vrijeme, `HH` sat, `mm` minute, `ss` sekunde, `sss` milisekunde i `Z` je offset vremenske zone. Primjerice, 27. rujna 2023. godine u 18:00 sati izgleda ovako: `2023-09-27 18:00:00`.

*Mala napomena* - `Date` objekt u JavaScriptu je vrlo opširan, nekima možda i nezgrapan budući da ima veliki broj zastarjelih metoda i konvencija. U modernom JavaScriptu, preporučuje se korištenje `moment.js` biblioteke za rad s datumima i vremenom. To možete proučiti sami, za potrebe ovog kolegija proći ćemo samo osnove `Date` objekta. `TC39` grupa (koja razvija JavaScript) radi na [novom standardu](#) za rad s datumima i vremenom, koji će zamijeniti `Date` objekt.

Novi datum možemo stvoriti koristeći `new Date()` konstruktor. Konstruktor može primiti različite argumente, ukupno njih 9, mi ćemo proći samo nekoliko:

| Sintaksa                                | Objašnjenje  | Primjer   |
|---|--|---|
| <code>new Date()</code>                 | stvara novi <code>Date</code> objekt s <b>trenutnim datumom i vremenom</b>   | <code>const d = new Date();</code>  |
| <code>new Date(date string)</code>      | stvara novi <code>Date</code> objekt iz <a href="#">date stringa</a>   | <code>new Date("October 13, 2014 11:13:00");</code> ili <code>new Date("2022-03-25");</code>        |
| <code>new Date(year, month, ...)</code> | stvara novi <code>Date</code> objekt sa specificiranim datumom i vremenom. <b>JavaScript broji mjesece od 0!</b> Dakle 0 = Siječanj, 11 = Prosinac | <code>const d = new Date(2019, 3, 24, 10, 33, 30);</code> <code>d = Wed Apr 24 2019 10:33:30</code> |
| <code>new Date(milliseconds)</code>     | stvara novi <code>Date</code> objekt s <b>brojem milisekundi od 1. siječnja 1970.</b> odnosno <code>unix oznakom</code>                            | <code>const d = new Date(1708436235000);</code>   |

Primjetite da kod ispisa `Date` objekta, u konzolu nećemo dobiti klasičan ispis objekta, kao što je slučaj kod `String` i `Number` objekata. Umjesto toga, dobit ćemo ispis u formatu koji podsjeća na string reprezentaciju datuma budući da JavaScript automatski poziva `toString()` metodu prilikom ispisa objekta.

Nakon što izradimo `Date` objekt, možemo koristiti razne metode za dohvaćanje i manipulaciju datuma i vremena. Ispod se nalazi tablica s nekoliko najčešće korištenih metoda `Date` objekta:

| Metoda                 | Objašnjenje  | Sintaksa                    | Primjer  | Output |
|------------------------|--|-----------------------------|--|--------|
| <code>getDate()</code> | Za dani datum, vraća <b>dan</b> u mjesecu kao broj (1-31).                       | <code>Date.getDate()</code> | <code>const rodendan = new Date("April 13, 2000");</code><br><code>rodendan.getDate() == 13</code> | 13     |
| <code>getDay()</code>  | za dani datum vraća <b>dan u tjednu</b> (0 za nedjelju, 1 za ponedjeljak, itd.). | <code>Date.getDay()</code>  | <code>const rodendan = new Date("April 13, 2000");</code><br><code>rodendan.getDay() == 4</code>   | 4      |
|                        | Za dani datum vraća  |                             |  |        |

|                                   |   |  |   |  |
|-----------------------------------|---|--|---|--|
| <code>getFullYear()</code>        | godinu. <b>Izbjegavajte</b> metodu <code>getYear()</code> budući da je izgubila podršku i radi pogrešno.  | <code>Date.getFullYear()</code>        | <pre>moonLanding = new Date("July 20, 69 00:20:18"); ; moonLanding.getFullYear() == 1969</pre>  | 1969   |
| <code>getMonth()</code>           | Za dani datum vraća mjesec (0 - Siječanj, 11 - Prosinac)  | <code>Date.getMonth()</code>           | <pre>const moonLanding = new Date('July 20, 69 00:20:18');</pre>  | 6  |
| <code>getHours()</code>           | Za dani datum vraća sate.   | <code>Date.getHours()</code>           | <pre>const xmas95 = new Date("1995-12-25T23:15:30"); ; xmas95.getHours() == 23</pre>  | 23   |
| <code>getMinutes()</code>         | Za dani datum vraća minute.   | <code>Date.getMinutes()</code>         | <pre>const xmas95 = new Date("1995-12-25T23:15:30"); ; xmas95.getMinutes() == 15</pre>  | 15   |
| <code>getSeconds()</code>         | Za dani datum vraća sekunde.  | <code>Date.getSeconds()</code>         | <pre>const xmas95 = new Date("1995-12-25T23:15:30"); ; xmas95.getSeconds() == 30</pre>  | 30   |
| <code>getTime()</code>            | Za dani datum vraća vraća koliko je prošlo milisekundi od 1. siječnja 1970, UTC. Ako je dani datum bio prije, vraća negativan broj.   | <code>Date.getTime()</code>            | <pre>const moonLanding = new Date('July 20, 69 20:17:40 GMT+00:00'); ; moonLanding.getTime() == -14182940000</pre>  | -14182940000   |
| <code>toLocaleDateString()</code> | Za dani datum vraća string prikaz datuma u definiranom lokalnom formatu. Prima opcionalne argumente <code>locales</code> i <code>options</code> . Npr. ako hoćemo datum napisati prema hrvatskom standardu, postavljamo <code>locales='hr'</code> . Ako želimo i datum i vrijeme, postoji varijanta - <code>toLocaleString()</code> .     | <code>Date.toLocaleDateString()</code> | <pre>let bozic23 = new Date("December 25, 23"); bozic23.toLocaleDateString("hr") == '25. 12. 2023.'</pre>   | '25. 12. 2023.'  |
| <code>toLocaleTimeString()</code> | Za dani datum vraća string prikaz vremena u definiranom lokalnom formatu. Prima opcionalne argumente <code>locales</code> i <code>options</code> . Npr. ako hoćemo datum napisati prema američkom standardu, postavljamo <code>locales='en-US'</code> . Ako želimo i datum i vrijeme, postoji varijanta - <code>toLocaleString()</code> . | <code>Date.toLocaleTimeString()</code> | <pre>const event = new Date('August 19, 1975 23:15:30'); ; event.toLocaleTimeString('en-US') == '11:15:30 PM'</pre>                                       | '11:15:30 PM'  |
| <code>toString()</code>           | Pretvara dani <code>Date</code> objekt u string format lokalne vremenske zone. Ova metoda poziva se <b>automatski</b> kod ispisa <code>Date</code> objekta. datuma.   | <code>Date.toString()</code>           | <pre>const event = new Date('August 19, 1975 23:15:30'); ; event.toString() == 'Tue Aug 19 1975 23:15:30 GMT+0100 (Central European Standard Time)'</pre> | 'Tue Aug 19 1975 23:15:30 GMT+0100 (Central European Standard Time)' |
|                                   | Statična metoda koja vraća unix timestamp trenutno vremena prošlog od 1. siječnja   |  |   |  |

|                           |   |                                     |  |  |
|---------------------------|---|-------------------------------------|--|--|
| <code>Date.now()</code>   | 1970, UTC. Budući da je metoda statična, ne stvaramo novi objekt s konstruktorom <code>new Date()</code> .  | <code>Date.now();</code>            | <code>let upravo_sada = Date.now();</code>       | 1708686440160  |
| <code>Date.parse()</code> | Parsira string reprezentaciju datuma i vraća broj milisekundi od 1. siječnja 1970, UTC. Budući da je metoda statična, ne stvaramo novi objekt s konstruktorom <code>new Date()</code> . | <code>Date.parse(dateString)</code> | <code>Date.parse("2024-02-20T14:37:15Z");</code> | 1645265835000<br>(ovisno o vremenskoj zoni, može se razlikovati) |

Tablica se većinom sastoji od `get` metoda za dohvaćanje pojedinih dijelova datuma i vremena. Popis vrlo sličnog skupa `set` metoda za postavljanje dijelova datuma i vremena možete pronaći [ovdje](#).

Što se dešava ako pokušamo "zbrojiti" dva `Date` objekta operatorom `+`?

Rekli smo da se metoda `toString()` automatski poziva kod ispisa `Date` objekta. Kada pokušamo zbrojiti dva `Date` objekta, JavaScript će pretvoriti objekte u stringove i konkatenerirati ih.

```
const d1 = new Date("2022-03-25");
const d2 = new Date("2022-03-26");

console.log(d1 + d2); // "Fri Mar 25 2022 01:00:00 GMT+0100 (Central European Standard Time)Sat Mar 26 2022 01:00:00 GMT+0100 (Central European Standard Time)"
```

Međutim operator `-` će izvršiti matematičku operaciju, odnosno oduzimanje UNIX timestampa jednog datuma od drugog.

Kako smo rekli da je vrijednost UNIX timestampa u milisekundama, rezultat će biti **razlika u milisekundama između dva datuma**.

```
const d1 = new Date("2022-03-25");
const d2 = new Date("2022-03-26");

console.log(d2 - d1); // 86400000
```

## Vježba 4

EduCoder šifra: `vrijeme_u_rh`

- Napišite funkciju `hrDatum()` koja vraća današnji datum u formatu `dd.mm.yyyy`. Funkcija ne prima argumente. Za implementaciju koristite metode iz `Date` objekta. Ispis ne smije sadržavati razmake. Regex izraz za pronalaženje svih razmaka u stringu je `/\s/g`.

 Rezultat:

```
console.log(hrDatum()); // Output: 23.02.2024. (ovisno o trenutnom datumu)
```

- Napišite funkciju `hrVrijeme()` koja vraća trenutno vrijeme u formatu `hh:mm:ss`. Funkcija ne prima argumente. Za implementaciju koristite metode iz `Date` objekta. Ispis ne smije sadržavati razmake.

✓Rezultat:

```
console.log(hrVrijeme()); // Output: 13:08:27 (ovisno o trenutnom vremenu)
```

3. Napišite funkciju `isWeekend()` koja provjerava je li uneseni datum vikend. Funkcija prima jedan argument `datum` koji je tipa `Date`. Funkcija vraća `true` ako je uneseni datum vikend, inače vraća `false`. Za implementaciju koristite metode iz `Date` objekta.

✓Rezultat:

```
console.log(isWeekend(new Date('2024-01-01'))); // Output: false
console.log(isWeekend(new Date('2024-03-31'))); // Output: true
```

## Vježba 5

EduCoder šifra: `calculateHours`

Napišite funkciju `calculateHours()` koja prima dva argumenta: `start` i `end`. Argumenti su tipa `Date`. Funkcija treba izračunati i vratiti razliku između dva datuma u satima. Za implementaciju koristite metode iz `Date` objekta.

✓Rezultat:

```
console.log(calculateHours(new Date(2024, 1, 14), new Date(2024, 1, 16))); // Output: 48
```

## 2.4 Usporedba JavaScript objekata

Naučili smo što su primitivni tipovi podataka, koji su i kako se koriste. Također smo prošli kroz osnovne ugrađene objekte te samu teoriju iza objekata. Također smo naučili da postoje ugrađeni objekti za već postojeće primitivne tipove, poput `String` i `Number` objekata.

Rekli smo da nema smisla komplicirati kôd instanciranjem nekih primitivnih tipova kao objekte, zbog automatske pretvorbe. Na primjer:

```
let x = "Hello"; // primitivni string
let y = new String("Hello"); // String objekt
```

Ili

```
let x = 5; // primitivni broj
let y = new Number(5); // Number objekt
```

Također smo zaključili da će operator `==` uspoređivati primitivne tipove podataka, a operator `===` uspoređivati objekte. No, što ako želimo usporediti dva objekta? Po toj logici, sljedeći primjer bi trebao vratiti `true`:

```
let a = new String("Hello");
let b = new String("Hello");
console.log(x == y); // true ?
console.log(x === y); // true ?
```

No to nije slučaj! Odgovor je jednostavan, objekte nema smisla uspoređivati operatorima `==` i `===` jer će se uspoređivati njihove reference, a ne vrijednosti koje oni sadrže.

**Objekti su referentni tipovi podataka, a primitivni tipovi su vrijednosni tipovi podataka.**

Usporedba objekata na spomenuti način će uvijek rezultirati s `false`, jer uspoređujemo memorijske lokacije gdje su objekti pohranjeni, a one će naravno biti različite.

```
let pet = new Number(5);
let isto_pet = new Number(5);
console.log(pet == isto_pet); // false
console.log(pet === isto_pet); // false

let auto = { marka: "Ford", model: "Mustang" };
let isti_auto = { marka: "Ford", model: "Mustang" };
console.log(auto == isti_auto); // false
console.log(auto === isti_auto); // false
```

Zbog jedinstvenih karakteristika objekata u JavaScriptu postoji i `Object` konstruktor koji se koristi za izradu objekata! No, o tome više na sljedećem predavanju.

## 2.4.1 instanceof operator

Kako možemo jednostavno provjeriti kojem objektu pripada neka varijabla? U prvoj skripti vrlo kratko smo spomenuli `instanceof` operator. `instanceof` operator vraća `true` ako objekt pripada određenom tipu, inače vraća `false`. Sintaksa je sljedeća:

```
object instanceof constructor
```

gdje je `object` objekt koji se provjerava, a `constructor` je funkcija koja opisuje svojstva i metode tog objekta.

Klasični `typeof` operator nam ovdje ne pruža dovoljno informacija, budući da će za sve objekte vratiti `object`. `instanceof` operator nam omogućuje da provjerimo pripada li objekt određenom tipu.

```
let pet = new Number(5);
console.log(pet instanceof Number); // true
console.log(pet instanceof String); // false

console.log(typeof(pet)); // object (ne daje dovoljno informacija)

function Auto(marka, model) {
  this.marka = marka;
  this.model = model;
}
let auto = new Auto("Ford", "Mustang");
```



```
console.log(auto instanceof Auto); // true
console.log(auto instanceof Date); // false

let datum = new Date();
console.log(datum instanceof Date); // true
console.log(datum instanceof String); // false
```

## Samostalni zadatak za vježbu 4

**Napomena:** Ne predaje se i ne boduje se. Zadatak možete i ne morate rješavati u [EduCoder](#) aplikaciji.

**EduCoder šifra:** UNIPU

- Napišite konstruktor za objekt `Grad` koji prima 3 argumenta: `ime`, `brojStanovnika` i `drzava`. Konstruktor treba stvoriti objekt s tim svojstvima.
  - Napišite metodu `ispisi()` koja ispisuje informacije o gradu u formatu: `Ime: [ime], Broj stanovnika: [brojStanovnika], Država: [drzava]`.
  - U konstruktor dodajte metodu `azurirajBrojStanovnika()`.
  - Kada to napravite, dodajte konstruktoru svojstvo `velicina`
  - ažurirajte metodu `ispisi()` da ispisuje i veličinu grada.
- Napišite funkciju `izbaciSamoglasnike()` koja prima rečenicu kao argument i vraća novu rečenicu bez samoglasnika. Za implementaciju koristite metode iz `String` objekta.
- Napišite funkciju `zaokruziBroj()` koja prima dva argumenta: `broj` i `decimale`. Funkcija vraća broj zaokružen na `decimale` decimala. Za implementaciju koristite metode iz `Number` i `Math` objekata.
  - Ako je proslijeđeni argument `broj` već cijeli, funkcija vraća string `Broj je već cijeli!`.
  - Ako je proslijeđeni argument `decimale` manji ili jednak 0, funkcija vraća string `"Pogrešno definirane decimale! Unijeli ste {decimale}, a očekuje se broj veći od 0."`.
- Napišite funkciju `daniOdPocetkaGodine()` koja vraća koliko je dana prošlo od početka godine do trenutnog datuma. Za implementaciju koristite metode iz `Date` objekta.
  - Dodajte poseban uvjet ako je trenutni datum 1. siječnja, funkcija onda vraća `Danas je 1. siječnja!`.
- Definirajte objekt `UNIPUKorisnik` s 3 svojstva: `korisnicko_ime`, `email` i `lozinka`. Napravite konstruktor za objekt `UNIPUKorisnik`. Uz spomenuta svojstva, implementirajte u konstruktor i sljedeće metode:
  - `promijeniEmail()` - prima novi email kao argument i mijenja email korisnika. U metodi morate provjeravati završava li novi email s `@unipu.hr`, ako ne, metoda ispisuje u konzolu: `Email mora završavati s '@unipu.hr'!`. Ako je email ispravan, metoda ispisuje u konzolu poruku: `Email uspješno promijenjen!`. Ako korisnik pokuša promijeniti email na trenutni, metoda ispisuje u konzolu: `Novi email je isti kao stari!`.
  - `promijeniLozinku()` - prima novu lozinku kao argument i mijenja lozinku korisnika. Nova lozinka korisnika mora sadržavati barem 8 znakova, od tog jedan broj i jedan specijalan znak (npr `!`). Za svaki od uvjeta koji nije zadovoljen, metoda mora ispisati odgovarajuću poruku u konzolu. Ako korisnik pokuša promijeniti lozinku na trenutnu, metoda ispisuje u konzolu: `Unijeli ste postojeću lozinku!`.

- u objekt dodajte novo svojstvo `datum_registracije` koje će pohraniti datum i vrijeme registracije korisnika, odnosno datum i vrijeme instanciranja objekta. Datum i vrijeme pohranite u formatu `dd.mm.yyyy. hh:mm:ss` koristeći metodu iz `Date` objekta.

## 3. Polja (eng. *Arrays*)

Polja (eng. **Arrays**) su strukture podataka koje, kao i u drugim programskim jezicima, omogućuju pohranu kolekcije podataka pod varijablom jednog naziva.

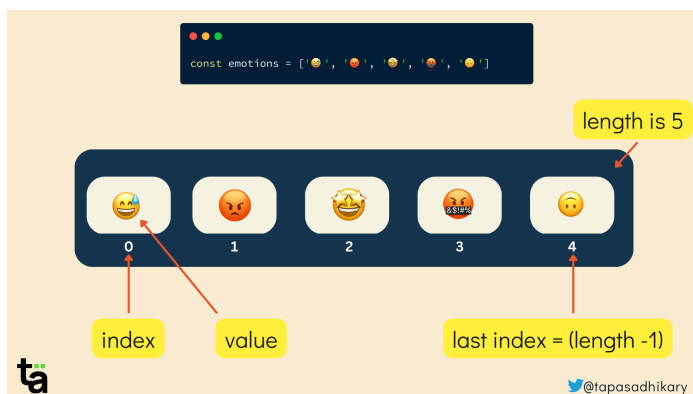
JavaScript polja se razlikuju po tome što mogu sadržavati različite tipove podataka, a ne samo jedan tip, kao što je slučaj u nekim drugim jezicima poput C i C++. Polja u JavaScriptu su dinamičke strukture podataka, što znači da se mogu proširivati i smanjivati tijekom izvođenja programa.

U JavaScriptu, gotovo svi elementi su objekti, pa tako i polja. Polja su ustvari specijalni tip objekata `Array` koji se koristi za pohranu više vrijednosti u jednoj varijabli. Ne možemo li to i s objektima? Možemo, ali polja su specijalizirana za pohranu više vrijednosti, dok su objekti specijalizirani za pohranu više parova

`ključ:vrijednost`.

**Polja nisu primitivi!** Polja su `Array` objekti s nekoliko ključnih karakteristika:

- Polja su **indeksirana** struktura podataka, što znači da svaki element polja ima svoj indeks. Indeksi počinju od 0, zadnji element je indeksiran s `length - 1`.
- Polja su **dinamička** struktura podataka, što znači da se mogu proširivati i smanjivati tijekom izvođenja programa.
- Polja mogu sadržavati **različite tipove podataka**, primjerice brojeve, stringove, objekte, druga polja, funkcije, itd.
- Polja nisu asocijativna, što znači da **nemaju ključeve**, već samo **indekse** koji su nenegativni cijeli brojevi.



Izvor: <https://bugfender.com/blog/javascript-arrays-guide/>

### 3.1 Sintaksa polja

Polja deklariramo koristeći uglate zagrade `[]`. Elementi polja se odvajaju zarezom. Polja mogu sadržavati različite tipove podataka, uključujući i druga polja.

Sintaksa:

```
let ime_polja = [element1, element2, element3, ...];
```

Razmaci i novi redovi nisu bitni, ali se preporučuje formatiranje kôda radi bolje čitljivosti. Moguće je polje deklarirati i ovako:

```
let ime_polja = [  
  element1,  
  element2,  
  element3,  
];
```

### 3.1.1 Pristup elementima polja

Moguće je napraviti prazno polje, i onda elemente dodavati naknadno. Elementima polja pristupamo preko **indeksa**, koji se nalazi u uglatim zagradama. Indeksi su cjelobrojne vrijednosti koji počinju od `0`, a zadnji element je indeksiran s `length - 1`. Dodavanje novog elementa u polje možemo izvesti na jednak način kao dodavanje novog svojstva u objekt, samo što se u ovom slučaju koristi indeks umjesto ključa.

```
let namirnice = [];  
namirnice[0] = "kruh";  
namirnice[1] = "mlijeko";  
namirnice[2] = "sir";
```

Ispis polja u konzolu možemo napraviti koristeći `console.log()` metodu. Ispis polja u konzolu će rezultirati ispisom svih elemenata polja, odvojenih zarezom.

```
console.log(namirnice); // Output: ["kruh", "mlijeko", "sir"] //Primjetite uglate zagrade,  
to je ispis polja  
  
console.log(namirnice[0]); // Output: "kruh"  
console.log(namirnice[1]); // Output: "mlijeko"  
console.log(namirnice[2]); // Output: "sir"
```

Dodavanje i pristup elementima s operatorom `.` nije moguće!

```
let namirnice = [];  
namirnice.0 = "kruh"; // SyntaxError: Unexpected number
```

### 3.1.2 Veličina polja

Veličinu polja možemo dohvatiti koristeći `length` svojstvo, kao i kod znakovnih nizova. `length` svojstvo vraća broj elemenata polja.

```
let namirnice = ["kruh", "mlijeko", "sir"];
console.log(namirnice.length); // Output: 3

namirnice[3] = "jaja";
console.log(namirnice.length); // Output: 4

namirnice[5] = "riža";
console.log(namirnice.length); // Output: 5 ili 6 ?
```

Nakon dodavanja elementa na indeks 5, `length` svojstvo će vratiti `6`, iako polje ima samo 5 elemenata. JavaScript automatski dodaje prazne elemente između indeksa 5 i 4. Ovo je jedna od karakteristika dinamičkih polja u JavaScriptu.

```
console.log(namirnice); // Output: ["kruh", "mlijeko", "sir", "jaja", empty, "riža"]
console.log(namirnice[4]); // Output: undefined
```

U polje, kao što smo već rekli, možemo dodavati različite tipove podataka, uključujući i druge objekte, funkcije, itd.

Primjer:

```
let mjesovito_polje = [1, "string", true, {ime: "Ivan", godine: 25}, function()
{console.log("Pozdrav iz funkcije!")}];
console.log(mjesovito_polje); // Output: [1, "string", true, {ime: "Ivan", godine: 25}, f
()]
```

U `C` i `Java` jezicima ovo nije moguće, budući da su polja u tim jezicima statičke strukture podataka, što znači da moraju sadržavati isti tip podataka. Međutim, i u JavaScriptu se preporučuje korištenje polja s istim tipom podataka, radi bolje čitljivosti i održavanja kôda. **Izbjegavajte mješovita polja!** Za mješovite tipove podataka koriste se **objekti**.

### 3.1.3 Izmjene u polju

Elemente u polje možemo dodavati čak i ako smo ga deklarirali kao konstantu. Isto tako, možemo mijenjati i brisati elemente iz polja.

```
const voće = ["jabuka", "kruška", "šljiva"];
voće[0] = "banana"; // voće = ["banana", "kruška", "šljiva"]
voće[3] = "naranča"; // voće = ["banana", "kruška", "šljiva", "naranča"]
```

Zašto je ovo moguće? Konstanta `voće` sadrži referencu na polje, a ne samo polje. Referenca se ne može mijenjati, ali se može mijenjati sadržaj na koji referenca pokazuje. Što ako pokušamo promijeniti referencu na skroz novo polje? To ne možemo!

```
const voće = ["jabuka", "kruška", "šljiva"];
voće = ["ananas", "kivi", "mango"]; // TypeError: Assignment to constant variable.
```

### 3.1.4 Array objekt sintaksa

Rekli smo da su polja ustvari `Array` objekti. Dakle, možemo stvoriti novo polje na isti način kao i bilo koji drugi objekt, pozivanjem konstruktora ključnom riječi `new`.

```
let voće = new Array("jabuka", "kruška", "šljiva");
let isto_voće = ["jabuka", "kruška", "šljiva"];

console.log(voće); // Output: ["jabuka", "kruška", "šljiva"]
console.log(isto_voće); // Output: ["jabuka", "kruška", "šljiva"]

typeof voće; // Output: "object"
typeof isto_voće; // Output: "object"

voće == isto_voće; // Output: false - različite reference
```

Vidimo da će `typeof` u oba slučaja vratiti `object`. `typeof` neće vratiti `array` kao što bi se očekivalo, budući da su polja ustvari objekti 🤖.

## 3.2 Zašto `Array` objekt?

Možemo si postaviti pitanje zašto koristiti `Array` objekt, ako možemo koristiti obične uglate zagrade. Kroz tu notaciju možemo dodavati elemente u polje, mijenjati ih, brisati, dohvaćati, itd. Koja je onda smisao `Array` objekta?

### Primjer 1 - dodavanje, brisanje i pretraživanje koristeći obične uglate zagrade

Imamo polje `stabla` koje sadrži nekoliko poznatih vrsta stabala u Hrvatskoj.

```
let stabla = ["hrast", "bukva", "javor", "bor", "smreka"];
```

Kako bismo dodali novi element u polje, moramo znati koliko elemenata polje trenutno sadrži, kako ne bi došlo do preklapanja indeksa i gubitka podataka.

```
let duljina = stabla.length; // duljina = 5
stabla[duljina] = "jela"; // Možemo, zato što je duljina polja 5, a indeks zadnjeg elementa je 4
console.log(stabla); // Output: ["hrast", "bukva", "javor", "bor", "smreka", "jela"]
```

Kako bismo izbrisali element iz polja, moramo znati indeks elementa kojeg želimo izbrisati.

```
delete stabla[2]; // stabla = ["hrast", "bukva", empty, "bor", "smreka", "jela"]
console.log(stabla); // Output: ["hrast", "bukva", empty, "bor", "smreka", "jela"]
```

Primjećujemo da je `delete` operator ostavio prazno mjesto na indeksu 2, umjesto da je izbrisao element.

Što ako želimo izbrisati zadnji element iz polja?

```
delete stabla[stabla.length - 1]; // stabla = ["hrast", "bukva", empty, "bor", "smreka",  
empty] - isti problem
```

Kako možemo pretraživati naše polje?

Polja su iterabilna struktura podataka, što znači da možemo koristiti petlje za prolazak kroz elemente polja.

```
let stabla = ["hrast", "bukva", "javor", "bor", "smreka"];  
for (let i = 0; i < stabla.length; i++) {  
    console.log(stabla[i]); // Output: "hrast", "bukva", "javor", "bor", "smreka"  
}
```

Recimo da hoćemo zaustaviti pretraživanje kada naiđemo na element `bor`. Kako bismo to napravili, koristimo `break` naredbu.

```
let stabla = ["hrast", "bukva", "javor", "bor", "smreka"];  
for (let i = 0; i < stabla.length; i++) {  
    if (stabla[i] == "bor") {  
        console.log("Pronašli smo bor!");  
        break;  
    }  
}
```

Naporno je svaki put računati indekse kako bi dodali novi element u polje, a i `delete` operator ne radi kako bi trebao. `Array` objekt nudi gotove metode za sve ove operacije, kao i mnoge druge. U većini slučajeva je bolje koristiti `Array` objekt, jer je brži i sigurniji, a kôd je mnogo čitljiviji!

## Primjer 2 - dodavanje, brisanje i pretraživanje koristeći `Array` objekt

Napravimo novo polje `stabla` koristeći `Array` objekt.

```
let stabla = new Array("hrast", "bukva", "javor", "bor", "smreka");
```

Kako bismo dodali novi element u polje, koristimo jednostavno `push()` metodu koja dodaje novi element na kraj polja. Ne moramo brinuti o indeksima, jer će `push()` metoda sama pronaći zadnji indeks i dodati novi element na kraj polja.

```
stabla.push("jela"); // To je to.  
console.log(stabla); // Output: ["hrast", "bukva", "javor", "bor", "smreka", "jela"]
```

Kako bismo izbrisali element iz polja, koristimo `pop()` metodu koja briše zadnji element iz polja.

```
stabla.pop(); // Briše zadnji element iz polja - "jela"  
console.log(stabla); // Output: ["hrast", "bukva", "javor", "bor", "smreka"] // "jela" je  
potpuno izbrisan, nema više praznog mjesta
```

Pretraživanje polja možemo napraviti koristeći `forEach()` metodu, koja prolazi kroz svaki element polja i izvršava zadanu funkciju za svaki element.

```
stabla.forEach(function(stablo) { // stablo je lokalna varijabla koja sadrži trenutni
    element polja. Ova funkcija naziva se callback funkcija, a koristi se u mnogim metodama
    polja.
    console.log(stablo); // Output: "hrast", "bukva", "javor", "bor", "smreka"
});
```

Recimo da hoćemo pretražiti polje s ciljem pronalaska elementa `bor`. Koristimo `find()` metodu koja vraća prvi element koji zadovoljava uvjet koji je definiran u `callback` funkciji.

```
let bor = stabla.find(function(stablo) {
    return stablo == "bor"; // vraća prvi element koji zadovoljava ovaj uvjet
});
console.log(bor); // Output: "bor"
```

Primjetite koliko je kôd čitljiviji i jednostavniji za razumijevanje 😊

Neke metode moguće je doslovno čitati prirodnim jezikom, na primjer sljedeći primjer čitamo: "Za svaki element polja `stabla` ispiši pojedino `stablo`"

```
let stabla = ["hrast", "bukva", "javor", "bor", "smreka"];
stabla.forEach(function(stablo) {
    console.log(stablo);
});
```

## Vježba 6

EduCoder šifra: `pliz_moze_2`

Napravite novo polje `ocjene_mat` koje sadrži ocjene iz matematike. U polje dodajte 10 ocjena: 5, 4, 3, 1, 2, 4, 5, 1, 4, 5. Ispišite polje u konzolu. Za negativne ocjene ispišite poruku: `Ocjena na poziciji polja [pozicija] je negativna!`. Nakon šta to napravite, iterirajte kroz polje još jednom i ispravite negativne ocjene na 2. Sumirajte sve ocjene i izračunajte prosjek. Ispišite `ново polje`, `sumu` ocjena i `prosjek` u konzolu.

## 3.2 Iteracije kroz polja

Kao što smo već spomenuli, polja su iterabilna struktura podataka, što znači da možemo koristiti petlje za prolazak kroz elemente polja.

Već smo se upoznali s `for` petljom i klasičnim načinom prolaska kroz sve elemente polja (`for let i=0; i < polje.length; i++`). Međutim, JavaScript nudi mnoge druge načine iteracije kroz polja, npr. `forEach()` metoda koja prolazi kroz svaki element polja i izvršava zadanu funkciju za svaki element.

No, krenimo od jednostavnijih principa, bez korištenja `callback` funkcija.

### 3.2.1 Tradicionalna `for` petlja

Tradicionalna `for` petlja, koju smo već koristili u prethodnim predavanjima, može se koristiti za prolazak kroz sve elemente polja, kao i za izmjene elemenata polja.

```
let polje = ["jabuka", "kruška", "šljiva", "naranča", "banana"];
for (let i = 0; i < polje.length; i++) { // Iteriramo za veličinu polja
  console.log(polje[i]); // Output: "jabuka", "kruška", "šljiva", "naranča", "banana"
}
```

Možemo svaki element izmjeniti u petlji, na primjer, npr. svakom elementu dodati prefiks `fruit_`.

```
let polje = ["jabuka", "kruška", "šljiva", "naranča", "banana"];
for (let i = 0; i < polje.length; i++) { // Iteriramo za veličinu polja
  polje[i] = "fruit_" + polje[i]; // Na ovaj način možemo jednostavno mijenjati elemente polja
}
console.log(polje); // Output: ["fruit_jabuka", "fruit_kruška", "fruit_šljiva", "fruit_naranča", "fruit_banana"]
```

## 3.2.2 `for...of` petlja

`for...of` petlja je novi način iteracije kroz polja koji je uveden u ES6 standardu JavaScripta. `for...of` petlja prolazi kroz sve elemente iterabilnih objekata (eng. **iterables**), uključujući polja (`Array`) i znakovne nizove (`String`) (ima ih još).

Sintaksa je sljedeća:

```
for (let element of iterable) {
  // blok kôda koji se izvršava za svaki element
}
```

`element` je lokalna varijabla proizvoljnog naziva koja sadrži trenutni element iterabilnog objekta, a `iterable` je iterabilni objekt kroz koji prolazimo.

Kako možemo iterirati kroz naše polje voća?

```
let voće = ["jabuka", "kruška", "šljiva", "naranča", "banana"];
for (let voćka of voće) { // `voćka` je lokalna varijabla proizvoljnog naziva koja sadrži trenutni element polja
  console.log(voćka); // Output: "jabuka", "kruška", "šljiva", "naranča", "banana"
}
```

Ili možemo koristiti `for...of` petlju za iteraciju kroz znakovni niz.

```
let ime = "Ivan";
for (let slovo of ime) { // `slovo` je lokalna varijabla proizvoljnog naziva koja sadrži trenutni znak u nizu
  console.log(slovo); // Output: "I", "v", "a", "n"
}
```



### 3.2.3 `for...in` petlja

`for...in` petlja se koristi za **iteraciju kroz svojstva objekta**. Međutim, može se koristiti i za iteraciju kroz indekse polja.

Sintaksa je sljedeća:

```
for (let key in object) {  
    // blok kôda  
}
```

`key` je lokalna varijabla proizvoljnog naziva koja sadrži ključ objekta, a `object` je objekt kroz koji "prolazimo".

```
let voće = ["jabuka", "kruška", "šljiva", "naranča", "banana"];  
for (let indeks in voće) { // `indeks` je lokalna varijabla proizvoljnog naziva koja sadrži  
    indeks polja  
    console.log(indeks); // Output: "0", "1", "2", "3", "4"  
}
```

Međutim, uzmimo za primjer objekt `auto` s prošlih vježbi:

```
const auto = {  
    marka: "Ford",  
    model: "Mustang",  
    godina_proizvodnje: 2020,  
    boja: "Crna",  
};  
for (let svojstvo in auto) {  
    console.log(svojstvo); // Output: "marka", "model", "godina_proizvodnje", "boja"  
}  
  
for (let svojstvo in auto) {  
    console.log(auto[svojstvo]); // Output: "Ford", "Mustang", 2020, "Crna"  
}
```

Zašto je ovo povezano s poljima? Kao što smo već rekli, polja su ustvari objekti, a indeksi polja su ključevi objekta. Zato možemo koristiti `for...in` petlju za iteraciju kroz indekse polja. Međutim, `for...in` petlja nije preporučena za iteraciju kroz polja, već se preporučuje korištenje klasične `for`, `for...of` ili `Array.forEach` petlje.

### 3.2.4 `Array.forEach` metoda

`Array.forEach` metoda je metoda koja prolazi kroz sve elemente polja i izvršava zadanu funkciju za svaki element. `Array.forEach` metoda je jednostavna za korištenje i često se koristi za iteraciju kroz polja.

Sintaksa je sljedeća:

```
polje.forEach(callbackFn)
```

`callback` funkcija je funkcija koja se izvršava za svaki element polja. `callback` funkcija prima tri argumenta: `element`, `index` i `array`. `element` je trenutni element polja, `index` je indeks trenutnog elementa, a `array` je polje koje se prolazi.

```
polje.forEach(function(element, index, array) {  
    // blok kôda koji se izvršava za svaki element  
});
```

Primjerice imamo polje `slova` koje sadrži nekoliko slova. U sljedećem primjeru ispisat ćemo elemente `callback` funkcije u konzolu.

```
let slova = ["a", "b", "c",];  
slova.forEach(function (trenutnaVrijednost, indeks, polje) { // primjetite da u callback  
    funkciji možemo koristiti bilo koje ime za argumente  
    console.log(  
        "Vrijednost: " + trenutnaVrijednost,  
        "Indeks: " + indeks,  
        "Cijelo polje: " + polje  
    );  
});  
// Vrijednost: a Indeks: 0 Cijelo polje: a,b,c  
// Vrijednost: b Indeks: 1 Cijelo polje: a,b,c  
// Vrijednost: c Indeks: 2 Cijelo polje: a,b,c
```

Ne moramo pozvati sve argumente `callback` funkcije, možemo koristiti samo one koji su nam potrebni.

```
let slova = ["a", "b", "c",];  
slova.forEach(function (trenutnaVrijednost) {  
    console.log("Slovo: " + trenutnaVrijednost);  
});  
// Slovo: a  
// Slovo: b  
// Slovo: c
```

Naša `callback` funkcija u danim primjerima samo ispisuje vrijednosti u konzolu, ali možemo koristiti `callback` funkciju za bilo koju operaciju koja nam je potrebna, primjerice za izračunavanje sume, prosjeka, filtriranje, itd. O tome ćete naučiti više na predavanjima i vježbama iz ugniježđenih struktura i naprednih funkcija...

### 3.3 Objekti unutar polja

Rekli smo da polja mogu sadržavati različite tipove podataka, uključujući i druge objekte.

Uzmimo za primjer polje `korisnici` koje sadrži nekoliko objekata `Korisnik`. Možemo iskoristiti konstruktor `Korisnik` koji smo definirali u prethodnom poglavlju.

```
function Korisnik(ime, prezime, godina_rodenja) {
  this.ime = ime;
  this.prezime = prezime;
  this.godina_rodenja = godina_rodenja;
  this.predstaviSe = function () {
    console.log(
      `Bok! Ja sam ${this.ime} ${this.prezime}. Rođen/a sam ${this.godina_rodenja} godine.`
    );
  };
}
```

Izradimo nekoliko korisnika pozivanjem konstruktora...

```
let korisnik1 = new Korisnik("Ivan", "Ivić", 1995);
let korisnik2 = new Korisnik("Marko", "Markić", 1990);
let korisnik3 = new Korisnik("Ana", "Anić", 1985);
```

...i dodajmo ih u polje `korisnici`.

```
let korisnici = [korisnik1, korisnik2, korisnik3];
console.log(korisnici); // Output: [Korisnik, Korisnik, Korisnik]
                        // 0: Korisnik {ime: "Ivan", prezime: "Ivić", godina_rodenja: 1995,
predstaviSe: f}
                        // 1: Korisnik {ime: "Marko", prezime: "Markić", godina_rodenja:
1990, predstaviSe: f}
                        // 2: Korisnik {ime: "Ana", prezime: "Anić", godina_rodenja: 1985,
predstaviSe: f}
```

## Primjer 3 - iteracija kroz polje objekata

Koristeći `for`, `for...of` ili `for...in` petlje, kao i `Array.forEach` metodu, možemo iterirati kroz polje i pozvati metodu `predstaviSe()` za svakog korisnika.

Idemo prvo iterirati kroz polje objekata koristeći `for` petlju.

```
for (let i = 0; i < korisnici.length; i++) {
  korisnici[i].predstaviSe(); //Output: Bok! Ja sam Ivan Ivić. Rođen/a sam 1995 godine.
                             //      Bok! Ja sam Marko Markić. Rođen/a sam 1990 godine.
                             //      Bok! Ja sam Ana Anić. Rođen/a sam 1985 godine.
}
```

- koristeći `for...in` petlju.

```
for (let i in korisnici) {
  korisnici[i].predstaviSe(); //Output: kao i u prethodnom primjeru
}
```

- koristeći `for...of` petlju.


```
for (let korisnik of korisnici) {  
  korisnik.predstaviSe(); //Output: kao i u prethodnom primjeru  
}
```

- koristeći `Array.forEach` metodu.

```
korisnici.forEach(function (korisnik) {  
  korisnik.predstaviSe(); //Output: kao i u prethodnom primjeru  
});
```

Glavna ideja polja je da pohranjujemo više istvornih podataka pod jednim nazivom te da imamo mogućnost iteracije i primjene metoda na svakom elementu polja. Ono što ne želimo je raditi polja koja sadrže različite tipove podataka, kao što smo već rekli, na primjer.

```
let korisnik = ["Ivan", "Ivić", 1995, function() {console.log("Pozdrav ja sam Ivan!")}]; //
```



Kako pristupiti imenu ovog korisnika?

```
console.log(korisnik[0]); // Output: "Ivan"
```

Međutim, ne znamo je li to ime, prezime, godina rođenja ili funkcija. Ovo je jako loša praksa i treba je izbjegavati.

Napravite objekt kada imate potrebu pohraniti ključ-vrijednost parove, a polje kada imate potrebu pohraniti više istovrsnih podataka.

```
let korisnik_ivan = { // ✓  
  ime: "Ivan",  
  prezime: "Ivić",  
  godina_rođenja: 1995,  
  predstaviSe: function() {  
    console.log(`Pozdrav ja sam ${this.ime}!`);  
  }  
};
```

Sada možemo korisnika pohraniti u polje, npr. `korisnici`.

```
let korisnici = [];  
korisnici.push(korisnik_ivan);  
console.log(korisnici[0].ime); // Output: "Ivan"
```

## Vježba 7

EduCoder šifra: `grocery_shopping`

Napravite novo polje `groceryList` koje će sadržavati objekte `Namirnica`. Objekt `Namirnica` mora se sastojati od svojstava: `ime`, `cijena` i `količina`. Prvo definirajte konstruktor `Namirnica` i dodajte svojstva. Napravite nekoliko namirnica i dodajte ih u polje `groceryList`. Dodajte novu metodu `ukupno()` u konstruktor koja će računati ukupnu cijenu za pojedinu namirnicu. Iterirajte kroz polje `groceryList` i ispišite sve namirnice u konzolu, kao i ukupnu cijenu za svaku namirnicu. Dodajte globalnu funkciju `shoppingUkupno(groceryList)` koja će kao argument primiti polje `groceryList`, izračunati ukupnu cijenu za sve namirnice i ispisati je u konzolu.

Primjer rezultata:

```
"Za namirnicu kruh trebate izdvojiti 3 eur."
"Za namirnicu mlijeko trebate izdvojiti 2 eur."
"Za namirnicu jaja trebate izdvojiti 3 eur."
"Ukupno za sve namirnice trebate izdvojiti 8 eur."
```

### 3.4 Osnovne metode `Array` objekta

Do sad smo spomenuli nekoliko osnovnih metoda `Array` objekta, kao što su `push()`, `pop()`, `forEach()`, itd. U ovom poglavlju ćemo se upoznati s još nekoliko jednostavnijih metoda `Array` objekta.

#### 3.4.1 Metode dodavanja, brisanja i stvaranja novih polja

| Metoda                  | Objašnjenje  | Sintaksa  | Primjer   | Output  |
|-------------------------|--|---|---|---|
| <code>length</code>     | Radi se o svojstvu, ne metodi. Dakle pozvat ćemo ju bez <code>()</code> operatora. Vraća veličinu polja kao cjelobrojnu vrijednost.  | <code>Array.length</code>                                 | <pre>const fruits = ["Banana", "Orange", "Apple", "Mango"]; fruits.size == 4</pre>                                  | 4   |
| <code>toString()</code> | Vraća polje u string obliku, gdje su vrijednosti odvojene zarezima.  | <code>Array.toString()</code>                             | <pre>const fruits = ["Banana", "Orange", "Apple", "Mango"]; fruits.toString();</pre>                                | 'Banana,Orange,Apple,Mango'   |
| <code>at()</code>       | Vraća vrijednost na danom indeksu u parametru <code>index</code> . Funkcija je implementirana u ES2022 standardu i ima isto ponašanje kao dohvaćanje elemenata koristeći <code>[]</code> . Ono što nije bilo moguće je tzv. <code>negative bracket indexing</code> , npr. dohvaćanje zadnjeg elementa u polju koristeći <code>[-1]</code> . Funkcija <code>Array.at()</code> rješava taj nedostatak. | <code>Array.at(index)</code>                              | <pre>const fruits = ["Banana", "Orange", "Apple", "Mango"]; fruits.at(2) == "Apple"; fruits.at(-1) == "Mango"</pre> | "Apple" ; "Mango"   |
| <code>join()</code>     | Metoda spaja elemente polja u jedinstveni string. Radi kao <code>toString()</code> metoda, ali se dodatno može definirati <code>separator</code> koji će odvajati elemente u novom stringu.  | <code>Array.join(separator)</code>                        | <pre>const elements = ['Fire', 'Air', 'Water']; elements.join('-') == "Fire-Air-Water"</pre>                        | "Fire-Air-Water"  |
| <code>push()</code>     | Metoda dodaje novi element/elemente na kraj polja, a kao povratnu vrijednost vraća veličinu polja <code>Array.length</code>  | <code>Array.push(element1, element2, ... elementN)</code> | <pre>const elements = ['Fire', 'Air', 'Water']; let count = elements.push("Earth");</pre>                           | <pre>elements = ["Fire", "Air", "Water", "Earth"]; count == 4</pre> |
| <code>pop()</code>      | Metoda briše zadnji element u polju, a kao povratnu vrijednost   | <code>Array.pop()</code>                                  | <pre>const plants = ['broccoli', 'cauliflower', 'cabbage', 'kale',</pre>  | tomato  |

|                        |  |  |  |  |
|------------------------|--|--|--|--|
|                        | vrća obrisani element.   |  | <pre>'tomato']; ; let deleted = plants.pop()</pre>   |  |
| <code>shift()</code>   | Metoda briše prvi element u polju, a kao povratnu vrijednost vraća obrisani element. Preostale elemente pomiče "ulijevo" na manji indeks, kako bi se riješilo prazno prvo mjesto.  | <code>Array.shift()</code>   | <pre>const plants = ['broccoli', 'cauliflower', 'cabbage', 'kale', 'tomato']; let shifted = plants.shift()</pre>   | <code>broccoli</code>  |
| <code>unshift()</code> | Metoda dodaje novi element/elemente na početak polja, i pomiče ostale elemente "udesno" za onoliko indeksa koliko je elemenata ubačeno. Vraća <code>Array.length</code> svojstvo poput metode <code>Array.push</code> .  | <code>Array.unshift(element1, element2, ... elementN)</code>           | <pre>const numbers = [1, 2, 3]; let count = numbers.unshift(4, 5);</pre>   | <pre>numbers = [4, 5, 1, 2, 3]; count = 5</pre>              |
| <code>concat()</code>  | Metoda spaja 2 ili više polja bez da mijenja originalna polja. Vraća novo-izrađeno polje.  | <code>Array.concat(Array1, Array2, ... ArrayN)</code>                  | <pre>const array1 = ['a', 'b', 'c']; const array2 = ['d', 'e', 'f']; ; const array3 = array1.concat(array2);</pre> | <pre>array3 = ["a", "b", "c", "d", "e", "f"]</pre>           |
| <code>slice()</code>   | Metoda stvara novo polje kao podskup originalnog, definirano <code>start</code> (gdje počinje ekstrakcija) i <code>end</code> (gdje završava ekstrakcija) parametrima - <code>[start, end)</code> . Ne mijenja originalno polje i vraća novo "podskup polje". Ako se pozove bez parametara, kopira cijelo polje. | <code>Array.slice(start, end)</code>                                   | <pre>const animals = ['ant', 'bison', 'camel', 'duck', 'elephant']; ; animals2 = animals.slice(2, 4) ;</pre>       | <pre>animals2 = ["camel", "duck"]</pre>                      |
| <code>splice()</code>  | Metoda mijenja sadržaj polja dodavanjem/brisanjem elemenata. Vraća obrisane elemente u novom polju. Parametri su <code>start</code> (gdje počinje promjena), <code>deleteCount</code> (koliko elemenata treba obrisati od <code>start</code> ) i <code>item1, item2, ...</code> (elementi koji se dodaju).       | <code>Array.splice(start, deleteCount, item1, item2, ... itemN)</code> | <pre>const months = ['Jan', 'March', 'April', 'June']; ; months.splice(1, 0, 'Feb');</pre>                         | <pre>months = ["Jan", "Feb", "March", "April", "June"]</pre> |

Metode `push()`, `pop()`, `shift()` i `unshift()` su metode koje se koriste za dodavanje i brisanje elemenata polja. Metode `concat()` i `slice()` su metode koje se koriste za stvaranje novih polja. Metoda `splice()` je metoda koja se koristi za mijenjanje sadržaja polja dodavanjem/brisanjem elemenata.

## Primjer 4 - `paginate` funkcija koristeći `slice` metodu

EduCoder šifra: `paginate`

Recimo da radimo na web stranici koja prikazuje objave korisnika. Kako ne bi preopterećivali korisnika s previše objava, želimo prikazati samo 5 objava po stranici od ukupno 100 objava. Kako bismo to napravili, koristimo `slice` metodu koja će nam omogućiti da izradimo "podskup" polja koji će sadržavati samo po 5 objava. Implementirati ćemo funkciju `paginate` koja će uzeti polje objava, trenutnu stranicu i broj objava po stranici, i vratiti "podskup" polja koji će sadržavati objave za trenutnu stranicu. Funkcija mora raditi za svaki broj objava po stranici i za svaku stranicu, kao i za bilo koji ukupni broj objava.

Prvo ćemo definirati nekoliko varijabli:

```

const objave = []; //Zamislite da je ovo polje koje sadrži 100 objava korisnika. Objave
mogu biti custom objekti, npr. {naslov: "Naslov objave", sadržaj: "Sadržaj objave", autor:
"Ime autora", datum: "Datum objave"}

const ukupnoObjava = 100; //Ukupan broj objava

for (let i = 1; i <= ukupnoObjava; i++) { // Dodajemo 100 dummy objava u polje
  objave.push({naslov: `Naslov objave ${i}`, sadržaj: `Sadržaj objave ${i}`, autor: `Ime
autora ${i}`, datum: `Datum objave ${i}`});
}

const trenutnaStranica = 1; //Trenutna stranica na kojoj se korisnik nalazi
const objavePoStranici = 5; //Broj objava koje želimo prikazati po stranici
const objaveNaTrenutnojStranici = paginate(objave, trenutnaStranica, objavePoStranici);

```

`paginate` funkciju možemo implementirati na sljedeći način. Zapamtite da je `startIndex` uključen, a `endIndex` nije uključen u "podskup" polja.

```

function paginate(objave, trenutnaStranica, objavePoStranici) {
  const startIndex = (trenutnaStranica - 1) * objavePoStranici; //Računamo indeks od kojeg
počinje "podskup" polja. Ako je trenutna stranica 1, onda je startIndex 0, ako je trenutna
stranica 2, onda je startIndex 5, itd.
  const endIndex = trenutnaStranica * objavePoStranici; //Računamo indeks na kojem završava
"podskup" polja. Ako je trenutna stranica 1, onda je endIndex 5, ako je trenutna stranica
2, onda je endIndex 10, itd.
  return objave.slice(startIndex, endIndex); //Vraćamo "podskup" [startIndex, endIndex)
polja koji sadrži objave za trenutnu stranicu
}

```

✓ Rezultat:

```

const trenutnaStranica = 1;
const objavePoStranici = 5;
const objaveNaTrenutnojStranici = paginate(objave, trenutnaStranica, objavePoStranici);
console.log(objaveNaTrenutnojStranici);

//Output:
// [
//   {naslov: "Naslov objave 1", sadržaj: "Sadržaj objave 1", autor: "Ime autora 1", datum:
"Datum objave 1"},
//   {naslov: "Naslov objave 2", sadržaj: "Sadržaj objave 2", autor: "Ime autora 2", datum:
"Datum objave 2"},
//   {naslov: "Naslov objave 3", sadržaj: "Sadržaj objave 3", autor: "Ime autora 3", datum:
"Datum objave 3"},
//   {naslov: "Naslov objave 4", sadržaj: "Sadržaj objave 4", autor: "Ime autora 4", datum:
"Datum objave 4"},
//   {naslov: "Naslov objave 5", sadržaj: "Sadržaj objave 5", autor: "Ime autora 5", datum:
"Datum objave 5"}
// ]

const trenutnaStranica = 2;

```

```

const objavePoStranici = 10;
const objaveNaTrenutnojStranici = paginate(objave, trenutnaStranica, objavePoStranici);
console.log(objaveNaTrenutnojStranici);

//Output:
// [
//   {naslov: "Naslov objave 11", sadržaj: "Sadržaj objave 11", autor: "Ime autora 11",
// datum: "Datum objave 11"},
//   {naslov: "Naslov objave 12", sadržaj: "Sadržaj objave 12", autor: "Ime autora 12",
// datum: "Datum objave 12"},
//   {naslov: "Naslov objave 13", sadržaj: "Sadržaj objave 13", autor: "Ime autora 13",
// datum: "Datum objave 13"},
//   {naslov: "Naslov objave 14", sadržaj: "Sadržaj objave 14", autor: "Ime autora 14",
// datum: "Datum objave 14"},
//   {naslov: "Naslov objave 15", sadržaj: "Sadržaj objave 15", autor: "Ime autora 15",
// datum: "Datum objave 15"},
//   {naslov: "Naslov objave 16", sadržaj: "Sadržaj objave 16", autor: "Ime autora 16",
// datum: "Datum objave 16"},
//   {naslov: "Naslov objave 17", sadržaj: "Sadržaj objave 17", autor: "Ime autora 17",
// datum: "Datum objave 17"},
//   {naslov: "Naslov objave 18", sadržaj: "Sadržaj objave 18", autor: "Ime autora 18",
// datum: "Datum objave 18"},
//   {naslov: "Naslov objave 19", sadržaj: "Sadržaj objave 19", autor: "Ime autora 19",
// datum: "Datum objave 19"},
//   {naslov: "Naslov objave 20", sadržaj: "Sadržaj objave 20", autor: "Ime autora 20",
// datum: "Datum objave 20"}
// ]

```

## 3.4.2 Metode pretraživanja polja

| Metoda                     | Objašnjenje   | Sintaksa   | Primjer   | Output |
|----------------------------|---|--|---|--------|
| <code>indexOf()</code>     | Metoda pretražuje polje za dani <code>searchElement</code> i vraća indeks prvog pronađenog elementa, ili <code>-1</code> ako element nije pronađen. Prima i opcionalni parametar <code>fromIndex</code> preko kojeg se može definirati od kojeg indeksa da se pretražuje. | <code>Array.indexOf(searchElement, fromIndex)</code>     | <pre>const beasts = ['ant', 'bison', 'camel', 'duck', 'bison']; beasts.indexOf('bison') == 1</pre>            | 1      |
| <code>lastIndexOf()</code> | Metoda pretražuje polje za dani <code>searchElement</code> i vraća indeks zadnjeg pronađenog elementa, ili <code>-1</code> ako element nije pronađen. Prima i opcionalni parametar <code>fromIndex</code> preko kojeg se može definirati od kojeg indeksa da se           | <code>Array.lastIndexOf(searchElement, fromIndex)</code> | <pre>const animals = ['Elephant', 'Tiger', 'Penguin', 'Elephant']; animals.lastIndexOf('Elephant') == 3</pre> | 3      |



|                              | pretražuje <b>unazad</b> .  |   |   |  |
|------------------------------|---|---|---|--|
| <code>includes()</code>      | Slično kao kod <code>String.includes()</code> metode, ova metoda provjerava sadrži li polje traženu vrijednost. Vraća <code>boolean</code> vrijednost ovisno o sadržavanju. Opcionalni <code>fromIndex</code> parametar koji definira od kojeg indeksa se pretražuje.   | <code>Array.includes(searchElement, fromIndex)</code> | <pre>const array1 = [1, 2, 3]; array1.includes(2) == true; const pets = ['cat', 'dog', 'bat']; pets.includes('cat', 1) == false</pre> | <code>true;</code><br><code>false</code>     |
| <code>find()</code>          | Vraća <b>vrijednost</b> prvog elementa u polju koji zadovoljava danu <code>callback</code> funkciju. Opcionalno, prima <code>thisArg</code> koji predstavlja lokalnu <code>this</code> vrijednost varijable u <code>callback</code> funkciji. Vraća <code>undefined</code> ako nema nijednog podudaranja.                           | <code>Array.find(callbackFn, thisArg)</code>          | <pre>const numbers = [4, 9, 16, 25, 29]; ; let first = numbers.find(function(value) {return value &gt; 18;});</pre>                   | <code>first ==</code><br><code>25</code>     |
| <code>findIndex()</code>     | Vraća <b>indeks</b> prvog elementa u polju koji zadovoljava danu <code>callback</code> funkciju. Opcionalno, prima <code>thisArg</code> koji predstavlja lokalnu <code>this</code> vrijednost varijable u <code>callback</code> funkciji. Vraća <code>-1</code> ako nema nijednog podudaranja.                                      | <code>Array.findIndex(callbackFn, thisArg)</code>     | <pre>const numbers = [4, 9, 16, 25, 29]; ; let firstIndex = numbers.find(function(value) {return value &gt; 18;});</pre>              | <code>firstIndex</code><br><code>== 3</code> |
| <code>findLast()</code>      | Vraća <b>vrijednost</b> prvog elementa u polju <b>iteriranjem unazad</b> koji zadovoljava danu <code>callback</code> funkciju. Opcionalno, prima <code>thisArg</code> koji predstavlja lokalnu <code>this</code> vrijednost varijable u <code>callback</code> funkciji. Vraća <code>undefined</code> ako nema nijednog podudaranja. | <code>Array.findLast(callbackFn, thisArg)</code>      | <pre>const array1 = [5, 12, 50, 130, 44];; array1.findLast(function(value) {return value &gt; 45;})</pre>                             | <code>130</code>                             |
| <code>findLastIndex()</code> | Vraća <b>indeks</b> prvog elementa u polju <b>iteriranjem unazad</b> koji zadovoljava danu <code>callback</code> funkciju. Opcionalno, prima <code>thisArg</code> koji predstavlja lokalnu <code>this</code> vrijednost varijable u <code>callback</code> funkciji. Vraća <code>-1</code> ako nema nijednog                         | <code>Array.findLastIndex(callbackFn, thisArg)</code> | <pre>const array1 = [5, 12, 50, 130, 44];; array1.findLastIndex(function(value) {return value &gt; 45;})</pre>                        | <code>3</code>                               |

Kada koristiti koju metodu pretraživanja?

- ako želimo pronaći **indeks prvog** pronađenog elementa, koristimo `indexOf()`
- ako želimo pronaći **indeks zadnjeg** pronađenog elementa, koristimo `lastIndexOf()`
- ako želimo pronaći indeks **prvog** elementa koji zadovoljava uvjet definiran u `callback` funkciji, koristimo `findIndex()`
- ako želimo pronaći indeks **zadnjeg** elementa koji zadovoljava uvjet definiran u `callback` funkciji, koristimo `findLastIndex()`
- ako želimo pronaći **vrijednost prvog** elementa koji zadovoljava uvjet definiran u `callback` funkciji, koristimo `find()`
- ako želimo pronaći **vrijednost zadnjeg** elementa koji zadovoljava uvjet definiran u `callback` funkciji, koristimo `findLast()`
- ako želimo provjeriti sadrži li polje traženu vrijednost, koristimo `includes()`

Postoji još metoda pretraživanja polja, poput `filter()`, `some()`, `every()`, `map()`, `reduce()` itd. O njima ćemo više na vježbama iz naprednih funkcija.

## Primjer 5 - Funkcija za brisanje korisnika iz polja

EduCoder šifra: `DELETEme`

Recimo da imamo polje `korisnici` koje sadrži nekoliko objekata `Korisnik`. Želimo implementirati funkciju `deleteUser` koja će primiti polje korisnika i korisničko ime, pronaći korisnika s tim korisničkim imenom i obrisati ga iz polja.

Upotrijebit ćemo konstruktor `Korisnik` i dodat ćemo još atribut `korisnicko_ime`.

```
function Korisnik(ime, prezime, godina_rodenja, korisnicko_ime) {
  this.ime = ime;
  this.prezime = prezime;
  this.godina_rodenja = godina_rodenja;
  this.korisnicko_ime = korisnicko_ime;
  this.predstaviSe = function () {
    console.log(
      `Bok! Ja sam ${this.ime} ${this.prezime}. Rođen/a sam ${this.godina_rodenja} godine.`
    );
  };
}
```

Izrađujemo nekoliko korisnika i dodajemo ih u polje `korisnici`.

```
let korisnik1 = new Korisnik("Ivan", "Ivić", 1995, "iivic");
let korisnik2 = new Korisnik("Marko", "Markić", 1990, "mmarkic90");
let korisnik3 = new Korisnik("Ana", "Anić", 1985, "aanic");
let korisnik4 = new Korisnik("Petra", "Petrović", 1970, "ppetrovic70");

let korisnici = [korisnik1, korisnik2, korisnik3, korisnik4];
```

Sada možemo implementirati funkciju `deleteUser` koja će primiti polje korisnika i korisničko ime, pronaći korisnika s tim korisničkim imenom i obrisati ga iz polja.

```
function deleteUser(korisnici, korisnicko_ime) {
  const delIndex = korisnici.findIndex(function (korisnik) { // Naša callback funkcija
    vraća indeks prvog korisnika koji ima korisničko ime koje tražimo
    return korisnik.korisnicko_ime === korisnicko_ime;
  });
  if (delIndex !== -1) { // Ako je korisnik pronađen, obriši ga iz polja
    korisnici.splice(delIndex, 1); //Brišemo jedan element na indeksu delIndex
  }
  return korisnici; //Vraćamo novo polje korisnika
}
```

Kao povratnu vrijednost funkcije vraćamo novo polje korisnika. Izbrisat ćemo korisnika s korisničkim imenom `mmarkic90`.

✓Rezultat:

```
console.log(deleteUser(korisnici, "mmarkic90")); //Output: [Korisnik, Korisnik, Korisnik]
// 0: Korisnik {ime: "Ivan", prezime: "Ivić",
godina_rođenja: 1995, korisnicko_ime: "iivic", predstaviSe: f}
// 1: Korisnik {ime: "Ana", prezime: "Anić",
godina_rođenja: 1985, korisnicko_ime: "aanic", predstaviSe: f}
// 2: Korisnik {ime: "Petra", prezime:
"Petrović", godina_rođenja: 1970, korisnicko_ime: "ppetrovic70", predstaviSe: f}
```

## Primjer 6 - Implementacija `removeDuplicates` funkcije

EduCoder šifra: `duplicates,duplicates`

Recimo da imamo polje `brojevi` koje sadrži nekoliko brojeva. Želimo implementirati funkciju `removeDuplicates` koja će primiti polje brojeva (ili stringove) i obrisati sve duplikate iz polja. Funkcija mora vratiti novo polje bez duplikata.

```
let brojevi = [1, 2, 3, 4, 5, 1, 2, 6, 7, 6];

let brojeviBezDuplikata = removeDuplicates(brojevi); // Output: [1, 2, 3, 4, 5, 6, 7] - ono
što želimo
```

Ovakvu funkciju možemo implementirati koristeći `filter` gotovu filter metodu, vrlo jednostavnu. Kako mi `filter` metodu još nismo odradili. Iskoristit ćemo znanje koje do sada imamo. Pokazat ćemo 2 načina implementacije ove funkcije.

1. način počiva na ideji da su ključevi objekta jedinstveni, pa ćemo iskoristiti objekt kao pomoćnu strukturu za brisanje duplikata.

```
function removeDuplicates(polje) {
  let element, rezultatPolje = [], pomocniObjekt = {}; // Varijable koje ćemo koristiti

  for (element = 0; element < polje.length; element++) { //Iteriramo kroz polje
    pomocniObjekt[polje[element]] = 0; //Dodajemo parove ključ-vrijednost. Vrijednost nam
    nije bitna, a ključevi će biti elementi polja
  }
  for (element in pomocniObjekt) { //Iteriramo kroz ključeve objekta
    rezultatPolje.push(element); //Dodajemo ključeve u novo polje
  }
  return rezultatPolje; //Vraćamo novo polje
}
```

Testirajmo funkciju:

```
let brojevi = [1, 2, 3, 4, 5, 1, 2, 6, 7, 6];
let brojeviBezDuplikata = removeDuplicates(brojevi);
console.log(brojeviBezDuplikata); // Output: [1, 2, 3, 4, 5, 6, 7]

let stringovi = ["jabuka", "kruška", "jabuka", "banana", "kruška", "jabuka"];
let stringoviBezDuplikata = removeDuplicates(stringovi);
console.log(stringoviBezDuplikata); // Output: ["jabuka", "kruška", "banana"]
```

- način bazira se na metodi `indexOf` za provjeru postojanja elementa u polju. Metoda vraća `-1` ako element nije pronađen, a indeks elementa ako je pronađen.

```
function removeDuplicates2(polje) {
  let rezultatPolje = [];
  for (let i = 0; i < polje.length; i++) {
    if (rezultatPolje.indexOf(polje[i]) === -1) { // Čitaj: Ako element nije pronađen u
    rezultatPolje
      rezultatPolje.push(polje[i]); // Ako element nije pronađen u rezultatPolje, dodajemo ga
    }
  }
  return rezultatPolje; //Vraćamo novo polje
}
```

Testirajmo funkciju:

```
let brojevi = [1, 2, 3, 4, 5, 1, 2, 6, 7, 6];
let brojeviBezDuplikata = removeDuplicates2(brojevi);
console.log(brojeviBezDuplikata); // Output: [1, 2, 3, 4, 5, 6, 7]

let stringovi = ["jabuka", "kruška", "jabuka", "banana", "kruška", "jabuka"];
let stringoviBezDuplikata = removeDuplicates2(stringovi);
console.log(stringoviBezDuplikata); // Output: ["jabuka", "kruška", "banana"]
```

## Samostalni zadatak za vježbu 5

**Napomena:** Ne predaje se i ne boduje se. Zadatak možete i ne morate rješavati u [EduCoder](#) aplikaciji.

**EduCoder šifra:** `sportski_duh`

1. Napišite JavaScript program koji će stvoriti polje `osobe` koje će sadržavati objekte `Osoba`. Objekt `Osoba` mora se sastojati od svojstava: `ime`, `prezime`, `godina_rodenja`, `spol` i `visina`. Prvo definirajte konstruktor `Osoba` i dodajte svojstva. Napravite nekoliko osoba i dodajte ih u polje `osobe`. Dodajte novu metodu `predstaviSe()` u konstruktor koja će ispisati sve podatke o osobi u konzolu. Iterirajte kroz polje `osobe` i ispišite sve osobe u konzolu, kao i sve podatke o svakoj osobi. Dodajte globalnu funkciju `prosjecnaVisina(osobe)` koja će kao argument primiti polje `osobe`, izračunati prosječnu visinu svih osoba i ispisati je u konzolu.
2. Napravite novi objekt `sportasi` koji će se sastojati od svojstava: `ime`, `prezime`, `godina_rodenja`, `spol`, `visina`, `tezina`, `sport`, `klub` i `broj_dresa`. Napravite nekoliko sportaša i dodajte ih u polje `sportasi`.
  - implementirajte globalnu funkciju `prosjecnaTezina(sportasi)` koja će kao argument primiti polje `sportasi`, izračunati prosječnu težinu svih sportaša i ispisati je u konzolu.
  - implementirajte globalnu funkciju `najteziSportas(sportasi)` koja će pronaći i vratiti objekt najtežeg sportaša.
  - deklarirajte novo polje `sportasi_senior` u koje ćete pohraniti sve sportaše starije od 30 godina. Koristite neke od metoda iz poglavlja 3.4.2 - Metode pretraživanja polja.
  - dodajte novo svojstvo u konstruktor `sportas`. Neka to bude polje `nastupi`. Dodajte i metodu `dodajNastup()` koja će dodati novi nastup sportašu, pojedini nastup neka bude običan string, npr. "2022 Zagreb Open". Dodajte nekoliko nastupa svakom sportašu.
  - dodajte metodu `nastupiSportasa()` koja će ispisati sve nastupe sportaša u konzolu u sljedećem formatu: "Nastup 1: `{nastup1}`, Nastup 2: `{nastup2}`, ... Nastup N: `{nastupN}`".
  - dodajte metodu `dohvatiZadnjaDvaNastupa()` koja će ispisati zadnja dva nastupa sportaša u konzolu. Koristite metodu `slice()`.
  - implementirajte globalnu funkciju `izbrisiSvimaPrviNastup(sportasi)` koja će obrisati svim sportašima prvi nastup. Koristite metodu `shift()`.
  - implementirajte globalnu funkciju `azurirajBrojDresa(sportasi, brojDresa, noviBrojDresa)` koja će ažurirati broj dresa sportaša u polju `sportasi`. Funkcija mora pronaći sportaša u polju po broju dresa i ažurirati mu broj dresa u novi broj dresa. Ako su brojevi dresa jednaki, funkcija mora obavijestiti korisnika. Ako sportaš nije pronađen, funkcija mora obavijestiti korisnika.
3. Koristeći danu funkciju `gcd_two_numbers(x, y)` koja vraća najveći zajednički djelitelj dva broja, implementirajte funkciju `gcd_array(arr)` koja će primiti polje brojeva i vratiti najveći zajednički djelitelj svih brojeva u polju. Morate koristiti funkciju `gcd_two_numbers(x, y)` unutar funkcije `gcd_array(arr)`.

```
function gcd_two_numbers(x, y) {  
  if ((typeof x !== 'number') || (typeof y !== 'number'))  
    return false;  
  x = Math.abs(x);  
  y = Math.abs(y);  
  while(y) {  
    var t = y;  
    y = x % y;  
    x = t;  
  }  
  return x;  
}
```

```
function gcd_array(array) {  
  // Vaš kód ovdje  
}  
console.log(gcd_array([3, 6, 9, 12])); // Output: 3  
console.log(gcd_array([10, 20, 30, 40])); // Output: 10
```