

Programiranje u skriptnim jezicima (PJS)

Nositelj: doc. dr. sc. Nikola Tanković

Asistenti:

- Luka Blašković, mag. inf.
- Alesandro Žužić, mag. inf.

Ustanova: Sveučilište Jurja Dabroli u Puli, Fakultet informatike u Puli



[1] JavaScript osnove



JavaScript je visoko svestran programski jezik široko korišten u web razvoju. Prvobitno osmišljen za unapređenje interaktivnosti web stranica, danas se koristi i za razvoj serverskih aplikacija, desktop softvera, mobilnih aplikacija i drugih naprednih softverskih rješenja.

Posljednje ažurirano: 2.8.2024.

Sadržaj

- [Programiranje u skriptnim jezicima \(PJS\)](#)
- [1. JavaScript osnove](#)
 - [Sadržaj](#)
 - [1.1 Uvod](#)
 - [1.2 Gdje pisati JavaScript kôd?](#)
 - [1.3 Gdje je taj "Hello World"?](#)
- [2. Izrazi, tvrdnje, varijable, tipovi podataka i operatori](#)
 - [2.1 Tipovi podataka](#)
 - [2.2 Operatori](#)
 - [2.2.1 Izrazi \(eng. *expressions*\) vs tvrdnje \(eng. *statements*\)](#)
 - [2.2.2 Tablica osnovnih JavaScript operatora](#)
 - [2.2.3 Dodatni primjeri korištenja operatora](#)
 - [2.2.3.1 Aritmetički i Pridruživanja](#)
 - [2.2.3.2 Usprendni i Logički](#)
 - [2.2.4 Typeof operator](#)

- [Vježba 1](#)
- [Vježba 2](#)
- [2.3 Koncept varijable u JavaScriptu](#)
 - [2.3.1 JavaScript Strings](#)
- [2.4 Eksponencijalna \(znanstvena\) notacija](#)
- [2.5 BigInt](#)
- [Vježba 3](#)
- [Vježba 4](#)
- [Samostalni zadatak za vježbu 1](#)

1.1 Uvod

1. **Web stranica:** Zamislimo da je web stranica ljudsko tijelo.
 - **HTML** (Hypertext Markup Language) je kostur koji daje strukturu i podršku tijelu.
 - **CSS** (Cascading Style Sheets) je koža koja daje izgled tijelu.
 - **JavaScript** je skupina mišića i tetiva koja omogućuje kretanje tijela.
2. **Interaktivnost:** S JavaScriptom možemo izrađivati interaktivne komponente web stranice, poput:
 - formi koje reagiraju kada ih ispunjavamo,
 - izbornika koji se "spušta" kada kliknemo na njega ili
 - animacije koja se pokreće kad joj se približimo mišem.
3. **Running everywhere!:** Danas se JavaScript izvodi u raznim okruženjima, ne samo u web pregledniku!

Može se izvoditi na:

 - serveru tj. poslužitelju
 - desktop aplikacijama
 - mobilnim uređajima
4. **Easy to learn, Hard to Master:** JavaScript je jedan od jednostavnijih jezika za naučiti. Ima jednostavnu sintaksu i rezultate izvođenja kôda možemo vidjeti gotovo odmah u web pregledniku.
5. **Bogat community:** JavaScript je jedan od najpopularnijih programskih jezika na svijetu. Ima veliku zajednicu developera, odlično je dokumentiran, ima puno biblioteka i razvojnih okruženja koji nam olakšavaju izradu web stranica/aplikacija.

1.2 Gdje pisati JavaScript kôd?

Pisanje JavaScripta na u web pregledniku (strana klijenta - eng. *client side*) možemo podijeliti na 3 načina:

1. **Inline JavaScript** - kôd se piše direktno unutar HTML elementa, npr. u atributu `onclick`:

```
<button onclick="console.log('Hello World!')>Hello World</button>
```

2. **Internal JavaScript** - kôd se piše unutar HTML dokumenta, ali u odvojenom `<script>` elementu:

```
<script>
  console.log("Hello World!");
</script>
```

3. **External JavaScript** - kôd se piše u odvojenom JavaScript dokumentu, npr. `script.js`:

```
<!--index.html-->
<!DOCTYPE html>
<html>
  <head>
    <title>Moja web stranica</title>
    <script src="script.js"></script>
  </head>
  <body>
    <h1>Dobrodošli na Moju web stranicu</h1>

    <button onclick="showMessage()">Klikni me!</button>
  </body>
</html>
```

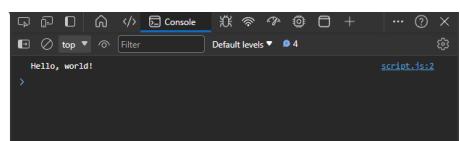
- Prednost ovog načina je što možemo koristiti isti kôd na više stranica, a i sam HTML dokument je čišći i pregledniji.
- Na isti način kao u kôdu iznad, `script.js` datoteku možemo uključiti i u druge HTML datoteke.

```
// script.js
function showMessage() {
  console.log("Hello World!");
}
```

1.3 Gdje je taj "Hello World"?

Kada otvorimo HTML dokument u web pregledniku, možemo otvoriti konzolu (F12) i vidjeti poruku "Hello World!", tako jednostavno!

Welcome to My Web Page

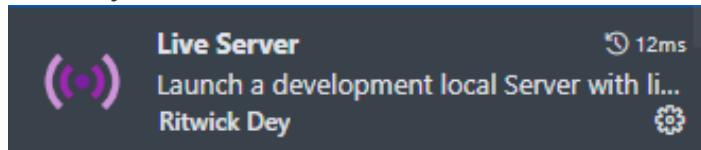


Idemo sada izmijeniti tekst koji nam ispisuje funkcija `showMessage()`. U `script.js` datoteci promijenimo tekst u `Hello JavaScript!`:

```
// script.js
function showMessage() {
  console.log("Hello JavaScript!");
}
```

Možemo primijetiti da se ponovnim klikom na gumb, tekst u konzoli nije promijenio. To je zato što je kôd iz `script.js` datoteke izvršen samo jednom, prilikom učitavanja stranice. Da bi promjena bila prikazana, moramo osvježiti stranicu (F5).

Naporno je svaki put osvježavati stranicu da bi vidjeli naše promjene. Iz tog razloga ćemo preuzeti [Live Server](#) ekstenziju za Visual Studio Code. Ona će nam omogućiti da otvorimo HTML dokument u web pregledniku i da se svaka promjena u kôdu automatski osvježi u web pregledniku. Nakon što instaliramo ekstenziju, kliknemo desnim klikom na HTML dokument i odaberemo `Open with Live Server`.



2. Izrazi, tvrdnje, varijable, tipovi podataka i operatori

Varijable su mesta u memoriji u koje spremamo podatke. Svaka varijabla ima svoje ime i vrijednost. Vrijednost varijable može se mijenjati tijekom izvođenja programa.

Varijable možemo deklarirati na 3 načina: `var`, `let` i `const`. Varijable deklarirane s ključnim riječima `var` i `let` su varijable koje se mogu mijenjati, dok je `const` konstanta koja se ne može mijenjati. U pravilu koristimo `const` za deklariranje varijabli, osim ako znamo da će se vrijednost varijable mijenjati, tada koristimo `let`. `var` izbjegavamo, budući da ga je `let` zamjenio u ES6 standardu JavaScripta. Koga zanima više zašto je uveden `let`, može pročitati [ovde](#).

```
let x = 5;
console.log(x); // 5

x = 10;
console.log(x); // 10

const y = 15;
console.log(y); // 15

y = 20; // TypeError: Assignment to constant variable.
console.log(y);
```

2.1 Tipovi podataka

JavaScript je slabo tipizirani jezik (eng. **weakly typed**), što znači da razlikuje različite tipove varijable, no ne moramo ih strogo navoditi prilikom deklaracije varijable. Tip podatka varijable određuje se automatski prilikom dodjele vrijednosti varijabli.

Za provjeru tipa podatka varijable koristimo `typeof` operator.

```

let a = 5; // number
let b = "5"; // string
let c = true; // boolean

console.log(typeof a); // number
console.log(typeof b); // string
console.log(typeof c); // boolean

```

Varijable definirane s `const`:

- ne mogu se ponovno deklarirati (eng. ***redeclare***)
- ne mogu se ponovno dodijeliti (eng. ***reassign***)
- moraju se inicijalizirati prilikom deklaracije (eng. ***initialize***)
- imaju blokovski opseg (eng. ***block scope***)

Konstante se ne mogu ponovno deklarirati

```

const PI = 3.141592653589793;
PI = 3.14; // Baca grešku!
PI = PI + 10; // Baca grešku!

```

Konstante se moraju inicijalizirati prilikom deklaracije

```

const PI = 3.141592653589793; // Točno!

const PI; // Netočno!

```

2.2 Operatori

2.2.1 Izrazi (eng. *expressions*) vs tvrdnje (eng. *statements*)

U JavaScriptu, **izraz** (eng. ***expression***) je bilo koji valjani kôd koji se evaluira/razlaže (eng. ***resolve***) u vrijednost.

Primjer izraza može biti bilo koja matematička operacija, npr. za `x = 3`, `5 + 5`, ili `x = 7`, ili `x = x + 5`.

Navedeni izrazi se evaluiraju u vrijednosti: `3`, `10`, `10` i `12`.

Izrazi ne moraju biti samo brojevi! Evo još primjera izraza da bude jasnije:

- aritmetički izrazi: `5 + 3` ili `4 * 2`
- izrazi znakovnog niza: `"Hello " + "World"`
- logički izrazi: `true && false`
- funkcjski izrazi: `function() { console.log("Hello World!"); }`

Najjednostavnije rečeno, računalni program je popis "instrukcija" koje računalo treba "izvršiti". U programiranju, te "instrukcije" nazivaju se **tvrdnje** (eng. ***statements***). JavaScript program je popis tvrdnji koje se izvršavaju redom. Tvrđnje mogu biti: deklaracije varijabli, izrazi, kontrolne strukture, petlje, pozivi funkcija, ključne riječi, komentari itd.

2.2.2 Tablica osnovnih JavaScript operatora

Operatori su simboli koji se koriste za izvođenje operacija nad podacima, preciznije: sve kompleksnije izraze spajamo pomoću operatora, poput `=` i `+`. Postoji više vrsta operatora, mi ćemo se baviti samo nekima od njih:

- **Aritmetički operatori** - primarno se koriste za izvođenje aritmetičkih operacija nad brojevima
- **Operatori pridruživanja** - koriste se za pridruživanje vrijednosti varijablama
- **Operatori usporedbe** - koriste se za usporedbu vrijednosti
- **Logički operatori** - koriste se za izvođenje logičkih operacija
- **Operatori tipa (eng. type)** - koriste se za provjeru tipa podatka

Operator	Vrsta	Broj operanada	Opis	Primjer
Osnovni aritmetički <code>+, -, *, /</code>	Aritmetički	binarni (2)	Standardni aritmetički operatori.	<code>2 + 3</code> vraća <code>5</code> , <code>5 * 6</code> vraća <code>30</code>
Unarni +	Aritmetički	unarni (1)	Pokušava pretvoriti operand u broj, ako već nije.	<code>+ "3"</code> vraća <code>3</code> , <code>+true</code> vraća <code>1</code>
Unarni -	Aritmetički	unarni (1)	Vraća negaciju operanda.	ako je <code>x=3</code> , <code>-x</code> vraća <code>-3</code>
Inkrement <code>++</code>	Aritmetički	unarni (1)	Povećava svoj operand za 1, vraćajući novu vrijednost ako se koristi kao prefix (<code>++x</code>), ili izvornu vrijednost ako se koristi kao postfix (<code>x++</code>).	ako je <code>x = 3</code> , onda <code>++x</code> postavlja <code>x</code> na <code>4</code> i vraća <code>4</code> . Ali, <code>x++</code> vraća <code>3</code> i nakon toga postavlja <code>x</code> na <code>4</code> .
Dekrement <code>--</code>	Aritmetički	unarni (1)	Umanjuje svoj operand za 1, vraćajući novu vrijednost ako se koristi kao prefix (<code>--x</code>), ili izvornu vrijednost ako se koristi kao postfix (<code>x--</code>).	ako je <code>x = 3</code> , onda <code>--x</code> postavlja <code>x</code> na <code>2</code> i vraća <code>2</code> . Ali, <code>x--</code> vraća <code>3</code> i nakon toga postavlja <code>x</code> na <code>2</code> .
Ostatak <code>%</code>	Aritmetički	binarni (2)	Vraća cjelobrojni ostatak dijeljenja dva operanda.	ako je <code>x=3</code> , <code>-x</code> vraća <code>-3</code>
Eksponiranje <code>**</code>	Aritmetički	binarni (2)	Računa eksponent kao <code>baza^eksponent</code> .	<code>2 ** 3</code> vraća <code>8</code> , <code>10 ** -1</code> vraća <code>0.1</code>
Pridruživanje <code>=</code>	Pridruživanja	binarni (2)	Pridružuje vrijednost varijabli ili svojstvu.	<code>x = 2, y = f(x)</code>
Zbroji i pridruži <code>+=</code>	Pridruživanja	binarni (2)	Zbroji vrijednosti 2 operanda i rezultat pridruži lijevom operandu.	<code>a = 2, a+=3</code> vraća <code>5</code>
Oduzmi i pridruži <code>-=</code>	Pridruživanja	binarni (2)	Oduzmi vrijednosti 2 operanda i rezultat pridruži lijevom operandu.	<code>a = 2, a-=3</code> vraća <code>-1</code>
Pomnoži i pridruži <code>*=</code>	Pridruživanja	binarni (2)	Pomnoži vrijednosti 2 operanda i rezultat pridruži lijevom operandu.	<code>a = 2, a*=3</code> vraća <code>6</code>
Podijeli i pridruži <code>/=</code>	Pridruživanja	binarni (2)	Podijeli vrijednosti 2 operanda i rezultat pridruži lijevom operandu.	<code>a = 2, a/=2</code> vraća <code>1.5</code>
Ostatak i pridruži <code>%=</code>	Pridruživanja	binarni (2)	Izračunaj cjelobrojni ostatak vrijednosti 2 operanda i rezultat pridruži lijevom operandu.	<code>a = 3, a%=2</code> vraća <code>1</code>
Jednako <code>==</code>	Usporedni	binarni (2)	Vrati <code>true</code> ako su operandi jednaki.	<code>1 == 1</code> vraća <code>true</code> , <code>'hello' == 'hello'</code> vraća <code>true</code> , <code>5 == '5'</code> također vraća <code>true</code>
Nejednako <code>!=</code>	Usporedni	binarni (2)	Vrati <code>true</code> ako operandi nisu jednaki.	<code>1 != 1</code> vraća <code>false</code> , <code>'hello' != 'world'</code> vraća <code>true</code>
Identično <code>===</code>	Usporedni	binarni (2)	Vrati <code>true</code> ako operandi su operandi jednaki i istog tipa podatka.	<code>1 === 1</code> vraća <code>true</code> , <code>'hello' === 'hello'</code> vraća <code>true</code> , <code>'1' === 1</code> vraća <code>false</code>

				false, 0 === false vraća false
Identično nejednako !==	Usporedni	binarni (2)	Vrati true ako su operandi jednaki ali različitog tipa, ili ako su različiti i istog tipa podatka.	1 !== 1 vraća false, 'hello' !== 'hello' vraća false, '1' !== 1 vraća true, 0 !== false vraća true
Veće od >, manje od <	Usporedni	binarni (2)	(>) Vrati true ako je lijevi operand veći od desnog operanda. (<) Vrati true ako je lijevi operand manji od desnog operanda.	5 > 2 vraća true, 'ab' > 'aa' vraća false, 5 < 3 vraća false
Veće ili jednako od >=, manje ili jednako od <=	Usporedni	binarni (2)	(>=) Vrati true ako je lijevi operand veći ili jednak desnom operandu. (<=) Vrati true ako je lijevi operand manji ili jednak desnom operandu.	5 >= 3 vraća true, 'ab' >= 'aa' vraća true, 3 <= 3 vraća true
Logički AND &&	Logički	binarni (2)	Za skup boolean operanada rezultat će biti true samo i samo ako su oba operanda true. Ako generaliziramo, vraća vrijednost prvog falsy operanda kod evaluacije s lijeva na desno, ili vrijednost zadnjeg operanda ako su svi true.	za a = 3 i b = -2, izraz (a > 0 && b > 0) vraća false, za izraz 5 && 6 vraća 6, ali 4 && false vraća false
Logički OR 	Logički	binarni (2)	Za skup boolean operanada rezultat će biti true ako je jedan ili više operanada true. Ako generaliziramo, vraća vrijednost prvog truthy operanda kod evaluacije s lijeva na desno, ili vrijednost zadnjeg operanda ako su svi false.	za a = 3 i b = -2, izraz (a > 0 b > 0) vraća true, true 0 vraća true, ali false 0 vraća 0
Logički NOT !	Logički	unarni (1)	Mjenja true izraz u false i obrnuto. Tipično se koristi sa boolean operandima, ali kada ne, vraća false kada se dodaje na tzv. truthy izraze, u suprotnom vraća true.	za a = 3 i b = -2, izraz (! (a > 0 b > 0)) vraća false. !"" vraća true, ali !"Hello World" vraća false
Operator tipa typeof	Type	unarni (1)	Vraća niz znakova koji označava vrstu operadora.	typeof(2) vraća "number", typeof("Banana") vraća "string", typeof(someFunction) vraća "function"

2.2.3 Dodatni primjeri korištenja operatora

2.2.3.1 Aritmetički i Pridruživanja

```

const a = 5; // Operator pridruživanja
const b = 10;
console.log(a + b); // 15

// Vrijede ista pravila o prioritetu izvođenja operacija kao i u matematici

console.log(a + b * 2); // 25
console.log((a + b) * 2); // 30

let a = 20;
let b = 2;
console.log(a / b); // 10 - količnik
console.log(a % b); // 0 - ostatak pri dijeljenju

let a = 5;
let b = 10;

```

```

console.log(a / b); // 0.5 - količnik
console.log(a % b); // 5 - ostatak pri dijeljenju

let c = 5;
c += 10; // Isto kao da smo napisali c = c + 10
console.log(c); // 15

let d = 5;
d -= 10; // Isto kao da smo napisali d = d - 10
console.log(d); // -5

let e = 5;
e *= 10; // Isto kao da smo napisali e = e * 10
console.log(e); // 50

let f = 5;
f /= 10; // Isto kao da smo napisali f = f / 10
console.log(f); // 0.5

let g = 5;
g %= 10; // Isto kao da smo napisali g = g % 10
console.log(g); // 5

let h = 10;
let h_kvadrirano = h ** 2;
console.log(h_kvadrirano); // 100

let brojac = 0;
brojac++; // Isto kao da smo napisali brojac = brojac + 1
console.log(brojac); // 1

brojac = 10;
brojac--; // Isto kao da smo napisali brojac = brojac - 1
console.log(brojac); // 9

let i = 5;
let j = i++; // j = 5, i = 6 - prvo se dodjeljuje vrijednost i, a zatim se povećava za 1

let k = 5;
let l = ++k; // l = 6, k = 6 - prvo se povećava za 1, a zatim se dodjeljuje vrijednost k

console.log(j, i, l, k); // 5 6 6 6

```

2.2.3.2 Usporedni i Logički

```

// Usporedni operatori
let a = 5;
let b = 10;

console.log(a == b); // false
console.log(a != b); // true

```

```

let c = 5;
let d = "5";
console.log(c == d); // true
console.log(c === d); // false - različiti tipovi podataka (number i string)

let e = 5;
let f = 10;
console.log(e > f); // false
console.log(e < f); // true
console.log(e >= f); // false
console.log(e <= f); // true

// Logički operatori
let g = true;
let h = false;
console.log(g && h); // false
console.log(g || h); // true

```

Što ako se ne koriste uz boolean operative?

JavaScript će pokušati pretvoriti operative u boolean vrijednosti (npr. 0 u false, 1 u true, prazan string u false, string sa sadržajem u true itd.

Googlaj: javascript type coercion

```

// Logički AND
console.log(5 && 6); // 6 (Pogledati u tablici - '&&' evaluira s lijeva na desno i vraća zadnji koji je 'true')
console.log(0 && 7); // 0 (Pogledati u tablici - '&&' evaluira s lijeva na desno i vraća prvi koji je 'false')
console.log(false && 0); // false

// Logički OR

console.log(5 || 6); // 5 (Pogledati u tablici - '||' evaluira s lijeva na desno i vraća prvi koji je 'true')
console.log(0 || 7); // 7 (Pogledati u tablici - '||' evaluira s lijeva na desno i vraća prvi koji je 'true')
console.log(false || 0); // 0 (Pogledati u tablici - '||' evaluira s lijeva na desno i vraća zadnji koji je 'false')

// Logički NOT

console.log(!true); // false
console.log(!false); // true
console.log(!"Hello World"); // false
console.log(! ""); // true
console.log(!0); // true
console.log(!5); // false

```

Naglasili smo da je izraz (eng. **expression**) u JavaScriptu bilo koji valjani kod koji se evaluira u vrijednost.

Primjer 1:

- `5 + 5` je izraz koji se evaluira u `10`,
- kao i izraz `5 < 10` koji se evaluira u `true`,
- ili `9 < 9` koji se evaluira u `false`.

Logički operatori `&&`, `||` i `!` su također izrazi, koji se evaluiraju u `true` ili `false`, kako smo već prikazali u tablici operatora.

Primjer 2:

- Izraz `true && true` se evaluira u `true`,
- Izraz `true && false` se evaluira u `false`.
- Izraz `true || false` se evaluira u `true`,
- Izraz `false || false` se evaluira u `false`.

Jednako tako se izrazi iz primjera 1 mogu koristiti kao operandi u izrazima iz primjera 2. Vrlo je važno pritom pametno imenovati varijable, kako bi se izrazi mogli čitati kao rečenice.

Primjer 3: Želimo definirati logički izraz i nekoliko varijabli kako bi zaključili jesmo li pročitali broj stranica knjige koji smo si zadali kao cilj za ovaj tjedan.

```
let brojStranicaProcitano = 100;
let ciljaniBrojStranica = 200;

let ciljPostignut = brojStranicaProcitano >= ciljaniBrojStranica; // false
```

Primjer 4: Želimo definirati logički izraz i nekoliko varijabli kako bi zaključili jesmo li obavili sve zadatke prije nego što možemo krenuti na putovanje.

```
let kupljeneAvionskeKarte = true;
let rezerviraniSmjestaj = true;

let spremniZaPutovanje = kupljeneAvionskeKarte && rezerviraniSmjestaj; // true
```

Recimo da postoji opcija i da idemo s vlakom.

```
let kupljeneKarteZaVlak = true;
let kupljeneAvionskeKarte = false;
let rezerviraniSmjestaj = true;

let spremniZaPutovanje =
  (kupljeneAvionskeKarte || kupljeneKarteZaVlak) && rezerviraniSmjestaj; // true - jer je
  bar jedan od uvjeta prijevoza ispunjen
```

Međutim i uvjet prijevoza možemo logično definirati kao varijablu!

```
let kupljeneKarteZaVlak = true;
let kupljeneAvionskeKarte = false;
let rezerviraniSmjestaj = true;
let uvjetPrijevoza = kupljeneAvionskeKarte || kupljeneKarteZaVlak; // true - jer je bar jedan od uvjeta prijevoza ispunjen
let spremniZaPutovanje = uvjetPrijevoza && rezerviraniSmjestaj; // true - sada oba moraju biti ispunjena!
```

Primjer 5. Želimo definirati nekoliko logičkih izraza i varijabli kako bi zaključili jesmo li zadovoljili sve uvjete za prolazak kolegija na fakultetu. Dani su sljedeći uvjeti:

- student mora imati više ili točno 50% bodova na završnom pismenom i više ili točno 50% bodova na završnom usmenom ispitu ili mora imati ukupno 50% bodova ostvarenih tijekom semestra
- student mora biti prisutan na više od 80% predavanja
- student mora predati projektni zadatak
- projektni zadatak mora biti ocijenjen s pozitivnom ocjenom

Kako možemo definirati prolaz preko ispita?

```
// Bodovi na pismenom i usmenom ispitu
let bodoviNaPismenom = 60;
let bodoviNaUsmenom = 40;

// Maksimalni broj bodova na pismenom i usmenom ispitu
let pismeniMaxBodova = 100;
let usmeniMaxBodova = 100;

// Prisustvo na predavanjima
let ukupniBrojPredavanja = 15;
let brojPrisustva = 14;

// Projektni zadatak
let predanProjektniZadatak = true;
let ocjenaProjektnogZadatka = 3;
```

Prvo ćemo definirati nekoliko logičkih i usporednih izraza, kako bi lakše ispisali konačan rezultat.

```

let prolazNaPismenom = bodoviNaPismenom / pismeniMaxBodova >= 0.5;
let prolazNaUsmenom = bodoviNaUsmenom / usmeniMaxBodova >= 0.5;

let prisutnostZadovoljavajuca = brojPrisustva / ukupniBrojPredavanja > 0.8;

let projektRijesen = predanProjektniZadatak && ocjenaProjektnogZadatka > 1;

let prolaz =
  prolazNaPismenom &&
  prolazNaUsmenom &&
  prisutnostZadovoljavajuca &&
  projektRijesen; // false

```

Dodat ćemo i alternativu polaganja putem kontinuiranog praćenja.

```

let kolokvij1 = 40;
let kolokvij2 = 60;
let kolokvijiMaxBodova = 200;
let prolazNaKolokvijima = (kolokvij1 + kolokvij2) / kolokvijiMaxBodova >= 0.5;

let prolaz =
  ((prolazNaPismenom && prolazNaUsmenom) || prolazNaKolokvijima) &&
  prisutnostZadovoljavajuca &&
  projektRijesen; // true

```

2.2.4 Typeof operator

Primitivni tipovi podataka u JavaScriptu predstavljaju vrijednosti koje se spremaju u memoriju bez dodatnih metoda i svojstava. Primitivni tipovi su:

- `string`
- `number`
- `boolean`
- `undefined`

`typeof` operator može vratiti jedan od tih primitivnih tipova.

```

// typeof
console.log(typeof 5); // number
console.log(typeof "5"); // string
console.log(typeof true); // boolean
console.log(typeof undefined); // undefined
console.log(typeof null); // object

```

Zašto je `typeof null` = objekt? U JavaScriptu, `null` doslovno predstavlja "ništa". Nažalost, `typeof` funkcija će vratiti da je tip podatka `null` objekt. Radi se o bugu koji je prisutan od samih početaka ovog jezika.

Kojeg će tipa biti sljedeća varijabla?

```
const secret_number;
```

► Spoiler Warning!

Odgovor je `undefined`. `undefined` je tip podatka koji se koristi kada varijabla nije inicijalizirana, dok je `null` je tip podatka koji se koristi kada varijabla nema vrijednost.

Vježba 1

Idemo napraviti kratku vježbu onoga što smo dosad prošli. U `script.js` datoteci deklarirajte varijable `a`, `b` i `c` i dodijelite im vrijednosti `5`, `"5"` i `true`. Ispišite vrijednosti varijabli u konzolu i provjerite njihove tipove. Kôd dodajte unutar funkcije `showMessage()`.

Nakon toga, `typeof` operatorom provjerite tipove varijabli i u konzolu ispišite tvrdnju za svaku varijablu, npr. "Varijabla `a` je tipa number". Izraze u `console.log()` možete spojiti pomoću `+` operatara.

Zašto `console.log(a == b)` vraća `true`? Objasnите.

Rezultat:

5	<code>script.js:6</code>
5	<code>script.js:7</code>
true	<code>script.js:8</code>
Varijabla <code>a</code> je tipa <code>number</code>	<code>script.js:10</code>
Varijabla <code>b</code> je tipa <code>string</code>	<code>script.js:11</code>
Varijabla <code>c</code> je tipa <code>boolean</code>	<code>script.js:12</code>

Vježba 2

Idemo sada napraviti jednostavan kalkulator. U `script.js` datoteci deklarirajte varijable `a` i `b` i dodijelite im vrijednosti `5` i `10`. Izračunajte zbroj, razliku, umnožak i količnik varijabli `a` i `b` i ispišite ih u konzolu. Dodatno, ispišite u konzolu ostatak pri dijeljenju varijabli `a` i `b` i rezultat eksponiranja varijable `a` na potenciju varijable `b`.

Rezultat:

Zbroj <code>a</code> i <code>b</code> je: 15	<code>script.js:5</code>
Razlika <code>a</code> i <code>b</code> je: -5	<code>script.js:6</code>
Umnožak <code>a</code> i <code>b</code> je: 50	<code>script.js:7</code>
Količnik <code>a</code> i <code>b</code> je: 0.5	<code>script.js:8</code>
Ostatak pri dijeljenju varijable <code>a</code> sa <code>b</code> je: 5	<code>script.js:9</code>
Rezultat eksponiranja varijable <code>a</code> sa <code>b</code> je: 9765625	<code>script.js:10</code>

2.3 Koncept varijable u JavaScriptu

Varijable u JavaScriptu mogu sadržavati bilo koju vrijednost, neovisno o tipu podatka. To znači da varijabla može sadržavati broj, string, boolean, objekt, funkciju, itd.

Ista varijabla može sadržavati i više različitih tipova podataka!

Važno je razumjeti što se dešava "ispod haube" kada deklariramo varijablu i dodijelimo joj vrijednost.

Bez tipova podataka, računalno neće znati interpretirati (na siguran način) sljedeće:

```
let x = 16 + "Volvo";
```

Ima li smisla? Hoće li ovo biti broj ili string? Ili ćemo dobiti grešku?

Kada JavaScript vidi da se koristi operator `+` na broju i stringu, on će automatski pretvoriti broj u string i spojiti ih. Ovo se zove **implicitna konverzija**.

```
let x = "16" + "Volvo";
```

Uzmimo za primjer sljedeći izraz?

```
let x = 16 + 4 + "Volvo";
```

Koji će biti rezultat? `"164Volvo"` ili `"20Volvo"`?

A ovdje?

```
let x = "Volvo" + 16 + 4;
```

► Spoiler Warning!

```
let x = 16 + 4 + "Volvo";
console.log(x); // 20Volvo
```

```
let x = "Volvo" + 16 + 4;
console.log(x); // Volvo164
```

Imajte na umu da prioritet i asocijativnost operatora utječu samo na redoslijed evaluacije **operatora**, ali ne i na redoslijed evaluacije **operanada**. Operandi se uvijek evaluiraju s lijeva na desno!, međutim njihovi rezultati se sastavljaju prema redoslijedu prioritera operatora!

JavaScript tipovi su dinamički, što znači da se tip podatka variable može promijeniti tijekom izvođenja programa.

```
let x;
console.log(typeof x); // undefined
x = 5;
console.log(typeof x); // number
x = "Petar";
console.log(typeof x); // string
```

2.3.1 JavaScript Strings

String je tekstualni podatak, radi se o nizu znakova. String možemo definirati s jednostrukim ili dvostrukim navodnicima.

```
let x = "Petar";
let y = "Petar";
```

Možemo koristiti i navodne znakove unutar stringa, ali moramo paziti da se ne podudaraju s vanjskim navodnicima.

```
let x = "Petar je rekao: 'Dobar dan!'";
```

Možemo koristiti i varijable unutar stringa, ali onda moramo koristiti *backtickse* `` te `{}$` za prikaz same varijable. Ovakva sintaksa se zove [template literals](#).

```
let ime = "Petar";
let predstavljanje = `Moje ime je ${ime}`;
console.log(predstavljanje); // Moje ime je Petar
```

Istu stvar možemo dobiti i sa `+` operatorom, ali [template literals](#) sintaksa je jednostavnija i puno čitljivija!

```
let ime = "Petar";
let predstavljanje1 = "Moje ime je " + ime;
let predstavljanje2 = `Moje ime je ${ime}`;

console.log(predstavljanje1 == predstavljanje2); // true
```

Još jedan primjer s brojevima!

```
const a = 5;
const b = 10;
console.log(`Petnaest je ${a + b} a ne ${2 * a + b}.`);
// Petnaest je 15 a ne 20.
```

2.4 Eksponencijalna (znanstvena) notacija

Eksponencijalna notacija se koristi za prikazivanje jako velikih ili jako malih brojeva. Zapisujemo ju koristeći `e` ili `E`.

```
let y = 123e5; // 12300000
let z = 123e-5; // 0.00123
```

Primjerice, broj 100 možemo zapisati kao:

```
100 = 10e1 //čitaj 10 puta 10 na prvu
```

Broj 1 možemo zapisati kao:

```
1 = 10e-1 //čitaj 10 puta 10 na minus prvu
```

Decimalni broj 200.5 možemo zapisati kao:

```
200.5 = 2.005e2 //čitaj 2.005 puta 10 na drugu
```

2.5 BigInt [DODATNO]

Random Fact, ali nije loše za zapamtiti:

Većina programskih jezika ima različite tipove podataka za:

1. Cijele brojeve

- byte (8-bit)
- short (16-bit)
- int (32-bit)
- long (64-bit)

2. Brojeve s decimalnim zarezom

- float (32-bit)
- double (64-bit)

Svi Javascript brojevi su uvijek istog tipa! A to je `double` (64-bit floating point).

JavaScript, sa ES2020 standardom, dobiva novi tip podatka `BigInt` koji može prikazati brojeve veće od `Number.MAX_SAFE_INTEGER`, odnosno ($2^{53} - 1$).

```
const x = Number.MAX_SAFE_INTEGER + 1;
const y = Number.MAX_SAFE_INTEGER + 2;

console.log(Number.MAX_SAFE_INTEGER);
// Očekivani output: 9007199254740991

console.log(x);
// Očekivani output: 9007199254740992

console.log(x === y);
// Očekivani output: false ?

// Međutim, rezultat je true. x i y su isti brojevi jer ne možemo premašiti
MAX_SAFE_INTEGER

// Koristimo BigInt
const z = BigInt(Number.MAX_SAFE_INTEGER) + BigInt(2);
console.log(z == y); // false
```

Vježba 3

Deklarirajte dvije varijable `ime` i `prezime` i dodijelite im vrijednosti `Marko` i `Marić`. Ispišite dvaput u konzolu rečenicu `Moje ime je Marko Marić.`, jednom koristeći `+` operator, a drugi put koristeći `template literals`.

Rezultat:

```
Moje ime je Marko Marić.          script.js:4  
Moje ime je Marko Marić.          script.js:5
```

Vježba 4

Želite si definirati nekoliko ciljeva za ovaj tjedan kako biste ispunili vaš `weekly_goal`. Vaši ciljevi definirani su sljedećim tvrdnjama, odnosno vaš `weekly_goal` je ispunjen ako:

- želim proučiti PJS1 skriptu iz JavaScripta
- želim pročitati barem 50 stranica omiljene knjige
- želim vježbatи JavaScript barem 2 sata ili riješiti barem 10 zadataka
- želim se svaki dan naspavati

Za svaku izjavu definirajte po nekoliko pomoćnih varijabli, npr. jednu za ciljanu vrijednost, jednu za ostvarenu vrijednost i jednu za rezultat ostvarenja (boolean). Na primjer, za izjavu `želim pročitati barem 50 stranica omiljene knjige` deklarirajte varijable `broj_procitanih_stranica` i `ciljani_broj_stranica` te varijablu `cilj_citanje`.

Rezultat:

Napišite u obliku: `weekly_goal = cilj1 && cilj2 && cilj3 && cilj4`

Samostalni zadatak za vježbu 1

Napomena: Ne predaje se i ne boduje se. Zadatak rješavate u [EduCoder](#) aplikaciji.

EduCoder šifra: `a_new_hope`

1. Deklarirajte tri konstante i jednu promjenjivu varijablu. Konstante neka budu vaše `ime` i `prezime` i `godina_rodenja`. Promjenjivu varijablu nazovite `trenutno_vrijeme`.
 - U varijable `ime` i `prezime` pohranite svoje ime i prezime, a u varijablu `godina_rodenja` pohranite godinu rođenja kao cijelobrojnu vrijednost. U varijablu `trenutno_vrijeme` pohranite trenutno vrijeme koristeći `new Date()` objekt.
 - Dodajte novu varijablu `godine` i u nju izračunajte koliko imate godina koristeći funkciju `getFullYear()` nad varijablom `trenutno_vrijeme` i varijablu `godina_rodenja`. Sintaksa je: `varijabla.getFullYear()`. Radi pojednostavljinjanja, prepostavljamo da je vaš rođendan već prošao ove godine.
2. Koristeći `template literals`, u konzolu ispišite "Bok moje ime je _ ** i imam ** godina.".
 - Deklarirajte dvije konstante `ime_duljina` i `prezime_duljina` u koje ćete pohraniti broj slova u vašem imenu i prezimenu koristeći funkciju `length` nad varijablama `ime` i `prezime`.
 - Ispišite u konzolu "Moje ime i prezime imaju ** i ** slova." koristeći `template literals`.

- Ispišite u konzolu "It is __ that my name and surname are of the same length" koristeći `template literals` i operator "je identično".
3. Pohranite u novu varijablu `x` kvadrat zbroja varijabli `ime_duljina` i `prezime_duljina`. Rezultat zbrojite s vašom godinom rođenja uvećanom za 1 koristeći operator `++` ispred varijable (uočite grešku, zašto nastaje, i napravite izmjenu!) te sve skupa podijelite s `2`. Sve navedeno definirajte u obliku **jednog izraza u jednoj liniji kôda**.
4. Recimo da si želite definirati daily routine koji se sastoji od nekoliko ciljeva. Koristeći logičke operatore i operatore usporedbe, definirajte varijablu `daily_routine_ostvaren`, temeljem sljedećih tvrdnjki. Vaš `daily_routine_ostvaren` je ispunjen ako:

- ste pročitali više od 50 stranica vaše omiljene knjige **ili** ste vježbali JavaScript barem 1 sat
- ste popili između litre i dvije litre vode
- ste vježbali minimalno 30 minuta **ili** ste prošetali minimalno 3 km
- ste naučili nešto novo
- ste se naspavali minimalno 7 sati
- ste se nasmijali

Za svaki od danih izraza deklarirajte varijable za ostvarenu vrijednost i ciljanu vrijednost, te boolean varijablu koja će sadržavati rezultat ostvarenja. Na primjer, za izraz `popiti između litre i dvije litre vode` deklarirajte varijable `unos_vode` i `ciljani_dnevni_unos_vode` te varijablu `dnevni_unos_vode_zadovoljen`.

- Definirajte varijablu `daily_routine_ostvaren` koja će sadržavati rezultat ostvarenja svih dnevnih ciljeva.

Programiranje u skriptnim jezicima (PJS)

Nositelj: doc. dr. sc. Nikola Tanković

Asistenti:

- Luka Blašković, mag. inf.
- Alesandro Žužić, mag. inf.

Ustanova: Sveučilište Jurja Dabrogle u Puli, Fakultet informatike u Puli



Fakultet informatike u Puli

[2] Funkcije, doseg varijabli i kontrolne strukture



Funkcije su jedan od temeljnih konstrukata u programiranju. One omogućuju grupiranje kôda u logičke cjeline koje se mogu ponovno koristiti kroz cijeli program kao i apstrakciju složenih operacija, što nam olakšava razumijevanje i održavanje kôda.

Kontrolne strukture su konstrukti u programiranju koji odlučuju o toku izvršavanja programa.

Posljednje ažurirano: 2.8.2024.

Sadržaj

- [Programiranje u skriptnim jezicima \(PJS\)](#)
- [2. Funkcije, doseg varijabli i kontrolne strukture](#)
 - [Sadržaj](#)
- [1. Uvod u funkcije](#)
 - [1.1 Osnovna sintaksa funkcija](#)
 - [1.2 Pozivanje funkcije](#)
 - [Vježba 1](#)
 - [1.3 Funkcije možemo koristiti raznoliko](#)
- [2. Doseg varijabli i funkcijski izrazi](#)
 - [2.1 Blokovski opseg \(eng. **block scope**\)](#)
 - [2.2 Ponovno deklariranje funkcija](#)
 - [2.3 Funkcijski izrazi](#)

- [Vježba 2](#)
- [Vježba 3](#)
- [3. Uvod u paradigmu funkciskog programiranja](#)
 - [3.1 Čiste funkcije](#)
 - [3.2 Imutabilnost](#)
 - [3.3 Funkcije višeg reda](#)
- [Samostalni zadatak za vježbu 2](#)
- [3. Kontrolne strukture](#)
 - [3.1 Selekcije \(eng. *Conditional statements*\)](#)
 - [3.1.1 `if` selekcija](#)
 - [3.1.2 `else` selekcija](#)
 - [3.1.3 `else if` selekcija](#)
 - [3.1.4 `switch` selekcija](#)
 - [3.2 Selekcije s logičkim operatorima](#)
 - [Primjer 1 - Selekcija vremena u danu \(operator `&&` + `if-else` selekcija\)](#)
 - [Primjer 2 - Provjera prihvatljivosti za zajam \(operator `||`, `&&` + `if-else` selekcija\)](#)
 - [Vježba 4](#)
 - [3.3 Iteracije/Petlje \(eng. *Iterations/Loops*\)](#)
 - [3.3.1 Klasična `for` petlja](#)
 - [Primjer 3 - Ispis brojeva od 1 do 100 koji su djeljivi s 3](#)
 - [3.3.2 `while` petlja](#)
 - [3.3.2.1 `do-while` petlja](#)
 - [3.3.3 Prekidanje petlji - `break` | `continue`](#)
 - [3.3.4 Petlje nad nizom znakova \(eng. *String*\)](#)
 - [3.3.5 Ugniježđene petlje](#)
 - [Primjer 4 - Ispis tablice množenja](#)
 - [Vježba 5](#)
 - [Vježba 6](#)
 - [3.4 Rekurzija \(eng. *Recursion*\)](#)
 - [3.5 Primjer 5 - Validacija forme](#)
- [Samostalni zadatak za vježbu 3](#)

1. Uvod u funkcije

Funkcije, kao što smo već spomenuli, omogućuju grupiranje kôda u logičke cjeline koje se mogu ponovno koristiti kroz cijeli program kao i apstrakciju složenih operacija, što nam olakšava razumijevanja i održavanje kôda. U JavaScriptu, funkcije ćemo deklarirati pomoću ključne riječi `function`, nakon koje slijedi:

- ime funkcije
- lista parametara funkcije, omeđena zagradama `()` i odvojena zarezima (ako ima više parametara)
- tijelo funkcije, omeđeno vitičastim zagradama `{}`

Na primjer, možemo definirati jednostavnu funkciju `kvadriraj` koja će kvadrirati broj koji joj proslijedimo kao *argument*.

```
function kvadriraj(broj) {  
    return broj * broj;  
}
```

Funkcija `kvadriraj` prima jedan parametar `broj` i vraća kvadrat tog broja. Ključnom riječi `return` funkcija vraća definiranu vrijednost. Ako funkcija ne vraća ništa, koristimo `return;` ili još jednostavnije izostavimo `return` naredbu.

Možemo primijetiti kako je funkcija `kvadriraj` zapravo vrlo slična matematičkoj funkciji $f(x) = x^2$. Funkcija `f` prima jedan parametar `x` i vraća kvadrat tog broja.

Ako povučemo paralelu sa `C` familijom jezika, možemo primijetiti da kod deklaracije funkcije u JavaScriptu, kao i varijabli, ne navodimo tip podataka parametara i povratne vrijednosti. Funkcija `kvadriraj` ekvivalentna je funkciji u C-u:

```
int kvadriraj(int broj) {  
    return broj * broj;  
}
```

Kada se izvršavaju funkcije u JavaScriptu? Funkcije u JavaScriptu se izvršavaju kada "nešto" pozove tu funkcije, primjerice to može biti:

- kada se dogodi neki događaj (eng. *event*), npr. pritisak neke tipke
- kada se pozove direktno iz Javascript kôda
- automatski (eng. *self-invoking*)

1.1 Osnovna sintaksa funkcija

Kako smo već rekli, funkcije se deklariraju ključnom riječi `function`, nakon koje slijedi **1. ime funkcije, 2. lista parametara i 3. tijelo funkcije**.

Imena funkcije mogu sadržavati slova, brojeve, povlake `_` i dolar `$` znak (ista pravila vrijede kao i kod imenovanja varijabli). Imena funkcija ne smiju počinjati brojem. Kôd koji se izvršava pišemo unutar vitičastih zagrada `{}`.

```
function imeFunkcije(parametar1, parametar2, parametar3) {  
    // tijelo funkcije koje obavlja neku operaciju  
}
```

Zapamtimo par pojmova:

- parametri funkcije (eng. **function parameters**) su navedeni unutar zagrada `()` u definiciji funkcije.
- argumenti funkcije (eng. **function arguments**) su vrijednosti koje se proslijeđuju funkciji kada se ona poziva.
- najvažnije, unutar funkcije, parametri (argumenti) se ponašaju kao **lokalne varijable**.

1.2 Pozivanje funkcije

Deklariranje funkcije neće pozvati funkciju, već samo definira funkciju. Da bismo pozvali funkciju, koristimo ime funkcije, operator `()` i unutar njega argumente koje proslijeđujemo funkciji. Primjerice, kako bi pozvali našu funkciju `kvadriraj` s argumentom `5` i ispisali rezultat u konzolu, pišemo sljedeći kôd:

```
console.log(kvadriraj(5)); // 25
```

Deklariramo funkciju `toCelsius` koja će pretvoriti temperaturu iz Fahrenheit u Celzijevu. Formula za pretvorbu je: $c = \frac{5}{9} * (F - 32)$.

Funkciju smo definirali ovako:

```
function toCelsius(fahrenheit) {  
    return (5 / 9) * (fahrenheit - 32);  
}
```

Idemo pozvati funkciju s argumentom `77` i ispisati rezultat u konzolu:

```
console.log(toCelsius(77)); // 25
```

Dobili smo rezultat `25`, odnosno 77°F je 25°C .

Što će ispisati sljedeći kôd?

```
let value = toCelsius();  
console.log(value); // ?
```

Odgovor je `Nan` (eng. **Not a Number**). Zašto? Funkcija `toCelsius` očekuje jedan argument, a mi nismo proslijedili niti jedan. Kako bismo izbjegli ovakve situacije, možemo postaviti defaultnu vrijednost za parametar funkcije, na primjer:

```
function toCelsius(fahrenheit = 0) {  
    return (5 / 9) * (fahrenheit - 32);  
}
```

Poziv funkcije `toCelsius()` sada će nam vratiti `0`, jer smo postavili defaultnu vrijednost za parametar `fahrenheit`.

Sada će nam `toCelsius()` vratiti `-17.777`.

JavaScript nam neće dati grešku ako slučajno pozovemo funkciju bez `()` operatora, već će to tretirati kao referencu na samu funkciju. Ovo može biti korisno u nekim situacijama, ali u pravilu želimo ovo izbjegavati.

```
let value = toCelsius;
console.log(value); // [Function: toCelsius]
```

Vježba 1

Napišite funkciju `pozdrav` koja će primati jedan argument `ime` te će ispisati poruku i vratiti string vrijednost "Pozdrav, `ime`!". Funkciju pozovite s argumentom `"Ivan"` i ispišite rezultat u konzolu. Kada to napravite dodajte defaultnu vrijednost za parametar `ime` koja će biti `"stranac"`.

Rezultat:

```
Pozdrav Ivan!                                     script.js:2
```

```
>
```

1.3 Funkcije možemo koristiti raznoliko

U JavaScriptu, funkcije se mogu koristiti na jednak način kao što koristimo varijable. To znači da ih možemo dodijeliti varijablama, proslijediti kao argumente drugim funkcijama, koristiti kao pridruživanje vrijednosti objektima i sl.

Primjerice, umjesto da koristimo varijablu za pohranu rezultata funkcije, možemo koristiti sam poziv funkcije!

Uzmimo našu funkciju `kvadriraj`:

```
function kvadriraj(broj) {
    return broj * broj;
}

let rezultat = kvadriraj(5);
let text = "Rezultat kvadriranja broja 5 je: " + rezultat;
console.log(text); // Rezultat kvadriranja broja 5 je: 25
```

možemo napisati i ovako:

```
let text2 = "Rezultat kvadriranja broja 5 je: " + kvadriraj(5);
console.log(text2); // Rezultat kvadriranja broja 5 je: 25
```

Što bi se dogodilo ako kôd posložimo na ovaj način?

```

let text3 = kvadriraj(5) + " je rezultat kvadriranja broja 5.";
function kvadriraj(broj) {
    return broj * broj;
}
console.log(text3); // ?

```

Primijetite da smo pozvali funkciju `kvadriraj` prije nego smo ju deklarirali. JavaScript će prvo pročitati sve deklaracije funkcija i varijabli prije nego počne izvršavati kôd, tako da ovaj kôd neće proizvesti grešku i ispisat će `25 je rezultat kvadriranja broja 5.`. Ovo ponašanje se zove **Function hoisting**. Dakle prethodni kôd je ekvivalentan ovome:

```

function kvadriraj(broj) {
    return broj * broj;
}
let text3 = kvadriraj(5) + " je rezultat kvadriranja broja 5.";
console.log(text3); // 25 je rezultat kvadriranja broja 5.

```

Napomena, navedeno ponašanje odnosi samo na deklaracije funkcija, ne i na funkcijске izraze (eng. *function expressions*). O funkcijskim izrazima više u nastavku skripte.

2. Doseg varijabli i funkcijski izrazi

Doseg varijabli (eng. *variable scope*) odnosi se na pravila gdje u kôdu varijabla može biti korištena/pročitana. U JavaScriptu, varijable deklarirane unutar funkcije su **lokalne varijable** i mogu se koristiti samo unutar te funkcije. Varijable deklarirane izvan funkcije su globalne varijable i mogu se koristiti bilo gdje u kôdu (ako nisu unutar nekog drugog bloka).

```

// Kôd ovdje ne može koristiti varijablu x
function myFunction() {
    let x = 10;
    // Kôd ovdje može koristiti varijablu x
    console.log(x); // 10
}
// Kôd ovdje ne može koristiti varijablu x
console.log(x); // ReferenceError: x is not defined

```

Budući da se lokalne varijable prepoznaju samo unutar njihovih funkcija, varijable s istim imenom mogu postojati u različitim funkcijama.

Važno je napomenuti da se lokalne varijable stvaraju svaki put kada se funkcija pozove, a dealociraju kada se funkcija završi.

```
// Ove varijable definirane su u globalnom dosegu
const number_1 = 20;
const number_2 = 10;

// Ova funkcija definirana je u globalnom dosegu
function pomnozi() {
    return number_1 * number_2;
}

console.log(pomnozi()); // 200
```

Ovo je jasno, međutim hoće li sljedeći kôd ispisati `100` ili dati grešku?

```
const number_1 = 20;
const number_2 = 10;

function pomnozi() {
    const number_1 = 2;
    const number_2 = 50;
    return number_1 * number_2;
}

console.log(pomnozi()); // ?
```

► Odgovor

```
console.log(pomnozi()); // 100
```

2.1 Blokovski opseg (eng. *block scope*)

U JavaScriptu, varijable deklarirane s ključnim riječima `let` i `const` imaju blokovski opseg. To znači da su vidljive samo unutar bloka kôda u kojem su deklarirane, slično kao lokalne varijable deklarirane unutar funkcija, blok kôda se definira vitičastim zagradama `{}`.

```
const x = 10;
// x ovdje iznosi 10
{
    const x = 2;
    // x ovdje iznosi 2
}
// x ovdje iznosi 10
console.log(x); // 10
```

Možemo primijetiti da se varijabla `x` deklarirana unutar bloka `{}` ponaša kao lokalna varijabla unutar bloka, a varijabla `x` deklarirana izvan bloka ponaša se kao globalna varijabla.

Ponovna deklaracija varijable s ključnom riječi `let` ili redeklaracija ključnom riječi `const`, unutar istog dosega, uzrokovat će grešku!

```

let x = 10; // Okej
const x = 2; // SyntaxError: Identifier 'x' has already been declared

{
  let x = 2; // Okej
  const x = 2; // SyntaxError: Identifier 'x' has already been declared
}
{
  const x = 2; //Okej
  const x = 2; // SyntaxError: Identifier 'x' has already been declared
}

```

Uočimo i ovaj primjer: Ponovna deklaracija `const` varijable, unutar istog dosega, uzrokovat će grešku!

```

const x = 10; // Okej
x = 2; // TypeError: Assignment to constant variable.
let x = 2; // SyntaxError: Identifier 'x' has already been declared
const x = 2; // SyntaxError: Identifier 'x' has already been declared

{
  const x = 2; // Okej
  x = 2; // TypeError: Assignment to constant variable.
  let x = 2; // SyntaxError: Identifier 'x' has already been declared
  const x = 2; // SyntaxError: Identifier 'x' has already been declared
}

```

Međutim, ponovna deklaracija `const` varijable, unutar različitih dosega, neće uzrokovati grešku!

```

const x = 10; // Okej
{
  const x = 2; // Okej
}
{
  const x = "Pas"; // Okej
}

```

Kao što je već rečeno u prethodnoj skripti, varijable deklarirane s ključnom riječi `var` nemaju blokovski opseg već funkcionalni, što znači da su vidljive unutar funkcija u kojoj su deklarirane, kao i unutar svih blokova i podfunkcija. Ovo ponašanje može dovesti do neočekivanih rezultata i grešaka, stoga se preporučuje korištenje isključivo ključnih riječi `let` i `const` koje imaju blokovski opseg, umjesto `var`.

```

var x = 1;
{
  var x = 2;
}
console.log(x); // 2 - neočekivano! Zadržimo se na ključnim riječima let i const!

```

```

let x = 1;
const y = 2;
{
  let x = 2;
  const y = 3;
}
console.log(x, y); // 1 2 - očekivano!

```

Za one koji žele naučiti više o blokovskom opsegu, i function hoistingu, link je [ovdje](#).

2.2 Ponovno deklariranje funkcija

Ponovno deklariranje funkcija u JavaScriptu s ključnom riječi `function` dozvoljeno je ovisno o dosegu gdje se funkcija deklarira.

Deklaracije funkcija s ključnom riječi `function` ponašaju se slično kao `var` i mogu se ponovno deklarirati s još jednom `function` ili `var` deklaracijom, ali ne sa `let`, `const` ili `class` deklaracijom.

```

function a(b) {}
function a(b, c) {}
console.log(a.length); // 2 - broj parametara zadnje deklarirane funkcije
let a = 2; // SyntaxError: Identifier 'a' has already been declared

```

Ako "overridamo" funkciju s `var` deklaracijom, to će raditi, ali još jednom, nije preporučljivo.

```

var a = 1;
function a() {}
console.log(a); // 1

```

2.3 Funkcijski izrazi

Funkcijski izrazi (eng. *function expressions*) su način definiranja funkcija kao vrijednosti varijable. Mogu se koristiti kako bi **definirali funkciju unutar izraza**.

Funkcijski izrazi također se definiraju s ključnom riječi `function`, ali se razlikuju od "deklaracija funkcija" po tome što se mogu dodijeliti varijablama, proslijediti kao argumenti drugim funkcijama, koristiti kao pridruživanje vrijednosti objektima i sl. Sintaksa je vrlo slična kao i kod klasične `function` deklaracije.

```

const izracunaj_povrsinu_pravokutnika = function (duzina, sirina) {
  return duzina * sirina;
};
console.log(izracunaj_povrsinu_pravokutnika(5, 3)); // 15 - funkcijски izraz pozivamo na isti način kao i deklarirane funkcije

```

Kako razlikujemo deklaraciju funkcije i funkcijske izraze? Uzmimo za primjer funkciju `zbroji` koja zbraja dva broja.

Deklaracija funkcije izgleda ovako:

```
function zbroji(a, b) {  
    return a + b;  
}
```

Funkcijski izraz izgleda ovako:

```
const zbroji = function (a, b) {  
    return a + b;  
};
```

Možemo primijetiti da se kod funkcijskog izraza funkcija "izrađuje" s desne strane operatora dodjeljivanja `=`.

Kako smo ranije spomenuli, u poglavlju 1.3, **function hoisting** ponašanje dovodi do toga da se deklaracije funkcija mogu pozvati prije nego su deklarirane. Međutim, to se ne odnosi na funkcijске izraze. Funkcijski izrazi se ponašaju kao bilo koja druga varijabla, i ne mogu se pozvati prije nego su deklarirane.

```
zbroji(2, 3); // 5  
function zbroji(a, b) {  
    console.log(a + b);  
    return a + b;  
}
```

Funkcijski izraz:

```
zbroji(2, 3); // TypeError: zbroji is not a function  
let zbroji = function (a, b) {  
    console.log(a + b);  
    return a + b;  
};
```

Možemo li deklarirati funkciju unutar funkcije? Naravno 😊

```
function vanjskaFunkcija() {  
    function unutarnjaFunkcija() {  
        console.log("Pozdrav iz unutarnje funkcije!");  
    }  
    console.log("Pozdrav iz vanjske funkcije!");  
    unutarnjaFunkcija();  
}  
vanjskaFunkcija();  
// Ispis:  
// Pozdrav iz vanjske funkcije!  
// Pozdrav iz unutarnje funkcije!
```

Isto tako, možemo deklarirati i funkcijski izraz unutar funkcije.

```

function vanjskaFunkcija() {
    const unutarnjaFunkcija = function () {
        console.log("Pozdrav iz unutarnje funkcije!");
    };
    console.log("Pozdrav iz vanjske funkcije!");
    unutarnjaFunkcija();
}
vanjskaFunkcija();
// Ispis:
// Pozdrav iz vanjske funkcije!
// Pozdrav iz unutarnje funkcije!

```

Svaka funkcija ima svoj lokalni doseg varijabli, što znači da varijable deklarirane unutar unutarnje funkcije nisu vidljive vanjskoj funkciji (vanjska je ona koja omeđuje unutarnju)?

```

function vanjskaFunkcija() {
    const unutarnjaFunkcija = function () {
        const x = 5;
        console.log("Pozdrav iz unutarnje funkcije!");
    };
    console.log("Pozdrav iz vanjske funkcije!");
    unutarnjaFunkcija();
    console.log(x); // ReferenceError: x is not defined
}
vanjskaFunkcija();

```

Međutim, kako svaka funkcija može vratiti vrijednost putem `return` naredbe, tako unutarnja funkcija može vratiti vrijednost vanjskoj funkciji.

```

function vanjskaFunkcija() {
    const unutarnjaFunkcija = function () {
        return "Pozdrav iz unutarnje funkcije!";
    };
    console.log("Pozdrav iz vanjske funkcije!");
    const poruka = unutarnjaFunkcija();
    console.log(poruka); // Pozdrav iz unutarnje funkcije!
}
vanjskaFunkcija();

```

Vježba 2

Napišite funkciju `sve_o_krugu(r)` s jednim parametrom `r` koji predstavlja radijus kruga. Funkcija treba sadržavati dvije unutarnje funkcije `povrsina` i `opseg` koje će računati površinu i opseg kruga i vraćati vanjskoj funkciji rezultate. Jedna od dvije unutarnje funkcije treba koristiti funkcionalni izraz, a druga deklaraciju funkcije. Vanjska funkcija treba ispisati rezultate unutarnjih funkcija u konzolu. Za vrijednost broja π koristite `Math.PI`. Vanjska funkcija treba u lokalnu varijablu `zbroj` pohraniti zbroj površine i opsega kruga i vratiti **tu vrijednost**. Rezultat funkcije `sve_o_krugu(3)` pohranite u globalnu varijablu `zbroj` te ju ispišite u konzolu.

EduCoder šifra: krug

Rezultat:

Površina kruga je: 28.27433882308138	script.js:8
Opseg kruga je: 18.84955592153876	script.js:9
47.12388980384689	index.html:10

Vježba 3

Napišite funkciju `lessby20_others(x, y, z)` koja prima tri cijelobrojna argumenta: `x`, `y` i `z`. Funkcija treba provjeriti i vratiti `true` ako bilo koji od ovih brojeva zadovoljava sljedeće uvjete:

- Broj je veći ili jednak 20.
- Broj je manji od barem jednog od preostala dva broja.

U svim ostalim slučajevima, funkcija treba vratiti `false`.

EduCoder šifra: `lessby20_others`

Rezultat:

```
console.log(lessby20_others(23, 45, 10)); //true
console.log(lessby20_others(23, 23, 10)); //false
console.log(lessby20_others(10, 25, 75)); //true
```

3. Uvod u paradigmu funkciskog programiranja

Funkcijsko programiranje (eng. *functional programming*) je paradigma programiranja koja se temelji na korištenju funkcija kao osnovnih gradivnih blokova.

Funkcijsko programiranje možemo zamisliti kao princip pisanja računalnih programa gdje primarno koristimo funkcije kao osnovne gradivne blokove, a ne npr. objekte, klase, varijable i sl.

Funkcijsko programiranje možemo usporediti s LEGO kockicama. Svaki LEGO blok (funkcija) ima svoju specifičnu svrhu i obavlja jednu stvar dobro. Kombiniranjem tih blokova možemo izgraditi složene strukture (programe).

3.1 Čiste funkcije

Jedno od svojstava funkciskog programiranja je **čista funkcija** (eng. *pure function*). Čista funkcija je funkcija koja ne mijenja stanje varijabli izvan svojeg dosega, ali niti ne ovisi o stanju varijabli izvan svog dosega. Čista funkcija uvijek vraća isti rezultat za iste ulazne parametre.

```
// Čista funkcija - ne ovisi o stanju varijabli izvan svog dosega i ne mijenja stanje
varijabli izvan svog dosega
function kvadriraj(broj) {
    return broj * broj;
}
console.log(kvadriraj(5)); // 25
```

```
// Nečista funkcija - ovisi o stanju varijabli izvan svog dosega i mijenja stanje
varijabli izvan svog dosega
let rezultat = 0;
let broj = 5;
function kvadriraj() {
    rezultat = broj * broj;
    return rezultat;
}
console.log(kvadriraj()); // 25
```

3.2 Imutabilnost

Imutabilnost odnosno nepromjenjivost (eng. **immutability**) je još jedno svojstvo funkcionskog programiranja. Imutabilnost se odnosi na to da se vrijednosti varijabli ne mijenjaju jednom nakon što su definirane.

Uzmimo za primjer inkrement/dekrement operatore `++` i `--` koji mijenjaju vrijednosti varijable nad kojom se koriste. U funkcionskom programiranju, umjesto da mijenjamo vrijednost varijable, htjeli bismo stvoriti novu varijablu s novom vrijednošću.

```
let x = 5;
x++; // mijenja vrijednost varijable x
console.log(x); // 6
```

```
let x = 5;
let y = x + 1; // stvara novu varijablu y s novom vrijednošću
console.log(x, y); // 5 6
```

ili

```
function inkrement(x) {
    return x + 1;
}
let x = 5;
let y = inkrement(x); // stvara novu varijablu y s novom vrijednošću
console.log(x, y); // 5 6
```

3.3 Funkcije višeg reda

Funkcije višeg reda (eng. **higher-order functions**) su funkcije koje primaju druge funkcije kao argumente ili vraćaju druge funkcije kao rezultat. Funkcije višeg reda omogućuju nam da apstrahiramo zajedničke obrasce u funkcijama i da ih koristimo kao argumente drugim funkcijama.

Idemo napraviti jednostavan kalkulator koji može zbrajati i oduzimati. Napišimo funkcije `zbroji` i `oduzmi` koje će primati dva argumenta i vraćati rezultat zbrajanja i oduzimanja.

```
function zbroji(a, b) {
    return a + b;
}

function oduzmi(a, b) {
    return a - b;
}

console.log(zbroji(5, 3)); // 8
console.log(oduzmi(5, 3)); // 2
```

Rekli smo da je funkcija višeg reda koja prima drugu funkciju kao argument ili vraća drugu funkciju kao rezultat. Možemo implementirati funkciju `izracunaj` koja će primiti funkciju `operacija` i dva broja `a` i `b` te će vratiti rezultat funkcije `operacija` s argumentima `a` i `b`.

```
function izracunaj(operacija, a, b) {
    return operacija(a, b);
}

console.log(izracunaj(zbroji, 5, 3)); // 8
console.log(izracunaj(oduzmi, 5, 3)); // 2
```

Želimo deklarirati funkcije `double` i `triple` koje će primati jedan broj i vraćati dvostruko odnosno trostruko veći broj.

```
function double(x) {
    return x * 2;
}

function triple(x) {
    return x * 3;
}

console.log(double(5)); // 10
console.log(triple(5)); // 15
```

Što ako želimo dodati funkcije `quadruple` i `quintuple` koje će vraćati četverostruko odnosno petostruko veći broj? Recimo da želimo ostati na tome da naša funkcija prima samo jedan argument. Možemo li to riješiti pomoću funkcija višeg reda?

Možemo! Deklarirat ćemo funkciju `multiplier` koja će primati jedan argument `value` te će vraćati funkciju koja će primati jedan argument `x` i vraćati `x * multiplier`.

Dakle `multiplier` je funkcija višeg reda jer vraća funkciju kao povratnu vrijednost.

```
function multiplier(value) {
    return function (x) {
        return x * value;
```

```

    };
}

let double = multiplier(2);
let triple = multiplier(3);
let quadruple = multiplier(4);
let quintuple = multiplier(5);

console.log(double(5)); // 10
console.log(triple(5)); // 15
console.log(quadruple(5)); // 20
console.log(quintuple(5)); // 25

```

Samostalni zadatak za vježbu 2

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

EduCoder šifra: `bmi_and_heron`

1. Napišite **funkciju** `provjera_parnosti` koja će provjeravati je li broj paran ili neparan. Funkcija treba primiti jedan parametar `broj` i vratiti boolean vrijednosti "true" za parnost ili "false" za neparnost. Funkciju napišite **bez** upotrebe selekcija (if, else, switch) Funkciju pozovite s argumentom `5` i ispišite rezultat u konzolu.
2. Napišite **funkcijski izraz** `izrazunaj_povrsinu` koji računa površinu pravokutnika. U varijablu `povrsina` pohranite taj funkcijski izraz. Ispišite vrijednost `povrsina(8,6)` u konzolu.
3. Napišite **funkcijski izraz** `BMI` koji računa BMI (Body Mass Index) osobe. BMI se računa prema formuli `BMI = težina / (visina * visina)`. Ispišite u konzolu BMI osobe koja ima težinu 75 kg i visinu 1.75 m.
4. Napišite **funkciju** `heron()` koja će računati površinu trokuta prema Heronovoj formuli. Funkcija treba primiti tri parametra `a`, `b` i `c` koji predstavljaju duljine stranica trokuta.
 - o Heronova formula: $P = \sqrt{(p * (p - a) * (p - b) * (p - c))}$ gdje je `p` poluopseg trokuta, a računa se prema formuli $p = (a + b + c) / 2$. Koristite funkciju `Math.sqrt()` za računanje korijena (Sintaksa je: `Math.sqrt(broj)`)
 - o Napišite funkcijski izraz `poluopseg` koji će primiti tri parametra `a`, `b` i `c` te vratiti poluopseg trokuta prema danoj formuli. Funkcijski izraz mora biti definiran unutar funkcije `heron()`.
 - o Unutar funkcije Heron, deklarirajte novu konstantu `p` koja će pohraniti vrijednost funkcijskog izraza `poluopseg(a, b, c)`.
 - o Rezultat funkcije `heron(3, 4, 5)` pohranite u varijablu `povrsina_trokuta` te ispišite u konzolu:
`Trokut s duljinama stranica 3, 4 i 5 ima površinu: povrsina_trokuta(3, 4, 5) cm2`
koristeći `template_literals`.
5. Sljedeći JavaScript kôd sadrži nekoliko grešaka. Pronađite i ispravite greške kako bi kôd radio ispravno. Provjerite s pozivom funkcije `izracunaj(x, y, z);` koji mora ispisati `17` i `3`.

```

const x = 10;
const y = 5;
const z = 2;

```

```

function izracunaj(x, y, z) {
    let x = 5;
    let y = 3;
    let z = 2;

    function = zbroji() {
        return x + y + z;
    }
    console.log(function(zbroji(x,y,z)))

    const oduzmi = function () = {
        return y - x - z;
    }
    console.log(oduzmi());
}

// Provjera: izracunaj(x, y, z); mora ispisati sljedeće:
// 17
// 3

```

3. Kontrolne strukture

Kontrolne strukture su konstrukti koji odlučuju o toku izvršavanja programa na temelju određenih uvjeta. Ako je uvjet ispunjen tada se izvršava određeni blok radnji, inače će se izvršavati drugi blok radnji koji zadovoljava taj uvjet. Kontrolne strukture možemo podijeliti u dvije kategorije:

1. Selekcije (eng. **Conditional statements**) - odlučuju o toku izvršavanja bloka kôda na temelju logičkog izraza koji se evaluira u `true` ili `false`.
2. Iteracije/Petlje (eng. **Iterations**) - omogućuju izvršavanje bloka kôda više puta dok se ne ispuni uvjet definiran logičkim izrazom, koji se evaluira u `true` ili `false`.

3.1 Selekcije (eng. **Conditional statements**)

U JavaScriptu, kao i u većini programskih jezika, selekcije se pišu pomoću ključnih riječi `if`, `else if` i `else` te `switch`. Kada koristimo koju selekciju ovisi o tome koliko uvjeta želimo provjeriti:

- `if` selekciju koristimo kako bi specificirali blok kôda koji se izvršava ako je evaluirani logički izraz `true`
- `else` selekciju koristimo kako bi specificirali blok kôda koji se izvršava ako je evaluirani logički izraz `false`
- `else if` selekciju koristimo kako bi provjerili novi logički izraz ako je prethodni izraz unutar `if` ili `if else` bio `false`
- `switch` selekciju koristimo kada imamo puno alternativnih uvjeta (logičkih izraza) koje želimo provjeriti

3.1.1 `if` selekcija

Koristimo `if` selekciju kako bi specificirali blok kôda koji se izvršava ako je evaluirani logički izraz `true`. Sintaksa je sljedeća:

```
if (logicki_izraz) {  
    // blok kôda koji se izvršava ako je logicki_izraz = true  
}
```

Pripazite da je blok kôda uvučen unutar vitičastih zagrada `{}`. Ako je logički izraz `true`, izvršava se blok kôda unutar vitičastih zagrada `{}`. Ako je logički izraz `false`, blok kôda se preskače. Primjer:

```
let x = 10;  
if (x < 5) {  
    console.log("x je manji od 5"); // neće se ispisati  
}
```

Ako izostavimo vitičaste zagrade `{}`, JavaScript će izvršiti samo prvu liniju kôda nakon `if` selekcije. Ovo ponašanje može dovesti do neočekivanih rezultata i grešaka, stoga se preporučuje korištenje vitičastih zagrada `{}`.

3.1.2 `else` selekcija

Koristimo `else` selekciju kako bi specificirali blok kôda koji se izvršava ako je evaluirani logički izraz `false`. Sintaksa je sljedeća:

```
if (logicki_izraz) {  
    // blok kôda koji se izvršava ako je logicki_izraz = true  
} else {  
    // blok kôda koji se izvršava ako je logicki_izraz = false  
}
```

Primjer:

```
let x = 10;  
if (x < 5) {  
    console.log("x je manji od 5"); // neće se ispisati  
} else {  
    console.log("x je veći ili jednak 5"); // ispisat će se  
}
```

3.1.3 `else if` selekcija

Koristimo `else if` selekciju kako bi provjerili novi logički izraz ako je prethodni bio `false`. Sintaksa je sljedeća:

```

if (logicki_izraz_1) {
    // blok kôda koji se izvršava ako je logicki_izraz_1 = true
} else if (logicki_izraz_2) {
    // blok kôda koji se izvršava ako je logicki_izraz_2 = true
} else {
    // blok kôda koji se izvršava ako su svi prethodni logički izrazi (logicki_izraz_1 &&
logicki_izraz_2) = false
}

```

Primjer:

```

let x = 10;
if (x < 5) {
    console.log("x je manji od 5"); // neće se ispisati
} else if (x === 5) {
    console.log("x je jednak 5"); // neće se ispisati
} else {
    console.log("x je veći od 5"); // ispisat će se
}

```

3.1.4 switch selekcija

`switch` selekcija koristi se kada imamo puno alternativnih uvjeta (logičkih izraza) koje želimo provjeriti. Selekcija se sastoji od ključnih riječi `switch`, `case` i `default`, gdje `switch` predstavlja izraz koji se provjerava, `case` predstavlja moguće vrijednosti izraza, a `default` predstavlja blok kôda koji se izvršava ako niti jedan od prethodnih uvjeta nije ispunjen. Sintaksa je sljedeća:

```

switch (izraz) {
    case vrijednost_1:
        // blok kôda koji se izvršava ako je izraz = vrijednost_1
        break;
    case vrijednost_2:
        // blok kôda koji se izvršava ako je izraz = vrijednost_2
        break;
    default:
        // blok kôda koji se izvršava ako niti jedan od prethodnih uvjeta nije ispunjen
}

```

Kao i u C jezicima, nakon svakog bloka kôda u `case` selekciji koristimo ključnu riječ `break` kako bi prekinuli izvršavanje selekcije. Ako izostavimo `break` naredbu, JavaScript će izvršiti sve blokove kôda nakon prvog koji zadovoljava uvjet, što može dovesti do neočekivanih rezultata i grešaka, stoga se gotovo uvijek koristi `break` naredba.

Primjer:

```

let dan = "srijeda";
switch (dan) {
    case "ponedjeljak":
        console.log("Danas je ponedjeljak");

```

```

        break;
    case "utorak":
        console.log("Danas je utorak");
        break;
    case "srijeda":
        console.log("Danas je srijeda");
        break;
    case "četvrtak":
        console.log("Danas je četvrtak");
        break;
    case "petak":
        console.log("Danas je petak");
        break;
    default:
        console.log("Vikend je!");
}

```

3.2 Selekcije s logičkim operatorima

Selekcije s logičkim operatorima koriste se kako bi provjerili više uvjeta istovremeno. U JavaScriptu, kao i u većini programskih jezika, primarno koristimo logičke operatore `&&` (i), `||` (ili) i `!` (negacija) kako bi provjerili više uvjeta istovremeno. Logički operatori vraćaju `true` ili `false` ovisno o rezultatu provjere uvjeta.

Najlakše je objasniti logičke operatore kroz konkretne primjere:

Primjer 1 - Selekcija vremena u danu (operator `&&` + `if-else` selekcija)

Prije nego što krenemo sa samim kôdom, zapisat ćemo nekoliko tvrdnji koje ćemo provjeravati logičkim operatorima:

- Ako je vrijeme između 6 i 12 sati, pozdravit ćemo s "Dobro jutro!"
- Ako je vrijeme između 12 i 18 sati, pozdravit ćemo s "Dobar dan!"
- Inače ćemo pozdraviti s "Dobra večer!"

Idemo prvo ugrubo definirati strukturu kôda:

```

let sat = 10;

if (uvjet) {
    izraz;
} else if (drugiUvjet) {
    izraz;
} else {
    izraz;
}

```

Krenimo s popunjavanjem onim redoslijedom kako smo naveli tvrdnje:

Prvi uvjet: Ako je vrijeme između 6 i 12 sati, pozdraviti ćemo s "Dobro jutro!" - `if (sat >= 6 && sat < 12)` - koristimo logički operator `&&` (i) kako bi provjerili oba uvjeta istovremeno. Ako je `sat` veći ili jednak `6` i manji od `12`, odnosno, `(6 <= sat < 12)` ispisat ćemo "Dobro jutro!" .

```
let sat = 10;
if (sat >= 6 && sat < 12) {
    console.log("Dobro jutro!");
} else if (drugiUvjet) {
    izraz;
} else {
    izraz;
}
```

Nastavljamo dalje, drugi uvjet: Ako je vrijeme između 12 i 18 sati, pozdraviti ćemo s "Dobar dan!" - `else if (sat >= 12 && sat < 18)` - koristimo logički operator `&&` (i) kako bi provjerili oba uvjeta istovremeno. Ako je `sat` veći ili jednak `12` i manji od `18`, odnosno, `(12 <= sat < 18)` ispisat ćemo "Dobar dan!" .

```
let sat = 10;
if (sat >= 6 && sat < 12) {
    console.log("Dobro jutro!");
} else if (sat >= 12 && sat < 18) {
    console.log("Dobar dan!");
} else {
    izraz;
}
```

I na kraju, treći uvjet: Inače ćemo pozdraviti s "Dobra večer!" - `else` - ako niti jedan od prethodnih uvjeta nije ispunjen, ispisati ćemo "Dobra večer!" .

```
let sat = 10;
if (sat >= 6 && sat < 12) {
    console.log("Dobro jutro!");
} else if (sat >= 12 && sat < 18) {
    console.log("Dobar dan!");
} else {
    console.log("Dobra večer!");
}
```

Primjer 2 - Provjera prihvatljivosti za zajam (operator `||`, `&&` + `if-else` selekcija)

U ovom primjeru simulirati ćemo provjeru prihvatljivosti klijenta za zajam na temelju nekoliko kriterija, koristeći logičke operatore `||` (ili) i `&&` (i).

Izmislit ćemo nekoliko tvrdnji koje ćemo provjeravati logičkim operatorima:

- Ako je klijent zaposlen i ima stabilne prihode veće od 7000 novčanih jedinica, može dobiti zajam.
- Ako je klijent samostalni obrtnik ili ima visoku kreditnu ocjenu, može dobiti zajam.

- Ako klijent ima barem 2 godine radnog iskustva ili je stariji od 25 godina i ima mjesечne prihode iznad 5000 novčanih jedinica, može dobiti zajam.

Svaka od tvrdnji je neovisna o drugima, odnosno barem jedna mora biti ispunjena kako bi klijent bio prihvativ za zajam!

Koje varijable možemo iščitati iz ovih tvrdnji?

- zaposlen - `boolean`
- obrtnik - `boolean`
- kreditnaOcjenaVisoka - `boolean`
- godineRadnogLskustva - `number`
- dob - `number`
- mjesecniPrihodi - `number`

Krenimo s popunjavanjem onim redoslijedom kako smo naveli tvrdnje:

Prvi uvjet: Ako je klijent zaposlen i ima stabilne prihode veće od 7000 novčanih jedinica, može dobiti zajam
`- if (zaposlen == true && mjesecniPrihodi > 7000)` - koristimo logički operator `&&` (i) kako bi provjerili oba uvjeta istovremeno. Ako je `zaposlen` i `mjesecniPrihodi` veći od `7000`, odnosno, `(zaposlen == true && mjesecniPrihodi > 7000)` klijent može dobiti zajam.

```
let zaposlen = true;
let mjesecniPrihodi = 8000;

if (zaposlen == true && mjesecniPrihodi > 7000) {
  console.log("Čestitamo! Možete dobiti zajam!");
} else {
  console.log("Nažalost, ne možete dobiti zajam.");
}
```

Prisjetimo se kratko kako JavaScript evaluira tvrdnje (eng. **expressions**) unutar kontrolnih struktura.

Što će vratiti (u što će se evaluirati), u kôdu iznad, izraz `zaposlen == true`? Odgovor je `true`.

Ako smo sigurni da je varijabla `zaposlen` uvijek tipa `boolean`, možemo izostaviti `== true` i napisati samo `if (zaposlen && mjesecniPrihodi > 7000)`.

```
let zaposlen = true;
let mjesecniPrihodi = 8000;

if (zaposlen && mjesecniPrihodi > 7000) {
  //Ovakav zapis je dovoljan, pa i čitljiviji
  console.log("Čestitamo! Možete dobiti zajam!");
} else {
  console.log("Nažalost, ne možete dobiti zajam.");
}
```

Drugi uvjet: Ako je klijent samostalni obrtnik ili ima visoku kreditnu ocjenu, može dobiti zajam - `else if (obrtnik || kreditnaOcjenaVisoka)` - koristimo logički operator `||` (ili) kako bi provjerili jedan od dva uvjeta. Ako je `obrtnik` ili `kreditnaOcjenaVisoka` istinita tvrdnja, odnosno, `(obrtnik || kreditnaOcjenaVisoka)` klijent može dobiti zajam. Primijetite da smo izostavili `== true` jer su `obrtnik` i `kreditnaOcjenaVisoka` tipa `boolean`.

```
let zaposlen = true;
let mjesecniPrihodi = 8000;

let obrtnik = true;
let kreditnaOcjenaVisoka = false;

if ((zaposlen && mjesecniPrihodi > 7000) || obrtnik || kreditnaOcjenaVisoka) {
    console.log("Čestitamo! Možete dobiti zajam!");
} else {
    console.log("Nažalost, ne možete dobiti zajam.");
}
```

Treći uvjet: Ako klijent ima barem 2 godine radnog iskustva ili je stariji od 25 godina i ima stabilne mjesecne prihode, može dobiti zajam - `(godineRadnogIskustva >= 2 || (dob > 25 && mjesecniPrihodi > 5000))` - koristimo logički operator `||` (ili) kako bi provjerili jedan od dva uvjeta. Ako je `godineRadnogIskustva` veće ili jednako `2` ili je `dob` veći od `25` i `mjesecniPrihodi` veći od `5000`, odnosno, `(godineRadnogIskustva >= 2 || (dob > 25 && mjesecniPrihodi > 5000))` klijent može dobiti zajam.

```
let zaposlen = true;
let mjesecniPrihodi = 8000;

let obrtnik = true;
let kreditnaOcjenaVisoka = false;

let godineRadnogIskustva = 3;
let dob = 28;

if (
    (zaposlen && mjesecniPrihodi > 7000) ||
    obrtnik ||
    kreditnaOcjenaVisoka ||
    godineRadnogIskustva >= 2 ||
    (dob > 25 && mjesecniPrihodi > 5000)
) {
    console.log("Čestitamo! Možete dobiti zajam!");
} else {
    console.log("Nažalost, ne možete dobiti zajam.");
}
```

Kako su uvjeti neovisni jedan o drugome, odnosno barem jedan uvjet mora biti ispunjen, možemo komplikirani izraz unutar `if` selekcije podijeliti u više manjih izraza kako bi kod bio čitljiviji.

```
// Varijable ostaju iste

if (zaposlen && mjesecniPrihodi > 7000) {
    console.log("Čestitamo! Možete dobiti zajam!");
} else if (obrtnik || kreditnaOcjenaVisoka) {
    console.log("Čestitamo! Možete dobiti zajam!");
} else if (godineRadnogIskustva >= 2 || (dob > 25 && mjesecniPrihodi > 5000)) {
    console.log("Čestitamo! Možete dobiti zajam!");
} else {
    console.log("Nažalost, ne možete dobiti zajam.");
}
```

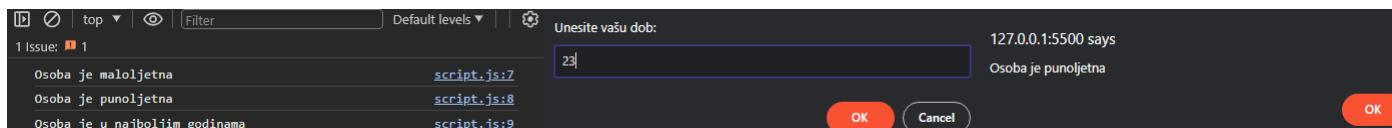
Vježba 4

Napiši funkciju `provjeriDob(dob)` koja vraća poruku ovisno o dobi korisnika. Za dob manju od 18 godina, funkcija vraća poruku "Osoba je maloljetna.". Za dob između 18 i 65 godina, funkcija vraća poruku "Osoba je punoljetna.". Za dob veću od 65 godina, funkcija vraća poruku "Osoba je u zlatnim godinama.". Pozovite `provjeriDob(15)`, `provjeriDob(25)` i `provjeriDob(70)` te ispišite rezultate u konzolu. Kada to napravite, umjesto da ručno mijenjate dob, koristite `prompt` funkciju kako bi korisnik unio dob, sintaksa je sljedeća: `let x = prompt(text, defaultText);`, gdje je `text` poruka koja se prikazuje korisniku, a `defaultText` je opcionalni argument koji predstavlja zadani tekst u polju za unos. Kada to napravite, zamjenite `console.log` sa `alert` funkcijom, sintaksa je sljedeća: `alert(poruka);`, gdje je `poruka` poruka koja se prikazuje korisniku.

EduCoder šifra: `zlatne_godine`

Napomena za EduCoder: U trenutnoj verziji EduCodera v1.4 možete pisati `prompt` i `alert` funkcije u JS dijelu editoru, no možete koristiti samo jednom. Ako želite više, preostale morate zakomentirati. U tom slučaju, preporuka je ovdje ugasiti automatsku evaluaciju i evaluirati kod ručno koristeći `CTRL/CMD + Enter`.

Rezultat:



3.3 Iteracije/Petlje (eng. Iterations/Loops)

Petlje su konstrukti koji omogućuju izvršavanje bloka kôda više puta dok se ne ispuni uvjet definiran logičkim izrazom. U JavaScriptu, kao i u većini programskih jezika, petlje se ostvaruju pomoću ključnih riječi `for` i `while`.

Petlje su korisne kada želimo određeni dio koda izvršavati više puta, svaki put s različitim ulaznim podacima. Na primjer, kada želimo ispisati brojeve od 1 do 10, možemo koristiti petlju umjesto da svaki broj ispisujemo ručno.

```
console.log(1);
console.log(2);
console.log(3);
console.log(4);
console.log(5);
console.log(6);
console.log(7);
console.log(8);
console.log(9);
console.log(10);
```

možemo napisati jednostavno:

```
for (let i = 1; i <= 10; i++) {
    console.log(i);
}
```

Postoji više vrsta `for` petlji u JavaScriptu, ali u pravilu sve rade istu stvar - ponavljaju radnju određeni broj puta (ili nijednom). Koju petlju koristimo zaključujemo ovisno o: ulaznim podacima, početku i kraju petlje, te koracima. Ova `for` petlja slična je `for` petljama u C i Java jezicima.

3.3.1 Klasična `for` petlja

Klasična `for` petlja koristi se kada znamo koliko puta želimo ponoviti blok kôda. Sastoji se od `initialization`, `condition` i `afterthought`. Sintaksa je sljedeća:

```
for (initialization; condition; afterthought) {
    statement; // blok kôda koji se izvršava dok je uvjet = true
}
```

1. `initialization` - izvršava se jednom prije početka petlje, ako postoji. Često inicijalizira varijable koje se koriste u petlji, npr. `let i = 0`, ali sintaksa dozvoljava bilo koji izraz.
2. `condition` izraz se evaluira prije svakog ponavljanja petlje. Ako je `true`, petlja i egzekucija `statement` izraza se nastavlja. Ako je `false`, petlja se prekida.
3. `statement` izraz se izvršava svaki put kada je `condition` = `true`.
4. `afterthought` izraz se izvršava nakon svakog ponavljanja petlje, ako postoji. Često se koristi za inkrementiranje ili dekrementiranje varijabli, npr. `i++`, ali sintaksa dozvoljava bilo koji izraz.

Primjer, želimo ispisati brojeve od `1` do `10`:

```
for (let i = 1; i <= 10; i++) {
    console.log(i); // ispisuje brojeve od 1 do 10 -> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
}
```

Možemo i za nazad:

```
for (let i = 10; i >= 1; i--) {
  console.log(i); // ispisuje brojeve od 10 do 1 -> 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
}
```

Kako možemo upotrijebiti `for` petlju za ispis svih parnih brojeva od `1` do `10`?

```
for (let i = 2; i <= 10; i += 2) {
  console.log(i); // ispisuje parne brojeve od 1 do 10 -> 2, 4, 6, 8, 10
}
```

Kako smo rekli da `initialization` dozvoljava bilo koji izraz pa i prazan, možemo koristiti `for` petlju i na sljedeće načine:

```
let i, j;
for (i = 0, j = 1; i < 10; i++, j++) {
  console.log(`${i} je manji za 1 od ${j}.`);
}
```

Uočite da smo varijable `i` i `j` inicijalizirali izvan petlje, ali smo ih koristili unutar petlje. Međutim, varijable je moguće deklarirati i unutar petlje, u tom slučaju ćemo ključnu riječ `let` koristiti samo jednom.

```
for (let i = 0, j = 1; i < 10; i++, j++) {
  console.log(`${i} je manji za 1 od ${j}.`);
}
```

Možemo pustiti `initialization` prazan, ali moramo imati `;` separator.

```
let i = 0;
for (; i < 10; i++) {
  console.log(i);
}
```

Izostavljanjem nekih od dijelova `for` petlje, možemo dobiti beskonačnu petlju.

Oprez, beskonačne petlje često dovode do crashanja web preglednika ili vaše aplikacije koja izvodi JavaScript kôd. Poželjno je izbjegavati beskonačne petlje.

```
// Navedene petlje će vrlo vjerojatno srušiti vaš web preglednik
for (;;) {
    console.log("Beskonačna petlja!"); // Nema inicijalizacije, uvjeta niti afterthought-a
}

for (let i = 0; ; i++) {
    console.log(i); // Nema uvjeta za prekid petlje
}

for (let i = 0; i < 10; ) {
    console.log(i); // Nema afterthought-a, petlja će beskonačno ispisivati 0
}
```

Primjer 3 - Ispis brojeva od 1 do 100 koji su djeljivi s 3

Izračunajte sumu svih brojeva od 1 do 100 koji su djeljivi s 3. Koristite `for` petlju. Ovaj zadatak zahtjeva korištenje petlje za iteriranje kroz brojeve od 1 do 100, uvjetne izjave za provjeru je li broj djeljiv s 3 i varijablu za praćenje ukupne sume.

Prvo ćemo napisati kôd koji ispisuje sve brojeve od 1 do 100.

```
for (let i = 1; i <= 100; i++) {
    console.log(i);
}
```

Dodat ćemo provjeru je li broj djeljiv s 3. To radimo s operatorom `%` koji vraća ostatak dijeljenja dva broja. Ako je ostatak dijeljenja nekog broja s 3 jednak 0, to znači da je broj djeljiv s 3.

```
for (let i = 1; i <= 100; i++) {
    if (i % 3 === 0) {
        console.log(i);
    }
}
```

Konačno, dodat ćemo varijablu `suma` koja će pohraniti sumu svih brojeva od 1 do 100 koji su djeljivi s 3.

```
let suma = 0;
for (let i = 1; i <= 100; i++) {
    if (i % 3 === 0) {
        console.log(i);
        suma += i;
    }
}
console.log(suma); // ispisuje sumu svih brojeva od 1 do 100 koji su djeljivi s 3 -> 1683
```

3.3.2 `while` petlja

`while` petlja koristi se kada u pravilu ne znamo koliko puta želimo ponoviti blok kôda. Sastoji se od `condition`. Sintaksa je sljedeća:

```

while (condition) {
    statement; // blok kôda koji se izvršava dok je uvjet = true
}

```

Ako je `condition = true`, izvršava se `statement`. Ako je `condition = false`, petlja se prekida. Kao i kod `for` petlje, `statement` izraz se izvršava svaki put kada je `condition = true`.

`condition` se evaluira prije statement izraza, stoga je moguće da se `statement` izraz nikada ne izvrši ako je `condition = false`.

Primjer, sljedeća petlja će iterirati dokle god je `n` manji od 3. Primijetite da u ovom slučaju, `n` mora biti deklariran izvan petlje.

```

let n = 0;
let x = 0;
while (n < 3) {
    n++;
    x += n;
}

```

Sa svakom iteracijom, `n` se inkrementira za `1` i dodaje se na `x`. Kada je `n = 3`, petlja se prekida. Tako da će se izvršiti `3` puta, a `x` i `n` će biti:

1. prolazak: `n = 1, x = 1`
2. prolazak: `n = 2, x = 3`
3. prolazak: `n = 3, x = 6`

Već smo rekli da beskonačne petlje želimo izbjegavati. Moramo osigurati da uvjet u `while` petlji kad tad postane `false`. Ako uvjet nikad ne postane `false`, petlja će se izvršavati beskonačno. Na primjer, sljedeća petlja će se izvršavati beskonačno:

```

while (true) {
    console.log("Beskonačna petlja!");
}

```

Dalje, pogledajmo sljedeći primjer:

```

let i = 0;
while (i < 10) {
    let text = "";
    text += "Broj " + i;
    i++;
    console.log(text); // ispisuje "Broj 0", "Broj 1", "Broj 2", "Broj 3", "Broj 4", "Broj
5", "Broj 6", "Broj 7", "Broj 8", "Broj 9"
}

```

Primijetimo da je varijabla `text` deklarirana unutar petlje. To znači da će se svaki put kada se petlja izvrši, varijabla `text` ponovno inicijalizirati. Kod petlji vrijede ista pravila o dosegu varijabli kao i kod funkcija - varijabla deklarirana unutar petlje neće biti dostupna izvan petlje.

Što ako je `i = 11`? Petlja se neće izvršiti niti jednom, jer je uvjet `i < 10` odmah `false`. Kako bismo ispisali "Broj 10", možemo koristiti varijantu `while` petlje - `do-while` petlju.

3.3.2.1 `do-while` petlja

`do-while` petlja koristi se kada želimo da se blok kôda izvrši barem jednom, a zatim se ponavlja dok je uvjet `= true`. Sastoji se od `condition`. Sintaksa je sljedeća:

```
do {
    statement; // blok kôda koji se izvršava barem jednom, a zatim se ponavlja dok je uvjet
    = true
} while (condition);
```

Prebacimo prethodni primjer u `do-while` petlju. Možemo primijetiti da se `statement` blok izvrši točno jednom, budući da je uvjet `i < 10` odmah `false`.

```
let i = 11;
do {
    let text = "";
    text += "Broj " + i;
    i++;
    console.log(text); // ispisuje "Broj 11"
} while (i < 10);
```

`do-while` petlja ima svoje prednosti, ali se u praksi koristi rjeđe od `for` i `while` petlji.

3.3.3 Prekidanje petlji - `break` | `continue`

Kako bismo "naglo" prekinuli izvršavanje petlje, koristimo ključnu riječ `break`. Kada se `break` naredba izvrši, petlja se prekida i izvršavanje se nastavlja s prvim redom kôda nakon petlje. Na primjer, želimo prekinuti petlju kada dođemo do broja `15` u petlji koja ispisuje brojeve od `1` do `100`.

```
for (let i = 1; i <= 100; i++) {
    if (i === 15) {
        break; // Prekida petlju kada je i = 15, dakle neće se ispisati brojevi od 15 do 100
    }
    console.log(i); // ispisuje brojeve od 1 do 14
}
```

Kako bismo preskočili trenutnu iteraciju petlje, koristimo ključnu riječ `continue`. Kada se `continue` naredba izvrši, trenutna iteracija petlje se prekida i izvršavanje se nastavlja s idućom iteracijom petlje. Na primjer, želimo ispisati sve brojeve od `1` do `100` osim brojeva koji su djeljivi s `3`.

```
for (let i = 1; i <= 100; i++) {
    if (i % 3 === 0) {
        continue; // Preskače trenutnu iteraciju petlje kada je i djeljiv s 3
    }
    console.log(i); // ispisuje sve brojeve od 1 do 100 osim brojeva koji su djeljivi s 3
}
```

`break` i `continue` naredbe možemo koristiti kod svih vrsta petlji - `for`, `while` i `do-while`.

`break` naredbu koristimo i unutar `switch` selekcija kako bi prekinuli njeno izvršavanje nakon ulaska u određeni `case` blok, međutim `continue` naredbu ne koristimo.

3.3.4 Petlje nad nizom znakova (eng. *String*)

Do sad smo koristili petlje za iteriranje kroz brojeve, ali možemo koristiti petlje i za iteriranje kroz nizove znakova. Na primjer, možemo ispisati svaki znak u nizu znakova. Kako bismo to postigli, koristimo `for` petlju i svojstvo `length` niza znakova koje nam govori koliko znakova niz sadrži. Kao i u C jezicima, indeksi znakova u nizu znakova počinju od `0` i idu do `length - 1`, a dohvaćamo ih koristeći operator `[]`.

```
let grad = "Pula";
for (let i = 0; i < grad.length; i++) {
    console.log(grad[i]); // ispisuje svaki znak u nizu znakova -> P, u, l, a
}
```

Idemo upotrijebiti sve znanje o petljama, selekcijama i funkcijama kako bismo napisali funkciju koja će zbrojiti ponavljanja određenog znaka u nizu znakova. Funkcija `brojPonavljanjaZnaka()` prima dva argumenta - niz znakova `niz` i znak `znak`. Funkcija vraća broj ponavljanja znaka `znak` u nizu znakova `niz`.

```
function brojPonavljanjaZnaka(niz, znak) {
    let brojac = 0;
    for (let i = 0; i < niz.length; i++) {
        if (niz[i] === znak) {
            // Provjerava je li trenutni znak u nizu znakova jednak znaku koji tražimo
            brojac++;
        }
    }
    return brojac;
}
console.log(brojPonavljanjaZnaka("Pula", "a")); // ispisuje broj ponavljanja znaka `a` u
nizu znakova Pula -> 1
console.log(brojPonavljanjaZnaka("JavaScript", "a")); // ispisuje broj ponavljanja znaka
`a` u nizu znakova JavaScript -> 2
console.log(brojPonavljanjaZnaka("JavaScript", "z")); // ispisuje broj ponavljanja znaka
`z` u nizu znakova JavaScript -> 0
```

3.3.5 Ugniježđene petlje

Ugniježđene petlje koriste se kada želimo iterirati kroz više dimenzija podataka. Na primjer, kada želimo ispisati sve parove `(i, j)` brojeva u rasponu od `1` do `3`:

```

for (let i = 1; i <= 3; i++) {
  for (let j = 1; j <= 3; j++) {
    console.log(i, j); // ispisuje sve kombinacije parova brojeva od 1 do 3 -> 1 1, 1 2, 1
3, 2 1, 2 2, 2 3, 3 1, 3 2, 3 3
  }
}

```

Kombinirati i ugnijezditi i različite vrste petlji, na primjer, `for` i `while` petlje:

```

let i = 1;
while (i <= 3) {
  for (let j = 1; j <= 3; j++) {
    console.log(i, j); // ispisuje sve kombinacije parova brojeva od 1 do 3 -> 1 1, 1 2, 1
3, 2 1, 2 2, 2 3, 3 1, 3 2, 3 3
  }
  i++;
}

```

`break` i `continue` naredbe u ugniježđenim petljama ponašaju se kao i kod jednostavnih petlji - prekidaju petlju ili preskaču trenutnu iteraciju petlje u kojoj se izvršavaju.

```

for (let i = 1; i <= 3; i++) {
  for (let j = 1; j <= 3; j++) {
    if (i === 2 && j === 2) {
      continue; // Preskače iteraciju gdje je i = 2 i j = 2
    } else {
      console.log(i, j); // ispisuje sve kombinacije parova brojeva od 1 do 3 osim 2 2 ->
1 1, 1 2, 1 3, 2 1, 2 3, 3 1, 3 2, 3 3
    }
  }
}

```

Primjer 4 - Ispis tablice množenja

Primjenjujući ugniježđene petlje možemo jednostavno ispisati tablicu množenja. U ovom primjeru implementirat ćemo funkciju za ispis tablice množenja za brojeve od 1 do 10. Funkcija će ispisati sve kombinacije brojeva od 1 do 10 i njihovih umnožaka.

Prvo definirajmo funkciju `tablicaMnozenja()` i unutar nje `for` petlju koja prolazi kroz brojeve od 1 do 10.

```

function tablicaMnozenja() {
  for (let i = 1; i <= 10; i++) {
    console.log(i);
  }
}
tablicaMnozenja();

```

Dalje, želimo svaki broj `i` pomnožiti s brojevima od `1` do `10`. To ćemo jednostavno postići ugniježđenom `for` petljom.

```
function tablicaMnozenja() {
    for (let i = 1; i <= 10; i++) {
        for (let j = 1; j <= 10; j++) {
            console.log(i, j, i * j); // ispisuje sve kombinacije brojeva od 1 do 10 i njihove umnoške
        }
    }
    tablicaMnozenja();
}
```

Kako bismo dobili tablicu, možemo dodati i formatiranje ispisa. Na primjer, možemo koristiti tabulator `\t` kako bi razdvojili brojeve za veličinu jednog taba.

U varijablu `red` spremamo sve umnoške brojeva `i` i `j` od `1` do `10`, odvajamo ih tabulatorom, a zatim ispisujemo napunjени `red` u vanjskoj petlji.

Rješenje:

```
function tablicaMnozenja() {
    for (let i = 1; i <= 10; i++) {
        let red = "";
        for (let j = 1; j <= 10; j++) {
            red += i * j + "\t";
        }
        console.log(red);
    }
    tablicaMnozenja();
}
```

Vježba 5

Napišite program koji će ispisati sve brojeve od `1` do `100`. Za brojeve koji su djeljivi s `3` umjesto broja ispišite `Fizz`, za brojeve koji su djeljivi s `5` ispišite `Buzz`, dok za brojeve koji su djeljivi i s `3` i s `5` ispišite `FizzBuzz`. Ne ispisujte svaku vrijednost koristeći `console.log()`, već pohranjujte vrijednosti u varijablu `output` i na kraju ispišite niz koristeći `console.log(output)`. Nakon svake vrijednosti dodajte zarez i razmak `(,)`, osim nakon posljednje vrijednosti, nakon nje dodajte `i kraj!`.

EduCoder šifra: `fizz_buzz`

Rezultat:

```
1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, Buzz, 11,      script.js:18
Fizz, 13, 14, FizzBuzz, 16, 17, Fizz, 19, Buzz, Fizz, 22, 23, Fizz,
Buzz, 26, Fizz, 28, 29, FizzBuzz, 31, 32, Fizz, 34, Buzz, Fizz, 37,
38, Fizz, Buzz, 41, Fizz, 43, 44, FizzBuzz, 46, 47, Fizz, 49, Buzz,
Fizz, 52, 53, Fizz, Buzz, 56, Fizz, 58, 59, FizzBuzz, 61, 62, Fizz,
64, Buzz, Fizz, 67, 68, Fizz, Buzz, 71, Fizz, 73, 74, FizzBuzz, 76,
77, Fizz, 79, Buzz, Fizz, 82, 83, Fizz, Buzz, 86, Fizz, 88, 89,
FizzBuzz, 91, 92, Fizz, 94, Buzz, Fizz, 97, 98, Fizz, Buzz i kraj!
```

Vježba 6

Napišite funkciju koja prima jedan argument `godina` i provjerava je li godina prijestupna ili nije. Prema Gregorijanskom kalendaru, godina je prijestupna ako:

- je djeljiva s 4, ali nije djeljiva s 100
- ako je djeljiva s 100, mora biti i s 400

Na primjer, godine `1700`, `1800` i `1900` nisu prijestupne, ali godina `2000` jest.

EduCoder šifra: `svake_prijestupne`

Rezultat:

```
console.log(leapyear(2016)); // true
console.log(leapyear(2000)); // true
console.log(leapyear(1700)); // false
console.log(leapyear(1800)); // false
console.log(leapyear(100)); // false
```

3.4 Rekurzija (eng. Recursion)

Rekurzija je proces kada funkcija poziva samu sebe. Rekursivne funkcije koriste se kada je problem koji rješavamo moguće podijeliti na manje probleme iste vrste. Rekursivne funkcije koriste se za rješavanje problema koji se mogu svesti na manje probleme iste vrste, kao što su problemi vezani uz matematičke operacije, npr. faktorijela, Fibonaccijev niz, Tower of Hanoi, itd.

Rekursivne funkcije imaju dvije komponente: bazni slučaj i rekursivni slučaj. Bazni slučaj je uvjet koji prekida rekursiju, a rekursivni slučaj je uvjet koji poziva samu funkciju za rješavanje manjeg problema iste vrste.

Primjer rekursivne funkcije za izračun faktorijele broja `n`. Faktorijel broja `n` označava se s `n!` i definira je kao umnožak svih pozitivnih cijelih brojeva manjih ili jednakih `n`. Na primjer, faktorijel broja `5` označava se kao `5!` i iznosi `5 * 4 * 3 * 2 * 1 = 120`.

```
function faktorijel(n) {
    if (n === 0) {
        return 1; // Bazni slučaj, prekida rekursiju
    } else {
        return n * faktorijel(n - 1); // Rekursivni slučaj, poziva samu funkciju za rješavanje
        manjeg problema iste vrste
    }
}
```

Rekurzija koristi stog memorije za pohranu svakog poziva funkcije. Ako se rekurzija ne prekine, može doći do prekoračenja stoga memorije i do rušenja aplikacije. Zato je važno osigurati da rekurzija ima bazni slučaj koji prekida rekursiju.

Kako izgleda poziv funkcije `faktorijel(5)`?

1. Početni poziv - `faktorijel(5)`
- bazni slučaj: `5` nije `0`, stoga se izvršava rekursivni slučaj

- rekurzivni slučaj: `5 * faktorijel(4)`, dakle mora se izračunati `faktorijel(4)`

2. Poziv - `faktorijel(4)`

- bazni slučaj: `4` nije `0`, stoga se izvršava rekurzivni slučaj
- rekurzivni slučaj: `4 * faktorijel(3)`, dakle mora se izračunati `faktorijel(3)`

3. Poziv - `faktorijel(3)`

- bazni slučaj: `3` nije `0`, stoga se izvršava rekurzivni slučaj
- rekurzivni slučaj: `3 * faktorijel(2)`, dakle mora se izračunati `faktorijel(2)`

4. Poziv - `faktorijel(2)`

- bazni slučaj: `2` nije `0`, stoga se izvršava rekurzivni slučaj
- rekurzivni slučaj: `2 * faktorijel(1)`, dakle mora se izračunati `faktorijel(1)`

5. Poziv - `faktorijel(1)`

- bazni slučaj: `1` nije `0`, stoga se izvršava rekurzivni slučaj
- rekurzivni slučaj: `1 * faktorijel(0)`, dakle mora se izračunati `faktorijel(0)`

6. Poziv - `faktorijel(0)`

- bazni slučaj: `0` je `0`, stoga se rekurzija prekida i vraća se `1`

7. Vraćanje vrijednosti - `faktorijel(0)` vraća `1` u poziv `faktorijel(1)`

8. Vraćanje vrijednosti - `faktorijel(1)` vraća `1 * 1 = 1` u poziv `faktorijel(2)`

9. Vraćanje vrijednosti - `faktorijel(2)` vraća `2 * 1 = 2` u poziv `faktorijel(3)`

10. Vraćanje vrijednosti - `faktorijel(3)` vraća `3 * 2 = 6` u poziv `faktorijel(4)`

11. Vraćanje vrijednosti - `faktorijel(4)` vraća `4 * 6 = 24` u poziv `faktorijel(5)`

12. Vraćanje vrijednosti - `faktorijel(5)` vraća `5 * 24 = 120`

Konačno, rezultat poziva `faktorijel(5)` je `120`.

Rekurzija nije uvijek najbolje rješenje za rješavanje problema. Rekurzivne funkcije mogu biti teže za razumjeti i održavati, a mogu dovesti i do prekoračenja stoga memorije. U praksi, rekurzija se koristi kada je problem koji rješavamo mogu se svesti na manje probleme iste vrste, a rekurzivno rješenje je jednostavnije i čitljivije od iterativnog rješenja.

3.5 Primjer 5 - Validacija forme

Recimo da imamo web formu koja sadrži polja za unos imena, prezimena, e-maila i lozinke. Želimo provjeriti jesu li sva polja ispravno popunjena prije nego što se forma pošalje na server. Kako možemo to postići koristeći JavaScript?

Prvo, deklarirat ćemo 4 varijable koje će pohraniti vrijednosti unesene u polja forme.

```
let ime;
let prezime;
let email;
let lozinka;
```

Dalje, deklarirat ćemo funkciju `validirajFormu(ime, prezime, email, lozinka)` koja će provjeriti jesu li sva polja ispravno popunjena. U praksi, funkcija će se pozivati kada korisnik klikne na gumb za slanje forme. Ako funkcija vrati `true`, podaci u formi će se poslati na server, a ako vrati `false`, podaci neće biti poslani i korisnika će se na neki način obavijestiti.

Idemo prvo dodati provjere da su sva polja popunjena. Ako nisu, funkcija vraća `false` i obavještava korisnika koristeći `alert()` funkciju.

```
function validirajFormu(ime, prezime, email, lozinka) {
    if (ime === "" || prezime === "" || email === "" || lozinka === "") {
        alert("Molimo da popunite sva polja forme!");
        return false;
    }
    return true;
}
```

Dodajmo našim varijablama proizvoljne vrijednosti:

```
ime = "Sanja";
prezime = "Sanjić";
email = "sanjasanjic@gmail.com";
lozinka = "123456";

console.log(validirajFormu(ime, prezime, email, lozinka)); // ispisuje true
```

Sada ćemo dodati provjeru da lozinka mora sadržavati barem 6 znakova.

```
function validirajFormu(ime, prezime, email, lozinka) {
    if (ime === "" || prezime === "" || email === "" || lozinka === "") {
        alert("Molimo da popunite sva polja forme!");
        return false;
    }
    if (lozinka.length < 6) {
        alert("Lozinka mora biti dugačka barem 6 znakova!");
        return false;
    }
    return true;
}
```

U redu, što ako korisnik za ime i prezime unese brojeve? Dodajmo provjeru da ime i prezime sadrže samo slova.

Unutar naše funkcije `validirajFormu()` dodajmo pomoćnu funkciju `containsNumber()` koja će provjeriti sadrži li niz znakova brojeve.

Funkcija `containsNumber()` radi na način da prolazi kroz svaki znak niza i provjerava je li znak broj. Ako je, vraća `true`, inače vraća `false`.

JavaScript pokušava pretvoriti znakove u brojeve kada koristimo operator `>=` i `<=`, stoga to možemo iskoristiti za usporedbu znakova s brojevima.

Ako znak (`string`) predstavlja broj (0 - 9), JavaScript ga uspješno pretvara u broj (`number`) i provodi aritmetičku provjeru.

```
function validirajFormu(ime, prezime, email, lozinka) {
    if (ime === "" || prezime === "" || email === "" || lozinka === "") {
        alert("Molimo da popunite sva polja forme!");
        return false;
    }

    if (lozinka.length < 6) {
        alert("Lozinka mora biti dugačka barem 6 znakova!");
        return false;
    }

    function containsNumber(str) {
        for (let i = 0; i < str.length; i++) {
            // Provjerava je li znak broj (0-9)
            if (str[i] >= 0 && str[i] <= 9) {
                return true;
            }
        }
        return false;
    }

    if (containsNumber(ime) || containsNumber(prezime)) {
        alert("Ime i prezime ne smiju sadržavati brojeve!");
        return false;
    }

    return true;
}
```

Sada ako pokušamo pozvati funkciju `validirajFormu(ime, prezime, email, lozinka)` s proizvoljnim argumentima, funkcija će provjeriti jesu li sva polja ispravno popunjena, je li lozinka dugačka barem 6 znakova i sadržavaju li ime i prezime brojeve.

```
ime = "Sanja";
prezime = "Sanji3";
email = "sanjasanjic@gmail.com";
lozinka = "123456";

console.log(validirajFormu(ime, prezime, email, lozinka)); // false
```

U JavaScriptu, znakovi (uključujući i brojeve i slova) se kodiraju koristeći **Unicode** skup znakova. U ASCII i Unicode skupovima znakova, znakovi se prikazuju numeričkim vrijednostima. Primjerice, u **ASCII** skupu, slovo `a` kodira se brojem `97`, a slovo `z` brojem `122`. Brojevi se kodiraju brojevima od `48` do `57`. Dok u **Unicode** skupu, znak `o` kodira se brojem `0030`, a znak `9` brojem `0039`.

Imajući to na umu, možemo dodati novu provjeru za `ime` i `prezime`. Funkciju koja provjerava sadrže li ime i prezime samo niz znakova `[a – z]`.

Na ovaj način ne uzimamo u obzir hrvatska slova: `č`, `ć`, `š`, `đ`, `ž`, `lj`, `nj`, `dž`.

```
function containsOnlyLetters(str) {
    for (let i = 0; i < str.length; i++) {
        let c = str[i];
        // Leksikografska usporedba znakova
        if (!(c >= "a" && c <= "z") && !(c >= "A" && c <= "Z")) {
            return false;
        }
    }
    return true;
}
```

U ovom slučaju, JavaScript će znakove pretvoriti u brojeve prema **Unicode** skupu znakova. Dano rješenje radi zato što znamo da su slova `[a – z]` kodirana brojevima od `97` do `122` i od `65` do `90` u ASCII skupu znakova. Drugim riječima, pohranjena su sekvencijalno!

Zapamtite:

- **containsNumber(str)** : Ovdje su nam operandi u selekciji **znak** i **broj**. JavaScript će u ovom slučaju nastojati pretvoriti znak u broj i provesti usporedbu.
- **containsOnlyLetters(str)** : Ovdje su nam operandi u selekciji **znak** i **znak**. JavaScript će u ovom slučaju usporediti znakove leksikografski, tj. po redoslijedu u ASCII skupu znakova.

Možemo još dodati provjeru je li e-mail ispravno formatiran. Na primjer, e-mail mora sadržavati znak `@` i barem jednu točku nakon znaka `@`.

Radi pojednostavljenja, nećemo provjeravati sadrži li e-mail nedozvoljene znakove ili više znakova `@`.

Naša konačna funkcija `validirajFormu()` izgleda ovako:

```
function validirajFormu(ime, prezime, email, lozinka) {
    if (ime === "" || prezime === "" || email === "" || lozinka === "") {
        alert("Molimo da popunite sva polja forme!");
        return false;
    }

    if (lozinka.length < 6) {
        alert("Lozinka mora biti dugačka barem 6 znakova!");
        return false;
    }

    function containsNumber(str) {
        for (let i = 0; i < str.length; i++) {
            // Provjerava je li znak broj (0-9)
```

```

        if (str[i] >= "0" && str[i] <= "9") {
            return true;
        }
    }
    return false;
}

function containsOnlyLetters(str) {
    for (let i = 0; i < str.length; i++) {
        let c = str[i];
        // Leksikografska usporedba znakova
        if (!(c >= "a" && c <= "z") && !(c >= "A" && c <= "Z")) {
            return false;
        }
    }
    return true;
}

function checkEmail(email) {
    let atFound = false;
    let dotFound = false;
    for (let i = 0; i < email.length; i++) {
        if (email[i] === "@") {
            atFound = true;
        }
        if (atFound && email[i] === ".") {
            dotFound = true;
        }
    }
    return atFound && dotFound;
}

if (containsNumber(ime) || containsNumber(prezime)) {
    alert("Ime i prezime ne smiju sadržavati brojeve!");
    return false;
}

if (!containsOnlyLetters(ime) || !containsOnlyLetters(prezime)) {
    alert("Ime i prezime smiju sadržavati samo slova a-z!");
    return false;
}

if (!checkEmail(email)) {
    alert("Email mora sadržavati @ i najmanje jednu točku (.) nakon @!");
    return false;
}

return true;
}

```

Samostalni zadatak za vježbu 3

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

EduCoder šifra: `ReversePrimeLongest`

1. Napišite funkciju `reverseString` koja prima znakovni niz (string) kao argument i vraća obrnuti string. Na primjer, ako je ulaz `"hello"`, funkcija treba vratiti `"olleh"`. Funkcija mora vratiti `"Not a string!"` ako je ulazni argument različitog tipa od stringa. Funkciju testirajte s argumentima `"hello"`, `"JavaScript"` i `123`.
2. Napišite funkciju `prost_broj` koja prima broj kao argument i vraća `true` ako je broj prost, odnosno `false` ako nije. Broj je prost ako je djeljiv samo s 1 i samim sobom. Funkciju pozovite s argumentima 7, 10 i 13.
3. Nadogradite prethodni zadatak na način da ćete ispisati sve proste brojeve od 1 do 100. Funkciju `prost_broj` pozivajte unutar petlje. Ispis mora izgledati ovako: `"Prosti brojevi od 1 do 100 su: 2, 3, 5, 7, itd."`
4. Napišite funkciju `pronadi_najduzu_rijec()` koja prima rečenicu kao argument i vraća najdužu riječ u rečenici. Rečenicu morate razložiti **koristeći petlju, bez pomoćnih funkcija/metoda!**
 - o Ako se funkciji proslijedi tip podatka koji nije string, funkcija vraća `"Nije rečenica!"`.
 - o Ako je rečenica prazna, funkcija vraća `"Rečenica je prazna!"`.
 - o Ako se rečenica sastoji od samo jedne riječi, funkcija vraća tu riječ.
 - o Ako se rečenica sastoji od više različitih najdužih riječi, funkcija vraća prvu riječ koja je pronađena.

Programiranje u skriptnim jezicima (PJS)

Nositelj: doc. dr. sc. Nikola Tanković

Asistenti:

- Luka Blašković, mag. inf.
- Alesandro Žužić, mag. inf.

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



Fakultet informatike u Puli

[3] Strukture podataka - Objekti i Polja



Strukture podataka su specijalizirani formati podataka namijenjeni efikasnijoj pohrani, organizaciji, dohvatu 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: 2.8.2024.

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 razumjeti 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 `{}`.

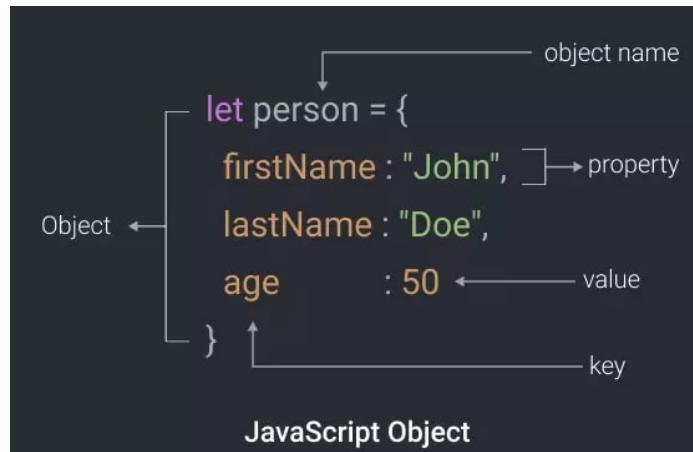
```
console.log(auto); // {}
```

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 svojstava (`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 pristupati 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 pristupati 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
auto 	auto.marka = "Ford" auto.model = "Mustang" auto.godina_proizvodnje = 2020 auto.boja = "Crna"	auto.izracunajStarost()

 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`.

Primijetite 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 km²

```
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 primijetit 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_rodenja`.

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

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

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

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

```
function stvoriKorisnika(ime, prezime, godina_rodjena) {
    const obj = {} // stvori prazan objekt
    obj.ime = ime; // dodaj svojstvo ime
    obj.prezime = prezime; // dodaj svojstvo prezime
    obj.godina_rodjena = godina_rodjena; // dodaj svojstvo godina_rodjena

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

return obj; // vrati objekt
}
```

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

```
const ana = stvoriKorisnika("Ana", "Anić", 1990);
const marko = stvoriKorisnika("Marko", "Marić", 1985);
const petar = stvoriKorisnika("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. substring) 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>	Izdvaja podskup niza znakova i vraća novi niz bez izmjene originalnog niza. Metoda će izdvajati podskup [<code>indexStart, indexEnd</code>], dakle <code>indexEnd</code> neće biti uključen. Ako je <code>indexStart > indexEnd</code> , <code>substring()</code> će ih zamjeniti. Ako su indeksi negativni brojevi, interpretirat će se kao <code>0</code> .	<code>string.substring(indexStart, indexEnd)</code>	<code>let novi = 'Novigrad'.substring(0, 4)</code>	<code>novi === 'Novi'</code>
<code>slice()</code>	Izdvaja podskup niza znakova i vraća novi niz bez izmjene originalnog niza. Metoda će izdvajati podskup [<code>indexStart, indexEnd</code>], dakле <code>indexEnd</code> neće biti uključen. Za razliku od <code>substring()</code> metode, ako je <code>indexStart > 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>	<code>let noviNiz = 'Novigrad'.slice(-4)</code>	<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 pronađe, 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 spremu u polje. Vraća polje podnizova bez izmjene originalnog znakovnog niza. Metoda ima i optionalni separator limit 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: 'Godina je 2024 i mjesec je 3'. <code>'Godina je 2024 i mjesec je 3'.match(/\d+/g)</code> . \d - broj [0-9], \d+ - traži poklapanje jednog ili više broja g - 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, 0 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 очekuje 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 optionalni <code>position</code> argument koji započinje pretragu na određenoj poziciji, 0 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 jednostruktih 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 pisačih strojeva, teleprintera 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 komplificira 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" - konkatenacija stringova
```

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

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>	Zaokružuje broj na zadani broj (<code>digits</code>) decimalnih mesta. Vraća string (zbog decimalne točke).	<code>number.toFixed(digits)</code>	<code>(5.56789).toFixed(2)</code>	<code>"5.57"</code>
<code>toPrecision()</code>	Za dati broj metoda vraća njegovu string reprezentaciju na zadani broj značajnih znamenki : <code>precision</code> parametar mora biti između 1 i 100.	<code>number.toPrecision(2)</code>	<code>(5.123456).toPrecision(2)</code>	<code>"5.1"</code>
<code>toString()</code>	Vraća string reprezentaciju broja. Opcionálni <code>radix</code> parametar, može biti između 2 i 36 i specificira bazu koja se koristi za reprezentaciju broja. Default je 10 (dekadski zapis)	<code>number.toString(radix=10)</code>	<code>(123).toString(); (100).toString(2)</code>	<code>"123"; "1100100"</code>
<code>parseInt()</code>	Metoda pretvara dati string u cijelobrojni ekvivalent. Kao i kod <code>toString()</code> , sadrži opcionálni <code>radix</code> parametar.	<code>Number.parseInt(string, radix)</code>	<code>parseInt("10.456"); parseInt("40 years")</code>	<code>10 ; 40</code>
<code>parseFloat()</code>	Metoda pretvara dati 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 boolean 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 boolean 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 π (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 2	0.6931471805599453
<code>Math.LN10</code>	Vraća vrijednost prirodnog (\ln) logaritma broja 10	2.302585092994046

Metoda	Objašnjenje	Sintaksa	Primjer	Output
<code>Math.abs(x)</code>	Vraća apsolutnu vrijednost broja x .	<code>Math.abs(x)</code>	<code>Math.abs(-4.5)</code>	4.5
<code>Math.ceil(x)</code>	Metoda zaokružuje i vraća najmanji cijeli broj veći ili jednak zadanim (x) broju.	<code>Math.ceil(x)</code>	<code>Math.ceil(4.3)</code>	5
<code>Math.floor(x)</code>	Metoda zaokružuje prema dolje i vraća najveći cijeli broj manji ili jednak zadanim (x) broju.	<code>Math.floor(x)</code>	<code>Math.floor(4.9)</code>	4
<code>Math.max(x, y)</code>	Vraća veći od dva broja x i y . 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>	10
<code>Math.min(x, y)</code>	Vraća manji od dva broja x i y . 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>	5
<code>Math.pow(x, y)</code>	Vraća rezultat potenciranja broja x na potenciju y .	<code>Math.pow(base, exponent)</code>	<code>Math.pow(2, 3)</code>	8
<code>Math.sqrt(x)</code>	Računa kvadratni korijen broja x .	<code>Math.sqrt(x)</code>	<code>Math.sqrt(9)</code>	3
<code>Math.round(x)</code>	Zaokružuje broj x na najbliži cijeli broj.	<code>Math.round(x)</code>	<code>Math.round(4.3)</code>	4
<code>Math.random()</code>	Generira pseudoslučajan broj između 0 i 1. 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 0 i 1)
<code>Math.log(x)</code>	Računa prirodnji logaritam (po bazi e) broja x .	<code>Math.log(x)</code>	<code>Math.log(Math.E)</code>	1
<code>Math.exp(x)</code>	Računa e na potenciju x .	<code>Math.exp(x)</code>	<code>Math.exp(1)</code>	2.718281828459045
<code>Math.sin(x)</code>	Računa sinus broja x (u radijanima).	<code>Math.sin(x)</code>	<code>Math.sin(Math.PI / 2)</code>	1
<code>Math.cos(x)</code>	Računa kosinus broja x (u radijanima).	<code>Math.cos(x)</code>	<code>Math.cos(Math.PI)</code>	-1
<code>Math.tan(x)</code>	Računa tangens broja x (u radijanima).	<code>Math.tan(x)</code>	<code>Math.tan(Math.PI / 4)</code>	1

Vježba 3

EduCoder šifra: matematika

- 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
```

- 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
```

- 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 trenutnim datumom i vremenom	<code>const d = new Date();</code>
<code>new Date(date string)</code>	stvara novi <code>Date</code> objekt iz date stringa	<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. JavaScript broji mjesecu od 0! 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 brojem milisekundi od 1. siječnja 1970. odnosno unix oznakom	<code>const d = new Date(1708436235000);</code>

Primijetite 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 dan u mjesecu kao broj (1-31).	<code>Date.getDate()</code>	<code>const rodendan = new Date("April 13, 2000");</code> <code>rodendan.getDate() == 13</code>	13
<code>getDay()</code>	za dani datum vraća dan u tjednu (0 za nedjelju, 1 za ponедjeljak, itd.).	<code>Date.getDay()</code>	<code>const rodendan = new Date("April 13, 2000");</code> <code>rodendan.getDay() == 4</code>	4
	Za dani datum vraća			

<code>getFullYear()</code>	godinu. Izbjegavajte 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 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 optionalne 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 optionalne 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 automatski 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 (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 konkatenirati 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 razmaka. 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 razmaka.

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 postojeće primitivne tipove, poput `String` i `Number` objekata.

Rekli smo da nema smisla komplikirati 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

1. Napišite konstruktor za objekt `Grad` koji prima 3 argumenta: `ime`, `brojStanovnika` i `drzava`. Konstruktor treba stvoriti objekt s tim svojstvima.
 - o Napišite metodu `ispisi()` koja ispisuje informacije o gradu u formatu: `Ime: [ime], Broj stanovnika: [brojStanovnika], Država: [drzava]`.
 - o U konstruktor dodajte metodu `azurirajBrojStanovnika()`.
 - o Kada to napravite, dodajte konstruktoru svojstvo `velicina`
 - o ažurirajte metodu `ispisi()` da ispisuje i veličinu grada.
2. Napišite funkciju `izbacisamoglasnike()` koja prima rečenicu kao argument i vraća novu rečenicu bez samoglasnika. Za implementaciju koristite metode iz `String` objekta.
3. 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.
 - o Ako je proslijeđeni argument `broj` već cijeli, funkcija vraća string `Broj je već cijeli!`.
 - o 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."`.
4. Napišite funkciju `daniodPocetkaGodine()` koja vraća koliko je dana prošlo od početka godine do trenutnog datuma. Za implementaciju koristite metode iz `Date` objekta.
 - o Dodajte poseban uvjet ako je trenutni datum 1. siječnja, funkcija onda vraća `Danas je 1. siječnja!`.
5. 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:
 - o `promjeniEmail()` - 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!`.
 - o `promjeniLozinku()` - 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)

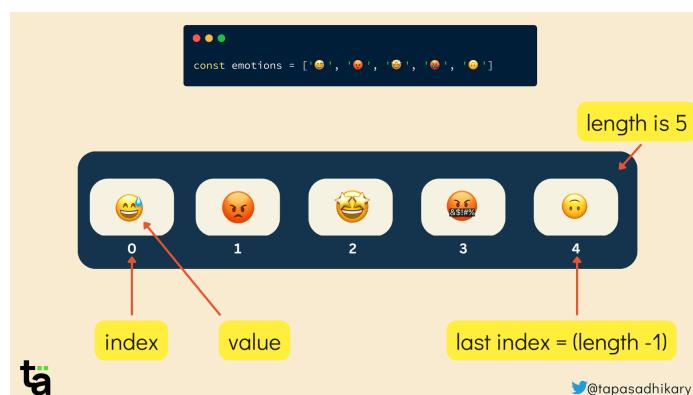
Polja (eng. **Arrays**) su strukture podataka koje, kao i u drugim programskim jezicima, omogućuju pohranu kolekcije podataka pod varijabljom jednog naziva.

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"] //Primijetite uglate zgrade,
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 najđemo 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 izbrisano, nema više praznog mesta
```

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 pronaleta 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"
```

Primijetite 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 što to napravite, iterirajte kroz polje još jednom i ispravite negativne ocjene na `2`. Sumirajte sve ocjene i izračunajte prosjek. Ispišite `novo 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 izmijeniti 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) { // Primijetite 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 pro 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 istovrsnih 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_rodenja: 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 primati 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>	<code>const fruits = ["Banana", "Orange", "Apple", "Mango"]; fruits.size == 4</code>	4
<code>toString()</code>	Vraća polje u string obliku, gdje su vrijednosti odvojene zarezima.	<code>Array.toString()</code>	<code>const fruits = ["Banana", "Orange", "Apple", "Mango"]; fruits.toString();</code>	'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>	<code>const fruits = ["Banana", "Orange", "Apple", "Mango"]; fruits.at(2) == "Apple"; fruits.at(-1) == "Mango"</code>	"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>	<code>const elements = ['Fire', 'Air', 'Water']; elements.join('-') == "Fire-Air-Water"</code>	"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>	<code>const elements = ['Fire', 'Air', 'Water']; let count = elements.push("Earth")</code>	<code>elements = ["Fire", "Air", "Water", "Earth"] ; count == 4</code>
<code>pop()</code>	Metoda briše zadnji element u polju, a kao povratnu vrijednost	<code>Array.pop()</code>	<code>const plants = ['broccoli', 'cauliflower', 'cabbage', 'kale']</code>	tomato

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

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. Implementirat 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 ukupnoObjave = 100; //Ukupan broj objava

for (let i = 1; i <= ukupnoObjave; i++) { // Dodajemo 100 dummy objave 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
indexOf()	Metoda pretražuje polje za dani <code>searchElement</code> i vraća indeks prvog pronađenog elementa, ili <code>-1</code> ako element nije pronađen. Prima i optionalni parametar <code>fromIndex</code> preko kojeg se može definirati od kojeg indeksa da se pretražuje.	<code>Array.indexOf(searchElement, fromIndex)</code>	<code>const beasts = ['ant', 'bison', 'camel', 'duck', 'bison']; beasts.indexOf('bison') == 1</code>	1
lastIndexOf()	Metoda pretražuje polje za dani <code>searchElement</code> i vraća indeks zadnjeg pronađenog elementa, ili <code>-1</code> ako element nije pronađen. Prima i optionalni parametar <code>fromIndex</code> preko kojeg se može definirati od kojeg indeksa da se	<code>Array.lastIndexOf(searchElement, fromIndex)</code>	<code>const animals = ['Elephant', 'Tiger', 'Penguin', 'Elephant']; animals.lastIndexOf('Elephant') == 3</code>	3

	pretražuje unazad .			
includes()	Slično kao kod <code>String.includes()</code> metode, ova metoda provjerava sadrži li polje traženu vrijednost. Vraća boolean vrijednost ovisno o sadržavanju. Opcionalni <code>fromIndex</code> parametar koji definira od kojeg indeksa se pretražuje.	<code>Array.includes(searchElement, fromIndex)</code>	<code>const array1 = [1, 2, 3]; array1.includes(2) == true; const pets = ['cat', 'dog', 'bat']; pets.includes('cat', 1) == false</code>	true; false
find()	Vraća vrijednost 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>	<code>const numbers = [4, 9, 16, 25, 29]; let first = numbers.find(function(value) {return value > 18;});</code>	first == 25
findIndex()	Vraća indeks 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>	<code>const numbers = [4, 9, 16, 25, 29]; let firstIndex = numbers.findIndex(function(value) {return value > 18;});</code>	firstIndex == 3
findLast()	Vraća vrijednost prvog elementa u polju iteriranjem unazad 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>	<code>const array1 = [5, 12, 50, 130, 44]; array1.findLast(function(value) {return value > 45;})</code>	130
findLastIndex()	Vraća indeks prvog elementa u polju iteriranjem unazad 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>	<code>const array1 = [5, 12, 50, 130, 44]; array1.findLastIndex(function(value) {return value > 45;})</code>	3

Kada koristit 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, "markic90");
let korisnik3 = new Korisnik("Ana", "Anić", 1985, "anic");
let korisnik4 = new Korisnik("Petra", "Petrović", 1970, "petrovic70");

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_rodenja: 1995, korisničko_ime: "iivic", predstaviSe: f}  
// 1: Korisnik {ime: "Ana", prezime: "Anić",  
godina_rodenja: 1985, korisničko_ime: "aanic", predstaviSe: f}  
// 2: Korisnik {ime: "Petra", prezime:  
"Petrović", godina_rodenja: 1970, korisničko_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 duplike 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"]

```

2. 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
            .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_rodjena`, `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 primati polje `osobe`, izračunati prosječnu visinu svih osoba i ispisati je u konzolu.
2. Napravite novi objekt `Sportas` koji će se sastojati od svojstava: `ime`, `prezime`, `godina_rodjena`, `spol`, `visina`, `tezina`, `sport`, `klub` i `broj_dresa`. Napravite nekoliko sportaša i dodajte ih u polje `sportasi`.
 - o implementirajte globalnu funkciju `prosjecnaTezina(sportasi)` koja će kao argument primati polje `sportasi`, izračunati prosječnu težinu svih sportaša i ispisati je u konzolu.
 - o implementirajte globalnu funkciju `najteziSportas(sportasi)` koja će pronaći i vratiti objekt najtežeg sportaša.
 - o 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.
 - o 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.
 - o dodajte metodu `nastupiSportasa()` koja će ispisati sve nastupe sportaša u konzolu u sljedećem formatu: `"Nastup 1: ${nastup1}, Nastup 2: ${nastup2}, ... Nastup N: ${nastupN}"`.
 - o dodajte metodu `dohvatizadnjaDvaNastupa()` koja će ispisati zadnja dva nastupa sportaša u konzolu. Koristite metodu `slice()`.
 - o implementirajte globalnu funkciju `izbrisisiSvimaPrviNastup(sportasi)` koja će obrisati svim sportašima prvi nastup. Koristite metodu `shift()`.
 - o 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 kôd ovdje
}
console.log(gcd_array([3, 6, 9, 12])); // Output: 3
console.log(gcd_array([10, 20, 30, 40])); // Output: 10
```

Programiranje u skriptnim jezicima (PJS)

Nositelj: doc. dr. sc. Nikola Tanković

Asistenti:

- Luka Blašković, mag. inf.
- Alesandro Žužić, mag. inf.

Ustanova: Sveučilište Jurja Dabrike u Puli, Fakultet informatike u Puli



Fakultet informatike u Puli

[4] Ugniježđene strukture i Napredne funkcije

#4
JS

"Baratanje" ugniježđenim strukturama (**eng. nested structures**) je jedna od ključnih vještina u programiranju. Bilo to u obliku ugniježđenih petlji, objekata, funkcija, ili polja. Dohvat podataka s različitih API-eva, obrada podataka, ili pisanje algoritama, sve to zahtijeva dobro poznavanje ugniježđenih struktura. U ovoj skripti naučit ćete pisati ugniježđene strukture u JavaScriptu i naučiti koristiti napredne funkcije i operatore za jednostavniji rad s njima.

Posljednje ažurirano: 2.8.2024.

Sadržaj

- [Programiranje u skriptnim jezicima \(PJS\)](#)
- [4. Ugniježđene strukture i Napredne funkcije](#)
 - [Sadržaj](#)
- [1. Uvod u ugniježđene strukture](#)
- [2. Ugniježđene strukture](#)
 - [2.1 Objekti unutar objekata](#)
 - [2.1.1 Manipulacije podataka unutar ugniježđenih objekata](#)
 - [Izmjena podataka unutar ugniježđenih objekata](#)
 - [Dodavanje novih podataka unutar ugniježđenih objekata](#)
 - [Brisanje podataka unutar ugniježđenih objekata](#)
 - [2.2 Polja unutar objekata](#)
 - [2.2.1 Iteracija kroz polje unutar objekata](#)
 - [2.3 Objekti unutar polja](#)
 - [Vježba 1](#)

- [Vježba 2](#)
- [2.4 Polja unutar polja](#)
 - [2.4.1 Iteracije kroz više dimenzija](#)
 - [2.4.2 Stvaranje višedimenzionalnih polja pomoću `Array` konstruktora](#)
 - [Vježba 3](#)
- [2.5 Sažetak ugniježđenih struktura](#)
 - [Vježba 4](#)
- [Samostalni zadatak za vježbu 6](#)
- [3. Napredne funkcije](#)
 - [3.1 Callback funkcije](#)
 - [3.1.1 Primjer callback funkcije](#)
 - [3.1.2 Osnovna podjela `callback` funkcija](#)
 - [1. Globalno definirana `callback` funkcija](#)
 - [2. Anonimna `callback` funkcija](#)
 - [3.2 Callback funkcije s poljima](#)
 - [3.2.1 Metoda `find\(callbackFn\)`](#)
 - [3.2.2 Metoda `forEach\(callbackFn\)`](#)
 - [3.2.3 Metoda `filter\(callbackFn\)`](#)
 - [Primjer 1: Tražilica !\[\]\(abdc8ebe3f6f78367052141d0514b0e4_img.jpg\)](#)
 - [Vježba 5](#)
 - [Vježba 6](#)
 - [3.3 Arrow funkcije \(`=>`\)](#)
 - [3.3.1 Funkcijski izrazi i deklaracije](#)
 - [3.3.2 Sintaksa `arrow` funkcija](#)
 - [3.3.3 Primjeri `arrow` funkcija](#)
 - [3.3.4 `arrow` funkcije kao callback funkcije](#)
 - [Primjer 2: Pronađi let !\[\]\(cbd982c1a8b4e7c5153185849951101e_img.jpg\)](#)
 - [Vježba 7](#)
 - [Vježba 8](#)
 - [3.3.5 `arrow` funkcije i `this` kontekst](#)
 - [3.4 Napredne metode `Array` objekta](#)
 - [3.4.1 Metoda `map\(\)`](#)
 - [3.4.2 Metoda `some\(\)`](#)
 - [3.4.3 Metoda `every\(\)`](#)
 - [3.4.4 Metoda `sort\(\)`](#)

- [3.4.5 Metoda `reduce\(\)`](#)

- [3.4.6 Kada koristiti koju metodu?](#)

- [Vježba 9](#)

- [Vježba 10](#)

- [Samostalni zadatak za vježbu 7](#)

1. Uvod u ugniježđene strukture

Do sad smo naučili da možemo ugniježđivati selekcije i petlje, pa i funkcije. U JavaScriptu međutim, kada pričamo o ugniježđenim strukturama, mislimo na razne složene strukture koje se pretežito sastoje od ugniježđenih objekata i polja. Prema tome, ugniježđene strukture možemo podijeliti u **4 kategorije**:

1. Objekti unutar objekata `{()}`

2. Polja unutar objekata `{[]}`

3. Objekti unutar polja `[{}]`

4. Polja unutar polja `[[]]`

Prije nego odradimo navedene kategorije, prisjetimo se ugniježđenih selekcija, petlji i funkcija.

Primjer ugniježđene selekcije:

```
if (zaposlen) {  
    if (placa > 1500) {  
        console.log("Kreditno sposoban!");  
    } else {  
        console.log("Ne diži kredit!");  
    }  
} else {  
    console.log("Ne diži kredit nikako!");  
}
```

Primjer ugniježđene petlje:

```
for (let i = 0; i < 3; i++) {  
    for (let j = 0; j < 6; j++) {  
        console.log(`i je ${i}, a j je ${j}`);  
    }  
}
```

Rekli smo da možemo ugniježđivati i funkcije. Možda to nije nešto što ćemo često raditi, ali je moguće. Evo primjera:

```

function prvaFunkcija() {
    console.log("Pozdrav iz prve funkcije!");
    function drugaFunkcija() {
        console.log("Pozdrav iz druge funkcije!");
    }
    drugaFunkcija();
}
prvaFunkcija();

```

Recimo da želimo pohraniti podatke o korisniku naše aplikacije: `ime`, `prezime`, `adresa` i `kontakt`. Pod adresu želimo pohraniti `ulica`, `grad` i `poštanski broj`. Pod kontakt želimo pohraniti `telefon` i `email`.

Prvo ćemo sve pohraniti u jednostavan objekt **bez** ugniježđenih struktura:

```

let korisnik = {
    ime: "Ivo",
    prezime: "Ivić",
    adresa: "Ulica 123, 52100 Pula",
    kontakt: "0911234567",
    email: "ivo@gmail.com",
};

```

Uočite zašto je ovakav zapis nezgrapan. Kako bi dohvatali `ulicu` moramo koristiti `split` metodu. Isti problem predstavlja `poštanski broj`.

Idemo problem rješiti **ugniježđenim objektima**.

```

let korisnik = {
    ime: "Ivo",
    prezime: "Ivić",
    adresa: {
        ulica: "Ulica 123",
        grad: "Pula",
        postanskiBroj: "52100",
    },
    kontakt: {
        telefon: "0911234567",
        email: "ivo@gmail.com",
    },
};

```

Sada možemo jednostavno dohvatiti `ulicu`, `grad`, `poštanski broj`, `telefon` i `email`, a naš kôd je pregledniji. Na jednaki način kako dohvaćamo atrIBUTE objekata, možemo dohvaćati i atrIBUTE ugniježđenih objekata, koristeći `.` operator.

```

console.log(korisnik.adresa.ulica); // Ispisuje "Ulica 123"
console.log(korisnik.adresa.grad); // Ispisuje "Pula"
console.log(korisnik.adresa.postanskiBroj); // Ispisuje "52100"

console.log(korisnik.kontakt.telefon); // Ispisuje "0911234567"
console.log(korisnik.kontakt.email); // Ispisuje "ivo@gmail.com"

```

2. Ugnježđene strukture

2.1 Objekti unutar objekata

Često ćemo se u programiranju susretati s potrebom za pohranjivanjem složenih podataka i specifikacije nekakve hijerarhijske strukture. Primjerice, kako ćemo pohraniti podatke o korisniku? Korisnik sadrži `ime`, `prezime`, `adresu` i `kontakt`. `Adresa` se sastoji od `ulice`, `grada` i `poštanskog broja`. `Kontakt` se sastoji od `telefona` i `emaila`. Navedeno možemo postići s pomoću ugnježđenih objekata, tj. **objekata unutar objekata**.

Objekte "ugnježđujemo" tako da stvaramo **objekte unutar objekata**, doslovno. Sintaksa je sljedeća:

```

let objekt1 = {
    svojstvo1: vrijednost1,
    svojstvo2: vrijednost2,
    objekt2: {
        svojstvo3: vrijednost3,
        svojstvo4: vrijednost4,
    },
    objekt3: {
        svojstvo5: vrijednost5,
        svojstvo6: vrijednost6,
    },
};

```

Već smo rekli da kod ugnježđenih objekata za dohvaćanje podataka koristimo već poznate operatore `.` ili `[]`. Primjer:

```

console.log(objekt1.svojstvo1); // Ispisuje vrijednost1
console.log(objekt1.objekt2.svojstvo3); // Ispisuje vrijednost3
console.log(objekt1["objekt2"]["svojstvo3"]); // Ispisuje vrijednost3

```

Zamislimo da radimo backend aplikacije. Gotovo uvijek bit će nam potrebna autentifikacija za korisnika, poveznica na bazu podataka te nekakav server koji će služiti kao podloga našoj aplikaciji. Idemo definirati dummy konfiguracijski objekt za našu aplikaciju. Konfiguracijski objekt se često definira kao objekt u koji ćemo definirati neke postavke tj. parametre naše aplikacije. Primjer:

```

let konfiguracija = {
    server: {
        host: "localhost",
        port: 8080,
    }
}

```

```

},
bazaPodataka: {
  url: "mongodb://localhost:27017",
  ime: "mojaBaza",
},
sigurnost: {
  tip: "OAuth2",
  tajna: "tajniKljuc",
},
};

console.log(konfiguracija.server.host); // Ispisuje "localhost"
console.log(konfiguracija.bazaPodataka.url); // Ispisuje "mongodb://localhost:27017"
console.log(konfiguracija.sigurnost.tip); // Ispisuje "OAuth2"

```

Podobjekt može biti definiran i izvan objekta `konfiguracija`:

```

let server = {
  // Podobjekt #1
  host: "localhost",
  port: 8080,
};

let bazaPodataka = {
  //Podobjekt #2
  url: "mongodb://localhost:27017",
  ime: "mojaBaza",
};

let sigurnost = {
  //Podobjekt #3
  tip: "OAuth2",
  tajna: "tajniKljuc",
};

// Glavni objekt
let konfiguracija = {
  server: server, // Podobjekt
  bazaPodataka: bazaPodataka, // Podobjekt
  sigurnost: sigurnost, // Podobjekt
};

```

Što ako ispišemo cijeli objekt `konfiguracija`? Rezultat ispisa će biti cijeli objekt, **uključujući i podobjekte**.

```

console.log(konfiguracija); // Ispisuje: {server: {...}, bazaPodataka: {...}, sigurnost: {...}}

```

Detaljni ispis objekta `konfiguracija`:

```

{
  bazaPodataka: {

```

```

        url: "mongodb://localhost:27017",
        ime: "mojaBaza"
    },
    server: {
        host: "localhost",
        port: 8080
    },
    sigurnost: {
        tip: "OAuth2",
        tajna: "tajniKljuc"
    }
}

```

2.1.1 Manipulacije podataka unutar ugniježđenih objekata

Izmjena podataka unutar ugniježđenih objekata

Kako mijenjati podatke unutar ugniježđenih objekata? Na primjer, kako promijeniti `host` servera u našem objektu `konfiguracija`? Na isti način kako dohvaćamo podatke iz ugniježđenih objekata, koristeći `.` operator ili notaciju uglatih zagrada `[]`.

```

konfiguracija.server.host = "192.168.5.5";
console.log(konfiguracija.server.host); // Ispisuje "192.168.5.5"

```

Možemo koristiti i notaciju uglatih zagrada `[]`:

```

konfiguracija["server"]["host"] = "192.168.5.5";
console.log(konfiguracija["server"]["host"]); // Ispisuje "192.168.5.5"

```

Dodavanje novih podataka unutar ugniježđenih objekata

Recimo da hoćemo dodati `protocol` podatak u naš objekt `server`. To radimo na isti način kao dodavanje novih podataka u obične objekte.

```

konfiguracija.server.protocol = "http";
console.log(konfiguracija.server.protocol); // Ispisuje "http"

```

Možemo i koristeći notaciju uglatih zagrada `[]`:

```

konfiguracija.server["protocol"] = "http";

```

ili

```

konfiguracija["server"]["protocol"] = "http";

```

Ima li smisla dodavati naknadno svojstva? Ako ne znamo unaprijed koja će svojstva biti potrebna, onda ima smisla. Ako znamo unaprijed, onda je bolje definirati sva svojstva odmah. Primjerice, ako znamo svojstva `server` konfiguracije, možemo odmah napisati:

```
let konfiguracija = {
  server: {
    host: "localhost",
    port: 8080,
    protocol: "http",
  },
};
```

Ako ne znamo, imamo više opcija:

1. Možemo definirati prazan objekt i dodavati svojstva kako ih trebamo.

```
let konfiguracija = {
  server: {},
};

konfiguracija.server.host = "localhost";
konfiguracija.server.port = 8080;
konfiguracija.server.protocol = "http";
```

2. Možemo napraviti isto, ali definirati i koja podsvojstva će imati `server` objekt.

```
let konfiguracija = {
  server: {
    host: "", // Prazni string jer nagađamo da će biti string
    port: null, // Null jer nagađamo da će biti broj
    protocol: "", // Prazni string jer nagađamo da će biti string
  },
};

konfiguracija.server.host = "localhost";
konfiguracija.server.port = 8080;
konfiguracija.server.protocol = "http";
```

Brisanje podataka unutar ugniježđenih objekata

Kako obrisati podatke unutar ugniježđenih objekata? Na primjer, tj. kako obrisati `port` servera u našem objektu `konfiguracija`? Koristimo `delete` naredbu.

```
delete konfiguracija.server.port; // vraća true
console.log(konfiguracija.server.port); // Ispisuje "undefined"
```

Naravno, objekte možemo i dublje ugniježđivati, koliko god želimo. U praksi, nećemo ići dublje od 3-4 razine ugniježđivanja, jer postaje nepraktično i teško za održavanje.

2.2 Polja unutar objekata

Zamislite da radite neku web trgovinu, morate na neki način pohranjivati podatke o kupcu i narudžbama. Podaci koje želimo pohraniti su: `ime`, `prezime`, `adresa`, `kontakt` i `narudžbe`. Pod adresu želimo pohraniti `ulica`, `grad` i `poštanski broj`. Pod kontakt želimo pohraniti `telefon` i `email`. Kako ćemo pohraniti narudžbe? Narudžba se sastoji od više podataka iste strukture (stavki/proizvoda), dakle moramo koristiti polja!

Prvo ćemo pohraniti osnovne podatke o kupcu:

```
let kupac = {
    ime: "Ivo",
    prezime: "Ivić",
    adresa: "Ulica 123, 52100 Pula",
    kontakt: "0911234567",
    email: "iivic@gmail.com",
};
```

Ideja je da svojstva `adresa` i `kontakt` budu objekti.

Definirat ćemo i objekt `narudžbe` gdje ćemo pohraniti proizvode koje je kupac naručio i ukupnu cijenu narudžbe.

Novi oblik ugniježđene strukture koji sad moramo koristiti jesu **polja unutar objekata**.

```
let kupac = {
    ime: "Ivo",
    prezime: "Ivić",
    adresa: {
        ulica: "Ulica 123",
        grad: "Pula",
        postanskiBroj: "52100",
    },
    kontakt: {
        telefon: "0911234567",
        email: "iivic@gmail.com",
    },
    narudžbe: {
        proizvodi: ["Mobitel", "Slušalice", "Punjač"],
        ukupnaCijena: 1500,
    },
};
```

Koristili smo polje `proizvodi` unutar objekta `narudžbe` kako bismo pohranili proizvode koje je kupac naručio. Polje `proizvodi` je niz stringova. Kako dohvatiti proizvode koje je kupac naručio?

```
console.log(kupac.narudžbe.proizvodi); // Ispisuje ["Mobitel", "Slušalice", "Punjač"]
```

Kako dohvatiti prvi proizvod iz niza proizvoda?

```
console.log(kupac.narudžbe.proizvodi[0]); // Ispisuje "Mobitel"
```

2.2.1 Iteracija kroz polje unutar objekata

Kako iterirati kroz **polje unutar objekata**? Na primjer, kako ispisati sve proizvode koje je kupac naručio? Možemo koristeći `for` petlju:

```
for (let i = 0; i < kupac.narudzbe.proizvodi.length; i++) {  
    console.log(kupac.narudzbe.proizvodi[i]); // Ispisuje svaki proizvod - "Mobitel",  
    "Slušalice", "Punjač"  
}
```

ili bolje, koristeći `for-of` petlju:

```
for (let proizvod of kupac.narudzbe.proizvodi) {  
    console.log(proizvod); // Ispisuje svaki proizvod - "Mobitel", "Slušalice", "Punjač"  
}
```

Ovo je u redu, međutim naši proizvodi u narudžbi će u web trgovini uvijek sadržavati i cijenu i neku naručenu količinu. Kako ćemo to pohraniti? Možemo koristiti **objekte unutar polja**.

2.3 Objekti unutar polja

Nastavljamo s prethodnim primjerom. Recimo da je kupac naručio 3 proizvoda: "Mobitel" 1 kom, "Slušalice" 1 kom i "Punjač" 2 kom. Cijene proizvoda su 300, 20 i 10 eur. Kako pohraniti proizvode?

Idemo proizvode pohraniti kao zasebne objekte, prvo izvan objekta `kupac`, a zatim ih dodati u objekt `kupac`.

```
let proizvod_1 = {  
    naziv: "Mobitel",  
    kolicina: 1,  
    cijena: 300,  
};  
let proizvod_2 = {  
    naziv: "Slušalice",  
    kolicina: 1,  
    cijena: 20,  
};  
let proizvod_3 = {  
    naziv: "Punjač",  
    kolicina: 2,  
    cijena: 10,  
};
```

Sada ćemo dodati proizvode u objekt `kupac`:

```
kupac.narudzbe.proizvodi.push(proizvod_1);  
kupac.narudzbe.proizvodi.push(proizvod_2);  
kupac.narudzbe.proizvodi.push(proizvod_3);
```

Objekt `kupac` sada izgleda ovako:

```
let kupac = {
    ime: "Ivo",
    prezime: "Ivić",
    adresa: {
        ulica: "Ulica 123",
        grad: "Pula",
        postanskiBroj: "52100",
    },
    kontakt: {
        telefon: "0911234567",
        email: "iivic@gmail.com",
    },
    narudzbe: {
        proizvodi: [
            // U polje smo dodali objekte proizvoda
            {
                naziv: "Mobitel",
                kolicina: 1,
                cijena: 300,
            },
            {
                naziv: "Slušalice",
                kolicina: 1,
                cijena: 20,
            },
            {
                naziv: "Punjač",
                kolicina: 2,
                cijena: 10,
            },
        ],
        ukupnaCijena: 0,
    },
};
```

Novi oblik ugniježđene strukture koji smo sad iskoristili jesu **objekti unutar polja**.

Idemo vidjeti kako sada dohvaćamo podatke. Polje `proizvodi` sadrži objekte, pa ćemo morati koristiti `.` operator za dohvaćanje svojstava objekata.

```
console.log(kupac.narudzbe.proizvodi[0].naziv); // Ispisuje "Mobitel"
console.log(kupac.narudzbe.proizvodi[0].kolicina); // Ispisuje 1
console.log(kupac.narudzbe.proizvodi[0].cijena); // Ispisuje 300
```

Kako možemo iterirati kroz proizvode i ispisati ih? Možemo koristiti `for-of` petlju:

Pripazite, `proizvod` je sada objekt, pa ćemo morati koristiti `.` operator za dohvaćanje svojstava objekta.

```

for (let proizvod of kupac.narudzbe.proizvodi) {
    console.log(proizvod.naziv); // Ispisuje naziv proizvoda
    console.log(proizvod.kolicina); // Ispisuje količinu proizvoda
    console.log(proizvod.cijena); // Ispisuje cijenu proizvoda
}

```

Kako izračunati ukupnu cijenu narudžbe? Iterirajmo kroz proizvode i zbrojimo cijene:

```

let ukupnaCijena = 0;
for (let proizvod of kupac.narudzbe.proizvodi) {
    ukupnaCijena += proizvod.kolicina * proizvod.cijena;
}
kupac.narudzbe.ukupnaCijena = ukupnaCijena;
console.log(kupac.narudzbe.ukupnaCijena); // Ispisuje 340

```

Uočite glavni problem: Narudžbe su ustvari objekt (`narudzbe`), gdje se svaka narudžba sastoji od više proizvoda (polje objekata) i ukupne cijene.

- Što ako kupac ima više narudžbi? Gdje to dodajemo i kako?

Rješenje je da svaka narudžba bude zaseban objekt koji ćeemo pohranjivati u tzv. **polje objekata**.

Dakle, do sada smo imali objekt `narduzbe` koji sadržava polje objekata `proizvodi`. Narudžbe su množina narudžbi, pa ima smisla da budu polje. Svaka narudžba sastoji se potencijalno više stavki (proizvoda), pa ima smisla da svaka narudžba bude objekt.

Dakle, definirajmo jednu narudžbu kao objekt:

```

let narudzba_1 = {
    stavke: [
        // Polje objekata
        {
            naziv: "Mobitel",
            kolicina: 1,
            cijena: 300,
        },
        {
            naziv: "Slušalice",
            kolicina: 1,
            cijena: 20,
        },
        {
            naziv: "Punjač",
            kolicina: 2,
            cijena: 10,
        },
    ],
    ukupnaCijena: 0,
};

```

Zašto ne bi zamijenili svojstvo za ukupnu cijenu s odgovarajućom metodom? Dodat ćemo metodu koja za svaku stavku (proizvod) računa ukupnu cijenu narudžbe.

```
let narudzba_1 = {
  stavke: [
    // Polje objekata
    {
      naziv: "Mobitel",
      kolicina: 1,
      cijena: 300,
    },
    {
      naziv: "Slušalice",
      kolicina: 1,
      cijena: 20,
    },
    {
      naziv: "Punjač",
      kolicina: 2,
      cijena: 10,
    },
  ],
  // Vraća ukupnu cijenu narudžbe (340)
  ukupnaCijena: function () {
    let ukupnaCijena = 0;
    for (let stavka of this.stavke) {
      ukupnaCijena += stavka.kolicina * stavka.cijena;
    }
    return ukupnaCijena;
  },
  valuta: "eur", // Možemo dodati i valutu kao zasebno svojstvo
};
```

Sada ćemo svojstvo `narudzbe` iz objekta `kupac` pretvoriti u polje objekata i u njega dodati našu narudžbu - `narudzba_1`.

```
let kupac = {
  ime: "Ivo",
  prezime: "Ivić",
  // Objekt unutar objekta `kupac`
  adresa: {
    ulica: "Ulica 123",
    grad: "Pula",
    postanskiBroj: "52100",
  },
  // Objekt unutar objekta `kupac`
  kontakt: {
    telefon: "0911234567",
    email: "iivic@gmail.com",
  },
  // Polje objekata unutar objekta `kupac`
```

```
narudzbe: [  
],  
};
```

Kako sad dohvati ukupnu cijenu prve narudžbe našeg kupca?

```
console.log(kupac.narudzbe[0].ukupnaCijena()); // 340
```

Da rezimiramo, u ovom primjeru imali smo **objekt** `narudzbe` koji je postao **polje objekata** `narudzba`.

Svaka narudžba je objekt koji sadržava **polje objekata** `stavke`.

Dodatno, svaka `stavka` je objekt (ima svojstva `naziv`, `kolicina`, `cijena`). Svaka narudžba ima svoju ukupnu cijenu, koja je **metoda objekta** `narudzba`.

Konačno, naš objekt `kupac` sada izgleda ovako:

```
let kupac = {  
    ime: "Ivo",  
    prezime: "Ivić",  
    adresa: {  
        ulica: "Ulica 123",  
        grad: "Pula",  
        postanskiBroj: "52100",  
    },  
    kontakt: {  
        telefon: "0911234567",  
        email: "iivic@gmail.com",  
    },  
    narudzbe: [  
        {  
            stavke: [  
                {  
                    naziv: "Mobitel",  
                    kolicina: 1,  
                    cijena: 300,  
                },  
                {  
                    naziv: "Slušalice",  
                    kolicina: 1,  
                    cijena: 20,  
                },  
                {  
                    naziv: "Punjač",  
                    kolicina: 2,  
                    cijena: 10,  
                },  
            ],  
            ukupnaCijena: function () {  
                let ukupnaCijena = 0;  
                for (let stavka of this.stavke) {  
                    ukupnaCijena += stavka.kolicina * stavka.cijena;  
                }  
            }  
        }  
    ]  
}
```

```

        }
        return ukupnaCijena;
    },
    valuta: "eur",
},
],
};

```

► Objekt kupac - s komentarima

```

// Glavni objekt
let kupac = {
    ime: "Ivo",
    prezime: "Ivić",
    // Objekt `adresa` unutar objekta `kupac`
    adresa: {
        ulica: "Ulica 123",
        grad: "Pula",
        postanskiBroj: "52100",
    },
    // Objekt `kontakt` unutar objekta `kupac`
    kontakt: {
        telefon: "0911234567",
        email: "iivic@gmail.com",
    },
    // Polje `narudzbe` unutar objekta `kupac`
    narudzbe: [
        // Objekt `narudzba_1` unutar polja `narudzbe`
        {
            // Polje `stavke` unutar objekta `narudzba_1`
            stavke: [
                // 3 objekta `proizvod` unutar polja `stavke`
                {
                    naziv: "Mobitel",
                    kolicina: 1,
                    cijena: 300,
                },
                {
                    naziv: "Slušalice",
                    kolicina: 1,
                    cijena: 20,
                },
                {
                    naziv: "Punjač",
                    kolicina: 2,
                    cijena: 10,
                },
            ],
            // Metoda `ukupnaCijena` unutar objekta `narudzba_1`
            ukupnaCijena: function () {
                let ukupnaCijena = 0;

```

```

        for (let stavka of this.stavke) {
            ukupnaCijena += stavka.kolicina * stavka.cijena;
        }
        return ukupnaCijena;
    },
    valuta: "eur",
},
],
};

```

Vježba 1

EduCoder šifra: `Valli`

Kino Valli je kino u Puli na adresi Giardini 1, 52100 Pula. Kino ima jednu dvoranu kapaciteta 209 sjedećih mjesta i prikazuje filmove gotovo svaki dan. Svoj program prikazuje putem web-a: <https://www.kinovalli.net>. Na web stranici možete pronaći Tjedni raspored filmova gdje se prikazuje koji filmovi se prikazuju u kojem terminu (datum i vrijeme). Isti film prikazuje se u više termina u tjednom rasporedu, a svaki film se dodatno sastoji od sekcije gdje se prikazuje naslov filma, trajanje, godina izlaska, kategorija/žanr, izvorno ime, period prikazivanja, IMDb link, kratki opis, režija te više fotografija.

Za rezervaciju karata potrebno je unijeti osobne podatke prilikom registracije: ime, prezime, adresa (ulica, grad) i kontakt (telefon, email). Također, potrebno je za određenu projekciju unijeti broj karata i odabrati sjedala, nakon čega se izračunava ukupna cijena rezervacije. Ovo realizirajte metodom `dodajRezervaciju()`. Možete dodati i pomoćne metode za provjeru dostupnih sjedala, maksimalnog broja prodanih karata (popunjavanje kapaciteta), izračuna ukupne cijene i sl.

Na temelju ugrubo danog opisa poslovnog procesa kina Valli, definirajte objekt `kinovalli` koji će sadržavati sve potrebne podatke za opisani poslovni proces. Za modeliranje ovog objekta koristite ugniježđene strukture objekata i polja.

Prvo definirajte objekte `film` koristeći sljedeće podatke:

Film 1: INTERSTELLAR, 169 min, 2014. god, znanstvena fantastika, Interstellar, 01.10.2014. - 07.10.2014., <https://www.imdb.com/title/tt0816692/>, režija: Christopher Nolan, **Fotografije:** "<https://www.kinovalli.net/Interstellar/fakePoveznicaSlika1>", "<https://www.kinovalli.net/Interstellar/fakePoveznicaSlika2>", "<https://www.kinovalli.net/Interstellar/fakePoveznicaSlika3>", **Opis:** "Skupina astronauta putuje u svemir i ulazi u crvotočinu kako bi pronašla novi planet na koji bi se ljudi mogli naseliti."

Film 2: DINA: DRUGI DIO, 166 min, 2023. god, znanstvena fantastika, Dune: Part Two, 29.2.2024. - 12.3.2024., <https://www.imdb.com/title/tt15239678/>, režija: Denis Villeneuve, **Fotografije:** "<https://www.kinovalli.net/Dune2/fakePoveznicaSlika1>", "<https://www.kinovalli.net/Dune2/fakePoveznicaSlika2>", "<https://www.kinovalli.net/Dune2/fakePoveznicaSlika3>", **Opis:** "Nove pustolovine Paula Atreidesa i Chani, kao i sudbine brojnih drugih likova iz svijeta temeljenog na romanima Franka Herberta."

Nakon toga definirajte objekt `kinovalli` koji će sadržavati sve potrebne podatke za opisani poslovni proces. Potrudite se da objekt bude što precizniji, **jedinstvenog rješenja nema**, ali pokušajte što bolje modelirati opisani poslovni proces, pokrivajući što veći broj mogućih slučajeva: npr. popunjeno dvorane, provjere dostupnih sjedala, brisanja rezervacije itd.

Jednom kad definirate objekt i metodu `dodajRezervaciju()`, pozovite metodu `dodajRezervaciju()`.

```
let kinoValli = {  
    // Vaš kôd ovdje...  
};  
  
kinoValli.dodajRezervaciju(...);
```

Vježba 2

EduCoder šifra: `rentaBoat`

Obrt `rentaBoat` bavi se iznajmljivanjem brodica i brodova za razne prigode. Njihova web stranica <https://www.rentaboat.net/> glavni je kanal komunikacije s korisnicima. Na web stranici se nalazi ponuda brodova i brodica, gdje se prikazuje koji brodovi su dostupni za najam, u kojem terminu (datum i vrijeme) te cijena najma. Svaki brod/brodica ima svoje karakteristike: naziv, maksimalni kapacitet, tip, godina proizvodnje, maksimalna brzina, snaga motora u KS, dodatna oprema, dnevna cijena najma.

U dodatnu opremu mogu spadati: tuš, hladnjak, GPS, radio, kuhinja, WC, utičnice za struju, tenda, gumenjak, oprema za ribolov, ehosonder.

Tipovi brodica i brodova mogu uključivati: gliser, jahta, brodica za ribolov, gumenjak, jedrilica, brodica s kabinom, mala brodica bez kabine.

Korisnici se moraju registrirati i unijeti osobne podatke, te za registraciju odabrati željeni termin najma (datumi od/do), broj osoba, željenu dodatnu opremu te naravno samu brodicu. Nakon što korisnik unese sve podatke, izračunava se ukupna cijena najma i korisnika se obavještava o uspješnoj rezervaciji.

Na temelju ugrubo danog opisa poslovnog procesa obrta `rentaBoat`, definirajte objekt `rentaBoat` koji će sadržavati sve potrebne podatke za opisani poslovni proces. Za modeliranje ovog objekta koristite ugniježđene strukture objekata i polja.

Prvo definirajte 3 objekta `brod` koristeći sljedeće podatke:

Brod 1: "Gliser", 2015. god, 20 čvorova, 150 KS, 6 osoba, "Tuš", "Hladnjak", "GPS", "Radio", "Tenda", "Oprema za ribolov", "Ehosonder", 250 eur/dan

Brod 2: "Jahta", 2018. god, 35 čvorova, 300 KS, 8 osoba, "Tuš", "Hladnjak", "GPS", "Radio", "Kuhinja", "WC", "Uticnice za struju", "Tenda", "Gumenjak", 1000 eur/dan

Brod 3. "Jedrilica", 2019. god, 12 čvorova, 50 KS, 4 osobe, "Tuš", "Hladnjak", "GPS", "Radio", "Kuhinja", "WC", "Uticnice za struju", "Gumenjak", "Oprema za ribolov" 300 eur/dan

Nakon toga definirajte objekt `rentaBoat` koji će sadržavati sve potrebne podatke za opisani poslovni proces. Potrudite se da objekt bude što precizniji, **jedinstvenog rješenja nema**, ali pokušajte što bolje modelirati opisani poslovni proces.

Jednom kad napravite objekt `rentaBoat`, definirajte metode `provjeriOpremu()`, `ukupnaCijena()` i `dodajRezervaciju()`. Ideja je da metoda `dodajRezervaciju()` poziva metode `provjeriOpremu()` i `ukupnaCijena()`. Na kraju pozovite metodu `dodajRezervaciju()`.

```
let rentaBoat = {
    // Vaš kôd ovdje...
};

rentaBoat.dodajRezervaciju(...);
```

2.4 Polja unutar polja

Ugniježđena polja su polja definirana unutar drugih polja, još se nazivaju **multidimenzionalnim poljima** (*eng. multidimensional arrays*). U praksi, multidimenzionalna polja se koriste za pohranu podataka koji su međusobno povezani.

Multidimenzionalna polja možemo definirati ugnježđivanjem polja definiranih uglatim zagradama `[]`.

Primjer jednodimenzionalnog polja:

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

Primjer dvodimenzionalnog polja (**2D matrica**)

```
let matrica = [
    [1, 2, 3], // Prvi redak
    [4, 5, 6], // Drugi redak
    [7, 8, 9], // Treći redak
];
```

U ovom primjeru imamo matricu dimenzija 3×3 . Matrica ima 3 redaka i 3 stupca. Svaki redak je polje koje sadrži 3 elementa. Matrica je dvodimenzionalna jer ima dvije (2) dimenzionalnosti (redak i stupac).

Kako možemo dohvatiti elemente matrice? Koristimo indekse redaka i stupaca.

```
console.log(matrica[0][0]); // Ispisuje 1 (prvi redak, prvi stupac)
console.log(matrica[1][1]); // Ispisuje 5 (drugi redak, drugi stupac)
console.log(matrica[2][0]); // Ispisuje 7 (treći redak, prvi stupac)
```

Možemo dohvatiti i samo cijeli redak matrice koristeći indeks redaka

```
console.log(matrica[0]); // Ispisuje [1, 2, 3] (prvi redak)
console.log(matrica[1]); // Ispisuje [4, 5, 6] (drugi redak)
console.log(matrica[2]); // Ispisuje [7, 8, 9] (treći redak)
```

Modifikacije elemenata višedimenzionalnih polja rade se na isti način kao i kod jednodimenzionalnih polja.

```
matrica[0][0] = 10; // Modificira prvi element matrice

console.log(matrica[0][0]); // Ispisuje 10

console.log(matrica); // Ispisuje [[10, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Matrice se u programiranju reprezentiraju višedimenzionalnim poljima. Ako se pokušate dosjetiti primjera gdje bi se mogli koristiti ovakvi zapisi, na prvu će vam višedimenzionalna polja možda izgledati komplikirana i nepotrebna, ali u praksi su vrlo korisna i često se koriste.

U računarstvu i informacijskoj znanosti, matrice se koriste za:

- računalnu grafiku (slike, video, 3D modeli i sl.)
- strojno učenje i umjetnu inteligenciju
- modeliranje i simulacije
- kriptografiju
- teorija grafova
- obrada signala
- linearne transformacije

2.4.1 Iteracije kroz više dimenzija

Iteracije kroz više dimenzija rade se na isti način kao i kod jednodimenzionalnih polja, samo što koristimo više petlji - odnosno koristimo **ugniježđene petlje**.

Idemo definirati jednu matricu dimenzija 5x5.

```
let matrica = [
  [10, 20, 45, 4, 3],
  [6, 7, 8, 18, 11],
  [30, 12, 70, 14, 5],
  [16, 22, 100, 19, 2],
  [18, 22, 23, 24, 266],
];

console.log(matrica); // [[10,20,45,4,3],[6,7,8,18,11],[30,12,70,14,5],[16,22,100,19,2],
[18,22,23,24,266]]
```

Idemo iterirati kroz matricu i ispisati sve elemente.

```
for (let i = 0; i < matrica.length; i++) {
  console.log(matrica[i]); // Ispisuje svaki redak matrice
  /*
    [10, 20, 45, 4, 3]
    [6, 7, 8, 18, 11]
    [30, 12, 70, 14, 5]
    [16, 22, 100, 19, 2]
    [18, 22, 23, 24, 266]
  */
}
```

Kôd iznad ispisuje 5 puta (5 elemenata), ne ispisuje svaki element matrice (25 elemenata).

Kako su rezultati ispisivanja redaka matrice **polja**, moramo iterirati ponovo kroz svaki element tih **5 polja**.

```

for (let i = 0; i < matrica.length; i++) {
    for (let j = 0; j < matrica[i].length; j++) {
        console.log(matrica[i][j]); // Ispisuje svaki element matrice
        //Ispisuje: 10, 20, 45, 4, 3, 6, 7, 8, 18, 11, 30, 12, 70, 14, 5, 16, 22, 100, 19, 2,
18, 22, 23, 24, 266
    }
}

```

Kako bismo definirali matricu dimenzija $3 \times 3 \times 3$, koristimo 3 ugniježđena polja koja sadrže po 3 elementa (također polja):

```

let matrica3D = [
    [
        // Prvi sloj
        [1, 2, 3],
        [4, 5, 6],
        [7, 8, 9],
    ],
    [
        // Drugi sloj
        [10, 11, 12],
        [13, 14, 15],
        [16, 17, 18],
    ],
    [
        // Treći sloj
        [19, 20, 21],
        [22, 23, 24],
        [25, 26, 27],
    ],
];

```

Primjer kako izgleda iteracija kroz 3D matricu:

```

for (let i = 0; i < matrica3D.length; i++) {
    for (let j = 0; j < matrica3D[i].length; j++) {
        for (let k = 0; k < matrica3D[i][j].length; k++) {
            console.log(matrica3D[i][j][k]); // Ispisuje svaki element 3D matrice
        }
    }
}

```

3D matricama možemo reprezentirati razne stvari, npr. u području fizike i inženjerstva možemo 3D matricom definirati tzv. **Stress tensor** (tenzor naprezanja) koji se koristi za opisivanje naprezanja u različitim točkama nekog tijela (Cauchy stress tensor).

U računalnoj grafici možemo 3D matricom definirati **voxel grid** gdje svaki element matrice predstavlja jedan voxel (3D piksel) koji sadrži informacije o boji, teksturi, materijalu i sl.

2.4.2 Stvaranje višedimenzionalnih polja pomoću `Array` konstruktora

U višedimenzionalna polja ne moraju biti pohranjeni samo brojevi (premda je to najčešće), već i bilo koji drugi tipovi podataka. U tom slučaju se višedimenzionalna polja više ne nazivaju matricama.

Npr., pohranimo u višedimenzionalno polje stringove.

```
let filmovi = [
  "Begin Again",
  "Soul",
  ["Matrix", "Matix Reloaded", "Matrix Revolutions"], // polje (sadrži samo stringove)
  ["Frozen", "Frozen 2", ["Tangled", "Alladin"]], // 2D polje (jer sadrži stringove i još jedno polje)
];
```

Drugi način je pozivanjem `Array` konstruktora.

```
let filmovi = new Array();

filmovi[0] = "Begin Again";
filmovi[1] = "Soul";
filmovi[2] = new Array("Matrix", "Matrix Reloaded", "Matrix Revolutions"); // polje
(sadrži samo stringove)
filmovi[3] = new Array("Frozen", "Frozen 2", new Array("Tangled", "Alladin")); // 2D polje
(jer sadrži stringove i još jedno polje)
```

Dakle `filmovi[2]` predstavlja jednodimenzionalno polje s tri elementa (`filmovi [string]`), a `filmovi[3]` predstavlja dvodimenzionalno polje s tri elementa (`filmovi [string]`) i polje s dva elementa (`filmovi [string]`).

Kako se raspoređuju elementi u višedimenzionalnim poljima? Pogledamo ilustraciju:



Izvor: <https://dev.to/sanchithasr/understanding-nested-arrays-2hf7>

Ako želimo dohvatiti film "Tangled" iz polja `filmovi`, koristimo indekse `[3][2][0]`.

```
console.log(filmovi[3][2][0]); // Ispisuje "Tangled"
```

Ako želimo dohvatiti film "Matrix Reloaded" iz polja `filmovi`, koristimo indekse `[2][1]`.

```
console.log(filmovi[2][1]); // Ispisuje "Matrix Reloaded"
```

Polje možemo "izravnati", odnosno **svesti višedimenzionalno polje na jednodimenzionalno polje** koristeći metodu `Array.flat()`.

Primjerice uzmimo više dimenzionalno polje koje želimo svesti na jednodimenzionalno polje (listu).

```
const arr1 = [0, 1, 2, [3, 4]];

console.log(arr1.flat()); // [0, 1, 2, 3, 4]
```

Metoda `Array.flat()` smanjuje dubinu polja za jedan nivo. Ako želimo smanjiti dubinu polja za više nivoa, unosimo argument `depth`.

```
const arr2 = [0, 1, 2, [3, 4, [5, 6]]];
console.log(arr2.flat()); // [0, 1, 2, 3, 4, [5, 6]]

// ali
console.log(arr2.flat(2)); // [0, 1, 2, 3, 4, 5, 6]
```

Već smo naveli moguće primjene višedimenzionalnih polja te naglasili da se u pravilu koriste za pohranu numeričkih podataka, koji su međusobno povezani odnosno predstavljaju neku **vrstu višedimenzionalne strukture**.

- U praksi, ovaj primjer nije nešto što želite pohraniti u višedimenzionalno polje. Dohvaćanje filmova postaje nezgrapno (više-dimenzionalno indeksiranje), značajno se smanjuje čitljivost kôda, a i održavanje postaje teže.

Filmove je bolje pohraniti koristeći ranije naučene ugniježđene strukture - **kombiniranjem objekata i polja**.

Recimo ovako:

```
let filmovi = {
  singleFilms: ["Begin Again", "Soul"],
  matrixSeries: ["Matrix", "Matrix Reloaded", "Matrix Revolutions"],
  disneyAnimations: {
    frozenSeries: ["Frozen", "Frozen 2"],
    classicTales: ["Tangled", "Alladin"],
  },
};
```

Sada možemo dohvatiti filmove na jednostavniji način:

```
console.log(filmovi.disneyAnimations.classicTales[0]); // Ispisuje "Tangled"
console.log(filmovi.matrixSeries[1]); // Ispisuje "Matrix Reloaded"
```

Vježba 3

EduCoder šifra: `matrix`

Definirajte dvodimenzionalno polje (matricu) dimenzija 3×3 koja će sadržavati random brojeve od 1 do 9. Matricu morate "izgraditi" s pomoću ugniježđenih petlji, ne ručno! Implementirajte funkciju `randomNumbers()` koja vraća random broj između 1 i 9 koristeći `Math.random()` metodu. Na kraju definirajte funkciju `ispisMatrice(matrix2D)` koja ispisuje sve elemente dvodimenzionalne matrice `matrix2D`.

2.5 Sažetak ugniježđenih struktura

Ugniježđene strukture su strukture koje se sastoje od više različitih struktura koje su međusobno povezane. U kontekstu ove skripte, one se odnose na ugniježđene objekte i polja. Ugniježđene strukture koje smo obradili su:

1. **Objekti unutar objekata** `{}{}`
2. **Polja unutar objekata** `[][]`
3. **Objekti unutar polja** `[{}]`
4. **Polja unutar polja** `[[[]]]`

U kontekstu web programiranja, naučili smo da često koristimo prve 3 strukture - primjerice za modeliranje raznih entiteta iz stvarnog života. Međutim, višedimenzionalna polja odnosno polja unutar polja su korisna za pohranu drugih vrsta podataka, npr. matrica, 3D modela, slika, videa, zvuka, tabličnih podataka i sl.

1. **Objekte unutar objekata** koristimo za modeliranje entiteta koji imaju svoje pod-entitete (npr. kupac s podentitetima adresa i kontakt). Kako adresa i kontakt sami po sebi nisu jasni entiteti, koristimo objekte kako bi ih razložili na detaljnije podatke.

```
let kupac = {  
    // Glavni objekt `kupac`  
    ime: "Ivo",  
    prezime: "Ivić",  
    // Podobjekt `adresa` unutar objekta `kupac`  
    adresa: {  
        ulica: "Ulica 123",  
        grad: "Pula",  
        postanskiBroj: "52100",  
    },  
    // Podobjekt `kontakt` unutar objekta `kupac`  
    kontakt: {  
        telefon: "0911234567",  
        email: "iivic@gmail.com",  
    },  
};
```

2. **Polja unutar objekata** koristimo za modeliranje entiteta koji imaju više podataka istog tipa (npr. kupac s više narudžbi). Kako narudžbe nisu jasni entiteti, modeliramo ih pomoću objekata kako bi ih razložili na detaljnije podatke, a potom te objekte pohranjujemo u polje.
3. Svaku stavku narudžbe predstavljamo kao **Objekt unutar polja**

```
let narudzbe = [  
    {  
        // Polje objekata `stavke` unutar objekta `narudzba`  
        stavke: [  
            {  
                // Objekt `stavka` unutar polja `stavke`  
                naziv: "Mobitel",  
                kolicina: 1,  
            },  
        ],  
    },  
];
```

```

        cijena: 300,
    },
    {
        // Objekt `stavka` unutar polja `stavke`
        naziv: "Slušalice",
        kolicina: 1,
        cijena: 20,
    },
    {
        // Objekt `stavka` unutar polja `stavke`
        naziv: "Punjač",
        kolicina: 2,
        cijena: 10,
    },
],
// Metoda unutar objekta `narudzba`
ukupnaCijena: function () {
    let ukupnaCijena = 0;
    for (let stavka of this.stavke) {
        ukupnaCijena += stavka.kolicina * stavka.cijena;
    }
    return ukupnaCijena;
},
},
];
console.log(narudzbe[0].ukupnaCijena()); // 340

// Iteracija kroz polje objekata (stavke svake narudžbe)
for (let i = 0; i < narudzbe.length; i++) {
    for (let j = 0; j < narudzbe[i].stavke.length; j++) {
        console.log(narudzbe[i].stavke[j]); // Ispisuje svaku stavku narudžbe
    }
}
}

```

4. **Polja unutar polja** koristimo za modeliranje podataka koji su međusobno povezani (npr. matrica, 3D modeli, slike, video, zvuka, tablični podaci). U ovom slučaju, **svaki element polja je polje**.

```

let matrica = [
    [10, 20, 45, 4, 3],
    [6, 7, 8, 18, 11],
    [30, 12, 70, 14, 5],
    [16, 22, 100, 19, 2],
    [18, 22, 23, 24, 266],
];
console.log(matrica[4][0]); // Ispisuje 18

// Iteracija kroz dvodimenzionalno polje (matricu)
for (let i = 0; i < matrica.length; i++) {
    for (let j = 0; j < matrica[i].length; j++) {
        console.log(matrica[i][j]); // Ispisuje svaki element matrice
}
}

```

```
}
```

Vježba 4

EduCoder šifra: student

Definirajte objekt `student` koji će sadržavati podatke o studentu: ime, prezime, adresa (ulica, grad, poštanski broj), kontakt (telefon, email), ocjene (polje objekata `Ocjena`).

- definirajte konstruktor `Ocjena` koji se sastoji od 2 svojstva: `numerickaOcjena` i `opisnaOcjena`. Konstruktor se mora pozivati samo s argumentom numeričke ocjene, opisna ocjena dodjeljuje se ovisno o numeričkoj ocjeni (npr. 5 - "odličan", 4 - "vrlo dobar", 3 - "dobar", 2 - "dovoljan", 1 - "nedovoljan", default = "nevažeća ocjena"). Primjer poziva konstruktora: `new Ocjena(5)` - stvara objekt `{numerickaOcjena: 5, opisnaOcjena: "odličan"}`.
- Dodajte studentu nekoliko ocjena (npr. 5 ocjena) implementacijom metode `dodajOcjenu()` i izračunajte prosječnu ocjenu studenta zaokruženu na dvije decimale (dodajte metodu `prosjecnaOcjena()` u objekt `student`).

Samostalni zadatak za vježbu 6

EduCoder šifra: restoran

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

Imate zadatak napraviti malu web aplikaciju za restoran kako bi gosti mogli naručiti hranu i piće preko tableta u restoranu. Definirajte objekt `restoran` koji će sadržavati podatke o restoranu: naziv, adresa (ulica, grad, poštanski broj), kontakt (telefon, email), objekt meni (sadrži polje objekata `Jelo` i `Pice`).

- definirajte konstruktor `Jelo` koji se sastoji od 5 svojstva: `naziv`, `cijena`, `opis`, `sastojci` i `kategorija`.

Primjer pozivanja konstruktora može biti: `new Jelo("Margherita", 7, "Pizza s rajčicom i mozzarellom sirom", ["rajčica", "sir"], "glavno jelo")`.

- Definirajte nekoliko jela pozivanjem konstruktora `Jelo`
- definirajte konstruktor `Pice` koji se sastoji od 4 svojstva: `naziv`, `cijena`, `opis` i `kategorija`. Primjer pozivanja konstruktora može biti: `new Pice("Coca-Cola", 2, "Osvježavajuće gazirano bezalkoholno piće", "bezalkoholno")`.
- Definirajte nekoliko pića pozivanjem konstruktora `Pice`

U objekt `restoran` dodajte metodu `dodajNarudzbu()` koja će dodati novu narudžbu u polje narudžbi. Metoda mora raditi na sljedeći način:

```
this.dodajNarudzbu = async function (narudzba) { // async funkcija zbog specifičnosti  
    EduCodera, inače nije potrebno  
    // Vaš kôd ovdje...  
};
```

- kada se pozove funkcija, korisniku se mora prikazati izbornik u konzoli koji sadrži sva jela i pića iz menija s indeksom koji počinje od 1 ispred zapisa. Primjer:

```
1. Margherita (Pizza s rajčicom i Mozarella sirom) - 7 eur
2. Coca-Cola (Osvježavajuće gazirano bezalkoholno piće) - 3 eur
3. Tjestenina s umakom od rajčice (Tjestenina s umakom od svježe rajčice) - 8 eur
4. Fanta (Osvježavajuće gazirano bezalkoholno piće) - 2 eur
5. Piletina s povrćem (Piletina s povrćem i umakom od vrhnja) - 10 eur
```

- korisnik unosi redni broj jela ili pića koje želi naručiti. Ako korisnik unese redni broj koji ne postoji, ispisuje se poruka "Narudžba ne postoji, pokušajte ponovno". Koristite `prompt()` funkciju za unos podataka.
- ako korisnik unese ispravan redni broj, traži ga se količina koju želi naručiti. Ako korisnik unese količinu manju od 1, ispisuje se poruka "Količina mora biti veća od 0, pokušajte ponovno". Koristite `prompt()` funkciju za unos podataka.
- preferencije koje korisnik unosi moraju se spremati u objekt `trenutna_narudzba` i polje `stavke`, a dodatno, u objekte jela i/ili pića potrebno je dodati svojstvo `kolicina`.

Primjer za naručivanje 2 Margherita pizza i 2 Coca-Cola pića:

```
let trenutna_narudzba = {
  stavke: [
    {
      // narudžba Margherita (objekt Jelo + količina)
      naziv: "Margherita",
      cijena: 7,
      opis: "Pizza s rajčicom i mozzarella sirom",
      sastojci: ["rajčica", "sir"],
      kategorija: "glavno jelo",
      kolicina: 2,
    },
    {
      // narudžba Coca-Cola (objekt Pice + količina)
      naziv: "Coca-Cola",
      cijena: 2,
      opis: "Osvježavajuće gazirano bezalkoholno piće",
      kategorija: "bezalkoholno",
      kolicina: 2,
    },
  ],
};
```

- korisnik završava s naručivanjem kada unese redni broj 0. Tada se ispisuje trenutna narudžba i ukupna cijena narudžbe.
- dodajte `timestamp` u objekt `trenutna_narudzba` koji će sadržavati trenutni datum i vrijeme narudžbe.
- jednom kad je narudžba uspješno dodana obavijestite o tome korisnika funkcijom `alert()`. U poruci obavijestite korisnika i o ukupnoj cijeni narudžbe.

3. Napredne funkcije

Napredne funkcije i metode odnose se na kompleksnije metode i tehnike koje se koriste za rješavanje određenih tipova problema. Studenti će kroz ove vježbe naučiti koristiti funkcije višeg reda (eng. **higher-order functions**). To su funkcije koje primaju funkcije kao argumente, poput: `map()`, `filter()`, `reduce()`, `sort()`.

Detaljnije ćemo obraditi `callback` funkcije koje smo već spomenuli u primjerima skripte PJS3 te ćemo naučiti pisati tzv. `arrow` ili anonimne funkcije koje nam pružaju konkretniju sintaksu za pisanje funkcijskih izraza, o kojima je bilo riječi u skripti PJS2.

Važno je prije prolaska kroz ovo poglavlje dobro ponoviti koncepte funkcija, funkcijskih izraza, objekata, polja te ugniježđenih struktura.

Izvor: <https://blog.khanacademy.org/lets-reduce-a-gentle-introduction-to-javascripts-reduce-method/>

3.1 Callback funkcije

3.1.1 Primjer callback funkcije

U poglavlju PJS3 već smo ukratko napravili uvod u `callback` funkcije.

`Callback` funkcije su funkcije koje se koriste kao argumenti drugih funkcija. Drugim riječima, `callback` funkcija je funkcija koja se poziva unutar druge funkcije.

Vidjeli smo da `callback` funkcije možemo koristiti kao argumente za neke od metoda `Array` objekta, kao što su `find` i `filter`.

Primjer koji smo prošli u prošloj skripti je bio:

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

Za definirano polje `stabla` pokazali smo kako pronaći stablo `bor` koristeći metodu `find()`.

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

Ovdje je `callback` funkcija je **anonimna funkcija** koja se koristi kao argument za metodu `find()`. Ova `callback` funkcija je anonimna jer nema ime i definirana je direktno unutar metode `find()`.

- Kako bi nam bilo jasnije, idemo razdvojiti `callback` funkciju od metode `find()` na način da ćemo ju pretočiti u funkciju `pronadiBor()` koja provjerava je li stablo "bor".

Metoda `find()` će pozvati funkciju `pronadiBor()` za svaki element polja `stabla`.

```
let bor = stabla.find(pronadiBor); // Pozovi metodu find() s callback funkcijom  
pronadiBor()
```

Funkcija `pronadiBor()` mora imati jedan argument (`stabla`) koji predstavlja svaki element polja `stabla`.

Primjer kako možemo implementirati našu funkciju `pronadiBor()`:

```
function pronadiBor(stabla) { // Definiraj funkciju pronadiBor() koja prima jedan argument  
"stabla"  
    return stabla == "bor"; // Vrati true ako je "stabla" jednako "bor"  
}
```

Naš kôd sada izgleda ovako:

```
let stabla = new Array("hrast", "bukva", "javor", "bor", "smreka");  
  
function pronadiBor(stabla) {  
    return stabla == "bor";  
}  
  
let bor = stabla.find(pronadiBor); // Callback funkciju pronadiBor() proslijedujemo bez  
zagrada ()  
console.log(bor); // Ispisuje "bor"
```

Ima li kôd grešaka? Funkciju `pronadiBor` proslijedujemo bez zagrada `()`. Zašto? **Zato što želimo proslijediti referencu na funkciju, a ne rezultat izvršavanja funkcije.**

- Grešku bi dobili da smo napisali `let bor = stabla.find(pronadiBor());`. U tom slučaju, `pronadiBor()` bi se izvršila odmah, a rezultat bi bio proslijeđen metodi `find()`.

3.1.2 Osnovna podjela `callback` funkcija

Najjednostavnije rečeno, u JavaScriptu, `callback` funkcija je funkcija proslijeđena kao argument drugoj funkciji - `callback` funkcije se koriste za izvršavanje koda nakon što je druga funkcija završila izvršavanje.

U primjeru sa stablima koristili smo `callback` funkcije na 2 načina:

1. koristili smo **globalno definiranu funkciju** kao `callback` funkciju (*definirana s imenom*)
2. koristili smo **anonimnu funkciju** kao `callback` funkciju (*bez imena*)

1. Globalno definirana `callback` funkcija

Pokazat ćemo prvo 1. primjer gdje koristimo `callback` funkciju definiranu izvana kao argument za metodu `forEach()`. Rekli smo da je metoda `forEach()` metoda koja prolazi kroz svaki element polja i izvršava `callback` funkciju za svaki element odnosno za svaki element polja izvršava neku operaciju.

Zadatak nam je da za svaki element polja `brojevi` ispišemo kvadrat tog broja.

Prvo ćemo definirati polje `brojevi`:

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

Zatim ćemo definirati globalnu `callback` funkciju `ispisiKvadrat()` koja će ispisati kvadrat broja.

```
// ovu funkciju ćemo koristiti kao callback funkciju
function ispisiKvadrat(broj) {
    console.log(broj * broj);
}
```

Za sada je sve poznato, idemo upotrijebiti metodu `forEach()` i proslijediti `callback` funkciju `ispisiKvadrat()`.

```
brojevi.forEach(ispisiKvadrat); // Pozovi metodu forEach() s callback funkcijom
ispisiKvadrat()

// Ispisuje:
// 1
// 4
// 9
// 16
// 25
```

VAŽNO: Primijetite da **nismo** pozivali `callback` funkciju niti definirali argument `broj`. Metoda `forEach()` će to učiniti za nas - mi smo samo **proslijedili referencu na funkciju** `ispisiKvadrat`.

2. Anonimna `callback` funkcija

Sada ćemo pokazati kako isto definirati anonimnom `callback` funkcijom.

Anonimne funkcije u programiranju su funkcije koje nisu vezane nekim identifikatorom (imenom). Često predstavljaju argumente koji se proslijeđuju drugim funkcijama. Ponovite si poglavlje: **Uvod u funkcionsko programiranje** u skripti **PJS2**.

Opet ćemo definirati polje `brojevi`:

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

Idea je da ovoga puta koristimo **anonimnu** `callback` funkciju koja će ispisati kvadrat broja.

```
brojevi.forEach(nasaAnonimnaFunkcija); // ???
```

Anonimne funkcije možemo definirati na potpuno isti način kao i obične funkcije, samo što im ne navodimo, pogađate, ime.

```

let brojevi = [1, 2, 3, 4, 5];
brojevi.forEach(function(broj) { // Anonimna `callback` funkcija koja ispisuje kvadrat
  broja (bez imena)
  console.log(broj * broj);
});

// Ispisuje:
// 1
// 4
// 9
// 16
// 25

```

3.2 Callback funkcije s poljima

Kroz primjere s metodama `forEach()` i `find()` napravili smo uvod u `callback` funkcije. U ovom poglavlju proći ćemo kroz još nekoliko metoda `Array` objekta koje koriste `callback` funkcije.

U 4. poglavlju - `Polja` naučili smo koristili osnovne metode `Arary` objekta. Podijelili smo ih u:

- **metode dodavanja, brisanja i stvaranja novih polja:** npr. `push()`, `pop()`, `shift()`, `unshift()`, `splice()`, `slice()`
- **metode pretraživanja polja:** npr. `indexOf()`, `lastIndexOf()`, `includes()`, `find()`

Neke od metoda pretraživanja polja koje smo već spomenuli koriste `callback` funkcije. Primjer:

- `find(callbackFn)` metoda pretražuje polje i vraća prvi element koji zadovoljava uvjet definiran u `callback` funkciji.
- `findIndex(callbackFn)` metoda pretražuje polje i vraća indeks prvog elementa koji zadovoljava uvjet definiran u `callback` funkciji.
- `findLast(callbackFn)` metoda pretražuje polje i vraća zadnji element koji zadovoljava uvjet definiran u `callback` funkciji.
- `findLastIndex(callbackFn)` metoda pretražuje polje i vraća indeks zadnjeg elementa koji zadovoljava uvjet definiran u `callback` funkciji.

U ovom poglavlju, kroz primjere ćemo detaljnije proći kroz navedene metode, kao i dodatne metode `Array` objekta koje koriste `callback` funkcije.

3.2.1 Metoda `find(callbackFn)`

Metodu `find()` koristili smo za pretraživanje polja stabala i pronalazak stabla "bor".

```

let stabla = new Array("hrast", "bukva", "javor", "bor", "smreka");
let bor = stabla.find(function(stablo) { // Anonimna funkcija koja provjerava je li
  "stablo" jednako "bor"
  return stablo == "bor";
});
console.log(bor); // Ispisuje "bor"

```

Metoda `find()` vraća **prvi element** polja koji zadovoljava uvjet definiran u `callback` funkciji. Ako nema elementa koji zadovoljava uvjet, vraća se `undefined`.

Imamo definirano polje objekata `studenti`:

```
let studenti = [
  {ime: "Ivo", prezime: "Ivić", ocjena: 5},
  {ime: "Ana", prezime: "Anić", ocjena: 4},
  {ime: "Maja", prezime: "Majić", ocjena: 3},
  {ime: "Ivan", prezime: "Ivanić", ocjena: 2},
  {ime: "Pero", prezime: "Perić", ocjena: 1},
];
```

Želimo pronaći studenta s prezimenom `Ivanić`. Koristimo metodu `find()` i `callback` funkciju koja provjerava je li prezime studenta jednako `Ivanić`.

```
let student = studenti.find(function(student) { // Anonimna funkcija koja provjerava je li
  prezime studenta jednako "Ivanić"
  return student.prezime == "Ivanić";
});
console.log(student); // Ispisuje {ime: "Ivan", prezime: "Ivanić", ocjena: 2}
```

Što ako želimo pronaći studenta s negativnom ocjenom? Potrebno je samo redefinirati uvjet u `callback` funkciji.

```
let student = studenti.find(function(student) { // Anonimna funkcija koja provjerava je li
  ocjena studenta jednaka 1
  return student.ocjena === 1;
});
```

Što ako želimo pronaći studenta s ocjenom većom od 3? Izmijenit ćemo uvjet i definirati u vanjskoj `callback` funkciji.

```
function ocjenaVecaOdTri(student) {
  return student.ocjena > 3;
}
let student = studenti.find(ocjenaVecaOdTri); // Pozovi metodu find() s callback funkcijom
ocjenaVecaOdTri.

console.log(student); // Ispisuje {ime: "Ivo", prezime: "Ivić", ocjena: 5}
```

Rezultat je samo 1 objekt iako imamo 2 studenta s ocjenom većom od 3. Metoda `find()` vraća **prvi element** polja koji zadovoljava uvjet.

Varijante postoje, to su metode: `findIndex()`, `findLast()` i `findLastIndex()`.

Međutim ako želimo pronaći sve studente (ne samo prve ili zadnje) koji zadovoljavaju uvjet, moramo koristiti neke druge metode.

3.2.2 Metoda `forEach(callbackFn)`

Vidjeli smo već metodu `forEach()` koja prolazi kroz svaki element polja i izvršava `callback` funkciju za svaki element. Međutim, metoda `forEach()` ne vraća ništa, već samo prolazi kroz polje. Svejedno to možemo iskoristiti za pronađak svih studenata s ocjenom većom od 3. Važno je naglasiti da ova metoda ne vraća novo polje već samo prolazi kroz polje ili vrši neku operaciju nad elementima polja (modificira originalno polje).

```
let studenti = [
    {ime: "Ivo", prezime: "Ivić", ocjena: 5},
    {ime: "Ana", prezime: "Anić", ocjena: 4},
    {ime: "Maja", prezime: "Majić", ocjena: 3},
    {ime: "Ivan", prezime: "Ivanić", ocjena: 2},
    {ime: "Pero", prezime: "Perić", ocjena: 1},
];

let studentiPrekoTri = []; // Inicijaliziraj prazno polje za spremanje studenata s ocjenom većom od 3

studenti.forEach(function(student) { // Anonimna funkcija koja provjerava je li ocjena studenta veća od 3
    if (student.ocjena > 3) {
        studentiPrekoTri.push(student); // Dodaj studenta u polje studentiPrekoTri
    }
});

console.log(studentiPrekoTri); // Ispisuje [{ime: "Ivo", prezime: "Ivić", ocjena: 5}, {ime: "Ana", prezime: "Anić", ocjena: 4}]
```

Ako bi izvukli `callback` funkciju iz metode `forEach()` i definirali ju izvan metode, ona bi izgledala ovako:

```
function ocjenaVecaOdTri(student) {
    if (student.ocjena > 3) {
        studentiPrekoTri.push(student);
    }
}
```

I na ovaj način ju možemo koristiti kao `callback` funkciju za metodu `forEach()`.

```
let studentiPrekoTri = []; // Inicijaliziraj prazno polje za spremanje studenata s ocjenom većom od 3
studenti.forEach(ocjenaVecaOdTri); // Pozovi metodu forEach() s callback funkcijom ocjenaVecaOdTri

console.log(studentiPrekoTri); // Ispisuje [{ime: "Ivo", prezime: "Ivić", ocjena: 5}, {ime: "Ana", prezime: "Anić", ocjena: 4}]
```

3.2.3 Metoda `filter(callbackFn)`

U prethodnom primjeru koristili smo metodu `forEach()` za prolazak kroz polje i filtriranje studenata s ocjenom većom od 3. Međutim, postoji metoda `filter()` koja radi upravo to - filtrira elemente polja prema zadanim kriterijima.

Metoda `filter()` vraća **novo polje** s elementima koji zadovoljavaju uvjet definiran u `callback` funkciji.

Sintaksa: `filter(callbackFn, thisArg)` - `thisArg` je opcionalni argument koji predstavlja vrijednost `this` u `callback` funkciji.

```
let studenti = [
  {ime: "Ivo", prezime: "Ivić", ocjena: 5},
  {ime: "Ana", prezime: "Anić", ocjena: 4},
  {ime: "Maja", prezime: "Majić", ocjena: 3},
  {ime: "Ivan", prezime: "Ivanić", ocjena: 2},
  {ime: "Pero", prezime: "Perić", ocjena: 1},
];

let studentiPrekoTri = studenti.filter(function(student) { // Anonimna funkcija koja provjerava je li ocjena studenta veća od 3
  return student.ocjena > 3;
});
console.log(studentiPrekoTri); // Ispisuje [{ime: "Ivo", prezime: "Ivić", ocjena: 5},
{ime: "Ana", prezime: "Anić", ocjena: 4}]
```

Ili koristeći globalno definiranu `callback` funkciju:

```
function ocjenaVecaOdTri(student) {
  return student.ocjena > 3;
}
let studentiPrekoTri = studenti.filter(ocjenaVecaOdTri); // Pozovi metodu filter() s callback funkcijom ocjenaVecaOdTri
console.log(studentiPrekoTri); // Ispisuje [{ime: "Ivo", prezime: "Ivić", ocjena: 5},
{ime: "Ana", prezime: "Anić", ocjena: 4}]
```

To je to! Metoda `filter()` je korisna za filtriranje polja prema zadanim kriterijima.

Primjer 1: Tražilica

EduCoder šifra: `trazilica`

Na web stranicama trgovina, često se koristi tražilica koja omogućuje korisnicima pretraživanje proizvoda upisivanjem ključnih riječi ili same riječi proizvoda. Na primjer, korisnik može upisati "mobitel" i dobiti sve proizvode koji sadrže riječ "mobitel" u nazivu. Neke bolje tražilice omogućuju i pretraživanje po cijeni, kategoriji, brendu i sl.

U ovom primjeru ćemo implementirati jednostavnu tražilicu koja će **pretraživati proizvode samo po nazivu**.

Upotrijebit ćemo novo znanje o `callback` funkcijama i metodi `filter()`, kao i poznavanje ugniježđenih struktura.

1. korak je definirati polje objekata `proizvodi` koje sadrži proizvode s nazivom, cijenom i kategorijom.

```

let proizvodi = [
  {naziv: "Mobitel", cijena: 300, kategorija: "elektronika"},
  {naziv: "Slušalice", cijena: 20, kategorija: "elektronika"},
  {naziv: "Punjač", cijena: 10, kategorija: "elektronika"},
  {naziv: "Bicikl", cijena: 500, kategorija: "sport"},
  {naziv: "Tricikl", cijena: 350, kategorija: "sport"},
  {naziv: "Tenisice", cijena: 100, kategorija: "sport"},
  {naziv: "Dres", cijena: 50, kategorija: "sport"},
];

```

Recimo da je naša trgovina vrlo raznolikog assortimana, dodat ćemo u polje `proizvodi` i proizvode iz kategorije `prehrana`.

```

proizvodi.push({naziv: "Jabuka", cijena: 1, kategorija: "prehrana"});
proizvodi.push({naziv: "Jogurt", cijena: 2, kategorija: "prehrana"});
proizvodi.push({naziv: "Mlijeko", cijena: 2, kategorija: "prehrana"});
proizvodi.push({naziv: "Kruh", cijena: 3, kategorija: "prehrana"});

```

2. korak - želimo definirati funkciju `pretraziProizvode()` koja će pretraživati proizvode po nazivu.

Funkcija će primati 2 argumenta: polje proizvoda i ključnu riječ za pretraživanje. Na primjer:

```

pretraziProizvode(proizvodi, "mob"); // Ispisuje [{naziv: "Mobitel", cijena: 300,
kategorija: "elektronika"}] // vraća polje s 1 elementom

pretraziProizvode(proizvodi, "ten"); // Ispisuje [{naziv: "Tenisice", cijena: 100,
kategorija: "sport"}] // vraća polje s 1 elementom

pretraziProizvode(proizvodi, "J"); // Ispisuje [{naziv: "Punjač", cijena: 10, kategorija:
"elektronika"}, {naziv: "Jabuka", cijena: 1, kategorija: "prehrana"}, {naziv: "Jogurt",
cijena: 2, kategorija: "prehrana"}, {naziv: "Mlijeko", cijena: 2, kategorija: "prehrana"}]
// vraća polje s 4 elementa

pretraziProizvode(proizvodi, "cikl"); // Ispisuje [{naziv: "Bicikl", cijena: 500,
kategorija: "sport"}, {naziv: "Tricikl", cijena: 350, kategorija: "sport"}] // vraća polje
s 2 elementa

```

Idemo definirati kostur funkcije `pretraziProizvode()`:

```

function pretraziProizvode(proizvodi, kljucnaRijec) {
  // Implementacija funkcije
}

```

Ideja je da koristimo metodu `filter()` za filtriranje proizvoda prema ključnoj riječi.

Kao rezultat želimo dobiti novo polje filtriranih proizvoda koji sadrže ključnu riječ u nazivu.

```

function pretraziProizvode(proizvodi, kljucnaRijec) {
    let filtriraniProizvodi = proizvodi.filter(function(proizvod) {
        // Implementacija anonimne callback funkcije koja provjerava je li ključna riječ
        // sadržana u nazivu proizvoda
    });
    return filtriraniProizvodi;
}

```

3. korak - implementacija `callback` funkcije koja provjerava je li ključna riječ sadržana u **nazivu proizvoda**.

```

function pretraziProizvode(proizvodi, kljucnaRijec) {
    let filtriraniProizvodi = proizvodi.filter(function(proizvod) {
        return proizvod.naziv.includes(kljucnaRijec); // Vraća true ako ključna riječ
        // sadrži naziv proizvoda
    });
    return filtriraniProizvodi;
}

```

Problem riješen! Sada možemo pretraživati proizvode po ključnoj riječi.

```
console.log(pretraziProizvode(proizvodi, "MOB")); // Ispisuje: ništa? – vraća prazno polje
```

Problem je što je naš korisnik zaboravio ugasiti Caps Lock 😢 Kako bi riješili ovaj problem, možemo koristiti metodu `toLowerCase()` koja će pretvoriti ključnu riječ u mala slova (normalizacija teksta).

```

function pretraziProizvode(proizvodi, kljucnaRijec) {
    let filtriraniProizvodi = proizvodi.filter(function(proizvod) {
        return proizvod.naziv.toLowerCase().includes(kljucnaRijec.toLowerCase()); // Vraća
        // true ako ključna riječ sadrži naziv proizvoda bez obzira na velika/mala slova
    });
    return filtriraniProizvodi;
}

```

Sada možemo pretraživati proizvode bez obzira na velika/mala slova.

```

console.log(pretraziProizvode(proizvodi, "MOB")); // Ispisuje: [{naziv: "Mobitel", cijena: 300, kategorija: "elektronika"}]

console.log(pretraziProizvode(proizvodi, "ten")); // Ispisuje: [{naziv: "Tenisice", cijena: 100, kategorija: "sport"}]

console.log(pretraziProizvode(proizvodi, "cikl")); // Ispisuje: [{naziv: "Bicikl", cijena: 500, kategorija: "sport"}, {naziv: "Tricikl", cijena: 350, kategorija: "sport"}]

```

Vježba 5

EduCoder šifra: samo_parni

Napišite funkciju `samoParni(brojevi)` koja prima polje brojeva i vraća novo polje koje sadrži samo parne brojeve iz polja `brojevi`. Za implementaciju ne smijete koristiti petlje `for` ili `while`, već metodu `filter()` s odgovarajućom `callback` funkcijom.

Primjer poziva funkcije `samoParni()`:

```
let brojevi = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
console.log(samoParni(brojevi)); // Ispisuje: [2, 4, 6, 8, 10]
```

Vježba 6

EduCoder šifra: `filtriraj_osobe`

Dano vam je polje objekata koje predstavlja skup ljudi s njihovim imenima, godinama i zemljama iz kojih dolaze:

```
const osobe = [
  { ime: "Ana", godine: 22, zemlja: "Hrvatska" },
  { ime: "Marko", godine: 16, zemlja: "Slovenija" },
  { ime: "Ivan", godine: 35, zemlja: "Hrvatska" },
  { ime: "Maja", godine: 28, zemlja: "Bosna i Hercegovina" },
  { ime: "Eva", godine: 17, zemlja: "Slovenija" },
  { ime: "Tomislav", godine: 43, zemlja: "Hrvatska" }
];
```

Napišite funkciju `filtrirajOsobe(osobe, minGodine, zemlja)` koja prima polje `osobe`, minimalnu dob `minGodine` i zemlju `zemlja` te vraća novo polje koje sadrži samo osobe minimalne dobi i starije te iz zemlje `zemlja`. Za implementaciju koristite metodu `filter()` s odgovarajućom `callback` funkcijom.

Primjer poziva funkcije `filtrirajOsobe()`:

```
console.log(filtrirajOsobe(osobe, 18, "Hrvatska")); // Ispisuje: [{ ime: "Ana", godine: 22, zemlja: "Hrvatska" }, { ime: "Ivan", godine: 35, zemlja: "Hrvatska" }, { ime: "Tomislav", godine: 43, zemlja: "Hrvatska" }]
```

3.3 Arrow funkcije (`=>`)

U JavaScriptu, `arrow` funkcije predstavljaju kompaktnu alternativu tradicionalnim funkcionskim izrazima (eng. **function expressions**). `Arrow` funkcije su kratke i čitljive, a koriste se za **definiranje anonimnih funkcija**.

Arrow funkcije definiraju se koristeći sintaksu strelice `=>`. Međutim, osim sintakse, `arrow` funkcije imaju nekoliko značajki/ograničenja na koje treba obratiti pažnju:

- `arrow` funkcije nemaju vlastiti `this` kontekst, već nasleđuju `this` kontekst iz roditeljskog okruženja (**najvažnija značajka**).
- `arrow` funkcije ne vežu se na argumente `arguments` objekta.
- `arrow` funkcije ne mogu biti konstruirane pomoću `new` ključne riječi, tj. ne mogu biti korištene kao konstruktori.

- `arrow` funkcije se ne mogu koristiti kao generatori.

Kako izgledaju `arrow` funkcije u usporedbi s tradicionalnim funkcijama? U lekciji Funkcije, doseg varijabli i kontrolne strukture, precizirali smo razliku između `function` deklaracija i `function` izraza odnosno funkcijskih izraza.

Kako bismo jasno definirali sintaksu `arrow` funkcija, prisjetit ćemo se sintakse funkcijskih izraza i deklaracija.

3.3.1 Funkcijski izrazi i deklaracije

Rekli smo da su **deklaracije funkcije** definirane ključnom riječi `function` i imenom funkcije. Deklaracije funkcija mogu se koristiti prije nego što su deklarirane (koncept **hoisting**).

```
function zbroji(a, b) {
    return a + b;
}
console.log(zbroji(2, 3)); // Ispisuje 5
```

Deklaracijom klasičnih JavaScript funkcijskih izraza na neki način dodjeljujemo funkciju varijabli.

```
let zbroji = function(a, b) {
    return a + b;
}
console.log(zbroji(2, 3)); // Ispisuje 5
```

Kao drugu točku limitacije `arrow` funkcija rekli smo da ne poznaju/ne vežu se na `arguments` objekt. `arguments` objekt je lokalna varijabla funkcije koja sadrži sve argumente koje je funkcija primila.

Na primjeru funkcije `zbroji()` koja prima 2 argumenta, možemo koristiti `arguments` objekt za pristup argumentima funkcije (`a` i `b`).

```
function zbroji(a, b) {
    console.log(arguments); // Ispisuje [2, 3]
    console.log(arguments[0]) // Ispisuje 2
    console.log(arguments[1]) // Ispisuje 3
    return a + b;
}
console.log(zbroji(2, 3)); // Ispisuje 5
```

3.3.2 Sintaksa `arrow` funkcija

Arrow funkciju definirat ćemo koristeći sintaksu strelice `=>`. Sintakse `arrow` funkcija su sljedeće:

Sintaksa 1 (više parametra i blok naredbi): `(parametar1, parametar2, parametar3, parametarN)
=> {blok naredbi}`

Definiramo parametre u zagradama `()` i tijelo funkcije u vitičastim zagradama `{}`.

```
const imeFunkcije = (parametar1, parametar2, ..., parametarN) => {
    // Tijelo funkcije
}
```

Sintaksa 2 (jedan parametar i blok naredbi): `parametar => {blok naredbi}`

Međutim ako se funkcija sastoji samo od jednog parametra, možemo izostaviti zagrade oko parametara.

```
const imeFunkcije = parametar => {
    // Tijelo funkcije
}
```

Sintaksa 3 (više parametara i jedna naredba): `(parametar1, parametar2, parametar3, parametarN) => naredba`

```
const imeFunkcije = (parametar1, parametar2, ..., parametarN) => naredba;
```

Sintaksa 4 (jedan parametar i jedna naredba): `parametar => naredba`

Ako se funkcija sastoji samo od jedne naredbe, možemo izostaviti vitičaste zagrade `{}` i `return` ključnu riječ.

```
const imeFunkcije = parametar => naredba;
```

Sintaksa 5 (nema parametra i blok naredbi): `() => {blok naredbi}`

Ako funkcija ne prima parametre, koristimo prazne zagrade `()`.

```
const imeFunkcije = () => {
    // Tijelo funkcije
}
```

Sintaksa 6 (nema parametra i jedna naredba): `() => naredba`

```
const imeFunkcije = () => naredba;
```

Na prvi pogled sintakse `arrow` funkcija mogu izgledati zbunjujuće, ali s vježbom ćete se naviknuti na njih. Iako su iznad navedene različite sintakse `arrow` funkcija, ne morate ih i nećete učiti napamet. Bitno je razumjeti pravila sintakse i znati ih primijeniti ovisno o situaciji.

Pravila sintakse `arrow` funkcija su:

Ako `arrow` funkcija ima više parametara moramo ih definirati u zagradama `()`, inače ih možemo izostaviti.

Ako `arrow` funkcija ima više naredbi, moramo koristiti vitičaste zagrade `{}`.

Ako nam se funkcija sastoji samo od jedne naredbe, možemo izostaviti vitičaste zagrade `{}` i `return` ključnu riječ.

Ako se funkcija sastoji od više parametara i više naredbi, u pravilu ne koristimo arrow funkcije

3.3.3 Primjeri arrow funkcija

Primjer 1: arrow funkcija koja zbraja 2 broja

Za početak ćemo definirati arrow funkciju koja zbraja 2 broja, dakle ekvivalentno funkciji `zbroji()` koju smo definirali ranije.

```
// Deklaracija funkcije zbroji() koja zbraja 2 broja
function zbroji(a, b) {
    return a + b;
}
console.log(zbroji(2, 3)); // Ispisuje 5
```

Naša funkcija `zbroji` sastoji se od 2 parametra i jedne naredbe. Možemo definirati arrow funkciju koja zbraja 2 broja koristeći sintaksu 3.

```
// Arrow funkcija koja zbraja 2 broja
const zbroji = (a, b) => a + b;
console.log(zbroji(2, 3)); // Ispisuje 5
```

Primjer 2: arrow funkcija koja ispisuje pozdravnu poruku

Sada ćemo definirati arrow funkciju koja ispisuje pozdravnu poruku. Funkcija `pozdrav()` prima jedan parametar `ime` i ispisuje poruku "Pozdrav, ime!".

```
// Deklaracija funkcije pozdrav() koja ispisuje pozdravnu poruku
function pozdrav(ime) {
    console.log(`Pozdrav ${ime}!`);
}
pozdrav("Ana"); // Ispisuje "Pozdrav Ana!"
```

Naša funkcija `pozdrav` sastoji se od 1 parametra i jedne naredbe. Možemo definirati arrow funkciju koja ispisuje pozdravnu poruku koristeći sintaksu 4.

```
// Arrow funkcija koja ispisuje pozdravnu poruku
const pozdrav = ime => console.log(`Pozdrav ${ime}!`);
pozdrav("Ana"); // Ispisuje "Pozdrav Ana!"
```

Primjer 3: arrow funkcija koja kvadrira broj

Definirat ćemo arrow funkciju koja kvadrira broj. Funkcija `kvadriraj()` prima jedan parametar `broj` i vraća kvadrat tog broja.

```
// Deklaracija funkcije kvadriraj() koja kvadrira broj
function kvadriraj(broj) {
    return broj * broj;
}
console.log(kvadriraj(5)); // Ispisuje 25
```

Naša funkcija `kvadriraj` sastoji se od 1 parametra i jedne naredbe. Možemo definirati `arrow` funkciju koja kvadrira broj koristeći sintaksu 4.

```
let kvadriraj = broj => broj * broj;
console.log(kvadriraj(5)); // Ispisuje 25
```

Primjer 4: `arrow` funkcija bez parametara

Definirat ćemo funkciju koja recimo da inicijalizira našu aplikaciju. Funkcija `inicijaliziraj()` ne prima parametre i ispisuje poruku "Aplikacija inicijalizirana".

```
// Deklaracija funkcije inicijaliziraj() koja inicijalizira aplikaciju
function inicijaliziraj() {
    console.log("Aplikacija inicijalizirana");
}
inicijaliziraj(); // Ispisuje "Aplikacija inicijalizirana"
```

Naša funkcija `inicijaliziraj` ne prima parametre i sastoji se od jedne naredbe. Možemo definirati `arrow` funkciju koja inicijalizira aplikaciju koristeći sintaksu 5 ili 6.

```
let inicijaliziraj = () => console.log("Aplikacija inicijalizirana");
inicijaliziraj(); // Ispisuje "Aplikacija inicijalizirana"
```

`arrow` funkcije su uvijek anonimne, tj. nikada ih ne imenujemo. Međutim, možemo ih dodijeliti varijabli ili koristiti kao argument funkcije, kao što smo pokazali u primjerima iznad.

Sljedeći primjeri `arrow` funkcija su također ispravni. Jedina razlika je što ih ovdje ne pohranjujemo u varijable, poput funkcijskih izraza.

Ove funkcije su anonimne i koriste se kao callback funkcije, same po sebi se neće pozvati.

```
(a,b) => a + b;
```

```
() => console.log("Hello, World!");
```

`arrow` funkcije su korisne za definiranje jednostavnih funkcija koje se koriste kao callback funkcije.

3.3.4 `arrow` funkcije kao callback funkcije

Jedna od najčešćih primjena `arrow` funkcija je kao callback funkcije.

Važno je da do ovog trenutka razlikujete nekoliko pojmova:

- **callback funkcija** - funkcija koja se koristi kao argument druge funkcije.
- **anonimna funkcija** - funkcija koja nema ime.
- **arrow funkcija** - anonimna funkcija koja koristi sintaksu strelice `=>`.
- **funkcijski izraz** - funkcija koja se dodjeljuje varijabli (može biti anonimna, imenovana obična, arrow funkcija)
- **funkcijska deklaracija** - funkcija koja se deklarira ključnom riječi `function`
- **metoda** - funkcija koja je dio objekta

Ako vam neki od ovih pojmova nije jasan, ili vam se čini da ih miješate, preporuka je da se vratite na prethodne lekcije i ponovite gradivo koje vam stvara poteškoće.

Kako koristimo `arrow` funkcije kao callback funkcije? U prethodnim primjerima smo definirali `arrow` funkcije i pozivali ih direktno. Međutim, **vrlo često se koriste kao callback funkcije**.

Primjer 1: `arrow` funkcija kao callback funkcija u metodi `find()`

Vratimo se na primjer s poljem studenata. Definirali smo polje `studenti` i koristili metodu `find()` za pronađazak studenta s prezimenom `Ivanić`.

```
let studenti = [
  {ime: "Ivo", prezime: "Ivić", ocjena: 5},
  {ime: "Ana", prezime: "Anić", ocjena: 4},
  {ime: "Maja", prezime: "Majić", ocjena: 3},
  {ime: "Ivan", prezime: "Ivanić", ocjena: 2},
  {ime: "Pero", prezime: "Perić", ocjena: 1},
];
```

Koristili smo anonimnu funkciju kao `callback` funkciju za metodu `find()`.

```
let student = studenti.find(function(student) { // Anonimna funkcija koja provjerava je li
  prezime studenta jednako "Ivanić"
  return student.prezime == "Ivanić";
});

console.log(student); // Ispisuje {ime: "Ivan", prezime: "Ivanić", ocjena: 2}
```

Isto tako smo rekli da anonimnu callback funkciju možemo imenovati i definirati izvan metode `find()`.

```
function prezimeIvanić(student) {
  return student.prezime == "Ivanić";
}

let student = studenti.find(prezimeIvanić); // Pozovi metodu find() s callback funkcijom
// prezimeIvanić
console.log(student); // Ispisuje {ime: "Ivan", prezime: "Ivanić", ocjena: 2}
```

Napokon, evo kako bismo istu anonimnu callback funkciju definirali kao `arrow` funkciju.

```
let student = studenti.find(student => student.prezime == "Ivanić"); // Arrow funkcija  
koja provjerava je li prezime studenta jednako "Ivanić"  
console.log(student); // Ispisuje {ime: "Ivan", prezime: "Ivanić", ocjena: 2}
```

Kao što vidimo, `arrow` funkcija je jednostavnija i čitljivija od obične anonimne funkcije!

Kako možemo riješiti primjer s pronalaskom prvog studenta s ocjenom većom od 3 koristeći `arrow` funkciju?

Bez `arrow` funkcije rekli smo da bi to izgledalo ovako:

```
let studentiPrekoTri = studenti.find(function(student) { // Anonimna funkcija koja  
provjerava je li ocjena studenta veća od 3  
    return student.ocjena > 3;  
});  
  
//ili pak ovako:  
  
function ocjenaVecaOdTri(student) {  
    return student.ocjena > 3;  
}  
  
let studentiPrekoTri = studenti.find(ocjenaVecaOdTri); // Pozovi metodu find() s callback  
funkcijom ocjenaVecaOdTri
```

Kako bismo to riješili koristeći `arrow` funkciju?

```
let studentiPrekoTri = studenti.find(student => student.ocjena > 3); // Arrow funkcija  
koja provjerava je li ocjena studenta veća od 3  
console.log(studentiPrekoTri); // Ispisuje {ime: "Ivo", prezime: "Ivić", ocjena: 5}
```

Iz ovih primjera možete vidjeti snagu `arrow` funkcija, posebno u situacijama kada se koriste kao callback funkcije.

Svi primjeri koje smo pokazali s običnim funkcijama mogu se zamijeniti `arrow` funkcijama, a sintaksa postaje puno čišća i čitljivija, do te mjere da se da napisati u jednoj liniji kôda.

Primjer 2: `arrow` funkcija kao callback funkcija u metodi `filter()`

U primjeru s filtriranjem studenata s ocjenom većom od 3 koristili smo anonimnu funkciju kao `callback` funkciju za metodu `filter()`.

```
let studenti = [  
    {ime: "Ivo", prezime: "Ivić", ocjena: 5},  
    {ime: "Ana", prezime: "Anić", ocjena: 4},  
    {ime: "Maja", prezime: "Majić", ocjena: 3},  
    {ime: "Ivan", prezime: "Ivanić", ocjena: 2},  
    {ime: "Pero", prezime: "Perić", ocjena: 1},  
];
```

```

let studentiPrekoTri = studenti.filter(function(student) { // Anonimna funkcija koja
provjerava je li ocjena studenta veća od 3
    return student.ocjena > 3;
});

console.log(studentiPrekoTri); // Ispisuje [{ime: "Ivo", prezime: "Ivić", ocjena: 5},
{ime: "Ana", prezime: "Anić", ocjena: 4}]

```

Kako bismo to riješili koristeći `arrow` funkciju?

```

let studentiPrekoTri = studenti.filter(student => student.ocjena > 3); // Arrow funkcija
koja provjerava je li ocjena studenta veća od 3
console.log(studentiPrekoTri); // Ispisuje [{ime: "Ivo", prezime: "Ivić", ocjena: 5},
{ime: "Ana", prezime: "Anić", ocjena: 4}]

```

Ne moramo koristiti objekte unutar polja, jednako tako možemo koristiti `arrow` funkcije za filtriranje brojeva, nizova, stringova i sl.

Primjer s filtriranjem parnih brojeva koristeći `arrow` funkciju:

```

let brojevi = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
let parniBrojevi = brojevi.filter(broj => broj % 2 == 0); // Arrow funkcija koja
provjerava je li broj paran
console.log(parniBrojevi); // Ispisuje [2, 4, 6, 8, 10]

```

Ili primjer s filtriranjem stringova koji sadrže ključnu riječ "PJS":

```

let kolekcija_skripta = ["PJS_1", "OOP_2", "PJS_2", "SPA_3", "PIS_2", "PJS_4"];
let skripte_PJS = kolekcija_skripta.filter(skripta => skripta.includes("PJS")); // Arrow
funkcija koja provjerava sadrži li skripta ključnu riječ "PJS"
console.log(skripte_PJS); // Ispisuje ["PJS_1", "PJS_2", "PJS_4"]

```

ili

```

let kolekcija_skripta = ["PJS_1", "OOP_2", "PJS_2", "SPA_3", "PIS_2", "PJS_4"];
let skripte_PJS = kolekcija_skripta.filter(skripta => skripta.startsWith("PJS")); // Arrow
funkcija koja provjerava počinje li skripta s ključnom riječju "PJS"
console.log(skripte_PJS); // Ispisuje ["PJS_1", "PJS_2", "PJS_4"]

```

Primjer 2: Pronađi let

EduCoder Šifra: `skyscanner`

Napišite funkciju `pronadiLet()` koja prima polje letova, željeni grad polaska, željeni grad dolaska i datum polaska. Funkcija treba pronaći sve letove koji odgovaraju zadanim parametrima i vratiti ih kao novo polje. Morate koristiti metodu `filter()` s odgovarajućom callback funkcijom koja provjerava odgovara li let zadanim parametrima definiranom `arrow` funkcijom. Korisnik može izostaviti datum polaska, u tom slučaju funkcija treba pronaći sve letove koji odgovaraju zadanim gradovima.

Definirano je polje letova `letovi` koje sadrži objekte sa svojstvima `polazak`, `dolazak` i `datum`.

```
let letovi = [
    {polazak: "Zagreb", dolazak: "Split", datum: new Date("2024-08-01")},
    {polazak: "Zagreb", dolazak: "Split", datum: new Date("2024-09-05")},
    {polazak: "Zagreb", dolazak: "Dubrovnik", datum: new Date("2024-08-02")},
    {polazak: "Split", dolazak: "Zadar", datum: new Date("2024-06-03")},
    {polazak: "Dubrovnik", dolazak: "Osijek", datum: new Date("2024-10-04")},
    {polazak: "Dubrovnik", dolazak: "Osijek", datum: new Date("2024-10-22")},
    {polazak: "Pula", dolazak: "Zagreb", datum: new Date("2024-05-05")},
    {polazak: "Pula", dolazak: "Zagreb", datum: new Date("2024-05-11")},
    {polazak: "Zagreb", dolazak: "Pula", datum: new Date("2024-07-06")},
    {polazak: "Zadar", dolazak: "Zagreb", datum: new Date("2025-09-07")},
];

```

Krenimo odmah s definiranjem funkcije `pronadiLet()` koja prima polje letova, željeni grad polaska, željeni grad dolaska i datum polaska.

```
function pronadiLet(letovi, polazak, dolazak, datum = null) { // datum je opcionalan parametar
    // Implementacija funkcije
}
console.log(pronadiLet(letovi, "Zagreb", "Split")); // Ispisuje [{polazak: "Zagreb", dolazak: "Split", datum: new Date("2024-08-01")}, {polazak: "Zagreb", dolazak: "Split", datum: new Date("2024-08-05")}]
```

Koristit ćemo metodu `filter()` za filtriranje letova prema zadanim parametrima.

```
function pronadiLet(letovi, polazak, dolazak, datum = null) {
    let filtriraniLetovi = letovi.filter(let => {
        // Implementacija arrow funkcije koja provjerava odgovara li let zadanim parametrima
    });
}
```

Provjerit ćemo prvo podudaraju li se gradovi polaska i dolaska leta s zadanim gradovima.

```
function pronadiLet(letovi, polazak, dolazak, datum = null) {
    let filtriraniLetovi = letovi.filter(let => {
        return let.polazak == polazak && let.dolazak == dolazak;
    });
}
```

Za datum možemo provjeriti je li datum leta jednak zadatom datumu. Ako je datum `null`, znači da korisnik nije unio datum polaska, u tom slučaju vraćamo sve letove koji odgovaraju zadanim gradovima.

```

function pronadiLet(letovi, polazak, dolazak, datum = null) {
    let filtriraniLetovi = letovi.filter(let => {
        return let.polazak == polazak && let.dolazak == dolazak && (datum == null || let.datum == datum);
    });
}

```

Ne moramo ovo pohranjivati u varijablu `filtriraniLetovi`, već možemo odmah vratiti rezultat.

```

function pronadiLet(letovi, polazak, dolazak, datum = null) {
    return letovi.filter(let => {
        return let.polazak == polazak && let.dolazak == dolazak && (datum == null || let.datum.getTime() === datum.getTime()); // koristimo metodu Date.getTime() budući da smo rekli da objekte ne možemo direktno uspoređivati.
    });
}

```

Sada možemo testirati funkciju `pronadiLet()`.

```

console.log(pronadiLet(letovi, "Zagreb", "Split")); // Ispisuje [{polazak: "Zagreb", dolazak: "Split", datum: new Date("2024-08-01")}, {polazak: "Zagreb", dolazak: "Split", datum: new Date("2024-08-05")}]  
  

console.log(pronadiLet(letovi, "Dubrovnik", "Osijek")); // Ispisuje [{polazak: "Dubrovnik", dolazak: "Osijek", datum: new Date("2024-10-04")}, {polazak: "Dubrovnik", dolazak: "Osijek", datum: new Date("2024-10-22")}]  
  

console.log(pronadiLet(letovi, "Zadar", "Zagreb", new Date("2025-09-07"))); // Ispisuje [{polazak: "Zadar", dolazak: "Zagreb", datum: new Date("2025-09-07")}]
```

Vježba 7

EduCoder šifra: `arrows`

Za dane deklaracije funkcija napišite ekvivalentne funkcijeske izraze koristeći `arrow` funkcije.

```

function hello(name) {
    return `Hello, ${name}!`;
}
const hello = /* arrow funkcija */;
```

```

function getFullName(firstName, lastName) {
    const fullName = `${firstName} ${lastName}`;
    return fullName;
}
const getFullName = /* arrow funkcija */;
```

```

function multiplyThenAdd(a, b, c) {
    const result = a * b + c;
    return result;
}
const multiplyThenAdd = /* arrow funkcija */;

```

```

let five = 5;
function fiveEven() {
    return five % 2 == 0;
}
const isEven = /* arrow funkcija */;

```

Vježba 8

EduCoder šifra: `idealni_zaposlenik`

Imate šefa koji želi zaposliti idealnog zaposlenika za svoju tvrtku, međutim na natječaj se javilo previše kandidata. Šef vas je zamolio da mu pomognete pronaći idealnog zaposlenika. Vi kao vrsni poznavatelji JavaScripta odlučili ste mu pomoći. Budući da šef ne zna programirati, a vama se neda ručno pregledavati sve prijave, odlučili ste napisati funkciju `idealni_zaposlenik()` koja će vratiti idealnog kandidata.

Šef ima svoje kriterije za idealnog zaposlenika te jedva čeka upotrijebiti vašu funkciju. Sve što vam je dao jest polje objekata `kandidati` koje sadrži informacije o kandidatima. Svaki kandidat ima svojstva: `ime`, `godine`, `godine_iskustva`, `strani_jezici`, `programske_jezici`.

```

let kandidati = [
    {ime: "Ana", godine: 25, godine_iskustva: 3, strani_jezici: ["engleski", "njemački"], programske_jezici: ["JavaScript", "Python"]},
    {ime: "Ivan", godine: 30, godine_iskustva: 5, strani_jezici: ["engleski", "francuski"], programske_jezici: ["JavaScript", "Java"]},
    {ime: "Maja", godine: 22, godine_iskustva: 1, strani_jezici: ["engleski", "njemački"], programske_jezici: ["JavaScript", "Python"]},
    {ime: "Marko", godine: 35, godine_iskustva: 7, strani_jezici: ["engleski", "njemački", "francuski"], programske_jezici: ["JavaScript", "Python", "Java"]},
    {ime: "Eva", godine: 28, godine_iskustva: 4, strani_jezici: ["engleski", "njemački"], programske_jezici: ["JavaScript", "Python"]},
    {ime: "Tomislav", godine: 40, godine_iskustva: 10, strani_jezici: ["engleski", "njemački", "francuski", "španjolski"], programske_jezici: ["JavaScript", "Python", "Java", "C++"]},
    {ime: "Lucija", godine: 26, godine_iskustva: 3, strani_jezici: ["engleski", "španjolski"], programske_jezici: ["Python", "R"]},
    {ime: "Dario", godine: 31, godine_iskustva: 6, strani_jezici: ["engleski", "ruski"], programske_jezici: ["C#", "Java", "Python"]},
    {ime: "Petra", godine: 29, godine_iskustva: 5, strani_jezici: ["engleski", "talijanski", "francuski"], programske_jezici: ["JavaScript", "Swift"]},
    {ime: "Nikola", godine: 32, godine_iskustva: 8, strani_jezici: ["engleski", "njemački"], programske_jezici: ["JavaScript", "Java", "Scala"]},
    {ime: "Lara", godine: 24, godine_iskustva: 2, strani_jezici: ["engleski", "kineski"], programske_jezici: ["Python", "JavaScript"]},
]

```

```
{ime: "Jakov", godine: 33, godine_iskustva: 9, strani_jezici: ["engleski",  
"španjolski", "portugalski"], programski_jezici: ["Java", "JavaScript", "Go"]},  
];
```

Napišite funkciju `idealni_zaposlenik(kandidati, godine, godine_iskustva, strani_jezici, programski_jezici)` koja prima navedene argumente te koristi metodu `Array.filter()` za filtriranje kandidata prvo prema stranim jezicima i programskim jezicima, a zatim koristi metodu `Array.find()` za prvog kandidata **koji ima barem** zadane godine i godine iskustva. Dakle tražite prvog kandidata, ne onog koji ima najviše godina i ili godina iskustva, već prvog koji zadovoljava sve uvjete.

Za provjeru jezika možete koristiti petlje i metode `Array.includes()`. Ako znate, možete koristiti i metodu `Array.every()`.

Morate koristiti `arrow` funkcije za definiranje callback funkcija u metodama `filter()` i `find()`.

```
let sretnik = idealni_zaposlenik(kandidati, 25, 3, ["engleski"], ["JavaScript",  
"Python"]);  
console.log(sretnik); // Ispisuje {ime: "Ana", godine: 25, godine_iskustva: 3,  
strani_jezici: ["engleski", "njemački"], programski_jezici: ["JavaScript", "Python"]}  
  
sretnik = pronadiZaposlenika(kandidati, 35, 3, ["engleski"], ["JavaScript", "Python"]);  
console.log(sretnik); // Ispisuje {ime: "Marko", godine: 35, godine_iskustva: 7,  
strani_jezici: ["engleski", "njemački", "francuski"], programski_jezici: ["JavaScript",  
"Python", "Java"]}
```

3.3.5 arrow funkcije i this kontekst

Kada smo radili objekte, naučili smo da svaki objekt ima svoj `this` kontekst. Općenito, `this` kontekst se odnosi se na kontekst gdje se izvršava određeni dio koda.

Najčešće je to kod objekata, gdje se `this` ključnom riječi referencira na svojstvo ili metodu unutar objekta.

Vrijednost `this` ovisi o kontekstu u kojem se pojavljuje: globalni `this` kontekst, `this` kontekst unutar funkcije, `this` kontekst unutar metode objekta, `this` kontekst unutar `arrow` funkcije itd.

Naučili smo da kod običnih funkcija (ne `arrow` funkcija) vrijednost `this` predstavlja objekt koji poziva funkciju.

Drugim riječima, ako je poziv funkcije (metode) u obliku: `objekt.metoda()`, tada će `this` referencirati na `objekt`.

```
const osoba = {  
    ime: "Ana",  
    pozdrav: function() {  
        console.log(`Pozdrav, ${this.ime}!`); // this se referencira na objekt osoba  
    }  
};  
console.log(osoba.pozdrav()); // Ispisuje "Pozdrav, Ana!"
```

Pokazali smo upotrebu `this` ključne riječi i u kontekstu definiranja konstruktora.

```

function Osoba(ime) {
    this.ime = ime;
    this.pozdrav = function() {
        console.log(`Pozdrav, ${this.ime}!`); // this se referencira na objekt koji se stvara
    }
}
let osoba = new Osoba("Ana");
console.log(osoba.pozdrav()); // Ispisuje "Pozdrav, Ana!"

```

Što se tiče `this` konteksta i tradicionalnih funkcija, `this` ključna riječ se mijenja ovisno o kontekstu u kojem se funkcija poziva. Generalno se `this` ključna riječ odnosi na objekt koji poziva funkciju, međutim postoje različita ponašanja kada se koriste metode poput `call()`, `apply()` i `bind()`. Mi se time nećemo baviti na ovom kolegiju.

Ono što vi morate zapamtitи jest da, kod `arrow` funkcija vrijednost `this` se ne mijenja, **već se nasljeđuje iz okoline u kojoj je definirana**. Drugim riječima, ključna riječ `this` u `arrow` funkcijama referencira na `this` kontekst izvan `arrow` funkcije.

Dakle kod stvaranja našeg objekta `osoba` i metode `pozdrav` koristeći `arrow` funkciju, `this` ključna riječ će se referencirati na objekt `osoba`.

```

// Konstruktor funkcija - možemo koristiti arrow funkciju
function Osoba(ime) {
    this.ime = ime;
    this.pozdrav = () => {
        console.log(`Pozdrav, ${this.ime}!`); // this se i ovdje referencira na objekt koji se stvara
    }
}
let osoba = new Osoba("Ana");
osoba.pozdrav(); // Ispisuje "Pozdrav, Ana!"

```

Međutim gdje dolazi do problema je kada koristimo `arrow` funkcije unutar metoda objekta.

```

// Objekt osoba s metodom pozdrav() koja neispravno koristi arrow funkciju
const osoba = {
    ime: "Ana",
    pozdrav() => {
        console.log(`Pozdrav, ${this.ime}!`); // this se ne referencira na objekt osoba,
        već na globalni objekt (u web pregledniku je to window)
    }
};
console.log(objekt.pozdrav()); // Ispisuje "Pozdrav, undefined!"

```

Razlika u ova dva pristupa je u tome što kod tradicionalnih funkcija `this` ključna riječ se mijenja ovisno o kontekstu u kojem se funkcija poziva, dok kod `arrow` funkcija `this` ključna riječ se nasljeđuje iz okoline u kojoj je definirana.

VAŽNO! U prvom primjeru kod stvaranja konstruktora, `this` ključna riječ se referencira na objekt koji se stvara, dok u drugom primjeru kod metode objekta, `this` ključna riječ se referencira na globalni objekt (u web pregledniku je to `window`).

Na predavanjima ste naučili koristiti HTML elemente i dodavati im event listener. **Event listeneri** su funkcije koje se pozivaju kada se dogodi određeni događaj na HTML elementu.

Pokazat ćemo tradicionalni način dodavanja event listenera na HTML element.

Želimo na `button` element dodati event listener koji će ispisati "Hello, World!" kada se klikne na gumb.

```
<button id="moj_button">Klikni me!</button>
```

```
document.getElementById("moj_button").addEventListener("click", function() {
    console.log("Hello, World!");
    console.log(this) // this se referencira na HTML element koji je kliknut (u ovom
    slučaju na gumb: moj_button)
    console.log(this.id); // Ispisuje "moj_button"
});
```

Međutim ako koristimo `arrow` funkciju kao callback funkciju, `this` ključna riječ će se referencirati na globalni objekt (u web pregledniku je to `window`).

```
document.getElementById("moj_button").addEventListener("click", () => {
    console.log("Hello, World!");
    console.log(this); // this se referencira na globalni objekt (u web pregledniku je to
    window)
    console.log(this.id); // Ispisuje "undefined"
});
```

Zaključno: važno je zapamtiti da kada se koriste metode s objektima, ako vam je potreban `this` za referenciranje na objekt, koristite tradicionalne funkcije. Ako vam `this` nije potreban, koristite `arrow` funkcije.

Arrow funkcije su najkorisnije kada se koriste kao callback funkcije, jer se `this` ne veže na samu funkciju već možete povući `this` iz okoline u kojoj je definirana, što je najčešće i potrebno.

3.4 Napredne metode Array objekta

U skripti PJS3 upoznali smo se s osnovnim metodama `Array` objekta. U ovoj skripti produljili smo priču s nešto naprednjim metodama, poput metode `find()` koja pronalazi prvi element koji zadovoljava uvjet, te metode `filter()` koja filtrira elemente prema zadanom uvjetu.

Također smo se kroz skriptu upoznali s `callback` funkcijama kao i `arrow` funkcijama koje su korisne kao callback funkcije. Sada kada znamo kako pisati kvalitetne `callback` funkcije, možemo napokon pokazati preostale napredne metode `Array` objekta.

Sve metode `Array` objekta (one koje smo do sad prošli pa i ove ispod) možemo koristiti i s `arrow` funkcijama, ali i bez - koristeći anonimne callback funkcije ili imenovane callback funkcije.

3.4.1 Metoda map()

Metoda `Array.map()` koristi se za stvaranje novog polja na temelju polja nad kojim se poziva. Metoda `map()` prima callback funkciju koja se poziva za svaki element polja.

Drugim riječima, metoda `map()` prolazi kroz svaki element polja i poziva callback funkciju za svaki element.
Rezultat metode je novo polje.

Sintaksa:

```
map(callbackFn)
map(callbackFn, thisArg)

const newArray = array.map(function(element, index, array) {
  // povratna vrijednost je NOVO POLJE
});

```

- `callbackFn` - funkcija koja se poziva za svaki element polja. Funkcija prima tri argumenta: `element`, `index` i `array`. Podsjetimo se prethodne skripte, `element` je trenutni element polja, `index` je indeks trenutnog elementa, a `array` je polje nad kojim se metoda poziva.
- `thisArg` - opcionalni argument koji predstavlja vrijednost `this` ključne riječi unutar callback funkcije.

Primjer 1: Stvaranje novog polja s kvadratima brojeva

```
let brojevi = [1, 2, 3, 4, 5];
let kvadrati = brojevi.map(broj => broj * broj); // Koristimo arrow callback funkciju koja
vraća kvadrat broja
console.log(kvadrati); // Ispisuje [1, 4, 9, 16, 25]
```

ili bez korištenja `arrow` funkcije:

```
let brojevi = [1, 2, 3, 4, 5];
let kvadrati = brojevi.map(function(broj) {
  return broj * broj;
});
console.log(kvadrati); // Ispisuje [1, 4, 9, 16, 25]
```

Primjer 2: Stvaranje novog polja s duljinama stringova

```
let imena = ["Ana", "Ivan", "Maja", "Pero"];
let duljine = imena.map(ime => ime.length); // Koristimo arrow callback funkciju koja
vraća duljinu stringa
console.log(duljine); // Ispisuje [3, 4, 4, 4]
```

Primjer 3: Stvaranje novog polja s objektima

```

let imena = ["Ana", "Ivan", "Maja", "Pero"];
let objekti = imena.map(ime => ({ime: ime})); // Koristimo arrow callback funkciju koja
vraća objekt s imenom
console.log(objekti); // Ispisuje [{ime: "Ana"}, {ime: "Ivan"}, {ime: "Maja"}, {ime:
"Pero"}]

```

Važno! Ako nemamo potrebu za korištenjem novih polja, već želimo promijeniti elemente u polju, samo ih ispisati ili drugo, koristimo metodu `forEach()` ili petlju `for`, a ne metodu `map()`.

Dakle sljedeće nema smisla:

```

let brojevi = [1, 2, 3, 4, 5];
let kvadrati = brojevi.map(broj => {
    console.log(broj * broj);
    return broj * broj;
});

```

Za primjer iznad, koristit ćemo metodu `forEach()`.

```

let brojevi = [1, 2, 3, 4, 5];
brojevi.forEach(broj => {
    console.log(broj * broj);
});

```

Primjer 4: Dodavanje indeksa elementima polja

```

let imena = ["Ana", "Ivan", "Maja", "Pero"];

let imenaIndeksi = imena.map((ime, index) => `${ime}_${index+1}`); // Koristimo arrow
callback funkciju koja vraća ime s indeksom
console.log(imenaIndeksi); // Ispisuje ["Ana_1", "Ivan_2", "Maja_3", "Pero_4"]

```

Primjer 5: Množenje elemenata obzirom na njihovu poziciju

```

let brojevi = [5, 10, 15, 20];
let mnozeno = brojevi.map((element, index) => {
    if (index % 2 == 0) { // Ako je indeks paran množimo s 2,
        return element * 2;
    } else { // inače množimo s 3
        return element * 3;
    }
})
console.log(mnozeno); // Ispisuje [10, 30, 30, 60]

```

3.4.2 Metoda `some()`

Metoda `Array.some()` koristi se za provjeru postoji li **barem jedan element u polju koji zadovoljava uvjet**. Metoda `some()` vraća `true` ako postoji barem jedan element koji zadovoljava uvjet, inače vraća `false`. Uvjet se definira callback funkcijom. Metoda ne mijenja originalno polje.

Sintaksa:

```
some(callbackFn)
some(callbackFn, thisArg)
```

Pravila su slična kao kod metode `filter()`. `callbackFn` je funkcija koja se poziva za svaki element polja. Funkcija prima tri argumenta: `element`, `index` i `array`. `thisArg` je optionalni argument koji predstavlja vrijednost `this` ključne riječi unutar callback funkcije.

Za razliku od metode `map` ova metoda ne stvara novo polje, već vraća `true` ili `false` ovisno o tome postoji li barem jedan element koji zadovoljava uvjet.

Primjer 1: Provjera postoji li barem jedan element veći od 10

```
let brojevi = [5, 10, 15, 20];
let postojiVeciOdDeset = brojevi.some(broj => broj > 10); // Koristimo arrow callback
funkciju koja provjerava je li broj veći od 10
console.log(postojiVeciOdDeset); // Ispisuje true
```

Primjer 2: Provjera postoji li barem jedan element koji sadrži ključnu riječ "PJS"

```
console.log(["PJS_1", "OOP_2", "PJS_2", "SPA_3", "PIS_2", "PJS_4"].some(skripta =>
skripta.includes("PJS"))); // Ispisuje true
```

Primjer 3: Provjera postoji li barem jedan element koji je paran

```
console.log([7, 11, 21, 5, 5].some(broj => broj % 2 == 0)); // Ispisuje false jer nema
parnih brojeva
```

Primjer 4: Provjera postoji li barem jedan element koji je manji od 0

```
function isSmallerThanZero(broj) {
    return broj < 0;
}
console.log([1, 2, 3, 4, 5].some(isSmallerThanZero)); // Ispisuje false jer nema
negativnih brojeva
```

Primjer 5: Provjerava postoji li barem jedan traženi element na zalihi

```

let zaliha = [
    {naziv: "jabuka", kolicina: 10},
    {naziv: "kruška", kolicina: 5},
    {naziv: "banana", kolicina: 0},
    {naziv: "naranča", kolicina: 15},
    {naziv: "limun", kolicina: 0}
];
function provjeriZalihe(naziv) {
    return zaliha.some(voce => voce.naziv === naziv && voce.kolicina > 0);
}
console.log(provjeriZalihe("banana")); // Ispisuje false jer nema banana na zalihi
console.log(provjeriZalihe("naranča")); // Ispisuje true jer ima naranči na zalihi

```

Da ne bi bilo nejasnoća, pokazat ćemo i bez `arrow` funkcije.

```

function provjeriZalihe(naziv) {
    return zaliha.some(function(voce) {
        return voce.naziv === naziv && voce.kolicina > 0;
    });
}
console.log(provjeriZalihe("banana")); // Ispisuje false jer nema banana na zalihi
console.log(provjeriZalihe("naranča")); // Ispisuje true jer ima naranči na zalihi

```

3.4.3 Metoda `every()`

Metoda `Array.every()` koristi se za provjeru jesu li svi elementi u polju zadovoljavaju uvjet. Metoda `every()` vraća `true` ako svi elementi zadovoljavaju uvjet, inače vraća `false`. Uvjet se definira callback funkcijom. Metoda ne mijenja originalno polje.

Dakle metoda je srodnja metodi `Array.some()`, međutim `every()` (kako joj i sam naziv kaže) vraća `true` **samo ako svi elementi zadovoljavaju uvjet**.

Sintaksa:

```

every(callbackFn)
every(callbackFn, thisArg)

```

Pravila su slična kao kod metode `some()`. `callbackFn` je funkcija koja se poziva za svaki element polja. Funkcija prima tri argumenta: `element`, `index` i `array`. `thisArg` je optionalni argument koji predstavlja vrijednost `this` ključne riječi unutar callback funkcije.

Primjer 1: Provjera jesu li svi elementi veći od 10

```

let brojevi = [15, 20, 25, 30];
let sviVeciOdDeset = brojevi.every(broj => broj > 10); // Koristimo arrow callback
// funkciju koja provjerava je li broj veći od 10
console.log(sviVeciOdDeset); // Ispisuje true

```

Primjer 2: Provjera je li svaki element paran broj

```
console.log([2, 4, 6, 8, 10].every(broj => broj % 2 == 0)); // Ispisuje true jer su svi  
brojevi parni
```

Primjer 3: Provjera je li jedno polje podskup drugog polja

```
const jePodskup = (polje1, polje2) =>  
    polje1.every((element) => polje2.includes(element));  
console.log(jePodskup([1, 2, 3], [1, 2, 3, 4, 5])); // Ispisuje true jer je [1, 2, 3]  
podskup [1, 2, 3, 4, 5]  
console.log(jePodskup([1, 2, 3], [1, 2, 4, 5])); // Ispisuje false jer [1, 2, 3] nije  
podskup [1, 2, 4, 5]
```

Primjer 4: Provjerava je li svaki element veći od prethodnog

```
let brojevi = [1, 2, 3, 4, 5];  
let sviVeciOdPrethodnog = brojevi.every((element, index, array) => {  
    if (index === 0) { // Ako je prvi element, preskoči  
        return true;  
    }  
    return element > array[index - 1]; // Provjerava je li trenutni element veći od  
    prethodnog  
});  
console.log(sviVeciOdPrethodnog); // Ispisuje true jer su svi elementi veći od prethodnog
```

3.4.4 Metoda `sort()`

Metoda `Array.sort()` koristi se za sortiranje elemenata u polju. Metoda `sort()` mijenja originalno polje, odnosno radi `in place`. Kao povratnu vrijednost vraća sortirano polje.

Prema defaultu, sortiranje je ascending (rastuće), odnosno sortira elemente od najmanjeg prema najvećem. Međutim, možemo definirati vlastitu funkciju za sortiranje koju ćemo definirati u callback funkciji.

Vremenska i prostora složenost metode `sort()` se ne može garantirati budući da ovisi o implementaciji JavaScript enginea.

Postoji varijanta ove metode koja ne mijenja originalno polje, a to je metoda `Array.toSorted()`.

Sintaksa:

```
sort()  
sort(compareFn)
```

- `compareFn` - funkcija koja se koristi za sortiranje elemenata. Funkcija prima dva argumenta `a` i `b` koji predstavljaju dva elementa koja se uspoređuju.
- Ako je rezultat negativan, `a` se smješta prije `b`,
- ako je rezultat pozitivan, `b` se smješta prije `a`,
- ako je rezultat 0, elementi ostaju na istom mjestu (a i b su jednaki).

Algoritam sortiranja u JavaScriptu ovisi o implementacija, najčešće se koriste algoritmi poput QuickSort i MergeSort.

Dakle `compareFn` callback funkciju možemo definirati sljedećom sintaksom:

```
function compareFn(a, b) {
  if (a manji od b po nekom kriteriju) {
    return -1;
  } else if (a veći od b po nekom kriteriju) {
    return 1;
  }
  // a i b su jednaki
  return 0;
}
```

Primjer 1: Sortiranje brojeva u rastućem redoslijedu

```
let brojevi = [5, 2, 8, 1, 3];
brojevi.sort((a, b) => a - b); // Sortira brojeve u rastućem redoslijedu
console.log(brojevi); // Ispisuje [1, 2, 3, 5, 8]
```

Zašto pišemo našu callback funkciju kao `a-b`?

- Ako je rezultat negativan, `a` se smješta prije `b`,
- ako je rezultat pozitivan, `b` se smješta prije `a`,
- ako je rezultat 0, elementi ostaju na istom mjestu (a i b su jednaki).

Primjer 2: Sortiranje stringova po duljini

```
let imena = ["Ana", "Ivan", "Maja", "Pero", "Aleksandar", "Marko", "Eva", "Tomislav"];
imena.sort((a, b) => a.length - b.length); // Sortira stringove po duljini
console.log(imena); // Ispisuje ["Ana", "Eva", "Ivan", "Maja", "Pero", "Marko",
"Tomislav", "Aleksandar"]
```

Primjer 3: Sortiranje objekata

```
let osobe = [
  {ime: "Ana", godine: 25},
  {ime: "Ivan", godine: 30},
  {ime: "Maja", godine: 22},
  {ime: "Marko", godine: 35},
  {ime: "Eva", godine: 28},
  {ime: "Tomislav", godine: 40},
  {ime: "Lucija", godine: 26},
  {ime: "Dario", godine: 31},
  {ime: "Petra", godine: 29},
  {ime: "Nikola", godine: 32},
  {ime: "Lara", godine: 24},
  {ime: "Jakov", godine: 33}]
```

```

];
// sortiranje po godinama
osobe.sort((a, b) => a.godine - b.godine);

// sortiranje po imenu

osobe.sort((a, b) => {
  const imeA = a.ime.toLowerCase(); // pretvaramo imena u mala slova
  const imeB = b.ime.toLowerCase(); // pretvaramo imena u mala slova

  if (imeA < imeB) {
    return -1;
  } else if (imeA > imeB) {
    return 1;
  }

  // imena su jednaka
  return 0;
});

```

3.4.5 Metoda `reduce()`

Metoda `Array.reduce()` koristi se za reduciranje polja u jednu vrijednost. Metoda `reduce()` prima callback funkciju koja se poziva za svaki element polja. Funkcija prima četiri argumenta: `accumulator`, `currentValue`, `currentIndex` i `array`.

Kod ove metode, `callback` funkcija je na neki način `reducer` funkcija koja akumulira vrijednosti (svaki element polja) u jednu vrijednost.

Konačna vrijednost `reduce` funkcije je uvijek jedna vrijednost, a ne polje!

Premda ova metoda mnogima zadaje glavobolju, kada jednom shvatite kako radi, postaje vrlo korisna. Da bi se shvatila, potrebno je prvo uvidjeti njenu korist.

Sjetite se primjera gdje smo nad svakim elementom polja zbrajali neku vrijednost (zadatak s košaricom i artiklima).

Zadatak je izgledao ovako:

```

function Namirnica(naziv, cijena, kolicina) {
  this.naziv = naziv;
  this.cijena = cijena;
  this.kolicina = kolicina;
  this.ukupno = function() {
    return this.cijena * this.kolicina;
  }
}

let kosarica = [
  new Namirnica("Jabuka", 2, 3),
  new Namirnica("Kruška", 3, 2),

```

```

    new Namirnica("Banana", 1, 5),
    new Namirnica("Naranča", 4, 1)
};

// globalna funkcija koja računa ukupnu cijenu

function ukupnaCijena(kosarica) {
    let ukupno = 0;
    for(let namirnica of kosarica) {
        ukupno += namirnica.ukupno();
    }
    return ukupno;
}

console.log(ukupnaCijena(kosarica)); // Ispisuje 21

```

Uočavate li gdje bi mogli koristiti metodu `reduce()`? Kod računanja ukupne cijene, svaki put smo zbrajali `ukupno` s `namirnica.ukupno()`.

Ako pogledate ponovo definiciju metode `reduce()`, vidjet ćete da je upravo to ono što nam treba - **zbrajanje svih elemenata polja u jednu vrijednost**.

Sintaksa:

```

reduce(callbackFn)
reduce(callbackFn, initialValue)

```

Naizgled jednostavna sintaksa, međutim komplikiraniji dio leži u samoj definiciji `callback` funkcije.

Callback funkcija prima četiri argumenta: `accumulator`, `currentValue`, `currentIndex` i `array`.

- `initialValue` predstavlja početnu vrijednost accumulator-a. Ako je definirana, prvi poziv funkcije koristi `initialValue` kao `accumulator`, inače koristi prvi element polja.
- `accumulator` - **akumulator**, početna vrijednost je `initialValue` (ako je definirana), inače je prvi element polja. Akumulator je vrijednost koja se akumulira tijekom izvođenja funkcije.
- `currentValue` - **vrijednost trenutnog elementa polja**. Ako je `initialValue` definiran, vrijednost `currentValue` je prvi element polja (`array[0]`), inače je drugi element polja (`array[1]`).
- `currentIndex` - **indeks pozicija trenutnog elementa polja**. Prvim pozivom vrijednost je 0 ako je `initialValue` definiran, inače je 1.
- `array` - **polje nad kojim se metoda poziva**.

Primjer 1: Ukupna cijena košarice

```

function Namirnica(naziv, cijena, kolicina) {
    this.naziv = naziv;
    this.cijena = cijena;
    this.kolicina = kolicina;
    this.ukupno = function() {
        return this.cijena * this.kolicina;
    }
}

```

```

}

let kosarica = [
  new Namirnica("Jabuka", 2, 3),
  new Namirnica("Kruška", 3, 2),
  new Namirnica("Banana", 1, 5),
  new Namirnica("Naranča", 4, 1)
];
// Akumulator je naša varijabla ukupno koju smo definirali u primjeru iznad
let ukupno = kosarica.reduce((accumulator, currentValue) => accumulator +
currentValue.ukupno(), 0); // Početna vrijednost akumulatora je 0, zatim zbrajamo sve
vrijednosti ukupno za svaku namirnicu
console.log(ukupno); // Ispisuje 21

```

Primjer 2: Zbrajanje svih brojeva u polju

```

let brojevi = [1, 2, 3, 4, 5];
let zbroj = brojevi.reduce((accumulator, currentValue) => accumulator + currentValue, 0);
// Početna vrijednost akumulatora je 0, zatim zbrajamo sve brojeve
console.log(zbroj); // Ispisuje 15

```

Možemo zapisati i bez korištenja `arrow` funkcije:

```

let brojevi = [1, 2, 3, 4, 5];
let zbroj = brojevi.reduce(function(accumulator, currentValue) {
  return accumulator + currentValue;
}, 0); // Uočite gdje se piše parametar initialValue, nakon callback funkcije!
console.log(zbroj); // Ispisuje 15

```

Primjer 3: Pronalazak najvećeg broja u polju

```

let brojevi = [5, 2, 8, 1, 3];
let najveci = brojevi.reduce((accumulator, currentValue) => Math.max(accumulator,
currentValue), brojevi[0]); // Početna vrijednost akumulatora je prvi element polja
console.log(najveci); // Ispisuje 8

```

Ovo funkcioniра зato što `Math.max()` враћа већи број од два броја, а `reduce()` функција користи тај резултат као нову vrijedност akumulatora.

Primjer 4: Grupiranje elemenata polja po količini

```

let kosarica = ["jabuka", "banana", "naranča", "kruška", "jabuka", "jabuka", "kruška"];
let grupirano = kosarica.reduce((accumulator, currentValue) => {
  if (!accumulator[currentValue]) { // Ako ne postoji ključ u objektu, dodajemo ga
    accumulator[currentValue] = 1;
  } else { // Inače povećavamo vrijednost ključa za 1
    accumulator[currentValue]++;
  }
  return accumulator;
}, {});
console.log(grupirano); // Ispisuje {jabuka: 3, banana: 1, naranča: 1, kruška: 2}

```

Primjer 5: Grupiranje objekata po svojstvu

```

let osobe = [
  {ime: "Ana", godine: 20},
  {ime: "Ivan", godine: 30},
  {ime: "Maja", godine: 20},
  {ime: "Marko", godine: 20},
  {ime: "Eva", godine: 28},
  {ime: "Tomislav", godine: 26},
  {ime: "Lucija", godine: 26},
  {ime: "Dario", godine: 31},
  {ime: "Petra", godine: 32},
  {ime: "Nikola", godine: 32},
  {ime: "Lara", godine: 33},
  {ime: "Jakov", godine: 33}
];
const grupiranoPoGodinama = osobe.reduce((accumulator, currentValue) => {
  if (!accumulator[currentValue.godine]) {
    accumulator[currentValue.godine] = [];
  }
  accumulator[currentValue.godine].push(currentValue.ime);
  return accumulator;
}, {});
console.log(grupiranoPoGodinama);

/*
{"20": ["Ana", "Maja", "Marko"], "26": ["Tomislav", "Lucija"], "28": ["Eva"], "30": ["Ivan"], "31": ["Dario"], "32": ["Petra", "Nikola"], "33": ["Lara", "Jakov"]}
*/

```

3.4.6 Kada koristiti koju metodu?

Napredne metode koje smo prošli su vrlo korisne i mogu vam uštedjeti puno vremena kod rješavanja problema. Međutim, važno je znati kada koristiti koju metodu.

- `Array.map()` - koristimo kada želimo **stvoriti novo polje na temelju starog polja**. Ako želimo promijeniti elemente u polju, koristimo map metodu (mapiramo).
- `Array.filter()` - koristimo kada želimo **dobiti novo polje na temelju starog međutim s manje elemenata**. Ako želimo filtrirati elemente u polju, koristimo filter metodu.

- `Array.forEach()` - koristimo kada želimo **proći kroz svaki element polja i ne želimo stvarati novo polje**. Primjer: ispis, ažuriranje elemenata u polju i sl. Metoda ne vraća novo polje već modificira originalno polje ako se tako implementira callback funkcija.
- `Array.some()` - koristimo kada želimo provjeriti **postoji li barem jedan element koji zadovoljava uvjet**.
- `Array.every()` - koristimo kada želimo provjeriti **zadovoljavaju li svi elementi uvjet**.
- `Array.sort()` - koristimo kada želimo **sortirati elemente u polju**.
- `Array.reduce()` - koristimo kada želimo **reducirati polje u jednu vrijednost**.

Vježba 9

EduCoder šifra: `advanced-functions-1`

Koristeći napredne metode `Array` objekta, riješite sljedeće zadatke. Ne smijete koristiti petlje `for` i `while`. Možete i ne morate koristiti `arrow` funkcije.

Zadatak 1:

```
const brojevi = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
// Koristeći metodu map() pohranite korijene svih brojeva u varijablu korijeni
let korijeni = /* Vaš kôd ovdje... */
```

Zadatak 2:

```
const imena = ["Ana", "Ivan", "Maja", "Pero", "Lucija", "Dario", "Petra", "Nikola"];
// Koristeći metodu filter() filtrirajte polje samo na imena koja sadrže slovo "a" i
pohranite ih u varijablu imenaSaA
let imenaSaA = /* Vaš kôd ovdje... */
```

Zadatak 3:

```
const numbers = [1, 2, 3, 4, 5];
function numberPositive(number){
  /* Vaš kôd ovdje... */
}
console.log() // koristeći metodu every() i numberPositive callback funkciju provjerite
jesu li svi brojevi pozitivni
```

Zadatak 4:

```
const months = [
  {name: "January", days: 31},
  {name: "February", days: 28},
  {name: "March", days: 31},
  {name: "April", days: 30},
  {name: "May", days: 31},
  {name: "June", days: 30},
  {name: "July", days: 31},
```

```

    {name: "August", days: 31},
    {name: "September", days: 30},
    {name: "October", days: 31},
    {name: "November", days: 30},
    {name: "December", days: 31}
]
// Koristeći metodu reduce() izračunajte ukupan broj dana u godini. Riješite arrow
callbackom u jednoj liniji koda.
console.log(/* Vaš kôd ovdje... */) // Ispisuje 365

```

Zadatak 5:

```

const imena = ["Ana", "Ivan", "Maja", "Pero", "Lucija", "Dario", "Petra", "Nikola"];
// Napišite funkciju sortirajPoDuljini koja sortira imena po duljini
// Napišite funkciju sortirajPoAbecedi koja sortira imena po abecedi (A -> Z)
// Koristite metodu sort() i arrow funkcije
// Ne smijete koristiti petlje (for, while) i selekcije (if, switch)

```

Vježba 10

EduCoder šifra: advanced-functions-2

Zadano je sljedeće polje studenata koje sadrži ugnijeđene objekte s podacima o studentima.

```

const studenti = [
    {ime: "Ana", prezime: "Anić", upisani_kolegiji: ["PIS", "OOP", "SPA"], prosjek: 4.5},
    {ime: "Ivan", prezime: "Ivić", upisani_kolegiji: ["PIS", "OOP", "PJS", "MAT1", "MAT2"], prosjek: 3.8},
    {ime: "Maja", prezime: "Majić", upisani_kolegiji: ["PIS", "OOP", "PJS", "ENG"], prosjek: 5.0},
    {ime: "Pero", prezime: "Perić", upisani_kolegiji: ["PIS", "OOP", "PJS", "ENG", "SPA"], prosjek: 4.0},
    {ime: "Lucija", prezime: "Lucić", upisani_kolegiji: ["PIS", "OOP", "PJS", "PROG", "SPA"], prosjek: 4.2},
    {ime: "Dario", prezime: "Darić", upisani_kolegiji: ["PIS", "OOP", "PJS", "ENG", "PROG"], prosjek: 3.8},
    {ime: "Petra", prezime: "Petrić", upisani_kolegiji: ["PROG", "OOP", "ENG", "WEBAPPS", "SPA"], prosjek: 4.6},
    {ime: "Nikola", prezime: "Nikolić", upisani_kolegiji: ["PIS", "BP1", "PJS", "ENG", "WEBAPPS"], prosjek: 4.8},
    {ime: "Lara", prezime: "Larić", upisani_kolegiji: ["PIS", "OOP", "BP1", "ENG", "PI"], prosjek: 4.9},
    {ime: "Jakov", prezime: "Jakić", upisani_kolegiji: ["PIS", "OOP", "BP1", "ENG", "PI"], prosjek: 3.7}
];

```

Riješite sljedeće zadatke koristeći napredne metode `Array` objekta. Ne smijete koristiti petlje `for` i `while`. Možete i ne morate koristiti `arrow` funkcije.

- Pohranite u varijablu `imePrezime` polje koje sadrži imena i prezimena svih studenata.

```
const imePrezime = /* Vaš kôd ovdje... */
console.log(imePrezime); // Ispisuje ["Ana Anić", "Ivan Ivić", "Maja Majić", "Pero Perić",
"Lucija Lucić", "Dario Darić", "Petra Petrić", "Nikola Nikolić", "Lara Larić", "Jakov
Jakić"]
```

2. Filtrirajte studente koji imaju prosjek veći od 4.5 i pohranite ih u varijablu `studentiVisokiProsjek`.

```
const studentiVisokiProsjek = /* Vaš kôd ovdje... */
console.log(studentiVisokiProsjek); // Ispisuje
[{"ime":"Maja","prezime":"Majić","upisani_kolegiji":
["PIS","OOP","PJS","ENG"],"prosjek":5},
 {"ime":"Petra","prezime":"Petrić","upisani_kolegiji":
["PROG","OOP","ENG","WEBAPPS","SPA"],"prosjek":4.6},
 {"ime":"Nikola","prezime":"Nikolić","upisani_kolegiji":
["PIS","BP1","PJS","ENG","WEBAPPS"],"prosjek":4.8},
 {"ime":"Lara","prezime":"Larić","upisani_kolegiji":
["PIS","OOP","BP1","ENG","PI"],"prosjek":4.9}]
```

3. Izmjenite originalno polje studenata tako da svim studentima dodate novi ključ `broj_kolegija` koji predstavlja broj kolegija koje student ima upisane.

```
/* Vaš kôd ovdje... */
console.log(studenati); // Ispisuje polje studenata s novim ključem broj_kolegija
```

4. Pohranite u varijablu `projekProsjeka` prosjek svih prosjeka studenata. Rezultat zaokružite na dvije decimale.

```
const projekProsjeka = /* Vaš kôd ovdje... */
console.log(projekProsjeka); // Ispisuje 4.33
```

5. Grupirajte studente po prosjeku u 3 grupe. Možete koristiti selekcije, ne smijete koristiti petlje.

- grupa 1: studenti s prosjekom između 3.5 i 3.9
- grupa 2: studenti s prosjekom između 4.0 i 4.4
- grupa 3: studenti s prosjekom između 4.5 i 5.0

Pohranite rezultat u varijablu `grupiraniStudenti`.

```

const grupiraniStudenti = /* Vaš kôd ovdje... */

console.log(grupiraniStudenti);

{
  "grupa1": [{"ime": "Ivan", " prezime": "Ivić", " upisani_kolegiji": [
    "PIS", "OOP", "PJS", "MAT1", "MAT2"], " prosjek": 3.8},
    {"ime": "Dario", " prezime": "Darić", " upisani_kolegiji": [
      "PIS", "OOP", "PJS", "ENG", "PROG"], " prosjek": 3.8},
    {"ime": "Jakov", " prezime": "Jakić", " upisani_kolegiji": [
      "PIS", "OOP", "BP1", "ENG", "PI"], " prosjek": 3.7}], 

  "grupa2": [{"ime": "Pero", " prezime": "Perić", " upisani_kolegiji": [
    "PIS", "OOP", "PJS", "ENG", "SPA"], " prosjek": 4},
    {"ime": "Lucija", " prezime": "Lucić", " upisani_kolegiji": [
      "PIS", "OOP", "PJS", "PROG", "SPA"], " prosjek": 4.2}], 

  "grupa3": [{"ime": "Ana", " prezime": "Anić", " upisani_kolegiji": [
    "PIS", "OOP", "SPA"], " prosjek": 4.5}, {"ime": "Maja", " prezime": "Majić", " upisani_kolegiji": [
      "PIS", "OOP", "PJS", "ENG"], " prosjek": 5},
    {"ime": "Petra", " prezime": "Petrić", " upisani_kolegiji": [
      "PROG", "OOP", "ENG", "WEBAPPS", "SPA"], " prosjek": 4.6},
    {"ime": "Nikola", " prezime": "Nikolić", " upisani_kolegiji": [
      "PIS", "BP1", "PJS", "ENG", "WEBAPPS"], " prosjek": 4.8},
    {"ime": "Lara", " prezime": "Larić", " upisani_kolegiji": [
      "PIS", "OOP", "BP1", "ENG", "PI"], " prosjek": 4.9}]
  ]
}

```

Samostalni zadatak za vježbu 7

EduCoder šifra: romobil

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

Napomena 2: Ovaj zadatak je svojim obujmom, složenošću i vremenskim ograničenjem vrlo sličan ispitu iz ove skripte (PJS4). Stoga, preporuka je da ga riješite. Za pitanja i pomoć, slobodno se javite na Slack kanal.

1. Dobili ste zadatak napraviti mobilnu aplikaciju za korištenje električnih romobila u gradu. Firma RomobiliPula d.o.o. želi aplikaciju koja će korisnicima omogućiti dijeljenje romobila po gradu. Kada korisnik otvorí aplikaciju, na karti će mu se prikazati svi dostupni romobili u blizini korisnika. Korisnik će moći otključati romobil, aktivirati ga i krenuti voziti. Kada korisnik završava s vožnjom, romobil zaključava aplikacijom koja računa cijenu vožnje. Plaćanje se vrši automatski putem aplikacije, odnosno podacima o kreditnoj kartici koje korisnik unese prilikom registracije.

Potrebno je u grubo izmodelirati ovaj poslovni proces kroz objekt RomobiliPula.

Koristeći ugniježđene strukture izradite objekt RomobiliPula koji će sadržavati sljedeće podatke:

- naziv grada (npr. "Pula")
- adresa sjedišta firme (npr. "Ulica 123, 52100, Pula")
- kontakt podaci (npr. "091 123 4567", "romobila@pula.hr")

- cijena otključavanja romobila (npr. 2 eur)
- cijena po prijeđenom kilometru (npr. 1 eur/km)
- polje više romobila gdje svaki romobil ima sljedeće podatke:
 - id (jedinstveni identifikator)
 - lokacija (npr. "Arena Pula")
 - baterija (npr. 100%)
 - status (boolean - npr. "slobodan", "zauzet")
 - trenutni korisnik (npr. "Ivan Ivić")
 - prijeđeni kilometri (npr. 0 km)
- polje korisnika gdje su sadržani podaci o korisnicima:
 - id (jedinstveni identifikator)
 - ime (npr. "Ivan")
 - prezime (npr. "Ivić")
 - email (npr. "ivanivic@gmail.com")

Ne morate raditi konstruktor funkcije, odmah krenite raditi objekt. Napravite po 2 objekta za romobile i korisnike.

```
let RomobiliPula = {
  /* Vaš kôd ovdje... */
}
```

Jednom kada ste izradili objekt `RomobiliPula`, potrebno je izraditi metode i funkcije koje će simulirati rad aplikacije. **Ne smijete koristiti petlje za niti jedan zadatak**, morate koristiti metode `Array` objekta.

2. Metoda `otkljucavanjeRomobila` koja prima **id romobila** i **id korisnika**. Metoda treba pronaći romobil s odgovarajućim id-em i postaviti status romobila na "zauzet". Također, metoda treba postaviti trenutnog korisnika na korisnika s odgovarajućim id-em. Metoda treba vratiti poruku `"Romobil je uspješno otključan."`. Ako je romobil već zauzet ili ne postoji, metoda treba vratiti poruku `"Romobil nije dostupan."`. Ako korisnik s odgovarajućim id-em ne postoji, metoda treba vratiti poruku `"Korisnik nije pronađen."`.

```
function otkljucavanjeRomobila(idRomobila, idKorisnika) {
  /* Vaš kôd ovdje... */
}

RomobiliPula.otkljucavanjeRomobila(2, 1); // Ispisuje "Romobil je uspješno otključan."
RomobiliPula.otkljucavanjeRomobila(500, 1); // Ispisuje "Romobil nije dostupan."
RomobiliPula.otkljucavanjeRomobila(2, "pero"); // Ispisuje "Korisnik nije pronađen."
```

3. Metoda `dohvatiDostupneRomobile` koja vraća novo polje svih dostupnih romobila. Dostupni romobili su oni koji imaju status "slobodan" i bateriju veću od 20%.

```
RomobilPula.dohvatiDostupneRomobile = function() {  
    /* Vaš kôd ovdje... */  
}
```

4. Dodajte metodu `zakljucajRomobil` koja prima argumente **id romobila**. Metoda treba pronaći romobil s odgovarajućim id-em i postaviti status romobila na "slobodan". Također, metoda treba postaviti trenutnog korisnika na `null`. Metoda treba vratiti poruku "`Romobil je uspješno zaključan.`". Ako romobil nije pronađen, metoda treba vratiti poruku "`Romobil nije pronađen.`". Dodatno, metoda mora izračunati ukupnu cijenu vožnje koja je jednaka: `cijena otključavanja + cijena po prijeđenom kilometru * prijeđeni kilometri za taj romobil`. Metoda mora vratiti ukupnu cijenu kao povratnu vrijednost. Nakon izračuna metoda mora postaviti prijeđene kilometre na 0.

```
RomobiliPula.zakljucajRomobil = function(idRomobila) {  
    /* Vaš kôd ovdje... */  
}
```

5. Potrebno je nadograditi metodu `zakljucajRomobil` tako da prije samog zaključavanja romobila, metoda pohrani ukupnu cijenu te pojedine vožnje u novo svojstvo objekta: `cijene_voznji` (polje brojeva) gdje se pohranjuju sve cijene vožnji kao cjelobrojne vrijednosti. Kada to napravite, dodajte globalnu varijablu `zarada` koja će predstavljati ukupnu zaradu firme. Varijabla mora biti definirana izvan objekta `RomobiliPula` te mora koristeći metodu `reduce()` i arrow callback funkciju pohraniti ukupnu vrijednost `RomobiliPula.cijene_voznji`.

```
RomobiliPula.zakljucajRomobil = function(idRomobila) {  
    /* Nadogradite Vaš kôd */  
}  
let zarada = /* Vaš kôd ovdje... */
```

6. Dodajte globalnu funkciju `pronadiRomobil()` koja prima argument **lokacija**. Funkcija treba pronaći sve romobile koji se nalaze na određenoj lokaciji i vratiti `true` ako postoji barem jedan romobil na toj lokaciji koji je dostupan. Romobil je dostupan ako je status "slobodan" i baterija veća od 20%. Ako nema romobila na lokaciji, funkcija treba vratiti `false`.

```
function pronadiRomobil(lokacija) {  
    /* Vaš kôd ovdje... */  
}
```

Programiranje u skriptnim jezicima (PJS)

Nositelj: doc. dr. sc. Nikola Tanković

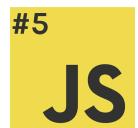
Asistenti:

- Luka Blašković, mag. inf.
- Alesandro Žužić, mag. inf.

Ustanova: Sveučilište Jurja Doprile u Puli, Fakultet informatike u Puli



[5] DOM, JSON i Asinkrono programiranje



Prilikom izrade web aplikacija i stranica, često ćete na neki način manipulirati strukturom dokumenata i njihovim sadržajem. U ovom poglavlju upoznat ćemo se s Document Object Model (DOM) standardom, koji predstavlja aplikacijsko programsko sučelje (API) za kontrolu HTML-a koristeći Document objekt. Važno je razumjeti kako funkcioniра DOM budući da se svi poznati JavaScript razvojni okviri temelje na njemu (React, VUE, Angular, jQuery...). Dodatno, upoznat ćemo se s JSON formatom (JavaScript Object Notation) koji se koristi za razmjenu podataka između klijenta i servera te predstavlja jedan od najčešćih, ako ne i najčešće korišteni format za razmjenu podataka. Za sam kraj ćemo proći asinkrono programiranje i time postaviti dobre temelje za uvod u svijet programskog inženjerstva i razvoja web aplikacija.

Posljednje ažurirano: 2.8.2024.

Sadržaj

- [Programiranje u skriptnim jezicima \(PJS\)](#)
- [5. DOM, JSON i Asinkrono programiranje](#)
 - [Sadržaj](#)
- [0. Ponavljanje HTML-a i CSS-a](#)
 - [Primjer upotrebe `id` atributa:](#)
 - [Primjer upotrebe `class` atributa:](#)
 - [Primjer kombiniranja `id` i `class` atributa:](#)
 - [CSS \(Cascading Style Sheets\)](#)
- [1. DOM manipulacija](#)
 - [1.1 Osnovni DOM element](#)
 - [1.2 Dohvaćanje DOM elemenata](#)
 - [Primjer 1. - Dohvaćanje elemenata](#)

- [1.3 Svojstva DOM elemenata](#)
 - [Primjer 2 - Pristupanje sadržaju, `id`-u, `tag`-u, atributima i klasama elementa](#)
 - [Vježba 1](#)
 - [Primjer 3 - Manipulacija klasama](#)
 - [Vježba 2](#)
 - [Primjer 4 - Dohvaćanje `child` i `sibling` elemenata.](#)
 - [Vježba 3](#)
- [1.4 Dodavanje i brisanje DOM elemenata](#)
 - [Primjer 5 - Stvaranje, dodavanje, brisanje i izmjena DOM elemenata](#)
 - [Vježba 4](#)
- [1.4 DOM events](#)
 - [Primjer 6 - `click` event](#)
 - [Vježba 5](#)
 - [Primjer 7 - `focus` events](#)
 - [Vježba 6](#)
 - [Primjer 8 - `mouse` events](#)
 - [Vježba 7](#)
 - [Primjer 9 - `input` event](#)
 - [Vježba 8](#)
- [Samostalni zadatak za vježbu 8](#)
- [2. JSON - JavaScript Object Notation](#)
 - [2.1 Struktura JSON-a](#)
 - [2.2 Primjeri ispravnog i neispravnog JSON formata](#)
 - [2.3 Online alati za prikaz/validaciju JSON formata](#)
 - [2.4 Rad s JSON formatom u JavaScriptu](#)
 - [2.4.1 `JSON.parse\(\)`](#)
 - [2.4.1 `JSON.stringify\(\)`](#)
 - [2.5 Lokalno čitanje JSON datoteka](#)
 - [2.5.1 Node.js](#)
 - [2.5.2 Web preglednik](#)
 - [Vježba 9](#)
- [3. Asinkrono programiranje](#)
 - [3.1 Razumijevanje asinkronog vs. sinkronog](#)
 - [3.2 Asinkrone callback funkcije](#)
 - [3.3 Fetch API - dohvaćanje podataka s web poslužitelja](#)
 - [Vježba 10](#)
 - [3.4 Promise objekt](#)

- [Primjer 10](#)
- [3.5 Async/Await](#)
- [Samostalni zadatak za vježbu 9](#)

0. Ponavljanje HTML-a i CSS-a

U ovoj sekciji ćemo se osvrnuti na osnove HTML-a i CSS-a. **HTML** (HyperText Markup Language) je markup jezik za označavanje struktura web stranica, dok je CSS (Cascading Style Sheets) jezik za stilizaciju tih struktura. Kombinacija ova dva jezika omogućuje nam da oblikujemo i prikažemo sadržaj web stranica na željeni način.

Kako bi vam HTML i CSS jezici već trebali biti poznati, u nastavku ćemo se osvrnuti na neke osnovne koncepte i primjere korištenja HTML i CSS elemenata. U skripti se dalje nećemo detaljno baviti HTML i CSS jezicima, već ćemo se fokusirati na JavaScript, odnosno kako možemo **manipulirati HTML elementima** pomoću JavaScripta.

HTML (HyperText Markup Language)

HTML je markup jezik (*eng. markup language*) koji se koristi za strukturiranje sadržaja web stranica. Sastoji se od HTML elemenata koji se koriste za označavanje dijelova sadržaja. Svaki HTML element sastoji se od otvarajuće oznake (*eng. start tag*), sadržaja elementa (*eng. content*) i zatvarajuće oznake (*eng. end tag*).

Osnovna struktura HTML dokumenta često se može podijeliti na `head` i `body` dijelove. `head` dio obično sadrži meta informacije o dokumentu, kao što su naslov stranice, veze prema CSS datotekama, skripte itd. `body` dio sadrži glavni sadržaj stranice, poput zaglavlja (**header**), navigacije (**nav**), članaka (**article**) i podnožja (**footer**).

```
<head>
    <title>Moja web stranica</title>
</head>

<body>
    <header>
        <h1>Naslov</h1>
        <nav>
            <ul>
                <li><a href="#">Početna</a></li>
                <li><a href="#">O nama</a></li>
                <li><a href="#">Kontakt</a></li>
            </ul>
        </nav>
    </header>

    <main>
        <article>
            <h2>Članak 1</h2>
            <p>Ovo je prvi članak.</p>
        </article>

        <article>
            <h2>Članak 2</h2>
            <p>Ovo je drugi članak.</p>
        </article>
    </main>
```

```
<footer>
  <p>&copy; 2022 Web Stranica</p>
</footer>
</body>
```

Treba napomenuti da je dolaskom HTML5 standarda uvedeno mnogo novih elemenata i atributa koji omogućuju bolje semantičko označavanje sadržaja web stranica. Semantičko označavanje je važno jer pomaže tražilicama i drugim alatima da bolje razumiju strukturu i značenje sadržaja na web stranici. Svakako je moguće gotovo sve elemente stilizirati pomoću CSS-a i svesti na `div` i `span` elemente, ali korištenje semantičkih elemenata poboljšava pristupačnost i [SEO](#) (Search Engine Optimization) web stranice.

U najnovijem HTML standardu za vrijeme pisanja ove skripte (svibanj 2024), postoji 142 HTML elementa. U tablici ispod prikazani su neki od najčešće korištenih HTML elemenata, zajedno s njihovim opisima, atributima i primjerima korištenja. Svakako provjerite i [HTML5 specifikaciju](#).

Naziv elementa	Opis	Atributi	Primjer
<html>	Korijenski element HTML dokumenta	lang	<html lang="en"> ... </html>
<head>	Sadrži meta-informacije o dokumentu	-	<head> ... </head>
<title>	Naslov dokumenta, prikazan u naslovnoj traci preglednika	-	<title>Naslov stranice</title>
<meta>	Definira metapodatke o HTML dokumentu	charset, name, content, http-equiv	<meta charset="UTF-8">
<link>	Povezuje vanjske resurse, poput CSS datoteka	rel, href, type	<link rel="stylesheet" href="style.css">
<style>	Sadrži CSS stilove za dokument	type	<style> body { background-color: lightblue; } </style>
<script>	Sadrži ili povezuje JavaScript kôd	src, type	<script src="script.js"></script>
<body>	Glavni sadržaj HTML dokumenta	-	<body> ... </body>
<h1> do <h6>	Naslovi različitih razina	-	<h1>Naslov 1</h1>
<p>	Paragraf teksta	-	<p>Ovo je paragraf.</p>
<a>	Hiperlink (poveznica)	href, target	Link
	Ugrađena slika	src, alt, width, height	
	Neuređena lista (bez numeriranja)	-	Prva stavkaDruga stavka
	Uređena lista (numerirana)	-	Prva stavka numeriranoDruga stavka numerirano
	Stavka liste	-	Stavka
<table>	Tablica	-	<table> ... </table>
<tr>	Redak u tablici	-	<tr> ... </tr>
<td>	Ćelija u tablici	colspan, rowspan	<td colspan="2">Ćelija</td>
<th>	Zaglavljene ćelije u tablici	colspan, rowspan, scope	<th scope="col">Zaglavljene</th>
<form>	Forma za unos podataka	action, method	<form action="/submit" method="post"> ... </form>
<input>	Unos podataka	type, name, value, placeholder	<input type="text" name="ime" placeholder="Unesite ime">
<button>	Gumb	type	<button type="submit">Pošalji</button>
<div>	Block element za grupiranje sadržaja	-	<div> ... </div>
	Inline element za grupiranje sadržaja	-	 ...
 	Prijelom linije	-	Tekst Preolomljena linija
<hr>	Horizontalna linija	-	<hr>
	Podebljani tekst	-	Podebljano
<i>	Kurziv tekst	-	<i>Kurziv</i>
<u>	Podvučeni tekst	-	<u>Podvučeno</u>

Primjer upotrebe id atributa:

Atribut `id` koristi se za jedinstveno identificiranje gotovo bilo kojeg HTML elementa na stranici. `id` atribut mora biti **jedinstven unutar cijelog dokumenta**.

Sintaksa: `<tag id="jedinstveniID">`.

```

<style>
    #jedinstveniElement {
        color: blue;
        font-size: 20px;
    }
</style>
<body>
    <p id="jedinstveniElement">Ovo je paragraf s jedinstvenim ID atributom.</p>
</body>
</html>

```

Primjer upotrebe `class` atributa:

Atribut `class` koristi se za grupiranje više elemenata koji dijele iste stilove ili ponašanje. Elementi mogu imati **više klasa odvojenih razmakom.**

Sintaksa: `<p class="klasa1 klasa2 klasa3 klasaN...">`.

```

<style>
    .crveniTekst {
        color: red;
    }
    .velikiTekst {
        font-size: 24px;
    }
</style>
<body>
    <p class="crveniTekst">Ovo je paragraf s crvenim tekstrom.</p>
    <p class="velikiTekst">Ovo je paragraf s velikim tekstrom.</p>
    <p class="crveniTekst velikiTekst">Ovo je paragraf s crvenim i velikim tekstrom.</p>
</body>
</html>

```

Primjer kombiniranja `id` i `class` atributa:

U ovom primjeru koristimo oba atributa kako bismo jedinstveno identificirali jedan element, dok ostala dva grupiramo pomoću klase `podnaslov`.

```

<style>
    #glavniNaslov {
        color: green;
        font-size: 28px;
    }
    .podnaslov {
        color: gray;
        font-size: 18px;
    }
</style>
<body>
    <h1 id="glavniNaslov">Ovo je glavni naslov s ID atributom.</h1>
    <h2 class="podnaslov">Ovo je podnaslov s class atributom.</h2>
    <h2 class="podnaslov">Ovo je još jedan podnaslov s class atributom.</h2>
</body>

```

```
</body>
```

CSS (Cascading Style Sheets)

CSS je jezik za stilizaciju HTML elemenata. Omogućuje nam definiranje izgleda i rasporeda elemenata na web stranici. CSS pravila sastoje se od selektora ([eng. selectors](#)) i deklaracija svojstava ([eng. declarations](#)).

Sljedeća tablica pokriva osnovna CSS svojstva:

Naziv svojstva	Opis	Vrijednosti	Primjer
<code>color</code>	Boja teksta	Naziv boje, heksadecimalna, RGB, RGBA, HSL, HSLA	<code>color: red;</code>
<code>background-color</code>	Boja pozadine	Naziv boje, heksadecimalna, RGB, RGBA, HSL, HSLA	<code>background-color: #f4f4f4;</code>
<code>background-image</code>	Slika pozadine	URL slike	<code>background-image: url('slika.jpg');</code>
<code>background-size</code>	Veličina slike pozadine	cover, contain, px, %	<code>background-size: cover;</code>
<code>font-size</code>	Veličina fonta	px, em, rem, %, vw, vh	<code>font-size: 16px;</code>
<code>font-family</code>	Obitelj fonta	Naziv fonta, lista fontova	<code>font-family: Arial, sans-serif;</code>
<code>font-weight</code>	Debljina fonta	normal, bold, bolder, lighter, 100-900	<code>font-weight: bold;</code>
<code>text-align</code>	Poravnanje teksta	left, right, center, justify	<code>text-align: center;</code>
<code>margin</code>	Vanjski razmak	px, em, %, auto	<code>margin: 10px;</code>
<code>padding</code>	Unutarnji razmak	px, em, %, auto	<code>padding: 10px;</code>
<code>border</code>	Obrub elementa	Širina tip stil boje	<code>border: 1px solid #ccc;</code>
<code>border-radius</code>	Zaobljenost rubova	px, %, em	<code>border-radius: 5px;</code>
<code>width</code>	Širina elementa	px, em, %, vw	<code>width: 100%;</code>
<code>height</code>	Visina elementa	px, em, %, vh	<code>height: 50px;</code>
<code>display</code>	Način prikaza	block, inline, inline-block, flex, grid, none	<code>display: block;</code>
<code>position</code>	Pozicioniranje elementa	static, relative, absolute, fixed, sticky	<code>position: absolute;</code>
<code>top, right, bottom, left</code>	Pozicija elementa	px, %, em	<code>top: 10px;</code>
<code>overflow</code>	Upravljanje preljevom sadržaja	visible, hidden, scroll, auto	<code>overflow: hidden;</code>
<code>z-index</code>	Redoslijed prikazivanja elemenata	brojčana vrijednost	<code>z-index: 10;</code>
<code>opacity</code>	Prozirnost elementa	vrijednost od 0 do 1	<code>opacity: 0.5;</code>
<code>box-shadow</code>	Sjenka okvira	x-offset y-offset blur-radius spread-radius boja	<code>box-shadow: 0 4px 8px rgba(0,0,0,0.1);</code>
<code>text-shadow</code>	Sjenka teksta	x-offset y-offset blur-radius boja	<code>text-shadow: 1px 1px 2px black;</code>
<code>flex-direction</code>	Smjer fleksibilnog kontejnera	row, row-reverse, column, column-reverse	<code>flex-direction: row;</code>
<code>justify-content</code>	Poravnanje stavki duž glavne osi	flex-start, flex-end, center, space-between, space-around	<code>justify-content: center;</code>
<code>align-items</code>	Poravnanje stavki duž poprečne osi	flex-start, flex-end, center, baseline, stretch	<code>align-items: center;</code>
<code>transition</code>	Definira glatku animaciju prijelaza između različitih stilova	svojstvo, trajanje, kašnjenje, funkcija	<code>transition: width 0.3s ease-in-out;</code>

Postoje i CSS pseudo-klase ([Pseudo-classes](#)) kao što su `:hover`, `:active`, `:focus` i druge koje se mogu nadodati na CSS klase. Mogu se koristiti za primjenu stilova na elemente u određenim stanjima ili uvjetima.

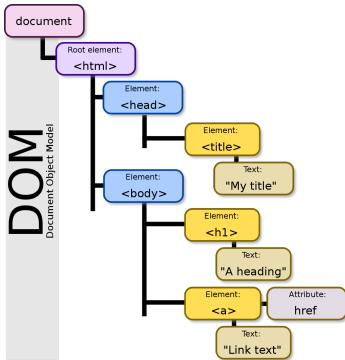
Tablica prikazuje neke od najčešće korištenih CSS pseudo-klasa:

Pseudo-klasa	Opis	Primjer
<code>:hover</code>	Primjenjuje stil kada korisnik prelazi mišem preko elementa.	<code>css a:hover { color: blue; }</code>
<code>:active</code>	Primjenjuje stil kada je element aktiviran (npr. kliknut).	<code>css button:active { background-color: green; }</code>
<code>:focus</code>	Primjenjuje stil kada je element u fokusu (npr. odabran tipkovnicom).	<code>css input:focus { border-color: red; }</code>
<code>:visited</code>	Primjenjuje stil na posjećene linkove.	<code>css a:visited { color: purple; }</code>
<code>:link</code>	Primjenjuje stil na neposjećene linkove.	<code>css a:link { color: blue; }</code>
<code>:first-child</code>	Primjenjuje stil na prvi dijete elementa.	<code>css p:first-child { font-weight: bold; }</code>
<code>:last-child</code>	Primjenjuje stil na zadnji dijete elementa.	<code>css p:last-child { font-style: italic; }</code>
<code>:nth-child(n)</code>	Primjenjuje stil na n-ti dijete elementa.	<code>css li:nth-child(2) { color: red; }</code>
<code>:not(selector)</code>	Primjenjuje stil na sve elemente koji ne odgovaraju zadanom selektoru.	<code>css p:not(.highlight) { color: grey; }</code>
<code>:checked</code>	Primjenjuje stil na odabrane (checked) elemente kao što su checkbox ili radio button.	<code>css input:checked { background-color: yellow; }</code>
<code>:disabled</code>	Primjenjuje stil na onemogućene (disabled) elemente.	<code>css input:disabled { background-color: lightgrey; }</code>
<code>:enabled</code>	Primjenjuje stil na omogućene (enabled) elemente.	<code>css input:enabled { background-color: white; }</code>
<code>:empty</code>	Primjenjuje stil na elemente koji nemaju djece (prazne elemente).	<code>css div:empty { display: none; }</code>

1. DOM manipulacija

Nakon stjecanja osnovnog razumijevanja JavaScript varijabli, funkcija, struktura i metoda, sada smo spremni za početak manipulacije **Document Object Model**, odnosno DOM-om, što uključuje dinamičko upravljanje HTML elementima i njihovim CSS svojstvima.

Document Object Model (DOM) je standard koji definira strukturu i način pristupa HTML dokumentima. DOM predstavlja HTML dokument kao stablo objekata, gdje svaki HTML element predstavlja objekt, a svaki atribut i sadržaj elementa predstavlja svojstvo tog objekta.



Izvor: https://en.wikipedia.org/wiki/Document_Object_Model

Zašto je važno naučiti DOM manipulaciju?

- Većina današnjeg digitalnog poslovanja bazirana je upravno na web tehnologijama.
- JavaScript je jedan od najpopularnijih jezika za web razvoj.
- Statične web stranice postale su prošlost, danas su gotovo sve web stranice dinamične (interaktivne).
- DOM manipulacija je ključna za stvaranje interaktivnih web stranica, samim time ih onda počinjemo nazivati i web aplikacijama.
- Poznavanje DOM-a je ključno za razumijevanja rada svih modernih JavaScript biblioteka i okvira (React, Angular, Vue, jQuery...).
- Poznavanje DOM-a je ključno za razvoj efikasnih i responzivnih korisničkih sučelja.

1.1 Osnovni DOM element

`Document` objekt je ključna komponenta u JavaScriptu koja predstavlja cijelu web stranicu u trenutnom pregledniku. On omogućava pristupanje i manipulaciju svim elementima na stranici, kao i njihovim svojstvima i sadržaju. Olakšava dinamičko upravljanje sadržajem stranice, što je ključno za stvaranje interaktivnih i responzivnih korisničkih iskustava.

Možemo ga zamisliti kao korijenski čvor HTML dokumenta (slika iznad).

`document` objekt možemo referencirati direktno ili preko `window` objekta.

```
// Referenciranje document objekta
let doc = document;

// ili
let doc = window.document;
```

`document` objekt ima brojna svojstva i metode, mi ćemo u ovoj skripti baviti prvenstveno metodama, no to mogu biti i svojstva. Na primjer, `document.title` vraća naslov stranice, a `document.URL` vraća URL stranice.

```
console.log(document.title); // Ispisuje naslov stranice
console.log(document.URL); // Ispisuje URL stranice
```

1.2 Dohvaćanje DOM elemenata

Kako bismo uopće mogli raditi s DOM elementima prvo ih moramo "dohvatiti".

Imamo zadan sljedeći HTML gdje želimo izvući vrijednosti iz pojedinih elemenata.

```
<p id="mojID">Ja sam paragraf 1</p> <!-- ID -->
<input type="text" name="ime" id="form_ime" class="mojInput" value="Marko" /> <!-- ID, class -->
<input type="text" name="prezime" id="form_prezime" class="mojInput" value="Marić" /> <!-- ID, class -->
<p class="mojaKlasa">Ja sam paragraf 2</p> <!-- class -->
<span class="mojaKlasa">Ja sam span</span> <!-- class -->
<p>Ja sam paragraf 3</p>
```

HTML elementi se mogu dohvatiti na sljedeće načine:

Metoda	Objašnjenje	Sintaksa	Primjer
getElementById(x)	Vraća prvi element po jedinstvenom Id-u.	document.getElementById(x)	document.getElementById("mojID")
getElementsByName(x)	Vraća sve elemente po HTML tag-u.	document.getElementsByName(x)	document.getElementsByName("p")
getElementsByClassName(x)	Vraća sve elemente po klasi/klasama ili kombinaciji HTML tag-a i klase (class).	document.getElementsByClassName(x)	document.getElementsByClassName("mojaKlasa")
getElementsByTagName(x)	Vraća sve elemente po imenu.	document.getElementsByTagName(x)	document.getElementsByTagName("ime")
querySelector(x)	Vraća prvi element koji odgovara određenom selektoru ili grupi selektora. Ako nema pronađenih podudaranja, vraća null.	document.querySelector(x)	document.querySelector("#mojID") document.querySelector(".mojaKlasa") document.querySelector("p") document.querySelector("input[name='ime']")
querySelectorAll(x)	Vraća sve elemente koji odgovaraju određenom selektoru ili grupi selektora. Ako nema pronađenih podudaranja, vraća null.	document.querySelectorAll(x)	document.querySelectorAll("#mojID") document.querySelectorAll(".mojaKlasa") document.querySelectorAll("p") document.querySelectorAll("input[name='ime']")

U sljedećem JavaScript kôdu možete vidjeti primjere dohvaćanja elemenata koristeći metode iz tablice.

```
// Dohvaćanje prvog DOM elementa po ID-u
```

```

const mojDiv = document.getElementById('mojID');
console.log("Dohvaćen po ID-u: " + mojDiv.innerHTML);

// Dohvaćanje DOM elemenata po tagu
const paragrafi = document.getElementsByTagName('p');
for (let paragraf of paragrafi){
    console.log("Dohvaćen po tagu: " + paragraf.innerHTML);
}

// Dohvaćanje DOM elemenata po klasi
const sveKlase = document.getElementsByClassName('mojaKlasa');
for (let klasa of sveKlase){
    console.log("Dohvaćen po klasi: " + klasa.innerHTML);
}

// Dohvaćanje DOM elemenata po imenu
const svaImena = document.getElementsByName('ime');
for (let ime of svaImena){
    console.log("Dohvaćen po imenu: " + ime.value);
}

// Dohvaćanje prvog DOM elemenata koristeći querySelector
const query1 = document.querySelector('#form_prezime');
console.log("Dohvaćen koristeći querySelector: " + query1.value);
const query2 = document.querySelector('span');
console.log("Dohvaćen koristeći querySelector: " + query2.innerHTML);
const query3 = document.querySelector('.mojaKlasa');
console.log("Dohvaćen koristeći querySelector: " + query3.innerHTML);
const query4 = document.querySelector("input[name='ime']");
console.log("Dohvaćen koristeći querySelector: " + query4.value);

// Dohvaćanje svih DOM elemenata koristeći querySelectorAll
const queryAll = document.querySelectorAll('p');
for (let query of queryAll){
    console.log("Dohvaćen koristeći querySelectorAll: " + query.innerHTML);
}

```

`querySelector` uvijek prvo pretražuje po `tag`-u, za pretraživanje po `id`-u treba koristiti oznaku `#`. za pretraživanje po klasi treba koristiti `.` dok za pretraživanje po imenu ili drugim atributima prvo treba staviti ime `tag`-a pa unutar uglatih zagrada pretragu `[atribut = vrijednost]`

Primjer 1. - Dohvaćanje elemenata

Za zadani HTML kôd, treba dohvatiti `<input>` s vrijednošću **TOČNO**. Ne smijemo koristiti `id` atribut i naknadno mijenjati HTML.

```

<div class="moja-forma glavni2">
    <span name="ime">KRIVO</span>
    <span name="prezime">KRIVO</span>
</div>
<span class="moja-forma glavni">
    <span name="ime">KRIVO</span>
    <span name="prezime">KRIVO</span>

```

```

</span>
<div class="moja-forma glavni">
  <span name="prezime">KRIVO</span>
  <span name="ime">TOČNO</span>
  <i name="ime">KRIVO</i>
</div>
<div class="druga-forma glavni">
  <span name="prezime">KRIVO</span>
  <span name="ime">KRIVO</span>
</div>

```

Treba dohvatiti prvi `` element s imenom `"ime"`: (``), a koji se nalazi unutar `<div>` elementa s klasama `"moja-forma glavni"` (`<div class="moja-forma glavni">`).

Rješenje:

```

const query = document.querySelector("div.moja-forma.glavni span[name='ime']");
// Pogledati definiciju selektora u tablici
console.log(query.innerHTML)

```

1.3 Svojstva DOM elemenata

DOM Elementi imaju mnogo svojstava, od kojih smo već neka koristili neka za dohvaćanje sadržaja elemenata poput `innerHTML`.

Možemo ih podijeliti u svojstva za:

- dohvaćanje i postavljanje atributa,
- dohvaćanje sadržaja,
- dohvaćanje stilova,
- dohvaćanje djece i susjeda.

U sljedećoj tablici su prikazana neka od bitnijih svojstava:

Svojstvo	Objašnjenje	Sintaksa
<code>id</code>	Vraća ili postavlja vrijednost <code>id</code> atributa elementa.	<code>element.id</code>
<code>tagName</code>	Vraća ime <code>tag</code> -a elementa velikim slovima.	<code>element.tagName</code>
<code>classList</code>	Vraća kolekciju klasa elementa.	<code>element.classList</code>
<code>className</code>	Vraća ili postavlja vrijednost <code>class</code> atributa elementa.	<code>element.className</code>
<code>innerHTML</code>	Vraća ili mijenja HTML sadržaj unutar elementa.	<code>element.innerHTML</code>
<code>outerHTML</code>	Vraća HTML elementa, uključujući sam element i njegov sadržaj.	<code>element.outerHTML</code>
<code>attributes</code>	Vraća kolekciju svih atributa elementa.	<code>element.attributes</code>
<code>style</code>	Vraća <code>style</code> atribut elementa.	<code>element.style</code>
<code>childElementCount</code>	Vraća broj direktne djece elementa.	<code>element.childElementCount</code>
<code>children</code>	Vraća kolekciju direktne djece elementa.	<code>element.children</code>
<code>firstElementChild</code>	Vraća prvo direktno dijete elementa.	<code>element.firstElementChild</code>
<code>lastElementChild</code>	Vraća posljednje direktno dijete elementa.	<code>element.lastElementChild</code>
<code>nextElementSibling</code>	Vraća sljedeći element nakon <code>element</code> u roditeljskom elementu.	<code>element.nextElementSibling</code>
<code>previousElementSibling</code>	Vraća element prije <code>element</code> u roditeljskom elementu.	<code>element.previousElementSibling</code>

Primjer 2 - Pristupanje sadržaju, `id`-u, `tag`-u, atributima i klasama elementa

```
<div class="text-5xl">
    Hi!
</div>
<div id="mojID" class="text-5xl h-16 w-36 overflow-scroll">
    Hello, World!
</div>
```

```
const element = document.getElementById('mojID');

console.log(element.innerHTML); // Output: "Hello, World!"
console.log(element.outerHTML); // Output: "<div id='mojID' class='text-5xl h-16 w-36 overflow-scroll'> Hello, World! </div>"

console.log(element.id); // Output: "mojID"
console.log(element.tagName); // Output: "DIV"

for (const attr of element.attributes) {
  console.log(`attr: ${attr.name} -> ${attr.value}`);
  // Output: "attr: id -> mojID"
  // Output: "attr: class -> text-5xl h-16 w-36 overflow-scroll"
}
```

Međutim, kolekciju klasa možemo dohvatiti preko `className` ili `classList` svojstva. `className` vraća `string` svih klasa dok `classList` vraća `DOMTokenList` objekt koji omogućava korištenje `forEach` petlje.

```
// Dohvaćanje klase preko className
console.log(element.className); // Output: "text-5xl h-16 w-36 overflow-scroll"

// Dohvaćanje klase preko classList
element.classList.forEach( klasa => {
  console.log(`class: ${klasa}`);
  // Output: class: -> text-5xl
  // Output: class: -> h-16
  // Output: class: -> w-36
  // Output: class: -> overflow-scroll
})
```

`attributes` svojstvo **ne vraća polje objekata već objekt objekata** kao povratnu vrijednost, tako da je za iteraciju najbolje koristiti `for of` petlju. Međutim, `classList` vraća `DOMTokenList` koja omogućava korištenje `forEach` petlje.

`attributes` vraća kolekciju tipa `NamedNodeMap` koja nema mogućnost korištenja `forEach` petlje.

Elementu možemo direktno mijenjati ili dodati `id` koristeći `id` svojstvo.

Možemo dohvatiti prvi `div` element s tekstrom "Hi!" i dodati mu `id: "prviDiv"`.

```
const element = document.querySelector('div')
console.log(element.outerHTML); //Output: "<div class='text-5xl'> Hi! </div>"
element.id = "prviDiv" // Dodavanje ID-a
console.log(element.outerHTML); //Output: "<div class='text-5xl' id='prviDiv'> Hi! </div>"
```

Sadržaj mu možemo promijeniti preko `innerHTML` svojstva.

```
element.innerHTML = " Pozdrav! " //Mijenjanje sadržaja
console.log(element.outerHTML); //Output: "<div class='text-5xl' id='prviDiv'> Pozdrav!
</div>"
```

Mijenjanjem sadržaja preko `outerHTML` svojstva možemo prekrižiti cijeli element pa nije pametno to koristiti.

Međutim, za navedeno postoje metode koje ćemo proći u poglavljju `Dodavanje i brisanje DOM elemenata`

Vježba 1

EduCoder šifra: `funte_u_eure`

Pronašli smo idealni web shop u Engleskoj, međutim sve cijene su prikazane u funtama, a stranica nema ugrađenu konverziju valuta. Želimo da nam se automatski prikažu sve cijene u valuti eura. Zadatak nam je malo "hakirati" ovaj web shop bez da mijenjamo HTML kôd.

```
<div class="item">
  <b>-</b>
  <span> Laptop </span>
  <u>1200</u>
  <span class="symbol">£</span>
```

```

</div>
<div class="item">
    <b>-</b>
    <span> PC </span>
    <u>1800</u>
    <span class="symbol">£</span>
</div>
<div class="item">
    <b>-</b>
    <span> Mouse & Keyboard </span>
    <u>200</u>
    <span class="symbol">£</span>
</div>

```

- Napišite funkciju `azurirajSimbol(klasa, noviSimbol)` koja će za danu klasu promijeniti unutarnji sadržaj svih klasa na novi sadržaj.
- Napišite funkciju `azurirajCijenu(tag)` koja će za dani `tag` napraviti konverziju unutarnjeg sadržaja (cijena) svih `tag`-ova iz cijene u funtama u cijenu u eurima, zaokruženo na dvije decimale.
- Devizni tečaj: `1£ = 1.16547€`

Rezultat:

- Laptop 1398.56 €
- PC 2097.85 €
- Mouse & Keyboard 233.09 €

Rješenje:

```

function azurirajSadrzaj(klasa, noviSimbol) {
    const query = document.querySelectorAll("." + klasa)
    for (let element of query) {
        element.innerHTML = noviSimbol; // Promjena sadržaja elementa
    }
}
function azurirajCijenu(tag) {
    const query = document.querySelectorAll(tag)
    for (let element of query) {
        element.innerHTML = (Number.parseFloat(element.innerHTML) * 1.16547).toFixed(2); // Promjena
        // sadržaja elementa (sadržaj dohvaćen s innerHTML je tipa string pa ga treba pretvoriti u broj,
        // zato koristimo parseFloat metodu)
    }
}
azurirajSadrzaj("symbol", "€"); // Promjena simbola svugdje gdje imamo klasu "symbol"
azurirajCijenu("u"); // Ažuriraj cijenu svugdje gdje imamo tag "u"

```

Primjer 3 - Manipulacija klasama

Ako DOM elementu želimo direktno mijenjati ili dodati klasu `class` onda koristimo `className` svojstvo, ne `classList` svojstvo.

Primjerice, želimo ažurirati klasu elementa s ID-om `prviDiv` iz `text-5xl` u `text-6xl`

```
<div id="prviDiv" class="text-5xl">
    Hi!
</div>
```

```
const element = document.querySelector('#prviDiv')
console.log(element.outerHTML); //Output: "<div class='text-5xl' id='prviDiv'> Hi! </div>"
element.className = "text-6xl" // Promjena klase elementa
console.log(element.outerHTML); //Output: "<div class='text-6xl' id='prviDiv'> Hi! </div>"
```

`className` služi za postavljanje i dohvaćanje cijelog atributa klase odabranog elementa. Za dodavanje, brisanje, promjenu i provjeru pojedine klase bolje je koristiti `classList` svojstvo.

Nad svojstvom `classList` mogu se pozvati dodatne metode koje nam olakšavaju manipulaciju klasom (`class`) elementa.

Metoda	Objašnjenje	Sintaksa
<code>add(className1, className2, ...)</code>	Dodaje jednu ili više CSS klase elementu	<code>element.classList.add(x);</code>
<code>contains(className)</code>	Provjerava sadrži li element određenu CSS klasu	<code>element.classList.contains(x);</code>
<code>remove(className1, className2, ...)</code>	Uklanja jednu ili više CSS klase iz elementa	<code>element.classList.remove(x);</code>
<code>replace(oldClassName, newClassName)</code>	Zamjenjuje postojeću CSS klasu s novom CSS klasom	<code>element.classList.replace(x, y);</code>
<code>toggle(className)</code>	Dodaje CSS klasu ako je element nema/uklanja ako je ima	<code>element.classList.toggle(x);</code>

```
<div id="mojID" class="text-6xl">
    Hello, World!
</div>
```

```
const element = document.querySelector('#mojID')
element.classList.add('italic'); // Dodajemo klasu "italic"
console.log(element.className); //Output: "text-6xl italic"

console.log(element.classList.contains('text-6xl')); // Output: true
element.classList.remove('text-6xl'); // Uklanjamo klasu "text-6xl"
console.log(element.className); //Output: "italic"

element.classList.replace('italic', 'underline'); // Zamjenjujemo klasu "italic" s "underline"
console.log(element.className); //Output: "underline"
element.classList.toggle('font-bold') // Dodajemo klasu "font-bold" jer je nema
console.log(element.className); //Output: "underline font-bold"
```

Vježba 2

EduCoder šifra: `books`

Naišli smo na staru web stranicu s bibliotekom knjiga. Na stranici su prikazane knjige u tablici. Želimo promijeniti stil tablice i ćelija kako bi bila nam bila preglednija.

Zadan je sljedeći CSS i HTML kôd:

```
<style>
    .slika {
        overflow: hidden;
        border-radius: 100%;
    }
    .tekst {
        color: black;
        font-size: 18px;
    }
    .tablica {
        width: 100%;
        background-color: white;
    }
    .celija {
        border: 1px solid #dddddd;
        text-align: left;
        padding: 8px;
    }
    .naslov {
        background-color: #00000065;
        font-size: 20px;
    }
    .velika {
        background-color: #4cb05065;
    }
    .broj-stranica {
        font-weight: bold;
    }
</style>
<table class="slika tekst">
    <tr>
        <th>Naslov</th>
        <th>Autor</th>
        <th>Broj stranica</th>
    </tr>
    <tr>
        <td>Harry Potter and the Philosopher's Stone</td>
        <td>J. K. Rowling</td>
        <td class="broj-stranica">500</td>
    </tr>
    <tr>
        <td>The Lord of the Rings</td>
        <td>J. R. R. Tolkien</td>
        <td class="broj-stranica">1077</td>
    </tr>
    <tr>
        <td>Don Quixote</td>
        <td>Miguel de Cervantes</td>
        <td class="broj-stranica">120</td>
    </tr>
</table>
```

Koristeći `querySelector` i `classList` metode dodajte "tablica", "celija" i "naslov" na odgovarajuće elemente, čeliji s najvećim brojem stranica dodajte klasu "velika".

Naslov	Autor	Broj stranica
Harry Potter and the Philosopher's Stone	J. K. Rowling	500
The Lord of the Rings	J. R. R. Tolkien	1077
Don Quixote	Miguel de Cervantes	120

Rješenje:

```
document.querySelector('table').classList.replace('slika', 'tablica'); // Zamjenjujemo klasu
// "slika" s "tablica"
document.querySelectorAll('th, td').forEach(element => { // Dodajemo klasu "celija" na sve
<th> i <td> elemente
    element.classList.add('celija');
});
document.querySelectorAll('th').forEach(element => { // Dodajemo klasu "naslov" na sve <th>
elemente
    element.classList.add('naslov');
});

let maxBrojStranica = 0;
let maxCelijaIndex = -1;
let query = document.querySelectorAll('.broj-stranica'); // Dohvaćamo sve elemente s klasom
// "broj-stranica"
query.forEach(celija, index) => { // Pronalazimo najveći broj stranica
    const brojStranica = parseInt(celija.innerHTML); // Pretvaramo sadržaj u broj
    if (brojStranica > maxBrojStranica) { // Ako je trenutni broj stranica veći od
maksimalnog
        maxBrojStranica = brojStranica; // Postavljamo novi maksimalni broj stranica
        maxCelijaIndex = index; // Postavljamo index najvećeg broja stranica
    }
};
query[maxCelijaIndex].classList.add('velika') // Dodajemo klasu "velika" na čeliju s
// najvećim brojem stranica
```

Primjer 4 - Dohvaćanje `child` i `sibling` elemenata.

Naučili smo dohvaćati elemente koristeći `querySelector` i `getElements` metode. Međutim, ponekad želimo dohvatiti djecu ili susjede određenog elementa. Za to možemo koristiti sljedeća svojstva:

```
childElementCount,
children,
lastElementChild,
nextElementSibling,
previousElementSibling
```

Imamo sljedeći HTML kôd:

```

<div>
    Hi!
</div>
<div id="mojID">
    <span>Hello</span>
    <span>, </span>
    <span>World!</span>
</div>
<div>
    Bye!
</div>

```

Upotrijebit ćemo `querySelector` metodu za dohvaćanje elementa s ID-om `mojID` te potom iskoristiti navedena svojstva za dohvaćanje djece i susjeda tog elementa.

```

const element = document.querySelector('#mojID')
// Dohvaćanje broja djece elementa
console.log(element.childElementCount); // Output: 3

// Dohvaćanje djece elementa
for (const child of element.children) {
    console.log(`child: ${child.outerHTML}`);
    // Output: "child: <span>Hello</span>"
    // Output: "child: <span>, </span>"
    // Output: "child: <span>World</span>"
}

// Ili tradicionalnom for petljom
for (let i = 0; i < element.childElementCount; i++) {
    console.log(`child: ${element.children.item(i).outerHTML}`);
    // Output: "child: <span>Hello</span>"
    // Output: "child: <span>, </span>"
    // Output: "child: <span>World</span>"
}

console.log(element.firstElementChild.outerHTML); // Output: <span>Hello</span>
console.log(element.lastElementChild.outerHTML); // Output: <span>World</span>

console.log(element.nextElementSibling.outerHTML); // Output: "<div class='text-5xl'> Bye!
</div>"
console.log(element.previousElementSibling.outerHTML); // Output: "<div class='text-5xl'> Hi!
</div>"
```

`children` svojstvo vraća `HTMLCollection` nad kojime se ne može pozvati `forEach` metoda. Međutim, imamo svojstvo `childElementCount` koje nam vraća broj djece elementa te omogućava iteraciju kroz djecu koristeći tradicionalnu `for` petlju i pristupanje pojedinom djetetu preko indexa metodom `item()`

Vježba 3

EduCoder šifra: `web_scraping`

Radimo na projektu analize podataka, gdje trebamo prikupiti informacije s više web stranica škola. Nažalost, te stranice nemaju svoj API za dohvaćanje podataka.

Možemo koristiti web scraping, tehniku koja se koristi za ekstrakciju podataka s web stranica. U te svrhe moramo dobro poznavati HTML strukturu stranice, kao i manipulaciju DOM elementima.

Zadan je sljedeći HTML kôd:

```
<div id="studenti">
  <div>
    <b>Ivo</b>
    <b>Ivić</b>
    <u class="email">ivoivic@skole.hr</u>
    <span>3</span>
  </div>
  <div>
    <b>Ana</b>
    <b>Anić</b>
    <span>5</span>
  </div>
  <div>
    <b>Maja</b>
    <b>Majić</b>
    <u class="email">majamajic@skole.hr</u>
    <span>none</span>
  </div>
  <div>
    <b>Marko</b>
    <b>Marić</b>
    <u class="email">markomaric@skole.hr</u>
    <span>1</span>
  </div>
</div>
```

Zadan je konstruktor `Student`:

```
function Student(ime, prezime, email, ocjena, opisnaOcjena) {
  this.ime = ime;
  this.prezime = prezime;
  this.email = email;
  this.ocjena = ocjena;
  this.opisnaOcjena = opisnaOcjena;
  this.oStudentu = () => console.log(` ${this.ime} ${this.prezime} s emailom ${this.email}
ima ocjenu ${this.opisnaOcjena}`)
}
```

1. Napišite funkciju `dodajStudente(id, poljeStudenata)` koja:

- Za svako dijete elementa s danim `id`-em
 - Ekstrahira podatke o *imenu, prezimenu, emailu i ocjeni* koristeći samo `firstElementChild`, `lastElementChild`, `nextElementSibling`, `previousElementSibling` i `classList` metode
 - Ako nedostaje *email*, postavlja ga na "nema podatka"

- Pretvara ocjenu u format: od "odličan" do "nedovoljan" za ocjene od 5 do 1, ili "nema ocjenu" za ostalo
 - Dodaje svakog studenta u polje studenti i vraća novoizgrađeno polje
2. Spremite sve studente koji imaju ocjenu u polje filtriraniStudenti.
- Za svakog studenta koji ima ocjenu, pozovite metodu ostudentu() (metoda već definirana u konstruktoru Student)

Napišite funkciju prosjekStudenata(poljeStudenata) koja vraća prosjek studenata:

- spremite u sumaOcjena varijablu sumu svih ocjena studenata koristeći reduce() metodu
- spremite u prosjek varijablu prosjek ocjena zaokruženo na dvije decimale

Konstruktor Student možete ažurirati ako je potrebno s dodatnim metodama ili ažurirati metodu ostudentu().

Rezultat:

```
dodajStudente('studenti');

// Output: "Ivo Ivić s emailom "ivoivic@gmail.com" ima ocjenu dobar"
// Output: "Ana Anić s emailom "nema podatka" ima ocjenu odličan"
// Output: "Marko Marić s emailom "markomaric@gmail.com" ima ocjenu nedovoljan"

console.log(`Prosjek ocjena studenata: ${prosjekStudenata(filtriraniStudenti)}`);

// Output: "Prosjek ocjena studenata: 3.00"
```

Rješenje:

```
function dodajStudente(html_element_id) {
    const elementiStudenata = document.getElementById(html_element_id).children;
    const poljeStudenata = [];
    // Iteriramo tradicionalnom for petljom kroz svu djecu elementa s danim ID-em
    (html_element_id)
    // Kod počiva na pretpostavci da su svi elementi u polju redom: ime, prezime, email,
    ocjena
    for (let i = 0; i < elementiStudenata.length; i++) {
        const student = elementiStudenata[i];

        const imeElement = student.firstChild;
        const ime = imeElement.innerHTML;
        const prezime = imeElement.nextElementSibling.innerHTML;

        const ocjenaElement = student.lastElementChild;
        const ocjena = student.lastElementChild.innerHTML;

        let opisnaOcjena = "nema ocjenu";
        let email = "nema podatka";

        if (ocjenaElement.previousElementSibling.classList.contains('email'))
            email = ocjenaElement.previousElementSibling.innerHTML;

        switch (ocjena) {
```

```

        case "5":
            opisnaOcjena = 'odličan';
            break;
        case "4":
            opisnaOcjena = 'vrlo dobar';
            break;
        case "3":
            opisnaOcjena = 'dobar';
            break;
        case "2":
            opisnaOcjena = 'dovoljan';
            break;
        case "1":
            opisnaOcjena = 'nedovoljan';
            break;
        default:
            opisnaOcjena = 'nema ocjenu';
            break;
    }
    // Pozivanjem konstruktora stvaramo novi objekt s ekstrahiranim podacima
    poljeStudenata.push(new Student(ime, prezime, email, ocjena, opisnaOcjena));
}
return poljeStudenata;
}

// Pozivamo funkciju za ID "studenti"
let studenti = dodajStudente('studenti');

// U polje filtriraniStudenti spremamo sve studente koji imaju ispravnu ocjenu
const filtriraniStudenti = studenti.filter(student => student.opisnaOcjena != "nema ocjenu");
filtriraniStudenti.forEach(student => student.oStudentu());

function prosjekStudenata(poljeStudenata) {
    let sumaOcjena = poljeStudenata.reduce((total, student) =>
total+Number.parseInt(student.ocjena), 0);
    let prosjek = (sumaOcjena/poljeStudenata.length).toFixed(2);
    return prosjek;
}

console.log(`Prosjek ocjena studenata: ${prosjekStudenata(filtriraniStudenti)}`);

```

1.4 Dodavanje i brisanje DOM elemenata

Dodavanje i brisanje elemenata omogućuje dinamičke izmjene stranice na temelju korisničkih akcija ili događaja.

Dosad smo naučili kako da dohvaćamo i mijenjamo elemente, međutim dodavanje novih elemenata je dosta nezgodno koristeći svojstvo `innerHTML`. Iz tog razloga postoje i metode za dodavanja, umetanje i brisanje elemenata:

Metoda	Objašnjenje	Sintaksa
<code>createElement()</code>	Stvara novi HTML element prema definiranom <code>tag</code> -u	<code>document.createElement(tagName)</code>
<code>append()</code>	Dodaje element(e)(<code>newElement</code>) kao posljedne dijete.	<code>element.append(child1, child2, ...)</code>
<code>prepend()</code>	Dodaje element(e)(<code>newElement</code>) kao prvo dijete.	<code>element.prepend(child1, child2, ...)</code>
<code>before()</code>	Dodaje element(e)(<code>newElement</code>) ispred odabranog elementa.	<code>element.before(newElement)</code>
<code>after()</code>	Dodaje element(e)(<code>newElement</code>) iza odabranog elementa.	<code>element.after(newElement)</code>
<code>insertAdjacentElement()</code>	Dodaje novi element u odabrani element, na zadalu poziciju (<code>position</code>).	<code>element.insertAdjacentElement(position, newElement)</code>
<code>insertAdjacentHTML()</code>	Dodaje HTML tekst u odabrani element, na zadalu poziciju (<code>position</code>).	<code>element.insertAdjacentHTML(position, html)</code>
<code>remove()</code>	Uklanja element iz DOM-a.	<code>element.remove()</code>
<code>replaceWith()</code>	Zamjenjuje odabrani element novim elementom (<code>newElement</code>).	<code>element.replaceWith(newElement)</code>

U kôdu za `insertAdjacentElement()` i `insertAdjacentHTML()` koristimo atribut `position` koji označava gdje se sadržaj dodaje, mora biti postavljen na jednu od sljedećih vrijednosti:

- `beforebegin`: prije elementa
- `afterbegin`: unutar elementa, prije njegovog prvog djeteta
- `beforeend`: unutar elementa, nakon njegovog posljednjeg djeteta
- `afterend`: nakon elementa

Primjer 5 - Stvaranje, dodavanje, brisanje i izmjena DOM elemenata

```
<style>
    #mojID {
        background-color: lightgray;
    }
    .hello-world {
        background-color: darkGray;
    }
</style>


<div class="hello-world">
        Hello, World!
    </div>
</div>


```

Dodavanje novih elemenata koristeći metode: `append()`, `prepend()`, `before()`, `after()`

```
const mojElement = document.getElementById('mojID');

// Stvaramo nove elemente
const divAppend = document.createElement('div')
const divPrepend = document.createElement('div')
const divAfter = document.createElement('div')
const divBefore = document.createElement('div')

// Postavljamo sadržaj novih elemenata
```

```

divAppend.innerHTML = 'divAppend'
divPrepend.innerHTML = 'divPrepend'
divAfter.innerHTML = 'divAfter'
divBefore.innerHTML = 'divBefore'

// Raspordeđujemo nove elemente
mojElement.append(divAppend);
mojElement.prepend(divPrepend);
mojElement.after(divAfter);
mojElement.before(divBefore);

```

Ili s metodom: `insertAdjacentElement()`

```

const mojElement = document.getElementById('mojID');

// Stvaramo nove elemente
const divAppend = document.createElement('div')
const divPrepend = document.createElement('div')
const divAfter = document.createElement('div')
const divBefore = document.createElement('div')

// Postavljamo sadržaj novih elemenata
divAppend.innerHTML = 'divAppend'
divPrepend.innerHTML = 'divPrepend'
divAfter.innerHTML = 'divAfter'
divBefore.innerHTML = 'divBefore'

// Raspordeđujemo nove elemente
mojElement.insertAdjacentElement("beforebegin", divBefore);
mojElement.insertAdjacentElement("afterbegin", divPrepend);
mojElement.insertAdjacentElement("beforeend", divAppend);
mojElement.insertAdjacentElement("afterend", divAfter);

```

Ili pak s metodom: `insertAdjacentHTML()`

```

const mojElement = document.getElementById('mojID');

// Stvaramo nove elemente u obliku HTML stringa
const divAppend = "<div> divAppend </div>"
const divPrepend = "<div> divPrepend </div>"
const divAfter = "<div> divAfter </div>"
const divBefore = "<div> divBefore </div>

// Raspordeđujemo nove elemente
mojElement.insertAdjacentHTML("beforebegin", divBefore);
mojElement.insertAdjacentHTML("afterbegin", divPrepend);
mojElement.insertAdjacentHTML("beforeend", divAppend);
mojElement.insertAdjacentHTML("afterend", divAfter);

```

```
divBefore  
divPrepend  
Hello, World!  
divAppend  
divAfter
```

Brisanje radimo jednostavno koristeći metodu `remove()`. Na primjer, brisanje prvog `child` elementa, `div`-a gdje je `id = "mojDiv"`

```
const elementZaBrisanje = mojElement.firstElementChild;  
elementZaBrisanje.remove()
```

```
divBefore  
Hello, World!  
divAppend  
divAfter
```

Izmjena `div` elementa nakon `div`-a gdje je `id = "mojDiv"`:

```
const elementZaMijenjanje = mojElement.nextElementSibling;  
  
const newBoldElement = document.createElement('b')  
newBoldElement.innerHTML = "newBoldElement"  
  
elementZaMijenjanje.replaceWith(newBoldElement)
```

```
divBefore  
Hello, World!  
divAppend  
newBoldElement
```

Vježba 4

EduCoder šifra: `html_from_object`

U web programiranju često ćemo morati grafički prikazati podatke iz dinamičkih struktura i različitih izvora podataka. Ako se mijenjaju samo pojedinačne vrijednosti u strukturi, dovoljno je samo izmijeniti postojeće definirane HTML elemente. Međutim, ako se struktura mijenja, ili se povećava/smanjuje količina podataka, tada je potrebno dinamički i dodavati/brisati HTML elemente.

U praksi često za nas ove probleme rješavaju razvojni okviri za JavaScript, ili neka biblioteka, ali je dobro znati kako stvari funkcioniraju "ispod haube".

Zadan je sljedeći HTML/CSS kôd:

```
<style>
```

```

#kupac {
    margin: 16px;
    padding: 16px;
    border-radius: 8px;
    background: #dedede;
    border: 1px solid darkgray;
    color: black;
    width: 400px;
    position: absolute;
    left: 50%;
    transform: translateX(-50%);
}

h1 {
    font-size: 32px;
    font-weight: bold;
    text-align: center;
}

hr {
    border: 1px solid black;
    margin: 12px 0px;
}

table {
    width: 100%;
}

th {
    text-align: left;
    border-bottom: 1px solid darkgray;
    padding-bottom: 4px;
}


```

</style>

<div id="kupac"></div>

Zadan je objekt `kupac`:

```

let kupac = {
    ime: "Ivo",
    prezime: "Ivić",
    adresa: {
        ulica: "Ulica 123",
        grad: "Pula",
        postanskiBroj: "52100",
    },
    kontakt: {
        telefon: "0911234567",
        email: "iivic@gmail.com",
    },
    narudzbe: [
        {
            stavke: [
                {
                    naziv: "Mobitel",
                    kolicina: 1,
                }
            ]
        }
    ]
}

```

```

        cijena: 300,
    },
    {
        naziv: "Slušalice",
        kolicina: 1,
        cijena: 20,
    },
    {
        naziv: "Punjač",
        kolicina: 2,
        cijena: 10,
    },
],
ukupnaCijena: function () {
    return this.stavke.reduce((ukupno, stavka) =>
ukupno+stavka.kolicina*stavka.cijena,0);
},
valuta: "eur",
},
],
};

```

Objekt treba prikazati u obliku HTML-a koristeći metode za dodavanje elemenata.

 Rezultat:

KUPAC

Ime i prezime: **Ivo Ivić**
Adresa: **Pula, 52100 - Ulica 123**
Email: **iivic@gmail.com** | Telefon: **0911234567**

Naziv	Kolicina	Cijena	Ukupno
Mobilni telefon	1	300	300
Slušalice	1	20	20
Punjač	2	10	20

Ukupno: **340 EUR**

Rješenje:

```
// Dohvaćamo element s ID-om "kupac"
const divKupac = document.getElementById("kupac");
// Dodajemo naslov "KUPAC" u div element
divKupac.insertAdjacentHTML("beforeend", "<h1>KUPAC</h1>")
// Dodajemo horizontalnu liniju
divKupac.append(document.createElement("hr"));
```

Kada smo dodali naslov i liniju, slijedi ispisivanje informacije o kupcu. Stvarat ćemo nove `div` elemente te im dodavati sadržaj koristeći `innerHTML` svojstvo.

```
const divImePrezime = document.createElement("div");
divImePrezime.innerHTML = `Ime i prezime: <b> ${kupac.ime} ${kupac.prezime} </b>`
divKupac.append(divImePrezime);

const divAdresa = document.createElement("div");
divAdresa.innerHTML = `Adresa: <b> ${kupac.adresa.grad}, ${kupac.adresa.postanskiBroj} - ${kupac.adresa.ulica}</b>`
divKupac.append(divAdresa);

const divKontakt = document.createElement("div");
divKontakt.innerHTML = `Email: <b> ${kupac.kontakt.email} </b> | Telefon: <b> ${kupac.kontakt.telefon} </b>`
divKupac.append(divKontakt);

// Dodajemo horizontalnu liniju
divKupac.append(document.createElement("hr"));
```

Nakon što smo dodali informacije o kupcu, slijedi dodavanje tablice s narudžbama. Stvorit ćemo prvo tablicu, a naslove stupaca dodati koristeći `insertAdjacentHTML()` metodu.

```
const tableNarudzbe = document.createElement("table");
const tableHeaders = document.createElement("tr");

tableHeaders.insertAdjacentHTML("beforeend", "<th>Naziv</th>")
tableHeaders.insertAdjacentHTML("beforeend", "<th>Kolicina</th>")
tableHeaders.insertAdjacentHTML("beforeend", "<th>Cijena</th>")
tableHeaders.insertAdjacentHTML("beforeend", "<th>Ukupno</th>")

tableNarudzbe.append(tableHeaders);
```

Nakon što smo dodali naslove stupaca, slijedi dodavanje redova tablice s podacima o narudžbama. Iterirat ćemo kroz sve stavke narudžbe i dodati ih u tablicu.

```

for (let stavka of kupac.narudzbe[0].stavke) {
  const tableRow = document.createElement("tr");
  tableRow.insertAdjacentHTML("beforeend", `<td>${stavka.naziv}</td>`)
  tableRow.insertAdjacentHTML("beforeend", `<td>${stavka.kolicina}</td>`)
  tableRow.insertAdjacentHTML("beforeend", `<td>${stavka.cijena}</td>`)
  tableRow.insertAdjacentHTML("beforeend", `<td>${stavka.cijena * stavka.kolicina}</td>`)
  tableNarudzbe.append(tableRow);
}

divKupac.append(tableNarudzbe);
// Još jedna horizontalna linija
divKupac.append(document.createElement("hr"));

```

Na kraju, dodajemo boldano ukupnu cijenu narudžbe.

```

const divUkupno = document.createElement("div");
divUkupno.innerHTML = `Ukupno: <b> ${kupac.narudzbe[0].ukupnaCijena()} </b>
${kupac.narudzbe[0].valuta.toUpperCase()}`
divKupac.append(divUkupno);

console.log(divKupac.outerHTML);

```

1.4 DOM events

DOM događaji (**eng. DOM events**) omogućuju JavaScriptu da reagira na korisničke akcije kao što su klikovi mišem, različite kretnje mišem, unos na tipkovnici i sl. Događaj se na `element` dodaje metodom `addEventListener(event, callbackFn)`.

- `callbackFn`: callback koji prima argument `event` a koji se odnosi na pozvani događaj. Da bi se moglo pristupati elementu nad kojim se pozvao `event`, koristi se svojstvo `target`.

Sintaksa ove callback funkcije u metodi `addEventListener` je sljedeća:

```

const element = document.querySelector('#elementID');

function callbackFn(event) {

  // Prevencija defaultne akcije (npr. spriječavanje slanja forme unutar <form> elementa)
  event.preventDefault();

  // Dohvaćanje svojstava: target, type, itd (najčešće se dohvaća target)
  const target = event.target;
  const eventType = event.type;

  // Neka akcija koja se izvršava kada se događaj aktivira
  console.log(`Event type: ${eventType}`);
  console.log(`Event target: ${target}`);

  // Primjer: change the background color of the element
  target.style.backgroundColor = 'yellow';
}

```

Uzmimo primjer gdje želimo promijeniti boju pozadine elementa kada se klikne na njega:

```
<button id="btn">Klikni me</button>
```

Prvo dohvaćamo element na koji želimo dodati događaj:

```
const btn = document.getElementById("btn");
```

Zatim dodajemo događaj klikanja na taj element:

```
// 1. način
btn.addEventListener("click", function (event) {
    console.log(event.target.outerHTML)
});
// Output: "<button id="btn">Klikni me</button>"
```

`callbackFn` se može pisati i kao arrow funkcija:

```
// 2. način
btn.addEventListener("click", (event) => {
    console.log(event.target.outerHTML)
});

// 3. način (kratki zapis jer je samo jedan argument)
btn.addEventListener("click", event => {
    console.log(event.target.outerHTML)
});

// 4. način (kratki zapis jer je samo jedan argument i jedna naredba)
btn.addEventListener("click", event => console.log(event.target.outerHTML));

// 5. način (callback funkcija je definirana izvan metode addEventListener)
const ispis = (event) => console.log(event.target.outerHTML);
btn.addEventListener("click", ispis);
```

`event` argument sadrži informacije o događaju koji se dogodio.

Napamet ih nema smisla učiti (osim najčešće korištenih poput `click`, `input`, `focus`...) jer se mogu lako pronaći na internetu, ovisno o potrebi.

Ima ih mnogo, neki od najčešće korištenih na webu su:

Metoda	Objašnjenje	Sintaksa	Primjer
click	Poziva se kada se klikne mišem na element.	<code>element.addEventListener('click', function() {})</code>	<code>button.addEventListener('click', function() { console.log('Kliknuto!'); })</code>
dblclick	Poziva se kada se dvaput klikne na element mišem.	<code>element.addEventListener('dblclick', function() {})</code>	<code>image.addEventListener('dblclick', function() { console.log('Dvaput kliknuto!'); })</code>
focus	Poziva se kada element dobije fokus.	<code>element.addEventListener('focus', function() {})</code>	<code>input.addEventListener('focus', function() { console.log('Fokusiranje!'); })</code>
focusin	Poziva se kada element ili njegovi potomci dobiju fokus.	<code>element.addEventListener('focusin', function() {})</code>	<code>div.addEventListener('focusin', function() { console.log('Fokusiranje!'); })</code>
focusout	Poziva se kada element ili njegovi potomci izgube fokus.	<code>element.addEventListener('focusout', function() {})</code>	<code>input.addEventListener('focusout', function() { console.log('Izgubio fokus!'); })</code>
blur	Poziva se kada element izgubi fokus.	<code>element.addEventListener('blur', function() {})</code>	<code>input.addEventListener('blur', function() { console.log('Izgubio fokus!'); })</code>
mousedown	Poziva se kada se pritisne miš na element.	<code>element.addEventListener('mousedown', function() {})</code>	<code>div.addEventListener('mousedown', function() { console.log('Miš pritisnut!'); })</code>
mouseenter	Poziva se kada miš uđe u element.	<code>element.addEventListener('mouseenter', function() {})</code>	<code>div.addEventListener('mouseenter', function() { console.log('Miš unutar elementa!'); })</code>
mouseleave	Poziva se kada miš napusti element.	<code>element.addEventListener('mouseleave', function() {})</code>	<code>div.addEventListener('mouseleave', function() { console.log('Miš izvan elementa!'); })</code>
mousemove	Poziva se kada se miš pomiče preko elementa.	<code>element.addEventListener('mousemove', function() {})</code>	<code>div.addEventListener('mousemove', function() { console.log('Miš se pomiče!'); })</code>
mouseout	Poziva se kada miš napusti element ili njegovog potomka.	<code>element.addEventListener('mouseout', function() {})</code>	<code>div.addEventListener('mouseout', function() { console.log('Miš napustio element!'); })</code>
mouseover	Poziva se kada miš uđe u element ili njegovog potomka.	<code>element.addEventListener('mouseover', function() {})</code>	<code>div.addEventListener('mouseover', function() { console.log('Miš ušao u element!'); })</code>
mouseup	Poziva se kada se miš otpusti iznad elementa.	<code>element.addEventListener('mouseup', function() {})</code>	<code>div.addEventListener('mouseup', function() { console.log('Miš otpušten!'); })</code>
input	Poziva se kada se promijeni vrijednost input elementa, a korisnik i dalje ostaje u polju.	<code>element.addEventListener('input', function() {})</code>	<code>input.addEventListener('input', function() { console.log('Vrijednost promijenjena!'); })</code>
change	Poziva se kada se promijeni vrijednost elementa, a korisnik se miče od polja.	<code>element.addEventListener('change', function() {})</code>	<code>inputElement.addEventListener('change', function() { console.log('Vrijednost promijenjena!'); })</code>

Kroz sljedeće primjere i vježbe ćemo pokazati kako se koriste DOM događaji u JavaScriptu.

U svim primjerima koristit ćemo sljedeći CSS kôd:

```
<style>  
div {  
    padding: 4px 16px;  
}  
input, button {  
    background: transparent;  
    border: 1px solid gray;
```

```

border-radius: 4px;
padding: 2px 8px;
margin: 4px 2px;
&:hover {
    background: #a9a9a950;
}
&:focus {
    background: #4cb05050;
}
&:active {
    background: #ffeb3c50;
}
}
</style>

```

Primjer 6 - click event

Dodajemo `input` polje i dva `button` elementa. Input polje predstavlja brojač, a dva button elementa povećavaju i smanjuju brojač za jedan.

`click` događaj smo rekli da se poziva kada se klikne na element jedanput.

Nemojte zaboraviti kopirati CSS kôd iznad.

```

<div>
    <button id="increaseBtn">+</button>
    <input type="number" disabled name="broj" value="0" />
    <button id="decreaseBtn">-</button>
</div>

```

Prvi korak je naravno dohvatanje elemenata:

```

const increaseBtn = document.getElementById("increaseBtn");
const decreaseBtn = document.getElementById("decreaseBtn");
const broj = document.getElementsByName("broj")[0];

```

Zatim dodajemo naredbe (inkrement/dekrement) na `click` event za oba buttona:

```

increaseBtn.addEventListener("click", () => broj.value++) // povećava brojač za 1
decreaseBtn.addEventListener("click", () => broj.value--) // smanjuje brojač za 1

```

Vježba 5

EduCoder Šifra: `methods_to_methods`

Želimo napraviti aplikaciju koja će omogućiti korisniku da unese podatke o korisniku (ime, prezime, email) i da ih dodaje u listu korisnika. Korisnik može dodavati korisnike na početak ili kraj liste, te ih može brisati s početka ili kraja liste.

Koristit ćemo dobro poznate metode `Array` objekta. Za dodavanje korisnika koristit ćemo metode `push()` i `unshift()`, a za brisanje korisnika metode `pop()` i `shift()`.

Zadan je sljedeći HTML kôd:

```

<div id="forma">
    Ime: <input type="text" name="Ime" placeholder="Ime..." />
    Prezime: <input type="text" name="Prezime" placeholder="Prezime..." />
    Email: <input type="text" name="Email" placeholder="Email..." />
</div>
<div>
    <!--Svaku metodu predstaviti ćemo zasebnim gumbom-->
    <button id="push">push</button>
    <button id="pop">pop</button>
    <button id="unshift">unshift</button>
    <button id="shift">shift</button>
</div>
<div style="font-size: 24px;"><b>Korisnici:</b></div>
<div id="lista">
</div>

```

Zadatak je napisati implementacije funkcija: `dohvativrijednosti()`, `dodajNoviElement(pozicija)`, `ukloniElement(pozicija)` i dodati eventListenere za svaki button.

```

const inputs = document.getElementsByTagName('input');
const lista = document.getElementById('lista');
const forma = document.getElementById('forma');

const btn_push = document.getElementById('push');
const btn_pop = document.getElementById('pop');
const btn_unshift = document.getElementById('unshift');
const btn_shift = document.getElementById('shift');

let emailPolje = []

function dohvativrijednosti() {
    // Pseudokod:

    // Funkcija dohvaća vrijednosti iz "inputs" i vraća novi formatirani div element
    // Koristeći "for of" petlju ili "forEach" metodu iterira se kroz "inputs" (name, value)
    za svaki input
    // Ako je input prazan, defaultna vrijednost je "blank"
    return div;
    /* Primjer div-a:
        <div>
            <b>Ime</b>: Ivan
            <b>Prezime</b>: Ivić
            <b>Email</b>: iivic@gmail.com
        </div>
    */
}

function dodajNoviElement(pozicija) {
    // Pseudokod:

    // Funkcija dodaje novi element ovisno o poziciji ("push", "unshift") switch(pozicija)
    // vrijednost elementa dohvaća pomoću funkcije dohvativrijednosti()
    // Prije dodavanja provjerava je li email već dodan, ako je:
    // dodaje upozorenje "<div id='upozorenje' style='color: red;'>Email već postoji!</div>"

```

```

    // inače dodaje novi element u polje i html te miče upozorenje
}
function ukloniElement(pozicija) {
    // Pseudokod:

    // Funkcija briše element ovisno o poziciji ("pop", "shift") switch(pozicija)
    // Prije brisanja provjerava postoji li element
    // Ako postoji briše element iz polja
    // Miče upozorenje bez obzira postoji li element ili ne
}

//dodati eventListener-e za svaki button

```

Rezultat:

Ime: Prezime: Email:

Email već postoji!

Korisnici:

Ime: Marko Prezime: Marić Email: mmarić@gmail.com

Ime: Ana Prezime: Anić Email: aanic@gmail.com

Ime: Ivan Prezime: Ivanić Email: iivanic@gmail.com

Ime: Zoran Prezime: blank Email: zzoric@gmail.com

Rješenje:

```

// Dohvaćamo sve elemente
const inputs = document.getElementsByTagName('input');
const lista = document.getElementById('lista');
const forma = document.getElementById('forma');

const btn_push = document.getElementById('push');
const btn_pop = document.getElementById('pop');
const btn_unshift = document.getElementById('unshift');
const btn_shift = document.getElementById('shift');

let emailPolje = []
// Funkcija koja dohvaca vrijednosti iz input polja i vraća novi formatirani div element
function dohvatiVrijednosti() {
    let div = document.createElement("div");
    for (const input of inputs) {
        div.innerHTML += `<b>${input.name}</b>: `;
        if (input.value == "") {
            div.innerHTML += "blank ";
        } else {
            div.innerHTML += input.value + " ";
        }
    }
}

```

```

    }
    return div;
}

// Funkcija koja dodaje novi element ovisno o poziciji ("push", "unshift")
function dodajNoviElement(pozicija) {
    const email = document.getElementsByName('Email')[0].value;
    if (emailPolje.includes(email)) {
        if (document.getElementById("upozorenje") == undefined)
            forma.insertAdjacentHTML("afterend", `<div id="upozorenje" style="color: red;">Email
već postoji!</div>`);
    }
    else {
        // Brišemo upozorenje
        let element = document.getElementById("upozorenje");
        if (element) element.remove();

        let vrijednost = dohvatiVrijednosti();
        switch(pozicija) {
            case "push":
                lista.append(vrijednost);
                emailPolje.push(email);
                break;
            case "unshift":
                lista.prepend(vrijednost);
                emailPolje.unshift(email);
                break;
        }
    }
}

// Funkcija koja briše element ovisno o poziciji ("pop", "shift")
function ukloniElement(pozicija) {
    switch(pozicija) {
        case "shift":
            if (lista.firstChild != undefined) {
                lista.firstChild.remove()
                emailPolje.shift();
            }
            break;
        case "pop":
            if (lista.lastElementChild != undefined) {
                lista.lastElementChild.remove()
                emailPolje.pop();
            }
            break;
    }
    // Brišemo upozorenje
    let element = document.getElementById("upozorenje");
    if (element) element.remove();
}

// Dodajemo eventListenere za svaki button

```

```
btn_push.addEventListener("click", () => dodajNoviElement("push"))
btn_pop.addEventListener("click", () => ukloniElement("pop"))
btn_unshift.addEventListener("click", () => dodajNoviElement("unshift"))
btn_shift.addEventListener("click", () => ukloniElement("shift"))
```

Primjer 7 - focus events

`focus` familija događaja se poziva kada element dobiva/gubi fokus u nekom kontekstu. U primjeru ćemo koristiti `focus`, `focusin`, `focusout` i `blur` događaje.

Dodat ćemo dva input polja za ime i prezime, te jedno za broj godina. Kada se fokusira na polje, ispisuje se koji je element fokusiran.

Nemojte zaboraviti kopirati CSS kôd iznad.

```
<div id="inputi">
  <b>Ime:</b> <input id="ime" placeholder="Ime ..." />
  <b>Prezime:</b> <input id="prezime" placeholder="Prezime ..." />
</div>

<div>
  <b>Godine:</b> <input id="brojGodina" type="number" placeholder="Godina ..." />
</div>

<div>
  <b>Element event:</b> <span id="event"> </span>
</div>
```

```
const inputi = document.getElementById('inputi');
const inputBrojGodina = document.getElementById('brojGodina');

const span = document.getElementById('event');
// Focus event se poziva kada element dobije fokus
inputBrojGodina.addEventListener('focus', event => span.textContent = "focus: " +
event.target.outerHTML);
// Focusin event se poziva kada element ili njegovi potomci dobiju fokus
inputi.addEventListener('focusin', event => span.textContent = "focusin: " +
event.target.outerHTML);
// Focusout event se poziva kada element ili njegovi potomci izgube fokus
inputi.addEventListener('focusout', event => span.textContent = "focusout: " +
event.target.outerHTML);
// Blur event se poziva kada element izgubi fokus
inputBrojGodina.addEventListener('blur', event => span.textContent = "blur: " +
event.target.outerHTML);
```

`focus` i `blur` se pozivaju samo za trenutni element, ignoriraju potomke, dok se `focusin` i `focusout` pozivaju za element i svakog potomka zasebno.

Vježba 6

EduCoder šifra: `please_focus`

Imate zadana dva input polja za unos lozinke i ponovnu lozinku.

Kada je **input fokusiran**, ovisno o inputu treba pisati odgovarajući hint:

za **Lozinku**:

- Minimalna duljina lozinke mora biti 8
- Lozinka mora imati barem jedno veliko slovo
- Lozinka mora imati barem jedan broj

za **Ponovi lozinku**:

- Lozinke moraju biti iste
- Kada se **izade iz fokusa**, hint treba biti prazan.

Zadan je sljedeći kôd:

```
<div id="inputs">
    Lozinka: <input name="password" type="password"/>
    Ponovi lozinku: <input name="repeatPassword" type="password"/>
</div>
<div id="hint" style="color: green;">
</div>
```

```
const inputs = document.getElementById('inputs');
const hint = document.getElementById('hint');
/*
Vaš kôd ovdje...
*/
```

Rezultat:

Lozinka:

Ponovi lozinku:

- Lozinka mora imati najmanje 8 karaktera
- Lozinka mora imati barem jedno veliko slovo
- Lozinka mora imati barem jedan broj

Rješenje:

```
const inputs = document.getElementById('inputs');
const hint = document.getElementById('hint');
// Koristimo focusin i focusout jer se focus i blur pozivaju samo za trenutni element,
// ignoriraju potomke
// Focusin se propagira kroz sve potomke unutar DOM stabla.
inputs.addEventListener("focusin", (event) => {
    switch(event.target.name) {
        case "password":
            hint.innerHTML = `
                - Minimalna duljina lozinke mora biti 8 <br>
                - Lozinka mora imati barem jedno veliko slovo <br>
                - Lozinka mora imati barem jedan broj
            `;
```

```

        break;
    case "repeatPassword":
        hint.innerHTML = `

- Lozinke moraju biti iste
`;
        break;
    }
}
inputs.addEventListener("focusout", () => {
    hint.innerHTML = ``;
})

```

Primjer 8 - mouse events

`mouse` familija događaja se poziva kada se događa neka akcija mišem. U primjeru ćemo koristiti `mouseover`, `mouseout`, `mouseenter`, `mouseleave`, `mousedown`, `mouseup` i `click` događaje.

Zadan je sljedeći kôd:

```

<div id="buttons" style="background: lightblue;">
    <button id="btn_1">1</button>
    <button id="btn_2">2</button>
    <button id="btn_3">3</button>
    <button id="btn_4">4</button>
</div>
<!-- Ispisuje koji je događaj aktiviran --&gt;
&lt;b&gt;Mouse Event:&lt;/b&gt; &lt;span id="event"&gt; &lt;/span&gt;
</pre>

```

```

// div koji sadrži sve gume
const buttons = document.getElementById('buttons');
// 4 gumba, svaki ima svoj id
const btn_1 = document.getElementById('btn_1');
const btn_2 = document.getElementById('btn_2');
const btn_3 = document.getElementById('btn_3');
const btn_4 = document.getElementById('btn_4');

// dodajemo ih u polje
const btnList = [btn_1, btn_2, btn_3, btn_4]

// dohvaćamo span element za ispis događaja
const span = document.getElementById('event');

// dodavanje event listenera za različite mouse događaje
buttons.addEventListener('mouseover', event => span.textContent = "mouseover: " +
event.target.id);
buttons.addEventListener('mouseout', event => span.textContent = "mouseout: " +
event.target.id);

buttons.addEventListener('mouseenter', event => console.log("mouseenter: " +
event.target.id));
buttons.addEventListener('mouseleave', event => console.log("mouseleave: " +
event.target.id));

```

```
// dodavanje event listenera za mousedown, mouseup i click događaje te ispis događaja
for (const btn of btnList) {
  btn.addEventListener('mousedown', event => {
    console.log("[1] mousedown: " + event.target.id);
    span.textContent = "mousedown: " + event.target.id
  });
  btn.addEventListener('mouseup', event => {
    console.log("[2] mouseup: " + event.target.id);
    span.textContent = "mouseup: " + event.target.id
  });
  btn.addEventListener('click', event => console.log("[3] click: " + event.target.id));
}

```

`mouseenter` i `mouseleave` se pozivaju samo za trenutni element, ignoriraju potomke, dok se `mouseover` i `mouseout` pozivaju za element i svakog potomka zasebno.

`click` se uvijek poziva nakon `mousedown` i `mouseup` i točno tim redoslijedom.

Vježba 7

EduCoder šifra: `gallery`

Želimo izložiti galeriju slika na našu web stranicu. Svaka slika ima svoj naziv, umjetnika i godinu nastanka. Kada se mišem pređe preko slike, treba se prikazati opis slike (naziv, umjetnik, godina). Kada se mišem "izađe" sa slike, opis se mora maknuti.

Imamo već zadan CSS i HTML kôd, a potrebno je dodati event listenere za `mouseover` i `mouseleave` događaje te ispisati podatke o slici u opisu.

```
<style>
.gallery {
  display: flex;
  flex-wrap: wrap;
}

.artwork {
  transition: all 0.2s ease-in-out;
  position: relative;
  height: 200px;
  width: auto;
  border-radius: 4px;
  margin: 10px;
  &:hover {
    scale: 105%;
  }
}

.opis { padding: 16px; }

h1 { font-size: 24px; font-weight: bold; }
h2 { font-size: 20px; }
h3 { font-size: 14px; }
</style>

<div class="gallery">
```

```




</div>
<!--Opis slike koji je ažurirati kroz JavaScript DOM manipulacijom-->
<div class="opis">
    <h1></h1>
    <h2></h2>
    <h3></h3>
</div>

```

Naši podaci o slikama spremljeni su u polju `galerija`.

```

let galerija = [
{
    id: "artwork1",
    naziv: "Mona Lisa",
    umjetnik: "Leonardo da Vinci",
    godina: 1503
},
{
    id: "artwork2",
    naziv: "The Weeping Woman",
    umjetnik: "Pablo Picasso",
    godina: 1937
},
{
    id: "artwork3",
    naziv: "The Starry Night",
    umjetnik: " Vincent van Gogh",
    godina: 1889
}
]

```

Potrebno je dodati dva event listenera:

Pseudokod:

- kada se mišem pređe preko slike
 - treba iz polja `galerija` iščitati točne podatke i prikazati ih u opisu
- kada se mišem izadje iz slike
 - isprazniti opis

 Rezultat:



The Starry Night

Vincent van Gogh

1889.

Rješenje:

```
let galerija = [
  {
    id: "artwork1",
    naziv: "Mona Lisa",
    umjetnik: "Leonardo da Vinci",
    godina: 1503
  },
  {
    id: "artwork2",
    naziv: "The Weeping Woman",
    umjetnik: "Pablo Picasso",
    godina: 1937
  },
  {
    id: "artwork3",
    naziv: "The Starry Night",
    umjetnik: " Vincent van Gogh",
    godina: 1889
  }
]
// Dohvaćamo galeriju i opis
const gallery = document.getElementsByClassName("gallery")[0];
const opis = document.getElementsByClassName("opis")[0];

// Dodajemo event listenere za mouseover i mouseleave
gallery.addEventListener("mouseover", event => {
  // Pronalazimo točnu sliku
  let id = event.target.id;
  if (id == "") return;
  let artwork = galerija.find(g => g.id == id);

  // Ažuriramo opis
  opis.children[0].innerHTML = artwork.naziv;
```

```

    opis.children[1].innerHTML = artwork.umjetnik;
    opis.children[2].innerHTML = artwork.godina+".";
})
gallery.addEventListener("mouseleave", event => {
    // Brišemo opis
    opis.children[0].innerHTML = "";
    opis.children[1].innerHTML = "";
    opis.children[2].innerHTML = "";
})

```

Primjer 9 - `input` event

Kroz primjer ćemo pokazati kako koristiti `input` događaj. `input` događaj se poziva kada se promijeni vrijednost `input` elementa u kojeg korisnik unosi tekst.

Definirat ćemo dva input polja za unos lozinke i ponovnu lozinku. Kada korisnik unese lozinku, ispod polja će se prikazati poruka je li ponovljena lozinka jednaka prvoj.

```

<div>
    Lozinka: <input id="password" type="password"/>
    Ponovi lozinku: <input id="repeatPassword" type="password"/>
    Lozinke iste: <b id="same"></b>
</div>

```

I naš JavaScript kôd:

```

// Dohvaćamo input polja
const password = document.getElementById("password");
const repeatPassword = document.getElementById("repeatPassword");
const same = document.getElementById("same");

// Dodajemo event listenere za input događaj. U dijelu naredbe provjeravamo jednakost lozinki
password.addEventListener("input", event => {
    same.innerHTML = event.target.value == repeatPassword.value;
});
repeatPassword.addEventListener("input", event => {
    same.innerHTML = event.target.value == password.value;
});

```

Vježba 8

EduCoder šifra: `recommend`

Na web stranicama i aplikacijama često se implementira preporuka pretrage. Kada korisnik počne unositi pojам u polje za pretragu, prikazuju se rezultati koji odgovaraju unesenom pojmu.

U ovom primjeru implementirat ćemo preporuku pretrage za unos u polje za pretragu. Kada korisnik počne unositi pojam, prikazat će se rezultati koji odgovaraju unesenom pojmu. Rezultati se prikazuju u padajućem izborniku ispod polja za pretragu. Kada korisnik klikne na rezultat, unos u polje za pretragu postaje taj rezultat, a padajući izbornik se skriva.

Samo filtriranje smo do sada naučili kroz primjere iz prošlih skripti, sada ćemo sve ukomponirati koristeći HTML i CSS te JavaScript za manipulaciju našim `input` poljem odnosno tražilicom.

Za stilizirani input kopirajte CSS kôd od ranije.

Zadan je sljedeći HTML/CSS kôd:

```
<style>
    #searchInput {
        width: 100%;
        margin-bottom: 20px;
    }
    #searchResults {
        border: 1px solid #ccc;
        border-radius: 4px;
        padding: 10px;
    }
    #searchResults div {
        margin-bottom: 5px;
        cursor: pointer;
        &:hover {
            background-color: #f0f0f0;
        }
    }
</style>

<div>
    <input type="text" id="searchInput" placeholder="Traži...">
    <div id="searchResults"></div>
</div>
```

JavaScript kôd:

```
const inputField = document.getElementById('searchInput');
const resultsContainer = document.getElementById('searchResults');

// Podaci za preporuku
const data = [
    'JavaScript',
    'HTML',
    'CSS',
    'React',
    'Node.js',
    'Express.js',
    'MongoDB',
    'Vue.js',
    'Angular',
    'TypeScript'
];

function showResults(searchTerm) {
    // Vaš kôd ovdje...
}
```

```
//odgovarajući eventListener  
// Vaš kôd ovdje...
```

 Rezultat:

j

JavaScript
Node.js
Express.js
Vue.js

Rješenje:

```
const inputField = document.getElementById('searchInput');  
const resultsContainer = document.getElementById('searchResults');  
  
// Podaci za preporuku  
const data = [  
    'JavaScript',  
    'HTML',  
    'CSS',  
    'React',  
    'Node.js',  
    'Express.js',  
    'MongoDB',  
    'Vue.js',  
    'Angular',  
    'TypeScript'  
];  
  
function showResults(searchTerm) {  
    // Filtriramo podatke (filter, normalizacija i Array.includes metoda)  
    const filteredData = data.filter(item =>  
        item.toLowerCase().includes(searchTerm.toLowerCase()));  
  
    resultsContainer.innerHTML = '';  
  
    // Prikazujemo rezultate u padajućem izborniku (za svaki rezultat radimo div element)  
    filteredData.forEach(item => {  
        const stavka = document.createElement('div');  
        stavka.textContent = item;  
        stavka.addEventListener("click", event => {  
            inputField.value = event.target.textContent;  
            resultsContainer.innerHTML = '';  
        })  
        resultsContainer.appendChild(stavka);  
    });  
}
```

```

    }

    inputField.addEventListener('input', event => {
      const searchTerm = event.target.value;
      showResults(searchTerm);
  });

```

Samostalni zadatak za vježbu 8

EduCoder šifra: kosarica

Imate zadatak izraditi sučelje za **košaricu web trgovine**. Sučelje aplikacije sastoji se od polja za unos naziva i cijene proizvoda te gumba za dodavanje proizvoda u košaricu. Također, prikazuje se lista proizvoda u košarici s informacijama o nazivu, količini, cijeni po komadu te ukupnoj cijeni za taj proizvod.

Kada korisnik unese naziv i cijenu proizvoda te klikne na gumb `Dodaj artikl`, proizvod se dodaje u košaricu.

- Ako proizvod nema ime, dugme se ne može kliknuti, mora biti zatamnjeno/onemogućeno
- Ako se proizvod s istim imenom već nalazi u košarici, količina se povećava za `1`
- Ako je proizvod novi, dodaje se na listu
- Cijena proizvoda ne može ići ispod `0`

Korisnik može mijenjati količinu proizvoda u košarici koristeći gumbe `+` i `-` pored svakog proizvoda. Također, postoji opcija za uklanjanje proizvoda iz košarice klikom na gumb `Ukloni`.

Nakon svake promjene u košarici, ukupna cijena se automatski ažurira kako bi korisnik imao uvid u trenutni trošak.

Jedan primjer implementacije:

- Izrada konstruktora `Proizvod(naziv, kolicina, cijena)` koji ima atributе `naziv`, `kolicina`, `cijena` i metodu `ukupnaCijena()` koja vraća ukupnu cijenu proizvoda zaokruženu na dvije decimale.
- Izrada objekta `kosarica` koja sadrži:
 - atribut `proizvodi` - lista/polje proizvoda
 - metodu `dodajProizvod(proizvod)` - dodaje proizvod u polje `proizvodi` i HTML element
 - metodu `dodajFunkcionalnosti(naziv)` - dodaje funkcionalnosti (`eventListener`-i) za svaki proizvod u košarici:
 - povećanje količine
 - smanjenje količine
 - brisanje proizvoda
 - metodu `azurirajUkupnuCijenu()` - koristeći `reduce` nad poljem računa ukupnu cijenu svih proizvoda zaokruženu na dvije decimale

Primjer:

Košarica

Naziv proizvoda: Cijena proizvoda: Dodaj artikl

Naziv	Količina	Cijena	Ukupno	
Jabuka	- <input type="button" value="4"/> +	0.25 €	1.00 €	Ukloni
Banana	- <input type="button" value="12"/> +	0.12 €	1.44 €	Ukloni
Lubenica	- <input type="button" value="1"/> +	4.48 €	4.48 €	Ukloni
Kruh	- <input type="button" value="3"/> +	2.00 €	6.00 €	Ukloni

UKUPNO: 12.92 €

Možete napisati vlastiti HTML i CSS kôd ili koristiti sljedeći:

```
<style>
body {
    padding: 64px;
    font-family: Sans-Serif;
}
.main {
    display: flex;
    width: 100%;
    height: 100%;
    justify-content: center;
}
.card {
    overflow: hidden;
    width: 100%;
    padding: 32px;
    display: flex;
    flex-direction: column;
    background-color: rgb(205, 205, 205, 0.1);
    border-radius: 8px;
    color: #353535;
    box-shadow: 0px 0px 3px rgba(0, 0, 0, 0.2);
}
input, button {
    transition: all 0.2s ease-in-out;
```

```

padding: 8px 16px;
background: rgba(255, 255, 255, 0.5);
outline: none;
border: 1px solid rgba(0, 0, 0, 0.1);
border-radius: 8px;
&:hover {
    background: rgba(255, 255, 255, 1);
    border: 1px solid rgba(0, 0, 0, 0.25);
}
&:focus {
    background: rgba(255, 255, 255, 1);
    border: 1px solid rgba(0, 0, 0, 0.25);
}
}
button {
background-color: #a5d6a7;
&:hover {
    background: #83c683;
    cursor: pointer;
}
}
form {
overflow-y: hidden;
min-height: 48px;
overflow-x: auto;
padding: 0px 16px;
display: flex;
margin-top: 16px;
gap: 8px;
justify-content: space-between;
align-items: center;
font-size: 14px;
font-weight: bold;
}
.content {
padding: 16px 16px;
height: 100%;
display: flex;
flex-direction: column;
overflow-y: auto;
}
.flex {
width: 100%;
display: inline-grid;
grid-template-columns: repeat(4, minmax(0, 1fr));
}
.item-list {
overflow-y: auto;
overflow-x: hidden;
padding: none !important;
border-radius: 8px;
margin-top: 16px;
}
.item {

```

```

background-color: rgba(169, 169, 169, 0.2);
padding: 8px 16px;
align-items: center
}
.item:nth-child(2n) {
    background-color: rgba(169, 169, 169, 0.1);
}
.item-kolicina {
    width: 64px;
    text-align: center;
    margin-left: 8px;
    margin-right: 4px;
}
.item-kolicina-button {
    padding: 0px 4px;
    font-size: 24;
}
.item-kolicina-button:hover {
    padding: 0px 4px;
    color: #1f87e8;
    cursor: pointer;
}
.item-ukloni {
    color: #d81b43;
    font-size: 14;
    &:hover {
        text-decoration: underline;
        cursor: pointer;
    }
}
#ukupno {
    font-weight: normal;
}
.disabled {
    opacity: 0.5;
    cursor: not-allowed !important;
}

```

</style>

```

<div class="main">
    <div class="card">

        <h1 class="text-4xl mb-4 font-bold">Košarica</h1>

        <hr />

        <form>
            <label for="naziv_proizvoda">Naziv proizvoda:</label>
            <input type="text" name="naziv_proizvoda" id="naziv_proizvoda" placeholder="Upiši naziv proizvoda..." />
            <label for="cijena_proizvoda">Cijena proizvoda:</label>
            <input type="number" value="1" min="0" name="cijena_proizvoda" id="cijena_proizvoda" placeholder="Upiši cijenu proizvoda..." />
            <button disabled class="whitespace nowrap disabled" type="button" name="dodaj_button" id="dodaj_button">Dodaj artikl</button>
        </form>
    </div>
</div>

```

```

</form>

<hr />

<div class="content">
    <div class="flex" style="font-size: 18; color: #787878;">
        <b> Naziv </b>
        <b> Količina </b>
        <b> Cijena </b>
        <b> Ukupno </b>
    </div>

    <div id="item_list" class="item-list">
        <div class="flex item" id="item_Jabuka"><b>
            Jabuka
        </b>
        <div style="display: flex; align-items: center">
            <b class="item-kolicina-button item-kolicina-minus"
id="item_kolicina_minus_Jabuka">-</b>
            <input name="kolicina" id="item_kolicina_Jabuka" class="item-
kolicina" value="4" disabled=""
            <b class="item-kolicina-button item-kolicina-plus"
id="item_kolicina_plus_Jabuka">+</b>
        </div>
        <div>
            0.25 €
        </div>
        <div class="flex">
            <span id="item_ukupnaCijena_Jabuka">1.00</span> €
            <div class="item-ukloni" id="item_ukloni_Jabuka">
                Ukloni
                <div>
                </div>
            </div>
        </div>
    </div>
</div>

<hr />
<form>
    <div> UKUPNO: <span id="ukupno"></span> € </div>
</form>

</div>
</div>

```

2. JSON - JavaScript Object Notation

JSON (JavaScript Object Notation) je **string format za razmjenu podataka** koji je jednostavan čovjeku za razumijevanje, ali i računalu za procesiranje. JSON format često se koristi za slanje podataka između web poslužitelja i klijenta. Radi se o tekstualnom formatu koji se sastoji od parova ključ-vrijednost i nizova, vrlo slične sintakse kao i JavaScript objekti.

JSON format je neovisan o jeziku, što znači da se može koristiti u bilo kojem programskom jeziku. Međutim, svojom sintaksom podsjeća na JavaScript objekte i polja. Format je nastao ranih 2000-tih, a danas je de facto **standard za razmjenu podataka na webu**.

JSON podaci se mogu spremiti u datoteku s ekstenzijom `.json` ili kao tekstualni podaci u bazi podataka.



2.1 Struktura JSON-a

- Podaci u JSON zapisu su organizirani kao `ključ:vrijednost` parovi. Ključevi **moraju biti nizovi znakova**, a vrijednosti mogu biti bilo kojeg tipa podataka (string, broj, objekt, polje, boolean, null).
- Vitičaste zagrade `{}` koriste se za definiranje objekata.
- Uglate zagrade `[]` koriste se za definiranje polja/niza.
- Svaki par `ključ:vrijednost` odvojen je zarezom.
- Ključevi su nizovi znakova (stringovi) i **morate ih staviti u dvostrukе navodnike**.
- u JSON se [ne mogu pisati komentari](#).

Primjer JSON formata s 2 `ključ:vrijednost` para:

```
{  
    "kljuc1": "vrijednost1",  
    "kljuc2": "vrijednost2"  
}
```

Primjer ugniježđenog JSON formata:

```
{  
    "kljuc1": "vrijednost1",  
    "kljuc2": {  
        "kljuc3": "vrijednost3",  
        "kljuc4": "vrijednost4"  
    }  
}
```

Kao glavnu razliku uočite dvostrukе navodnike oko ključeva

Još jedan popularan format za razmjenu podataka je **XML** (Extensible Markup Language). Sintaksa XML-a je nešto složenija od JSON-a te više nalikuje HTML-u.

Primjer istog podataka u XML i JSON formatima:

XML:

```

<employees>
  <employee>
    <firstName>John</firstName> <lastName>Doe</lastName>
  </employee>
  <employee>
    <firstName>Anna</firstName> <lastName>Smith</lastName>
  </employee>
  <employee>
    <firstName>Peter</firstName> <lastName>Jones</lastName>
  </employee>
</employees>

```

JSON:

```

{
  "employees": [
    {
      "firstName": "John",
      "lastName": "Doe"
    },
    {
      "firstName": "Anna",
      "lastName": "Smith"
    },
    {
      "firstName": "Peter",
      "lastName": "Jones"
    }
  ]
}

```

2.2 Primjeri ispravnog i neispravnog JSON formata

Primjer JSON formata s podacima o osobi:

 Ispravan JSON format:

```

{
  "ime": "Ana",
  " prezime": "Anić",
  " godine": 25,
  " zaposlen": true,
  " adresa": {
    " ulica": "Ulica 1",
    " grad": "Zagreb",
    " poštanski broj": "10000"
  },
  " hobи": [ "čitanje", "pisanje", "plivanje"]
}

```

Primjer JSON formata s podacima o knjizi:

 Ispravan JSON format:

```
{  
    "naslov": "Harry Potter and the Philosopher's Stone",  
    "autor": "J.K. Rowling",  
    "godina izdanja": 1997,  
    "žanrovi": ["fantasy", "drama"],  
    "izdavač": {  
        "naziv": "Bloomsbury",  
        "lokacija": "London"  
    }  
}
```

Postoji i mogućnost da se JSON podaci nalaze u polju, bez omotavanja vitičastim zagradama:

 Ispravan JSON format:

```
[  
    {  
        "ime": "Ana",  
        " prezime": "Anić",  
        " godine": 25  
    },  
    {  
        "ime": "Ivan",  
        " prezime": "Ivić",  
        " godine": 30  
    }  
]
```

Međutim ne možemo imati više objekata bez omotavanja u polje:

 Neispravan JSON format:

```
{  
    "ime": "Ana",  
    " prezime": "Anić",  
    " godine": 25  
},  
{  
    "ime": "Ivan",  
    " prezime": "Ivić",  
    " godine": 30  
}
```

Komentare nije moguće pisati u JSON formatu, jer JSON je strogo definiran format podataka.

 Neispravan JSON format:

```
{
    "ime": "Ana",
    " prezime": "Anić",
    " godine": 25,
    "adresa": "Zagreb" // Adresa stanovanja
}
```

Ključeve je potrebno uvijek staviti u dvostrukе navodnike:

 Neispravan JSON format:

```
{
    ime: "Ana",
    prezime: "Anić",
    godine: 25
}
```

Za razliku od JavaScript objekata, JSON ne podržava funkcije, niti metode.

 Neispravan JSON format:

```
{
    "ime": "Ana",
    " prezime": "Anić",
    " godine": 25,
    "pozdravi": function() {
        return "Pozdrav!";
    }
}
```

2.3 Online alati za prikaz/validaciju JSON formata

Postoje online alati koji vam mogu pomoći u validaciji i prikazu JSON formata:

- [JSONFormatter.org](https://jsonformatter.org)
- [Codebeautify.org](https://codebeautify.org/jsonviewer)
- [JSONEditorOnline.org](https://jsoneditoronline.org)
- [JSONViwer Chrome ekstenzija](#)



2.4 Rad s JSON formatom u JavaScriptu

JSON je isključivo tekstualni format podataka u koji se ne mogu unositi metode odnosno funkcije.

Moguće je pretvoriti JSON format u JavaScript objekt i obrnuto, koristeći ugrađene metode `JSON.parse()` i `JSON.stringify()`.

2.4.1 `JSON.parse()`

Kada podaci pristignu s web servera, gotovo uvijek su u JSON formatu koji je tekstualni format. Da bismo s njima mogli raditi u JavaScriptu, moramo ih pretvoriti u JavaScript objekt. To radimo pomoću metode `JSON.parse()`.

Recimo da smo dobili JSON podatke o korisniku s web servera:

JSON:

```
{"ime": "Petar", " prezime": "Perković", "email": "pperkovic@unipu.hr", "lozinka": "98fd88c8fdc81c8efbd3a158007a97a6"}
```

Koristeći metodu `JSON.parse()`, možemo pretvoriti JSON podatke u JavaScript objekt:

Uočite da smo koristili dvostrukе navodnike oko ključeva, što je obavezno u JSON formatu. Međutim, cijeli JSON je string tako da ga ovdje moramo omotati u jednostrukе navodnike (`' '`) ili *backticks* navodnike.

JavaScript:

```
const korisnik = JSON.parse('{"ime": "Petar", "prezime": "Perković", "email": "pperkovic@unipu.hr", "lozinka": "98fd88c8fdc81c8efbd3a158007a97a6"}');

console.log(korisnik.ime); // Petar
console.log(korisnik.prezime); // Perković
console.log(korisnik.email); // pperkovic@unipu.hr
console.log(korisnik.lozinka) // 98fd88c8fdc81c8efbd3a158007a97a6
```

Primijetite da jednom kad parsiramo JSON u JavaScript objekt, možemo s njim raditi kao s bilo kojim drugim objektom u JavaScriptu.

Isto vrijedi i za JSON polje:

JSON:

```
{
  "naslov": "Harry Potter and the Philosopher's Stone",
  "autor": "J.K. Rowling",
  "godina izdanja": 1997,
  "žanrovi": ["fantasy", "drama"],
  "izdavač": {
    "naziv": "Bloomsbury",
    "lokacija": "London"
  }
}
```

JavaScript:

```
const knjiga = JSON.parse('{"naslov": "Harry Potter and the Philosopher\\'s Stone", "autor": "J.K. Rowling", "godina izdanja": 1997, "žanrovi": ["fantasy", "drama"], "izdavač": {"naziv": "Bloomsbury", "lokacija": "London"} }');
```

```

console.log(typeof knjiga); // object

console.log(knjiga.naslov); // Harry Potter and the Philosopher's Stone
console.log(knjiga.žanrovi); // ["fantasy", "drama"]
console.log(knjiga.izdavač.naziv); // Bloomsbury
console.log(knjiga.žanrovi[0]); // fantasy

for (let zanr of knjiga.žanrovi) {
    console.log(zanr);
}
// fantasy
// drama

```

2.4.1 JSON.stringify()

Metoda `JSON.stringify()` koristi se za pretvaranje JavaScript objekta u JSON format. Ova metoda je korisna kada želimo poslati podatke na web server, jer web serveri uglavnom očekuju JSON format.

Primjer pretvaranja JavaScript objekta u JSON format:

```

const korisnik = {
    ime: "Petar",
    prezime: "Perković",
    email: "pperkovic@gmail.com",
    lozinka: "super_sigurna_lozinka123"
};

console.log(typeof korisnik); // object

const korisnikJSON = JSON.stringify(korisnik);

console.log(typeof korisnikJSON); // string

console.log(korisnikJSON);

// Ispis:
'{"ime": "Petar", "prezime": "Perković", "email": "pperkovic@gmail.com", "lozinka": "super_sigurna_lozinka123"}'

```

Kako funkcije/metode nisu dozvoljene u JSON formatu, metoda `JSON.stringify()` će ih ignorirati prilikom pretvaranja objekta u JSON format, i ključ i vrijednost će biti izostavljeni.

```

const obj = {name: "John", age: function () {return 30;}, city: "New York"};

console.log(JSON.stringify(obj)); // '{"name": "John", "city": "New York"}'

```

Ako koristimo `JSON.stringify()` na objektu koji sadrži polje, metoda će automatski pretvoriti polje u JSON format.

```
const obj = {name: "John", age: 30, cars: ["Ford", "BMW", "Fiat"]};

console.log(JSON.stringify(obj)); // '{"name": "John", "age": 30, "cars": ["Ford", "BMW", "Fiat"]}'
```

Međutim ako ubacite u objekt `Date` objekt, metoda će ga automatski pretvoriti u string.

```
const obj = {name: "John", age: 30, birthdate: new Date()};

console.log(JSON.stringify(obj)); // {"name": "John", "age": 30, "birthdate": "2024-05-19T21:12:05.712Z"}
```

2.5 Lokalno čitanje JSON datoteka

JSON podaci se često koriste za razmjenu podataka između web poslužitelja i klijenta, ali se mogu koristiti i za lokalno čitanje i spremanje podataka.

U JavaScriptu, ovisno o okruženju, možemo koristiti različite metode za čitanje i spremanje JSON datoteka.

2.5.1 Node.js

U **Node.js** okruženju, možemo koristiti ugrađeni modul `fs` ([File System](#)) za čitanje i pisanje datoteka.

Funkcija `require` uključuje ugrađeni modul `fs`, i pišemo ju na početku datoteke.

Primjer čitanja JSON datoteke u Node.js okruženju. Pročitajmo datoteku `harry_potter.json` koja sadrži podatke o "Harry Potter" knjigama.

Node.js program možemo pokrenuti u terminalu naredbom `node index.js`, odnosno `node naziv_datoteke.js`.

```
{
  "naslov": "Harry Potter and the Philosopher's Stone",
  "autor": "J.K. Rowling",
  "godina izdanja": 1997,
  "žanrovi": ["fantasy", "drama"],
  "izdavač": {
    "naziv": "Bloomsbury",
    "lokacija": "London"
  }
}
```

```

const fs = require('fs');

fs.readFile('harry_potter.json', 'utf8', (err, data) => { // callback funkcija koja se poziva
  nakon što se datoteka pročita
    if (err) {
      console.log(err);
      return;
    }

    const podaci = JSON.parse(data); // pretvaranje JSON podataka u JavaScript objekt
    console.log(podaci); // ispis podataka
  });

```

```

console.log(podaci["harry_potter_books"][0]);
/*
{
  title: "Harry Potter and the Philosopher's Stone",
  author: 'J.K. Rowling',
  publication_date: '1997-06-26',
  publisher: 'Bloomsbury',
  isbn: '978-0747532699',
  summary: 'Harry Potter discovers on his eleventh birthday that he is the orphaned son of
two powerful wizards and possesses unique magical powers of his own.'
}
*/

```

2.5.2 Web preglednik

U web pregledniku, možemo koristiti `XMLHttpRequest` ili `fetch` API za čitanje JSON datoteke.

Pokazat ćemo noviji `fetch` API, koji je jednostavniji za korištenje.

Detaljnije o `fetch` API-u bit će u posljednjem poglavlju o asinkronom programiranju.

Primjer čitanja JSON datoteke u web pregledniku koristeći `fetch` API:

```

fetch('harry_potter.json') // Putanja do JSON datoteke lokalno
  .then(response => response.json()) // Više o ovim koracima u sljedećem poglavlju
  .then(data => {
    console.log(data);
  })
  .catch(error => {
    console.log('Error:', error);
  });

```

Također, isto možemo postići koristeći ES6 `import` sintaksu:

```

import data from './harry_potter.json';

console.log(data);

```

Napomena: Direktno čitanje datoteka iz lokalnog sustava datoteka nije moguće iz sigurnosnih razloga. U stvarnim aplikacijama, JSON podaci se obično čitaju s web poslužitelja. Pokretanje lokalnog http servera često je dobar "workaround" za ovaj problem.

Vježba 9

EduCoder šifra: [JSON](#)

Ispravite greške u sljedećim JSON podacima:

```
{  
{  
  "books": [  
    {  
      "title": "To Kill a Mockingbird",  
      "author": "Harper Lee",  
      "genre": "Fiction",  
      "publishedYear": 1960,  
      "ISBN": "978-0-06-112008-4",  
      "availableCopies": 4  
    },  
    {  
      "title": "1984",  
      "author": "George Orwell",  
      "genre": "Dystopian",  
      "publishedYear": 1949,  
      "ISBN": "978-0-452-28423-4",  
      "availableCopies": 6  
    },  
    {  
      "title": "The Great Gatsby",  
      "author": "F. Scott Fitzgerald",  
      "genre": "Fiction",  
      "publishedYear": 1925,  
      "ISBN": "978-0-7432-7356-5",  
      "availableCopies": undefined  
    },  
  ]  
}
```

Nakon što ispravite greške, napišite funkciju `printBooks(JSONbooks)` koja prima JSON podatke o knjigama, pretvara ih u JavaScript objekt i ispisuje sve knjige u konzolu.

```
function printBooks(JSONbooks) {  
  // Vaš kôd ovdje...  
}
```

3. Asinkrono programiranje

Posljednje poglavlje ove skripte, kao i gradivo ovog kolegija, odnosi se na asinkrono programiranje.

Asinkrono programiranje (eng. Asynchronous programming) je način programiranja u kojem se operacije izvršavaju neovisno jedna o drugoj, bez čekanja na završetak prethodne operacije. Ovo je posebno korisno kada se radi s operacijama koje zahtijevaju vrijeme, kao što su čitanje podataka s web poslužitelja, pisanje u bazu podataka i sl.

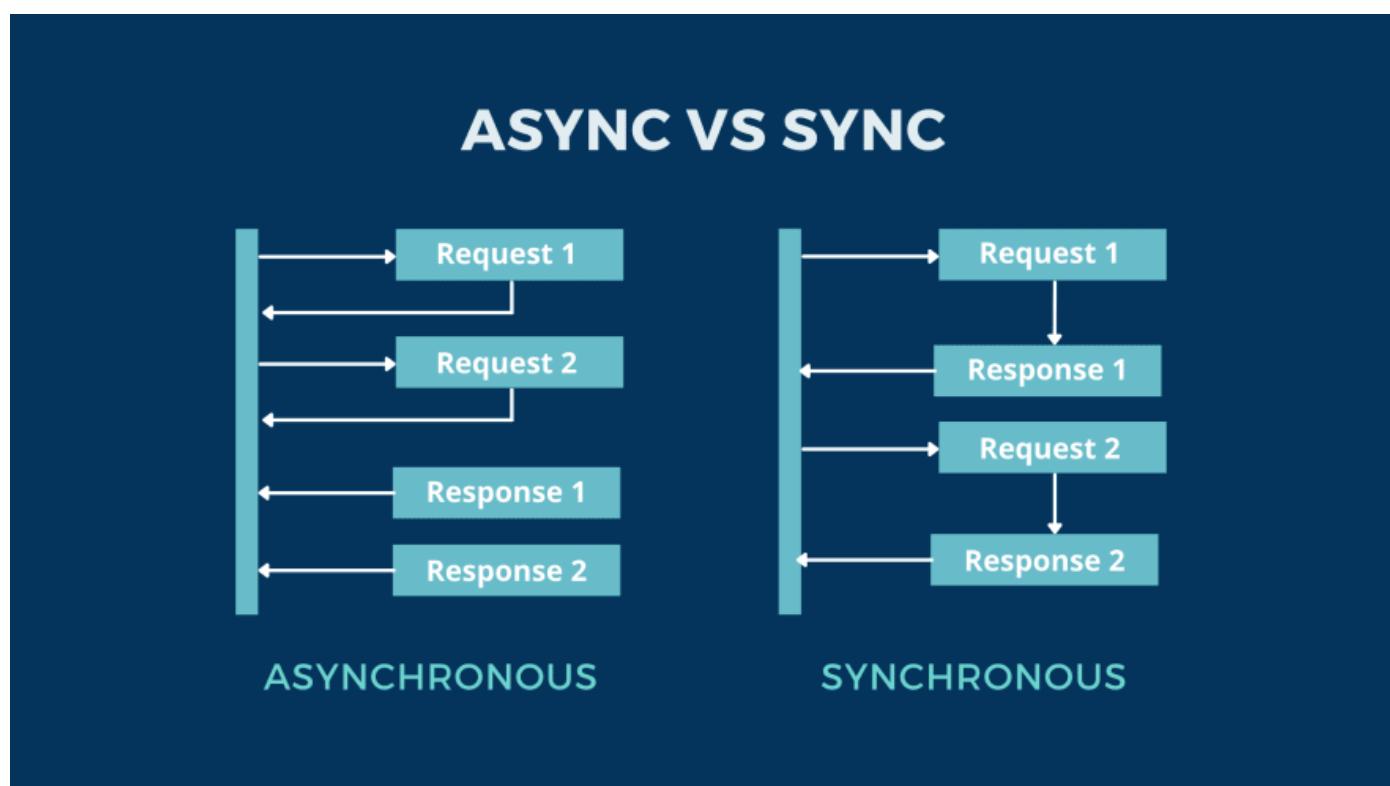
Recimo da želimo dohvatiti podatke s API-a (**eng. Application Programming Interface**) koji se nalazi na udaljenom web poslužitelju. Primjerice, radimo aplikaciju koja prikazuje vremensku prognozu za gradove diljem svijeta. Da bismo dohvatili podatke s API-a, moramo poslati zahtjev na web poslužitelj, pričekati odgovor i zatim prikazati podatke korisniku. Navedena operacija može potrajati nekoliko sekundi, ovisno o brzini interneta i udaljenosti web poslužitelja.

Međutim, protok podataka može biti spor (ili skroz puknuti) zbog različitih faktora, kao što su:

- Spora veza s internetom
- Preopterećenost web poslužitelja
- Dugotrajne operacije na poslužitelju
- Dugotrajne operacije na klijentskoj strani
- Blokirajuće operacije
- ISP (Internet Service Provider) problemi
- Vanjski događaji (npr. kibernetički napadi, prirodne katastrofe, vremenske neprilike)

Kako bi se izbjeglo blokiranje glavne dretve (**eng. main thread**), možemo koristiti asinkrono programiranje bazirano na konkurentnosti. Asinkrono programiranje omogućuje izvršavanje više operacija istovremeno, bez čekanja na završetak prethodne operacije.

Drugim riječima, ako naš korisnik čeka na odgovor s web poslužitelja, kod recimo dohvaćanja podataka o vremenskoj prognozi, ne želimo da mu se cijela aplikacija zamrzne dok čeka. Umjesto toga, želimo da korisnik može nastaviti koristiti aplikaciju dok se podaci dohvaćaju te mu na korektan način dati povratnu informaciju o tome što se događa.



Izvor: <https://dev.to/vinaykishore/how-does-asynchronous-javascript-work-behind-the-scenes-4bjl>

Asinkronim programiranjem bavit ćemo se intenzivnije na kolegijima: [Programsko inženjerstvo, Web aplikacije](#) i [Raspodijeljeni sustavi](#).

3.1 Razumijevanje asinkronog vs. sinkronog

Najjednostavnije rečeno, u **sinkronom programiranju**, operacije se izvršavaju jedna za drugom, redom. Kada se jedna operacija završi, tek tada se izvršava sljedeća operacija. Sve ispite, zadaće i vježbe do sad iz ovih skripti - pisali smo sinkrono.

Primjer:

Sinkrono programiranje:

```
console.log('Početak');
console.log('Operacija');
console.log('Kraj');
```

Ispisuje:

```
Početak
Operacija
Kraj
```

U **asinkronom programiranju**, kôd se može izvršavati "preko reda". Operacije mogu započinjati i završavati u različito vrijeme, neovisno o glavnom protoku programa. JavaScript je [single-threaded](#) jezik, što znači da se sve operacije izvršavaju na jednoj dretvi. Međutim, JS koristi asinkrono programiranje kako bi se izbjeglo blokiranje glavne dretve. Asinkrone potrebe uključuju radnje poput: čitanja podataka s web poslužitelja, pisanja u bazu podataka, čekanja na korisnički unos ([I/O operacije](#)).

Idemo simulirati čekanje dohvata podataka s nekog web poslužitelja. Recimo da je web server dosta udaljen i imamo spor internet, pa će dohvat podataka trajati 2 sekunde. Simulirat ćemo navedeno pomoću `setTimeout` funkcije koja prima 2 argumenta:

- callback funkciju koja se izvršava nakon određenog vremena i
- vrijeme čekanja u milisekundama.

Asinkrono programiranje:

```
function fetchData(callback) {
    setTimeout(() => { // simulacija dohvata podataka s web poslužitelja kroz setTimeout
        callback('Podaci su dohvaćeni');
    }, 2000);
}

console.log('Start');
fetchData((message) => {
    console.log(message);
});
console.log('End');
```

Kojim redoslijedom će se ispisati poruke?

► Spoiler Warning!

```
Start
End
Podaci su dohvaćeni
```

Zašto je ovako?

Sinkroni redoslijed izvršavanja:

1. Ispisuje se `Start`
2. Poziva se funkcija `fetchData` koja simulira dohvata podataka s web poslužitelja
3. Ispisuje se `End`
4. Nakon 2 sekunde, vraća se odgovor s "web poslužitelja" i ispisuje se `Podaci su dohvaćeni`

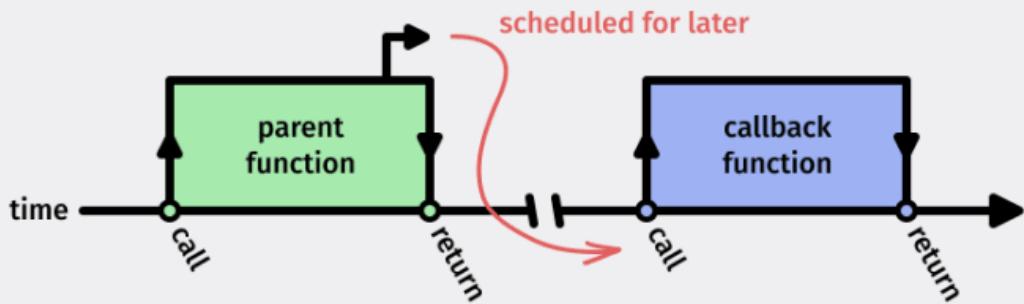
3.2 Asinkrone callback funkcije

Upoznali smo se s callback funkcijama u skripti PJS4 u kontekstu pomoćnih funkcija za obradu podataka kod `Array` objekta. Rekli smo da su to funkcije koje proslijedujemo kao argumente drugim funkcijama (u našem slučaju je bilo metodama `Array` objekta), a koje se pozivaju nakon završetka izvršavanja te funkcije/metode.

U kontekstu manipulacije DOM-om, koristili smo callback funkcije za dodavanje event listenera na HTML elemente.

U kontekstu asinkronog programiranja, callback funkcije koristimo za **rukovanje asinkronim operacijama**.

Asynchronous Callback



JavaScriptWithMarek.com

Izvor: <https://dev.to/marek/are-callbacks-always-asynchronous-bah>

U prethodnom primjeru vidjeli smo kako se koristi callback funkcija za rukovanje asinkronim operacijama.

Pojednostavimo primjer:

```

console.log('Start');

setTimeout(() => {
    console.log('Ovo je asinkroni callback');
}, 2000); // 2000 milisekundi = 2 sekundi

console.log('End');

```

Ispisuje:

```

Start
End
Ovo je asinkroni callback (nakon 2 sekunde)

```

Primjer iznad možemo podijeliti na sinkronu i asinkronu egzekuciju:

1. Sinkrona egzekucija:

- o Ispisuje se Start
- o Ispisuje se End

2. Asinkrona egzekucija:

- o Poziva se setTimeout funkcija koja postavlja timer na 2 sekunde i definira asinkronu callback funkciju u event queue-u
- o kada timer istekne, callback funkcija se stavlja u call stack i izvršava: ispisuje se ovo je asinkroni callback

3.3 Fetch API - dohvaćanje podataka s web poslužitelja

U JavaScriptu, `fetch` API je sučelje koje omogućuje asinkrono dohvaćanje resursa s web poslužitelja preko HTTP protokola. `fetch` API je moderna zamjena za zastarjelu `XMLHttpRequest` metodu.

`fetch` API koristi `Promise` objekte za rukovanje asinkronim operacijama. `Promise` objekt predstavlja eventualni rezultat asinkronog procesa i njegovo konačno stanje (rezoluciju ili odbijanje). Više o `Promise` objektima u sljedećem poglavljju, i nadolazećim kolegijima.

Ovaj API možemo direktno koristiti u web pregledniku ili u Node.js okruženju bez da uključujemo dodatne biblioteke. Bez obzira na to, postoje i biblioteke kao što su `axios`, `jQuery.ajax`, `superagent` koje pojednostavljaju rad s [HTTP zahtjevima](#).

Postoji mnoštvo servisa koji pružaju besplatne API-eve za testiranje i učenje asinkronog programiranja. Na primjer:

- [JSONPlaceholder](#)
- [PokeAPI](#)
- [TheDogAPI](#)
- [TheCatAPI](#)

Ogromnu listu razno-raznih API-eva možete pronaći [ovdje](#).

Napomena, neki od API-eva mogu biti zastarjeli, imati ograničenja ili biti nedostupni. **Prije slanja HTTP zahtjeva na API, provjerite dokumentaciju!**

`fetch` API možemo koristiti za dohvaćanje podataka s web poslužitelja, ali i lokalnih JSON datoteka (primjer s "Harry Potter" knjigama).

Sad ćemo pokazati kako koristiti `fetch` API za dohvaćanje podataka s web poslužitelja koristeći `JSONPlaceholder` servis.

```
fetch('https://jsonplaceholder.typicode.com/todos/1') // Obavezni argument je URL (ima i drugih međutim za sad ćemo preskočiti)
  .then(response => response.json()) // arrow funkcija koja pretvara odgovor u JSON format
  .then(json => console.log(json)) // arrow funkcija koja ispisuje JSON podatke

// Ispisuje:
// { userId: 1, id: 1, title: 'delectus aut autem', completed: false }
```

Isto možemo raspisati i bez arrow funkcija:

```
fetch('https://jsonplaceholder.typicode.com/todos/1')
  .then(function(response) { // response kao argument je rezultat fetch metode
    return response.json();
  })
  .then(function(json) { // json kao argument je rezultat prethodne then metode
    console.log(json);
  });
}
```

U gornjem primjeru, `fetch` funkcija prima URL kao argument i vraća `Promise` objekt. Nakon što se podaci dohvate, koristimo `then` metodu za rukovanje odgovorom. Prva `then` metoda pretvara odgovor u JSON format, a druga `then` metoda ispisuje JSON podatke.

Kôd možemo doslovno čitati kao: "**Dohvati** podatke s URL-a, **onda** pretvori odgovor u JSON format, **onda** ispiši JSON podatke".

`json()` metoda koristi se za parsiranje `Response` odgovora koji je u JSON formatu u JavaScript objekt. Sintaksa je: `response.json()`.

Osim `then()` metode, možemo koristiti i `catch()` metodu za rukovanje greškama. Ako dođe do greške prilikom dohvaćanja podataka, `catch()` metoda će uhvatiti grešku i ispisati je.

```
fetch('https://jsonplaceholder.typicode.com/todos/1')
  .then(response => response.json())
  .then(json => console.log(json))
  .catch(error => console.log('Greška:', error));
```

ili bez arrow funkcija...

```

fetch('https://jsonplaceholder.typicode.com/todos/1')
  .then(function(response) { // response kao argument je rezultat fetch metode
    return response.json();
  })
  .then(function(json) { // json kao argument je rezultat prethodne then metode
    console.log(json);
  })
  .catch(function(error) { // error kao argument je rezultat greške
    console.log('Greška:', error);
  });

```

Detalje o ovom API-ju i njegovim mogućnostima možete pronaći na [MDN web dokumentaciji](#).

Primijetite da smo u gornjem primjeru radili dvostrukе `then` metode. Ovo je korisno kada želimo izvršiti više operacija nakon dohvatanja podataka, međutim može doći do tzv. **callback hell**-a, odnosno dubokog gniježđenja callback funkcija, što može biti teško za održavanje i čitanje kôda, ali i sklono greškama.

U poglavlju o **Async/Await** sintaksi, pokazat ćemo kako možemo izbjegći callback hell i olakšati rad s asinkronim operacijama.

Vježba 10

EduCoder šifra: `cat_fact`

Ako rješavate ovaj zadatak u EduCoderu, ugasite automatsku evaluaciju budući da bi vas servis mogao blokirati zbog prevelikog broja zahtjeva.

Radite svoj web blog i želite korisnicima prikazati slučajno odabrane činjenice o mačkama. Za to koristite **TheCatAPI** servis koji pruža besplatne slučajne činjenice o mačkama. Potrebno je koristeći `fetch` API dohvatiti podatke s **TheCatAPI** servisa i ispisati slučajnu činjenicu o mačkama.

- <https://catfact.ninja/fact>

Kada ste uspješno dohvatili činjenicu o mačkama, pohranite ju u varijablu `catFact`. Dodajte novi HTML `<div>` element s ID-om `cat-fact` u vaš HTML dokument. Koristeći DOM manipulaciju dodijelite tekstualni sadržaj varijable `catFact` novom `<div>` elementu te dodajte stil po želji.

Primjer rezultata

Pregled

When a cat drinks, its tongue - which has tiny barbs on it -
scoops the liquid up backwards.

3.4 Promise objekt

U JavaScriptu, `Promise` objekt predstavlja **eventualni rezultat asinkronog procesa** i njegovo konačno stanje (**rezoluciju** ili **odbijanje**). `Promise` objekt može biti u jednom od tri stanja:

- **Pending:** Inicijalno stanje, očekuje se rezolucija ili odbijanje

- **Fulfilled:** Operacija je završena uspješno
- **Rejected:** Operacija je završena s greškom

`Promise` objekt ima tri metode: `then()`, `catch()` i `finally()`.

- `then()` metoda se koristi za rukovanje rezolucijom,
- `catch()` metoda se koristi za rukovanje odbijanjem.
- `finally()` metoda se koristi za izvršavanje kôda nakon što se `Promise` završi, bez obzira na rezultat.

Iako mnogima predstavlja muke, za potpuno razumijevanje asinkronog programiranja u JavaScriptu, `Promise` objekt je ključan. Dodatno, potrebno je razumjeti koncepte koje smo prošli u gradivu ovog kolegija, kao što su funkcije, callback funkcije, arrow funkcije, objekti, konstruktori, JSON format i sl.

`Promise` objekt možemo napraviti pozivanjem njegovog konstruktora, a kao argument prosljeđujemo callback funkciju s dva argumenta: `resolve` i `reject`.

`resolve()` se koristi za rezoluciju, `reject` za odbijanje.

Primjer izrade `Promise` objekta:

```
const promise = new Promise((resolve, reject) => {
  const uspjeh = true; // simulacija uspješne operacije, u pravilu je ovo rezultat neke
  asinkrone operacije

  if (uspjeh) {
    resolve('Operacija je uspješna!'); // označi operaciju kao uspješnu
  } else {
    reject('Operacija nije uspješna!'); // označi operaciju kao neuspješnu
  }
});
```

U gornjem primjeru, `promise` je `Promise` objekt koji simulira uspješnu operaciju.

- Ako je `uspjeh` varijabla `true`, operacija je uspješna i rezolucija se poziva s porukom "Operacija je uspješna!".
- Ako je `uspjeh` varijabla `false`, operacija nije uspješna i odbijanje se poziva s porukom "Operacija nije uspješna!".

Metode `resolve()` i `reject()` se mogu pozvati samo jednom. Nakon što se `Promise` objekt *rezolva* (upotpuni) ili odbije, ne može se ponovno *rezolvati* ili odbiti. Ove metode mogu primiti argumente:

```
resolve(vrijednost); // uspješna rezolucija s danom vrijednošću u argumentu

reject(greska); // odbijanje s danom greškom u argumentu (razlogom za odbijanje)
```

Rezultat `Promise` objekta ne možemo direktno ispisati koristeći `console.log()`. Umjesto toga, koristimo `then()` i `catch()` metode za rukovanje rezolucijom i odbijanjem.

```

promise
  .then((rezultat) => { // arrow funkcija koja se poziva nakon rezolucije
    console.log(rezultat); // Operacija je uspješna!
  })
  .catch((greska) => { // arrow funkcija koja se poziva nakon odbijanja
    console.log(greska); // 'Operacija nije uspješna!'
  });
  .finally(() => {
    console.log('Kraj operacije'); // Kraj operacije
  });

```

Prilikom korištenja `fetch` API-a za dohvaćanje podataka s web poslužitelja, `fetch` funkcija vraća `Promise` objekt. Kao takav, možemo koristiti `then()` i `catch()` metode za rukovanje rezolucijom ili odbijanjem direktno!

Primjer 10

EduCoder šifra: `bored`

Napomena, ako radite u EduCoderu, ugasite automatsku evaluaciju budući da bi vas servis mogao blokirati zbog prevelikog broja zahtjeva (limit: 100 zahtjeva svakih 15 minuta).

Idemo složiti malu aplikaciju koja će nam pritiskom na gumb prikazati neku zanimaciju koju bi mogli raditi u slobodno vrijeme, kada nam je dosadno. Podatke ćemo dohvatiti s [Bored API](https://bored-api.appbrewery.com/random) i to slučajnu aktivnost:

<https://bored-api.appbrewery.com/random>.

Koristit ćemo `fetch` API za dohvaćanje podataka s web poslužitelja i `Promise` objekt za rukovanje rezolucijom i odbijanjem.

Primjer JSON objekta kojeg dobivamo s API-a:

```
{
  "activity": "Learn Express.js",
  "availability": 0.25,
  "type": "education",
  "participants": 1,
  "price": 0.1,
  "accessibility": "Few to no challenges",
  "duration": "hours",
  "kidFriendly": true,
  "link": "https://expressjs.com/",
  "key": "3943506"
}
```

Prvo ćemo složiti HTML strukturu:

```

<style>
body {
  font-family: Arial, sans-serif;
  display: flex;
  flex-direction: column;
  align-items: center;
  justify-content: center;
  height: 100vh;
}

```

```

background-color: #f0f0f0;
}
h1 {
  color: #333;
}
#activity-container {
  margin-top: 20px;
  padding: 20px;
  background-color: #fff;
  border: 1px solid #ccc;
  border-radius: 5px;
  box-shadow: 0 2px 4px rgba(0,0,0,0.1);
}
button {
  padding: 10px 20px;
  font-size: 16px;
  color: #fff;
  background-color: #007bff;
  border: none;
  border-radius: 5px;
  cursor: pointer;
  transition: background-color 0.3s;
}
button:hover {
  background-color: #0056b3;
}

```

<body>

```

<h1>Što raditi kad nam je dosadno?</h1>
<!-- Gumb za dohvaćanje aktivnosti (iskoristit ćemo event listener) -->
<button id="fetch-activity-btn">Prikaži aktivnost</button>
<!-- Container za prikaz aktivnosti (proširit ćemo ga DOM manipulacijom) -->
<div id="activity-container"></div>

```

</body>

Prvi korak je dohvatiti gumb i dodati event listener koji će se pozvati pritiskom na gumb. To smo rekli da postižemo koristeći `addEventListener` metodu. Međutim, callback funkciju ćemo definirati izvana, a ona će biti upravo naša funkcija za dohvaćanje aktivnosti s API-a.

```

document.getElementById('fetch-activity-btn').addEventListener('click', fetchActivity); //  

dodajemo event listener na gumb i pozivamo funkciju fetchActivity

```

Sada definiramo funkciju `fetchActivity` koja će dohvatiti podatke s API-a i ispisati ih u HTML-u. Unutar funkcije koristimo `fetch` API za dohvaćanje podataka s web poslužitelja. Nakon što se podaci dohvate, koristimo `then()` metodu za rukovanje rezolucijom i `catch()` metodu za rukovanje odbijanjem.

```

function fetchActivity() {
    fetch('https://bored-api.appbrewery.com/random')
        .then(response => response.json())
        .catch(error => {
            console.log('Error fetching activity:', error);
        });
}

```

Sada ćemo dodati `then()` metodu za rukovanje našim podacima koje je odradila prva `then()` metoda. U ovaj metodi ćemo napisati callback funkciju koja će ispisati aktivnost u HTML-u. Rekli smo da koristimo `.json()` metodu za parsiranje JSON podataka u JavaScript objekt onda kada podaci dolaze s web poslužitelja.

```

document.getElementById('fetch-activity-btn').addEventListener('click', fetchActivity);

function fetchActivity() {
    fetch('https://bored-api.appbrewery.com/random')
        .then(response => response.json())
        .then(data => {
            const activityContainer = document.getElementById('activity-container'); // dohvaćamo container za prikaz aktivnosti i pohranjujemo varijable
            activityContainer.innerHTML =
                `

## Prijedlog aktivnosti:


                <p><strong>Aktivnost:</strong> ${data.activity}</p>
                <p><strong>Tip:</strong> ${data.type}</p>
                <p><strong>Broj sudionika:</strong> ${data.participants}</p>
                <p><strong>Cijena:</strong> ${data.price}</p>
                <p><strong>Pristupačnost:</strong> ${data.availability}</p>
            `;
        })
        .catch(error => {
            console.log('Error fetching activity:', error);
        });
}

```

 Rezultat:

Što raditi kad nam je dosadno?

Prikaži aktivnost

Prijedlog aktivnosti:

Aktivnost: Go see a movie in theaters with a few friends

Tip: social

Broj sudionika: 4

Cijena: 0.2

Pristupačnost: 0.3

To bi bilo što se tiče `Promise` objekta. U sljedećem poglavlju ćemo pokazati kako koristiti `async` i `await` sintaksu za olakšavanje rada s asinkronim operacijama.

`Promise` objekt ćemo detaljnije obrađivati na budućim kolegijima.

3.5 Async/Await

`async` i `await` sintaksa je novija sintaksa u JavaScriptu koja olakšava rad s asinkronim operacijama. JavaScript je dodao podršku za `async` i `await` sintaksu u ECMAScript 2017 (ES8).

`async` funkcija vraća `Promise` objekt, dok `await` čeka na rezoluciju `Promise` objekta. `await` se koristi samo unutar `async` funkcija.

`async` i `await` sintaksa je **syntactic sugar** za rad s `Promise` objektima. Ova sintaksa čini kôd čitljivijim i lakšim za održavanje, posebno kod dubokog gniježđenja callback funkcija (callback hell).

Uzmimo za primjer funkciju koja vraća novi `Promise` objekt koji se uspješno *rezolvira*.

```
function funkcija() {
    return Promise.resolve('Uspješna rezolucija');
}

funkcija().then(console.log); // Ispisuje: 'Uspješna rezolucija'
```

Isto možemo napisati koristeći `async` i `await` sintaksu. Rekli smo da `async` funkcija vraća `Promise` objekt, a `await` čeka na rezoluciju `Promise` objekta.

```
async function funkcija() { // async funkcija
    return 'Uspješna rezolucija';
}

console.log(await funkcija()); // Ispisuje: 'Uspješna rezolucija'
```

Osnovna `await` sintaksa izgleda ovako:

```
const rezultat = await promise;
```

await čeka na rezoluciju promise objekta. Kada se promise rezolva, await vraća rezultat. Ako promise odbije, await baca grešku.

Idemo primijeniti `async` i `await` sintaksu na primjeru s funkcijom `setTimeout` koja simulira asinkronu operaciju.

Pokazali smo kako se koristi `setTimeout` funkcija za simulaciju asinkronog procesa. U ovom primjeru, koristimo `setTimeout` funkciju unutar `async` funkcije i `await` čekamo na završetak procesa.

```
console.log('Start');

setTimeout(() => {
    console.log('Ovo je asinkroni callback');
}, 2000); // 2000 milisekundi = 2 sekundi

console.log('End');
```

Isto možemo napisati koristeći `async` i `await` sintaksu:

```
async function asinkronaFunkcija() {
    console.log('Start');

    await new Promise(resolve => setTimeout(resolve, 2000)); // čekamo 2 sekunde

    console.log('End');
}
```

U gornjem primjeru, `asinkronaFunkcija` je `async` funkcija koja čeka 2 sekunde prije nego što ispiše `End`. await čeka na rezoluciju `Promise` objekta koji se *rezolvira* nakon 2 sekunde.

Pokazat ćemo kako koristiti `async` i `await` sintaksu za dohvaćanje podataka s web poslužitelja koristeći `fetch` API. Kako više ne koristimo `then()` i `catch()` metode, ali svejedno imamo asinkrone pozive kroz `async` i `await` sintaksu, omotat ćemo kôd u `try` i `catch` blokove. `try` blok sadrži kôd koji može izazvati grešku, dok `catch` blok rukuje greškom.

Sintaksa:

```
try {
    // kôd koji može izazvati grešku
} catch (error) {
    // kôd koji rukuje greškom
}
```

Više o upravljanju iznimkama na budućim kolegijima.

Rješenje Primjera 10 s `async` i `await` sintaksom:

```
async function fetchActivity() {
    try {
```

```

        const response = await fetch('https://www.boredapi.com/api/activity'); // uočite
await i pohranu rezultata u varijablu
        const data = await response.json(); // .json() metoda vraća Promise objekt, koristimo
await za rezoluciju i pohranjujemo podatke u varijablu

        const activityContainer = document.getElementById('activity-container');
activityContainer.innerHTML =
    <h2>Prijedlog aktivnosti:</h2>
    <p><strong>Aktivnost:</strong> ${data.activity}</p>
    <p><strong>Tip:</strong> ${data.type}</p>
    <p><strong>Broj sudionika:</strong> ${data.participants}</p>
    <p><strong>Cijena:</strong> ${data.price}</p>
    <p><strong>Pristupačnost:</strong> ${data.accessibility}</p>
`;
} catch (error) { // catch block nam se ne mijenja mnogo
    console.log('Error fetching activity:', error);
}
}

```

KRAJ! WOOHO! 

Samostalni zadatak za vježbu 9

EduCoder šifra: `bitcoin`

Ako rješavate ovaj zadatak u EduCoderu, ugasite automatsku evaluaciju budući da bi vas servis mogao blokirati zbog prevelikog broja zahtjeva.

Student ste na Fakultetu informatike u Puli i polažete kolegij "Programiranje u skriptnim jezicima". Većinu svojeg slobodnog vremena provodite u EduCoderu marljivo rješavajući zadatke iz skripti. Pripremate se za ispit i jednostavno ne stignete pratiti vijesti o kriptovalutama iako ste čuli da je Bitcoin u posljednje vrijeme u velikom porastu. Kako EduCoder ima mehanizme za prevenciju varanja, ne možete otvoriti CoinMarketCap ili CoinGecko stranice kako biste provjerili trenutnu cijenu Bitcoina, a silno vas zanima koliko je Bitcoin vrijedan i je li pravo vrijeme za kupiti dip 😊.

Prilikom rješavanja zadatka, došli ste i do zadnje skripte napokon i naučili koristiti `fetch` API za dohvaćanje podataka s web poslužitelja. Odlučili ste iskoristiti svoje novo znanje i malo nadograditi EduCoder, u kojem sad provodite većinu slobodnog vremena, s jednim widgetom koji će vam u svakom trenutku prikazivati trenutnu cijenu Bitcoina!

Za dohvaćanje trenutne cijene Bitcoina koristite [CoinDesk API](#):

["https://api.coindesk.com/v1/bpi/currentprice.json"](https://api.coindesk.com/v1/bpi/currentprice.json) koji vraća JSON objekt s podacima o trenutnoj cijeni Bitcoina u USD, EUR i GBP valutama.

Vaš zadatak je napraviti widget koji će prikazivati trenutnu cijenu Bitcoina u USD, EUR i GBP valutama. Widget treba sadržavati trenutnu cijenu Bitcoina u svim valutama, te datum i vrijeme kada su podaci ažurirani. Dodatno, widget treba sadržavati gumb za ručno ažuriranje podataka.

Koristite `fetch` API za dohvaćanje podataka s web poslužitelja i `async` i `await` sintaksu za olakšavanje rada s asinkronim operacijama.

Primjer widgeta



Programiranje u skriptnim jezicima (PJS)

Nositelj: doc. dr. sc. Nikola Tarković
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

[1] JavaScript osnove



JavaScript je programski jezik često korišten u web programiranju. Inicijalno je bio namijenjen kako bi učinio web stranice interaktivnijima. Međutim, danas se koristi i za izradu server-side aplikacija, desktop aplikacija, mobilnih aplikacija itd.

41

Pregled

Trenutna cijena Bitcoina

USD: **69,294.087**

EUR: **63,808.213**

GBP: **54,530.081**

Last updated: May 20, 2024 20:40:33 UTC

Ažuriraj cijenu