Programiranje u skriptnim jezicima (PJS)

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

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

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

[4] Ugniježđene strukture i Napredne funkcije

"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: 21.5.2024.

Sadržaj

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 dohvatili `ulicu` moramo koristiti `split` metodu. Isti problem predstavlja `poštanski broj`.

Idemo problem riješiti **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. Ugniježđene 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 ugniježđenih 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 ugniježđenih 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 ugnježđivati, koliko god želimo. U praksi, nećemo ići dublje od 3-4 razine ugnježđivanja, 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 adresa 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 `narudzbe` 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",
  },
  narudzbe: {
    proizvodi: ["Mobitel", "Slušalice", "Punjač"],
    ukupnaCijena: 1500,
  },
};
```

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

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

Kako dohvatiti prvi proizvod iz niza proizvoda?

```
console.log(kupac.narudzbe.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 koje ćemo 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 potencijano 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 dohvatiti 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. popunjenost 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, ekosonder.
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", "Ekosonder", 250 eur/dan
>
> **Brod 2:** "Jahta", 2018. god, 35 čvorova, 300 KS, 8 osoba, "Tuš", "Hladnjak", "GPS", "Radio", "Kuhinja", "WC", "Utičnice za struju", "Tenda", "Gumenjak", 1000 eur/dan
>
> **Brod 3.** "Jedrilica", 2019. god, 12 čvorova, 50 KS, 4 osobe, "Tuš", "Hladnjak", "GPS", "Radio", "Kuhinja", "WC", "Utičnice 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 3x3. 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 kompliciranа 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 3x3x3, 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 ineženjerstva 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:

0	1	2			3			
Begin Again	*Soul*	0	1	2	0	1	2	
		Matrix	*Matrix Reloaded*	*Matrix Revolutions*	*Frozen*	*Frozen 2*	0	1
							Tangled	*Alladin*

> 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 3x3 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, videa, 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 mozzarella 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,
    },
  ],
};
```

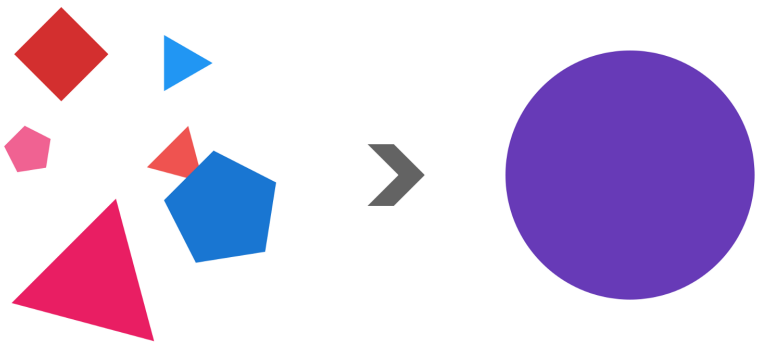
- korisnik završava s naručivanjem kada unese redni broj `0`. Tada se ispisuje trenutna narudžba i ukupna cijena narudžbe.
- dodajte `timestamp` u objekt `trenutna_narudzba` koji će sadržavati trenutni datum i vrijeme narudžbe.
- jednom kad je narudžba uspješno dodana obavijestite o tome korisnika funkcijom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

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

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

Naš kôd sada izgleda ovako:

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

function pronadiBor(stablo) {
    return stablo == "bor";
}

let bor = stabla.find(pronadiBor); // Callback funkciju pronadiBor() prosljeđujemo bez zagrada ()
console.log(bor); // Ispisuje "bor"
```

> Ima li kôd grešaka? Funkciju `pronadiBor` prosljeđujemo 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: Primjetite 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 funkcijsko programiranje** u skripti **PJS2**.

Opet ćemo definirati polje `brojevi`:

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

Ideja 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? Izmjenit 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 pronalazak 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 ugniježdenih 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 asortimana, 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

pretraživiProizvode(proizvodi, "ten"); // Ispisuje [{naziv: "Tenisice", cijena: 100, kategorija: "sport"}] // vraća polje s 1 elementom

pretraživiProizvode(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

pretraživiProizvode(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 funkcijskim 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 nasljeđuju `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 ekvilaventno 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 funnkcijskih 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 pronalazak 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 zadanom 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 ekvivalente funkcijske 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`, `programski_jezici`.

```
let kandidati = [
    {ime: "Ana", godine: 25, godine_iskustva: 3, strani_jezici: ["engleski", "njemački"],
programski_jezici: ["JavaScript", "Python"]},
    {ime: "Ivan", godine: 30, godine_iskustva: 5, strani_jezici: ["engleski",
"francuski"], programski_jezici: ["JavaScript", "Java"]},
    {ime: "Maja", godine: 22, godine_iskustva: 1, strani_jezici: ["engleski", "njemački"],
programski_jezici: ["JavaScript", "Python"]},
    {ime: "Marko", godine: 35, godine_iskustva: 7, strani_jezici: ["engleski", "njemački",
"francuski"], programski_jezici: ["JavaScript", "Python", "Java"]},
    {ime: "Eva", godine: 28, godine_iskustva: 4, strani_jezici: ["engleski", "njemački"],
programski_jezici: ["JavaScript", "Python"]},
    {ime: "Tomislav", godine: 40, godine_iskustva: 10, strani_jezici: ["engleski",
"njemački", "francuski", "španjolski"], programski_jezici: ["JavaScript", "Python",
"Java", "C++"]},
    {ime: "Lucija", godine: 26, godine_iskustva: 3, strani_jezici: ["engleski",
"španjolski"], programski_jezici: ["Python", "R"]},
    {ime: "Dario", godine: 31, godine_iskustva: 6, strani_jezici: ["engleski", "ruski"],
programski_jezici: ["C#", "Java", "Python"]},
    {ime: "Petra", godine: 29, godine_iskustva: 5, strani_jezici: ["engleski",
"talijanski", "francuski"], programski_jezici: ["JavaScript", "Swift"]},
    {ime: "Nikola", godine: 32, godine_iskustva: 8, strani_jezici: ["engleski",
"njemački"], programski_jezici: ["JavaScript", "Java", "Scala"]},
    {ime: "Lara", godine: 24, godine_iskustva: 2, strani_jezici: ["engleski", "kineski"],
programski_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(objekt.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 zapamtiti 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 konsturktora, `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 listenere. 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 `buton` 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 nastavli smo priču s nešto naprednijim 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 opcionalni 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 srodna 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 opcionalni 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 komplciraniji dio leži u samoj definiciji `callback` funkcije.

Callback funkcija prima četiri argumenta: `accumulator`, `currentValue`, `currentIndex` i `array`.

- `initialValue` predstavlja početnu vrijednost accumulatora. 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 funkcionira zato što `Math.max()` vraća veći broj od dva broja, a `reduce()` funkcija koristi taj rezultat kao novu vrijednost 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 temeljeu 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 **zadovaljaju 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 ugniježđene 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.

1. 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(studenti); // Ispisuje polje studenata s novim ključem broj_kolegija
```

4. Pohranite u varijablu `prosjekProsjeka` prosjek svih prosjeka studenata. Rezultat zaokržite na dvije decimale.

```
const prosjekProsjeka = /* Vaš kôd ovdje... */
console.log(prosjekProsjeka); // 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 otvori 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", "romobili@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 važnje 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... */
}
```