Univerzitet Singidunum
www.singidunum.ac.rs

Funkcionalno programiranje

Školska 2024/25 godina

Letnji semestar

Tema 10: JS asinhrono programiranje

Sadržaj

- Sinhronost i asinhronost
- Povratni poziv i “pakao povratnog poziva”
- Mehanizmi u jeziku JS za asinhrono programiranje
  - Promise
  - Asinhroni iteratori i generatori

Sinhronost i asinhronost

- U sinhronom modelu programiranja stvari se dešavaju **po jedna u jednom vremenskim trenutku**.
  - Kada pozovete funkciju koja se dugo izvršava, ceo program mora da čeka dok se ona ne završi - iz nje se vraća tek kada je njen posao završen. Sve dok se ona izvršava, ništa drugo se ne dešava u programu.
  - Dakle: Sinhrono je kada funkcija **na kraju svog izvršavanja vraća rezultat a dok se izvršava blokira onog ko je poziva**.
- U asinhronom modelu dozvoljeno je da se **istovremeno dešava više stvari**.
  - Kada započne neka akcija, program nastavlja da se izvršava. Kada se akcija završi, program se o tome obaveštava i dobija rezultat (na primer, podatke učitane sa diska).
  - Dakle: Asinhrono je kada funkcija **na kraju svog izvršavanja vraća rezultat, ali u toku svog izvršavanja ne blokira onog ko je poziva**.

Poređenje sinhronog i asinhronog izvršavanja

- Primer: program pribavlja sa mreže dva resursa, resurs A u jednom formatu i resurs B u drugom formatu, zatim resurs A transformiše u format resursa B i na kraju spaja resurse A i B.
  - Za očekivati je da kritična stvar u ovom programu bude vreme koje je potrebno za pribavljanje resursa
- **Sinhroni način**: Postoji jedna nit izvršavanja u kojoj se akcije izvršavaju sekvencijalno: pribavi se resurs A, zatim se pribavi resurs B, transformiše se resurs A u format resursa B i na kraju se spoje resursi A i B. Svaki naredni korak može da započne tek kada se prethodni završi.
- **Asinhroni način**: Program koji inicira akcije razbija pribavljanje resursa A i B na dve niti. U međuvremenu može da radi druge poslove.

Sinhrono izvršavanje

```
let sync1 = () => {
  return 'ja sam sync1 '
}
let sync2 = () => {
  return 'ja sam sync2 '
}
let sync3 = () => {
  return 'ja sam sync3 '
}
console.log(sync1())
console.log(sync2())
console.log(sync3())
```

- Ovo mi vrlo lako čitamo i iz redosleda pozivanja funkcija odmah nam je jasno šta će da bude ispisano na konzoli.

Asinhrono izvršavanje

```
let async1 = () => {
  return 'ja sam async1 '
}
let async2 = () => {
  return 'ja sam async2 '
}
let async3 = () => {
  return 'ja sam async3 '
}
setTimeout(function cb(){console.log (async1())}, 5000);
setTimeout(function cb(){console.log (async2())}, 4000);
setTimeout(function cb(){console.log (async3())}, 3000);
```

- Ovde moramo malo više da se skoncentrišemo da bismo uočili da će ispisi biti različiti od onoga što očekujemo na osnovu redosleda pozivanja funkcija.

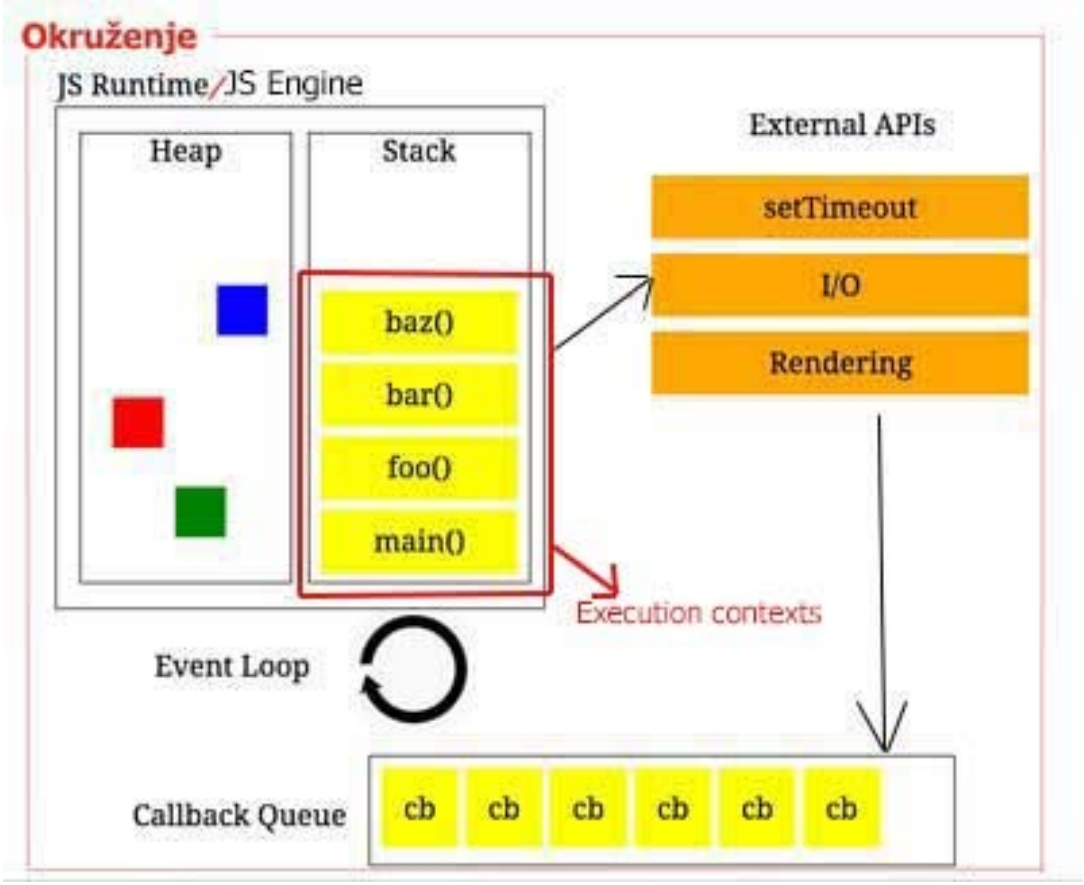
Problem asinhronog modela

- Problem u *asinhronom modelu je da se obezbedi ispravan redosled izvršavanja akcija*
  - U primeru pribavljanja i kombinovanja resursa to bi značilo da program mora da sačeka da se izvrši pribavljanje resursa A pre no što započne njegovu transformaciju u format resursa B, i da pre spajanja resursa sačeka da se izvrši transformacija resursa A i da se pribavi resurs B.
- Taj proces "uređivanja redosleda izvršavanja" zvani *sinhronizacija* je vrlo mukotrpan posao, podložan greškama.
- Ljudski mozak je "sinhrona mašina" – mi imamo vrlo ograničenu sposobnost asinhronog delovanja
- Zbog toga programski jezici obezbeđuju infrastrukturu za asinhrono programiranje koje programerima izgleda "sinhrono" .

Sinhrono i asinhrono u javaScript-u

- JavaScript je ***jedno-nitni*** (**single thread**) programski jezik i on sam ne može da izvršava više istovremenih radnji.
- JavaScript je skriptni jezik koji zahteva da se “smesti” u neki kontejner koji bi mu omogućio rad što uključuje i više istovremenih radnji.
- Kontejner se zove **JS Runtime Environment** i on obezbeđuje izvršavanje više “istovremenih ” radnji (*asihrono izvršavanje*).
  - Mehanizam izvršavanja “istovremenih ” radnji je **funkcija povratnog poziva** (**callback**)
  - U JS-u postoje dva tipa okruženja: okruženja za rad na klijentu (**browser**) i okruženja za rad na serveru (na primer, **node.js** ili **Deno**).

JS Runtime Environment



Funkcija povratnog poziva: primer izvršavanja

```
console.log('Pozdrav') // (1)
setTimeout(function cb(){console.log (' svima na ')}, 5000); // (2)
console.log ('Konferenciji JS 2023'); // (3)
```

- JS izvšava kod liniju po liniju:
- Prvo stavi na stek liniju (1) i izvrši je.
- Zatim stavlja poziva `setTimeout()` na vrh steka i izvršava ga.
- Nakon izvršenja, `setTimeout()` se sklanja sa steka , a njegova callback funkcija (cb) se prebacuje u sekciju webAPI i uključuje tajmer.
- Nakon toga JS engine nastavlja sa obradom linije (3) koju stavlja na steka i izvršava je.
- Kada istekne vreme i stek poziva se oslobodi, poziva se callback funkcija (`console.log (' svima na ')`) i nakon njenog izvršenja program se terminira.

- U primeru to znači sledeći ispis:

  Pozdrav
  Konferenciji JS 2023
  svima.

Asinhrono pozivanje funkcije u funkciji 1/2

```
// Lošom sinhronizacijom lako se dobije beskonačna petlju
let async1 = (fn) => {
//neke asinhrone operacije
    let rezPod = 'Ja sam async 1'
      setTimeout(function cb(){fn (rezPod)}, 2000);
      setTimeout(function cb(){async2()}, 1000);
      setTimeout(function cb(){async3()}, 1000);
       }
let async2 = () => {
let rezPod = 'Ja sam async 2'
    console.log(rezPod);
}
```

Asinhrono pozivanje funkcije u funkciji 2/2

```
let async3 = () => {
//neka asinhrona operacija
    let rezPod = 'Ja sam async 3'
    console.log(rezPod);
       setTimeout(function cb(){async1(console.log)}, 1000);
       }
async1(console.log)
```

Piramida propasti/ pakao povratnog poziva

- Termin “pakao povratnog poziva” alias “pirmida propasti” u programerskom rečniku odnosi se na situaciju u asinhronom programiranju gde se višestruki ugneždeni povratni pozivi koriste za rukovanje asinhronim operacijama.
- To su situacije u kojima su povratni pozivi duboko ugneždeni jedan u drugom, čineći strukturu koda složenom, teškom za čitanje, razumevanje i održavanje.

Piramida propasti: slika vredi više od hiljadu reči

```
const btn1 = document.querySelector(".btn-1");
const btn2 = document.querySelector(".btn-2");
const btn3 = document.querySelector(".btn-3");
const btn4 = document.querySelector(".btn-4");
const btn5 = document.querySelector(".btn-5");
const btn6 = document.querySelector(".btn-6");


btn1.addEventListener("click", function () {
  fadeOut(this, 500, function () {
    fadeIn(btn2, 500, function () {
      btn2.addEventListener("click", function () {
        fadeOut(this, 500, function () {
          fadeIn(btn3, 500, function () {
            btn3.addEventListener("click", function () {
              fadeOut(this, 500, function () {
                fadeIn(btn4, 500, function () {
                  btn4.addEventListener("click", function () {
                    fadeOut(this, 500, function () {
                      fadeIn(btn5, 500, function () {
                        btn5.addEventListener("click", function () {
                          fadeOut(this, 500, function () {
                            fadeIn(btn6, 500, function () {
                              btn6.addEventListener("click", function () {
                                fadeOut(this, 500, function () {
                                  fadeIn(document.body, 500, function () {
                                    alert("all callbacks completed");
                                  });
                                });
                              });
                            });
                          });
                        });
                      });
                    });
                  });
                });
              });
            });
          });
        });
      });
    });
  });
});
```

Zašto je to pakao i kako ga izbeći?

- **Zašto je pakao**
  - **Čitljivost i mogućnost održavanja**: Pakao povratnog poziva čini kod teškim za čitanje i razumevanje zbog njegove duboko ugneždene strukture. Teško je pratiti tok koda i održavati ga.
  - **Struktura i organizacija koda**: Formater u editoru radi uvlačenje svake linije zbog čega kod izgleda pretrpan i ometa se organizacija koda.
  - **Rukovanje greškama**: Rukovanje greškama postaje ozbiljan problem u ovakvom kodu.
- **Kako ga izbeći**
  - Koristiti infrastrukturu koju pruža JS

Mehanizmi za asinhrono programiranje u JS-u

Mehanizam **Promise**

Stanja asinhrone funkcije

- Asinhrona funkcija može imati dva moguća krajnja rezultata:
  - *uspešno izvršena operacija* , ili
  - *neuspešno izvršena opercija.*
- Između ta dva krajnja rezultata dešava se operacija koja može da traje i vrlo dugo.
- U takvoj situaciji nam je potreban instrument koji predstavlja stanje operacije i koji nam omogućuje da reagujemo na krajnji rezultat.

Mehanizam `Promise`

- Šta je i čemu služi
  - `Promise` je proksi za vrednost koja nije nužno poznata kada se `Promise` kreira.
  - Omogućava da se povežu rukovaoci sa vrednošću u slučaju uspešno izvršene operacije ili razlogom neuspeha poziva asinhrone funkcije.
  - Omogućava asinhronim metodama da vrate vrednosti kao da su sinhrone metode: umesto da odmah vrati konačnu vrednost, asinhroni metod odmah vraća obećanje da će obezbediti vrednost u nekom trenutku u budućnosti.

Objekat `Promise`

- Sintaksa koju je uveo ES2015
  - Sa svojim API-jem obezbedjuje bolji i **pregledniji način za organizovanje callback funkcija**.
  - Naročito pogodna za asinhrone operacije jer veoma liči na standardnu sinhronu sintaksu.
- `Promise`
  - Je JavaScript objekat koji predstavlja **“placeholder” za rezultate asihrone funkcije** sve dok traje izvršavanje asinhrone operacije.
  - Je **sinhrono vraćen objekat pri asinhronoj operaciji, koji predstavlja privremenu zamenu za moguće rezultate te asinhrone operacije.**
  - Umesto krajnje vrednosti daje “obećanje” da će dostaviti tu vrednost u nekom trenutku u budućnosti što omogućava da preko njega **vežemo rukovaoce za budući rezultat asinhrone operacije**.
  - Skoro da izjednačuje sinhrone i asinhrone operacije jer i sinhrone i asinhrone operacije mogu da vraćaju neku vrednosti, stim što sinhrone odmah vraćaju krajnji podatak a asinhrone “placeholder” za budući podatak.

Asinhrona funkcija i `Promise`

- Asinhrona funkcija može imati dva moguća krajnja rezultata:
  - *uspešno izvršena operacija* , ili
  - *neuspešno izvršena opercija.*
- `Promise` se može nalaziti u jednom od tri stanja:
  - **`Pending`** – stanje kada se **asinhrona radnja još uvek izvršava**
  - **`Fulfilled`** – stanje kada je **asinhrona radnja završena uspešno**
  - **`Rejected`** – stanje kada je **asinhrona radnja neuspešno završena** (greškom).

`Promise` mehanizam: struktura

- Sastoji se iz dva dela:

1. Kreiranje objekta **`Promise`** *unutar asinhrone funkcije*
2. Korišćenje kreiranog objekta **Promise** u kodu koji se nalazi *izvan asinhrone funkcije.*

Promise mehanizam: Kreiranje `Promise`

- Da bi se budući krajnji rezultat asinhrone funkcije zamenio sa `Promise` objektom, potrebno je da funkcija vrati novi `Promise` objekat :

```
function asinhronaFunkcija() {
    return new Promise(function (){...});
}
```

- Svakom novom objektu `Promise` se kroz parametar prosledjuje **funkcija** (*executor* funkcija) koja obradjuje samu asinhronu operaciju i buduće rezultate te asinhrone operacije.

Promise mehanizam: Kreiranje `Promise` – executor funkcija$_{1/2}$

- Vrlo čest scenario je obrada rezultata asinhrone operacije tako što se, u zavisnosti od ishoda asinhrone operacije, poziva jedna od dve funkcije koje se asinhronoj operaciji prosleđuju kao parametri:
  - Funkcija **`resolve()`**,
  - Funkcija **`reject()`**.

Promise mehanizam: Kreiranje `Promise` – executor funkcija$_{2/2}$

- Funkcija **`resolve()`** se poziva kada se obradjuje uspešno završena asinhrona operacija.
  - Parametar ove funkcije predstavlja dobijeni podatak iz uspešno završene operacije, stoga se funkcija `resolve()` koristi da kroz svoj parametar **prosledi rezultujući podatak odgovarajućoj metodi rukovaocu (hendler)** (npr. `then()` ili `Promise.all()`…)
- Funkcija **`reject()`** se poziva kada se pojavi problem sa izvršavanjem asinhrone operacije.
  - Ona kroz svoj parametar **prosledjuje razlog neuspešnosti asinhrone operacije odgovarajućem hendleru** (najčešće `catch()` metodi).

Kreiranje objekta `Promise` – primer executor funkcije

```
function asinhronaFunkcija() {
    return new Promise(function (resolve, reject) {

            //...kod asinhrone operacije...

            if (uspešna operacija) {
             resolve(result_value);
            } else {
             reject(error);
            }
        }
    );
}
```

Primer kreiranja objekta **Promise**

```
function asinhronaOperacija() {
    return new Promise((resolve, reject) => {
    /* U primeru se koristi setTimeout() za
       simuliranje asinhronog koda. Poziva se
       resolve() jer pretpostavljamo da  je
       asinhrona operacija uspešno izvršena.*/
      setTimeout(() => {
        resolve("Asinhrona operacija koja je
         trajala 2500 ms je uspela!");
      }, 2500);
   });
}
```

Primer korišćenja objekta **Promise**

```
console.log ('Sada pozivam asinhronu
operaciju koja će da traje 2500 ms')
let mojPrviPromise = asinhronaOperacija()
mojPrviPromise.then((porukaUspeha) => {
      /* porukaUspeha je ono što je prosleđeno
      funkciji resolve(). Ovde je string. */
  console.log(`Super! ${porukaUspeha}`);
});
console.log ('Dok asinhrona operacija radi,
ja se bavim drugim poslom')
```

Primer korišćenja objekta `Promise`: rezultat

Sada pozivam asinhronu operaciju koja će da traje 2500 ms

Dok asinhrona operacija radi, ja se bavim drugim poslom

Super! Asinhrona operacija koja je trajala 2500 ms je uspela!

`Promise` mehanizam: korišćenje

- Korišćenje `Promise` predstavlja obradu rezultata dobijenih po završetku asinhrone operacije.
- `Promise` reaguje na promenu svoga stanja pozivom callback funkcije.
- Postoji više ugrađenih metoda koje su namenjene ovom poslu od kojih ćemo mi da prikažemo sledeće:
  - `Promise.prototype.then()` – stanje **`fulfilled`**
  - `Promise.prototype.catch()` – stanje **`rejected`**
  - `Promise.resolve()`
  - `Promise.reject()`
  - `Promise.all()`
  - `Promise.race()`

Pravljenje `Promise` od drugih tipova - `Promise.resolve()`

- Može da primi tri različita tipa kao parametar, nakon čega vraća `Promise`:
  - **`Promise.resolve(vrednost)`**
    Ono što je prosleđeno kao `vrednost`, biće prosledjeno `then()` funkciji kroz njen parametar.
  - **`Promise.resolve(drugiPromise)`**
    Kroz parametar metode `then()` prosledjuje se **eventualno stanje** prihvaćenog `Promise` objekta. Ovo je najčešći slučaj, jer se koristi za konvertovanje `Promise` tipova kreiranih od strane drugih biblioteka.
  - **`Promise.resolve(thenableObject)`** može da prihvati kao parametar i tzv. `thenable object` (objekat ili funkcija koji definiše `then` metodu) .

`Promise.resolve()`: Primer

```
var pocetniPromise = Promise.resolve(33);
var drugiPromise = Promise.resolve(pocetniPromise);
  drugiPromise.then(function(value) {
  console.log('value: ' + value);
});
console.log('pocetniPromise === drugiPromise je '
          + (pocetniPromise === drugiPromise));
// Izlaz na konzoli:
// "pocetniPromise === drugiPromise je true"
// value: 33
/*  Redosled ispisa je drugačiji od redosleda pravljenja objekata  u
  kodu, zato što je then hendler pozvan asinhrono. */
```

Promise.prototype.then()

- Ukoliko je nakon promene stanja `Promise` u stanju **`fullfiled`**, poziva se metoda **`then()`** koja `prihvata` dva parametra tipa `Function`, parametar **`onFulfilled`** i parametar **`onRejected`** .
- Sintaksa

  `Promise.prototype.then(onFulfilled(), onRejected())`

  - Funkcija **`onFulfilled`** hendluje uspešno završenu asinhronu operaciju. Prihvata jedan parametar kroz koji joj se prosledjuje podatak dobijen asinhronom operacijom.
  - Funkcija **`onRejected`** hendluje neuspešno završenu asinhronu operaciju i takodje prihvata jedan parametar kroz koji joj se prosledjuje razlog neuspeha.

```
nekiPromise.then(function1(podatak) {// uspešna asinhrona operacija },
function2(razlog)  { //neuspešna asinhrona operacija });
```

Promise.prototype.then()

```
function makeRequest (method, url) { // asinhrona funkcija
  return new Promise(function (resolve, reject) {
    var xhr = new XMLHttpRequest();
    xhr.open(method, url);
    xhr.onload =  function() {
        if (xhr.status === 200) {
            resolve(xhr.response);
        } else {
            reject(new Error(xhr.statusText));
        }
    };
    xhr.onerror = () => reject(new Error("Network error"));
    xhr.send();
  });
}

// Deo izvan asinhrone funkcije:
makeRequest('GET', 'http://example.com')
  .then( function(data){console.log(data);},
         function(err){console.error(err);} )
```

Promise.prototype.catch()

- Ukoliko je nakon promene stanja `Promise` u stanju **`rejected`**, poziva se metoda **`catch()`:**

  `Promise.prototype.`**`catch`**`(onRejected())`

- Metoda `catch()` kao parametar prihvata callback funkciju (`onRejected()`), koja je zadužena za prihvatanje i obradu greške:

```
asinhronaFunkcija().then(result_value => {
                        ... }).catch(error => {
                        ... });
```

Promise.prototype.catch(): primer

```
function makeRequest (method, url) {
  return new Promise(function (resolve, reject) {
    var xhr = new XMLHttpRequest();
    xhr.open(method, url);
    xhr.onload =  function() {
            if (xhr.status === 200) {
                resolve(xhr.response);
            } else {
                reject(new Error(xhr.statusText));
            }
    };
    xhr.onerror = () => reject(new Error("Network error"));
    xhr.send();
  });
}

// Deo izvan asinhrone funkcije:

makeRequest('GET', 'http://example.com’)
     .then(function (data) {
         console.log(data);
     })
     .catch(function (err) {
         console.error('Imamo problem!', err);
     });
```

Rešenje za pakao povratnog poziva: ulančavanje `Promise`-a

- Metode `then()` i `catch()` **uvek vraćaju novi `Promise`.**
- `Promise` se mogu ulančavati.
- To je mehanizam da se na elegantan način reši problem sa mnogo ulančanih dogadjaja koje pozivaju callback funkcije (*callback hell*).

Ulančavanje **Promise**-a: primer 1/2

```
const promise = new Promise( function(resolve, reject) {
  setTimeout( () => {
    resolve( [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9] );
  }, 2000);
});

function ispisiRezultat(value) {
  console.log(value);
  return value;
}

function samoParni(array) {
  return array.filter( (value) => {
    return (value % 2) === 0;
  });
}
```

Ulančavanje `Promise`-a: primer 2/2

```
function sumaClanovaNiza(array) {
  return array.reduce( (a, b) => {
    return a + b;
  }, 0);
}

function errorHandler(err) {
  console.log("GREŠKA!");
  console.log(err);
}

promise
  .then(ispisiRezultat)      /* [0, 1, 2, 3, 4,
                                5, 6, 7, 8, 9] */
  .then(samoParni)
  .then(ispisiRezultat)      // [0, 2, 4, 6, 8]
  .then(sumaClanovaNiza)
  .then(ispisiRezultat)      // 20
  .catch(errorHandler);
```

Promise.resolve(thenableObject): primer

```
var p1 = Promise.resolve({
      then: function(onFulfill, onReject) {
      onFulfill('fulfilled!'); }
});
console.log(p1 instanceof Promise) // true,
// object kastovan na Promise

p1.then(function(v) {
    console.log(v); // "fulfilled!"
  }, function(e) {
    // ne poziva se
});
```

Promise.reject()

- Uzima za *"reason object"* `String` ili `Error`, a vraća `Promise` koji se nalazi u stanju `rejected` uz odgovarajući razlog odbijanja.

```
Promise.reject(new Error('Ja sam pogrešan'))
       .then(function() {/* deo za uspešno,
                            ne poziva se */},
  function(error) { console.log(error); });
```

Metoda `Promise.all()` 1/2

- Zadatak metode je obezbedi ponašanje nad listom `Promise` objekata, preciznije da vrati jedan `Promise` iz liste `Promise` objekata.
- Kao argument prima iterabilnu listu `Promise` objekata.
- *V*raća **jedan `Promise`** u trenutku **kada su svi `Promise` iz liste rezolvirani** (rešeni uspešno ili neuspešno).
- Metoda će vratiti `Promise` čak i kada joj se prosledi prazna lista ili element koji nije `Promise`.
- **Redosled izvršavanja (koji će objekat biti vraćen) liste `Promise` nije zagarantovan**, jedino je zagarantovano da će vratiti poslednji `Promise` ako su svi `Promise` iz liste u stanju `fullfiled`.

Metoda `Promise.all()` 2/2

- U slučaju da je jedan `promise` iz liste u stanju `rejected`, metoda `Promise.all()` vraća odbijeni `Promise` bez obzira da li u listi postoji neki `Promise` koji je uspešan.
  - Uz odbijen `Promise` se vraća i razlog odbijanja.
  - U slučaju da ima više odbijenih `Promise`-a, prosledjeni **razlog odnosi se na prvi odbijeni** `Promise`!
- `Promise.all()` se u svim slučajevima ponaša asinhrono, osim u slučaju **kada je umesto liste prosledjen prazan objekat - tada se ponaša sinhrono**.

Promise.all(): Primer 1

```
let p1 = Promise.resolve(3);
let p2 = 1337;
let p3 = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, 'foo');
});

Promise.all([p1, p2, p3]).then(values => {
  console.log(values); // Vraća: [3, 1337, "foo"]
});
```

Promise.all(): Primer 2

```
var p1 = new Promise((resolve, reject) => {
  setTimeout(resolve, 1000, 'jedan');
});
var p2 = new Promise((resolve, reject) => {
  setTimeout(resolve, 2000, 'dva');
});
var p3 = new Promise((resolve, reject) => {
  setTimeout(resolve, 3000, 'tri');
});
var p4 = new Promise((resolve, reject) => {
  reject('Ovaj promise je namerno odbijen.');
});
var p5 = new Promise((resolve, reject) => {
  setTimeout(resolve, 4000, 'pet');
});

Promise.all([p1, p2, p3, p4, p5]).then(values => {
  console.log(values);
}).catch(reason => {
  console.log("Nema svih sastojaka.", reason);
});  // Vraća: "Nema svih sastojaka Ovaj promise je namerno odbijen"
```

Asinhronost i sinhronost `Promise.all()`: Primer

```
var prazanPromise = Promise.all([]);
var listaPromise = [Promise.resolve("neki podatak"), Promise.resolve(123)];
var nekiPromise = Promise.all(listaPromise);

// Sinhrono stampanje izlaza

console.log(prazanPromise)
// Vraća: Promise { <state>: "fulfilled", <value>: Array[0] }

console.log(nekiPromise);
// Vraća: Promise { <state>: "pending"}

// Asinhrono stampanje izlaza

setTimeout(function(){
    console.log(prazanPromise);
// Vraća: Promise { <state>: "fulfilled", <value>: Array[2] }
});

setTimeout(function(){
    console.log(nekiPromise);
// Vraća: Promise { <state>: "fulfilled", <value>: Array[2] }
});
```

Metoda `Promise.race()`

- Takođe obezbeđuje ponašanje nad listom `Promise` objekata koje vraća jedan `Promise` iz liste `Promise` objekata.
- Metoda kao argument prima iterabilnu listu `Promise` objekata a vraća `Promise`.
- Ponašanje: Vraća rezultat koji "pristigne prvi", bez obzira da li je uspešan ili ne.
  - To znači da može biti vraćen `Promise` ili sa krajnjim podatkom kod uspešno izvršene asinhrone operacije ili sa razlogom za neuspešnu operaciju.

Promise.race(): Primer 1

```
var p1 = new Promise(function(resolve, reject)
  {
   setTimeout(resolve, 500, `prvi`);
});
var p2 = new Promise(function(resolve, reject)
  {
   setTimeout(resolve, 100, `drugi`);
});

Promise.race([p1, p2]).then(function(value) {
   console.log(value); // `drugi`
   // Oba su rezolvirana, ali je p2 brži
});
```

Promise.race(): Primer 2

```
//-----------------------------------------------
var p3 = new Promise(function(resolve, reject) {
  setTimeout(resolve, 100, "treći");
});
var p4 = new Promise(function(resolve, reject) {
  setTimeout(reject, 500, "četvrti");
});

Promise.race([p3, p4]).then(function(value) {
  console.log(value); // "treći"
  // p3 je brži, pa on rezolvira
}, function(reason) {
  // Ne poziva se
});
```

Promise.race(): Primer 3

```
//-----------------------------------------------
var p5 = new Promise(function(resolve, reject) {
  setTimeout(resolve, 500, "peti");
});
var p6 = new Promise(function(resolve, reject) {
  setTimeout(reject, 100, "šesti");
});

Promise.race([p5, p6]).then(function(value) {
  // Not called
}, function(reason) {
  console.log(reason); // "šesti"
  // p6 je brži, pa on odbacuje
});
```

Kada koristiti `Promise.prototype.then()`

- koristiti kada je bitan redosled izvršavanja:

```
nabaviDrvo()
    .then(() => napraviCamac())
    .then(ploviRekom());
```

Kada koristiti `Promise.all()`

- Koristiti kada redosled izvršavanja nije bitan, već je bitan trenutak kada su svi spremni:

```
var nabaviBrasno = new Promise((resolve, reject) => {
  setTimeout(resolve, 1000, 'jedan');
});
var nabaviMleko= new Promise((resolve, reject) => {
  setTimeout(resolve, 2000, 'dva');
});

Promise.all([nabaviBrasno, nabaviMleko])
    .then(values => {
       console.log('Napravi krofne')
    })
    .catch(reason => {
      console.log(reason)
    });
```

Korišćenje `Promise.race` za detekciju statusa `Promise` 1/2

```
/* Koršćenje metode promise.race za detekciju statusa
Promise objekta */

function promiseState(promise) {
  const pendingState = { status: "pending" };

  return Promise.race([promise, pendingState]).then(
    (value) =>
      value === pendingState ? value : { status:
                "fulfilled", value },
    (reason) => ({ status: "rejected", reason }),
  );
}
```

Korišćenje `Promise.race` za detekciju statusa `Promise` 2/2

```
const p1 = new Promise((res) => setTimeout(() => res(100), 100));
const p2 = new Promise((res) => setTimeout(() => res(200), 200));
const p3 = new Promise((res, rej) => setTimeout(() => rej(300), 100));

async function getStates() {
  console.log(await promiseState(p1));
  console.log(await promiseState(p2));
  console.log(await promiseState(p3));
}

console.log("Immediately after initiation:");
getStates();
setTimeout(() => {
  console.log("After waiting for 100ms:");
  getStates();
}, 100);
```

Asinhroni iteratori i generatori

Iteratori

- U JavaScript-u, **iterator** je objekat koji definiše sekvencu vrednosti i potencijalno povratnu vrednost nakon terminiranja.
- Specifično, iterator je bilo koji objekat koji implementira ***iterativni protokol*** (standardan način za generisanje sekvence vrednosti) tako što ima `next( )` metodu koja vraća objekat sa dva svojstva:
  - `value` Sledeća vrednost u iterativnoj sekvenci.
  - done Vrednost `true` ako je poslednja vrednost u sekvenci konzumirana. Ako se zada `value` uz done, to je povratna vrednost iteratora.
- Važno svojstvo standardnog iteratora je ***sinhronost*** – očekuje se da iterator definiše sekvencu vrednosti "glatko", bez ikakvih prekida i čekanja između iterativnih koraka.

Iteratori: primer iteratora

```
function makeRangeIterator(start = 0, end = Infinity, step = 1) {
    let nextIndex = start;
    let iterationCount = 0;

    const rangeIterator = {
       next: function() {
           let result;
           if (nextIndex < end) {
               result = { value: nextIndex, done: false }
               nextIndex += step;
               iterationCount++;
               return result;
           }
           return { value: iterationCount, done: true }
       }
    };
    return rangeIterator;
}
```

Iteratori: primer korišćenja iteratora

```
const it = makeRangeIterator(1, 10, 2);

let result = it.next();
while (!result.done) {
 console.log(result.value); // 1 3 5 7 9
 result = it.next();
}

console.log("Iterirano nad sekvencom veličine: ",
  result.value); // result.value je 5
// vraćeno je 5 brojeva, iz intervala: 0 do 10
```

Generatori

- Regularne funkcije vraćaju samo jednu vrednost ili ne vraćaju ništa.
- Generatori su posebna vrsta funkcija koja može na zahtev da vrati više vrednosti, jednu za drugom (`yield <value>`) .
- One su prirodno povezane sa iterabilnim tipovima (tipovi koji se mogu procesirati u `for ... of` maniru) za jednostavno kreiranje strimova podataka.

Kreiranje generatora

- Za kreiranje generatora koristi se poseban sintaktički konstrukt zvani ***generatorska funkcija*** koji je sledećeg oblika:

```
function*
```

- Primer:

```
function* generateSequence() { /* vraća
  vrednosti 1, 2 na zahtev (ključna reč yield i
  vrednost 3 “normalnom” return naredbom */
   yield 1;      // <value> je 1
   yield 2;      // <value> je 2
   return 3;
}
```

Ponašanje generatorske funkcije

- Generatorske funkcije se ponašaju drugačije od regularnih funkcija
- Kada se takva funkcija pozove, ona **ne pokreće svoj kod** već **vraća specijalni objekat** koji se zove ***generatorski objekat*** i koji **upravlja izvršavanjem funkcije**.

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}

let generator = generateSequence(); /* poziv kojim
  generatorska funkcija generateSequence() kreira svoj
  generatorski objekat */

console.log(generator); /* generateSequence {<suspended>} */
```

Kako izgleda objekat `Generator`

[[Prototype]]: Generator
- **constructor**: GeneratorFunction {
  prototype: Generator,
  Symbol(Symbol.toStringTag): 'GeneratorFunction',
  constructor: *f*}
- **next**: *f next()* – **ovo je main metoda generatora**
- **return**: *f return()*
- **throw**: *f throw()*
- **Symbol(Symbol.toStringTag)**: "Generator"
- [[Prototype]]: Object

[[GeneratorState]]: "suspended"

[[GeneratorFunction]]: *f* generateSequence()*

[[GeneratorReceiver]]: Window

[[Scopes]]: Scopes[3]

Izvršavanje generatorske funkcije

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}
```

Izvršavanje koda funkcije još nije počelo:

- `main` metoda generatora je `next()`.
  1. Kod se izvršava do najbilže naredbe `yield <value>` (podrazumevano `value` je `undefined`).
  2. Naredba `yield <value>` privremeno zaustavlja izvršavanje i spoljašnjem kodu se vraća `value` .
  3. Sledeći poziv počinje sa izvršavanjem prve naredbe iza izvršene naredbe `yield <value>`
- Rezultat metode `next()` je objekat sa dva svojstva:
  - `value`: vraćena vrednost (`value`).
  - `done`: `true` ako je kod funkcije izvršen do kraja, u protivnom `false`.

Izvršavanje generatorske funkcije: primer

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}

let generator = generateSequence(); // kreiranje objekta

let one = generator.next();
console.log(' prvi next poziv ' + JSON.stringify(one)); // {value: 1,
  done: false}

let two = generator.next();
console.log(' drugi next poziv ' + JSON.stringify(two)); // {value:
  2, done: false}

let three = generator.next();
console.log(' treci next poziv ' + JSON.stringify(three)); // {value:
  3, done: true}

let four = generator.next();
console.log(' cetvrti next poziv ' + JSON.stringify(four)); // {done:
  true}
```

Generatori su iterabilni

- Nad vrednostima generatora može se iterirati koristeći `for..of`:

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}

let generator = generateSequence();

for(let value of generator) {
  console.log(value); // 1, zatim 2 ali nema 3
}
```

Rade i sa **spread** sintaksom

```
function* generateSequence() {
  yield 1;
  yield 2;
  yield 3;
  return 4;
}

let sequence = [0, ...generateSequence()];
console.log(sequence); /* [0, 1, 2, 3] – nema
                                        4 */
```

Kompozicija generatora

- Kompozicija generatora je specijalna mogućnost koja dozvoljava da se generatori transparentno “ugrađuju” jedan u drugi.
  - To je prirodan način za umetanje toka jednog generatora u drugi generator.
  - Kompozcija generatora ne koristi dodatnu memoriju za skladištenje međurezultata.

Primer kompozicije generatora

- Zadatak: data je funkcija koja generiše sekvencu brojeva:

```
function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) yield
 i;
}
```

- Korišćenjem ove funkcije generisati složenije sekvence:
  - prvo, cifre 0..9 (sa kodovima 48…57),
  - Iza cifara velika slova alfabeta A..Z (sa kodovima 65…90)
  - I nakon velikih slova, mala slova alfabeta a..z (sa kodovima 97…122)

Kompozicija generatora: `yield*`

```
function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) yield i;
}
function* generatePasswordCodes() {
  // 0..9
  yield* generateSequence(48, 57);

  // A..Z
  yield* generateSequence(65, 90);

  // a..z
  yield* generateSequence(97, 122);

}
```

- Naredba `yield*` *delegira* izvršenje drugom generatoru.
- Delegiranje ovde znači da `yield*` <gen> iterira nad gen i transparentno prosleđuje svoje rezultate napolje.

Rezultat kompozicije

```
let str = '';

for(let code of generatePasswordCodes()) {
  str += String.fromCharCode(code);
}

console.log(str);
 /*0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabc
 defghijklmnopqrstuvwxyz */
```

`yield` je dvosmerna ulica

- Pored toga što vraća rezultat, `yield` može i da prosledi vrednost u generator.
- To se radi tako što se pozove `generator.next(arg)` sa argumentom:

```
function* gen() {
  let ask1 = yield "2 + 2 = ?";
  alert(ask1); // 4
  let ask2 = yield "3 * 3 = ?"
  alert(ask2); // 9
}
let generator = gen();
alert( generator.next().value ); // "2 + 2 = ?"
alert( generator.next(4).value ); // "3 * 3 = ?"
alert( generator.next(9).done ); // true
```

Rukovanje greškama

- Spoljašnji kod može da prosledi vrednost u generator kao rezultat od `yield`.
- Ali može i da inicira grešku – i greška je vrsta rezultata.
- Da bi se prosledila greška u `yield`, treba pozvati `generator.throw(err)`.
  - U takvom slučaju greška se signalizira zajedno sa tim `yield`.

generator.throw(err): Primer 1/2

```
function* gen() {
  try {
    let result = yield "2 + 2 = ?"; //(1)
    alert("Izvršenje ne dolazi dovde, gore je bila greška");
  } catch(e) {
    alert(e); // prikazuje grešku
  }
}

let generator = gen();
let question = generator.next().value;
generator.throw(new Error("Nema odgovora u mojoj bazi")); //(2)
```

`generator.throw(err)`: Primer 2/2

- Tekuća linija u kodu koji poziva je linija sa `generator.throw`. Dakle, možemo je tu i **uhvatiti** na primer ovako:

```
function* generate() {
  let result = yield "2 + 2 = ?"; /* Greška u ovoj liniji */
}

let generator = generate(); // kreira objekat

let question = generator.next().value; // pokreće generator

try {
  generator.throw(new Error("Nema odgovora u mojoj bazi"));
} catch(e) {
  console.log(e); // prikazuje grešku
}
```

Asinhroni iteratori$_{1/3}$

- Asinhroni iteratori omogućuju iteriranje nad podacima koji dolaze asinhrono, na zahtev.
  - Na primer, preuzimanje nečega komad-po-komad putem mreže.
- Trenutno u JS-u jedini ugrađeni asinhroni iterator je `AsyncGenerator` objekat koji vraća asinhrone generatorske funkcije.
- Ugrađeni asinhroni iteratori su i asinhrono iterabilni pa se mogu koristiti u `for await...of` petlji kao u primeru na sledećem slajdu.

Asinhroni iteratori 2/3

```
const asyncIterator = (async function* () {
  yield 1;
  yield 2;
  yield 3;
})();
(async () => {
  for await (const value of asyncIterator) {
    console.log(value);
  }
})();
// Logovi: 1, 2, 3
```

Asinhroni iteratori 3/3

- Od regularnih iteratora razlikuju se u sledećem:

1. Da bi se objekat učinio asinhrono iterabilnim, mora da ima metodu `Symbol.asyncIterator`.
2. Ta metoda mora da vrati objekat sa `next()` metodom koja vraća `Promise` .
3. Metoda `next()` ne mora da bude `async`, može da bude regularna metoda koja vraća tip `Promise`, ali `async` dozvoljava da se koristi `await`, što je zgodno.
4. Za iteriranje koristimo `for await (let value of range)`, naime dodajemo“await” nakon “for”. To poziva jednom `range[Symbol.asyncIterator]()`, a zatim njegov `next()` za vrednosti.

`Symbol` tip

- Symbol je primitivan tip.
- Vrednost tipa `Symbol` se zove symbol vrednost.
- U JavaScript runtime okruženju, symbol vrednost se kreira pozivanjem funkcije `Symbol`, koja dinamički pravi anonimnu jedinstvenu vrednost.
- `Symbol` može da se koristi kao svojstvo objekta.

Symbol jedinstvenost

```
// Ovde su dva simbola sa istim opisom:
let Sym1 = Symbol("Sym");
let Sym2 = Symbol("Sym");


console.log(Sym1 === Sym2) /* vraća
 "false"; Symbol-i su garantovano
 jedinstveni. Čak i kada se kreira više
 simbola sa istim opisom, oni se smatraju
 različitim vrednostima. */
```

Dobro-poznati `symbol`-i

- Klasa `Symbol` ima konstante za tzv. *dobro-poznate simbole*.
- Ti simboli omogućuju da se konfiguriše način kako JS tretira objekat koristeći ih kao svojstva.
- Primeri tih simbola su: Symbol.iterator za nizolike objekte, ili Symbol.search za string objekte.
- Spisak je dat u tabeli Well-known symbols :
  - `Symbol.hasInstance`
  - `Symbol.isConcatSpreadable`
  - `Symbol.iterator`
  - `Symbol.toPrimitive`
  - ...itd.

Globalni registar simbola

- Postoji globalni registar simbola koji sadrži sve raspoložive simbole.
- Interfejs za pristup registru su metode Symbol.for() i Symbol.keyFor(); one vrše medijaciju između globalne tabele simbola (ili "registry“-a) i izvršnog okruženja.
- Metod Symbol.for(*tokenString*) vraća vrednost simbola iz registra, a Symbol.keyFor(*symbolValue*) vraća token string iz registra; međusobno su inverzni:

```
Symbol.keyFor(Symbol.for("tokenString")) ===
  "tokenString" // true
```

Regularni i asinhroni iteratori: poređenje$_{1/2}$

- JavaScript podržava generatore i oni su iterabilni.
- U regularnim generatorima ne može se koristiti `await`. Sve vrednosti moraju da dolaze sinhrono – nema mesta za odlaganja u `for..of`, to je sinhroni konstrukt.
- Ali se lako može napraviti asinhroni generator, samo se ispred imena doda ključna reč `async`, kao u primeru koji sledi

Regularni i asinhroni iteratori: poređenje$_{2/2}$

	Regularni iteratori	Asinhroni iteratori
Objektna metoda koja obezbeđuje iterator	`Symbol.iterator`	`Symbol.asyncIterator`
`next()` povratna vrednost je	bilo koji tip	`Promise`
za petlju koristiti	`for..of`	`for await..of`

Funkcionalnosti koje zahtevaju regularne, sinhrone iteratore ne rade sa asinhronim iteratorima. Na primer `spread` sintaksa neće raditi:

```
alert( [...range] ); // Error, no Symbol.iterator
```

Ovo je očekivano zato što `spread` sintaksa očekuje da nađe `Symbol.iterator`, kao i `for..of` bez `await` a ne `Symbol.asyncIterator`.

Asinhroni generatori: primer

```
async function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) {
    // super, može se koristiti  await!
    await new Promise(resolve =>
  setTimeout(resolve, 1000));
    yield i;
  }
}

(async () => {
  let generator = generateSequence(1, 5);
  for await (let value of generator) {
    console.log(value); /* 1, pa 2, pa 3, pa 4, pa
                           5 (na sekundu) */
  }
})();
```

Asinhroni generatori: korišćenje

- Sada imamo asinhroni generator iterabilan sa `await...of`.
- Dodavanje ključne reči `async` omogućuje generatoru da može koristiti `await` unutar sebe, oslonac na tip `promise` i druge asinhrone funkcije.
- Još jedna razlika je da je `generator.next()` metoda asinhronog generatora takođe asinhrona pa vraća tip `promise`.
- Za dobijanje vrednosti od asinhronog generatora treba koristiti:

```
result = await generator.next();
// result = {value: ..., done: true/false}
```

Primeri generatora i njihovog korišćenja

Primer kreiranja Generatora

```
function* gen() {
return 'first generator';
}
let generatorResult = gen()
console.log(' Generator pre instanciranja', generatorResult)
let instanca = generatorResult.next()
console.log (' Instanca generatora:', instanca)
console.log (' Vrednost u instanci:', instanca.value)
```

Generator: makeRangeIterator

```
function* makeRangeIterator(start = 0, end = 100, step
  = 1) {
    let iterationCount = 0;
    for (let i = start; i < end; i += step) {
        iterationCount++;
        yield i;
    }
    return iterationCount;
}
let rangeResult = makeRangeIterator()
let instanca = rangeResult.next()
console.log (' Instanca generatora:', instanca)
console.log (' Vrednost u instanci:', instanca.value)
let instanca1 = rangeResult.next()
console.log (' Vrednost u instanci:', instanca1.value)
```

Sažetak

- Regularni iteratori i generator su dobri za podatke koji ne zahtevaju puno vremena za generisanje.
- Ako podaci stižu asinhrono, sa kašnjenjima koriste se asinhrone varijante iteratora i generataora i kosntrukt `for await..of` umesto `for..of`.
- Ta je situacija vrlo česta na Web-u gde podaci dolaze asinhrono – preuzimanje ili postavljanje velikih fajlova.
- Za obradu takvih podataka su posebno zgodni asinhroni generatori.
- Valja pomenuti da neka okruženja (n.pr. Brauzeri) imaju specijalne interfejse (API zvani Streams) za rad sa strimovima gde je moguća i razmena podataka među strimovima).

Literatura za predavanje

1. Z. Konjović, Funkcionalno programiranje, **Tema 09 – JS Asinhrono programiranje**, slajdovi sa predavanja, dostupni na folderu **FP kurs 2023-24→Files →Slajdovi sa predavanja**
2. M. Haverbeke, *Eloquent JavaScript A Modern Introduction to Programming*, No Starch Press, Inc., 2019, Chapter 11: Asynchronous Programming