Promise

Prethodna lekcija završena je pričom o callback funkcijama. Reč je o funkcijama koje se već veoma dugo vremena koriste kako bi se određena logika izvršila tek nakon neke druge, veoma često vremenski zahtevne operacije. Tako su callback funkcije inkorporirane u veliki broj funkcionalnosti koje web pregledači izlažu na korišćenje JavaScript kodu koji mi samostalno pišemo. U prethodnoj lekciji mogli ste videti da se osnovni način obrade događaja zasniva na upotrebi callback funkcija. Ista je situacija i sa tajming funkcijama – `setInterval()` i `setTimeout()`. Tako se može reći da su callback funkcije osnovni mehanizam za uticanje na linearnost izvršavanja JavaScript programskog koda, koji koriste brojne ugrađene funkcionalnosti web pregledača.

Sa pojavom ES6 specifikacije, JavaScript jezik je obogaćen još jednom funkcionalnošću koja olakšava asinhrono programiranje i omogućava da se logika koja je dosad bila realizovana callback funkcijama dodatno uprosti. Reč je o pojmu koji se naziva Promise.

Promisei se koriste od strane brojnih modernih aplikativnih programskih interfejsa web pregledača. Stoga je njihovo razumevanje od presudne važnosti za ozbiljno bavljenje poslom frontend programera. U najvećem broju slučajeva, rad sa Promiseima podrazumeva korišćenje Promisea koji se dobijaju od raznih Web API-ja. Ipak, frontend programer ima mogućnost kreiranja i sopstvenih Promisea. Stoga će u nastavku ove lekcije biti prikazani različiti pristupi za korišćenje Promise objekata koji se dobijaju od web pregledača, ali i za njihovo samostalno kreiranje.

Problem callback funkcija

Kako biste na pravi način mogli da razumete prednosti koje Promisei donose, na početku ove lekcije biće prikazan osnovni nedostatak callback funkcija. Za početak, evo primera pretplate na `load` događaj, koji podrazumeva korišćenje callback funkcije:

```
window.addEventListener('load', function(){

    console.log("Web page has been loaded!");

});
```

Primer ilustruje pretplatu na load događaj koji se aktivira kada se kompletna stranica učita, uključujući i sve resurse, odnosno dodatne fajlove stilizacije, slike i slično. Kada se tako nešto obavi, aktivira se callback funkcija, koja je u obliku anonimne funkcije prosleđena `addEventListener()` metodi kao drugi argument (parametar).

Sada ćemo ovakvom primeru dodati još logike:

```
window.addEventListener('load', function() {

    document.getElementById("my-
button").addEventListener('click', function(){

    });

});
```

Logika je sada proširena, pa je tako unutar callback funkcije smeštena još jedna naredba u kojoj se definiše još jedna callback funkcija. Ona će se aktivirati kada se klikne na `my-button` element, koji se nalazi na stranici. Već sada možete naslutiti šta se ovim primerom nastoji prikazati – pristup koji podrazumeva korišćenje callback funkcija veoma često stvara duboka gnežđenja, odnosno smeštanje jednog bloka koda unutar drugog, što na kraju u značajnoj meri komplikuje kod i otežava održavanje.

```
            window.addEventListener('load', function() {

                document.getElementById("my-
button").addEventListener('click', function(){

                    setInterval(function() {

                        myHeading.style.color = myHeading.style.color ==
'black' ? 'blue' : 'red';

                    }, 500);

                });

            });
```

Primeru je sada dodat i treći nivo gnežđenja callback funkcija, uvođenjem još jednog bloka kojim se upotrebom funkcije `setInterval()` boja teksta naslova na svakih pola sekunde menja između dve boje – crne i crvene. Na ovaj način, stigli smo samo do tri nivoa gnežđenja, ali zamislite koliko bi snalaženje u kodu bilo otežano u slučaju pet ili više nivoa gnežđenja. Poseban problem nastaje i prilikom obrade eventualnih grešaka koje mogu da nastanu prilikom izvršavanja asinhronih operacija. U takvom slučaju, komplikovana situacija dodatno se komplikuje. Kako bi se prevazišli ovakvi problemi, u JavaScript je uveden pojam Promise.

Šta je Promise?

Promise je objekat kojim se predstavlja završetak neke asinhrone operacije. Drugim rečima, Promise je objekat koji se emituje od strane asinhronih operacija, a koji se zatim može koristiti za definisanje zasebne logike, koja će se aktivirati kada se takva asinhrona operacija završi uspešno ili neuspešno.

Promisei menjaju način na koji se rukuje asinhronim operacijama. Uvođenjem pojma Promise, omogućeno je da se asinhronim operacijama rukuju kao da su sinhrone. Drugim rečima, one odmah emituju povratnu vrednost u vidu Promise objekta, koji se dalje može koristiti za definisanje logike koja će se aktivirati kada se asinhrona operacija završi.

Prvi primer upotrebe Promisea neće biti realan, već demonstrativan i uprošćen, kako bismo se mogli posvetiti suštini. On će ilustrovati pozivanje jedne funkcije koja svoju logiku obavlja asinhrono. Prvo pogledajte kako može da izgleda pozivanje takve funkcije kada se za dojavu o završetku operacije koriste callback funkcije:

```
      function successCallback(result) {
            console.log("Async operation successfully finished. Result is:" +
result);
      }

      function failureCallback(error) {
            console.error("Async operation failed. Error is: " + error);
      }

      doSomethingAsync(successCallback, failureCallback);
```

U primeru su prvo definisane dve funkcije – `successCallback()` i `failureCallback()`. Reč je zapravo o dve callback funkcije, koje se prosleđuju `doSomethingAsync()` funkciji. One će se aktivirati kada se vremenski zahtevna operacija izvrši uspešno, odnosno neuspešno.

Ovo je bio primer tradicionalnog pristupa, koji je podrazumevao korišćenje callback funkcija. Ukoliko se unutrašnja logika `doSomethingAsync()` funkcije promeni, tako da ona umesto callback funkcija koristi Promise, njena upotreba bi izgledala ovako:

```
let promise = doSomethingAsync();
promise.then(successCallback, failureCallback);
```

Iz prikazanih naredbi može se videti da funkcija `doSomethingAsync()` više ne prihvata callback funkcije, već se ona sada poziva bez parametara. Ipak, za razliku od prethodnog primera, ona sada emituje jednu povratnu vrednost. Reč je o Promise objektu, čija se referenca u primeru smešta unutar promenljive `promise`. Ovakva situacija se može interpretirati na sledeći način – pozivanjem neke vremenski zahtevne operacije, od nje se dobija **obećanje** (*engl. promise*) da će nam se ona javiti kada završi svoju logiku.

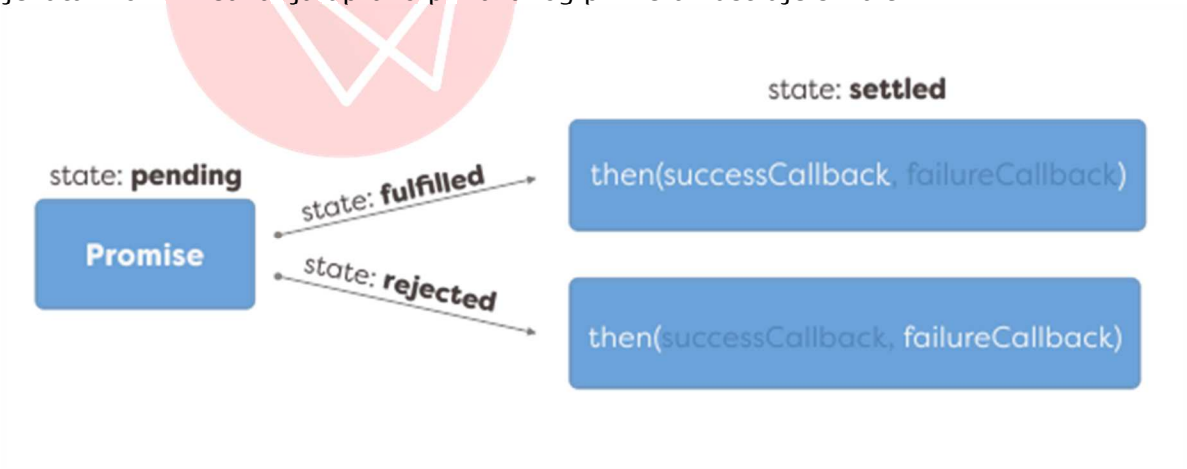
Obećanje (Promise) koje se dobija od vremenski zahtevne operacije u primeru se koristi za definisanje logike koja će se aktivirati kada se takva operacija završi. Drugim rečima, nad Promise objektom poziva se metoda **then()** i njoj se prosleđuje logika koja će se aktivirati u slučaju uspešnog ili neuspešnog izvršavanja, respektivno.

Prethodni primer namerno je razdvojen u dve naredbe – prvu, u kojoj se poziva vremenski zahtevna operacija i preuzima Promise objekat, i drugu – u kojoj se definišu `successCallback()` i `failureCallback()` funkcije. To je učinjeno kako biste na bolji način mogli da razumete šta se događa prilikom poziva vremenski zahtevne operacije koja emituje Promise. Ipak, u praksi se najčešće sve ovo obavlja korišćenjem jedne naredbe:

```
doSomethingAsync().then(successCallback, failureCallback);
```

Stanja Promise objekta

Na osnovu upravo prikazanog primera, može se govoriti i o prvoj značajnoj osobini Promise objekata. Tok izvršavanja upravo prikazanog primera ilustruje slika 9.1.



Slika 9.1. Stanja Promise objekta

Na slici 9.1. može se videti i tok logike unutar Promise objekta u upravo prikazanom primeru. Odrednicama *state* na slici su obeležena stanja kroz koja prolazi Promise objekat tokom izvršavanja njegove operacije. Takva osnovna stanja Promise objekta mogu biti:

- **pending** – inicijalno stanje; stanje neispunjenog Promisea;
- **fulfilled** – stanje koje označava da je operacija uspešno obavljena;
- **rejected** – stanje koje označava neuspešno obavljanje operacije;
- **settled** – stanje koje označava da je operacija završena, bilo uspešno ili neuspešno.

Uspešnim izvršavanjem operacije, stanje Promisea prelazi u *fulfilled* i tada se poziva funkcija `successCallback()`. Neuspešnim izvršavanjem operacije, Promise prelazi u stanje *rejected* i tada se aktivira funkcija `failureCallback()`.

Prelazak u stanje *fulfilled* ili *rejected* drugačije se naziva ispunjenje Promisea (ispunjenje obećanja).

U ovom trenutku mogu se navesti i prve osobine Promise objekata koje ih izdvajaju od pristupa koji podrazumeva korišćenje callback funkcija:

- Promise se mora ispuniti, odnosno on mora preći ili u stanje *fulfilled* ili u stanje *rejected*
- jedna od funkcija koje se definišu za uspešno, odnosno neuspešno izvršavanje logike Promisea mora se aktivirati; pri tome, to će uvek biti samo jedna funkcija, odnosno u prikazanom primeru ili funkcija `successCallback()` ili funkcija `failureCallback()`
- nakon završetka logike Promisea, odgovarajuća funkcija se aktivira samo jednom, nikako više puta;
- u slučaju da je logika Promisea završena pre definisanja funkcija koje će se aktivirati u slučaju uspešnog, odnosno neuspešnog izvršavanja, jedna od njih će se i tada svakako aktivirati, što nije slučaj kod asinhronih funkcija koje kao parametre prihvataju callback funkcije.

Pitanje

Promise se ne mora ispuniti onda kada se operacija koju predstavlja završi neuspešno.

- Tačno.
- **Netačno.**

Objašnjenje:

Prelazak u stanje fulfilled ili rejected drugačije se naziva ispunjenje Promisea. Promise se mora ispuniti, odnosno on mora preći ili u stanje fulfilled ili u stanje rejected.

Prvo samostalno kreiranje Promisea

Nakon prvog demonstrativnog primera, čiji je cilj bio razumevanje osnovne svrhe Promisea, biće prikazan način na koji se Promisei samostalno kreiraju. Samostalno kreiranje Promisea omogućiće nam da upoznamo još neke njihove značajne osobine.

Osnovna sintaksa za kreiranje Promisea izgleda ovako:

```
var promise = new Promise(function(resolve, reject) {

    //do something; in most cases long running async operation

    if(/*operation completed successfully*/) {
        resolve("Operation completed successfully.");
    } else {
        reject(Error("Something is not right!"));
    }

});
```

Blok koda ilustruje osnovnu sintaksu za kreiranje Promise objekta. Kreiranje objekta započinje kao i kod bilo kog drugog objekta. Ipak, konstruktorskoj funkciji se kao parametar prosleđuje jedna anonimna funkcija sa dva parametra – `resolve` i `reject`. Ova dva parametra će JavaScript samostalno popuniti odgovarajućim referencama. Već možete da naslutite da je reč o referencama na metode koja će se aktivirati kada se logika Promisea završi uspešno, odnosno neuspešno, respektivno.

Napomena

Anonimna funkcija koja se prosleđuje Promiseu prilikom kreiranja drugačije se naziva egzekutor (*engl. executor*), zato što se unutar nje definiše konkretna logika Promisea. Dalje, dva parametra koje prihvata egzekutor ne moraju biti imenovana sa `resolve` i `reject`, već je moguće upotrebiti proizvoljne nazive.

Logika Promisea, što je najčešće neka vremenski zahtevna operacija, postavlja se direktno unutar tela funkcije egzekutor. U primeru je umesto takve operacije postavljen odgovarajući komentar - *do something; in most cases long running async operation.*

Nakon završetka vremenski zahtevne operacije, unutar Promisea neophodno je pozvati jednu od funkcija na koje upućuju parametri funkcije egzekutor:

- **resolve()** – za dojavu o uspešno završenoj logici;
- **reject()** – za dojavu o grešci prilikom izvršavanja logike Promisea.

Opet je bitno napomenuti da korišćenje naziva `resolve` i `reject` nije obavezno.

Sve ovo na primeru prvog upotrebljivog koda jednog Promisea može da izgleda ovako:

```
var promise = new Promise(function(resolve, reject) {

    //logic start
    let result = Math.random() * 10;
    //logic end

    resolve(result);
});
```

Iako je primer i dalje isključivo u domenu demonstracije, sada je reč o upotrebljivom kodu koji će stvoriti jedan Promise objekat. Za njegovu logiku je definisana jedna naredba u kojoj se vrši generisanje nasumičnog broja korišćenjem metode `random()`, `Math` objekta. Nakon završetka takve, vrlo jednostavne logike, iz Promisea se signalizira njen završetak pozivanjem metode `resolve()`.

Ovako kreiran Promise moguće je upotrebiti na sledeći način:

```
            promise.then(function(result) {
                console.log("Operation completed successfully. Result is: " +
result);
            }, function(error){
                console.log("Promise failed. " + error);
            });
```

Nad objektom Promisea poziva se metoda `then()`, kojoj se prosleđuju dve callback funkcije. Prva će se aktivirati kada se unutar Promisea pozove metoda `resolve()`. S obzirom na to da je upravo to metoda koja se unutar Promisea i poziva u prikazanom primeru, unutar konzole se dobija sledeći rezultat:

```
Operation completed successfully. Result is: 6.328985353316801
```

Na osnovu teksta koji se dobija unutar konzole može se zaključiti sledeće:

- poziva se prva callback funkcija koja se prosleđuje metodi `then()`, zato što je logika Promisea uspešno završena;
- vrednost koja se unutar Promisea prosleđuje metodi `resolve()` koristi se kao ulazni parametar callback funkcije koja se aktivira za uspešno završenu logiku.

Ukoliko se iz nekog razloga logika Promisea ne izvrši uspešno, unutar Promisea je neophodno pozvati metodu `reject()`:

```
            var promise = new Promise(function(resolve, reject) {

                //logic start
                let result = Math.random() * 10;
                //logic end

                reject("Unknown error.");

            });
```

Sada se unutar Promisea obavlja pozivanje metode `reject()`. Na taj način se signalizira da je logika Promisea neuspešno obavljena. Metodi `reject()` je moguće proslediti bilo koji parametar, baš kao u primeru. Ipak, dobra praksa jeste emitovanje `Error` objekta, i to na sledeći način:

```
var promise = new Promise(function(resolve, reject) {

    //logic start
    let result = Math.random() * 10;
    //logic end

    reject(new Error("Unknown error."));

});
```

Sada se metodi `reject()`, prosleđuje objekat `Error` koji se anonimno kreira.

Kada se logika Promisea neuspešno završi i obavi pozivanje `reject()` metode, dolazi do aktiviranja druge callback funkcije koja se prosleđuje metodi `then()`. Stoga se u ovakvoj situaciji unutar konzole dobija:

```
Promise failed. Error: Unknown error.
```

Asinhrona priroda Promisea

Nakon prvog primera samostalnog kreiranja Promisea, bitno je razumeti još jednu njihovu osobinu. Callback metode koje se prosleđuju metodi `then()` uvek se izvršavaju asinhrono, odnosno tek kada se u potpunosti isprazni stek poziva:

```
var promise = new Promise(function(resolve, reject) {

    //logic start
    let result = Math.random() * 10;
    //logic end

    resolve(result);

});

promise.then(function(result) {
    console.log("Operation completed successfully. Result is: " +
result);
}, function(error){
    console.log("Promise failed. " + error);
});

console.log("Hello to everyone.");
```

Sada, nakon kreiranja i obrade Promisea, definisana je još jedna naredba kojom se upisuje neki tekst unutar konzole. Unutar konzole se dobija:

```
Hello to everyone.
Operation completed successfully. Result is: 2.6107150160214387
```

Na osnovu ispisa u konzoli može se videti da se logika koja se nalazi nakon obrade Promise objekta izvršava prva. Tek nakon nje, u potpunosti se oslobađa stek poziva, pa se aktivira i odgovarajuća callback funkcija.

Primer – kreiranje funkcije delay() upotrebom Promisea

Nakon prvog primera samostalnog kreiranja Promisea, biće prikazan još jedan, ovoga puta mnogo realniji i upotrebljiviji. Naime, već je rečeno da Promisee najviše koriste novije, ugrađene funkcionalnosti web pregledača. Pored toga, tendencija je da se i unutar većine starijih funkcionalnosti callback funkcije menjaju Promiseima. Upravo jedan takav primer mi ćemo u nastavku samostalno realizovati. Naime, iz prethodne lekcije poznata vam je ugrađena funkcija za odloženo izvršavanje logike:

```
setTimeout(function, delay, arg);
```

Ovo je sintaksa metode `setTimeout()`. Ona kao parametar prihvata callback funkciju koja se aktivira nakon vremena definisanog drugim parametrom. Pored toga, `setTimeout()` može da prihvati i treći parametar, koji se odnosi na eventualni parametar koji će biti prosleđen callback funkciji.

U nastavku ovoga primera, ovakva `setTimeout()` metoda nama će poslužiti kao osnova za kreiranje logike koja će se zasnivati na korišćenju Promisea:

```
var promise = new Promise(function(resolve, reject) {
    setTimeout(resolve, 2000);
});
```

Unutar Promisea sada se po prvi put obavlja jedna vremenski zahtevna operacija. Metodi `setTimeout()` je na mestu prvog parametra, koji smo dosad popunjavali referencom na callback funkciju, prosleđena referenca `resolve` argument. Tako će na ovaj način, unutar Promisea, nakon dve sekunde biti pozvana metoda na koju upućuje `resolve` parametar, čime će se dojaviti uspešan završetak logike Promisea.

Sve dosad nije izrečena jedna veoma važna osobina Promise objekata – **logika koja se definiše unutar Promisea započinje izvršavanje odmah nakon kreiranja objekta Promisea**. Drugim rečima, u našem primeru otkucavanje timeouta započinje onoga trenutka kada se izvrši upravo prikazana naredba kreiranja Promise objekta. S obzirom na to da u većini slučajeva to nije ponašanje koje želimo, pribegava se smeštanju Promisea unutar zasebnih funkcija:

```
function delay(time) {
    return new Promise(function (resolve, reject) {
        setTimeout(resolve, time);
    });
}
```

Kreiranje Promise objekta sada je prebačeno unutar zasebne funkcije `delay()`. Sada na praktičnom primeru možete videti ono što je rečeno na početku lekcije – da su Promisei objekti koji se kao povratne vrednosti emituju od strane funkcija koje logiku obavljaju asinhrono. Upravo to smo dobili u našem primeru.

Smeštanjem Promisea unutar zasebne funkcije dobili smo još nešto – mogućnost da samostalno definišemo vreme zadrške. Dosad je parametar koji se prosleđivao metodi `setTimeout()` bio *zakucan* na vrednost `2000`, dok je sada uvedena mogućnost da se takva vrednost definiše kao parametar koji će se proslediti funkciji `delay()`.

Sada se ovako kreirana funkcija `delay()` može uposliti na sledeći način:

```
delay(4000).then(function(){
    console.log("4 seconds passed...");
});
```

Nad povratnom vrednošću metode `delay()` poziva se metoda `then()` i njoj se prosleđuje anonimna funkcija koja se aktivira kada Promise završi svoju logiku. Stoga, nakon četiri sekunde, unutar konzole dobija se poruka:

```
4 seconds passed...
```

Bitno je da primetite da se ovoga puta metodi `then()` prosleđuje samo jedan parametar, odnosno samo jedna callback funkcija. Drugim rečima, u ovom primeru nismo zainteresovani za dojavu o neuspehu izvršavanja logike unutar Promisea. Zato smo obavili prosleđivanje samo jedne callback funkcije.

Kod kompletnog primera:

```
function delay(time) {
    return new Promise(function (resolve, reject) {
        setTimeout(resolve, time);
    });
}

delay(4000).then(function () {
    console.log("4 seconds passed...");
});
```

Metode za konzumiranje Promisea

Promise objekti se mogu doživeti kao spona između koda koji predstavlja neku operaciju i zainteresovanih konzumenata za rezultat takve operacije. Dosad je prikazana osnovna metoda koja se koristi za konzumiranje rezultata proizvodnog koda. Reč je o metodi `then()`. Ipak, pored ove metode, Promise objekti poseduju još neke metode koje je moguće koristiti za konzumiranje rezultata Promisea. Tako postoje ukupno tri različite metode koje je moguće koristiti za obavljanje takvog posla:

- **then()**
- **catch()**
- **finally()**

then()

`then()` je osnovna metoda koja se koristi za konzumiranje rezultata proizvodnog koda koji se nalazi unutar Promisea. Osnovna sintaksa ove metode izgleda ovako:

```
promise.then(
    function (result) { /* handle a successful result */ },
    function (error) { /* handle an error */ }
);
```

Metoda `then()` može da prihvati dva parametra. Oba se odnose na callback funkcije koje se aktiviraju kada se proizvodni kod završi uspešno, odnosno neuspešno, respektivno. Drugim rečima, prva metoda se aktivira kada se logika Promisea završi uspešno, a druga metoda se aktivira kada se takva logika završi neuspešno.

Primeri korišćenja `then()` metode prikazani su već dosta puta u dosadašnjem toku izlaganja o Promiseima:

```
let promise = new Promise(function (resolve, reject) {
    setTimeout(() => resolve(4), 3000);
})


promise.then(
    function (result) { console.log(result) },
    function (error) { console.log(error) }
);
```

U primeru je prvo kreiran jedan Promise objekat, a zatim je korišćenjem metode `then()` obavljeno definisanje dve callback funkcije koje će konzumirati rezultat Promisea. Unutar Promisea je simulirana jedna vremenski zahtevna operacija, korišćenjem metode `setTimeout()`. Tako će u primeru Promise nakon tri sekunde da emituje rezultat svim pretplaćenim konzumentima. S obzirom na to da će se proizvodni kod završiti uspešno, biće aktivirana prva callback funkcija, pa će se unutar konzole dobiti:

```
4
```

Ukoliko bi se proizvodni kod završio neuspešno, bila bi pozvana druga callback funkcija:

```
let promise = new Promise(function (resolve, reject) {
    setTimeout(() => reject(new Error("Unknown error.")), 3000);
})


promise.then(
    function (result) { console.log(result) },
    function (error) { console.log(error) }
);
```

Sada se unutar Promisea poziva metoda `reject()`, čime se svim zainteresovanim konzumentima dojavljuje neuspešan završetak operacije. Stoga se aktivira druga callback funkcija i unutar konzole dobija:

```
Error: Unknown error.
```

Metodi `then()` nije neophodno proslediti dva parametra. Ukoliko smo zainteresovani samo za dojavu o uspešnom završetku logike unutar Promisea, moguće je proslediti samo jednu callback funkciju:

```
let promise = new Promise(function (resolve, reject) {
    setTimeout(() => resolve(4), 3000);
})

promise.then(
    function (result) { console.log(result) }
);
```

Na sličan način, ukoliko smo zainteresovani samo za dobijanje dojave o neuspešnom završetku logike unutar Promisea, metodi `then()` je moguće proslediti samo drugu callback funkciju. Na mesto prve callback funkcije u takvoj situaciji postavlja se vrednost `null`:

```
let promise = new Promise(function (resolve, reject) {
    setTimeout(() => reject(new Error("Unknown error.")), 3000);
})

promise.then(
    null, function (error) { console.log(error) }
);
```

catch()

Kada smo zainteresovani samo za dobijanje dojave o neuspehu proizvodnog koda, baš kao u prethodnom primeru, moguće je koristiti i jednu posebnu metodu za konzumiranje rezultata Promise. Reč je o metodi `catch()`:

```
let promise = new Promise(function (resolve, reject) {
    setTimeout(() => reject(new Error("Unknown error.")), 3000);
})

promise.catch(function (error) { console.log(error) });
```

Metoda `then()` je sada zamenjena metodom `catch()`. Metoda `catch()` prihvata samo jedan parametar, s obzirom na to da se koristi samo ukoliko želimo da dobijamo dojavu o neuspehu proizvodnog koda.

Metoda `catch(f)` samo je skraćeni oblik za pozivanje metode sa prvim parametrom čija je vrednost `null` - `then(null, f)`.

finally()

Promise objekti poseduju još jednu metodu koju je moguće koristiti za dobijanje dojave o završetku proizvodnog koda, bez obzira na ishod izvršavanja. Reč je o metodi `finally()`. Ova metoda se poziva onoga trenutka kada Promise dobije stanje *settled.* Reč je o stanju koje označava da je operacija završena, bilo uspešno ili neuspešno.

Metoda `finally()` prihvata samo jedan parametar koji se odnosi na callback funkciju koja će se aktivirati kada se logika Promise objekta završi. Ipak, bitno je znati da, za razliku od `then()` i `catch()` metoda, callback funkciji `finally()` metode ne prosleđuje se nijedan parametar. Drugim rečima, unutar `finally()` metode ne može se znati da li se Promise završio uspešno ili ne. To nije toliko ni važno, s obzirom na to da se `finally()` metoda uglavnom koristi kako bi se obavila neka generalizovana operacija, koja nije zavisna od uspešnosti ili neuspešnosti Promise operacije.

```
            let promise = new Promise(function (resolve, reject) {
                setTimeout(() => resolve("Loaded!"), 3000);
            })

            promise.finally(function () {
                console.log("The operation was completed. We do not know if
it is successful or not.");
            });
```

Bitno je da primetite ono što je rečeno nešto ranije. Callback funkcija `finally()` metode poziva se bez ikakvih parametara. Unutar konzole se dobija poruka:

```
The operation was completed. We do not know if it is successful or not.
```

Identična poruka će se dobiti i ukoliko se unutar Promisea pozove `reject()` metoda:

```
            let promise = new Promise(function (resolve, reject) {
                setTimeout(() => reject(new Error("Unknown error")), 3000);
            })

            promise.finally(function () {
                console.log("The operation was completed. We do not know if
it is successful or not.");
            });
```

Primer – prikaz loadera tokom trajanja logike Promisea

Kako biste još bolje mogli da razumete namenu `finally()` metode, biće prikazan još jedan realni primer koji će podrazumevati upotrebu takve metode. Naime, primer će podrazumevati prikaz loadera sve dok traje logika Promisea:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
```

```
        <meta name="viewport" content="width=device-width, initial-
scale=1.0">
        <title>Promise finally example</title>
        <style>
            #result-box {
                text-align: center;
                line-height: 180px;
                font-size: 33px;
                font-family: sans-serif;
                font-weight: bold;
            }
            #loader,
            #loader:before,
            #loader:after {
                background: #5485c4;
                -webkit-animation: load1 1s infinite ease-in-out;
                animation: load1 1s infinite ease-in-out;
                width: 1em;
                height: 4em;
            }
            #loader {
                color: #5485c4;
                text-indent: -9999em;
                margin: 88px auto;
                position: relative;
                font-size: 11px;
                -webkit-transform: translateZ(0);
                -ms-transform: translateZ(0);
                transform: translateZ(0);
                -webkit-animation-delay: -0.16s;
                animation-delay: -0.16s;
            }
            #loader:before,
            #loader:after {
                position: absolute;
                top: 0;
                content: '';
            }
            #loader:before {
                left: -1.5em;
                -webkit-animation-delay: -0.32s;
                animation-delay: -0.32s;
            }
            #loader:after {
                left: 1.5em;
            }
            @-webkit-keyframes load1 {
                0%,
                80%,
                100% {
                    box-shadow: 0 0;
                    height: 4em;
                }
                40% {
                    box-shadow: 0 -2em;
                    height: 5em;
                }
```

```
            }
            @keyframes load1 {
                0%,
                80%,
                100% {
                    box-shadow: 0 0;
                    height: 4em;
                }
                40% {
                    box-shadow: 0 -2em;
                    height: 5em;
                }
            }
        </style>
    </head>
    <body>
        <div id="loader"></div>
        <div id="result-box"></div>
        <script>
            let promise = new Promise(function (resolve, reject) {
                setTimeout(() => resolve("Loaded!"), 3000);
            })
            promise.finally(function () {
                let loader = document.getElementById("loader");
                loader.style.display = "none";
            }).then(function (result) {
                let resultBox = document.getElementById("result-box");
                resultBox.innerHTML = result;
            });
        </script>
    </body>
</html>
```

Animacija 9.1. Primer skrivanja loadera upotrebom Promisea

Upravo prikazani primer, pored praktičnog korišćenja `finally()`metode, po prvi put ilustruje i jedan poseban pristup – nadovezivanje Promise objekata. Naime, u primeru možete videti da je nad povratnom vrednošću `finally()`metode obavljeno pozivanje metode `then()`. Kako je ovako nešto moguće biće objašnjeno u narednoj lekciji.

Rezime

- ECMAScript 2015 (ES6) specifikacija uvodi pojam Promisea;
- Promise je objekat koji predstavlja obećanje koje asinhrone operacije emituju kao svoju povratnu vrednost;
- Promise omogućava da se asinhronom operacijom rukuje kao da je reč o sinhronoj;
- tokom svog životnog toka, Promise može proći kroz nekoliko stanja, koja su u vezi sa izvršavanjem logike unutar Promisea: *pending*, *fulfilled*, *rejected* i *settled*
- Promise se kreira korišćenjem `Promise()` konstruktorske funkcije, kojoj se prosleđuje anonimna funkcija sa dva parametra;
- anonimna funkcija koja se prosleđuje Promiseu prilikom kreiranja drugačije se naziva egzekutor;
- logika Promisea definiše se unutar funkcije egzekutor;
- u slučaju uspešnog završetka logike, iz funkcije egzekutor se poziva funkcija čiju referencu čuva prvi prosleđeni parametar;
- u slučaju neuspešnog završetka logike, iz funkcije egzekutor se poziva funkcija čiju referencu čuva drugi prosleđeni parametar;
- nad Promise objektom je moguće definisati šta će se dogoditi kada se asinhrona operacija završi – to se naziva konzumiranje Promisea;
- Promise je moguće konzumirati korišćenjem metoda `then()`, `catch()` i `finally()`