Fakultet elektrotehnike, računarstva
i informacijskih tehnologija Osijek

Web programiranje

LV2: ECMASCRIPT 2015 - ES6

2019-2020 | ver 1.0 | Autor: Luka Omrčen

Sadržaj

1 Uvod

JavaScript je dinamičan skriptni programski jezik visoke razine. JavaScript (JS) je interpretativnog tipa što znači da se JS kod direktno izvršava (JIT- Just in time) bez pretvorbe u strojni jezik (binarni). JS omogućuju interpretaciju klijentske strane logike unutar samog web preglednika. Prvobitno kreiran za web preglednike, JS se unutar zadnjih par godina popularizacijom okvira Node.js koristi i na serverskoj strani (backend), kao i u izradi desktop/mobilnih aplikacija (Electron, Phaser.js).

Osim što je definiran objektno orijentiranom paradigmom, JavaSCript podržava i funkcionalno programiranje. U JavaScriptu sve je predstavljeno kao objekt. Nema postoji razlika između vrsta objekata. Ono što razlikuje JavaScript od drugih objektno orijentiranih programa (pr. Java) je to što nasljeđivanje funkcionira putem mehanizma prototipa, a svojstva i metode mogu se dinamički dodati bilo kojem objektu, dok u Javi objekti su podijeljeni u klase i instance, a nasljeđivanje se provodi putem hijerarhije klasa. Klase i instance ne mogu dinamički dodavati svojstva ili metode. Kako se JavaScript nalazi unutar web preglednika i na klijentskoj strani - nije moguće vršiti I/O radnje s klijentovog diska.

JavaScript je standardiziran prema Ecma International - Europskoj udruzi za standardizaciju informacijskih i komunikacijskih sustava (ECMA je ranije bio skraćenica za Europsko udruženje proizvođača računala) radi isporuke standardiziranog međunarodnog programskog jezika koji se temelji na JavaScriptu. Ova standardizirana inačica JavaScripta, nazvana ECMAScript, ponaša se na isti način u svim aplikacijama koje podržavaju navedeni standard. Najnovija verzija je ECMAScript 2019, ali većina web preglednika i njihovih starijih verzija nemaju implementirane funkcionalnosti iz najnovijeg ECMAScript standarda. ECMAScript 2016 ili skraćeno ES6 riješio je probleme iz ranijih verzija ECMAScript-a, definirao modernu sintaksu te se bolje prilagodio i uveo brojne značajke za razvoj velikih aplikacija.

2 Ecmascript 2015 - ES6

2.1 Deklariranje varijabli i razine okvira (scoping)

Za deklariranje varijabli unutar JavaScripta potrebno je dodati ključnu riječ **var**. U kodu 1 varijabla X deklarirana je u glavnom okviru te predstavlja globalnu varijablu. Definiran je novi blok vitičastim zagradama koji ujedno predstavlja i novi okvir unutar glavnog okvira. Deklariranjem varijable s ključnom riječi **Var** definiramo joj funkcijski okvir - što znači da varijabla koja je deklarirana u većem okviru vidljiva je manjim okvirima u kojemu se oni nalaze. To predstavlja potencijalni problem prilikom definiranja prava pristupa unutar koda.

```
var x = 3;
{
  x = 6;
}
console.log(x); // 6
```

Kod 1: Primjer deklariranje varijabli s ključnom riječi var

Primjer koda 2 prikazuje funkciju koja je definirana funkcijskom deklaracijom i varijablu foo koja predstavlja funkcijski izraz. Varijabla boo pohranjuje u svoj adresni prostor anonimnu funkciju koja

vraća vrijednost "no more boo". Kako se radi o deklaraciji istog imena funkcije, funkcijski izraz koji je napisan poslije funkcijske deklaracije "preotet" će funkcionalnost funkcije boo.

```
function boo() {
  return "boo";
}

var boo = function() {
  return "no more boo";
};

console.log(boo()); // no more boo
```

Kod 2: Primjer deklariranje varijabli s ključnom riječi let

U primjeru koda 4 definirana je funkcija main koja predstavlja globalnu funkciju. Funkcija main sastoji se od lokalne varijable X koju sadrži u sebi i metode local koja se poziva unutar funkcije main. Nakon što funkcija main bude pozvana izvana, automatski će se pozvati i funkcija local. U navedenom primjeru vidljivo je kako funkcija local mijenja vrijednost varijable X. Kao i u gornjem primjeru, varijabla X se nalazi u višem funkcijskom okviru koji je vidljiv okviru metode local u kojemu se ona nalazi.

```
function main() {
  var x = 15;
  local();

  function local() {
    for (x = 0; x < 100; x++) {}
  }
  console.log(x); // 100
}

main();
```

Kod 3: Primjer deklariranje varijabli s ključnom riječi let

U tu svrhu ES6 definira novu **let** deklaraciju koja predstavlja blok okvir. Primjer 4 rješava problem funkcijskog okvira **var** deklaracije.

```
// Primjer 1
let x = 3;
{
  let x = 6;
  console.log(x) // 6
}
console.log(x); // 3
// ------------------------------------------------------
// Primjer 2
function main() {
  let x = 15;
```

```
  local();

  function local() {
    for (let x = 0; x < 100; x++) {}
  }
  console.log(x); // 15
}
main();
```

Kod 4: Primjer deklariranje varijabli unutar funkcije s ključnom riječi var

Iz gornjeg primjera moguće je vidjeti kako varijable nisu promijenile vrijednosti. Svaka varijabla definirana ključnom riječi let unutar svog okvira zadržava vrijednost koja je specifična za taj blok okvir, i nije vidljiva izvan njega.

Kako bi spriječili moguće greške iz primjera koda 5 ES6 osim **let** definira deklaraciju **const** koja deklarira varijablu kao konstantu.

```
const boo = function() {
  return "boo";
};

function boo() {
  return "no more boo";
}

console.log(boo()); // SyntaxError: redeclaration of const boo note: Previously declared at
    line 1, column 6
```

Kod 5: Primjer deklariranje varijabli s ključnom riječi const

Iznimka postoji kod JavaScript objekata. Prilikom korištenja deklaracije const objekt kao tip podatka smatrat će se konstantom, ali vrijednosti objekta (svojstva) su izložene promjenama (Kod 6).

```
const objekt = { broj: 3 };
objekt = 4;
console.log(obj); // Type Error

// Ali ...
objekt.broj = 12;
console.log(objekt.broj); // 12
```

Kod 6: Primjer deklariranje objekta s ključnom riječi const

Ukoliko promjene nad svojstvima objekta nisu poželjne, potrebno je koristiti ugrađene metode Object.freeze() i Object.seal() za blokiranje promjene strukture objekta (Kod 7).

```
const objekt = { broj: 3 };
Object.freeze(objekt);
Object.seal(objekt);
```

```
objekt.broj = 12;
objekt.novoSvojstvo = "Test";

console.log(objekt.broj); // 3  -- metoda freeze
console.log(objekt.novoSvojstvo); // undefined -- metoda seal
```

Kod 7: Primjer deklariranje objekta s ključnom riječi const

2.2 Definiranje vrijednosti unutar teksta

Varijable ili vrijednosti definiraju se unutar $ znaka. Navodnici (‘ ‘) koriste se za ograđivanje tekstualnog niza neovisno o prelamanju u novi red. Osim prikaza vrijednosti, važno je znati da JavaScript vrši implicitnu pretvorbu podatka. U navedenom slučaju prvo vrši numeričku operaciju, a zatim provjerava dali se navedena vrijednost može pretvoriti u znakovni tip podatka koji se onda pridružuje ostatku teksta.

```
let name = "Laki";
let text = ‘
  Hi ${name} how are you, how much is ${3 * 5} ?
  Whos that ?
‘;

console.log(text); //  Hi Laki how are you, how much is 15 ? Who’s that ?
```

Kod 8: Definiranje vrijednosti unutar teksta

2.3 Deklaracija objekta

Objekt je instanca koja sadrži skup parova ključnih vrijednosti. Za razliku od primitivnih vrsta podataka (znakovni tip, numerički), objekti mogu predstavljati višestruke ili složene vrijednosti i mogu se mijenjati tijekom svog životnog vijeka. Vrijednosti mogu biti primitivne vrijednosti, funkcije ili niz drugih objekata.

```
const object = {
  name: "foo",
  num: 5,
  sayHello() {
    return ‘Hello world from‘;
  },
  // ako je definirani parametar prazan, postavi vrijednost na "me"
  extendHelloWorld(input = "me") {
    return ‘${this.sayHello()} from ${this.name} and ${input}‘;
  }
};
console.log(object.num); // 5
console.log(object.extendHelloWorld()); // Hello world from foo and me
```

Kod 9: Deklaracija objekta

Kako bi referencirali svojstva ili metode unutar samog objekta potrebno je koristiti ključnu riječ **this**.

2.4 ES6 Klase

Klasa u smislu OOP-a predstavlja nacrt za stvaranje objekata. Klasa objedinjuje podatke koje će biti definirane unutar objektu. Definicija klase mora sadržavati sljedeće:

- **Konstruktor** - odgovoran za raspoređivanje memorije objekata klase.
- **Metode** - predstavljaju radnje koje objekt može izvršiti. Funkcije klase drugi su naziv za metode.

```
class Animal {
  constructor(type = "animal") {
    this._type = type;
  }
  get animalType() {
    return this._type;
  }
  set animalType(val) {
    this._type = val.toUpperCase();
  }
  makeSound() {
    return "Default animal sound";
  }
}
// Kreiranje objekta
let test = new Animal("Dog");
// Pozivanje gettera
console.log(test.animalType); // Dog
// Pozivanje settera
test.animalType = "Cat"; // Drugi nacin -> test["animalType"] = "Cat"
// Ponovno pozivanje gettera
console.log(test.animalType); // CAT
```

Kod 10: Definiranje JS klase

Osim definiranja konstruktora, unutar koda 10 prikazano je definiranje gettera i settera. U JavaScriptu, svako svojstvo unutar klase je public tipa. Kako bi označili svojstvo kao privatan tip podatka (zapravo ono nije) - definira se crtom ispred riječi svojstva. Unutar getter funkcije vraćamo privatno svojstvo, dok setter funkcija mijenja vrijednost iz vanjskog atributa i postavlja vrijednost velikim slovima.

ES6 podržava koncept nasljeđivanja. Nasljeđivanje je sposobnost kreiranje novih nacrta klase iz postojećih klasa. Klasa koja se proširuje radi stvaranja novih klasa naziva se roditeljska ili super klasa. Novostvorena klasa naziva se dijete ili podklasa.

Nasljeđivanje se definira pomoću ključne riječi '**extends**'. Potklasa nasljeđuje sva svojstva i metode osim konstruktora iz roditeljske klase. Unutar potklase potrebno je definirati novi konstruktor i pomoću

ključne riječi **super** pozvati svojstva konstruktora iz super klase (primjer kod 11). Ako iz klase Cat maknemo metodu makeSound(), metoda će biti pozvana iz roditeljske klase i ona će vratiti vrijednost "Default animal sound".

```
class Cat extends Animal {
  constructor() {
    /**
    * Definiranjem super kao da smo kopirali konstruktor roditelja
    * u novu klasu s predanim parametrom type
    *  constructor(type = "animal") {
      this._type = type;
      }
    */
    super("cat");
  }

  makeSound() {
    return "Meoow";
  }
}

let cat = new Cat();
console.log(cat.animalType); // cat
console.log(cat.makeSound()); // Meoow
```

Kod 11: Koncept nasljeđivanja unutar ES6 Klasa

ES6 klase podržavaju definiranje statičkih metoda unutar klase bez prisustva konstruktora. Kako bi pristupili statičkoj metodi potrebno je koristiti ime klase zajedno s operatorom pridruživanja (primjer kod 13

```
class Utilities {
  static increase(x) {
    return x + 1;
  }
  static decrease(x) {
    return x - 1;
  }
}

let y = Utilities.increase(2);
console.log(y); // 3
y = Utilities.decrease(2);
console.log(y); // 1
```

Kod 12: Definiranje statičkih metoda klase

2.5 ES6 "Arrow" Funkcije

"Arrow" funkcije sintaktički je kompaktna alternativa regularnom izrazu definiranja funkcije. "Arrow" prikladni su za definiranje metoda i ne mogu se koristiti kao konstruktori.

```
let classic = function(a, b) {
  return a + b;
};

let classic2 = function(a) {
  return a;
};
// Ekvivalent gornjem primjeru koda
let arrow = (a, b) => {
  return a + b;
};

let arrow2 = a => {
  return a;
};
// Ukoliko funkcija ima samo return moze se pisati skraceni zapis
let arrowShorthand = (a, b) => a + b;
let arrowShorthand2 = a => a;
```

Kod 13: Definiranje funkcijskih izraza "Arrow" metodom

2.6 Metode za rad s poljima

2.6.1 Array.map

Metoda map stvara novo polje s rezultatima koje se dobiju pozivanjem određene funkcije na svaki element u predanom polju. Unutar koda 14 definiramo polje array s određenim vrijednostima.

Polju newArray pridružujemo polje array s definiranom funkcijom map. Funkcija map kao parametar ima anonimnu callback funkciju (callback funkcija - metoda map čeka da se anonimna funkcija izvrši i vrati rezultat). Anonimna funkcija ima parametar item koji predstavlja člana novog polja. Vrijednost člana polja prije nego što se pohrani u novo polje bit će pomnožena s dva. Map će se iterirati onoliko puta koliko ima predanih članova polja.

```
let array = [1, 2, 3, 4, 5];
let newArray = array.map(item => {
  return item * 2;
});
console.log(newArray); // Array(5) [ 2, 4, 6, 8, 10 ]

// Stara sintaksa
var array = [1, 2, 3, 4, 5];
var newArray = [];
```

```
for (var i = 0; i < array.length; i++) {
  newArray.push(array[i] * 2);
}
```

Kod 14: Primjer Array.map metode

2.6.2 Array.filter

Za razliku od metode map, metoda filter stvara novo polje sa svim elementima koji prolaze definirani uvijet unutar anonimne funkcije. Princip iteracije je isti kao kod metode map.

```
let array = [1, 2, 3, 4, 5];
let newArray = array.filter(item => {
  return item % 2 === 0;
});
console.log(newArray); // Array [ 2, 4 ]
//Stara sintaksa
var array = [1, 2, 3, 4, 5];
var newArray = [];

for (var i = 0; i < array.length; i++) {
  if (array[i] % 2 === 0) {
    newArray.push(array[i]);
  }
}
```

Kod 15: Primjer Array.filter metode

2.6.3 Array.reduce

Metoda reduce definira funkciju koja vrši manipulaciju nad članovima polja pomoću dva parametra (prethodna vrijednost, trenutna vrijednost) kako bi na kraju definirala krajnju (reduciranu) vrijednost cijelog polja. U navedenom primjeru - iteracijom kroz polje svi članovi polja su zbrojeni u jednu vrijednost.

```
let array = [1, 2, 3, 4, 5];
let sum = array.reduce((previousValue, value) => {
  return previousValue + value;
});
console.log(sum); // 15
//Stara sintaksa
var array = [1, 2, 3, 4, 5];
var sum = 0;
for (var i = 0; i < array.length; i++) {
  sum = array[i] + sum;
}
```

Kod 16: Primjer Array.reduce metode

2.6.4 Ostale Array metode

- **Array.from() metoda** - metoda koja kreira polje od objekta koji ima definiranu kljuc:vrijednost strukturu - za posljednji parametar potrebno je definirati duzinu polja.

```
let object = {
  0: "zero",
  1: "one",
  2: "two",
  3: "three",
  length: 5
};

let array = Array.from(object);
console.log(array); //Array(5) [ "zero", "one", "two", "three", undefined ]
```

Kod 17: Primjer Array.from metode

- **Array.find() metoda** - metoda koja pretražuje zadano polje. Kada dođe do ispunjena uvijeta vraća vrijednost i zaustavlja daljnju iteraciju polja

```
let array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
let value = array.find(x => {
  return x > 5;
});
console.log(value); // 6
```

Kod 18: Primjer Array.find metode

- **Array.findIndex() metoda** - slična metoda kao i find. Metoda vraća mjesto unutar polja vrijednosti koja ispunjava uvijet.

```
let array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
let value = array.findIndex(x => {
  return x > 5;
});
console.log(value); // 5
```

Kod 19: Primjer Array.findIndex metode

Ostale metode s poljima moguće je istražiti na poveznici [1].

2.7 Metode za rad sa znakovnim tipom podatka

ES 6 uveo je nove metode za rad sa tekstom. Više metoda moguće je istražiti na poveznici [5].

```
console.log("Hello".startsWith("Hell")); // true
console.log("Goodbye".endsWith("bye")); // true
console.log("Jar".repeat(2)); //JarJar
console.log("abcdef".includes("bcd")); // true
```

Kod 20: Primjer korištenje metoda za rad sa znakovnim tipom podatka

2.8 Metode za rad sa numeričkim tipom podataka

ES 6 uveo je nove metode i konstante za rad sa numeričkim tipom podatka. Više metoda moguće je istražiti na poveznici [6].

```
console.log(Number.EPSILON); // 2.220446049250313e-16
console.log(Number.MAX_SAFE_INTEGER); // 9007199254740991
console.log(Number.MIN_SAFE_INTEGER); // -9007199254740991
console.log(Number.isNaN(5)); // false
console.log(Number.isFinite(1234)); // true
console.log(Number.isInteger(5)); // true
console.log(Number.isSafeInteger(6123.5)); // true
console.log(Number.parseFloat("32.12345")); // 32.12345
console.log(Number.parseInt(12.5)); // 12
```

Kod 21: Primjer korištenja metoda za rad sa numeričkim tipom podataka

2.9 Iteratori

2.9.1 For .. in petlja

Petlja for ... in se koristi za kretanje kroz svojstva objekta. U svakoj iteraciji jedno svojstvo objekta dodijeljeno je imenu varijable i ta petlja se nastavlja sve dok obuhvati sva svojstva objekta (Kod 22).

```
let obj = { a: 1, b: 2, c: 3 };
for (let prop in obj) {
  console.log(obj[prop]); // Iterira 1,2,3
}
// Nije preporucena za rad s poljima
let polje = ["a,b,c,d"];
for (let prop in polje) {
  console.log(polje[prop]); Vraca string a,b,c,d
}
```

Kod 22: Primjer korištenja for in petlje

2.9.2 For .. of petlja

For of petlju koristi se kada je potrebno raditi iteraciju kroz definirano polje. Ukoliko se vrši iteracija nad objektima parser vraća grešku da se radi o pogrešnom tipu podatka (kod 23)

```
let array = [1, 2, 3,{1:"test"}];
for (let item of array) {
  console.log(item); // Iterira vrijednosti 1,2,3,Object { 1: "test" }
}
// Ako pokusamo koristiti for of na objektu
let object = { a: 1, b: 2, c: 3 };
for (let item of object) {
  console.log(object[item]); // TypeError: object is not iterable
}
```

Kod 23: Primjer korištenja for of petlje

2.9.3 "Spread" operator

"Spread" operator omogućava proširivanje iterabilnog tipa na mjestima na kojimu kojima se očekuje jedan ili više argumenata (za pozive funkcija), elementima (za polje) ili vrijednosti objekta gdje se očekuje jedan ili više parova ključ-vrijednost.

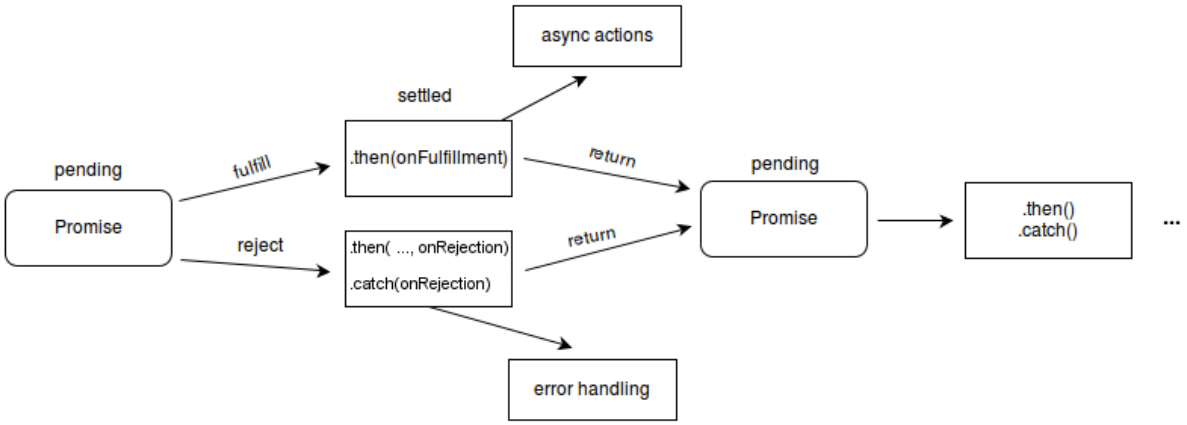
```
let array1 = ["from", "me"];
let array2 = ["hello", "world", ...array1];
console.log(array2); //Array(4) [ "hello", "world", "from", "me" ]

function counter(first, second, ...[a,b,c]) {
  console.log(...[a,b,c]) // 4 5 6
  return first + second + a + b + c;
}
console.log(counter(2, 3, 4, 5 ,6)); // 20

let foo = "hello world";
console.log(...foo);  // h e l l o   w o r l d
console.log([...foo]) // Array(11) [ "h", "e", "l", "l", "o", " ", "w", "o", "r", "l", ...]
```

Kod 24: Primjer korištenja spread operatora

2.10 ES6 Promise objekti

Promise predstavlja objekt koji definira završetak (ili neuspjeh) asinkrone operacije i dobivene vrijednosti. Promise predstavlja "semafor" za vrijednost koja trenutno nije poznata kada se promise inicijalizira. To omogućava asinkronoj metodi da vrati vrijednosti poput sinkronih metoda - umjesto da odmah vrati konačnu vrijednost, asinkrona metoda vraća "obećanje" da će vrijednost vratiti u nekom trenutku u budućnosti.

Promise se sastoji od tri stanja:



Slika 1: Tijek izvođenja promise objekta

- **Čekanje** - Predstavlja početno stanje. Nije ispunjeno niti je odbijeno
- **Ispunjeno** - Predstavlja operaciju koja je uspješno završila.
- **Odbijeno** - Operacija nije uspješno završila

Tijek izvođenja promise objekta prikazan je na slici 1. Metode **Promise.prototype.then()** i **Promise.prototype.catch()** vraćaju promise, te se mogu ulančavati.

Primjer 25 prikazuje inicijalizaciju promise objekta i hvatanje "obećane" vrijednosti. Promise prima 2 parametra, jedan za uspješnu obradu zathjeva i jedan za neuspjeh. Promise se inicijalizira i definira setTimeout asinkronu funkciju koja pauzira svoje izvršavanje na jednu sekundu. Funkcija resolve() vraća "obećanu" vrijednost i pokazuje da je asinkrona metoda uspješno vratila vrijednost, odnosno "obećanje". S metodom .then() obrađujemo Promise objekt i prikazujemo dobivenu vrijednost iz Promise objekta.

```
let mojPrviPromise = new Promise((resolve, reject) => {
  // Pozivamo resolve() kada asinkrona metoda vrati vrijednost uspjesno, a reject() kad nije
    uspjelo.
  // U ovom primjeru koristimo setTimeout (...) za simulaciju asinkronog koda.
  console.log("promise postavljen")
  setTimeout(function () {
    resolve("Uspjesno obraden zahtjev!");
  }, 1000);
});

mojPrviPromise.then((successMessage) => {
console.log("resolvamo promise")
// atribut unutar then metode je proizvoljan i predstavlja vrijednost koja se nalazi unutar
// resolve metode iz navedenog promise objekta
console.log(successMessage);
});
```

Kod 25: Primjer korištenja promise objekta

U sljedećem primjeru koristimo nativni Fetch API u kombinaciji s promise-ima kako bismo prikazali popis koji sadrži podatke s krajnje točke API-ja.

Fetch API pruža sučelje za dohvaćanje resursa (uključujući mrežu). Za razliku od XMLHttpRequest), fetch API pruža više mogućnosti i fleksibilniji pristup u radu s dohvaćanjem eksternih resursa.

Fetch API prilikom postavljanja HTTP zahtjeva na resurs vraća Promise objekt - drugim riječima, Fetch trenutno čeka odgovor s api strane. [Kod 26].

```
const apiData = fetch("https://jsonplaceholder.typicode.com/posts");
console.log(apiData); // Promise { <state>: "pending" }
```

Kod 26: Primjer dohvaćanja eksternog resursa Fetch api-em

Definirani promise je u fazi čekanja i može prijeći u ispunjeno stanje ako api vrati tražene vrijednosti. Ukoliko api ne vrati tražene vrijednosti, promise prelazi u odbačeno stanje. Jednom kada se promise ispuni, odnosno pređe u ispunjeno stanje, ne može više mijenjati više svoje stanje.

Pomoću metode **Promise.prototype.then** definira se callback funkcija koja se izvršava nakon što promise bude ispunjen.

```
const apiData = fetch("https://jsonplaceholder.typicode.com/posts");
console.log(apiData); //Promise { <state>: "pending" }

apiData.then((response) => {
  console.log(response);
  //Response { type: "cors", url: "https://jsonplaceholder.typicode.com/posts", redirected:
    false, status: 200, ok: true, statusText: "OK", headers: Headers, body: ReadableStream,
    bodyUsed: false }
  //test.html:13:19
})
```

Kod 27: Dohvaćanje vrijednosti vraćene promise objektom

Vrijednosti vraćene od strane promise-a nalaze se u stream obliku podatka. Kako bi pročitali response vrijednosti, pomoću metode json() potrebno je pretvoriti stream u JSON format podatka. Navedena metoda je također asinkrona - što znači da vraća Promise objekt. U tu svrhu, potrebno je ulančati Promise metode [Kod 28].

```
const apiData = fetch("https://jsonplaceholder.typicode.com/posts");
console.log(apiData); //Promise { <state>: "pending" }

apiData
  .then(response => {
    return response.json();
  })
  .then(jsonData => {
    console.log(jsonData);
```

```
    // Array(100) [ {...}, {...}, {...}]
    // 0: Object { userId: 1, id: 1, title: "sunt aut facere repellat provident occaecati
    excepturi optio reprehenderit" }
    // 1: Object { userId: 1, id: 2, title: "qui est esse" }
  });
```

Kod 28: Dohvaćanje vrijednosti vraćene promise objektom

Navedena vrijednost nalazi se unutar JSON polja objekata. Pomoću metoda za rad s poljima moguće je manipulirati s dobivenim podatcima [kod 29].

```
const apiData = fetch("https://jsonplaceholder.typicode.com/posts");
console.log(apiData); //Promise { <state>: "pending" }

apiData
  .then(response => {
    return response.json();
  })
  .then(jsonData => {
    // fetch data and create new array with title start with S
    let titles = jsonData
      .map(item => {
        return item.title;
      })
      .filter(item => {
        return item.startsWith("s");
      })
    console.log(titles);
    // 0: "sunt aut facere repellat provident occaecati excepturi optio reprehenderit"
    // 1: "sint suscipit perspiciatis velit dolorum rerum ipsa laboriosam odio"
   // 2: "soluta aliquam aperiam consequatur illo quis voluptas"
})
```

Kod 29: Manipuliranje s vraćenom promise vrijednosti

Više o promise objektima i njegovim metodama nalazi se na poveznici [4].

2.11 Uvoz/izvoz ES Modula

ES6 ima ugrađenu podršku za definiranje modula. Svaka JavaScript datoteka može predstavljati određeni modul. Pomoću modula definirane objekte,funkcije,klase ili varijable moguće je učiniti dostupnima u određenim dijelovima datoteka programskog koda, čineći programsku strukturu organiziranom i lakšom za održavati.

Navedeni primjeri prikazuju uvoz i izvoz varijable, funkcija, klasa te cijelog modula.

```
// Unutar HTML-a <script type="module" src="index.js"></script>

// index.js
import { helloWorld } from "./module/helloworld.js";
```

```
console.log(helloWorld);

// module/helloworld.js
let helloWorld = "HELLO WORLD !!!";
export { helloWorld };
```

Kod 30: Uvoz i izvoz varijable iz es6 modula

```
// Unutar HTML-a <script type="module" src="index.js"></script>

// module/math.js
function cube(x) {
  return x * x * x;
}
const foo = Math.PI + Math.SQRT2;
let graph = {
  options: {
    color:'white',
    thickness:'2px'
  },
  draw: function() {
    console.log('From graph draw function');
  }
}
export { cube, foo, graph }

// index.js
import { cube, foo, graph } from './module/math.js';
graph.options = {
  color:'blue',
  thickness:'3px'
};
graph.draw();
console.log(cube(3)); // 27
console.log(foo);    // 4.555806215962888
```

Kod 31: Primjer uvoza više elemenata iz es6 modula

U prijašnjim primjerima eksplicitno je navedeno koji podatci su uvezeni ili izvezeni. U primjeru koda 32 uvezene su sve komponente unutar modula bez pojedinačnog navođenja.

```
import * from '/module/mymodule.js';
```

Kod 32: Primjer uvoza cijelog modula bez

Više o uvozu i izvozu modula nalazi se na poveznicama [3, 2].

Literatura

[1] *MDN - Arrays*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array.

[2] *MDN - export*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export.

[3] *MDN - import*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import.

[4] *MDN - Promise*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise.

[5] *MDN - Strings*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String.

[6] *MDN - Strings*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number.