

# Razvoj programske podpore za web

- predavanja -  
2020./2021.

---

## 7. JavaScript

3/3

# Jednodretvenost i sinkronost u JavaScriptu

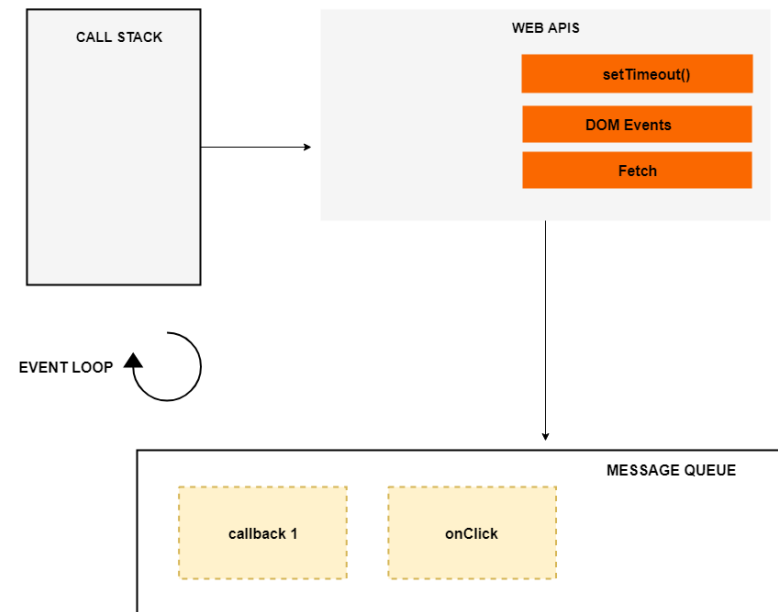
- JavaScript je osmišljen kao **jednodretveni** jezik
  - U *principu* ne postoji mogućnost da jedan proces otvara više dretvi (eng. thread)
- U *principu*, naredbe u JavaScriptu se izvršavaju jedna po jedna, **sinkrono**
  - Nakon što završi jedna naredba, prelazi se na drugu itd.
- Navedeno može izazvati problem ako dretva počne izvršavati naredbu koja dugo traje, čime dretva postaje blokirana, i s izvršavanjem programa se čeka

```
function longOperation(){  
    // Here a very long operation takes  
}  
// Some code - before  
longOperation();  
// Some code - after
```

Ovaj kôd se neće izvršiti sve dok funkcija longOperation() ne završi

# Asinkronost u JavaScriptu (1)

- Iako je sinkron, JavaScript omogućava nekoliko **asinkronih načina izvedbe kôda** kojima se može riješiti problem operacija koje dugo traju:
  - Brojač (eng. timer) – metoda `setTimeout`
  - Događaji DOM-a
  - Dohvat resursa - metoda `fetch`
  - Obećanja (eng. promises)
- Sistemski stog (eng. call stack) je dio JavaScriptovog stroja (eng. engine)
  - web APIs i red poruka (eng. message queue) **nisu!**
  - web APIs i red poruka pripadaju radnoj okolini internetskog preglednika ili radnoj okolini Node.js

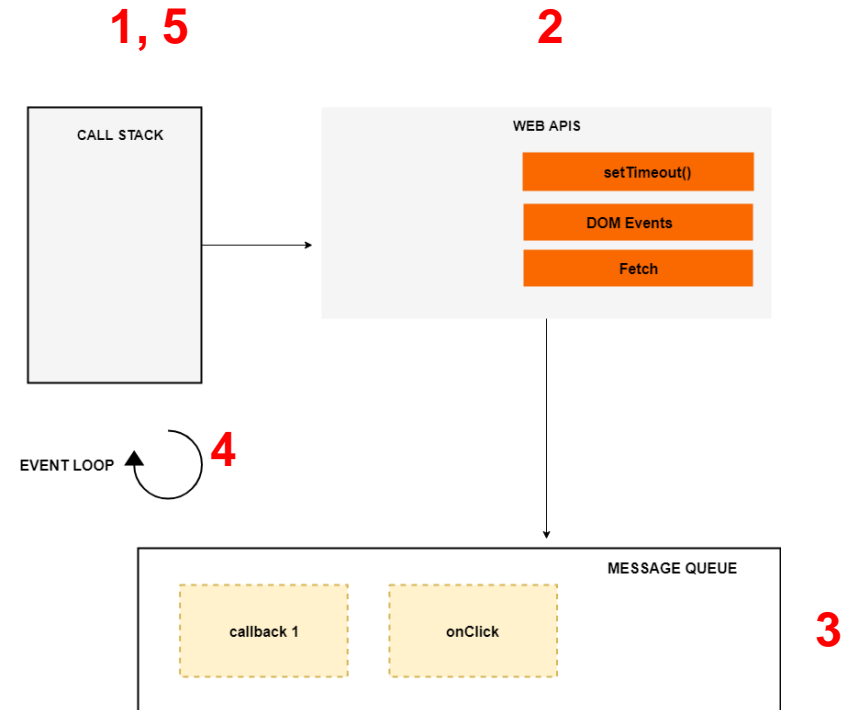


<https://blog.bitsrc.io/understanding-asynchronous-javascript-the-event-loop-74cd408419ff>

# Asinkronost u JavaScriptu (2)

- Prilikom izvedbe asinkronog kôda slijedi se ovaj tijek:

1. Naredba koju treba izvršiti se stavlja na **sistemiški stog** i pokušava izvršiti sinkrono. Ako je naredba asinkrona preusmjerava se na web APIs kako bi započelo njeno asinkrono izvršavanje
2. **Web APIs** preuzima asinkronu naredbu i osigurava da se naredba izvrši (što može trajati)
3. Nakon što je naredba izvršena, povratna funkcija (eng. callback) se stavlja u **red čekanja poruka** (eng. message queue)
4. **Petlja događaja** (eng. event loop) kontinuirano provjerava red čekanja poruka, i ako u njemu naiđe na povratnu funkciju, stavlja ju na stog
5. Započinje sinkrono izvršavanje povratne funkcije koju je petlja događaja postavila na **sistemiški stog**

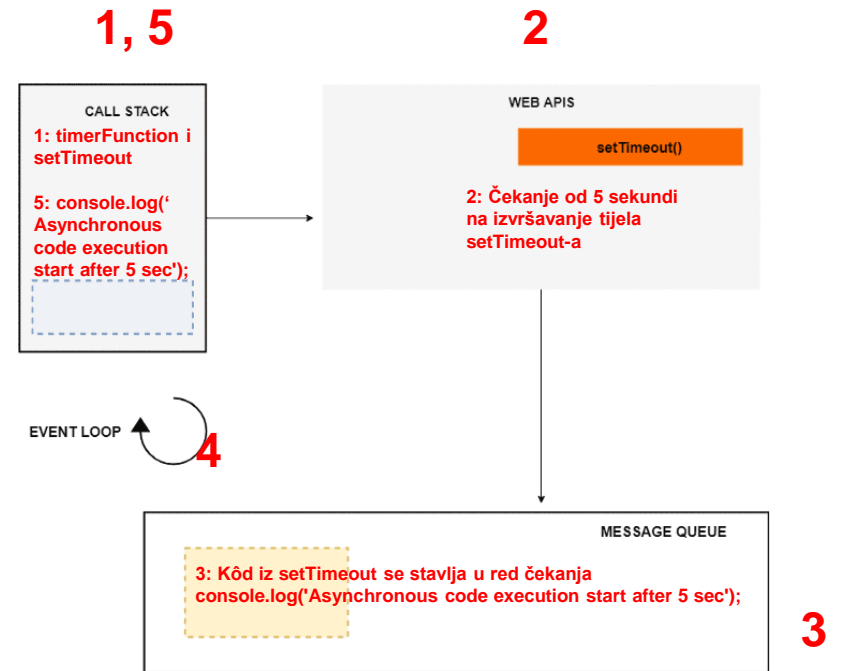


<https://blog.bitsrc.io/understanding-asynchronous-javascript-the-event-loop-74cd408419ff>

# Primjer: Brojač (eng. timer)

- Prilikom izvedbe asinkronog kôda slijedi se ovaj tijek:
  1. `timerFunction` i `setTimeout` se stavljaju na sistemski stog.  
`timerFunction` i `setTimeout` odmah završavaju
  2. Čeka se 5 sekundi na asinkrono izvršavanje tijela `setTimeout`
  3. Nakon 5 sekundi se kôd iz `setTimeout`-a stavlja u red čekanja
  4. Petlja događaja uzima kôd iz reda čekanja i stavlja ga na stog
  5. Kôd iz `setTimeout` se skida sa stoga i izvršava

```
let timerFunction = () => {  
  setTimeout(() => {  
    console.log('Asynchronous code execution start after 5 sec');  
  }, 5000);  
};  
console.log('Before timerFunction call');  
timerFunction();  
console.log('After timerFunction call');
```



<https://blog.bitsrc.io/understanding-asynchronous-javascript-the-event-loop-74cd408419ff>

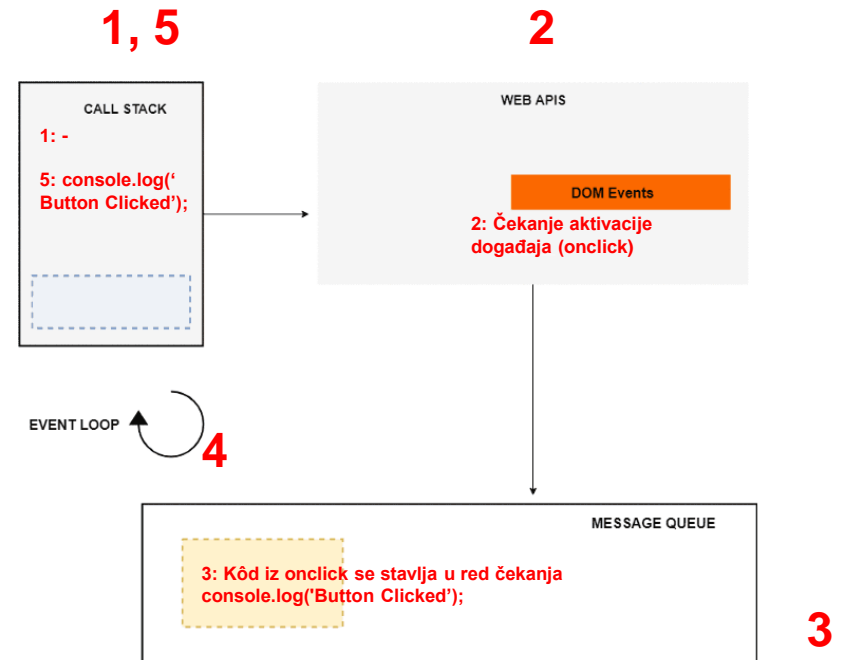
Before timerFunction call

After timerFunction call

Asynchronous code execution start after 5 sec

# Primjer: Događaj DOM-a (eng. DOM event)

- Prilikom izvedbe asinkronog kôda slijedi se ovaj tijek:
  1. -
  2. Slušać događaja (eng. event listener) čeka u okolini web APIs na aktivaciju događaja.
  3. Nakon što se dogodi klik na gumb, kôd sadržan u onclick se stavlja u red čekanja
  4. Petlja događaja uzima kôd iz reda čekanja i stavlja ga na stog
  5. Kôd se skida sa stoga i izvršava



<https://blog.bitsrc.io/understanding-asynchronous-javascript-the-event-loop-74cd408419ff>

```
<!DOCTYPE html>
<html lang="en">
<head>
...
</head>
<body>
  <button onclick="console.log('Button Clicked')">Call click event handler
</button>
</body>
</html>
```

//Stranica se učitava  
//Korisnik klikne na gumb

Button Clicked

# Događaji i asinkrone funkcije

- Asinkrone funkcije se mogu pokrenuti:
  - Nakon nekog događaja (eng. event)
  - Nakon isteka određenog vremena korištenjem ugrađene funkcije `setTimeout`

```
<script>
  function rightClickFunction(){
    alert("Someone right-clicked me!");
  }
  let btn = document.getElementById("btn");
  btn.onclick = function() { alert("Someone clicked me!"); };
  btn.oncontextmenu = rightClickFunction;
  setTimeout(() => {
    alert("This appears 5 secs after the page has loaded");
  }, (5000));
</script>
```

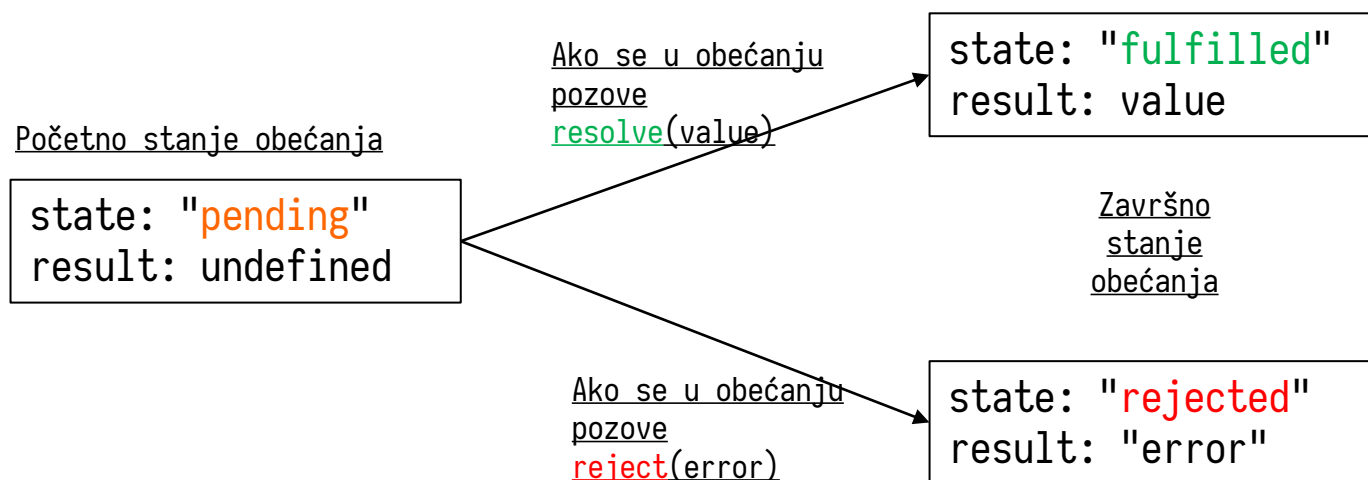
Definicija anonimne funkcije i  
dodjela anonimne funkcije događaju  
`onclick` elementa `btn`

Definicija anonimne funkcije  
pomoću lambda-izraza i dodjela te  
funkcije metodi `setTimeout` za  
pokretanje 5 sekundi nakon što je  
stranica učitana

Dodjela  
postojeće  
funkcije  
događaju  
`oncontextmenu`  
elementa `btn`  
(desni klik)

# Obećanja (eng. promises) (1)

- Uz brojače i događaje JavaScript omogućava i napredne mehanizme ostvarivanja asinkronih poziva
- Obećanja (eng. *promises*) su programski entiteti unutar kojih se postavlja posao koji može potrajati
  - Posao se obavlja unutar tzv. izvršne funkcije (eng. executor)
  - Nakon što je posao završen obećanje prelazi u stanje uspjeha (*fulfilled*) ili neuspjeha (*rejected*)





# Obećanja (eng. promises) (2)

- Obećanje (eng. promise) se *stvara* instanciranjem objekta Promise (pomoću `new Promise`)

U konstruktoru klase Promise se predaje izvršna funkcija koja će pokrenuti asinkroni posao

Izvršna funkcija kao argument prima dvije funkcije: `resolve` i `reject`, koje se po potrebi poziva u tijelu izvršne funkcije

`resolve` i `reject` se definiraju izvan obećanja, u pozivajućem programu

Poziv navedenih funkcija mijenja stanje objekta-obećanja. Ispravno je pozvati samo jednu: ili `resolve` (primjena stanja u `fulfilled`) ili `reject` (promjena stanja u `rejected`)

```
let promise = new Promise(function(resolve, reject) {  
  // this code executes immediately when the constructor is called  
});
```

# Obećanja (eng. promises) (3)

- Obećanja se *koriste* tako da se na njih pretplati (registrira) korištenjem metoda:
  - `.then` - aktivira se ako je izvršna funkcija vratila uspjeh ili neuspjeh
  - `.catch` - aktivira se **samo** ako je izvršna funkcija vratila neuspjeh
  - `.finally` - aktivira se ako je izvršna funkcija vratila uspjeh ili neuspjeh (ali za razliku od `then` ne daje mogućnost izbora između uspjeha ili neuspjeha)

```
let promise = new Promise(function(resolve, reject) {  
  // the code executes immediately when the constructor is called  
});  
promise.then(  
  function(result) { /* handle a successful result */ },  
  function(error) { /* handle an error */ }  
);
```

Uz pomoć `.then`, izvršena je pretplata na obećanje. Ako se u obećanju pozove `resolve`, obavlja se prva funkcija iz `then`, a u slučaju `reject` druga.

Drugu funkciju (`function(error)`) je moguće izostaviti

# Primjer: Obećanje i setTimeout

- Unutar obećanja se obično obrađuje zadatak koji traje
- U ovome primjeru je to setTimeout koji će trajati 3 sekunde

Obećanje koje u sebi pokreće asinkroni posao pomoću funkcije setTimeout

Nakon 3 sekunde obaviti će se posao (console.log) te pozvati metoda resolve

S obzirom da je asinkroni zadatak uspješno završio, nakon njegova završetka se poziva prva funkcija koja je dana kao parametar then-u (function(result)).

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => {
    console.log("After 3 seconds - from the promise");
    resolve("After 3 seconds - msg from promise to then")
  },
  3000);
});
promise.then(
  function(result) { console.log(result) },
  function(error) { /* handle an error */ }
);
console.log("Program continues with other tasks...");
```

Ispis:

Program continues with other tasks...  
After 3 seconds - from the promise  
After 3 seconds - msg from promise to then

# Obećanja – primjeri s resolve i reject (1)

```
let promise = new Promise(function(resolve, reject){
  let randomNumber = Math.floor(Math.random() * 10);
  if(randomNumber%2==1){
    setTimeout(() => {
      resolve("An odd random number was generated.");
    }, 5000);
  }
  else{
    setTimeout(() => {
      reject("An even random number was generated.");
    }, 1000);
  }
});

promise.then(
  function(){ console.log("This appears if resolve is called!");},
  function(){ console.log("This appears if reject is called!");}
);
```

Generiranje  
nasumičnog broja  
od 0 do 9

Ako je broj  
neparan poziva se  
funkcija resolve -  
s kašnjenjem od 5  
sekundi  
(kašnjenje  
predstavlja neku  
akciju koja traje)

Ako je broj paran poziva se funkcija reject - s  
kašnjenjem od 1 sekunde (kašnjenje predstavlja  
neku akciju koja traje)

then kao argumente prima dvije anonimne funkcije: prva  
će se pozvati kada (i ako) izvršna funkcija obećanja pozove  
resolve, a druga kada (i ako) pozove reject

# Obećanja – primjeri s resolve i reject (2)

```
promise = new Promise(function(resolve, reject){
  let randomNumber = Math.floor(Math.random() * 10);
  console.log(randomNumber);
  if(randomNumber%2==1){
    setTimeout(() => {
      resolve("An odd random number was generated.");
    }, 5000);
  }
  else{
    setTimeout(() => {
      reject(new Error("This is generated from the promise executor!"));
    }, 1000);
  }
});

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

Umjesto slanja teksta poruke kao argumenta metode reject, može se poslati instancirani objekt tipa Error.

catch kao argument prima anonimnu funkciju koja će se pozvati kada (i ako) izvršna funkcija obećanja pozove reject. Ako izvršna funkcija pozove resolve, ova anonimna funkcija se ne poziva.

Ako je randomNumber paran, ispisat će se "This is generated from the promise executor".

# Obećanja – primjeri s resolve i reject (3)

```
let promise = new Promise(function(resolve, reject){
  let randomNumber = Math.floor(Math.random() * 10);
  console.log(randomNumber);
  if(randomNumber%2==1){
    setTimeout(() => {
      resolve("An odd random number was generated.");
    }, 5000);
  }
  else{
    setTimeout(() => {
      reject(new Error("This is generated from the promise executor!"));
    }, 1000);
  }
});

promise.catch(
  function(error){ console.log(error);}
).then(
  function(result){ console.log("Resolve:" + result); },
  function(result){ console.log("Reject:" + result) }
);
```

U slučaju generiranja parnog broja i greške, nad obećanjem se poziva catch, i ispisuje poruku greške, a zatim se tijek programa prosljeđuje u then. Aktivira se prva funkcija i ispisuje "Resolve: undefined". Vidljivo je da objekt tipa Error nije dobro prenesen iz catch u then.

U slučaju generiranja neparnog broja aktivira se prva funkcija i ispisuje "Resolve:An odd random number was generated"

## Obećanja – primjeri s resolve i reject (4)

- Ako želimo adekvatno koristiti ulančane pozive funkcija `then/catch/finally` potrebno je iz svake od njih vratiti novi objekt tipa `Promise`
  - na taj način se objekti tipa `Promise` ulančano prenose u iduću razinu i ponovno omogućuju rad metoda `then/catch/finally`

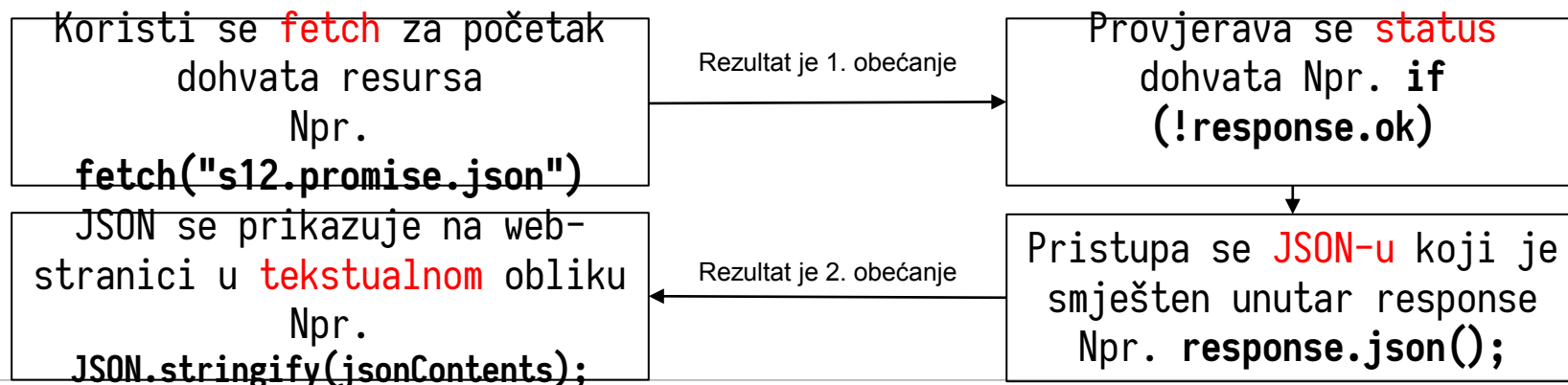
```
promise.catch(  
  function(error){  
    return new Promise(function(resolve, reject){  
      reject(error);  
    });  
  })  
  .then(  
    function(result){ console.log("Resolve:" + result) },  
    function(result){ console.log("Reject:" + result) }  
  );
```

Unutar `catch` se stvara novi objekt tipa `Promise`, koji se linijom `reject(error)` postavlja u stanje greške i prosljeđuje u `then`.

then sada prima gore stvoreni objekt tipa `Promise`, prepoznaje da je u stanju greške te se ispisuje:  
`Reject:Error: This is generated from the promise executor!`

# Primjer: metoda fetch i ulančavanje obećanja (1)

- Metoda fetch u JavaScriptu je dio internetskog preglednika i služi učitavanju različitih resursa (npr. datoteke ili web-stranice)
- fetch je asinkrona metoda zbog činjenice da pristup resursima može trajati (npr. datoteka je velika i treba vremena da se učitava)
  - fetch vraća objekt tipa Promise koji sadrži status dohvata (je li dohvat uspio) – **1. obećanje**
  - Dohvaćeni sadržaj je također dostupan kao novo **2. obećanje**





# Primjer: metoda fetch i ulančavanje obećanja (2)

Koristi se fetch za dohvat datoteke u kojoj je pohranjen JSON

U prvom then-u se obrađuje obećanje dobiveno od fetch-a

U drugom se then-u obrađuje obećanje dobiveno od response.json

```
let promise = fetch("s12.promise.json");
promise
.then(
  function(response) {
    if (!response.ok) { throw new Error("Cannot load json file"); }
    else { return response.json(); }
  },
  function(error) { throw error; }
)
.then(jsonContents => {
  let spanElement = document.createElement('pre');
  spanElement.innerHTML = JSON.stringify(jsonContents);
  document.body.appendChild(spanElement);
})
.catch(err => {
  console.log("An error occured while loading json");
});
```

Prvo obećanje vraća odgovor za koji je potrebno provjeriti svojstvo ok (koje govori da li je učitavanje uspješno)

S response.json() se stvara novo obećanje koje se proslijeđuje u idući then blok na daljnju obradu

Prikaz JSON-a na web-stranici

U slučaju greške bilo u kojem od then blokova ispisuje se poruka na konzolu

# Funkcije s async/await (1)

- Ključne riječi `async` i `await` se tipično koriste u okviru iste funkcije
- Ključnom riječi `await` **se čeka na izvršavanje** neke operacije (tipično dugotrajne)
  - Primjerice, učitavanje datoteke, dohvat resursa s weba i sl.
- `await` je **jedino** moguće koristiti unutar funkcije obilježene s `async`
  - Svaka funkcije obilježena s `async` mora vratiti objekt tipa `Promise`
  - Programska linija s `await` se **ponaša sinkrono** (blokira se tijekom programa **unutar funkcije** i čeka se na

```
async function f() {  
  return 1;  
}
```

Svaka `async` funkcija vraća objekt tipa `Promise`, čak ako se vraća jednostavna vrijednost (npr. `return 1`)

```
async function f() {  
  return 1;  
}  
f().then(alert);
```

Zbog toga što funkcija vraća objekt tipa `Promise`, mogu se nad rezultatom funkcije koristiti funkcije `then/catch/finally`

`await` čeka da se izvrši `Promise.resolve(1)`, te vraća to dobiveno obećanje. Za vrijeme čekanja kôd je blokiran (dolazi do sinkronog čekanja u funkciji `f`)

```
async function f() {  
  return result = await  
    Promise.resolve(1);  
}  
f().then(alert);
```

# Primjer: async/await i fetch

Funkcija je obilježena s ključnom riječi `async` i stoga se u njoj smije koristiti ključna riječ `await`

Prvi `await` blokira izvršavanje **unutar** funkcije sve dok se ne učitava datoteka s JSON-om (**sinkrono** izvršavanje)

Nema više `then`-ova kao kod korištenja obećanja

```
async function LoadJSON(){
  let promise = await fetch("s15.asyncAwaitFetch.json");
  if (!promise.ok) { throw new Error("Cannot load json file"); }
  else { var jsonContents = await promise.json(); }
  let spanElement = document.createElement('pre');
  spanElement.innerHTML = JSON.stringify(jsonContents);
  document.body.appendChild(spanElement);
}
```

Poziva se funkcija `LoadJSON()` **asinkrono**, te se hvataju eventualne greške

Prikaz JSON-a na web-stranici

Drugi `await` čeka da se izvrši metoda `json()` nad obećanjem koje je pohranjeno u `promise` (**sinkrono** izvršavanje)

Greške se hvataju s `catch`, budući da funkcije obilježene s `async` uvijek vraćaju obećanja (čak i ako se dogodi greška)

```
LoadJSON().catch(err => {
  console.log("An error occurred while loading json: " + err);
});
```

# Sličnosti i razlike obećanja i `async/await`

- Obećanja i `async/await` su ravnopravni programski konstrukti, i koriste se ovisno o tome što se programom želi postići
- Obećanja su načelno asinkrona i često se ulančavaju korištenjem `then/catch/finally`
- `Async/await` u odnosu na obećanja ima nešto jednostavniji kôd (nema stalnih poziva `then/catch/finally`), ali uzrokuje potencijalna čekanja u kôdu
  - `await` čeka na sinkrono izvršavanje naredbi i blokira kôd **unutar funkcije** sve dok se naredba sa `await` ne izvrši
  - U slučaju da se `await` ekstenzivno koristi, moguće je sporije izvršavanje kôda
- Obećanja i `async/await` moguće je kombinirati – primjerice pozvati `async` funkciju `f` (iz neke druge funkcije `g`) **asinkrono** koristeći obećanje, a da se unutar funkcije `f` koristi `await` **sinkrono**.

# Moduli (eng. modules) (1)

- Moduli služe organizaciji kôda u JavaScriptu i tipično se koriste kod programskih rješenja s puno programskih linija
- Moduli su datoteke s kôdom u JavaScriptu i tipično imaju ekstenziju .js
  - Svaka datoteka je jedan modul
- Iz jednog je modula moguće po potrebi učitati drugi modul
- Pristup iz jednog modula drugom modulu se regulira direktivama:
  - `export` - označavaju se programski entiteti (varijable, funkcije, klase....) koji su vidljivi izvan modula u kojem se nalaze
  - `import` - omogućava uvoz funkcionalnosti iz drugog modula
- Važna napomena: moduli se tipično koriste kada je aplikacija postavljena na web-poslužitelj
  - Prilikom korištenja na lokalnom datotečnom sustavu učitavanje modula iz internetskog preglednika javlja grešku - internetski preglednik nema dozvole učitavati datoteke s lokalnog datotečnog sustava

# Moduli (eng. modules) (2)

SimpleModule.html

```
<!DOCTYPE html>
<html lang="hr">
  <head>
    <title>Modules</title>
    <meta charset="UTF-8"/>
    <meta name="keywords" content="JavaScript, HTML, programming"/>
    <script type="module" src="SimpleModule1.js"></script>
  </head>
  <body>
    <h1>Modules</h1>
  </body>
</html>
```

Modul "SimpleModule1.js" se uključuje korištenjem oznake script. Kao atribut type postavlja se "module".

Ključnom riječi import uvozi se metoda sum modula SimpleModule2.js u modul SimpleModule1.js

SimpleModule1.js

```
import { sum } from "./SimpleModule2.js";
const a = 3;
const b = 4;
const result = sum(a, b);
alert("Result of a+b is: " + result);
```

SimpleModule2.js

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

# Moduli (eng. modules) (3)

- Ako se jedan te isti modul uveze više puta unutar jednog programa, njegov se kôd izvodi samo jednom, prilikom prvog uvoza
- Prilikom učitavanja modula `m` se stvaraju programski entiteti obilježeni s `export`, te su od tada dostupni ostalim modulima koji uvoze modul `m`
- Ključna riječ `this` nije dostupna unutar modula (ona je `undefined`)
- Ako HTML dokument navodi module (oznakom `script` s `type="module"`), prvo će se učitati HTML dokument, a tek tada moduli (odgođeno učitavanje - eng. `deferred loading`)
  - Ako postoji kôd u JavaScriptu unutar HTML-a, tada se taj kôd učitava prije modula

# Primjer: Korištenje vanjskih paketa

- U vlastite stranice moguće je uključiti i vanjske pakete
  - Npr: [Lodash](#) – paket za manipulaciju objektima i nizovima

```
<!DOCTYPE html>
<html lang="hr">
  <head>
    ...
    <script
src="https://cdn...min.js"></scrip
t>
    ...
  </head>
  <body>
    <script>
      const lodash = _.noConflict();
      const arr = [1, 2, [3, 4], 6, [7, [8]]];
      console.log(lodash.flattenDeep(arr)); // [1, 2, 3, 4,
    </script>
```

Inicijalno postavljanje paketa. `_` je referenca na cijeli paket, a `noConflict()` vraća referencu koju se koristi na stranici

Rekurzivna funkcija `FlattenDeep` stvara jednostavno polje u kojem nema ugniježđenih polja



# Igra CookieClicker v2 (1)

- Igra koju smo ostvarili u prethodnom predavanju bit će nadograđena:
  - Podjelom kôda u module u cilju bolje organizacije
  - Učitavanjem inicijalnih rezultata za rang-ljestvicu iz JSON datoteke
  - Uvođenjem obećanja, asinkronih funkcija i `async/await`
- Promjene će biti samo u JavaScriptu, dok HTML i CSS ostaju praktički isti



# Igra CookieClicker v2 (2)

```
<!DOCTYPE html>
<html lang="hr">
  <head>
    <meta charset="UTF-8">
    <title>Učimo web!</title>
```

**HTML**  
(kao i prije)

```
    <link rel="stylesheet" href="style.css" />
    <script type="module" src="script.js"></script>
  >
</head>
<body>
  ...
</body>
</html>
```

script.js

```
import { CookieGame } from './moduleCookieGame.js'; JS
import { HighScoreManager } from './moduleHighScore.js';
import { Utils } from './moduleUtils.js';
...
```

```
html {
  height: 100%;
}
```

**CSS**  
(kao i prije)

```
body {
  background-color: #0070FF;
  height: 100%;
  display: flex;
  justify-content: space-evenly;
  align-items: center;
  color: #0070FF;
}
...
```

moduleCookieGame.js

```
import { HighScoreManager } from './moduleHighScore.js'; JS
import { Utils } from './moduleUtils.js';

//The main class for the CookieClicker game v1
export class CookieGame {...
```

moduleHighScore.js










```
export class HighScoreManager {... JS
```

moduleUtils.js

```
export class Utils {... JS
```




# Igra CookieClicker v2 (3)

## moduleCookieGame.js

- ✓  CookieGame
  -  cookieCount
  -  gameIntervalTimer
  -  updateTimerUI
  -  updateHighScoreListUI
  -  resetGameUI
  -  constructor
  - >  resetGame
  - >  startGame





CookieGame - glavna klasa koja upravlja igrom

## moduleHighScore.js

- ✓  HighScoreManager
  - >  getHighScores
  - >  updateHighScore

HighScoreManager – klasa s funkcijama za upravljanje najboljim rezultatima

## moduleUtils.js

- ✓  Utils
  - >  setDummyHighscores
  -  formatTwoDigits
  - >  formatTime

Utils - klasa s pomoćnim funkcijama