daftcode

ASYNCHRONICZNY JS

Prowadzący Mariusz Witkowski

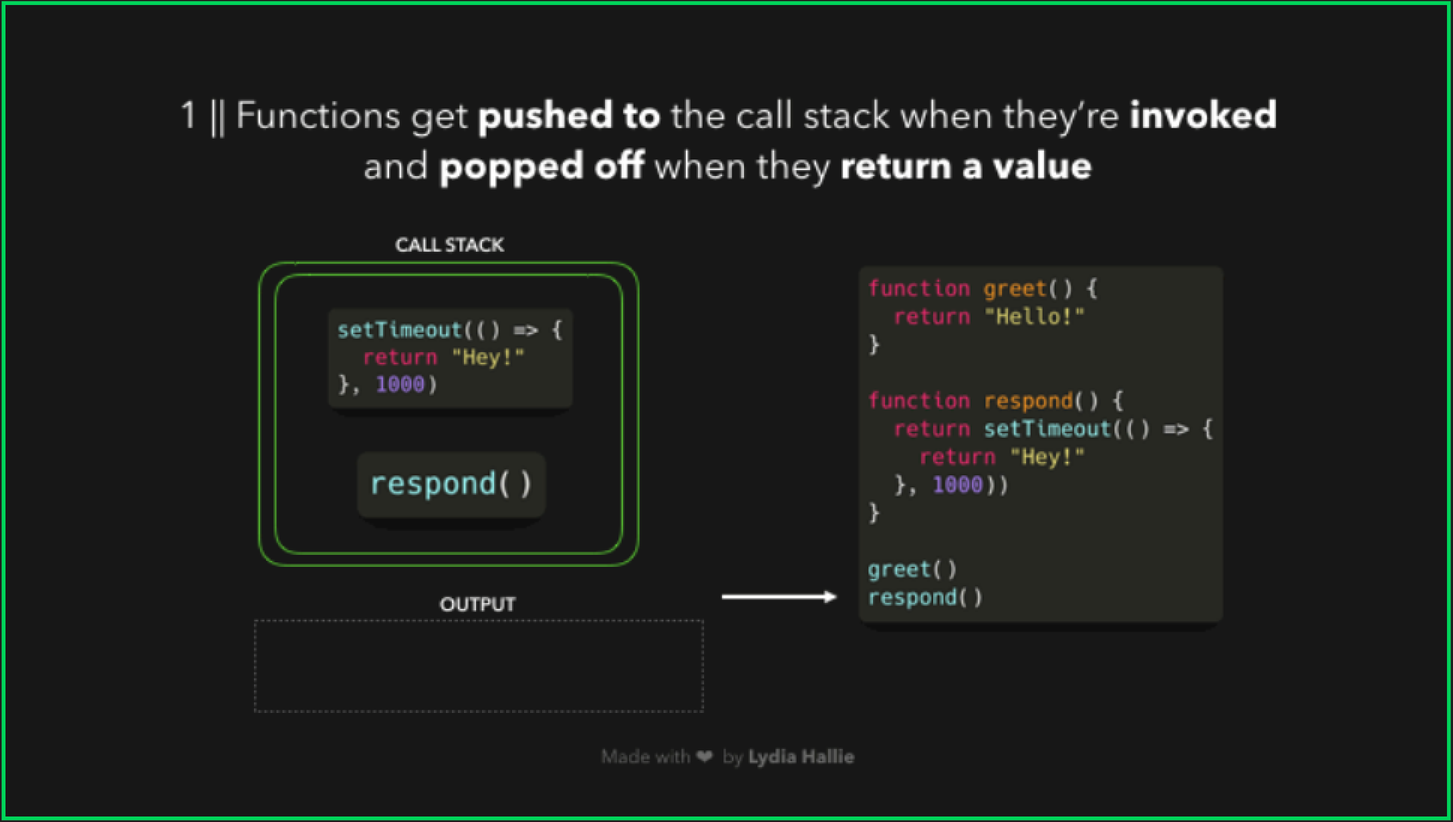
JAVASCRIPT

JavaScript jest jednowątkowym językiem programowania.

Oparty na tzw **Event Loop**

EVENT LOOP

- Call stack *(stos wywołań)*
- Web API *(API przeglądarki)*
- Task Queue *(kolejka zadań)*



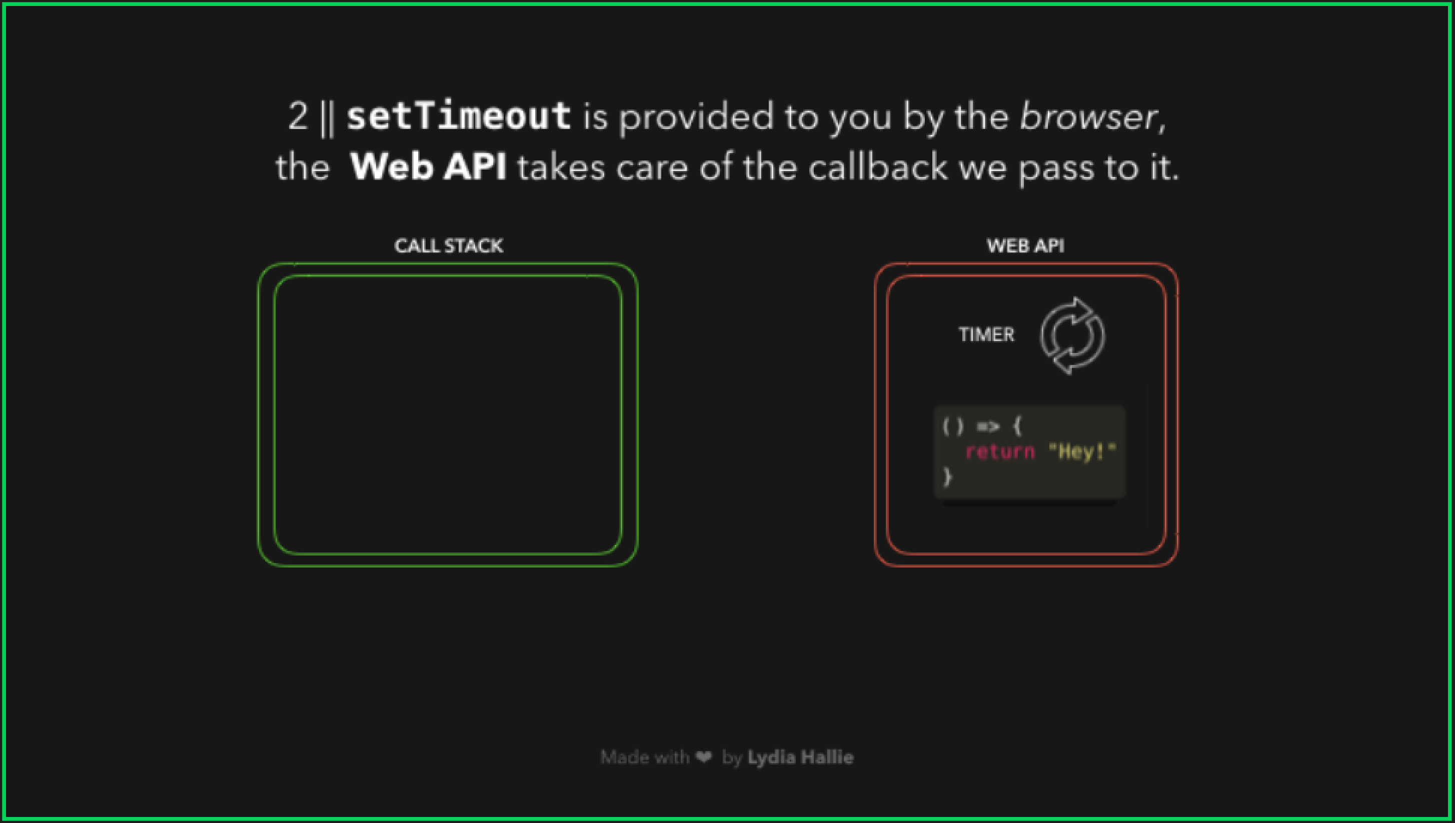
1 || Functions get **pushed to** the call stack when they're **invoked** and **popped off** when they **return a value**

```
setTimeout(() => {
  return "Hey!"
}, 1000)
```

```
respond()
```

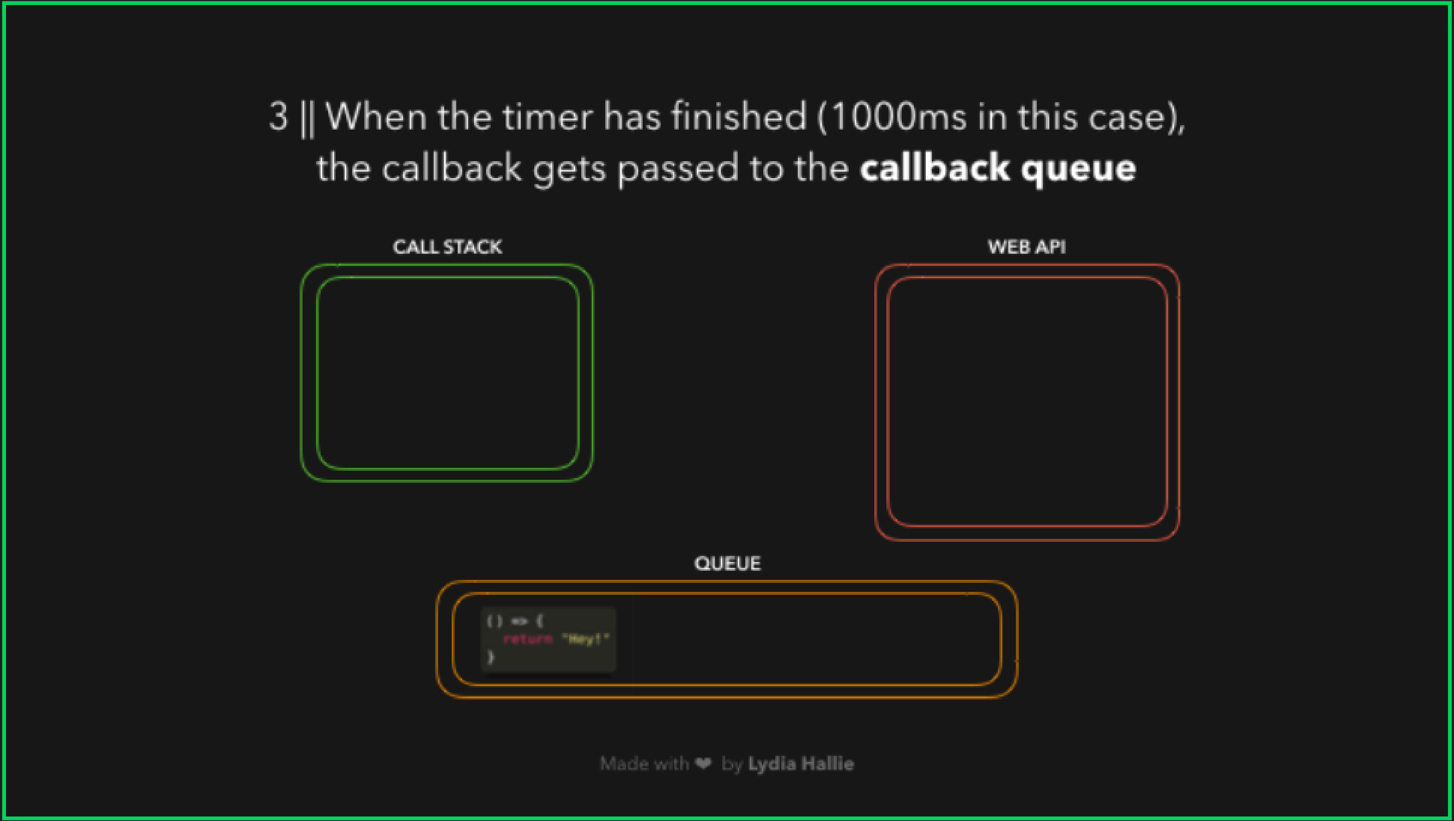
```
function greet() {
  return "Hello!"
}

function respond() {
  return setTimeout(() => {
    return "Hey!"
  }, 1000))
}

greet()
respond()
```

Source: https://dev.to/lydiahallie/javascript-visualized-event-loop-3dif


2 || **`setTimeout`** is provided to you by the *browser*, the **Web API** takes care of the callback we pass to it.

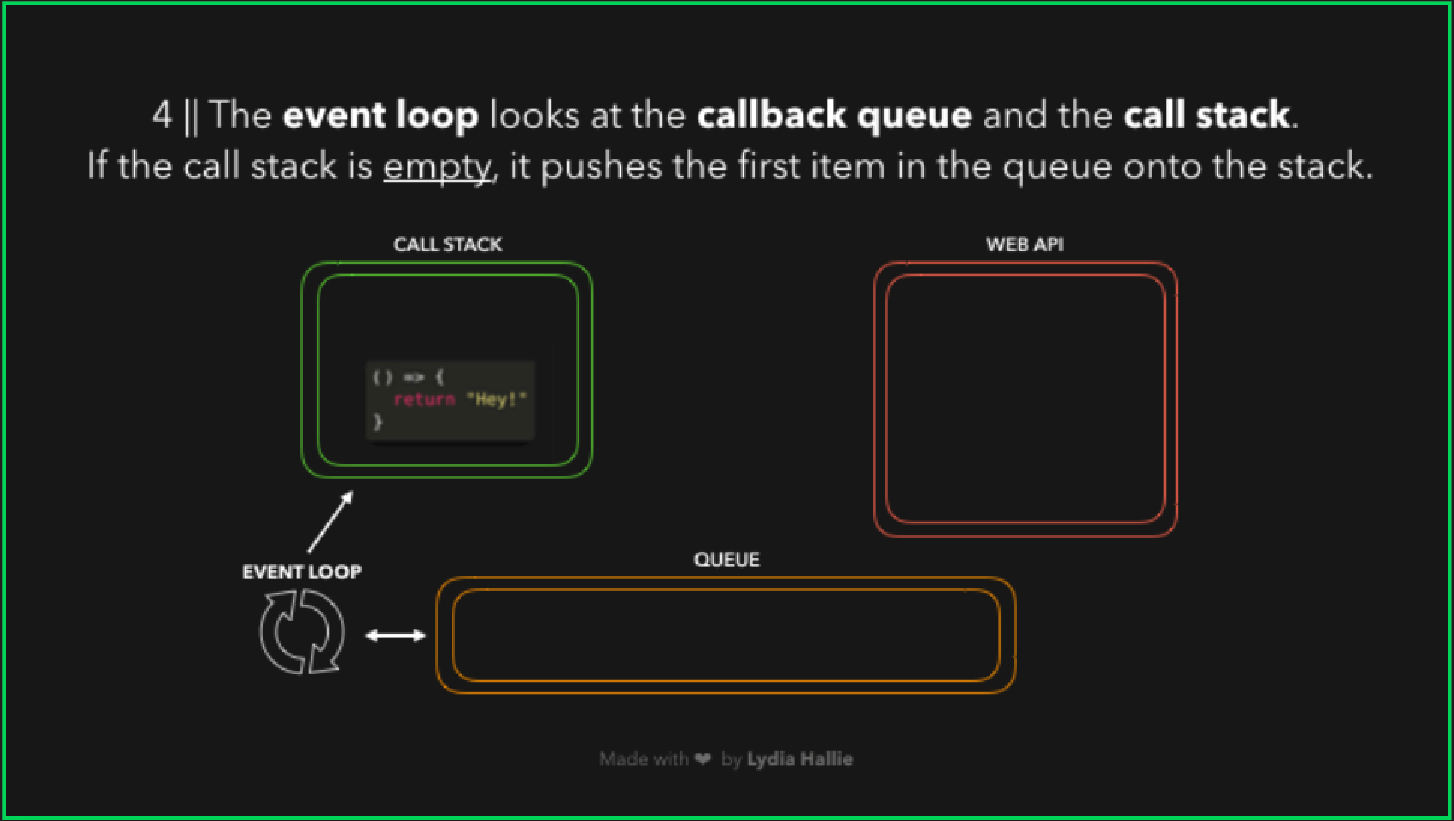
```
() => {
  return "Hey!"
}
```

Source: https://dev.to/lydiahallie/javascript-visualized-event-loop-3dif


3 || When the timer has finished (1000ms in this case), the callback gets passed to the **callback queue**

```
() => {
  return "Hey!"
}
```

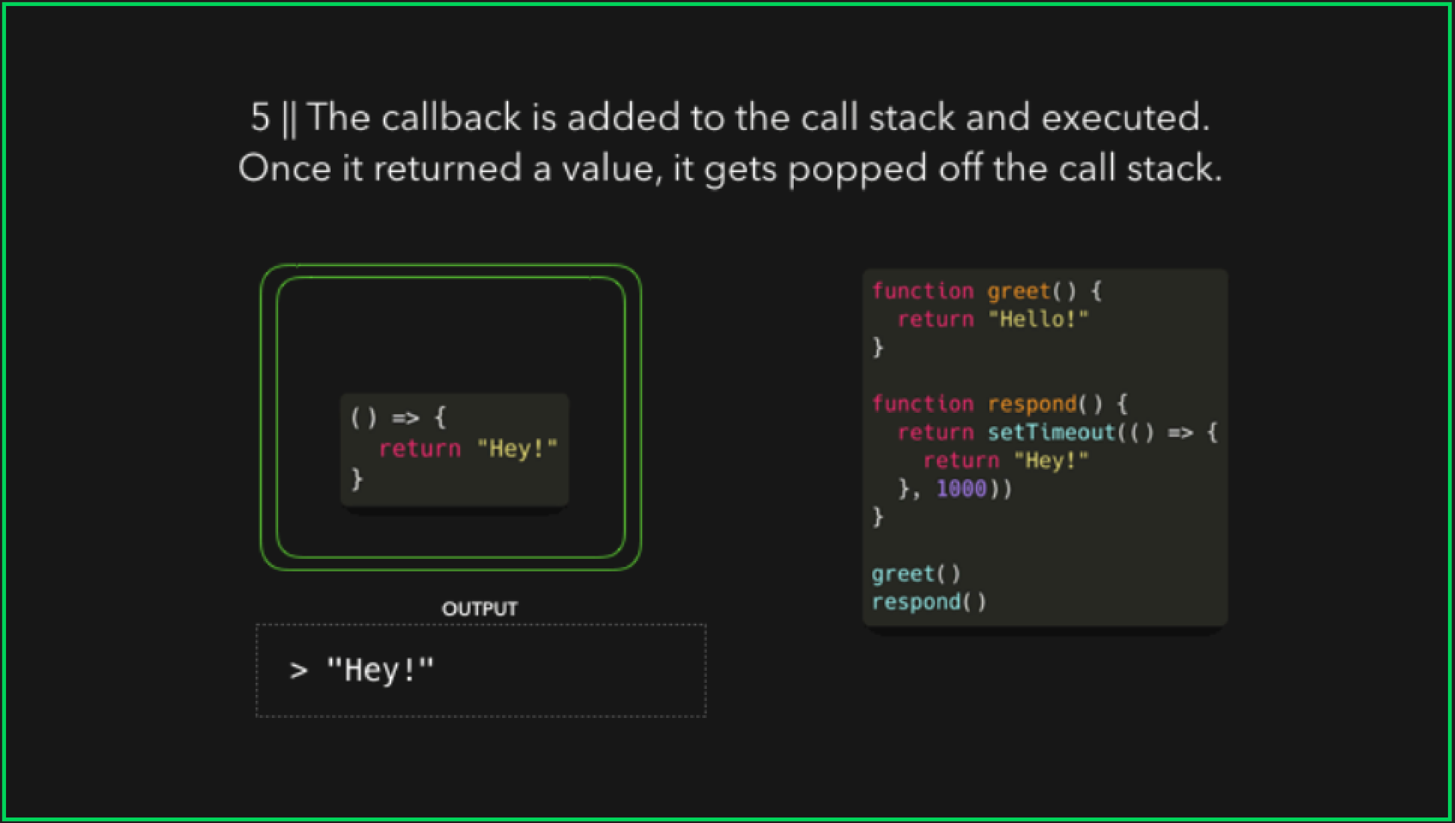
Source: https://dev.to/lydiahallie/javascript-visualized-event-loop-3dif



4 || The **event loop** looks at the **callback queue** and the **call stack**.
If the call stack is empty, it pushes the first item in the queue onto the stack.

```
() => {
  return "Hey!"
}
```

Source: https://dev.to/lydiahallie/javascript-visualized-event-loop-3dif



5 || The callback is added to the call stack and executed. Once it returned a value, it gets popped off the call stack.

```
() => {
  return "Hey!"
}
```

OUTPUT

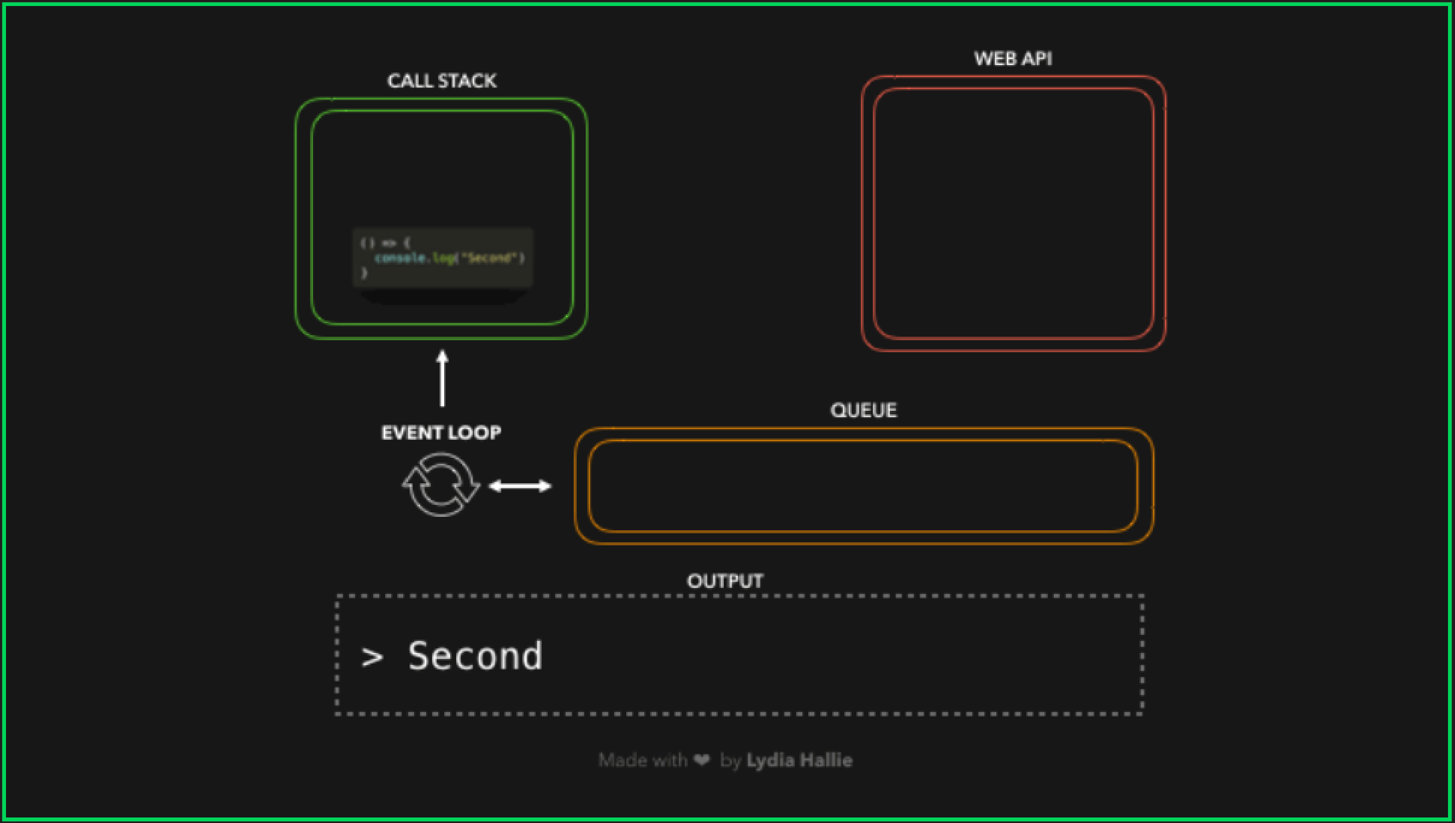
```
> "Hey!"
```

```
function greet() {
  return "Hello!"
}

function respond() {
  return setTimeout(() => {
    return "Hey!"
  }, 1000))
}

greet()
respond()
```

Source: https://dev.to/lydiahallie/javascript-visualized-event-loop-3dif

PRZYKŁAD

```
const foo = () => console.log("First");
const bar = () => setTimeout(() => console.log("Second"), 500);
const baz = () => console.log("Third");

bar();
foo();
baz();
```



```
() => {
  console.log("Second")
}
```

```
> Second
```

Source: https://dev.to/lydiahallie/javascript-visualized-event-loop-3dif

CALLBACK

```
setTimeout(function callback() {}, 1000);

window.addEventListener('load', function callback() {});

request('https://www.example.org', function callback() {});
```

PROBLEM Z CALLBACKAMI

CALLBACK HELL

```
loadTags(function(error, tags) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadPosts(tags, function(error, posts) {
      if (error) {
        handleError(error);
      } else {
        // ...
        loadAuthors(posts, function(error, authors) {
          if (error) {
            handleError(error);
          } else {
            // wykonanie operacji gdy już mamy wszystkie dane
          }
        });
      }
    })
  }
});
```

```
loadTags(tagsCallback);
function tagsCallback(error, tags) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadPosts(tags, postsCallback);
  }
}

function postsCallback(error, posts) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadAuthors(posts, authorsCallback);
  }
}

function authorsCallback(error, authors) {
  if (error) {
    handleError(error);
  } else {
    // wykonanie operacji gdy już mamy wszystkie dane
  }
}
```

INVERSION OF CONTROL

```
request('http://www.somepage.com', function(data) {
// some very important business logic here
});
```

```
function success(data) {
  console.log( data );
}
function failure(err) {
  console.error( err );
}
ajax( "http://some.url.1", success, failure );
```

```
function response(err,data) {
  if (err) {
    throw new Error(err);
  }
  console.log( data );
}
ajax( "http://some.url.1", response );
```

PROMISE

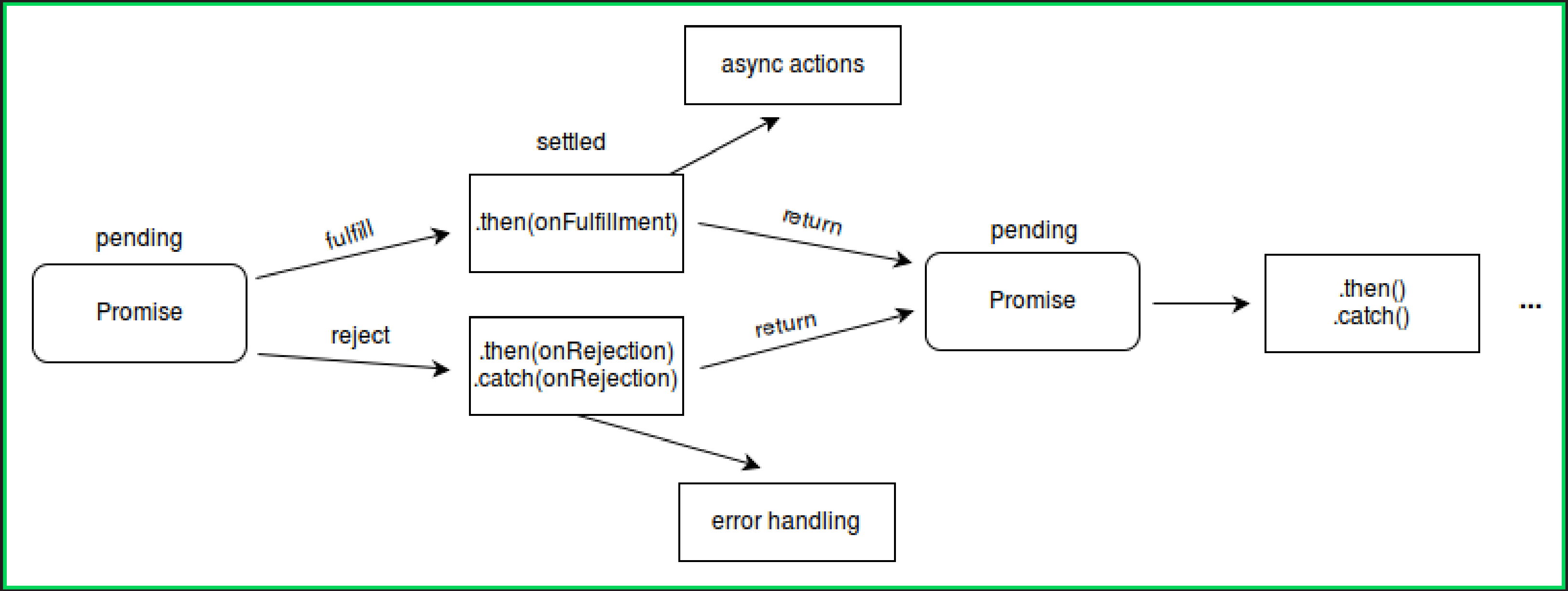
Obiekt **Promise** reprezentuje ewentualne zakończenie (lub porażkę) asynchronicznej operacji i jej wartości.

```
const promise = new Promise(executor);
```

```
function executor(resolve, reject) {
// typically, some asynchronous operation.
}
```

STANY PROMISE'A

- PENDING
- FULFILLED *(resolved)*
- REJECTED



Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise `resolve(value)`

- State: FULFILLED
- Result: value

PRZYKŁAD

- State: FULFILLED
- Result: 'done!'

```javascript
const delaySuccess = function(time) {
  return new Promise((resolve, reject) => {
    setTimeout(() => resolve('done!'), time);
  });
};

const testSuccess = delaySuccess(500)
console.log(testSuccess) // Promise {<pending>}
// after 0.5sec
console.log(testSuccess) // Promise {<resolved>: "done!"}
```

`reject(error)`

- State: REJECTED
- Result: error

PRZYKŁAD

- State: REJECTED
- Result: 'failure'

```
const delayFailure = function(time) {
  return new Promise((resolve, reject) => {
    setTimeout(() => reject('failure!'), time);
  });
};

const testFailure = delayFailure(500)
console.log(testFailure) // Promise {<pending>}
// after 0.5sec
console.log(testFailure) // Promise {<rejected>: "failure!"}
```

UWAGA

Promise może mieć tylko jeden wynik

Po zmianie z **PENDING** na **FULFILLED**/**REJECTED** jego stan nie zmienia się

```
const promise = new Promise(function(resolve, reject) {
  resolve("done");

  reject(new Error("...")); // ignored
  setTimeout(() => resolve("...")); // ignored
});
```

KONSUMERZY

- .then()
- .catch()
- .finally()

.then()

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

PRZYKŁAD

```
delaySuccess(1000).then(
  (result) => console.log(result), // > done!
  (error) => console.error(result), // ignored
);

delayFailure(1000).then(
  (result) => console.log(result), // ignored
  (error) => console.error(result), // > failure!
);
```

.catch()

```
delayFailure(1000).catch(
  (error) => console.error(result), // > failure!
);
```

```
delayFailure(1000).then(
  null,
  (error) => console.error(result), // > failure!
);
```

.finally()

```
delaySuccess(1000).finally(
  (result) => console.log('Finally', result); // > Finally undefined
);

delayFailure(1000).finally(
  (result) => console.log('Finally', result); // > Finally undefined
);
```

PROMISE CHAINING

```javascript
new Promise(function(resolve, reject) {
  setTimeout(() => resolve(1), 1000);
}).then((result) => {
  console.log(result); // 1
  return result * 2;
}).then((result) => {
  console.log(result); // 2
  return result * 2;
}).then((result) => {
  console.log(result); // 4
  return result * 2;
});
```

TO NIE JEST PROMISE CHAINING:

```
const promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve(1), 1000);
});

promise.then((result) => {
  console.log(result); // 1
  return result * 2;
});

promise.then((result) => {
  console.log(result); // 1
  return result * 2;
});

promise.then((result) => {
  console.log(result); // 1
  return result * 2;
});
```

```javascript
const promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve(1), 1000);
});

const promise_1 = promise.then((result) => {
  console.log(result); // 1
  return result * 2;
});

const promise_2 = promise_1.then((result) => {
  console.log(result); // 2
  return result * 2;
});

const promise_3 = promise_2.then((result) => {
  console.log(result); // 4
  return result * 2;
});
```

ZWRACANIE PROMISE Z .then

```
new Promise(function(resolve, reject) {
  setTimeout(() => resolve(1), 1000);
}).then((result) => { // po jednej sekundzie
  console.log(result); // 1
  return result * 2;
}).then((result) => { // natychmiast
  console.log(result); // 2
  return new Promise((resolve, reject) => {
    setTimeout(() => resolve(result * 2), 2000);
  });
}).then((result) => { // po kolejnych dwóch sekundach
  console.log(result); // 4
});
```

PORÓWNUJĄC DO CALLBACKÓW

```
loadTags(tagsCallback);
function tagsCallback(error, tags) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadPosts(tags, postsCallback);
  }
}

function postsCallback(error, posts) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadAuthors(posts, authorsCallback);
  }
}

function authorsCallback(error, authors) {
  if (error) {
    handleError(error);
  } else {
    // wykonanie operacji gdy już mamy wszystkie dane
  }
}
```

```
function loadTags() {
  return Promise(...)
}
function loadPosts(tags) {
  return Promise(...)
}
function loadAuthors(posts) {
  return Promise(...)
}
function authorsCallback(authors) {
  // wykonanie operacji gdy już mamy wszystkie dane
}

loadTags()
.then(loadPosts)
.then(loadAuthors)
.then(authorsCallback)
.catch(handleError);
```

UWAGA!

```
loadTags().then(function (tags) {
  loadPosts(tags).then(function (posts) {
    loadAuthors(posts).then(function (authors) {
      // wykonanie operacji gdy już mamy wszystkie dane
    });
  });
})
.catch(handleError); // ???????????????????????
```

ERROR HANDLING - PROMISES

```
fetch('https://www.nie-ma-takiego-adresu.pl') // REJECT
  .then(response => response.json()) // ignore
  .then(json => stuff(json)) // ignore
  .catch(err => alert(err)) // TypeError: Failed to fetch

fetch('https://www.jest-taki-adres-ale-nie-umie-w-jsona.pl')
  .then(response => response.json()) // REJECT
  .then(json => stuff(json)) // ignore
  .catch(err => alert(err)) // SyntaxError: Unexpected token < in JSON at position 0
```

BŁĄD W EGZEKUTORZE

Jest traktowany tak samo jak reject

```
new Promise((resolve, reject) => {
  throw new Error("Whoops!");
}).catch(alert); // Error: Whoops!

new Promise((resolve, reject) => {
  reject(new Error("Whoops!"));
}).catch(alert); // Error: Whoops!
```

RETHROWING

```
// the execution: catch -> then
new Promise((resolve, reject) => {

  throw new Error("Whoops!");

}).catch(function(error) {

  alert("The error is handled, continue normally");

}).then(() => alert("Next successful handler runs"));
```

.catch() może rzucić błędem...

```
// the execution: catch -> catch -> then
new Promise((resolve, reject) => {

  throw new Error("Whoops!");

}).catch(function(error) { // (*)

  if (error instanceof URIError) {
    // handle it
  } else {
    alert("Can't handle such error");

    throw error; // throwing this or another error jumps to the next catch
  }

}).then(function() {
  /* doesn't run here */
}).catch(error => { // (**)

  alert(`The unknown error has occurred: ${error}`);
  // don't return anything => execution goes the normal way

});
```

UNHANDLED REJECTION

```
new Promise(function() {
  noSuchFunction(); // Error here (no such function)
})
  .then(() => {
    // successful promise handlers, one or more
  }); // without .catch at the end!
```

HANDLE UNHANDLED REJECTION

```
window.addEventListener('unhandledrejection', function(event) {
  // the event object has two special properties:
  alert(event.promise); // [object Promise] - the promise that generated the error
  alert(event.reason); // Error: Whoops! - the unhandled error object
});

new Promise(function() {
  throw new Error("Whoops!");
}); // no catch to handle the error
```

PROMISE API

`Promise.resolve()`

Zwraca resolve'owany Promise z daną wartością

```
function loadData(url) {
  const cache = getCache();
  if (cache.has(url)) {
    return Promise.resolve(cache.get(url));
  }

  return fetch(url)
    .then(response => response.json())
    .then(text => {
      cache.set(url, text);
      return text;
    });
}
```

Promise.reject()

Zwraca reject'owany Promise z daną wartością

```
function loadUser(id) {
  if (id == null) {
    return Promise.reject('ID is required to get user!');
  }

  return fetch(`/users/${id}`)
    .then(response => response.json())
    .then(user => {
      return user;
    });
}
```

Promise.all([promises])

- Argumentem jest tablica promises
- Jeżeli wszystkie resolve'ują, zwraca tablicę z ich wynikami

```
const resolveAfterThreeSec = new Promise(resolve => setTimeout(() => resolve(1), 3000));
const resolveAfterTwoSec = new Promise(resolve => setTimeout(() => resolve(2), 2000));
const resolveAfterOneSec = new Promise(resolve => setTimeout(() => resolve(3), 1000));

Promise.all([
  resolveAfterThreeSec,
  resolveAfterTwoSec,
  resolveAfterOneSec,
]).then((result) => { // after 3 seconds
  console.log(result) // [1, 2, 3]
});
```

`Promise.all([promises])`

- Jeżeli którakolwiek reject'uje, zwraca jej błąd

```
const resolveAfterThreeSec = new Promise(resolve => setTimeout(() => resolve(1), 3000));
const resolveAfterTwoSec = new Promise(resolve => setTimeout(() => resolve(2), 2000));
const failureAfterOneSec = new Promise((resolve, reject) => setTimeout(() => reject('Failure!'), 1000));

Promise.all([
  resolveAfterThreeSec,
  resolveAfterTwoSec,
  failureAfterOneSec,
]).catch((error) => { // after 1 seconds
  console.log(error) // Failure!
});
```

Promise.race([promises])

- Argumentem jest tablica promises
- Resolve'uje się z pierwszym poprawnym promisem

```
const resolveAfterThreeSec = new Promise(resolve => setTimeout(() => resolve(1), 3000));
const resolveAfterTwoSec = new Promise(resolve => setTimeout(() => resolve(2), 2000));
const resolveAfterOneSec = new Promise(resolve => setTimeout(() => resolve(3), 1000));

Promise.race([
  resolveAfterThreeSec,
  resolveAfterTwoSec,
  resolveAfterOneSec,
]).then((result) => { // after 1 seconds
  console.log(result) // 3
});
```

Promise.race([promises])

- Jeżeli którakolwiek reject'uje, zwraca jej błąd

```
const resolveAfterThreeSec = new Promise(resolve => setTimeout(() => resolve(1), 3000));
const resolveAfterTwoSec = new Promise(resolve => setTimeout(() => resolve(2), 2000));
const failureAfterOneSec = new Promise((resolve, reject) => setTimeout(() => reject('Failure!'), 1000));

Promise.race([
  resolveAfterThreeSec,
  resolveAfterTwoSec,
  failureAfterOneSec,
]).catch((error) => { // after 1 seconds
  console.log(error) // Failure!
});
```

Promise.any([promises]) *STAGE 3*

- Argumentem jest tablica promises
- Czeka na pierwszy resolve'wany promise i zwraca jego wynik

```
const resolveAfterThreeSec = new Promise(resolve => setTimeout(() => resolve(1), 3000));
const resolveAfterTwoSec = new Promise(resolve => setTimeout(() => resolve(2), 2000));
const failureAfterOneSec = new Promise((resolve, reject) => setTimeout(() => reject('Failure!'), 1000));

Promise.any([
  resolveAfterThreeSec,
  resolveAfterTwoSec,
  failureAfterOneSec,
]).then((result) => { // after 2 seconds
  console.log(result) // 2
}).catch((error) => {
  console.log(error)
});
```

async/await

Specjalna składnia do pracy z Promises imitująca synchroniczność

async

Funkcja zawsze zwraca Promise

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

foo().then(result => console.log(result)); // 1
```

`await`

- Może być użyty **tylko** w funkcji `async`
- Nakazuje JS poczekać na wynik Promise'a

```
const delaySuccess = function(time) {
  return new Promise((resolve) => {
    setTimeout(() => resolve('done!'), time);
  });
};

async function foo() {
  const result = await delaySuccess(1000); // CZEKA 1s
  return result; // 'done!'
};

foo();
```

```
// ES5
async function foo() {
  const result = await delaySuccess(1000); // CZEKA 1s
  return result; // 'done!'
}

// arrow function
const bar = async () => {
  const result = await delaySuccess(1000); // CZEKA 1s
  return result; // 'done!'
};

// class syntax
class Obj {
  async baz() {
    const result = await delaySuccess(1000); // CZEKA 1s
    return result; // 'done!'
  }
}
```

ERROR HANDLING

```
async function foo() {
  try {
    const response = await fetch('https://www.nie-ma-takiego-adresu.pl');
    return response
  } catch (error) {
    console.error(error); // TypeError: Failed to fetch
  }
};

foo();
```

```
async function bar() {
  const response = await fetch('https://www.nie-ma-takiego-adresu.pl');
  return response
}

bar().catch((error) => console.error(error)); // TypeError: Failed to fetch
```

SYNCHRONICZNE OCZEKIWANIE

```
const loadAfterThreeSec = () => new Promise(resolve => setTimeout(() => resolve(1), 3000));
const loadAfterTwoSec = () => new Promise(resolve => setTimeout(() => resolve(2), 2000));
const loadAfterOneSec = () => new Promise(resolve => setTimeout(() => resolve(3), 1000));

async function loadScripts() {
  const result1 = await loadAfterThreeSec();
  const result2 = await loadAfterTwoSec();
  const result3 = await loadAfterOneSec();

  return result1 + result2 + result3
}

const result = await loadScripts(); // after 6 sec
console.log(result) // 6
```

await Promise.all([promises])

```javascript
const loadAfterThreeSec = () => new Promise(resolve => setTimeout(() => resolve(1), 3000));
const loadAfterTwoSec = () => new Promise(resolve => setTimeout(() => resolve(2), 2000));
const loadAfterOneSec = () => new Promise(resolve => setTimeout(() => resolve(3), 1000));

async function loadScripts() {
  const results = await Promise.all([
    loadAfterThreeSec(),
    loadAfterTwoSec(),
    loadAfterOneSec(),
  ]);

  return results.reduce(sum)
}

const result = await loadScripts(); // after 3 sec
console.log(result) // 6
```

BROWSER API'S

- setTimeout()
- setInterval()
- requestAnimationFrame()
- requestIdleCallback()

`requestAnimationFrame()`

- Wywołuje się przed następnym repaint'em
- Stworzona głównie do animacji
- Dopasowuje się do częstotliwości odświeżania ekranu, ale generalnie stara się wykonywać 60 razy na sekundę

```
let start = null;
const element = document.getElementById("SomeElementYouWantToAnimate");

function step(timestamp) {
  if (!start) start = timestamp;
  const progress = timestamp - start;
  element.style.left = Math.min(progress/10, 200) + "px";
  if (progress < 2000) {
    window.requestAnimationFrame(step);
  }
}

window.requestAnimationFrame(step);
```

requestIdleCallback()

- Kolejkuje funkcję do wywołania, kiedy przeglądarka nie ma co robić.
- Przeznaczona do wykonywania mało istotnych rzeczy

```
const handle = requestIdleCallback(doLowPriorityTask);
const handle2 = requestIdleCallback(doSecondLowPriorityTask, { timeout: 3000 });

cancelIdleCallback(handle);
```

ŹRÓDŁA

- Event loop
  - https://dev.to/lydiahallie/javascript-visualized-event-loop-3dif
  - https://geek.justjoin.it/event-loop-a-kolejnosc-wykonywania-kodu-javascript/
- Promises
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise
  - https://javascript.info/promise-error-handling
  - https://sung.codes/blog/2019/05/18/promise-race-vs-promise-any-and-promi
    all-vs-promise-allsettled/
- Others
  - https://developer.mozilla.org/pl/docs/Web/API/Window/requestAnimationFra
  - https://developer.mozilla.org/en-US/docs/Web/API/Window/requestIdleCallba

daftcode

THANK YOU!