UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

WEB TECHNOLOGIES — LECTURE 08

JAVASCRIPT: ASYNCHRONISM AND NETWORK REQUESTS

Luigi Libero Lucio Starace, PhD

luigiliberolucio.starace@unina.it

https://luistar.github.io

https://www.docenti.unina.it/luigiliberolucio.starace

PREVIOUSLY, ON WEB TECHNOLOGIES

So far we’ve learned the core concepts of JavaScript, and the functionalities available within the **JavaScript Browser Environment**

Today, we’ll complete our overview of JavaScript, and we’ll see:

- **Browser data storage** (**Cookies**, **Local/Session storage**, **IndexedDB**)
- **Asynchronous** Programming constructs
- **Network Requests**

BROWSER DATA STORAGE

BROWSER STORAGE APIs

Modern web browsers provide a number of APIs that can be used via JavaScript to **store** and **retrieve application data**

- **Cookies**
- **Local/Session Storage**
- **IndexedDB**

COOKIES

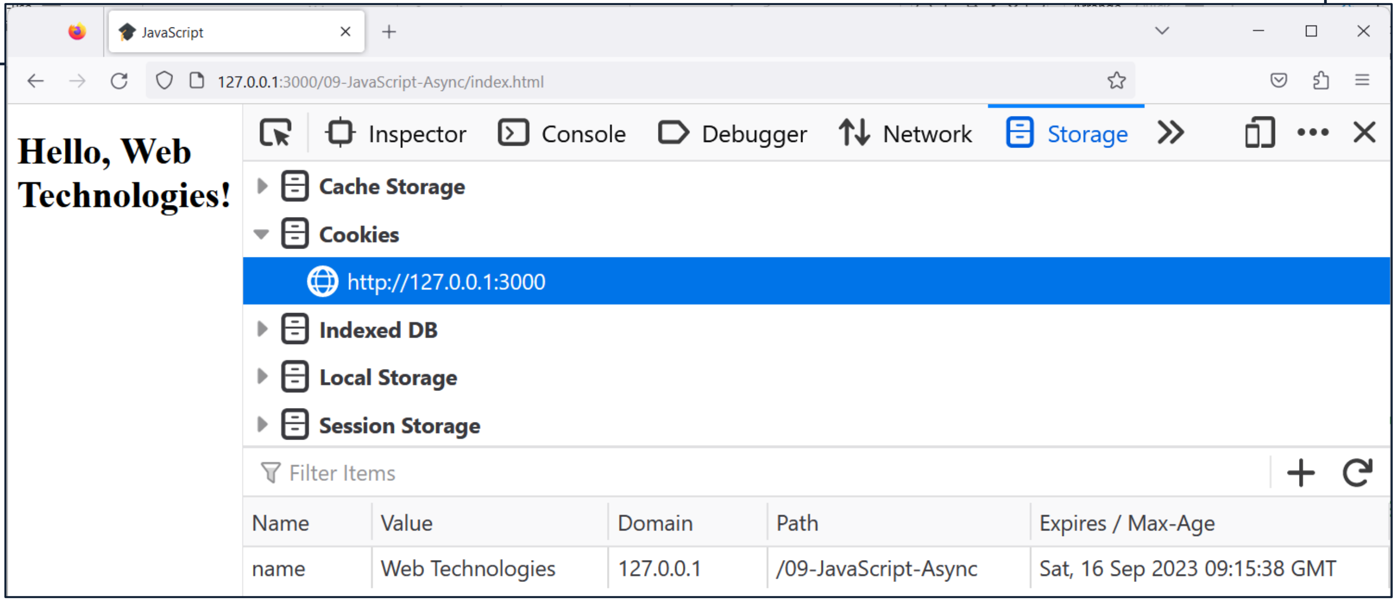
- Cookies are small strings of data
- Part of the HTTP protocol, used to «overcome» its statelessness
- They are typically set in a HTTP response, with the **Set-Cookie** header
- Browsers store them, and send them in all subsequent requests to the same domain, using the **Cookie** header of HTTP requests

COOKIES IN JAVASCRIPT

- Cookies for the current website can be accessed via the **document.cookie** property.
- The value of **document.cookie** is a string of **name=value** pairs, delimited by «**;**».
    - Each pair is a separate cookie.

COOKIES: EXAMPLE

```
<h1>Hello!</h1>
<script>
  if(document.cookie.indexOf("name=") === -1){ //no cookie called "name" found
    let name = prompt("Please, state your name:");
    document.cookie = `name=${name}; max-age=10` //expires in 10 seconds
  }
  let name = document.cookie.replace("name=", "");
  document.querySelector("h1").innerHTML = `Hello, ${name}!`;
</script>
```



Hello, Web Technologies!

Name	Value	Domain	Path	Expires / Max-Age
name	Web Technologies	127.0.0.1	/09-JavaScript-Async	Sat, 16 Sep 2023 09:15:38 GMT

WEB STORAGE: LOCALSTORAGE

Dealing with cookies via JavaScript is not very straightforward.

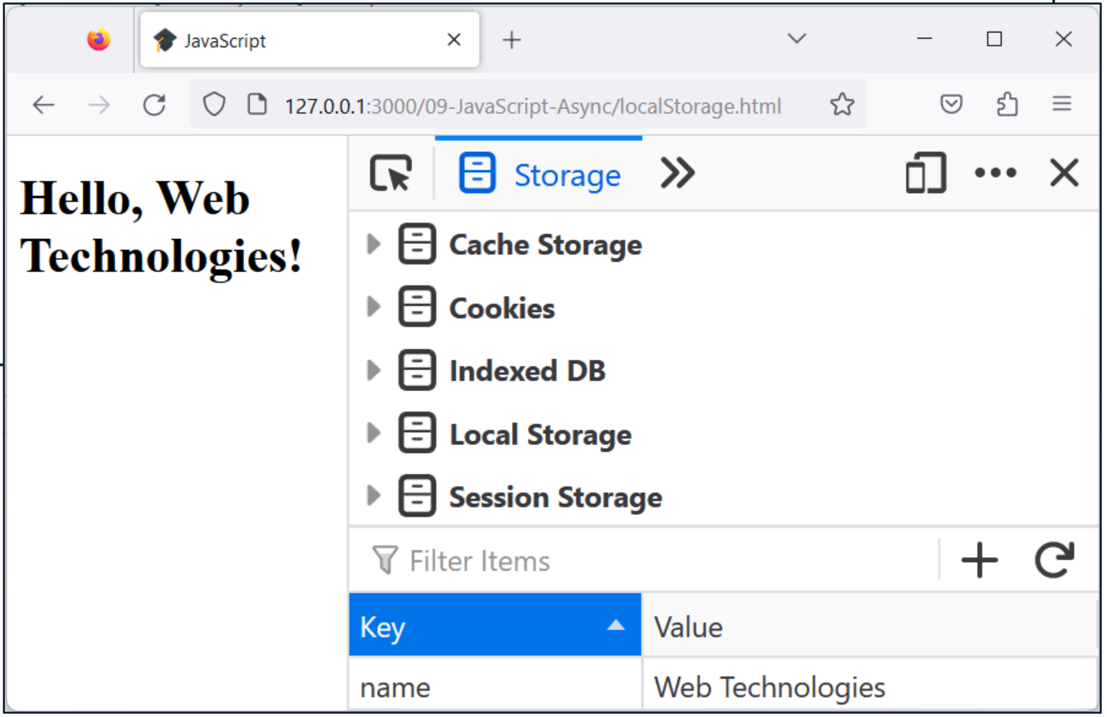
Expecially when dealing with multiple cookies, we need to deal with strings, split them, use regular expressions…

The **localStorage** object provide easy ways to store key/value pairs in web browsers:

- Data is not sent with every request (we can store much more data)
- Quite similar to a **Map**
- Can only store string values
- Different localStorage objects for each **domain/protocol/port**

LOCALSTORAGE: EXAMPLE

```
<h1>Hello!</h1>
<script>
  if(localStorage.getItem("name") === null){ //no "name" stored
    let name = prompt("Please, state your name:");
    localStorage.setItem("name", name);
  }
  let name = localStorage.name;
  document
    .querySelector("h1")
    .innerHTML = `Hello, ${name}!`;
</script>
```



Hello, Web Technologies!

Key	Value
name	Web Technologies

WEB STORAGE: SESSIONSTORAGE

- **localStorage** data survives even a full browser restart
- **sessionStorage** provides the same functionlities as localStorage, but is used much more rarely
- **sessionStorage** only exists within the current browser tab
- Data survives a page refresh, but not closing/reopening the tab.

IndexedDB

IndexedDB is a database built into a browser

- Support different key types, transactions, key-range queries
- Can store even more data than localStorage

- Unnecessarily powerful for traditional client-server web apps, mostly intended for offline apps
- Just keep in mind it exists, we won’t see it in detail

(A)SYNCHRONISM

ASYNCHRONISM

Typically, JavaScript code is executed **synchronously**

- Each instruction **waits** for the previous ones to complete
- Intuitive way of programming, but has a few drawbacks
  - Some instructions might take some time to complete
    - waiting for user prompts
    - waiting for a network request to complete
    - waiting for some complex computation to terminate
  - Subsequent important instructions might be blocked waiting for them
- **Asynchronism** can help address these issues!

SYNCHRONOUS STYLE

```
function firstOperation(value){
  return 1 + value;
}
function secondOperation(value){
  return 2 + value;
}
function thirdOperation(value){
  return 3 + value;
}

function setupPage(){
  let result = 0;
  result = firstOperation(result);
  result = secondOperation(result);
  result = thirdOperation(result);
  console.log(`Result is ${result}`);
}
```

CALLBACKS

Callbacks are functions passed as arguments to other functions, with the expectation that they will be invoked at the appropriate time

Thinking of event handlers? Well, they are callbacks indeed!

ASYNCHRONOUS STYLE: CALLBACKS

```
function firstOperation(value, cb){
  cb(1 + value);
}
function secondOperation(value, cb){
  cb(2 + value);
}
function thirdOperation(value, cb){
  cb(3 + value);
}
```

Callback Hell or Pyramid of Doom!

```
function setupPage(){
  let result = 0;
  firstOperation(result, (r1) => {
    secondOperation(r1, (r2) => {
      thirdOperation(r2, (r3) => {
        console.log(`Result is ${r3}`);
      });
    });
  });
  console.log("Done");
}
```

CALLBACKS: LIMITATIONS

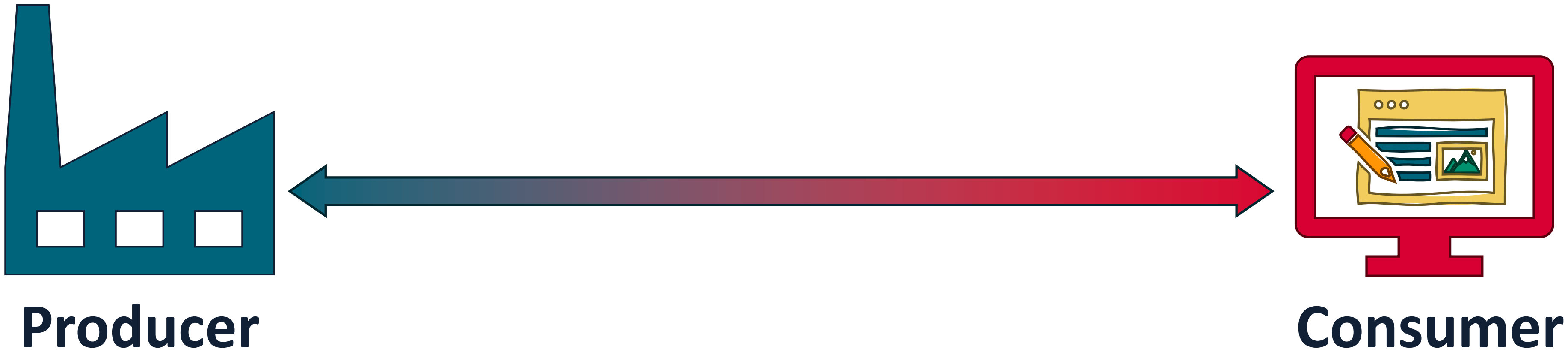
- When handling many callbacks within callbacks, code gets really messy really fast
    - It could also get more complicated. Think of handling errors…
    - There’s a reason it’s called Callback Hell, or Pyramid of Doom
- Old JavaScript handled asynchronism using callbacks.
- In Modern JavaScript, we typically use a new construct: **Promise**

PROMISES: PRODUCERS AND CONSUMERS

In (JavaScript) programming, it often happens that there is:

1. Some «**producer**» code, that does something and takes some time
   - Wait for user inputs, fetches network data, computes complex stuff…
2. Some «**consumer**» code, that needs the results of a producer

Promises are a way to «**link**» producers and consumers



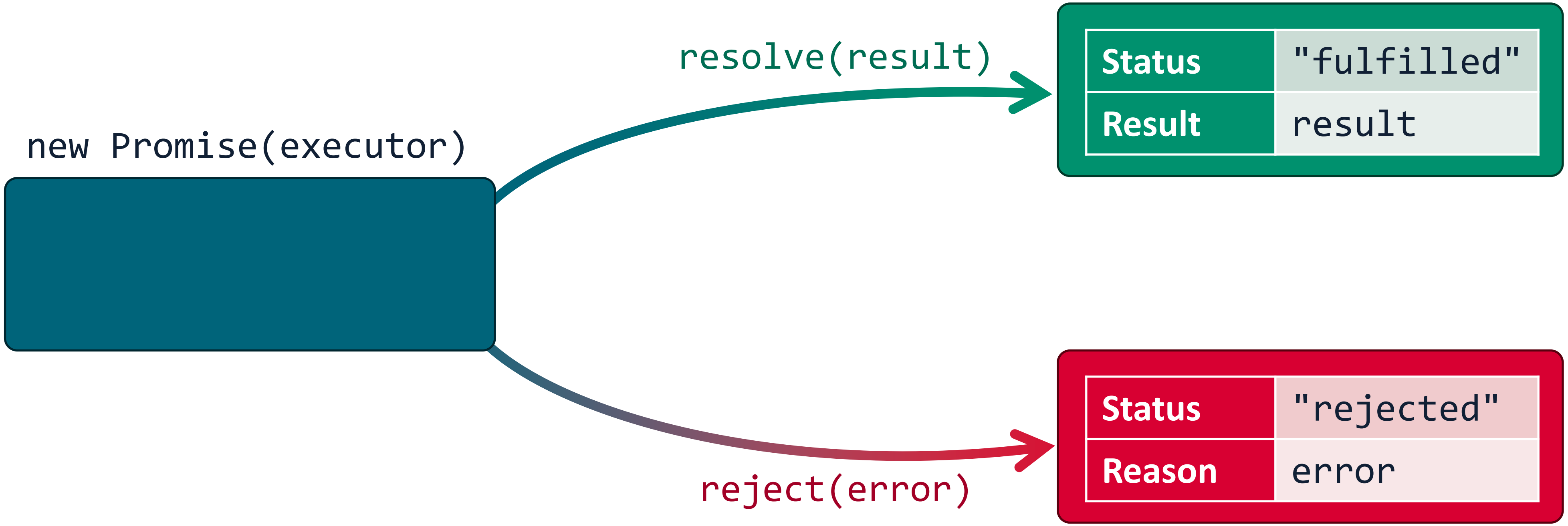
CREATING PROMISES

- A Promise object can be created using the appropriate constructor

```
let promise = new Promise(function(resolve, reject){
  //Executor: producer code here
});
```

- The function given as an argument is the **executor**
- The executor is **immediately invoked** when the Promise is created
- **`resolve`** and **`reject`** are callbacks provided by JavaScript
- When the executor code finishes, it should call either
  - **`resolve(value)`**: if it completed successfully, with result value
  - **`reject(error)`**: if an error occurred (error is the error object)

PROMISES: LIFECYCLE



Status	"fulfilled"
Result	result

Status	"rejected"
Reason	error

Note: Status and Result/Reason are **internal properties** and can’t be directly accessed!

PROMISES: EXAMPLE

```
<script>
let promise = new Promise(function(resolve, reject){
  let number = Math.random();
  setTimeout( () => { // setTimeout takes as input a callback and a time in ms
    if(number > 0.5){
      resolve(number);
    } else {
      reject(new Error("Number too small"));
    }
  }, 1000);
});

console.log(promise); // Promise { <state>: "pending" }
</script>
```

PROMISES: CONSUMER CODE

- «Consumer» code receives a **Promise** that some data will be made available at some point in the future.
- The actions to perform when the Promise is **fulfilled** (or **rejected**) can be registered using specific methods offered by the Promise object.
- The most important of these methods is **.then()**

PROMISES: THEN

- The **.then()** method takes as input **two callbacks**

```
promise.then(
  function(result) { /* called when the promise is fulfilled */ },
  function(error)  { /* called when the promise is rejected */ }
)
```

- It is also possible to register only one function to execute when the promise is fulfilled

```
promise.then(
  (result) => { /* called when the promise is fulfilled */ }
)
```

PROMISES: CATCH

- When we’re only interested in handling error scenarios, we can pass `null` as the first argument:

```
promise.then(
  null,
  function(error)  { /* called when the promise is rejected */ }
)
```

- Or we can use the **`.catch()`** method, which is equivalent

```
promise.catch(
  (error) => { /* called when the promise is rejected */ }
)
```

PROMISES: FINALLY

- Just as try/catch blocks, Promises also feature a **`.finally()`** method
- This method takes a callback with no input
- It is invoked as soon as the Promise is settled: be it resolve or reject
- It is intended to be used to perform clean-ups and other operations that should be performed regardless of the status of the Promise.

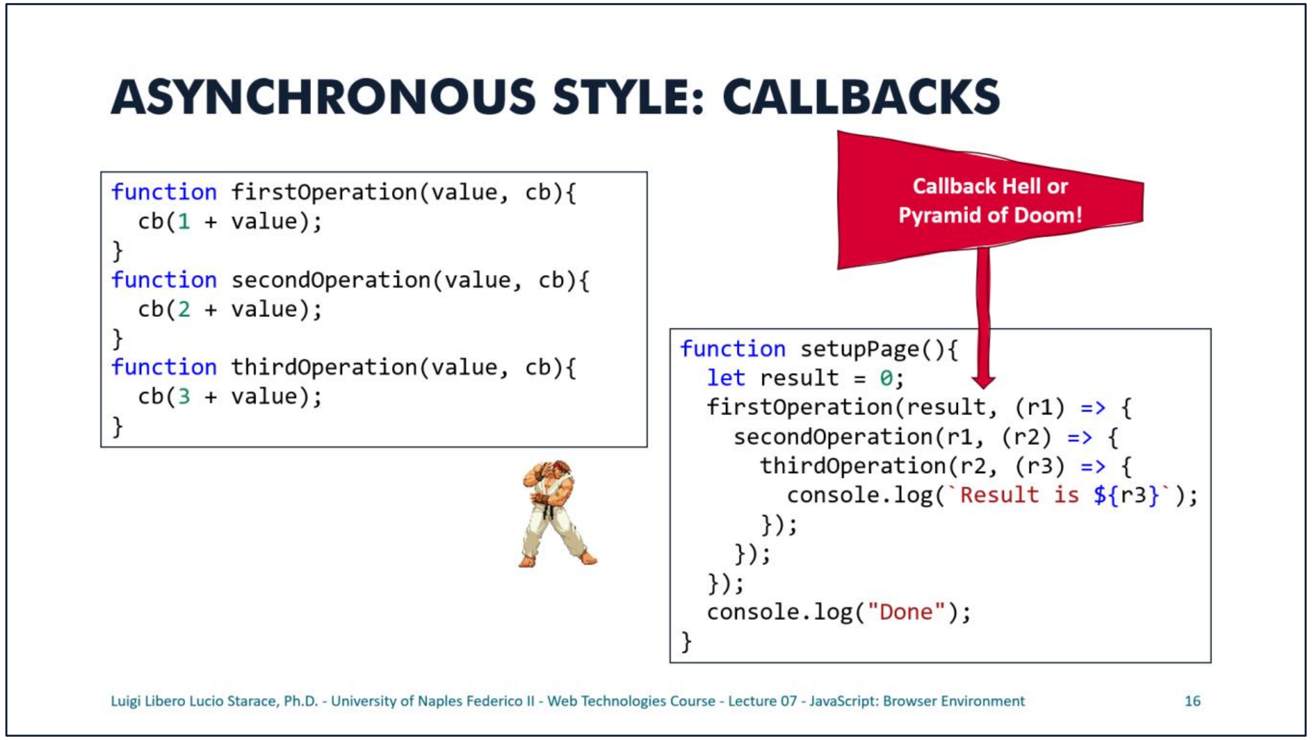
PROMISES: EXAMPLE

```
let promise = new Promise(function(resolve, reject){
  let number = Math.random();
  setTimeout( () => { //setTimeout takes as input a callback and a time in ms
    if(number > 0.5){
      resolve(number);
    } else {
      reject(new Error(`Number too small: ${number}`));
    }
  }, 1000);
});
console.log("Here");
promise.finally( () => {console.log("Promise settled")} )
promise.then(
  (result) => {console.log(`Done: ${result}`)},
  (error)  => {console.log(error.message)}
);
console.log("There");
```

```
// Console output
Here
There
Promise settled
Done: 0.6 (or "Number too small")
```

PROMISES: CHAINING

- What if we need to perform a sequence of asynchronous steps?



ASYNCHRONOUS STYLE: CALLBACKS

```
function firstOperation(value, cb){
  cb(1 + value);
}
function secondOperation(value, cb){
  cb(2 + value);
}
function thirdOperation(value, cb){
  cb(3 + value);
}
```

Callback Hell or Pyramid of Doom!

```
function setupPage(){
  let result = 0;
  firstOperation(result, (r1) => {
    secondOperation(r1, (r2) => {
      thirdOperation(r2, (r3) => {
        console.log(`Result is ${r3}`);
      });
    });
  });
  console.log("Done");
}
```

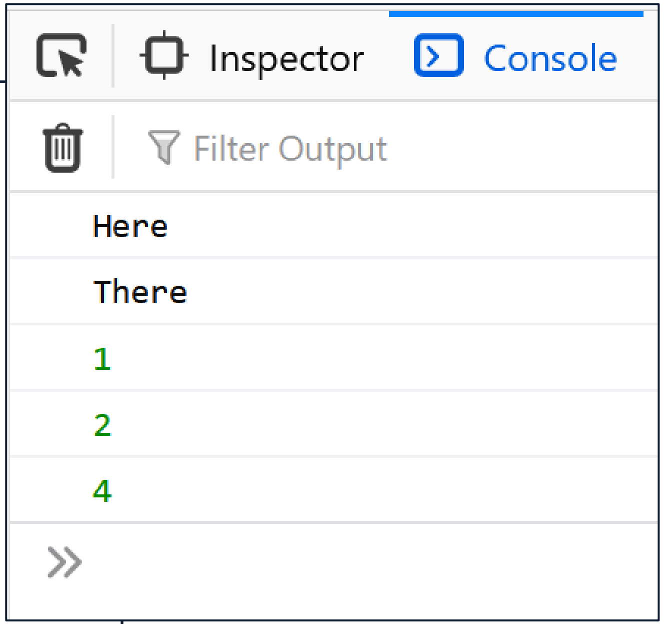
Luigi Libero Lucio Starace, Ph.D. - University of Naples Federico II - Web Technologies Course - Lecture 07 - JavaScript: Browser Environment 16

- **Promise chaining** is the way to deal with such scenarios

PROMISE CHAINING

```
<script>
let promise = new Promise(function(resolve, reject){
  let number = Math.random();
  setTimeout( () => {
    resolve(1);
  }, 1000);
});

console.log("Here");
promise
  .then( (result) => {console.log(result); return result*2;})
  .then( (result) => {console.log(result); return result*2;})
  .then( (result) => {console.log(result); return result*2;})
console.log("There");
</script>
```



```
Here
There
1
2
4
```

PROMISE CHAINING: HOW DOES IT WORK?

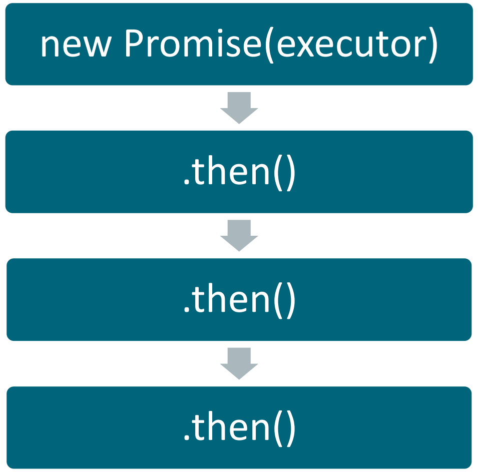
- How does that even work?
- The original Promise is only resolved once, with a result of 1

- Well, each **.then()** invocation actually returns a new Promise obj.
- These new promises represent the completion of the handler itself
- When an handler returns a value, it becomes the result of that new Promise object

PROMISE CHAINING: NEWBIE MISTAKES

- The following snippets are very different!

```
promise //prints 1, then 2, then 4
  .then( (r) => {console.log(r); return r*2;})
  .then( (r) => {console.log(r); return r*2;})
  .then( (r) => {console.log(r); return r*2;});
```



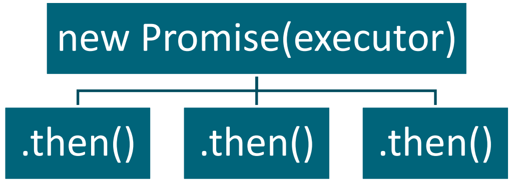
new Promise(executor)

.then()

.then()

.then()

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

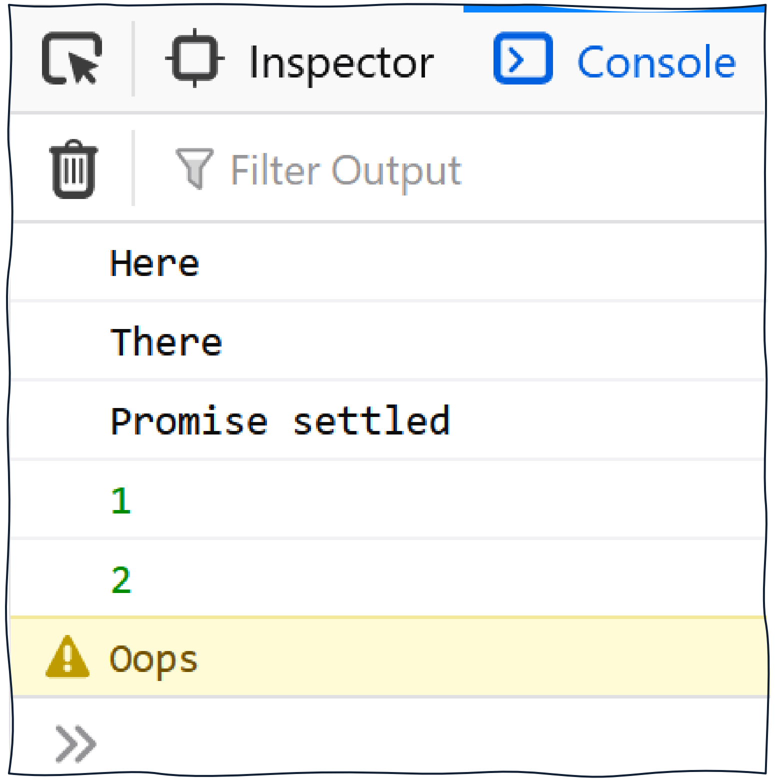


new Promise(executor)

.then() .then() .then()

PROMISE CHAINING: EXAMPLE

```
promise.finally(() => {
  console.log("Promise settled");
}).then((r) => {
  console.log(r);
  return r * 2;
}).then((r) => {
  console.log(r);
  throw new Error("Oops"); //implicit reject()
  return r * 2;
}).then((r) => {
  console.log(r);
  return r * 2;
}).catch((error) => {
  console.warn(error.message);
});
```



```
Here
There
Promise settled
1
2
⚠ Oops
»
```

PROMISE API: PROMISE.ALL()

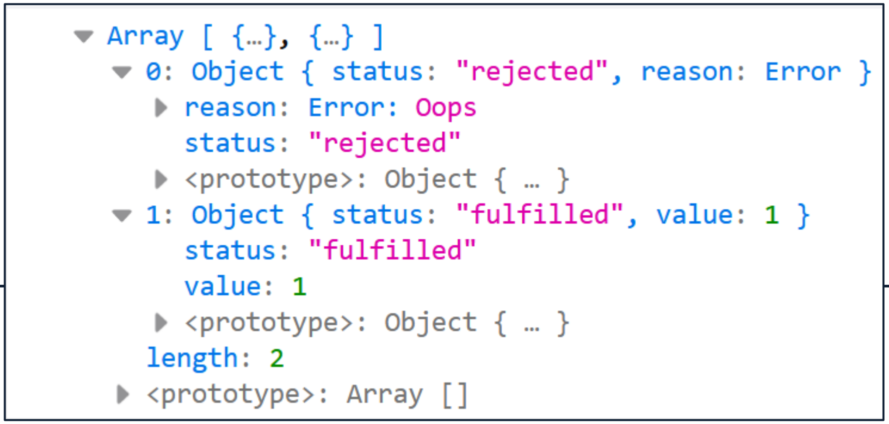
- The **`Promise.all()`** static method takes as input an array of Promises, and returns a new Promise representing the completion of **all** input Promises
- The result is an array, each input Promise contributes and element
- If one Promise rejects, Promise.all immediately rejects

```
let prom = Promise.all([
  new Promise( resolve => setTimeout( () => resolve(3) ), 3000),
  new Promise( resolve => setTimeout( () => resolve(2) ), 2000),
  new Promise( resolve => setTimeout( () => resolve(1) ), 1000),
]).then((result) => {
  console.log(result); //Array(3) [3, 2, 1]
});
```

PROMISE API: PROMISE.ALLSETTLED()

- **Promise.all** rejects if any of the input Promises rejects (**all or nothing**)
- **Promise.allSettled()** is similar: it waits for all the input Promises to settle, but returns as a result an array of objects representing the status of each input Promise.

```
let p = Promise.allSettled([
  new Promise((res, rej) => { setTimeout( () => rej(new Error("Oops")), 1000)}),
  new Promise((res, rej) => { setTimeout( () => res(1), 2000)}),
]).then((result) => {
  console.log(result);
}).catch((err) => {
  console.warn(err.message); //Not executed
});
```



```
▼ Array [ {…}, {…} ]
  ▼ 0: Object { status: "rejected", reason: Error }
    ▶ reason: Error: Oops
      status: "rejected"
    ▶ <prototype>: Object { … }
  ▼ 1: Object { status: "fulfilled", value: 1 }
      status: "fulfilled"
      value: 1
    ▶ <prototype>: Object { … }
    length: 2
  ▶ <prototype>: Array []
```

PROMISE API: PROMISE.RACE()

- Similar to **`Promise.all()`**, but is settled as soon as one of the input Promises settles, and gets its result (or error)

```
let p = Promise.race([
  new Promise((res, rej) => { setTimeout( () => rej(new Error("Oops")), 2000)}),
  new Promise((res, rej) => { setTimeout( () => res(1), 1000)}),
]).then((result) => {
  console.log(result); //1
});

let p2 = Promise.race([
  new Promise((res, rej) => { setTimeout( () => rej(new Error("Oops")), 1000)}),
  new Promise((res, rej) => { setTimeout( () => res(1), 2000)}),
]).catch((err) => {
  console.warn(err.message) //Oops
});
```

PROMISE API: PROMISE.ANY()

- Similar to **`Promise.race()`**, but waits for the first **resolved** Promise

```
let p = Promise.any([
  new Promise((res, rej) => { setTimeout( () => res(3), 3000)}),
  new Promise((res, rej) => { setTimeout( () => rej(new Error("Oops")), 1000)}),
  new Promise((res, rej) => { setTimeout( () => res(1), 2000)}),
]).then((result) => {
  console.log(result); //1
}).catch((err) => {
  console.warn(err.message); //Not executed: Promise.any is resolved
});
```

PROMISE API: PROMISE.ANY()

- If all Promises reject, Promise.any rejects with an AggregateError containing all the Errors of the input Promises

```
let p = Promise.any([
  new Promise((res, rej) => { setTimeout( () => rej(new Error("Woah")), 2000)}),
  new Promise((res, rej) => { setTimeout( () => rej(new Error("Oops")), 1000)}),
  new Promise((res, rej) => { setTimeout( () => rej(new Error("Ouch")), 1000)}),
]).then((result) => {
  console.log(result); //not executed, Promise.any rejected
}).catch((err) =>{
  console.warn(err.message); //No promise in Promise.any was resolved
  console.warn(err.errors[0].message); //Woah
  console.warn(err.errors[1].message); //Oops
  console.warn(err.errors[2].message); //Ouch
});
```

PROMISE API: PROMISE.RESOLVE / REJECT

- **`Promise.resolve(result)`** and **`Promise.reject(error)`** create Promises that are already settled (resp. Resolved or Rejected)

```
let p = Promise.resolve(1); //same as p = new Promise( resolve => resolve(1) )
p.then( result => console.log(result) );
```

- Sometimes they are used for compatibility (e.g.: when a function is expected to return a Promise)

JAVASCRIPT: ASYNC/AWAIT

Modern JavaScript includes a fancy special syntax to work with Promises

- It consists of two keywords: `async` and `await`

- The `async` keyword can be placed before a function to ensure that the function **always returns a Promise**
  - Any other value returned by an `async function` is implicitly wrapped in a resolved Promise
  - Thrown errors are wrapped to a rejected Promise

```
async function f(){
  return new Promise(...);
}
```

```
async function g(){
  return 1; //returns Promise.resolve(1)
}
```

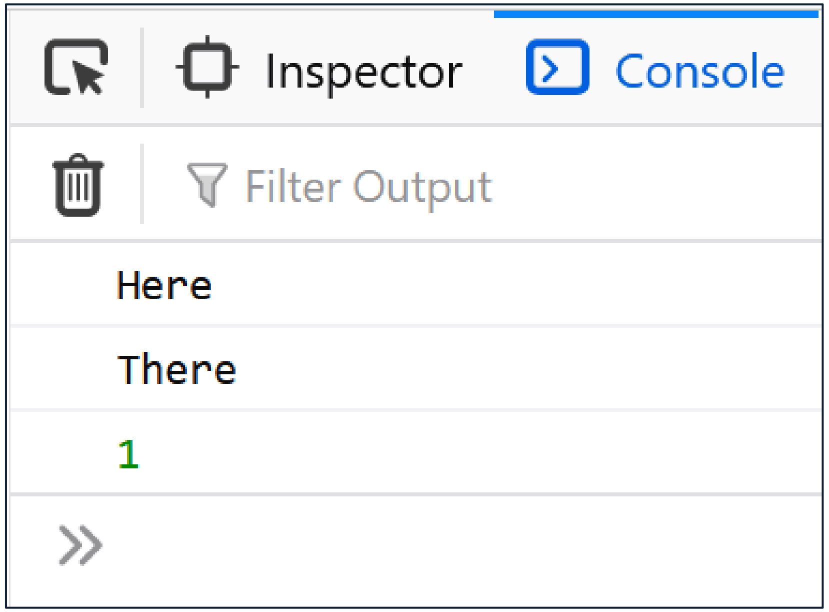
JAVASCRIPT: ASYNC

```
console.log("Here");

async function f(){
  //throw new Error("Ouch");
  return 1;
}

f().then(console.log).catch(console.warn);

console.log("There");
```

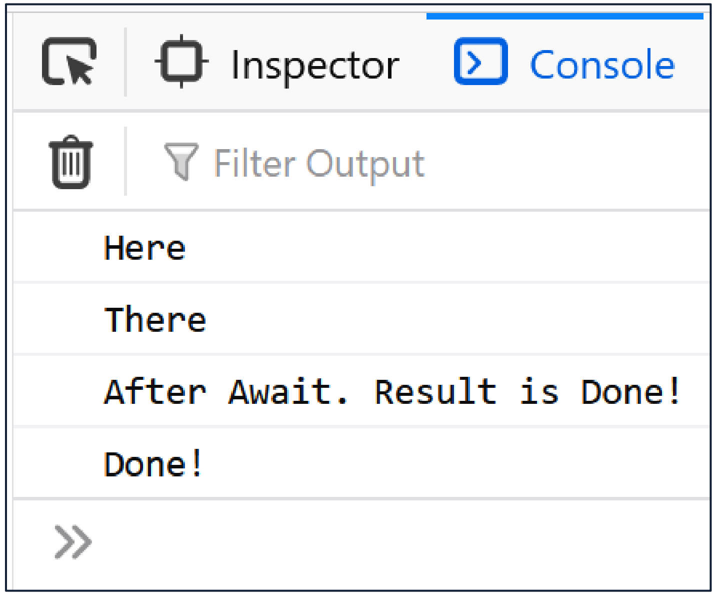


```
Here
There
1
```

JAVASCRIPT: AWAIT

- The await keyword can be used **only in async functions**
- It **pauses** code execution until a Promise is settled

```
console.log("Here");
async function f(){
  let promise = new Promise(function(res, rej){
    setTimeout(() => {res("Done!")}, 2000);
  });
  let result = await promise; //execution pauses here
  console.log(`After Await. Result is ${result}`);
  return result;
}
f().then(console.log);
console.log("There");
```



```
Here
There
After Await. Result is Done!
Done!
```

NETWORK REQUESTS

NETWORK REQUESTS

- JavaScript can also fetch data from the internet
- In old JavaScript, programmers used XMLHttpRequest objects
  - Callbacks were used to handle status events

```
<h1>Cat Facts</h1>
<button id="btn">Click Here To Load a Cat Fact</button> <p id="fact"></p>
<script>
  let btn = document.getElementById("btn");
  btn.onclick = () => {
    let req = new XMLHttpRequest(); req.onload = handleFact;
    req.open("GET", "https://catfact.ninja/fact"); req.send();
    function handleFact(result) {
      let fact = JSON.parse(this.responseText);
      document.getElementById("fact").innerHTML = fact.fact;
    }
  }
</script>
```

NETWORK REQUESTS: THE FETCH API

- The modern way of doing HTTP requests within JavaScript is **fetch()**
- **fetch()** takes as input an **URL** and an **optional array of options**

```
let promise = fetch("url", [options]);
```

- Options can be used to specify HTTP method, request headers, etc…
- A Promise is returned, which resolves to a **Response** object

FETCH API: WORKING WITH RESPONSES

- The Promise returned by fetch is resolved **as soon as the response headers are received**
- At this stage, we can check **HTTP status**, **headers**, but the **body** of the response might not be available yet

FETCH API: WORKING WITH RESPONSES

```javascript
async function f(){
  let response = await fetch("./example.json"); //there is no such file
  //get one specific header
  console.log(response.headers.get('Content-Type')); // application/json
  //iterate over all response headers
  for(let [key, value] of response.headers){
    console.log(`Header ${key}: ${value}`);
  }
  let status = response.status;
  console.log(status);
  if(response.ok){
    console.log("Request was successful");
  } else {
    console.log("Some error occurred");
  }
}
```

FETCH API: GETTING THE RESPONSE BODY

- When a fetch promise resolves, the body might not be there yet
- The Response provides several promise-based methods to access the body, in various formats. For example, **.json()** parses the body as a JSON as soon as it becomes available.

```
async function f(){
  let response = await fetch("https://catfact.ninja/fact");
  if(response.ok){
    let fact = await response.json();
    document.getElementById("fact").innerHTML = fact.fact;
  } else {
    console.log("Some error occurred");
  }
}
```

FETCH API: GETTING THE RESPONSE BODY

- Some other methods to access the body are:

.text()	Read the response body and return it as text
.json()	Parse the response body as JSON
.blob()	Return a Blob (Binary data with type)
.arrayBuffer()	Return an ArrayBuffer (low level repr. of binary data)
.formData()	Return a FormData object (represents data submitted via forms)

- **Important:** we can only choose one body-reading method! If you have already called **response.text()**, **response.json()** won’t work because the body content has already been consumed!

FETCH API: BLOB EXAMPLE

```
<input id="msg" placeholder="Insert your message here" type="text">
<button id="btn">Generate QR</button><img id="img">
<script>
document.getElementById("btn").onclick = () => {
  let msg = document.getElementById("msg").value;
  let url = `https://image-charts.com/chart?chs=200x200&cht=qr&chl=${msg}&choe=UTF-8`;
  console.log(url);
  let promise = fetch(url);
  promise.then((response) => {
    return response.blob();
  }).then((blob) => {
    let img = document.getElementById("img");
    img.src = URL.createObjectURL(blob);
  });
}
</script>
```

REFERENCES

- **The Modern JavaScript Tutorial**
  Freely available at https://javascript.info/ or on GitHub
  Part 2: Storing data in the browser (4.1, 4.2)
  Part 1: Promises and async/await (11.1 to 11.5, 11.8)
  Part 3: Network requests (3.1, 3.8)
- **Eloquent JavaScript (3rd edition)**
  By Marijn Haverbeke
  Freely available at https://eloquentjavascript.net/
  Chapters 11, 18
- **JavaScript Reference**
  MDN web docs
  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference