ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

JavaScript напредни концепти

Iterators

- In JavaScript an iterator is an object which defines a sequence and potentially a return value upon its termination.
- Specifically, an iterator is any object which implements the Iterator protocol by having a **next()** method that returns an object with two properties:
  - **value** - The next value in the iteration sequence.
  - **done** - This is **true** if the last value in the sequence has already been consumed. If **value** is present alongside **done**, it is the iterators return value.
- Once created, an iterator object can be iterated explicitly by repeatedly calling **next()**.
- Iterating over an iterator is said to consume the iterator, because it is generally only possible to do once.
- After a terminating value has been yielded additional calls to **next()** should simply continue to return **{done: true}**.

Iterators (2)

- Here is an example which can do just that. It allows creation of a simple range iterator which defines a sequence of integers from **start** (inclusive) to **end** (exclusive) spaced **step** apart.

```
function makeRangeIterator(start = 0, end = Infinity, step = 1) {
    let nextIndex = start;
    let iterationCount = 0;

    const rangeIterator = {
       next: function() {
           let result;
           if (nextIndex < end) {
               result = { value: nextIndex, done: false }
               nextIndex += step;
               iterationCount++;
               return result;
           }
           return { value: iterationCount, done: true }
       }
    };
    return rangeIterator;
}
```

Iterators (3)

- Using the iterator then looks like this:

```
const it = makeRangeIterator(1, 10, 2);

let result = it.next();
while (!result.done) {
 console.log(result.value); // 1 3 5 7 9
 result = it.next();
}

console.log("Iterated over sequence of size: ", result.value); // [5 numbers returned, that took interval in between: 0 to 10]
```

Generators

- While custom iterators are a useful tool, their creation requires careful programming due to the need to explicitly maintain their internal state.
- Generator functions provide a powerful alternative: they allow you to define an iterative algorithm by writing a single function whose execution is not continuous.
- Generator functions are written using the **function*** syntax.
- When called, generator functions do not initially execute their code. Instead, they return a special type of iterator, called a **Generator**. When a value is consumed by calling the generator's **next** method, the **Generator** function executes until it encounters the **yield** keyword.

Generators (2)

- The function can be called as many times as desired, and returns a new Generator each time. Each Generator may only be iterated once.

```
function* makeRangeIterator(start = 0, end = 100, step = 1) {
    let iterationCount = 0;
    for (let i = start; i < end; i += step) {
        iterationCount++;
        yield i;
    }
    return iterationCount;
}
```

Iterables

- An object is iterable if it defines its iteration behavior
  - values are looped over in a for...of construct
- Built-in iterables
  - String, Array, TypedArray, Map and Set

```
for (const x of "Iterables") {
  // code block to be executed
}

for (const x of [1,2,3,4,5]) {
  // code block to be executed
}
```

Promises

- A “producing code” that does something and takes time. For instance, some code that loads the data over a network.
- A “consuming code” that wants the result of the “producing code” once it’s ready. Many functions may need that result.
- A promise is a special JavaScript object that links the “producing code” and the “consuming code” together. The “producing code” takes whatever time it needs to produce the promised result, and the “promise” makes that result available to all of the subscribed code when it’s ready.

Promises (2)

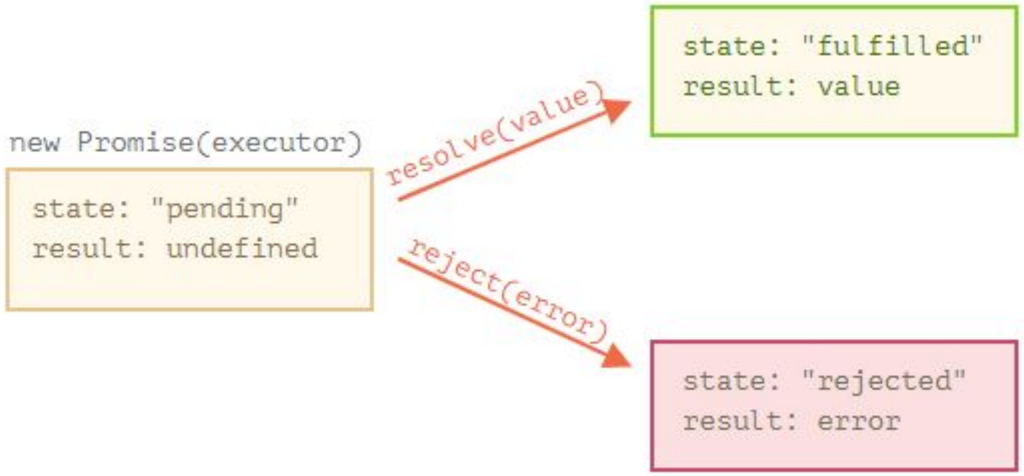
- The constructor syntax for a promise object is:

```
let promise = new Promise(function(resolve, reject) {
  // executor (the producing code, "singer")
});
```

- The function passed to **new Promise** is called the executor. When **new Promise** is created, the executor runs automatically. It contains the *producing code* which should eventually produce the result.
- Its arguments **resolve** and **reject** are callbacks provided by JavaScript itself. *Our code* is only inside the executor.
- When the executor obtains the result, be it soon or late, doesn't matter, it should call one of these callbacks:
  - **resolve(value)** — if the job finished successfully, with result **value**.
  - **reject(error)** — if an error occurred, **error** is the error object.

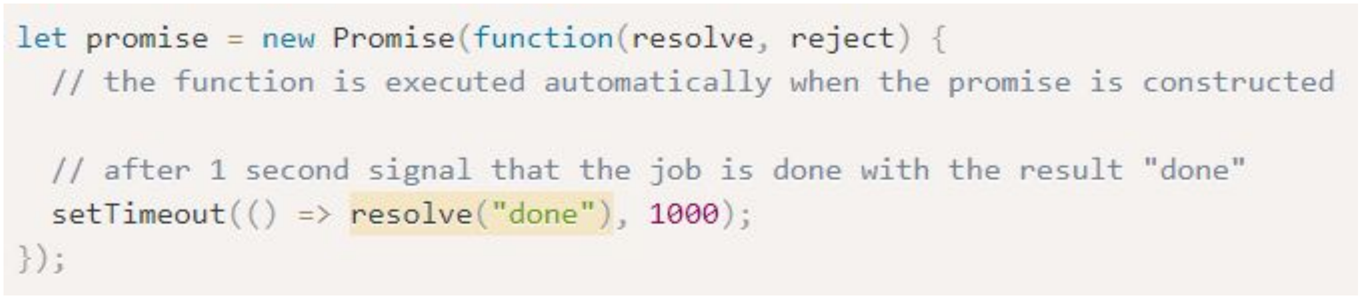
Promises (3)

- The **promise** object returned by the **new Promise** constructor has these internal properties:
  - **state** — initially "*pending*", then changes to either "*fulfilled*" when **resolve** is called or "*rejected*" when reject is called.
  - **result** — initially **undefined**, then changes to **value** when **resolve(value)** called or **error** when **reject(error)** is called.
- The executor eventually moves **promise** to one of these states:

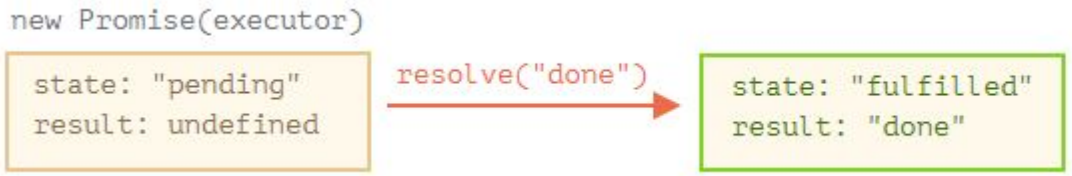


Promises (4)

- Here’s an example of a promise constructor and a simple executor function with “producing code” that takes time (via **setTimeout**):



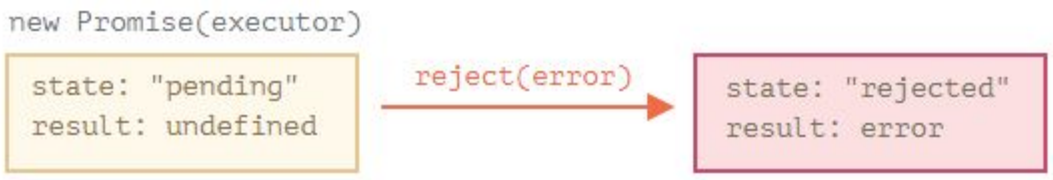
```
let promise = new Promise(function(resolve, reject) {
  // the function is executed automatically when the promise is constructed

  // after 1 second signal that the job is done with the result "done"
  setTimeout(() => resolve("done"), 1000);
});
```



Promises (5)

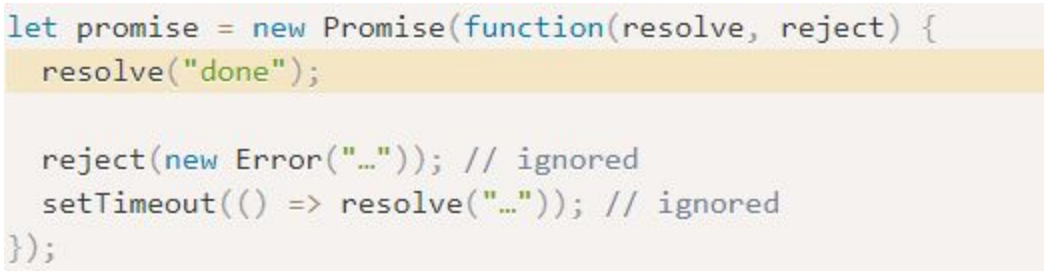
- And now an example of the executor rejecting the promise with an error:

```
let promise = new Promise(function(resolve, reject) {
  // after 1 second signal that the job is finished with an error
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});
```



Promises (6)

- The executor should call only one **resolve** or one **reject**. Any state change is final.



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

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

- Also, **resolve**/**reject** expect only one argument (or none) and will ignore additional arguments.

Promises (7)

- The properties **state** and **result** of the **Promise** object are internal. We can't directly access them. We can use the methods **.then**/**.catch**/**.finally** for that.

- A **Promise** object serves as a link between the executor (the "producing code") and the consuming, which will receive the result or error.

- Consuming functions can be registered (subscribed) using methods **.then**, **.catch** and **.finally**.

Promises (8)

- The most important, fundamental one is **.then**.

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

- The first argument of **.then** is a function that runs when the promise is resolved and receives the result.
- The second argument of **.then** is a function that runs when the promise is rejected and receives the error.

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve("done!"), 1000);
});

// resolve runs the first function in .then
promise.then(
  result => alert(result), // shows "done!" after 1 second
  error => alert(error) // doesn't run
);
```

Promises (9)

- And in the case of a rejection, the second one:

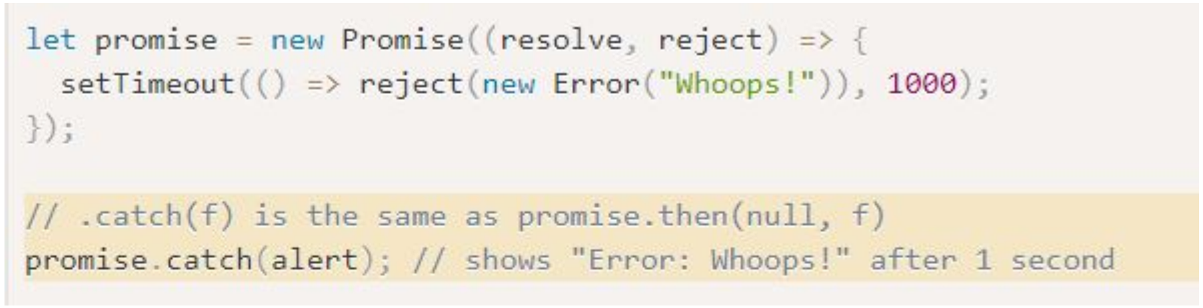
```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});

// reject runs the second function in .then
promise.then(
  result => alert(result), // doesn't run
  error => alert(error) // shows "Error: Whoops!" after 1 second
);
```

- If we're interested only in successful completions, then we can provide only one function argument to .**then**:

```
let promise = new Promise(resolve => {
  setTimeout(() => resolve("done!"), 1000);
});

promise.then(alert); // shows "done!" after 1 second
```

Promises (10)

- If we’re interested only in errors, then we can use null as the first argument: **.then(null, errorHandlingFunction)**.
- Or we can use **.catch(errorHandlingFunction)**



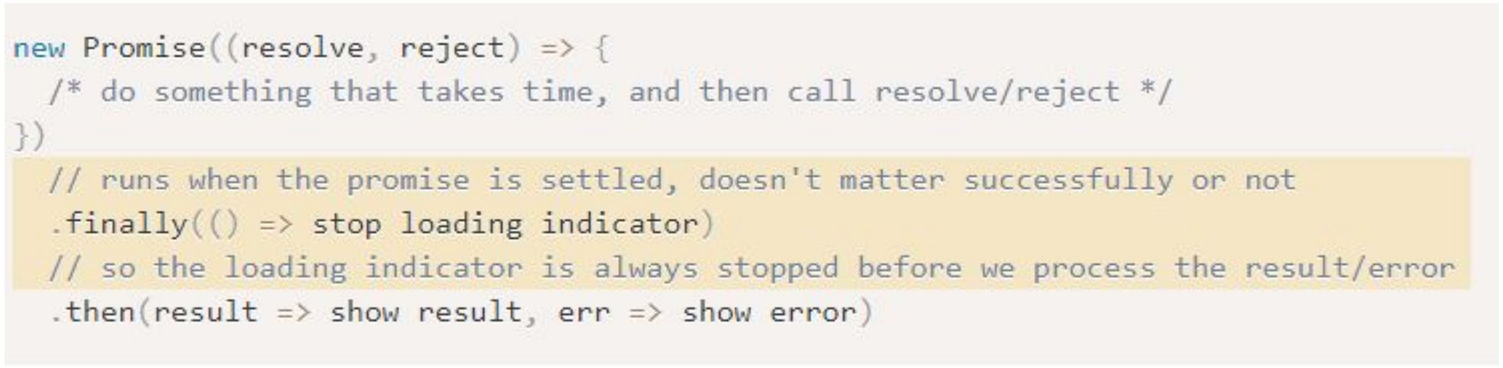
```
let promise = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});

// .catch(f) is the same as promise.then(null, f)
promise.catch(alert); // shows "Error: Whoops!" after 1 second
```

- The call **.catch(f)** is a complete analog of **.then(null, f)**, it’s just a *shorthand*.

Promises (11)

- The call **.finally(f)** is similar to **.then(f, f)** in the sense that f always runs when the promise is settled: be it **resolve** or **reject**.

- **finally** is a good handler for performing cleanup, e.g. stopping our loading indicators, as they are not needed anymore, no matter what the outcome is.



```
new Promise((resolve, reject) => {
  /* do something that takes time, and then call resolve/reject */
})
  // runs when the promise is settled, doesn't matter successfully or not
  .finally(() => stop loading indicator)
  // so the loading indicator is always stopped before we process the result/error
  .then(result => show result, err => show error)
```

Promises (12)

- That said, **finally(f)** isn’t exactly an alias of **then(f,f)** though. There are few subtle differences:
  - A **finally** handler has no arguments. In **finally** we don’t know whether the promise is successful or not. That’s all right, as our task is usually to perform “general” finalizing procedures.
  - A **finally** handler passes through results and errors to the next handler.
- For instance, here the result is passed through **finally** to **then**:

```
new Promise((resolve, reject) => {
  setTimeout(() => resolve("result"), 2000)
})
  .finally(() => alert("Promise ready"))
  .then(result => alert(result)); // <-- .then handles the result
```

- And here there’s an error in the promise, passed through **finally** to **catch**:

```
new Promise((resolve, reject) => {
  throw new Error("error");
})
  .finally(() => alert("Promise ready"))
  .catch(err => alert(err));  // <-- .catch handles the error object
```

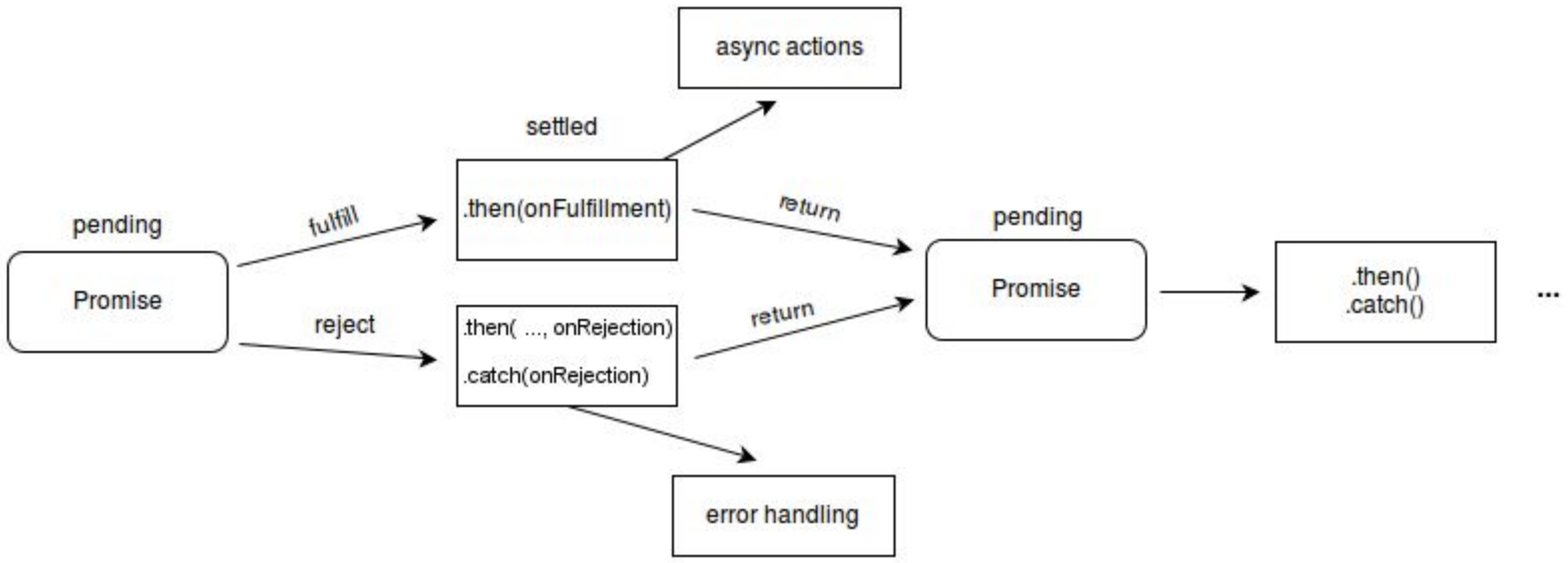
Promises (13)

- .**catch** handles errors in promises of all kinds: be it a **reject()** call, or an error thrown in a handler.

- We should place .**catch** exactly in places where we want to handle errors and know how to handle them. The handler should analyze errors (custom error classes help) and rethrow unknown ones (maybe they are programming mistakes).

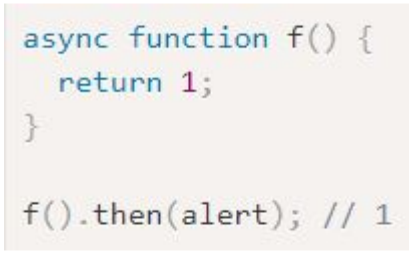
- It's ok not to use .**catch** at all, if there's no way to recover from an error.

Promises (14)


async actions
settled
pending
Promise
fulfill
.then(onFulfillment)
return
pending
Promise
.then()
.catch()
...
reject
.then(..., onRejection)
.catch(onRejection)
return
error handling

Async/await

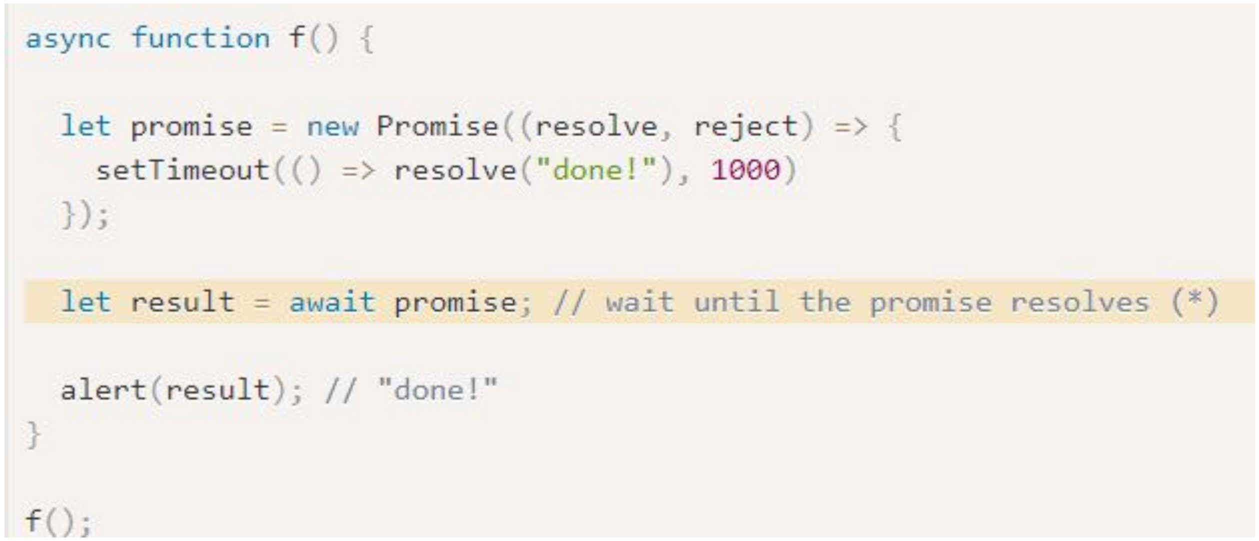
- There’s a special syntax to work with promises in a more comfortable fashion, called “async/await”. It’s surprisingly easy to understand and use.
- The word “async” before a function means one simple thing: a function always returns a promise. Other values are wrapped in a resolved promise automatically.
- For instance, this function returns a resolved **promise** with the result of **1**:



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

f().then(alert); // 1
```

Async/await (2)

- There’s another keyword, **await**, that works only inside **async** functions
- The keyword **await** makes JavaScript *wait* until that **promise** settles and returns its result.



```
async function f() {

  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("done!"), 1000)
  });

  let result = await promise; // wait until the promise resolves (*)

  alert(result); // "done!"
}

f();
```

Fetch

- As you probably already now, JavaScript can send network requests to the server and load new information whenever it’s needed.

- For example, we can use a network request to:
  - Submit an order
  - Load user information
  - Receive latest updates from the server
  - …etc.

- …And all of that without reloading the page!

Fetch (2)

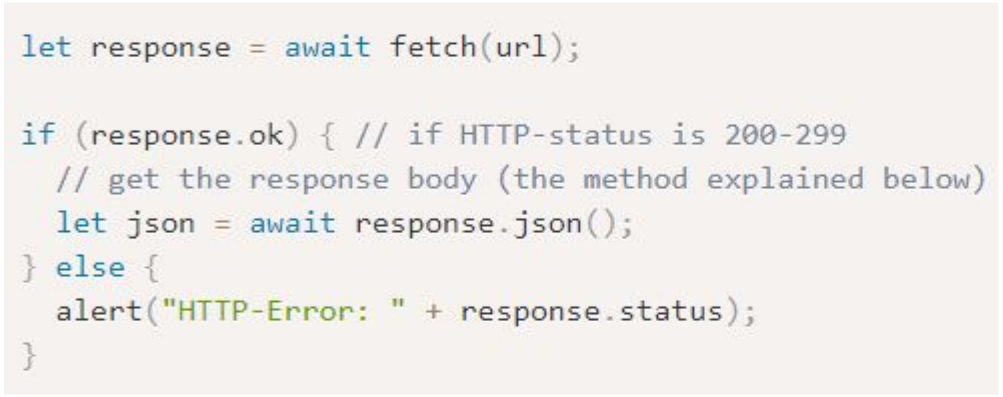
- The **fetch()** method is modern and versatile, so we'll start with it. It's not supported by old browsers, but very well supported among the modern ones.

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

- Getting a response is usually a two-stage process.
- First, the **promise**, returned by **fetch**, resolves with an object of the built-in **Response** class as soon as the server responds with headers.
- The promise rejects if the **fetch** was unable to make HTTP-request, e.g. network problems, or there's no such site. Abnormal HTTP-statuses, such as 404 or 500 do not cause an error.
- We can see HTTP-status in response properties:
  - **status** – HTTP status code, e.g. 200.
  - **ok** – boolean, true if the HTTP status code is 200-299.

Fetch (3)

- For example



```
let response = await fetch(url);

if (response.ok) { // if HTTP-status is 200-299
  // get the response body (the method explained below)
  let json = await response.json();
} else {
  alert("HTTP-Error: " + response.status);
}
```

Fetch (4)

- Second, to get the response body, we need to use an additional method call.

- **Response** provides multiple promise-based methods to access the body in various formats:
  - **response.text()** – read the response and return as text
  - **response.json()** – parse the response as JSON
  - **response.formData()** – return the response as **FormData** object
  - **response.blob()** – return the response as **Blob** (binary data with type)
  - **response.arrayBuffer()** - return the response as **ArrayBuffer**
  - additionally, response.body is a **ReadableStream** object, it allows you to read the body chunk-by-chunk

Fetch (5)

- For instance, let's get a JSON-object with latest commits from GitHub (from javascript.info):

```
let url = 'https://api.github.com/repos/javascript-tutorial/en.javascript.info/commits';
let response = await fetch(url);

let commits = await response.json(); // read response body and parse as JSON

alert(commits[0].author.login);
```

- Or, the same without **await**, using pure promises syntax:

```
fetch('https://api.github.com/repos/javascript-tutorial/en.javascript.info/commits')
  .then(response => response.json())
  .then(commits => alert(commits[0].author.login));
```

- To get the response text, await **response.text()** instead of **.json()**:

```
let response = await fetch('https://api.github.com/repos/javascript-tutorial/en.javascript.info/commits');

let text = await response.text(); // read response body as text

alert(text.slice(0, 80) + '...');
```

Fetch (6)

- The response headers are available in a Map-like headers object in **response.headers**

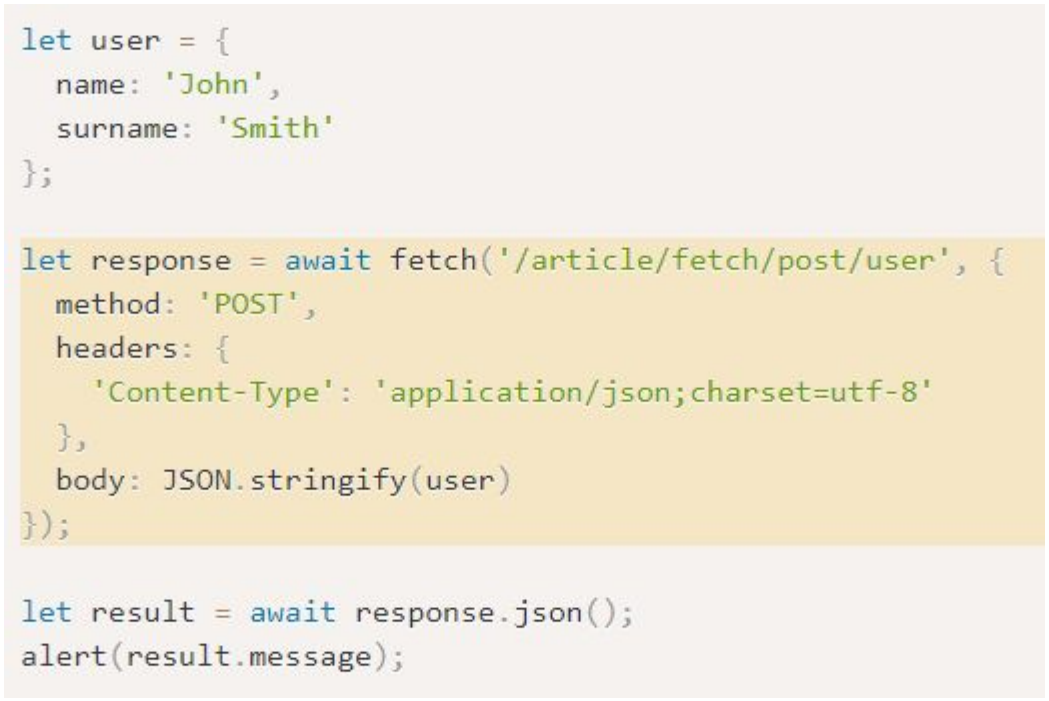
```
let response = await fetch('https://api.github.com/repos/javascript-tutorial/en.javascript.info/commits');

// get one header
alert(response.headers.get('Content-Type')); // application/json; charset=utf-8

// iterate over all headers
for (let [key, value] of response.headers) {
  alert(`${key} = ${value}`);
}
```

- To set a request header in **fetch**, we can use the **headers** option. It has an object with outgoing headers

```
let response = fetch(protectedUrl, {
  headers: {
    Authentication: 'secret'
  }
});
```

Fetch (7)

- To make a **POST** request, or a request with another method, we need to use **fetch** option
  - **method –** HTTP-method, e.g. **POST**
  - **body** – which is the request body. Can be a string (e.g. JSON-econded), form data, blob
- The JSON format is used most of the time.



```
let user = {
  name: 'John',
  surname: 'Smith'
};

let response = await fetch('/article/fetch/post/user', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json;charset=utf-8'
  },
  body: JSON.stringify(user)
});

let result = await response.json();
alert(result.message);
```

Strict mode

- For a long time, JavaScript evolved without compatibility issues. New features were added to the language while old functionality didn’t change.
- This was the case until 2009 when ECMAScript 5 (ES5) appeared. It added new features to the language and modified some of the existing ones. To keep the old code working, most such modifications are off by default. You need to explicitly enable them with a special directive: "**use strict**".
- Strict mode makes several changes to normal JavaScript semantics:
  - Eliminates some JavaScript silent errors by changing them to throw errors.
  - Fixes mistakes that make it difficult for JavaScript engines to perform optimizations: strict mode code can sometimes be made to run faster than identical code that's not strict mode.
  - Prohibits some syntax likely to be defined in future versions of ECMAScript.
- The directive looks like a string: **"use strict"** or **'use strict'**. When it is located at the top of a script, the whole script works the “*modern*” way.

```
"use strict";

// this code works the modern way
...
```

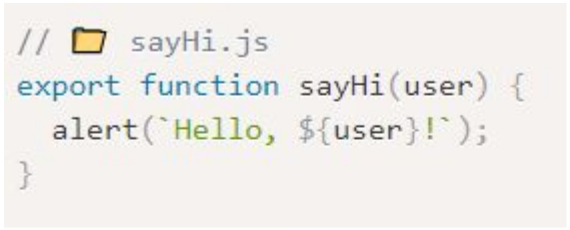
- Modern JavaScript supports “**classes**” and “**modules**” – advanced language structures (we’ll surely get to them), that enable use strict automatically. So we don’t need to add the **"use strict"** directive, if we use them.

Modules

- As our application grows bigger, we want to split it into multiple files, so called “modules”. A module may contain a class or a library of functions for a specific purpose.
- A module is just a file. One script is one module. As simple as that.
- Modules can load each other and use special directives **export** and **import** to interchange functionality, call functions of one module from another one:
  - **export** keyword labels variables and functions that should be accessible from outside the current module.
  - **import** allows the import of functionality from other modules.

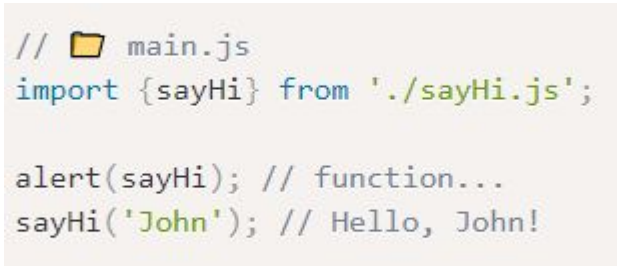
Modules (2)

- For instance, if we have a file sayHi.js exporting a function:



```
// 📁 sayHi.js
export function sayHi(user) {
  alert(`Hello, ${user}!`);
}
```

- …Then another file may import and use it:



```
// 📁 main.js
import {sayHi} from './sayHi.js';

alert(sayHi); // function...
sayHi('John'); // Hello, John!
```

- The **import** directive loads the module by path **./sayHi.js** relative to the current file, and assigns exported function **sayHi** to the corresponding variable.

Modules (3)

- Modules always **use strict**, by default. E.g. assigning to an undeclared variable will give an error.

```
<script type="module">
  a = 5; // error
</script>
```

- Each module has its own top-level scope. In other words, top-level variables and functions from a module are not seen in other scripts.
- In the example below, two scripts are imported, and hello.js tries to use user variable declared in user.js, and fails:

index.html

```
<!doctype html>
<script type="module" src="user.js"></script>
<script type="module" src="hello.js"></script>
```

user.js

```
let user = "John";
```

hello.js

```
alert(user); // no such variable (each module has independent variables)
```

Modules (4)

- Modules are expected to export what they want to be accessible from outside and import what they need.
- So, we should import **user.js** into **hello.js** and get the required functionality from it instead of relying on global variables.
- Or, to correct the previous example:

user.js

```
export let user = "John";
```

index.html

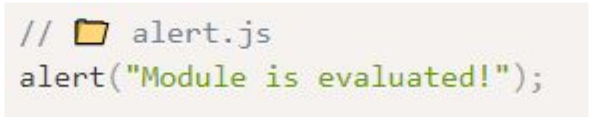
```
<!doctype html>
<script type="module" src="hello.js"></script>
```

hello.js

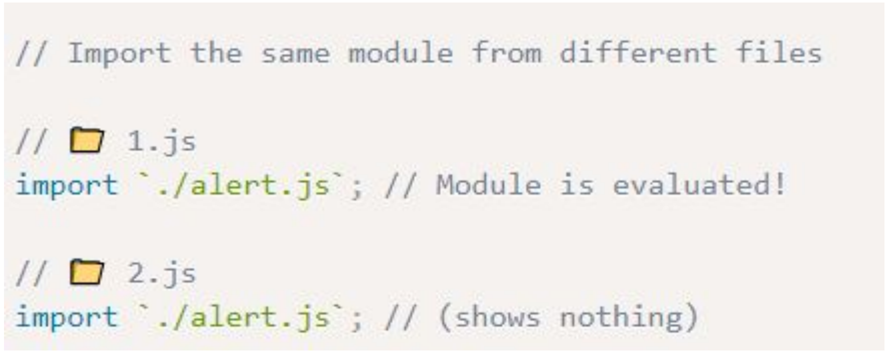
```
import {user} from './user.js';

document.body.innerHTML = user; // John
```

Modules (5)

- If the same module is imported into multiple other places, its code is executed only the first time, then exports are given to all importers.



```
// 📁 alert.js
alert("Module is evaluated!");
```



```
// Import the same module from different files

// 📁 1.js
import `./alert.js`; // Module is evaluated!

// 📁 2.js
import `./alert.js`; // (shows nothing)
```

Modules (6)

- In practice, top-level module code is mostly used for initialization, creation of internal data structures, and if we want something to be reusable – export it.

```
// 📁 admin.js
export let admin = {
  name: "John"
};
```

- If this module is imported from multiple files, the module is only evaluated the first time, **admin** object is created, and then passed to all further importers.
- All importers get exactly the one and only **admin** object:

```
// 📁 1.js
import {admin} from './admin.js';
admin.name = "Pete";

// 📁 2.js
import {admin} from './admin.js';
alert(admin.name); // Pete

// Both 1.js and 2.js imported the same object
// Changes made in 1.js are visible in 2.js
```

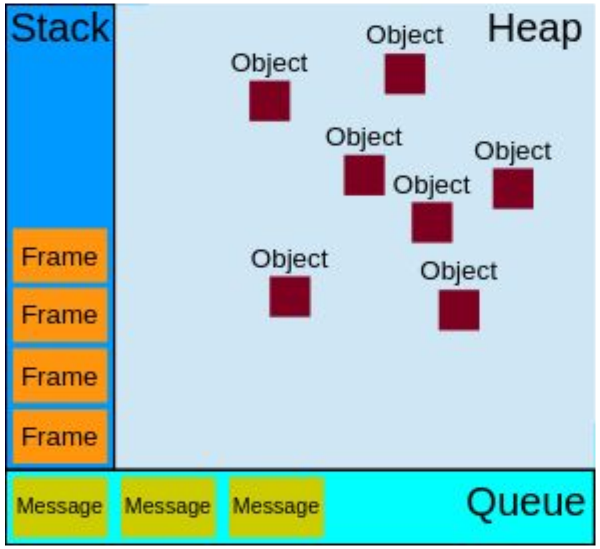
Hoisting

- Hoisting allows you to declare variables and functions after their assignment
- Enables you to use a variable or function before declaring it
- You can only take advantage of this using the function and var keywords
- If you use const or let, the interpreter will not hoist the variables or functions you declare.

```
var coffeeBlend;
console.log(coffeeBlend); // Output: undefined
coffeeBlend = "Dark Roast";


brewCoffee();
function brewCoffee() {
console.log("Brewing a fresh pot of coffee!");
}
// Output: "Brewing a fresh pot of coffee!"
```

Event Loop

- Stack - Function calls form a stack of frames.
- Heap - Objects are allocated in a heap which is just a name to denote a large (mostly unstructured) region of memory.
- Queue - A JavaScript runtime uses a message queue, which is a list of messages to be processed. Each message has an associated function that gets called to handle the message.



Event Loop

- Fundamental core mechanism enabling asynchronous programming
- "Event loop" continuously monitors the call stack and the message queue data structures
- Call stack knows about functions that are currently being executed
- The message queue holds a list of messages (events)
- If the call stack is empty, the event loop checks the message queue for any pending messages

```
while (queue.waitForMessage()) {
  queue.processNextMessage();
}
```

Event Loop - Example

```
const seconds = new Date().getTime() / 1000;

setTimeout(() => {
  // prints out "2", meaning that the callback is not called immediately after 500 milliseconds.
  console.log(`Ran after ${new Date().getTime() / 1000 - seconds} seconds`);
}, 500);

while (true) {
  if (new Date().getTime() / 1000 - seconds >= 2) {
    console.log("Good, looped for 2 seconds");
    break;
  }
}
```

```
console.log('start');

setTimeout(() => {console.log('setTimeout');}, 0);

console.log("middle");

Promise.resolve().then(() => {
  console.log('Promise');
});

console.log('end');
```

Destructuring

- Destructuring allows extracting and assigning values from objects and arrays into variables in a very concise and readable way
- Destructuring Arrays

```
const numbers = [1, 2, 3, 4, 5];

const [first, second, ...rest] = numbers;

console.log(first); // Output: 1
console.log(second); // Output: 2
console.log(rest); // Output: [3, 4, 5]
```

- Destructuring Objects

```
const person = {
  name: 'Paul Knulst',
  role: 'Tech Lead',
  address: {
    street: 'Blogstreet',
    city: 'Anytown',
    country: 'Germany'
  }
};

const {name, role, address: {city, ...address}} = person;

console.log(name); // Output: Paul Knulst
console.log(role); // Output: Tech Lead
console.log(city); // Output: Anytown
console.log(address); // Output: {street: 'Blogstreet', country: 'Germany'}
```