TWeb

Asynchronous Web Applications

Bertil Chapuis

Overview of Today's Class

- Quiz about last week's lecture
- JSON
- Asynchronous Programming
- Network Programming (Browser)
- Network Programming (Server)
- Introduction of next week's assignment
- Correction of last week's assignment

Quiz

Speakup

You can answer to the following Quiz on Speakup.

http://www.speakup.info/

Room Number: **XXXXX**

Once connected, answer to the first test question.

❓ Question 1

Quelle est la valeur retournée par la regex suivante?

```
/ab+c*d/.test("acd");
```

- `true`
- ***`false`***
- `"acd"`
- `undefined`
- `null`
- Aucune réponse correcte

❓ Question 2

Quelle est la valeur retournée par la regex suivante?

```
/[a-z]*/i.test("Hello");
```

- ***true***
- `false`
- `"hello"`
- `undefined`
- `null`
- Aucune réponse correcte

❓ Question 3

Quelles sont les valeurs de groupe extraites avec la regex suivante?

```
"ABC D E".matchAll(/([A-Z])/g);
```

- A
- ABC, D, E
- ***A, B, C, D, E***
- A, B, C
- D, E
- Aucune réponse correcte

Question 4

Etant donné le formulaire suivant hébergé sur l'URL http://www.example.com/, quel est l'URL résultant d'un clique sur le bouton "Send"?

```
<form method="GET" action="send.html">
    <input type="text" name="firstname" value="John" />
    <input type="text" name="lastname" placeholder="Doe" />
    <input type="submit" value="Send" />
</form>
```

- `http://www.example.com/send.html`
- ***`http://www.example.com/send.html?firstname=John&lastname=`***
- `http://www.example.com/submit.html`
- `http://www.example.com/submit.html?firstname=John&lastname=Doe`
- Aucune réponse correcte

Questions ?


JS
JSON

JavaScript Object Notation (JSON) *

JavaScript Object Notation (JSON) is:

- a standard text-based format for representing structured data based on JavaScript object syntax
- used for serializing objects, arrays, numbers, strings, booleans, and null
- based upon JavaScript syntax but is distinct from it: some JavaScript is not JSON

```
{
    "firstname": "John",
    "lastname": "Doe",
    "age": 28,
    "interests": ["ski", "bike"]
}
```

JSON objects are commonly used for transmitting data in web applications. They can be stored in text files with a `.json` extension an `application/json` MIME type.

* https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON

JavaScript to JSON with Stringify *

The JSON global object allows to stringify javascript objects in JSON:

```
// A JavaScript Object
var person = {
    firstname: "John",
    lastname: "Doe",
    age: 28,
    interests: ["ski", "bike"]
};

// And its JSON representation
JSON.stringify(person);
```

* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON

JSON to JavaScript with Parse *

The JSON global object allows to parse JSON strings into JavaScript objects:

```
// A JSON object
var person = '{"firstname":"John","lastname":"Doe","age":28,"interests":["ski","bike"]}';

// And its Javascript representation
JSON.parse(person);
```

* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON

✋ JSON is NOT JavaScript! *

Why does the following snippet results in a SyntaxError?

```
// A JSON Object?
var person = '{firstname:"John",lastname:"Doe",age:28,interests:["ski","bike"]}';

// Uncaught SyntaxError: Unexpected token f in JSON at position 1
JSON.parse(person);
```

Why is the following snippet considered harmful?

```
// DON'T DO THIS!!!
eval('var person = {firstname:"John",lastname:"Doe",age:28,interests:["ski","bike"]}');
```

* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON

Asynchronous Programming

Synchronous vs Asynchronous Programming *

In a **synchronous programming model**, things happen one at a time. When you call a function that performs a long-running action, it returns only when the action has finished and it can return the result. This stops your program for the time the action takes.

An **asynchronous model** allows multiple things to happen at the same time. When you start an action, your program continues to run. When the action finishes, the program is informed and gets access to the result (for example, the data read from disk).

One approach to **asynchronous programming** is to make functions that perform a slow action take an extra argument, a callback function. The action is started, and when it finishes, the **callback** function is called with the result.

* Eloquent Javascript - https://eloquentjavascript.net/11_async.html#h_HH3wvnWMnd

The Event Loop *

Callbacks are not necessarily called by the code that scheduled them.

A JavaScript runtime uses a **message queue**, which is a list of messages to be processed by the **event loop**.

```
// The Event Loop
while (queue.waitForMessage()) {
  queue.processNextMessage();
}
```

Each message has an **associated function** (the callback) which gets called in order to handle the message.

In browsers, `setTimeout`, `setInterval`, event listeners and HTTP requests, typically append messages to the **message queue** of the runtime.

```
setTimeout(function() {
    console.log("called later by the event loop")
}, 1000);

console.log("called by the main program");
```

* https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop

JS Callback Hell *

Asynchronous programming with callbacks can result in a so-called **callback hell**. The introduction of **asynchronous programming** constructs in the JavaScript language (such as `Promise`, `async`, `await`) help at adressing this issue.

```
function countDown(arg, callback) {
    console.log(arg);
    setTimeout(callback, 1000);
}

countDown("five...", function() {
    countDown("four...", function() {
        countDown("three...", function() {
            countDown("two...", function() {
                countDown("one...", function() {
                    console.log("fire!!!");
                });
            });
        });
    });
});
```

JS Promise*

A `Promise` represents the eventual completion (or failure) of an **asynchronous** operation, and its resulting value. In other words, a `Promise` is a proxy for a value that is not necessarily known when the promise is created.

```
var promise = Promise.resolve("Hello, World!");
```

A `Promise` is in one of these states:

- `pending`: initial state, neither fulfilled nor rejected.
- `resolved`: meaning that the operation completed successfully.
- `rejected`: meaning that the operation failed.

A promise is said to be `settled` if it is either `resolve` or `rejected`.

* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

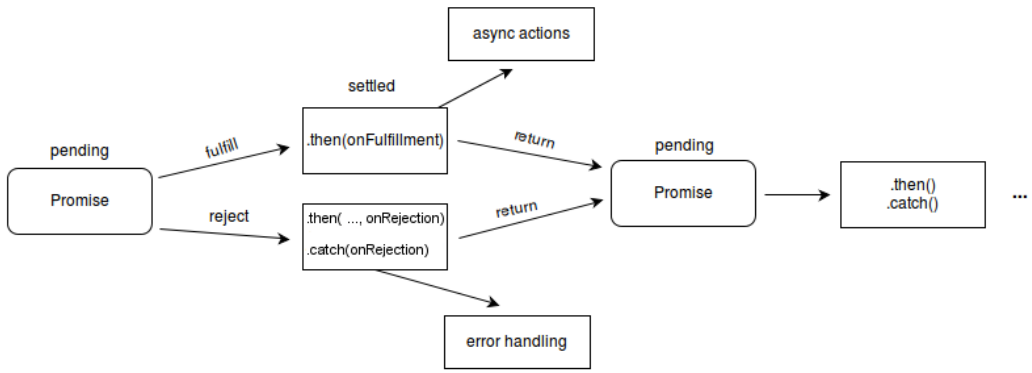
JS Promise*

The constructor of a promise receives an `executor`, a function that is passed with the arguments `resolve` and `reject`. `resolve` and `reject` are functions that can be used to `settle` the promise.

```
var promise = new Promise(function(resolve, reject) {
  setTimeout(function() {
    if (Math.random() > 0.5) {
        resolve(42);
    } else {
        reject("The ultimate question to life, the universe and everything has no answer!")
    }
  }, 1000);
});
```

Promise*

As the `Promise.prototype.then()` and `Promise.prototype.catch()` methods return promises, they can be chained.



The `then` method appends a fullfilment and possibly a rejection handler and returns a new promise.

```
promise.then(value => console.log(value), reason => console.error(reason));
```

The `catch` method appends a rejection handler and returns a new promise.

```
promise.then(value => console.log(value)).catch(reason => console.error(reason));
```

* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

JS Promise Helpers *

The `Promise` object comes with usefull helper methods:

- `Promise.all(iterable)` waits for all promises to be resolved, or for any to be rejected.
- `Promise.allSettled(iterable)` waits until all promises have settled (each may resolve, or reject).
- `Promise.race(iterable)` waits until any of the promises is resolved or rejected.
- `Promise.resolve(value)` and `Promise.reject(reason)` return settled promises.

* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

JS Async Function *

An **asynchronous function** is a function which operates asynchronously via the **event loop**, using an implicit `Promise` to return its result. The resulting code feels much more like standard synchronous functions.

```
function deepThought() {
    return new Promise(function(resolve, reject) {
        setTimeout(function() {
            if (Math.random() > 0.5) resolve(42);
            else reject("The ultimate question to life, the universe and everything has no answer!");
        }, 1000);
    });
}
async function asyncFunction() {
    return await deepThought();
}
var promise = asyncFunction();
```

The `await` expression pauses the execution of the `async` function and waits for the resolution of the `Promise`, and then resumes the `async` function's execution and evaluates as the resolved value.

The `await` keyword is only valid inside `async` functions

* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function

Hands on! *

Rewrite the `countDown` function with a Promise.

- Use the `then` method to perform the count down.
- Use the `await` keyword to perform the count down.

Network Programming (Browser)

XMLHTTPRequest *

`XMLHTTPRequest` is used to interact with servers without having to do a full page refresh.

```
var url = 'https://api.github.com/users/tweb-classroom';
var request = new XMLHttpRequest();
request.open('GET', url);
request.responseType = 'json';
request.onload = function() {
    console.log(request.getAllResponseHeaders());
    console.log(request.status);
    console.log(request.response);
}
request.send();
```

By default, `XMLHTTPRequest` loads content **asynchronously** and uses a `callback` mechanism. Other callbacks include `onerror`, `ontimeout` and `onprogress`.

Asynchronous calls occur in the background and do not load a new page, nor affect what the user is doing. For instance, it enables a Web page to update just part of its content.

Despite its name, `XMLHTTPRequest` is not restricted to XML and can be used with JSON and other formats.

* https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON

JS Fetch API *

The Fetch API provides an interface for fetching resources across the network. It provides a more powerful and flexible feature set than `XMLHttpRequest`.

By default, `fetch` performs GET requests and returns a `Promise`.

```
var promise = fetch('https://api.github.com/users/tweb-classroom');
```

Additional parameters enables to change the method (HEAD, POST, PUT, etc.) and add headers.

```
var promise = fetch("https://api.github.com/users/tweb-classroom", {
    method: `POST`,
    headers: {'Content-Type': 'application/json'},
    body: JSON.stringify({'value': 'Hello, World!' })
});
```

* https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

Network Programming (Server)

NodeJS HTTP client

NodeJS provides an HTTP client that relies on a callback mechanism.

```
const https = require('https');

const req = https.get('https://api.github.com/users/tweb-classroom',
    { headers: { 'User-Agent': 'NodeJS' } },
    res => {
        var body = '';
        res.on('data', chunk => {
            body += chunk;
        })
        res.on('end', function () {
            console.log(body);
        });
    });

req.end()
```

* https://nodejs.dev/making-http-requests-with-nodejs

NodeJS Fetch

A server side implementation of Fetch can be installed as a module.

```
npm install node-fetch --save
```

Notice that babel is required to enable `async` and `await`.

* https://www.npmjs.com/package/node-fetch

Correction

Ex5: Express and Form Validation