Server-side
Web Development

Unit 14b. Javascript review

Fidel Oltra, Ricardo Sánchez

IES Jaume II El Just
Tavernes de la Valldigna
Departament d'Informàtica
Curs 2024-25

Index

1 Javascript review

To successfully complete this part of the course, the student must already have prior knowledge of Javascript, especially regarding arrow functions, callbacks and promises. These notes serve as a reminder or review of these concepts.

1.1 Functions and arrow functions

Let's now see the different ways of defining functions in JavaScript. We will introduce a concept that has become very common, and that we will use very often. It is an alternative notation to define methods or functions, the so-called **arrow** or **lambda** functions.

1.1.1 Traditional functions

Let's see a traditional function with its use:

```
function add(num1, num2) {
    return num1 + num2;
}

// Using the function
console.log(add(3, 2)); // Displays 5
```

1.1.2 Anonymous Functions

This same function can also be expressed as an **anonymous function**. These functions are declared "on the fly" and are usually assigned to a variable for later naming or calling:

```
let addAnonymous = function(num1, num2) {
    return num1 + num2;
};
console.log(addAnonymous(3, 2));
```

1.1.3 Arrow Functions

Arrow functions provide a way to define functions using a *lambda expression* to specify parameters in parentheses and the function code in curly braces, separated by an arrow. The `function` keyword is omitted when defining them.

The previous function, expressed as an arrow function, would look like this:

```
let add = (num1, num2) => {
    return num1 + num2;
};
```

Similar to anonymous functions, their value can be assigned to a variable for later use or defined on the spot in a specific code snippet.

In fact, the above code can be further simplified: in cases where the function simply returns a value, the curly braces and the 'return' keyword can be omitted:

```
let add = (num1, num2) => num1 + num2;
```

Additionally, if the function has a single parameter, the parentheses can be omitted. For example, the following function returns twice the number it receives as a parameter:

```
let double = num => 2 * num;
console.log(double(3)); // Displays 6
```

1.1.4 Direct Use of Arrow Functions

As mentioned before, arrow functions, like anonymous functions, have the advantage of being used directly where they are needed. For example, given the following list of personal data:

```
let data = [
    {name: "John", phone: "966343434", age: 40},
    {name: "Anne", phone: "9112565656", age: 55},
    {name: "Marcus", phone: "612998877", age: 13},
    {name: "Mary", phone: "611664366", age: 17}
];
```

If we want to filter people who are of legal age, we can do it with an anonymous function combined with the 'filter' function:

```
let adults = data.filter(function(person) {
    return person.age >= 18;
});
console.log(adults);
```

We can also use an arrow function instead:

```
let adults = data.filter(person => person.age >= 18);
console.log(adults);
```

Note that we don't assign the function to a variable for later use; instead, they are used at the same point where they are defined. The code becomes more concise using an arrow function.

The difference between arrow functions and traditional notation or anonymous functions is that with arrow functions, we cannot access the `this` or `arguments` elements, which are available in anonymous or traditional functions. Therefore, if it is necessary to do so, we should opt for a normal or anonymous function in such cases.

1.2 Callbacks

One of the two pillars on which asynchronous programming in JavaScript is based is callbacks. A **callback** is a function, A, passed as a parameter to another function, B, which will be called at some point during the execution of B (usually when B completes its task). This concept is fundamental for giving Node.js and JavaScript asynchronous behavior: a function is called, and it is instructed on what to do when it finishes, allowing the program to focus on other tasks in the meantime.

An example is the `setTimeout` function in JavaScript. This function can be given a function to call and a time (in milliseconds) to wait before calling it. After executing the `setTimeout` line, the program continues, and when the time expires, the specified callback function is called.

Let's write this example in a file called 'callback.js':

```
setTimeout(function() {console.log("Callback finished");}, 2000);
console.log("Hello");
```

If we run the example, we will see that the first message displayed is "Hello" and, after two seconds, the "Callback finished" message appears. That is, we called `setTimeout`, the program continued, wrote "Hello" on the screen, and once the specified time had passed, the callback was called to do its job.

We will extensively use callbacks throughout this course, especially to process the result of some promises that we will use or the handling of some service requests.

1.3 Promises

Promises are another important mechanism for introducing asynchronous behavior into JavaScript. They are used to define the completion (successful or not) of an asynchronous operation. In our code, we can define promises to perform asynchronous operations or (more commonly) use promises defined by others in the use of their libraries.

Throughout this course, we will use promises to, for example, send operations to a database and collect the results when they finish, without blocking the main program. However, to better understand what we will do, it is useful to have a clear understanding of the structure of a promise and the possible responses it offers.

1.3.1 Creating a Promise. Elements to Consider

In the case that we want to create a promise, we will create an object of type `Promise`. To this object is passed a function with two parameters:

- The *callback* function to call if everything goes well.
- The *callback* function to call if there has been any error.

These two parameters are usually called 'resolve' and 'reject,' respectively. So, a basic promise skeleton, using an arrow function to define the function to execute, would be:

```
let variableName = new Promise((resolve, reject) => {
    // Code to execute
    // If everything is OK, call "resolve"
    // If something fails, call "reject"
});
```

Internally, the function will do its job and call its two parameters in any case. In the case of `resolve`, the result of the operation is usually passed as a parameter, and in the case of `reject`, the produced error is usually passed.

Let's see an example. The following promise looks for adults in the list of people seen in a previous example. If results are found, they are returned with the 'resolve' function. Otherwise, an error is generated and sent with 'reject':

```
let data = [
    {name: "John", phone: "966343434", age: 40},
    {name: "Anne", phone: "9112565656", age: 55},
```

```
    {name: "Marcus", phone: "612998877", age: 13},
    {name: "Mary", phone: "611664366", age: 17}
];

let adultsPromise = new Promise((resolve, reject) => {
    let result = data.filter(person => person.age >= 18);
    if (result.length > 0)
        resolve(result);
    else
        reject("No results found");
});

console.log(adultsPromise);
```

The function that defines the promise could also be defined in this other way:

```
let adultsPromise = list => {
    return new Promise((resolve, reject) => {
        let result = list.filter(person => person.age >= 18);
        if (result.length > 0)
            resolve(result);
        else
            reject("No results found");
    });
};
```

This way, we avoid using global variables, and the array is passed as a parameter to the function itself, which returns the `Promise` object once it is concluded.

1.3.2 Consuming Promises

In the case of wanting to use a previously defined promise (or created by others in some library), we simply call the function or object that triggers the promise and collect the result. In this case:

- To collect a successful result ('resolve'), we use the 'then' clause.
- To collect an erroneous result ('reject'), we use the 'catch' clause.

Thus, the previous promise can be used in this way (again, using arrow functions to process the 'then' clause with its result or the 'catch' with its error):

```
adultsPromise(data).then(result => {
    // If we enter here, the promise has been processed successfully
    // In "result," we can access the obtained result
    console.log("Matches found:");
    console.log(result);
}).catch(error => {
    // If we enter here, there has been an error processing the promise
    // We can check "error" for details
    console.log("Error:", error);
});
```

Copy this code below the previous code in the 'promise_test.js' file created earlier to check the functionality and what the promise shows.

Note that, when defining the promise, the structure of the result or the error is also defined. In this case, the result is a vector of people matching the search criteria, and the error is a text string. However, they can be of any data type.

1.3.3 The async/await Specification

Since ECMAScript 7, a new way of working with promises is available through the async/await specification. It is a more convenient way to call asynchronous functions and collect their result before calling another, without the need to nest 'then' clauses to link the result of one promise with the next.

We will not go into details on how to use it for now. We will do so later, when we are familiar with promises.