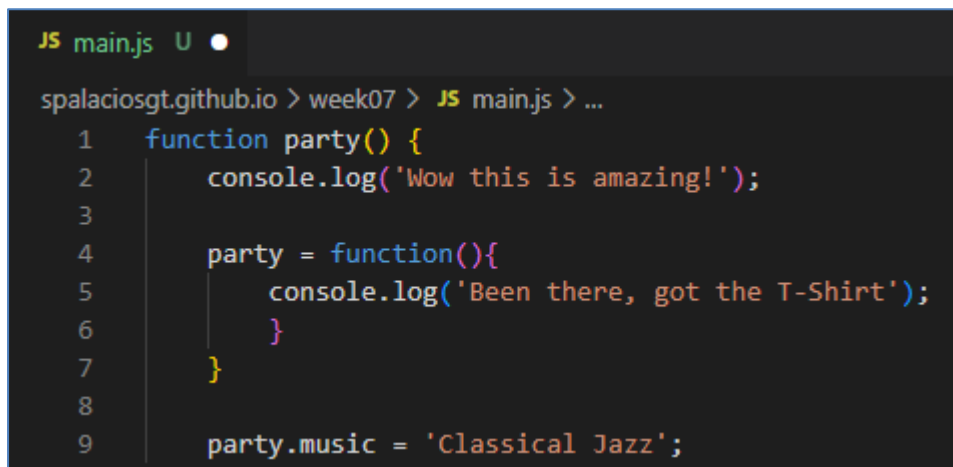


Readings Assignment Notes

1. Ch11 Further Functions

- ✓ Functions are instances or objects of **first-class object**. This means that like any other object, a function can be passed as a parameter, and that is a principle for executing lambda functions, literals, callbacks and doing many more advanced things in JavaScript.
- ✓ Functions in JS have **many properties and methods**, such as the "length" property that returns the number of declared parameters, the reference to the "this" object, the use of cache through the "memoization" property, and the ability to declare and execute a function instantly (immediately invoked function expressions).
- ✓ The scope of the functions are **local code blocks**, which allow the use of variables and constants, which are temporary, and allow any type of initialization and reservation of spaces in memory that are worked when they are executed.
- ✓ **Versatility**, the fact that JavaScript is not typed, allows a lot of flexibility, for example a function can be redefined at run time, that is, a function can be defined and then its code and signature change even properties, this is something impossible in typed languages.



```
JS main.js U ●
spalaciosgt.github.io > week07 > JS main.js > ...
1  function party() {
2      console.log('Wow this is amazing!');
3
4      party = function(){
5          console.log('Been there, got the T-Shirt');
6      }
7  }
8
9  party.music = 'Classical Jazz';
```

```

12
13     function ride(){
14         if (window.unicorn) {
15             ride = function(){
16                 // some code that uses the brand new and sparkly unicorn methods
17                 return 'Riding on a unicorn is the best!';
18             }
19         } else {
20             ride = function(){
21                 // some code that uses the older pony methods
22                 return 'Riding on a pony is still pretty good';
23             }
24         }
25         return ride();
26     }
27

```

- ✓ Like any high-level language, JS also allows you to use **recursion** to take advantage of the computational capacity and speed of a computer. There are problems in which it is easier and more efficient to solve with recursion than with repetition.

```

28
29     function factorial(n) {
30         if (n === 0) {
31             return 1;
32         } else {
33             return n * factorial(n - 1);
34         }
35     }
36

```

- ✓ **Callbacks**, it's a cool thing JS must do to execute asynchronous code. Basically, it is calling a function within another, which will be executed without interrupting the flow of the main function and at the right time. It is ideal for working on topics that are complementary to a transaction, for example, where the user does not have to wait to finish the main flow and then the elements are arranged on the screen with a callback.
- ✓ There are also the **promises objects**, these objects allow to indicate the completion of a callback or asynchronous function. A function of this nature can complete with success or failure, and these objects help us indicate this in JS.

```

62
63     var img1 = document.querySelector('.img-1');
64
65     function loaded() {
66         // woo yey image loaded
67     }
68
69     if (img1.complete) {
70         loaded();
71     }
72     else {
73         img1.addEventListener('load', loaded);
74     }
75
76     img1.addEventListener('error', function() {
77         // argh everything's broken
78     });
79

```

- ✓ Like callbacks, **asynchronous functions** can also be defined per se, that is, it is not necessary to do more than call them so that their execution is in a different thread than the flow of the main function that calls it.

```

80
81
82     async function loadGame(userName) {
83
84         try {
85             const user = await login(userName);
86             const info = await getPlayerInfo (user.id);
87             // load the game using the returned info
88         }
89
90         catch (error){
91             throw error;
92         }
93     }
94

```

- ✓ Again, the **power and flexibility** of JS, not only can you send functions as parameters (like lambda functions), but also a function can return another function. It's an incredible idea, you could even create a function generator that returns the most appropriate behavior that you want to assign to an object or class, for example.

```

95
96     function outer() {
97         const outside = 'Outside!';
98         function inner() {
99             const inside = 'Inside!';
100             console.log(outside);
101             console.log(inside);
102         }
103         return inner;
104     }
105
106     function counter(start){
107         let i = start;
108         return function() {
109             return i++;
110         }
111     }

```

- ✓ **Functional programming** is beautiful, I saw it last year with the CSE121A **Clojure** course, I understood how this paradigm works, and how the fact of not handling mutable data gives great advantages to this type of programming compared to other paradigms; Also, I got used to not mentioning variables and enclosing everything in parentheses, which was great. In JS it can be handled as well, but the fact that JS handles mutable data no longer gives it the same performance and focus as pure functional programming languages. Below an example of Clojure.

```

15 ;; Flavor Lists
16 (def list01 ["Vanilla" "Chocolate" "Cherry Ripple"])
17 (def list02 ["Lemon" "Butterscotch" "Licorice Ripple"])
18
19 ;; Start Function
20 (defn cartesian-product
21   ([ ] '())
22   ([xs & more]
23    (mapcat #(map (partial cons %)
24                  (apply cartesian-product more))
25            xs)))
26
27 ;; Main Function
28 (defn main [ ]
29   (println "")
30   (println "Flavor Lists - Ice Cream Shop")
31
32   ;;
33   ;; Task 1
34   ;;
35
36   ;; Reference 1: How to println a map
37   ;; https://stackoverflow.com/questions/38035839/how-to-print-each-item-of-list-in-separate-l
38   (println "")
39   (println "Task 1: Chocolate Flavor Options")
40   ;; I combine Chocolate flavor as the first with all flavors from list 2

```

2. Ch13 AJAX

- ✓ This topic is interesting, it is the basis of asynchronous requests to server resources that can be made without reloading all the elements of a web page. **AJAX**, in a nutshell, allows you to submit a form via POST or GET to a server-side resource and get the response with the ability to modify only the elements that reflect that result without reloading the entire page, that's AJAX.
- ✓ The web communication model has not changed much in recent years, it is the sending of forms and reception of results in the form of parameters, XML files, streams, that the server sends to the client. In the end it all comes down to clients making requests, and the server giving a response (**Request - Response**).
- ✓ A basic and **simple approach (basic usage)** is to make a request to a resource (first line), then get the response (second line), and if there was an error, report it (third line). We could have a web function on the other side that does a calculation on the database, and get the result and display it to the user, for example

```
113
114 try {
115     fetch('https://example.com/data')
116         .then( Console.log("code that handles the response"));
117         .catch( Console.log("code that runs if the server returns an error"));
118 } catch(Exception e) {
119 }
120
```

- ✓ **Response Interface.** As mentioned, everything starts with a request from the web client. This is sent through a form by a POST or GET to a resource on the server through AJAX, and the server resource responds through a response. There are **many types of response** we can use in JS with **fetch function**, it can be a flat data, file, streaming, etc. Below is an example that brings plain text, then the different types of response are placed.

```
const url = 'https://amm.org.gt/pre-despacth/2022/data';

fetch(url)
    .then((response) => {
    ✓ if(response.ok) {
        return response;
    }
    throw Error(response.statusText);
    })
    .then( response => console.log('do something with response'))
    .catch( error => console.log('There was an error!'))
```

Type	Description
Redirect ()	A promises-type object is obtained which, at the end, redirects to another URL.
Text ()	It's to get plain text as a response, it can be a stream of text or a bunch, it's pure plain text.
File ()	When the response is a binary object such as a file, when a certain request results in a download or byte transfer of a file as response.
JSON ()	This is one of the most used, it is when the response is a file or document in JSON format (Key - Value). The JSON format is a standard today, and JS already has functions that allow converting plain text in JSON format to objects in memory.
Response ()	You can also get a generic "response" type object, which is one of the simplest.

```
const response = new Response( 'Hello!', {
  ok: true,
  status: 200,
  statusText: 'OK',
  type: 'cors',
  url: '/api'
});
```

- ✓ **Request Interface.** Very similar to the response interface, this is done by placing certain data or header parameters, and always through what a web form and a web method do: GET or POST. There are other HTTP methods, but those are the most used. In the following example, a request is made, which could be processed later to see the response through the "fetch" function.

```
const request = new Request('https://www.amm.org.gt/pre-dispatch/2022/data', {
  method: 'GET',
  mode: 'cors',
  redirect: 'follow',
  cache: 'no-cache'
});
```

- ✓ Putting it all together, in this example you can consume a **server-side resource** via the fetch function by making a request and receiving a response.

```

1  <!doctype html>
2  <html lang='en'>
3      <head>
4          <meta charset='utf-8'>
5          <title>Ajax Example</title>
6      </head>
7      <body>
8          <button id='number'>Number Fact</button>
9          <button id='chuck'>Chuck Norris Fact</button>
10         <div id='output'>
11             Ajax response will appear here
12         </div>
13         <script src='main.js'></script>
14     </body>
15 </html>
16

```

```

const url = 'https://example.com/data';
const headers = new Headers({ 'Content-Type': 'text/plain', 'Accept-Charset' : 'utf-8', 'Accept-Encoding': '' });

const request = (url,{
    headers: headers
});

fetch(request)
.then( function(response) {
    if(response.ok) {
        return response;
    }
    throw Error(response.statusText);
})
.then( response => console.log("do something with response"))
.catch( error => console.log('There was an error!') )

```