

# Note Web Application

Brendon Mendicino

March 15, 2023

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Objects . . . . .	4
1.2	Functions . . . . .	5
1.3	Dates . . . . .	6
1.4	Asynchronous Programming . . . . .	6
1.4.1	Database Access (SQLite) . . . . .	7
1.4.2	Await/Async . . . . .	8

## 1 Introduction

JS is backward compatible, to be able to use the previous features is use the directive:

```
1 "use strict";
```

JS has primitive types and non-primitive types, JS is also and strongly typed language, the primitive types are: string, number, boolean, null, undefined. The non-primitive are the objects, which can be: array, function, user-defined.

The all possible false values in JS: 0, -0, NaN, undefined, null, '', in JS there are two main comparison operators:

```
1 a == b    // equal, convert types and compare
2 a === b   // strict equal, inhibits automatic type conversion
```

In JS you can create variable with:

```
1 // modern
2 let a = 10;    // can be changed
3 const b = 'a'; // constant
4
5 // old
6 var k = 9;
7 j = 30;
```

The difference between null and undefined, is that variable with null they old a value which is null, on the other way if a variable is declared and nothing is associated with it the value olds by default undefined.

A scope is defined by a **block**, which is created with ...

There two kinds of **foreach** in JS, using **in** allows iterating over objects, while **of** allows iterating over iterable objects:

```
1 for (let a in object) {
2   ...
3 }
4
5 for (let b of iterable) {
6   ...
7 }
```

Using arrays:

```
1 let a = [1, 2, 'ok', false];
2 let b = Array.of(1, 2, true);
3 a.push(5);    // append an element
4 b.unshift(2); // insert at the beginning
5
6 let copy = Array.from(a); // shallow copy, it does not deep copy
```

The **destructuring assignment** can be done, it extracts the values from the mast left-hand side:

```
1 let [x, y] = [1, 2];
2 [x, y] = [y, x]    // swap
```

The **spread operator** (`...`) expands on iterable object into it's values:

```
1 let [x, ...y] = [1, 2, 3, 4];      // y == [2, 3, 4]
2
3 const a = [1, 2];
4 const b = [0, ...a, 3]; // [0, 1, 2, 3]
```

Spreading can be from the left or from the right, usually the spread operator is used for copying array:

```
1 const a = [1, 2];
2 const b = [...a];
```

A **string** in JS is an immutable type (like python) encoded in Unicode. The **template literals** can be done with the **tick** operator ``` (expression like Kotlin):

```
1 let name = 'Bre';
2 let sur = 'Mend';
3 // Template literal
4 let fullName = `${name} ${sur}`;
```

## 1.1 Objects

JS is **prototype based language**, which means that there are no declarations of classes. In JS property names must be strings and can be modified, the value of the property can be any other type of type or object. To create an object in JS you use curly braces and the defined properties:

```
1 const movie = {
2   title: 'Inception',
3   genre: 'sci-fi',
4   duration: 180
5 }
6
7 console.log(movie)
8 console.log(movie['title'])
9 console.log(movie.title)
```

It is also possible to add a property by simply assigning a new name to a type, it is also possible to delete a property with the keyword **delete**. There are two helper functions:

- `Object.key(object)`: return only the key;
- `Object.entries(object)`: return an array with the key and value;

To copy an object it is possible to use:

```
1 const copied = Object.assign({}, original)
2 const withSpread = {...original} // it also possible to use the spread
   operator
3
```

```
4 // assign can also be used to merge objects
5 const merged = Object.assign({}, copied, {something: 'test'})
```

## 1.2 Functions

In JS functions are objects, so it is possible to assign a function to a property or use it in a parameter in another function. There three possible ways to define a function:

```
1 // 1. Function
2 function do(a, b = 1) {
3   ...
4 }
5
6 // parameters can also have a default value
7 function some(par1, par2, ...variable) { // ... is the 'rest' operator, like
8   ...
9 }
10
11 // 2. Function Expression
12 const fn = function(params) { }
13
14 // 3. Arrow Function
15 const func = (params) => { }
```

In JS **Closure** can be created, with closure it is possible to use parameters of the scope where the function is defined, even if that scope does not exist any more.

```
1 function greeter(name) {
2   const myname = name;
3
4   const hello = () => {
5     return "Hello " + myname;
6   }
7
8   return hello;
9 }
10
11 const helloTest = greeter('test');
12
13 console.log(helloTest()); // 'Hello test'
```

To create an object there are also **constructor functions**:

```
1 function Movie(title, director, duration) {
2   this.title = title;
3   this.director = director;
4   this.duration = duration;
5   this.isLong = () => this.duration > 120;
6 }
7
```

```
8 const movie = new Movie('Inception', 'Nolan', 180);
9 console.log(movie.isLong); // true
```

## 1.3 Dates

We use `dayjs()` objects in JS to build a data, it is an external library. The return of `dayjs()` fetches the time from the locale time, other than that it can create a data from ISO8601 strings, 8 digit dates, etc. To install this library: `$ npm install dayjs`. The string value of the standard format is in ISO9601 in UTC time, it's important to remember that the days and the month inside the object start *counting from 0*. Other than that the library has some methods to compare different `dayjs` objects, also by choosing the level of *granularity* (year, month, day, ...).

```
1 const date = dayjs('2023-03-15');
2 const now = dayjs();
3
4 now.isAfter(date, 'day'); // comparing 'now' and 'date' by day
```

## 1.4 Asynchronous Programming

In JS when passing functions to other functions it's called a **callback**, this functions can be *synchronous* or *asynchronous*.

```
1 function logQuote(quote) {
2   console.log(quote);
3 }
4
5 function createQuote(quote, callback) {
6   const myQuote = 'Like I always say, ${quote}';
7   callback(quote);
8 }
9
10 createQuote('sium', logQuote);
```

In order to have functional features in language there some need properties:

- *functions as first class citizen;*
- *higher-order functions;*
- *function composition;*
- *call chaining;*

In JS arrays have functional methods, for example:

```
1 a.forEach(item => ...); // action on every element of the array
2 a.every(x => x > 10);   // return true if all elements satisfy the condition,
   false otherwise
3 a.some(x => x < 10);    // return true if at least one element satisfy the
   condition
4 a.map(x => `${x}`);     // return a new array with every element mapped to a new
   one
5 a.filter(x => x === 0); // return a new array with all elements that satisfy the
   condition
6 a.reduce((x, y) => x + y, 0); // return a reduced value
```

Even though JS is executed on a single thread it is possible to create concurrent code, for example a function that allows to excute a callback after a certain amount of time is the `setTimeout()` function:

```
1 const f = (task) => {
2   // do something
3 };
4
5 setTimeout(f, 2000, task);
```

This is possible because JS runs in the **Event Loop**, which periodically checks if there are some part of the code that needs to be executed.

There is a function that allows asynchronous callback after a timeout:

```
1 const onesecond = setTimeout(() \implies {
2   console.log('1 second has passed');
3 }, 1000);
```

There is also the `setInterval()` function that periodically runs:

```
1 const period = setInterval(() => {}, 2000);
2 clearInterval(period);
```

### 1.4.1 Database Access (SQLite)

The module for `sqlite3` allows calling sql queries via his APIs, first there needs to be an open with the database, to open a connection use:

```
1 const sqlite = require('sqlite3');
2
3 const db = new sqlite.Database('exams.sqlite',
4   (err) => { if (err) throw err; });
```

Example of query:

```
1 let result = [];
2 let sql = "SELECT * FROM course LEFT JOIN score ON course.code=score.coursecode";
3 db.all(sql, (err, row) => {
4   if (err) throw err;
5   for (let row of rows
6 });
```

The problem with execution this queries is that they are *all asynchronous*, and they can cause race conditions. The solution to this problem are **Promise**, which helps simplifying asynchronous programming. A **Promise** handles a **resolve** and a **reject** which needs to be called when the callback fails or succeeds. The values passed to **resolve** can be accessed by in the **then** method, which gets called when the **Promise** is completed.

```
1 function waitPromise(duration) {
2   return new Promise((resolve, reject) => {
3     if (duration < 0) {
4       reject(new Error('...'));
5     } else {
6       setTimeout(resolve, duration);
7     }
8   })
9 }
10
11 waitPromise(1000).then((result) => {
12   console.log('Success :', result);
13 }).catch((error) => {
14   console.log('Error :', error);
15 });
```

A promise has 3 main methods: **then**, **catch**, **finally**, which are similar behaviour to the try catch block in Java. Promises can also work concurrently with **Promise.all()** or **Promise.race()**.

### 1.4.2 Await/Async

The keywords **async** and **await** allows to convert pieces of code to a **Promise**:

```
1 function resolveAfter2Seconds() {
2   return new Promise(resolve => {
3     setTimeout(() => {
4       resolve('resolved');
5     }, 2000);
6   });
7 }
8
9 async function asyncCall() {
10  console.log('calling');
11  const result = await resolveAfter2Seconds();
12  console.log(result);
13 }
14
15 asyncCall(); // this returns is a promise
```

In fact a function marked as **async** returns a promise.

This method can be combined with the database queries:



```
1  
2 async function main() {  
3 }  
4  
5 main();
```