Note Web Application

Brendon Mendicino

March 14, 2023

Contents

1 Introduction

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

```
"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:

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

In JS you can create variable with:

```
// modern
let a = 10;    // can be changed
const b = 'a'; // constant

// old
var k = 9;
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:

```
for (let a in object) {
  ...
}

for (let b of iterable) {
  ...
}
```

Using arrays:

```
let a = [1, 2, 'ok', false];
let b = Array.of(1, 2, true);
a.push(5);      // append an element
b.unshift(2);   // insert at the beginning

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:

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

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

```
let [x, ...y] = [1, 2, 3, 4];      // y == [2, 3, 4]

const a = [1, 2];
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:

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

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

```
let name = 'Bre';
let sur = 'Mend';
// Template literal
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 and object in JS you use curly braces and the defined properties:

```
const movie = {
  title: 'Inception',
  genre: 'sci-fi',
  duration: 180
}

console.log(movie)
console.log(movie['title'])
console.log(movie.title)
```

It is also possible to add a property by simple 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:

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

```

```
// assign can also be used to merge objects
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. Function
function do(a, b = 1) {
  ...
}

// parameters can also hava a deafult value
function some(par1, par2, ...variable) { // ... is the 'rest' operator, like
    varargs, rest parameters can be iterated
  ...
}

// 2. Function Expression
const fn = function(params) { }

// 3. Arrow Function
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.

```
function greeter(name) {
  const myname = name;

  const hello = () => {
    return "Hello " + myname;
  }

  return hello;
}

const helloTest = greeter('test');

console.log(helloTest());   // 'Hello test'
```

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

```
function Movie(title, director, duration) {
  this.title = title;
  this.director = director;
  this.duration = duration;
  this.isLong = () => this.duration > 120;
}

```

```
cosnt movie = new Movie('Inception', 'Nolan', 180);
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, other than that it can create a data from ISO, 8 digit dates, etc. To install this library: `$ npm install dayjs`.

1.4 Asynchronous Programming

In JS when passing functions to other functions it is called **callback**, this functions can *synchronous* or *asynchronous*.

```
function logQuote(quote) {
  console.log(quote);
}

funtion createQuoute(quote, callback) {
  const myQuote = `Like I always say, ${quote}`;
  callback(quote);
}

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:

```
a.forEach(item => ...);   // action on every element of the array
a.every(x => x > 10);     // return true if all elements satisfy the condition,
    false otherwise
a.some(x => x < 10);      // return true if at least one element satisfy the
    condition
a.map(x => `${x}`);       // return a new array with every element mapped to a new
     one
a.filter(x => x === 0);   // return a new array with all elements that satisfy the
     condition
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 `setTimeout`:

```
const f = (task) => {
  // do something
};

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:

```
const onesec = setTimeOut(() \implies {
  console.log('1 second has passed');
}, 1000);
```

There is also the a function that as periodically runs:

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

1.5 Database Access (SQLite)

To use sqlite with `npm`:

```
const sqlite = require('sqlite3');

const db = newsqlite.Database('exams.sqlite',
  (err) => { if (err) throw err; });
```

Example of query:

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

The problem with execution this queries is that they are asynchronous, and they can cause race conditions. The solution to this problem are the `Promise`, which helps simplifying asynchronous programming. A `Promise` handles a `resolve` and a `reject` which are functions that needs to be called when the callback fails or succedes.

```
function waitPromise(duration) {
  return new Promise((resolve, reject) => {
    if (duration < 0) {
      reject(new Error('...'));
    } else {
      setTimeout(resolve, duration);
    }
  }
}

```

```
waitPromise(1000).then((result) => {
  colsole.log('Success :', result);
}).catch((error) => {
  colsole.log('Error :', error);
});
```

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 `all()` or `race()`.

1.6 Await/Async

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

```
function resolveAfter2Seconds() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('resolved');
    }, 2000);
  });
}

async function asyncCall() {
  console.log('calling');
  const result = await resolveAfter2Seconds();
  console.log(result);
}

asyncCall();
```

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