Novedades en ES6 para programar mejor

Usar let/const en lugar de var

ES5

```
var snack = 'Meow Mix';

function getFood(food) {
   if (food) {
       var snack = 'Friskies';
       return snack;
   }
   return snack;
}

getFood(false); // undefined
```

ES6

```
let snack = 'Meow Mix';

function getFood(food) {
   if (food) {
       let snack = 'Friskies';
       return snack;
   }
   return snack;
}

getFood(false); // 'Meow Mix'
```

Reemplazar funciones anónimas por bloques

ES5

```
(function () {
   var food = 'Meow Mix';
}());

console.log(food); // Reference Error
```

ES6

```
{
   let food = 'Meow Mix';
};

console.log(food); // Reference Error
```

Funciones ‘arrow’

ES5

```
function Person(name) {
   this.name = name;
}

Person.prototype.prefixName = function (arr) {
   var that = this; // Guarda el contexto de this
   return arr.map(function (character) {
       return that.name + character;
   });
};
```

ES6

```
function Person(name) {
   this.name = name;
}

Person.prototype.prefixName = function (arr) {
   return arr.map(character => this.name + character);
};
```

Nuevos métodos con cadenas inludes() y repeat()

ES5

```
var string = 'food';
var substring = 'foo';

console.log(string.indexOf(substring) > -1);
```

```
function repeat(string, count) {
    var strings = [];
    while(strings.length < count) {
        strings.push(string);
    }
    return strings.join('');
}
```

ES6

```
const string = 'food';
const substring = 'foo';

console.log(string.includes(substring)); // true
```

```
'meow'.repeat(3); // 'meowmeowmeow'
```

Desestructuraciones de arrays y objetos

ES5

```
var arr = [1, 2, 3, 4];
var a = arr[0];
var b = arr[1];
var c = arr[2];
var d = arr[3];
```

```
var luke = { occupation: 'jedi', father:
'anakin' };
var occupation = luke.occupation; // 'jedi'
var father = luke.father; // 'anakin'
```

ES6

```
let [a, b, c, d] = [1, 2, 3, 4];

console.log(a); // 1
console.log(b); // 2
```

```
let luke = { occupation: 'jedi', father: 'anakin' };
let {occupation, father} = luke;

console.log(occupation); // 'jedi'
console.log(father); // 'anakin'
```

Exportar variables, funciones y objetos

ES5

Antes de ES6, usábamos librerías como Browserify para crear módulos y poder exportar variables o funciones

```
module.exports = 1;
module.exports = { foo: 'bar' };
module.exports = ['foo', 'bar'];
module.exports = function bar () {};
```

ES6

```
// exportamos variables
export let name = 'David';
// exportamos funciones
export function sumOne(a) {
   return a + 1;
}
// exportamos objetos
function sumTwo(a, b) {
   return a + b;
}
let api = {
   sumOne,
   sumTwo
};
export default api;
```

Parámetros en funciones

ES5

```
// Default Parameters
function addTwoNumbers(x, y) {
   x = x || 0;
   y = y || 0;
   return x + y;
}
```

```
// Rest Parameters
function logArguments() {
   for (var i=0; i < arguments.length; i++) {
       console.log(arguments[i]);
   }
}
```

ES6

```
// Default Parameters
function addTwoNumbers(x=0, y=0) {
   return x + y;
}
```

```
// Rest Parameters
function logArguments(...args) {
   for (let arg of args) {
       console.log(arg);
   }
}
```

```
//Spread operator
let cities = ['San Francisco', 'Los Angeles'];
let places = ['Miami', ...cities, 'Chicago'];
```

Clases y extensiones

ES5

```
function Personal(name, age, gender, occupation,
hobby) {
   Person.call(this, name, age, gender);
   this.occupation = occupation;
   this.hobby = hobby;
}

Personal.prototype = Object.create(Person.prototype);
Personal.prototype.constructor = Personal;
Personal.prototype.incrementAge = function () {
   Person.prototype.incrementAge.call(this);
   this.age += 20;
   console.log(this.age);
};
```

ES6

```
class Personal extends Person {
   constructor(name, age, gender, occupation, hobby) {
       super(name, age, gender);
       this.occupation = occupation;
       this.hobby = hobby;
   }

   incrementAge() {
       super.incrementAge();
       this.age += 20;
       console.log(this.age);
   }
}
```

Maps

ES5

```
var map = new Object();
map[key1] = 'value1';
map[key2] = 'value2';
```

ES6

```
let map = new Map();
map.set('name', 'david');
map.get('name'); // david
map.has('name'); // true

let map = new Map([
   ['name', 'david'],
   [true, 'false'],
   [1, 'one'],
   [{}, 'object'],
   [function () {}, 'function']
]);
for (let key of map.keys()) {
   console.log(typeof key);
}
```

Promesas

ES5

```
func1(function (value1) {
    func2(value1, function (value2) {
        func3(value2, function (value3) {
            func4(value3, function (value4) {
                func5(value4, function (value5) {
                    // Do something with value 5
                });
            });
        });
    });
});
```

ES6

```
func1(value1)
    .then(func2)
    .then(func3)
    .then(func4)
    .then(func5, value5 => {
        // Do something with value 5
    });
```

Promesas

En las promesas tenemos dos handlers:

- Resolve: una función llamada cuando se cumple la Promesa.
- Reject: una función llamada cuando la Promesa es rechazada.

```
var request = require('request');

let mypromise = new Promise((resolve, reject) => {
 request.get(url, (error, response, body) => {
   if (body) {
       resolve(JSON.parse(body));
     } else {
       resolve({});
     }
 });
});
```

Promesas en paralelo con Promise.all()

```
let urls = [
   '/api/commits',
   '/api/issues/opened',
 ];
  let promises = urls.map((url) => {
   return new Promise((resolve, reject) => {
     $.ajax({ url: url })
       .done((data) => {
         resolve(data);
       });
   });
 });
  Promise.all(promises)
   .then((results) => {
     // Do something with results of all our
promises
  });
```

Async Await

Nos permite lanzar una promesa y parar la ejecución del código cuando haga falta el resultado de la promesa

```
var request = require('request');
function getJSON(url) {
 return new Promise(function(resolve, reject) {
   request(url, function(error, response, body) {
     resolve(body);
   });
 });
}
async function main() {
 let data = await getJSON();
 console.log(data); // NOT undefined!
}
main();
```

Funciones get y set

```
class Employee {
   constructor(name) {
       this._name = name;
   }
   get name() {
     if(this._name) {
       return 'Mr. ' + this._name.toUpperCase();
     } else {
       return undefined;
     }
   }
   set name(newName) {
     if (newName) {
       this._name = newName;
     } else {
       return false;
}}}
```

```
var emp = new Employee("James Bond");

// uses the get method in the background
if (emp.name) {
 console.log(emp.name);  // Mr. JAMES BOND
}

// uses the setter in the background
emp.name = "Bond 007";
console.log(emp.name);  // Mr. BOND 007
```

.bind()

Debido a que se puede perder el contexto de this dentro de cada clase, en ES5 asinábamos una variable a la instancia this, ES6 nos da otras opciones

```
// bind
function Person(name) {
   this.name = name;
}

Person.prototype.prefixName = function (arr) {
   return arr.map(function (character) {
       return this.name + character;
   }.bind(this));
};
```

```
// arrow function
function Person(name) {
   this.name = name;
}

Person.prototype.prefixName = function (arr) {
   return arr.map(character => this.name +
character);
};
```