

## TEXT CLASS REVIEW

### TEMAS A TRATAR EN EL CUE:

- Definición de variables, scope y bloques.
- Creación de funciones y las características de la arrow function.
- Uso de cadenas de texto y concatenación con variables.
- Desestructuración de objetos.
- Valores por defecto en parámetros de función.
- Spread Operator y ciclo for of.

### DEFINICIÓN DE VARIABLES

A nivel general, **ECMAScript** es una especificación para lenguajes de programación, proporcionado por la organización **ECMA International**.

Su finalidad es definir un estándar, por el cual deben funcionar y ser interpretados lenguajes como JavaScript.

Existen diferentes versiones de esta implementación. La más estable y conocida es la **ES5**, pero en 2016 se originó la **ES6**.

Esta nueva versión contiene nuevas características para el lenguaje JavaScript, las cuales facilitan el trabajo del programador.

Revisaremos algunas de las características más utilizadas actualmente.

Antes de la versión **ES6**, solo se podía utilizar la palabra clave **"var"** para declarar variables.

Una variable declarada con la palabra clave, presenta las siguientes características:

- Su valor puede ser redefinido.
- Su **scope** o alcance pertenece a la función, o es global.
- Ahora gracias a **ES6**, tenemos dos tipos de variables más: **let y const**.
- Ambas variables son de **scope**, o alcance de bloque.
- El valor de **let** puede ser redefinido.
- El valor de **const** no puede ser redefinido.

Veámoslas con un ejemplo gráfico.

Definimos nuestras variables en un archivo js.

```
JS index.js  X
JS index.js > ...
1  var variableVar = 'Palabra clave var';
2  let variableLet = 'Palabra clave let';
3  const variableConst = 'Palabra clave const';
4
5
6
```

Ahora redefiniremos el valor de las variables **let y var**, y luego las imprimimos por consola. Todo correcto hasta aquí.

```
JS index.js  X
JS index.js > ...
1  var variableVar = 'Palabra clave var';
2  let variableLet = 'Palabra clave let';
3  const variableConst = 'Palabra clave const';
4
5  variableVar = 'Redefine var';
6  variableLet = 'Redefine let';
7
8  console.log(variableVar);
9  console.log(variableLet);
```

PROBLEMS   OUTPUT   TERMINAL   DEBUG CONSOLE

```
C:\Users\Es6>nodemon
[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
Palabra clave var
Redefine let
[nodemon] clean exit - waiting for changes before restart
[]
```

Tratemos de redefinir el valor de la variable **const**, e imprimir su valor. Como puedes notar, una vez establecido el valor de una variable declarada con la palabra clave **const**, no podremos volver a redefinirla.

```
JS index.js > ...
1 var variableVar = 'Palabra clave var';
2 let variableLet = 'Palabra clave let';
3 const variableConst = 'Palabra clave const';
4
5 variableVar = 'Redefine var';
6 variableLet = 'Redefine let';
7 variableConst = 'Redefine const';
8
9 console.log(variableVar);
10 console.log(variableLet);
11 console.log(variableConst);
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
C:\Users\Ese6>nodemon
[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting 'node index.js'
C:\Users\Ese6\index.js:7
variableConst = 'Redefine const';
^
TypeError: Assignment to constant variable.
    at Object.<anonymous> (C:\Users\Ese6\index.js:7:15)
    at Module._compile (internal/modules/cjs/loader.js:1063:30)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:1092:10)
    at Module.load (internal/modules/cjs/loader.js:928:32)
    at Function.Module._load (internal/modules/cjs/loader.js:769:14)
    at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.js:72:12)
    at internal/main/run_main_module.js:17:47
[nodemon] app crashed - waiting for file changes before starting...
```

Veamos qué sucede al declarar una variable dentro de un bloque, y consultando su valor fuera de éste.

## VARIABLE LET

```
JS index.js > ...
1 function alcanceVariables() {
2   if (1 === 1) {
3     var variableVar = 'Palabra clave var';
4     let variableLet = 'Palabra clave let';
5     const variableConst = 'Palabra clave const';
6   }
7   console.log(variableLet);
8 }
9
10 alcanceVariables();
11
12
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
C:\Users\Ese6>nodemon
[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting 'node index.js'
C:\Users\Ese6\index.js:7
  console.log(variableLet);
  ^
ReferenceError: variableLet is not defined
    at alcanceVariables (C:\Users\Ese6\index.js:7:17)
    at Object.<anonymous> (C:\Users\Ese6\index.js:10:1)
    at Module._compile (internal/modules/cjs/loader.js:1063:30)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:1092:10)
    at Module.load (internal/modules/cjs/loader.js:928:32)
    at Function.Module._load (internal/modules/cjs/loader.js:769:14)
    at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.js:72:12)
    at internal/main/run_main_module.js:17:47
[nodemon] app crashed - waiting for file changes before starting...
```

## VARIABLE CONST

```
JS index.js > ...
1  function alcanceVariables() {
2      if (1 === 1){
3          var variableVar = 'Palabra clave var';
4          let variableLet = 'Palabra clave let';
5          const variableConst = 'Palabra clave const';
6      }
7      console.log(variableConst);
8  }
9
10  alcanceVariables();
11
12
```

PROBLEMS OUTPUT **TERMINAL** DEBUG CONSOLE

```
C:\Users\Es6>nodemon
[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
C:\Users\Es6\index.js:7
  console.log(variableConst);
                ^
ReferenceError: variableConst is not defined
    at alcanceVariables (C:\Users\Es6\index.js:7:17)
    at Object.<anonymous> (C:\Users\Es6\index.js:10:1)
    at Module._compile (internal/modules/cjs/loader.js:1063:30)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:1092:10)
    at Module.load (internal/modules/cjs/loader.js:928:32)
    at Function.Module._load (internal/modules/cjs/loader.js:769:14)
    at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.js:72:12)
    at internal/main/run_main_module.js:17:47
[nodemon] app crashed - waiting for file changes before starting...
█
```

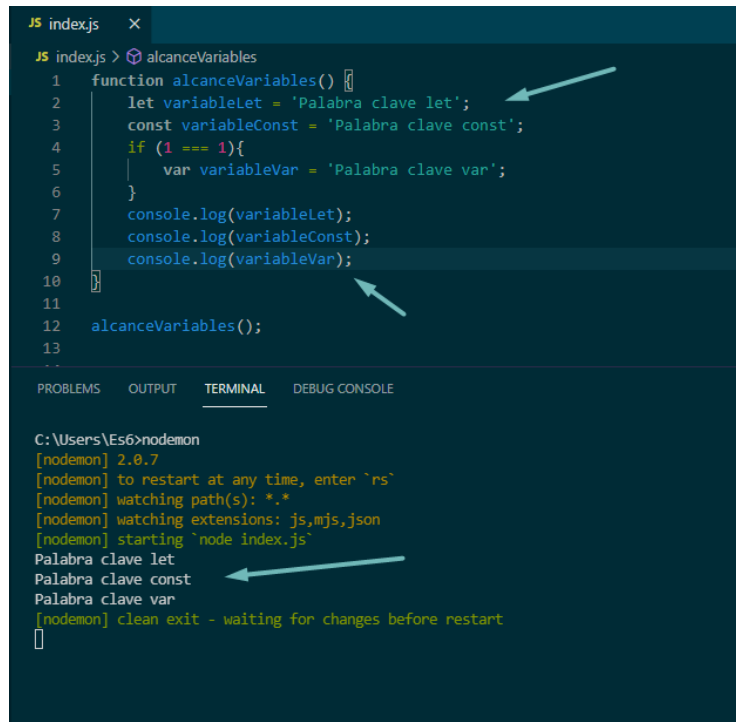
## VARIABLE VAR

```
JS index.js X
JS index.js > ...
1  function alcanceVariables() {
2      if (1 === 1){
3          var variableVar = 'Palabra clave var';
4          let variableLet = 'Palabra clave let';
5          const variableConst = 'Palabra clave const';
6      }
7      console.log(variableVar);
8  }
9
10  alcanceVariables();
11
12
```

PROBLEMS OUTPUT **TERMINAL** DEBUG CONSOLE

```
C:\Users\Es6>nodemon
[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
Palabra clave var
[nodemon] clean exit - waiting for changes before restart
█
```

Como podemos observar, las variables **let** y **const** solo viven dentro del bloque de ejecución, es decir, "{}", mientras que la variable de tipo **var**, puede ser consultada fuera del bloque. En este caso, bastaría con definir nuestras variables **let** y **const** fuera del bloque **if**.



```

JS index.js x
JS index.js > alcanceVariables
1  function alcanceVariables() {
2      let variableLet = 'Palabra clave let';
3      const variableConst = 'Palabra clave const';
4      if (1 === 1){
5          var variableVar = 'Palabra clave var';
6      }
7      console.log(variableLet);
8      console.log(variableConst);
9      console.log(variableVar);
10 }
11
12 alcanceVariables();
13
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

C:\Users\Es6>nodemon
[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
Palabra clave let
Palabra clave const
Palabra clave var
[nodemon] clean exit - waiting for changes before restart
  
```

El comportamiento de la variable **var** puede ocasionar resultados inesperados, o provocar una reasignación de ésta sin darnos cuenta. Es por ello que actualmente se desaconseja su uso, y se utilizan, en cambio, las de tipo **const** y **let**.

## ARROW FUNCTIONS O FUNCIONES DE FLECHA

Ahora existe una nueva sintaxis para la creación de funciones, conocida como: **"arrow function"** o función de flecha.

Ésta suele utilizarse para funciones de **callback**, y también para definir una función dentro de una variable **const**.

La función de flecha nos permite escribir funciones en una línea.

```

16
17 objeto.metodo(argumento, () => { 'Funcion de flecha' } );
18
19 const funcionDeFlecha = () => 'Funcion de flecha';
20
21

```

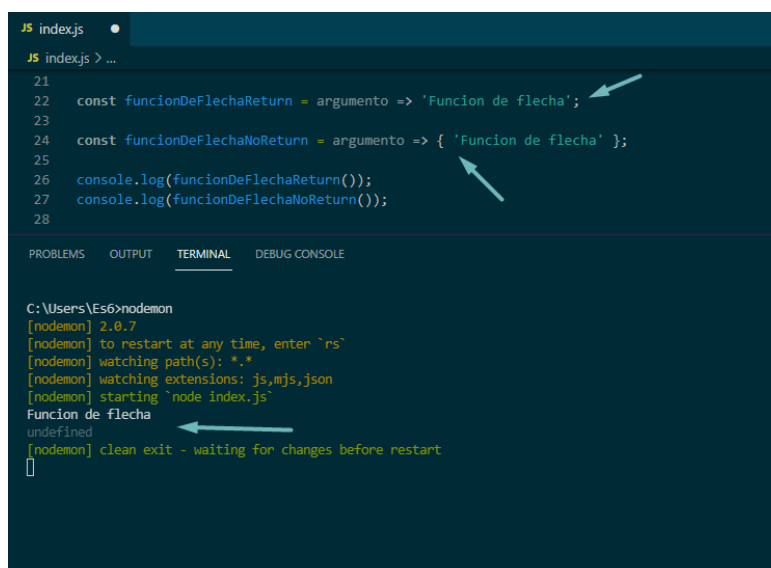
Si necesitamos pasar un único argumento, podemos eliminar el paréntesis; en caso de necesitar más argumentos, no podremos omitir su uso.

```

0
1
2 const funcionDeFlecha = argumento => 'Funcion de flecha';
3
4 const funcionDeFlecha = (argumento1, argumento2) => 'Funcion de flecha';
5
6
7

```

Cuando nuestra función es escrita en una sola línea, podemos omitir las llaves `{ }`; al hacer esto, la palabra clave `return` se encuentra implícita en la función.



```

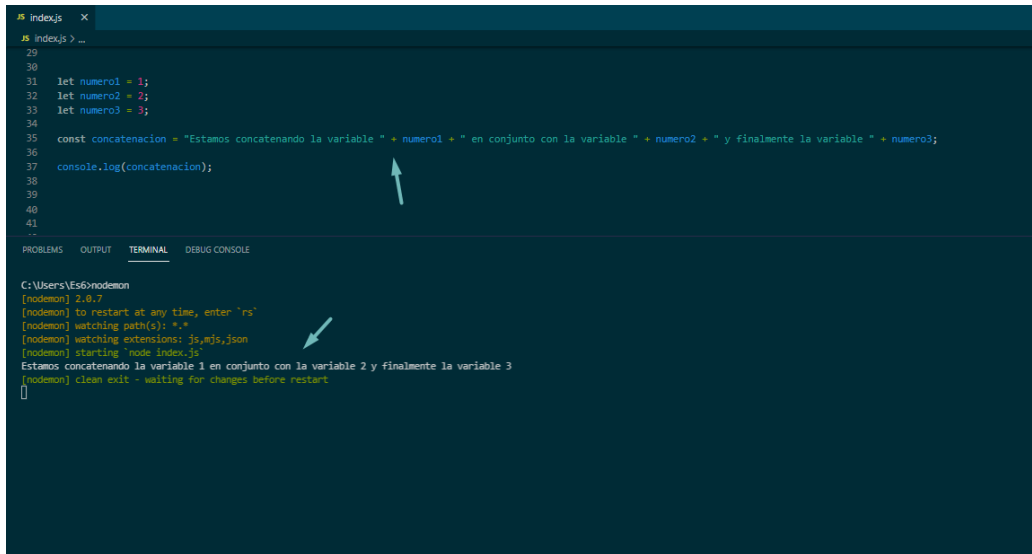
JS index.js
JS index.js > ...
21
22 const funcionDeFlechaReturn = argumento => 'Funcion de flecha';
23
24 const funcionDeFlechaNoReturn = argumento => { 'Funcion de flecha' };
25
26 console.log(funcionDeFlechaReturn());
27 console.log(funcionDeFlechaNoReturn());
28
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

C:\Users\E56>nodemon
[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
Funcion de flecha
undefined
[nodemon] clean exit - waiting for changes before restart

```

## TEMPLATE LITERALS O PLANTILLAS DE CADENA

Antes de **ES6**, la manera que teníamos para concatenar cadenas de texto y variables era la siguiente.



```

29
30
31 let numero1 = 1;
32 let numero2 = 2;
33 let numero3 = 3;
34
35 const concatenacion = "Estamos concatenando la variable " + numero1 + " en conjunto con la variable " + numero2 + " y finalmente la variable " + numero3;
36
37 console.log(concatenacion);
38
39
40
41

```

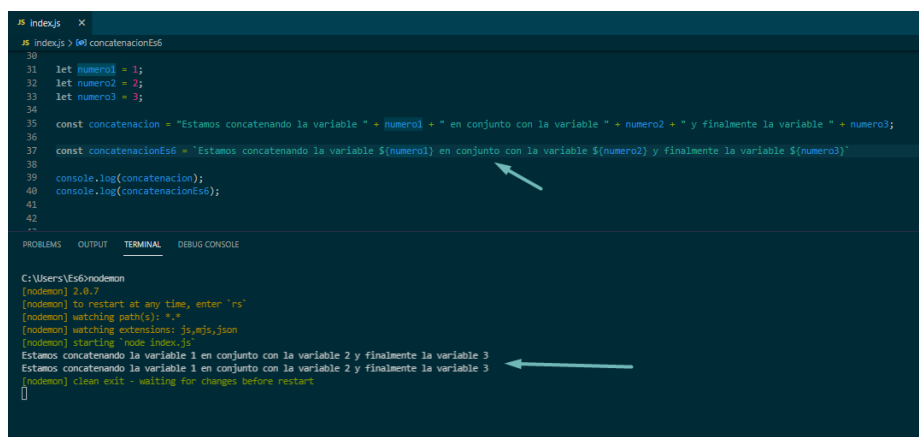
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```

C:\Users\Ese6\node_modules\nodemon
[nodemon] 2.0.7
[nodemon] to restart at any time, enter 'rs'
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting 'node index.js'
Estamos concatenando la variable 1 en conjunto con la variable 2 y finalmente la variable 3
[nodemon] clean exit - waiting for changes before restart

```

Ahora tenemos la siguiente sintaxis, utilizando: **“template literals”**. Ésta permite una forma mucho más humana y fácil de leer una cadena de texto concatenada, y el resultado es el mismo. Para hacer uso de esta característica, debes emplear comillas invertidas **“ ` ”** en vez de las simples o dobles, y cuando quieras hacer uso de una variable, escribirla utilizando signo peso, seguido de ella envuelta en paréntesis de llave: **`${variable}`**.



```

30
31 let numero1 = 1;
32 let numero2 = 2;
33 let numero3 = 3;
34
35 const concatenacion = "Estamos concatenando la variable " + numero1 + " en conjunto con la variable " + numero2 + " y finalmente la variable " + numero3;
36
37 const concatenacionEs6 = `Estamos concatenando la variable ${numero1} en conjunto con la variable ${numero2} y finalmente la variable ${numero3}`;
38
39 console.log(concatenacion);
40 console.log(concatenacionEs6);
41
42

```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```

C:\Users\Ese6\node_modules\nodemon
[nodemon] 2.0.7
[nodemon] to restart at any time, enter 'rs'
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting 'node index.js'
Estamos concatenando la variable 1 en conjunto con la variable 2 y finalmente la variable 3
Estamos concatenando la variable 1 en conjunto con la variable 2 y finalmente la variable 3
[nodemon] clean exit - waiting for changes before restart

```

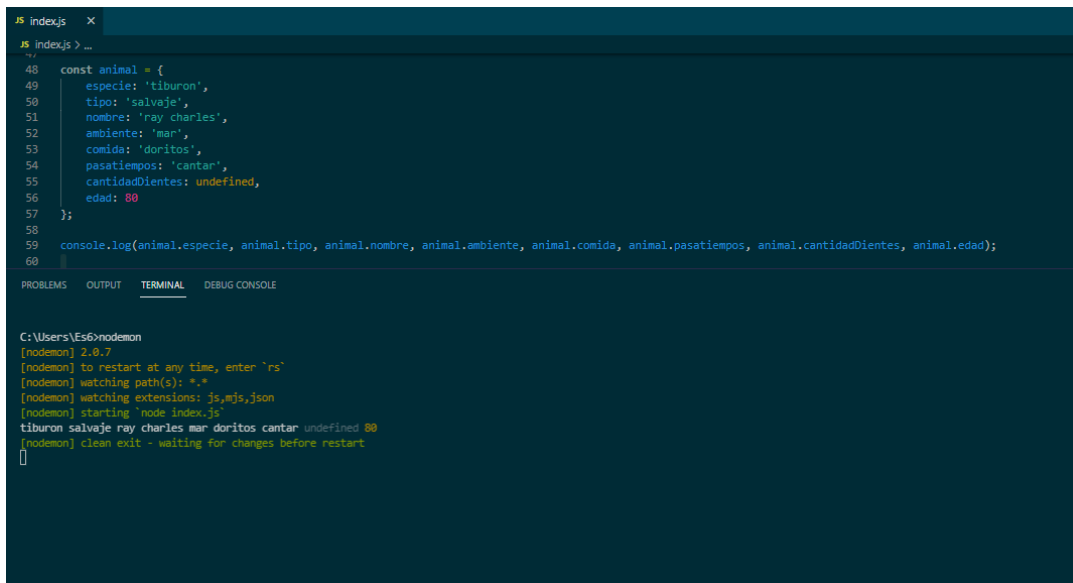
## DESTRUCTURING O DESESTRUCTURACIÓN

El último concepto que revisaremos respecto a ES6, es "Destructuring" o Desestructuración de objetos.

La forma tradicional para acceder a los valores dentro de un objeto, es la siguiente:

```
46
47
48 const animal = { especie: 'tiburon', tipo: 'salvaje', nombre: 'ray charles' };
49
50 console.log(animal.especie, animal.tipo, animal.nombre);
51
52
```

No existe problema en esto, hasta que se presenta un objeto más grande, y con más información.



```
JS index.js x
JS index.js > ...
48 const animal = {
49   especie: 'tiburon',
50   tipo: 'salvaje',
51   nombre: 'ray charles',
52   ambiente: 'mar',
53   comida: 'doritos',
54   pasatiempos: 'cantar',
55   cantidadDientes: undefined,
56   edad: 80
57 };
58
59 console.log(animal.especie, animal.tipo, animal.nombre, animal.ambiente, animal.comida, animal.pasatiempos, animal.cantidadDientes, animal.edad);
60
```

PROBLEMS OUTPUT **TERMINAL** DEBUG CONSOLE

```
C:\Users\E66\nodemon
[nodemon] 2.0.7
[nodemon] to restart at any time, enter 'rs'
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting 'node index.js'
tiburon salvaje ray charles mar doritos cantar undefined 80
[nodemon] clean exit - waiting for changes before restart
```



Podrás notar que para cada valor estamos repitiendo el nombre del objeto constantemente. La desestructuración de objetos nos permite llamar a una llave dentro de uno, definiéndola como variable, y utilizando la siguiente sintaxis.

```
JS index.js X
JS index.js > ...
47
48 const animal = {
49   especie: 'tiburon',
50   tipo: 'salvaje',
51   nombre: 'ray charles',
52   ambiente: 'mar',
53   comida: 'doritos',
54   pasatiempos: 'cantar',
55   cantidadDientes: undefined,
56   edad: 80
57 };
58
59 const { especie } = animal;
60
61 console.log(especie);
62
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
C:\Users\Es6>nodemon
[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
tiburon
[nodemon] clean exit - waiting for changes before restart
```

Utilizando el mismo ejemplo anterior, podemos ver como nuestro código se vuelve mucho más legible.

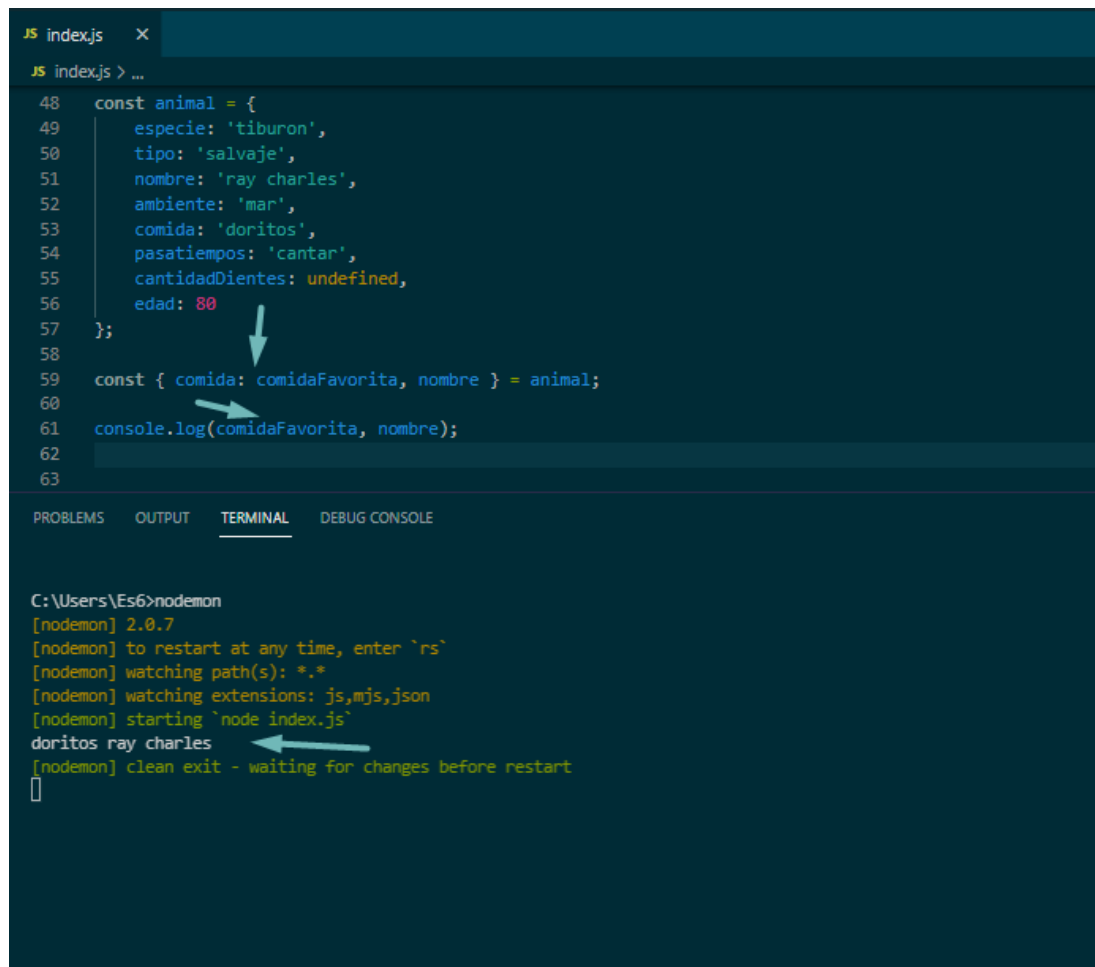
```
JS index.js X
JS index.js > ...
47
48 const animal = {
49   especie: 'tiburon',
50   tipo: 'salvaje',
51   nombre: 'ray charles',
52   ambiente: 'mar',
53   comida: 'doritos',
54   pasatiempos: 'cantar',
55   cantidadDientes: undefined,
56   edad: 80
57 };
58
59 const { especie,tipo,nombre,ambiente,comida,pasatiempos,cantidadDientes,edad } = animal;
60
61 console.log(especie,tipo,nombre,ambiente,comida,pasatiempos,cantidadDientes,edad);
62
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
C:\Users\Es6>nodemon
[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
tiburon salvaje ray charles mar doritos cantar undefined 80
[nodemon] clean exit - waiting for changes before restart
```

Para hacer uso de esta característica, debemos emplear el nombre de la llave del objeto para acceder a su valor, y si quisiéramos cambiar el nombre de la variable, usamos la siguiente sintaxis.

Puedes pensar la desestructuración de objetos como una extracción de las llaves, guardando su información en una variable de una sola línea.



```
JS index.js x
JS index.js > ...

48 const animal = {
49   especie: 'tiburon',
50   tipo: 'salvaje',
51   nombre: 'ray charles',
52   ambiente: 'mar',
53   comida: 'doritos',
54   pasatiempos: 'cantar',
55   cantidadDientes: undefined,
56   edad: 80
57 };
58
59 const { comida: comidaFavorita, nombre } = animal;
60
61 console.log(comidaFavorita, nombre);
62
63
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
C:\Users\Es6>nodemon
[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
doritos ray charles
[nodemon] clean exit - waiting for changes before restart
```

## DEFAULT VALUES O PRÁMETROS PREDETERMINADOS

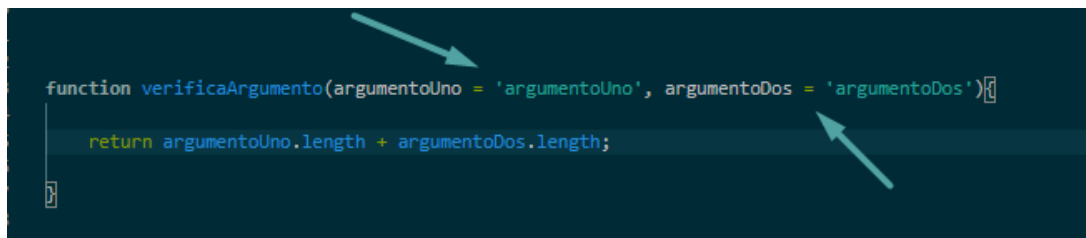
Cuando empiezas a programar y a crear tus propias funciones, notarás que en ciertos momentos, una función puede recibir parámetros de un tipo inesperado que causará un error en el programa; una de las validaciones más básicas que debes hacer, es verificar si el argumento recibido no es indefinido o nulo. Generalmente, la verificación para el argumento dentro de tu función se ve así:

```
function verificaArgumento(argumentoUno, argumentoDos){  
    if(argumentoUno == undefined || argumentoDos == undefined){  
        return console.log("Debes ingresar datos validos para esta funcion")  
    }  
  
    return argumentoUno.length + argumentoDos.length;  
}
```

O bien, si quisieras definir un valor por defecto para tus argumentos, se vería así:

```
function verificaArgumento(argumentoUno, argumentoDos){  
    if( argumentoUno == undefined ) {  
        argumentoUno = 'argumentoUno';  
    }  
  
    if( argumentoDos == undefined ){  
        argumentoDos = 'argumentoDos';  
    }  
  
    return argumentoUno.length + argumentoDos.length;  
}
```

Gracias a los valores predeterminados, se puede definir un valor por defecto en la declaración del argumento, lo que permite refactorizar nuestro código de la siguiente manera:




```
function verificaArgumento(argumentoUno = 'argumentoUno', argumentoDos = 'argumentoDos') {  
    return argumentoUno.length + argumentoDos.length;  
}
```

## SPREAD OPERATOR

Que puede traducirse como operador de expansión, nos permite ciertas utilidades respecto a la manipulación de **arrays** y objetos. Da la posibilidad de pasar los valores contenidos por un objeto o array hacia otro, haciéndolo parte de éste, y sin crear una anidación. Puede parecer complicado al principio, pero utilizaremos ejemplos prácticos para entender su lógica.

Supongamos que tenemos un array, el cual queremos fusionar con otro:



```
JS ejercicio5.js x  
JS ejercicio5.js > ...  
60  
61  
62 const primerArray = [ 'dos', 'tres' ];  
63  
64 const segundoArray = [ 'uno', ...primerArray, 'cuatro', 'cinco' ]  
65  
66 console.log(segundoArray);  
67  
68  
69  
--  
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE  
C:\Users\paquetes>node ejercicio5.js  
[ 'uno', 'dos', 'tres', 'cuatro', 'cinco' ]
```

El **spread operator** se trata de estos tres puntos, seguidos de un array u objeto `...`. Esta característica puede ser utilizada para clonarlos.

```
JS ejercicio5.js X
JS ejercicio5.js > ...
68
69
70 const arrayOriginal = [ 'soy', 'el', 'original' ];
71
72 const arrayClonado = [ ...arrayOriginal ];
73
74 console.log(arrayOriginal);
75 console.log(arrayClonado);
76
77
78
```

PROBLEMS OUTPUT **TERMINAL** DEBUG CONSOLE

```
C:\Users\paquetes>node ejercicio5.js
[ 'soy', 'el', 'original' ]
[ 'soy', 'el', 'original' ]

C:\Users\paquetes>
```

Y también puedes añadir más ítems al mismo tiempo.

```
JS ejercicio5.js X
JS ejercicio5.js > arrayClonado
68
69
70 const arrayOriginal = [ 'soy', 'el', 'original' ];
71
72 const arrayClonado = [ ...arrayOriginal, 'pues', 'ya', 'no' ];
73
74 console.log(arrayOriginal);
75 console.log(arrayClonado);
76
77
78
```

PROBLEMS OUTPUT **TERMINAL** DEBUG CONSOLE

```
C:\Users\paquetes>node ejercicio5.js
[ 'soy', 'el', 'original' ]
[ 'soy', 'el', 'original', 'pues', 'ya', 'no' ]

C:\Users\paquetes>
```

Esta sintaxis también nos permite pasar los valores de un array hacia los parámetros de una función:

```
JS ejercicio5.js x
JS ejercicio5.js > ...
75
76 const arrayOriginal = [ 'soy', 'el', 'original' ];
77
78
79 function retornaArray(itemUno, itemDos, itemTres){
80
81     return `${itemUno} ${itemDos} ${itemTres}`
82 }
83
84
85 console.log(retornaArray(...arrayOriginal));
86
87
88
89
```

PROBLEMS OUTPUT **TERMINAL** DEBUG CONSOLE

```
C:\Users\paquetes>node ejercicio5.js
soy el original
C:\Users\paquetes>
```

Y en objetos, podemos usarlo para agregar o actualizar propiedades de él; solo tenemos que expandir el objeto, y luego definir las propiedades que queremos agregar o actualizar.

Objeto Clonado:

```
JS ejercicio5.js x
JS ejercicio5.js > ...
88
89
90 const objetoOriginal = { cancion: 'Uptown Funk', autor: 'Bruno Mars', año: 2014 };
91
92 const objetoClonado = { ...objetoOriginal };
93
94 console.log(objetoOriginal);
95 console.log(objetoClonado);
96
97
98
99
100
101
102
```

PROBLEMS OUTPUT **TERMINAL** DEBUG CONSOLE

```
C:\Users\paquetes>node ejercicio5.js
{ cancion: 'Uptown Funk', autor: 'Bruno Mars', 'año': 2014 }
{ cancion: 'Uptown Funk', autor: 'Bruno Mars', 'año': 2014 }
C:\Users\paquetes>
```

Objeto Actualizado:

```
JS ejercicio5.js x
JS ejercicio5.js > ...
88
89
90 const objetoOriginal = { cancion: 'Uptown Funk', autor: 'Bruno Mars', año: 2014 };
91
92 const objetoClonado = { ...objetoOriginal };
93
94 const objetoActualizado = { ...objetoOriginal, cancion: 'When I was your man', año: 2013 };
95
96
97 console.log(objetoOriginal);
98 console.log(objetoClonado);
99 console.log(objetoActualizado);
100
101
102
103
```

PROBLEMS OUTPUT **TERMINAL** DEBUG CONSOLE

```
C:\Users\paquetes>node ejercicio5.js
{ cancion: 'Uptown Funk', autor: 'Bruno Mars', 'año': 2014 }
{ cancion: 'Uptown Funk', autor: 'Bruno Mars', 'año': 2014 }
{ cancion: 'When I was your man', autor: 'Bruno Mars', 'año': 2013 }
```

C:\Users\paquetes>

Objeto con propiedades nuevas:

```
JS ejercicio5.js x
JS ejercicio5.js > ...
89
90 const objetoOriginal = { cancion: 'Uptown Funk', autor: 'Bruno Mars', año: 2014 };
91
92 const objetoClonado = { ...objetoOriginal };
93
94 const objetoActualizado = { ...objetoOriginal, cancion: 'When I was your man', año: 2013 };
95
96 const objetoNuevasPropiedades = { ...objetoOriginal, premios: true, genero: 'Pop Funk' };
97
98 console.log(objetoOriginal);
99 console.log(objetoClonado);
100 console.log(objetoActualizado);
101 console.log(objetoNuevasPropiedades);
102
103
104
```

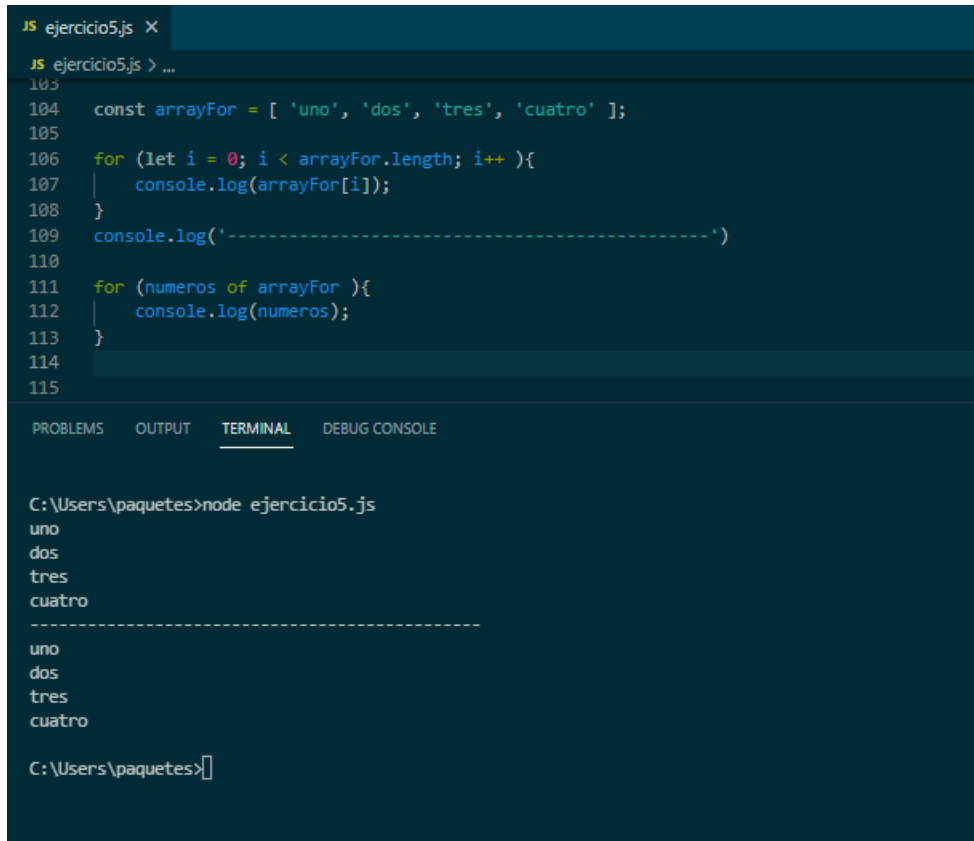
PROBLEMS OUTPUT **TERMINAL** DEBUG CONSOLE

```
C:\Users\paquetes>node ejercicio5.js
{ cancion: 'Uptown Funk', autor: 'Bruno Mars', 'año': 2014 }
{ cancion: 'Uptown Funk', autor: 'Bruno Mars', 'año': 2014 }
{ cancion: 'When I was your man', autor: 'Bruno Mars', 'año': 2013 }
{
  cancion: 'Uptown Funk',
  autor: 'Bruno Mars',
  'año': 2014,
  premios: true,
  genero: 'Pop Funk'
}
```

C:\Users\paquetes>

## CICLO FOR OF

Existe una nueva sintaxis para realizar ciclos **for**, en la que ya no necesitamos conocer el largo de nuestro ítem a iterar. Veamos esta comparación entre un ciclo **for** clásico, y un ciclo **for of**:



```
JS ejercicio5.js X
JS ejercicio5.js > ...
103
104   const arrayFor = [ 'uno', 'dos', 'tres', 'cuatro' ];
105
106   for (let i = 0; i < arrayFor.length; i++){
107     |   console.log(arrayFor[i]);
108   }
109   console.log('-----')
110
111   for (numeros of arrayFor ){
112     |   console.log(numeros);
113   }
114
115

PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

C:\Users\paquetes>node ejercicio5.js
uno
dos
tres
cuatro
-----
uno
dos
tres
cuatro

C:\Users\paquetes>
```

El ciclo **for of** permite una sintaxis más simple para iterar sobre **arrays** y **strings**. Es importante destacar que los objetos no son iterables, por lo tanto, no pueden ser usados dentro de éste.

Como puedes ver, las nuevas características y sintaxis de **ES6** nos permiten escribir un código más claro, optimizado y fácil de leer. Por lo que optimiza el trabajo como desarrolladores, y lo hace agradable.

Esta diferencia en el estilo de código es tan notoria, que existen empresas y ofertas de trabajo que solicitan explícitamente el uso de estas características, por lo tanto, siempre será beneficioso conocer y dominar todo lo expuesto en el presente documento.