

Universidad Politécnica Salesiana

INGENIERÍA EN CIENCIAS DE LA COMPUTACIÓN

INFORME 02



JavaScript

Autor:
Ricardo Romo

April 29, 2020

Contents

1	Variable Let/Var	2
2	Templates	4
3	Destructuración	6
4	Flechas	8
5	Funciones Callbacks	9
6	Promesas	12
7	Async - Await	14

Chapter 1

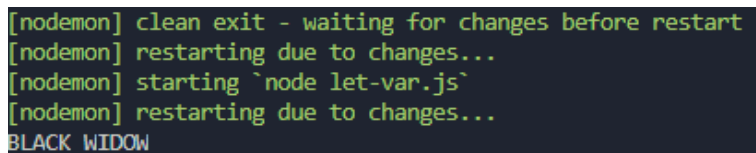
Variable Let/Var

El principal uso de **var** y **let** son para declarar variables, pero existe una gran diferencia entre estas dos.

Las variables instanciadas con **var** pueden repetir su instancia en cualquier momento, es decir si en asignamos **var** a una variable con el mismo nombre varias veces este no dará error, es decir solo se sobre escribiría el contenido

```
1 //console.log(`valor final de i =${i}`);
2
3 var superheroe = "SPIDEMAN"
4 var superheroe = "IROMAN"
```

La respuesta que obtendríamos sería la siguiente:



```
[nodemon] clean exit - waiting for changes before restart
[nodemon] restarting due to changes...
[nodemon] starting `node let-var.js`
[nodemon] restarting due to changes...
BLACK WIDOW
```

A diferencia de **let** esta solo se puede asignar el nombre de una variable una única vez, si se vuelve a instanciar con el mismo esta dará error. El uso de las variables **let**, es el que solo se instancia dentro de la función donde se va usar y subfunciones que desciendan de estas, no pueden ser usadas por funciones superiores a estas.

```
1 let nombre = "Wolverine2";
2
3 if (true) {
4     let nombre = "Magneto";
5 }
6
7 //console.log(`hola ${nombre}`);
```

El resultado que obtendremos será el siguiente:

```
[nodemon] clean exit - waiting for changes before restart
[nodemon] restarting due to changes...
[nodemon] starting `node let-var.js`
hola Wolverine2
```

Esto significa que la variable nombre no puede ser remplazada por la variable nombre instanciada dentro del **if** por lo que solo se imprime el nombre ya antes asignado.

Las variables **let** si pueden ser usadas por funciones si estas son declaradas fuera de estas y así mismo pueden cambiar de valor si estas no han sido instanciadas dentro de la función.

```
1 let i;
2 for (i = 0; i <= 5; i++) {
3   console.log(`i = ${i}`);
4 }
5 console.log(i);
```

```
[nodemon] starting `node let-var.js`
i = 0
i = 1
i = 2
i = 3
i = 4
i = 5
6
```

Chapter 2

Templates

El significado de templates dentro de lo que estamos estudiando, se refiere a como manipulamos los datos tipos cadenas.

En el siguiente ejemplo tenemos dos formas diferentes en la cual podemos crear variables con información ya antes obtenida.

```
1 let nombre = "deadpool";
2 let real = 'Wade Winston';
3 let nombre_completo = nombre + " " + real;
4 let nombreTemplate = `${nombre} ${real}`;
```

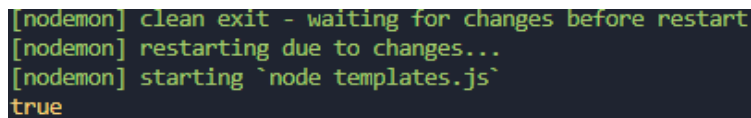
En la primer podemos usar el signo `+` para concatenar las variables **nombre** y **real**.

En el segundo utilizamos los signos `${ variable }` los cuales hacen referencia a la variable que se va a usar, esto evita mucho el uso de `""` poniendo dentro de uno solo.

Podemos tambien compara valores para saber si los contenidos de las dos varibales son iguales en estructura y contenido, y para esto usamos el `===` (**tripe igual**).

```
1 let nombre = "deadpool";
2 let real = 'Wade Winston';
3 let nombre_completo = nombre + " " + real;
4 let nombreTemplate = `${nombre} ${real}`;
```

Obteniendo el resultado siguiente:



```
[nodemon] clean exit - waiting for changes before restart
[nodemon] restarting due to changes...
[nodemon] starting `node templates.js`
true
```

El valor obtenido en la consola es un valor logico, diciendonos que el contenido y estructura son iguales.

Asi mismo podemos usar funciones utilizando estas variables

```
1 let nombre = "deadpool";
2 let real = 'Wade Winston';
3 let nombre_completo = nombre + " " + real;
4 let nombreTemplate = `${nombre} ${real}`;
5
6
7 function getNombre() {
8     return `${nombre} ${real}`;
9 }
10 console.log(`El nombre es: ${getNombre()}`);
```

```
[nodemon] clean exit - waiting for changes before restart
[nodemon] restarting due to changes...
[nodemon] starting `node templates.js`
El nombre es: deadpool Wade Winston
```

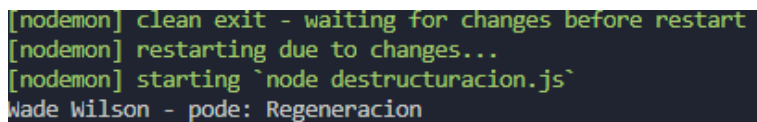
Chapter 3

Destructuración

Nos referimos a **deestructuración** al uso de variables como objetos, es decir usar variables para guardar diferentes tipos de variables, incluso usando atributos de esta variable como el siguiente ejemplo.

```
1 let deadpool = {
2   nombre: "Wade",
3   apellido: "Wilson",
4   poder: 'Regeneracion',
5   getNombre: function() {
6     return `${this.nombre} ${this.apellido} - pode: ${
7       this.poder}`;
8   }
9 }
```

En este ejemplo creamos la variable **deadpool** como una variable de deestructuración, dándole atributos como **nombre**, **apellido**, **poder**, e incluso asignándole **una función** la cual retorna una cadena con todos los atributos antes mostrados con solo llamar a la variable y su función.

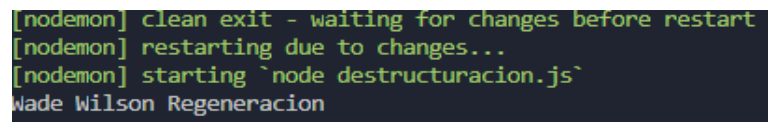


```
[nodemon] clean exit - waiting for changes before restart
[nodemon] restarting due to changes...
[nodemon] starting `node deestructuracion.js`
Wade Wilson - pode: Regeneracion
```

Así mismo podemos crear nuevas variables y asignarles los valores de los atributos ya insertados.

```
1 let deadpool = {
2   nombre: "Wade",
3   apellido: "Wilson",
4   poder: 'Regeneracion',
5   getNombre: function() {
6     return `${this.nombre} ${this.apellido} - pode: ${
7       this.poder}`;
8   }
9 }
```

```
10 let { nombre, apellido, poder } = deadpool;
11 console.log(nombre, apellido, poder);
```



A terminal window with a dark background and light green text. It shows the following output: [nodemon] clean exit - waiting for changes before restart, [nodemon] restarting due to changes..., [nodemon] starting `node deestructuracion.js`, and Wade Wilson Regeneracion.

```
[nodemon] clean exit - waiting for changes before restart
[nodemon] restarting due to changes...
[nodemon] starting `node deestructuracion.js`
Wade Wilson Regeneracion
```

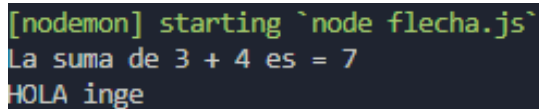

Chapter 4

Flechas

El uso de **flechas** ayuda mucho dentro del tema **optimización de código** lo cual es fundamental para el tema de uso de recursos.

Las flechas son prácticamente funciones optimizadas en código, así mismo a través de estas podemos enviar atributos, retornar valores y hacer operaciones.

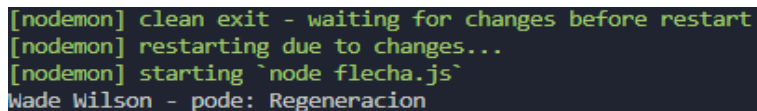
```
1 let sumar = (a, b) => a + b;  
2 let saludar = (nombre) => `HOLA ${nombre}`;  
3 //console.log(`La suma de 3 + 4 es = ${sumar(3,4)}`);  
4 //console.log(`${saludar("inge")}`);
```



```
[nodemon] starting `node flecha.js`  
La suma de 3 + 4 es = 7  
HOLA inge
```

Las **flechas**, así mismo se puede usar como una función normal dentro de una variable con atributos dentro.

```
1 let deadpool = {  
2   nombre: "Wade",  
3   apellido: "Wilson",  
4   poder: "Regeneracion",  
5   getNombre: () => {  
6     return `${deadpool.nombre} ${deadpool.apellido} - pode:  
7       ${deadpool.poder}`;  
8   }  
9 }  
10 console.log(`${deadpool.getNombre()}`);
```



```
[nodemon] clean exit - waiting for changes before restart  
[nodemon] restarting due to changes...  
[nodemon] starting `node flecha.js`  
Wade Wilson - pode: Regeneracion
```

Chapter 5

Funciones Callbacks

Una **función de callback** es una función que se pasa a otra función como un argumento, que luego se invoca dentro de la función externa para completar algún tipo de rutina o acción. La funcionalidad principal del ejemplo a presentar, es el de presentar buscar a una persona mediante su id, mostrar su sueldo y en caso de no encontrarla mostrar un mensajes sobre la falla o error encontrado.

1. Instanciamos dos Variables tipo Array, la primera almacenara el id y nombre de los participantes, la segunda, el id y el sueldo que este recibe.

```
1 let empleados = [  
2   { id: 1, nombre: "Ricardo" },  
3   { id: 2, nombre: "Gabriel" },  
4   { id: 3, nombre: "Will" }  
5 ];  
6  
7 let salarios = [  
8   { id: 1, salario: 800 },  
9   { id: 2, salario: 950 }  
10 ]
```

2. Creación del callback getEmpleado

```
1 let getEmpleado = (id, callback) => {  
2   let empleadosDB = empleados.find(empleo =>  
3     empleo.id === id);  
4   if (!empleadosDB) {  
5     callback(`No existe un empleado con id ${id}`)  
6   } else {  
7     callback(null, empleadosDB);  
8   }  
9 }
```

- (a) La variable getEmpleado almacena una función tipo flecha, recibiendo los atributos del id, y la función llamada callback,

- (b) La variable empleadoDb recorre la variable empleados (ids y nombres) buscando cual id es igual al ingresado
- (c) En la condición if, si la variable empleadoDb no tiene información (es decir no se encontró al empleado), ingresara dentro de la función callback el mensaje de **No existe empleado**
- (d) Caso contrario retornara un valor nulo y la variable empleadoDB.

3. Creacion callback getSalario

```

1 let getSalario = (empleado, callback) => {
2   let salarioDb = salarios.find(salario => salario.id
3     === empleado.id);
4   if (!salarioDb) {
5     callback(`No se encontro el salario para el
6       empleado ${empleado.nombre}`);
7   } else {
8     callback(null, { nombre: empleado.nombre, salario:
9       salarioDb.salario });
10  }
11 };

```

- (a) La variable getSalario guarda una función tipo flecha con los atributos a recibir de un objeto y una función desarrollarse.
- (b) La variable salarioDB recorre el array salario (id y salario) antes creado hasta encontrar cual es igual al id del salario con el del empleado.
- (c) En la sentencia if, si la variable no almaceno ningún valor (es decir que no se encontró) retornara un mensaje de error confirmando que no se encontró salario para tal usuario.
- (d) En caso de si almacenar un valor, se ejecuta la función callback enviando como un objeto creado con el nombre del empleado y el salario que este recibe.

4. Invocando a las funciones.

```

1 getEmpleado(2, (err, usuario) => {
2   if (err) {
3     return console.log(err);
4   }
5   getSalario(usuario, (err, respuesta) => {
6     if (err) {
7       return console.log(err);
8     }
9     console.log(`el salario de ${respuesta.nombre} es
10       de ${respuesta.salario}`);
11   });
12 });

```

- (a) Invocamos la función getEmpleado enviando como parámetro un **id = 2** y el siguiente atributo que enviamos es una función recibiendo los atributos de **erro** y el de **empleadoDB**

- (b) En la condición if, en caso de recibir un error, mandara a imprimir en consola
- (c) caso de no recibir un error, invocara a la función `getSalario`, la cual envía al objeto **usuario** recibido(**empleadoDB**), y otra función dos atributos, uno de error y el con la respuesta.
- (d) En caso de encontrar un error se nos imprimirá este, caso contrario obtendremos nuestra respuesta.

```
[nodemon] starting `node callbacks2.js`  
el salario de Gabriel es de 950  
[nodemon] clean exit - waiting for changes before restart
```

Chapter 6

Promesas

Una promesa es un objeto devuelto al cuál se adjuntan funciones callback, en lugar de pasar callbacks a una función.

El ejemplo a mostrar es el mismo que el anterior, realizado en callbacks pero ahora, se desarrollara con promesas.

1. Inicializamos las variables empleados y salarios

```
1 let empleados = [  
2   { id: 1, nombre: "Ricardo" },  
3   { id: 2, nombre: "Gabriel" },  
4   { id: 3, nombre: "Will" }  
5 ];  
6  
7 let salarios = [  
8   { id: 1, salario: 800 },  
9   { id: 2, salario: 950 }  
10 ]
```

2. Promesa getEmpleado

```
1 ]  
2  
3 let getEmpleado = (id, callback) => {  
4   let empleadosDB = empleados.find(emplado =>  
5     emplado.id === id);  
6   if (!empleadosDB) {  
7     callback(`No existe un empleado con id ${id}`)  
8   } else {  
9     callback(null, empleadosDB);  
10  }  
11 }
```

- (a) Creamos la variable getEmpleado el cual almacena una función tipo flecha con instanciando dentro de su ejecución una promesa.
- (b) Dentro de la promesa tenemos dos atributos a recibir **resolve** (Si la función se cumple correctamente), **reject**(si ocurre algún error)

- (c) Así mismo en la variable empleadoDB guardamos la búsqueda dentro del array empleados que tenga igual al id ingresado.
- (d) En caso de no guardar ninguna información utilizaremos el atributo **reject** de la promesa para enviar un error
- (e) Caso contrario utilizaremos el atributo **resolve** para enviar la respuesta.

3. Promesa getSalario

```

1
2 let getSalario = (empleado, callback) => {
3     let salarioDb = salarios.find(salario => salario.id
4         === empleado.id);
5     if (!salarioDb) {
6         callback(`No se encontro el salario para el
7             empleado ${empleado.nombre}`);
8     } else {
9         callback(null, { nombre: empleado.nombre, salario:
            salarioDb.salario });
10    }
11 };

```

- (a) Creamos la variable getSalario almacenando una función recibiendo los atributos empleado(objeto).
- (b) Instanciamos una promesa con sus atributos **resolve** y **reject**
- (c) la variable almacena el salario que pertenezca al empleado recibido a través de su id.
- (d) en caso no almacenar ningún dato salarioDB se ejecuta el atributo **reject** enviando un mensaje de error
- (e) Caso contrario Se crea un objeto guardando el nombre y sueldo del empleado.

4. Invocacion de la promesa

- (a) Al invocar getEmpleado mandamos un id en este caso no valido.
- (b) **.then** nos ayuda a identificar si la promesa se ejecutó con éxito, guardando el resultado en una variable llamada empleados.
- (c) al ejecutarse con éxito la promesa getSalario es invocada mandamos la respuesta almacenada llamada empleados.
- (d) **.catch()** nos ayuda atrapar los errores lanzados por las promesas almacenándolos en una variable llamada err.

```

[nodemon] clean exit - waiting for changes before restart
[nodemon] restarting due to changes...
[nodemon] starting `node promesas.js`
No existe un empleado con id 34

```

Chapter 7

Async - Await

Las funciones asíncronas utilizan la sintaxis `async` y `await` para esperar que una promesa sea resuelta. En este capítulo usaremos el ejemplo del desarrollado en las promesas, pero implementando las sintaxis de **async-await**

1. Inicializar variables empleados y salarios

```
1 let empleados = [  
2   { id: 1, nombre: "Ricardo" },  
3   { id: 2, nombre: "Gabriel" },  
4   { id: 3, nombre: "Will" }  
5 ];  
6 let salarios = [  
7   { id: 1, salario: 800 },  
8   { id: 2, salario: 950 }  
9 ]
```

2. Inicializar promesa getEmpleado

```
1 let getEmpleado = async(id) => {  
2   let empleadosDB = empleados.find(empleado =>  
3     empleado.id === id);  
4   if (!empleadosDB) {  
5     throw new Error(`No existe un empleado con id ${id}`)  
6   } else {  
7     return (empleadosDB);  
8   }  
9 }
```

- (a) A diferencia de las promesas al inicializar la función antes de los atributos agregamos la palabra `async`, la cual hace referencia a que la función va a devolver una respuesta y va a ser capturada por un `await` en algún momento.
- (b) pedimos de parámetros el `id` del empleado

- (c) buscamos a que empleado le corresponde el id y guardamos en la variable empleadoDB.
- (d) en caso de que no se encuentra creamos y lanzamos un error con el comando **throw**.
- (e) en caso de si encontrarlo retornamos la respuesta que es un objeto.

3. Inicializar promesa getSalario

```

1 let getSalario = (empleado) => {
2     let salarioDb = salarios.find(salario => salario.id
3       === empleado.id);
4     if (!salarioDb) {
5         throw new Error(`No se encontro el salario para el
6           empleado ${empleado.nombre}`);
7     } else {
8         return ({ nombre: empleado.nombre, salario:
9           salarioDb.salario });
10    }
11 }

```

- (a) Recibimos de atributo un objeto llamado empleado.
- (b) Buscamos el salario que corresponda a tal empleado a través de sus ids.
- (c) En caso no encontrar lanzamos un error con el comando **throw**
- (d) Caso contrario se retorna un objeto con el nombre y salario del empleado.

4. Inicializar async getInformacion

```

1 let getInformacion = async(id) => {
2     let empleado = await getEmpleado(id);
3     let salario = await getSalario(empleado);
4     return `El salario de ${salario.nombre} es de: ${
5       salario.salario}`;
6 }

```

- (a) Recibimos un atributo llamado id.
- (b) creamos una variable llamado empleado, la cual con el comando await recibe la solución realizada por la promesa getEmpleado.
- (c) Al crear una variable salario, junto con el comando await recibe la solución de la promesa a la cual se envía la variable empleado.
- (d) Al final de la operación retornamos un mensaje con la solución

5. LLamada al async getInformacion

```

1
2 getInformacion(1)
3   .then(mensaje => console.log(mensaje))

```

- (a) Llamamos al async `getInformacion` y el enviamos el id 1
- (b) `.then()` nos ayuda a crear una variable (**mensaje**) la cual va a recibir la respuesta.
- (c) `.catch()` nos ayuda a recibir los errores que son lanzados en el transcurso del código, imprimiendo en consola cual fue el error.

```
[nodemon] clean exit - waiting for changes before restart
[nodemon] restarting due to changes...
[nodemon] starting `node async-await.js`
El salario de Ricardo es de: 800
```