Apuntes del módulo

DigitalHouse >
Coding School

Índice

1 | Node.js

Módulos y require

¿Qué es un módulo? Un módulo es una **unidad de código que se encarga de resolver una problemática en particular** y es, naturalmente, reutilizable. En un futuro utilizaremos módulos externos para expandir las características de nuestra aplicación.

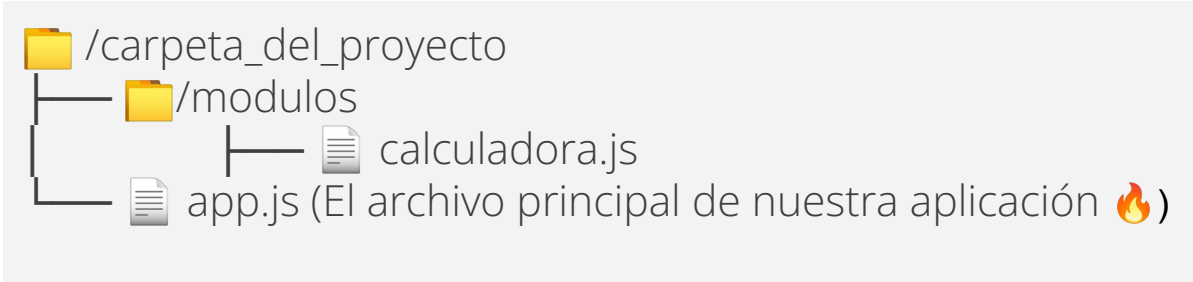
Así como los tenemos externos, también podremos crear nuestros propios módulos. La idea es aislarlos en archivos independientes para poder ordenarlos, llevar un control de los mismos y, de esta manera, darles mantenimiento de manera fácil y organizada.

Módulos y require

Imaginen entonces que tenemos una carpeta que contiene todo nuestro proyecto. Dentro de ella hay un archivo principal `app.js`, donde estaremos realizando diversas tareas. Si quisiéramos hacer un módulo que pueda sumar, restar, dividir y multiplicar, podríamos entonces crear un módulo llamado `calculadora.js`, al que vamos a ubicar por fuera de nuestro archivo principal. Para ello, vamos a crear una carpeta llamada módulos y aquí iremos ubicando este y los demás módulos que creemos.

Módulos y require

El árbol de archivos de este proyecto se vería así:



La idea de cada módulo es **agrupar funciones y datos** que vamos a reutilizar en el nuevo archivo principal, o en otros también. Entonces, nuestro archivo de calculadora quedaría algo así:

JS

```
let calculadora = {
    sumar: function(numeroA,numeroB) {
    return numeroA + numeroB
    },
    restar: function(numeroA,numeroB) {
    return numeroA - numeroB
    }
}
module.exports = calculadora;
```

Módulos y require

En la línea `module.exports = calculadora` estamos aclarando que hay algo dentro de la variable calculadora que será aquello que queremos tener disponible para que otros archivos puedan utilizar. De alguna manera, estamos exportando lo que hay guardado en la variable calculadora. Pero no termina acá.

Ahora deberíamos —en el archivo donde quisiéramos usar la calculadora— **requerir este módulo**. ¿Cómo hacemos esto? Vayamos al archivo `app.js` y veamos cómo queda el código.

```
// Primero vamos a requerir el módulo y guardar lo que se exporta
en una variable
let unaCalculadora = require('./modulos/calculadora')

// no hace falta ponerle .js
console.log( unaCalculadora.sumar(2, 9) )

// Si ejecutamos app.js por terminal, debería imprimirse en la
consola el número 11
```

Documentación

Documentación de Node.js:
https://nodejs.org/dist/latest-v12.x/docs/api/

Módulo de FS - documentación:
https://nodejs.org/api/fs.html

Módulo process:
https://nodejs.org/dist/latest-v12.x/docs/api/process.html

2 | Tipos de datos

DigitalHouse> Coding School

Numéricos (number)

```
let edad = 35; // número entero
let precio = 150.65; // decimales
```

Como JavaScript está escrito en inglés usaremos un punto para separar los decimales.

Cadenas de caracteres (string)

```
let nombre = 'Mamá Luchetti'; // comillas simples
let ocupacion = "Master of the sopas"; // comillas dobles tienen el mismo resultado
```

Lógicos o booleanos (boolean)

```
let laCharlaEstaReCopada = true;
let hayAsadoAlFinal = false;
```

Objetos (object)

A diferencia de otros tipos de datos que pueden contener un solo dato, los objetos son **colecciones** de datos y en su interior pueden existir todos los anteriores.

Los podemos reconocer porque se declaran con llaves **{ }**.

```
{}
let persona = {
  nombre: 'Javier', // string
  edad: 34, // number
  soltero: true // boolean
}
```

Array

Al igual que los objetos, los arrays son colecciones de datos. Los podemos reconocer porque se declaran con corchetes **[ ]**.

Los arrays son un tipo especial de objetos, por eso **no los consideramos como un tipo de dato más**.

Los mencionamos de manera especial porque son muy comunes en todo tipo de código.

```
{}
let comidasFavoritas = ['Milanesa napolitana',
'Ravioles con bolognesa', 'Pizza calabresa'];

let numerosSorteados = [12, 45, 56, 324, 452];
```

NaN (Not a Number)

Indica que el valor no puede ser parseado como un número.

```
{}  let malaDivision = "35" / 2; // NaN no es un número
```

Null (valor nulo)

Lo asignamos nosotros para indicar un valor vacío o desconocido.

```
{}  let temperatura = null; // No llegó un dato, algo falló
```

Undefined (valor sin definir)

Indica la ausencia de valor.

Las variables tienen un valor indefinido hasta que les asignamos uno.

```
{}
let saludo; // undefined, no tiene valor
saludo = "¡Hola!"; // Ahora sí tiene un valor
```

3 Funciones, arrow functions y callbacks

Funciones: estructura básica

```
function sumar (a, b) {
    return a + b;
}
```

Funciones declaradas

Son aquellas que se declaran usando la **estructura básica**. Pueden recibir un **nombre**, escrito a continuación de la palabra reservada **function**, a través del cual podremos invocarla.

Las funciones con nombre son **funciones nombradas**.

```
{}
function hacerHelado(cantidad) {
    return '🍦'.repeat(cantidad);
}
```

Funciones expresadas

Son aquellas que **se asignan como valor** de una variable. En este caso, la función en sí no tiene nombre, es una **función anónima**.

Para invocarla podremos usar el nombre de la variable que declaremos.

```
{}
let hacerSushi = function (cantidad) {
    return '🍣'.repeat(cantidad);
}
```

Invocando una función

Antes de poder invocar una función, necesitamos que haya sido declarada. Entonces, vamos a declarar una función:

```
function hacerHelado() {
    return '🍦';
}
```

La forma de **invocar** (también se puede decir llamar o ejecutar) una función es escribiendo su nombre seguido de apertura y cierre de paréntesis.

```
hacerHelado(); // Retornará '🍦'
```

Invocando una función

Si la función tiene parámetros, se los podemos pasar dentro de los paréntesis cuando la invocamos. **Es importante respetar el orden**, ya que JavaScript asignará los valores en el orden en que lleguen.

{}

```
function saludar(nombre, apellido) {
    return 'Hola ' + nombre + ' ' + apellido;
}

saludar('Robertito', 'Rodríguez');
// retornará 'Hola Robertito Rodríguez'
```

Invocando una función

También es importante tener en cuenta que, cuando tenemos parámetros en nuestra función, JavaScript va a esperar que se los indiquemos al ejecutarla.

```
function saludar(nombre, apellido) {
    return 'Hola ' + nombre + ' ' + apellido;
}

saludar(); // retorna 'Hola undefined undefined'
```

En este caso, al no haber recibido el argumento que necesitaba, JavaScript le asigna el tipo de dato **undefined** a los parámetros *nombre* y *apellido*.

Invocando una función

Para casos como el anterior podemos definir **valores por defecto**.

Si agregamos un igual = luego un parámetro, podremos especificar su valor en caso de que no llegue ninguno.

```
{}
function saludar(nombre = 'visitante',
    apellido = 'anónimo') {
    return 'Hola ' + nombre + ' ' + apellido;
}

saludar(); // retornará 'Hola visitante anónimo'
```

Las **arrow functions** reciben su nombre por el operador `=>`, que se parece a una flecha.
En inglés suele llamarse fat arrow (flecha gorda) para diferenciarlo de la flecha simple `->`.

{código}

```
let saludo = () => 'Hola Mundo!';

let dobleDe = numero => numero * 2;

let suma = (a, b) => a + b;

let horaActual = () => {
    let fecha = new Date();
    return fecha.getHours() + ':' +
    fecha.getMinutes();
}
```

Función arrow **sin parámetros**.

Requiere de los paréntesis para iniciarse.

Al tener una sola línea de código, y que esta misma sea la que queremos retornar, el return queda implícito.

{código}

```
let saludo = () => 'Hola Mundo!';

let dobleDe = numero => numero * 2;

let suma = (a, b) => a + b;

let horaActual = () => {
    let fecha = new Date();
    return fecha.getHours() + ':' +
    fecha.getMinutes();
}
```

Función arrow con un **único parámetro** (no necesitamos los paréntesis para indicarlo) y con un return implícito.

{código}

```
let saludo = () => 'Hola Mundo!';

let dobleDe = numero => numero * 2;

let suma = (a, b) => a + b;

let horaActual = () => {
    let fecha = new Date();
    return fecha.getHours() + ':' +
    fecha.getMinutes();
}
```

Función arrow con **dos parámetros**.

Necesita de los paréntesis y tiene un return implícito.

{código}

```
let saludo = () => 'Hola Mundo!';

let dobleDe = numero => numero * 2;

let suma = (a, b) => a + b;
```

```
let horaActual = () => {
    let fecha = new Date();
    return fecha.getHours() + ':' +
    fecha.getMinutes();
}
```

Función arrow **sin parámetros** y con un **return explícito**.

En este caso hacemos uso de las llaves y del return ya que la lógica de esta función se desarrolla en más de una línea de código.

“

Un **callback** es una **función** que se pasa como **parámetro** de otra **función**.

La función que lo recibe es quien se encarga de **ejecutarla** cuando sea necesario.

”

El callback anónimo

En este caso, la función que pasamos como **callback** no tiene nombre. Es decir, es una **función anónima**.

Como las **funciones anónimas** no pueden ser llamadas por su nombre, debemos escribirla dentro de la función que se encargará de llamar al callback.

```
{}
setTimeout( function(){
    console.log('Hola Mundo!')
} , 1000)
```

El callback definido

La función que pasamos como **callback** puede ser una función previamente **definida**. Al momento de pasarla como parámetro de otra función, nos referiremos a la misma por su nombre.

```
{}
let miCallback = () => console.log('Hola mundo!');
setTimeout(miCallback, 1000);
```

Al escribir una función como parámetro lo hacemos **sin los paréntesis** para evitar que se ejecute. Será la función que la recibe quien se encargue de ejecutarla.

4 | Condicionales

DigitalHouse> Coding School

“

Nos permiten **evaluar condiciones** y realizar diferentes acciones según el resultado de esas evaluaciones.

”

Condicional simple

Versión más básica del `if`. Establece una condición y un bloque de código a ejecutar en caso de que sea verdadera.

```
{}
if (condición) {
  // código a ejecutar si la condición es verdadera
}
```

Condicional con bloque else

Igual al ejemplo anterior, pero agrega un bloque de código a ejecutar en caso de que la condición sea falsa.

Es importante tener en cuenta que el bloque `else` es opcional.

```
{}
if (condición) {
  // código a ejecutar si la condición es verdadera
} else {
  // código a ejecutar si la condición es falsa
}
```


DigitalHouse> Coding School

Condicional con bloque else if

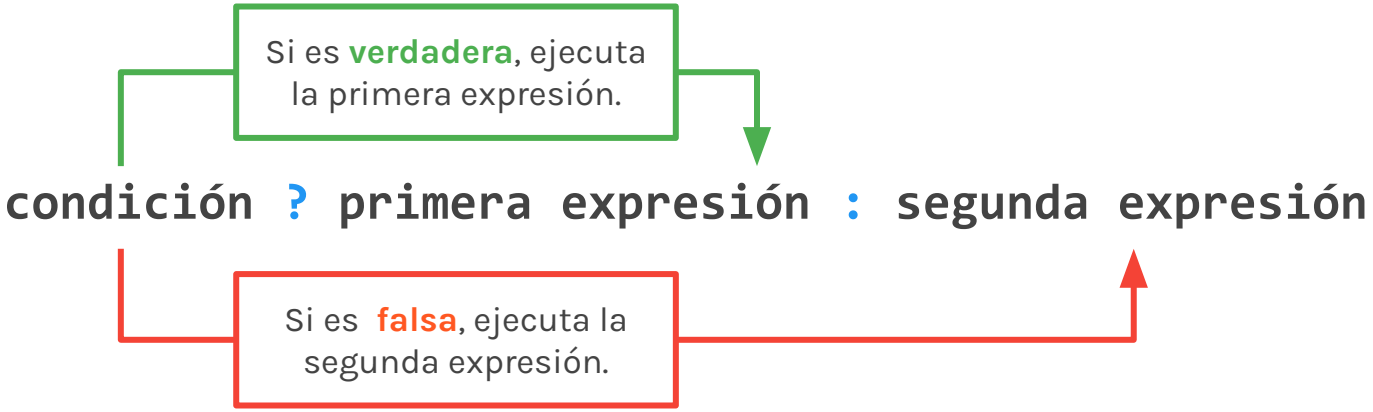
Igual que el ejemplo anterior, pero agrega un `if` adicional. Es decir, otra condición que puede evaluarse en caso de que la primera sea falsa.

Podemos agregar todos los bloques `else if` que queramos, **solo uno podrá ser verdadero**. De lo contrario entrará en acción el bloque `else`, si existe.

```
if (condición) {
  // código a ejecutar si la condición es verdadera
} else if (otra condición) {
  // código a ejecutar si la otra condición es verdadera
} else {
  // código a ejecutar si todas las condiciones son falsas
}
```

If ternario: estructura básica

A diferencia de un if tradicional, el **if ternario** se escribe de forma **horizontal**. Al igual que el if tradicional tiene el mismo flujo (si esta condición es verdadera hacer esto, si no, hacer esto otro), pero en este caso **no hace falta** escribir la palabra **if ni la palabra else**.



If ternario: estructura básica

Para el if ternario **es obligatorio** poner código en la **segunda expresión**. Si no queremos que pase nada, podemos usar un string vacío `''`.

```
{}  4 > 10  ? 'El 4 es más grande' : 'El 10 es más grande';
```

Condición

Declaramos una expresión que se evalúa como true o false.

Primera expresión

Si la condición es verdadera, se ejecuta el código que está después del signo de interrogación.

Segunda expresión

Si la condición es falsa, se ejecuta el código que está después de los dos puntos.

Es obligatorio escribirla.

Switch: estructura básica

El switch está compuesto por una expresión a evaluar, seguida de diferentes casos, tantos como queramos, cada uno contemplando un escenario diferente.

Los casos deberán terminar con la palabra reservada **break** para evitar que se ejecute el próximo bloque.

```
{}
switch (expresión) {
    case valorA:
        // código a ejecutar si la expresión es igual a valorA
        break;
    case valorB:
        // código a ejecutar si la expresión es igual a valorB
        break;
}
```

Agrupamiento de casos

El switch también **nos permite agrupar casos** y ejecutar un mismo bloque de código para cualquier caso de ese grupo.

```
{}
switch (expresión) {
    case valorA:
    case valorB:
        // código a ejecutar si la expresión es igual a ValorA o B
        break;
    case valorC:
        //código a ejecutar si valorC es verdadero
        break;
}
```

5 Arrays y métodos de arrays

DigitalHouse> Coding School

Los **arrays** nos permiten generar una **colección** de **datos ordenados**.

Estructura de un array

Utilizamos corchetes `[]` para indicar el **inicio** y el **fin** de un array. Utilizamos comas `,` para **separar** sus elementos.

Dentro, podemos almacenar la cantidad de elementos que queramos, sin importar el tipo de dato de cada uno.

Es decir, podemos tener en un mismo array datos de tipo string, number, boolean y todos los demás.

```
{}  let miArray = ['Star Wars', true, 23];
```

Posiciones dentro de un array

Cada dato de un array ocupa una posición numerada conocida como **índice**. La **primera posición** de un array es **siempre 0**.

```
{}  let pelisFavoritas = ['Star Wars', 'Kill Bill', 'Alien'];
```

Para acceder a un elemento puntual de un array, nombramos al array y, **dentro de los corchetes**, escribimos el **índice** al cual queremos acceder.

```
{}  pelisFavoritas[2];
    // accedemos a la película Alien, el índice 2 del array
```

Longitud de un array

Otra propiedad útil de los arrays es su longitud, o cantidad de elementos. Podemos saber el número de elementos usando la propiedad length.

```
{}  let pelisFavoritas = ['Star Wars', 'Kill Bill', 'Alien'];
```

Para acceder al total de elementos de **un array**, nombramos al array y, **seguido de un punto .**, escribiremos **la palabra length**.

```
{}  pelisFavoritas.length;
    // Devuelve 3, el número de elementos del array
```


DigitalHouse> Coding School

.push()

Agrega uno o varios elementos al final del array.

- **Recibe** uno o más elementos como parámetros.
- **Retorna** la nueva longitud del array.

```
let colores = ['Rojo','Naranja','Azul'];
colores.push('Violeta'); // retorna 4
console.log(colores); // ['Rojo','Naranja','Azul','Violeta']
{}

colores.push('Gris','Oro');
console.log(colores);
// ['Rojo','Naranja','Azul','Violeta','Gris','Oro']
```

.pop()

Elimina el último elemento de un array.

- **No recibe** parámetros.
- **Devuelve** el elemento eliminado.

```
let series = ['Mad Men','Breaking Bad','The Sopranos'];

// creamos una variable para guardar lo que devuelve .pop()
let ultimaSerie = series.pop();

console.log(series); // ['Mad men', 'Breaking Bad']
console.log(ultimaSerie); // ['The Sopranos']
```

.shift()

Elimina el primer elemento de un array.

- **No recibe** parámetros.
- **Devuelve** el elemento eliminado.

```
let nombres = ['Frida','Diego','Sofía'];

// creamos una variable para guardar lo que devuelve .shift()
let primerNombre = nombres.shift();

console.log(nombres); // ['Diego', 'Sofia']
console.log(primerNombre); // ['Frida']
```

.unshift()

Agrega uno o varios elementos al principio de un array.

- **Recibe** uno o más elementos como parámetros.
- **Retorna** la nueva longitud del array.

```
let marcas = ['Audi'];

marcas.unshift('Ford');
console.log(marcas); // ['Ford', 'Audi']

marcas.unshift('Ferrari','BMW');
console.log(marcas); // ['Ferrari','BMW','Ford', 'Audi']
```

.join()

Une los elementos de un array utilizando el separador que le especifiquemos. Si no lo especificamos, utiliza comas.

- **Recibe** un separador (string), **es opcional**.
- **Retorna** un string con los elementos unidos.

```
let dias = ['Lunes','Martes','Jueves'];

let separadosPorComa = dias.join();
console.log(separadosPorComa); // 'Lunes,Martes,Jueves'

let separadosPorGuion = dias.join(' - ');
console.log(separadosPorGuion); // 'Lunes - Martes - Jueves'
```

.indexOf()

Busca en el array el elemento que recibe como parámetro.

- **Recibe** un elemento a buscar en el array.
- **Retorna** el primer índice donde encontró lo que buscábamos. Si no lo encuentra, retorna un -1.

```
{}
let frutas = ['Manzana','Pera','Frutilla'];
frutas.indexOf('Frutilla');
// Encontró lo que buscaba. Devuelve 2, el índice del elemento

frutas.indexOf('Banana');
// No encontró lo que buscaba. Devuelve -1
```

.lastIndexOf()

Similar a `.indexOf()`, con la salvedad de que empieza buscando el elemento por el **final del array** (de atrás hacia adelante).

En caso de haber elementos repetidos, devuelve la posición del primero que encuentre (o sea el último si miramos desde el principio).

```
let clubes = ['Racing','Boca','Lanús','Boca'];

clubes.lastIndexOf('Boca');
{}  // Encontró lo que buscaba. Devuelve 3

clubes.lastIndexOf('River');
// No encontró lo que buscaba. Devuelve -1
```

.includes()

También similar a `.indexOf()`, con la salvedad que retorna un booleano.

- **Recibe** un elemento a buscar en el array.
- **Retorna** *true* si encontró lo que buscábamos, *false* en caso contrario.

```
let frutas = ['Manzana','Pera','Frutilla'];

frutas.includes('Frutilla');
{}  // Encontró lo que buscaba. Devuelve true

frutas.includes('Banana');
// No encontró lo que buscaba. Devuelve false
```

.map()

Este método recibe una función como parámetro (callback).

Recorre el array y devuelve un nuevo array modificado.

Las modificaciones serán aquellas que programemos en nuestra función de callback.

```
{}
array.map(function(elemento){
   // definimos las modificaciones que queremos
   // aplicar sobre cada elemento del array
});
```

.reduce()

Este método recorre el array y devuelve un **único valor**.

Recibe un callback que se va a ejecutar sobre cada elemento del array. Este, a su vez, recibe dos parámetros: un **acumulador** y el **elemento actual** que esté recorriendo.

```
{}
array.reduce(function(acumulador, elemento){
    // definimos el comportamiento que queremos
    // implementar sobre el acumulador y el elemento
});
```

6 Strings y métodos de strings

“

Para JavaScript los strings son como un array de caracteres.

Por esta razón disponemos de **propiedades** y **métodos** muy útiles a la hora de trabajar con la información que hay adentro.

”

Los strings en JavaScript

En muchos sentidos, para JavaScript, un **string** no es más que un **array de caracteres**. Al igual que en los arrays, la primera posición siempre será 0.

```
{}  let nombre = 'Fran';
```

Para acceder a un carácter puntual de un string, nombramos al string y, **dentro de los corchetes**, escribimos el **índice** al cual queremos acceder.

```
{}  nombre[2];
    // accedemos a la letra a, el índice 2 del string
```

.length

Esta **propiedad** retorna la **cantidad total de caracteres** del string, incluidos los espacios.

Como es una propiedad, al invocarla, no necesitamos los paréntesis.

```
let miSerie = 'Mad Men';
miSerie.length; // devuelve 7

let arrayNombres = ['Bart', 'Lisa', 'Moe'];
arrayNombres.length; // devuelve 3

arrayNombres[0].length; // Corresponde a 'Bart', devuelve 4
```

.indexOf()

Busca, en el string, el string que recibe como parámetro.

- **Recibe** un elemento a buscar en el array.
- **Retorna** el primer índice donde encontró lo que buscábamos. Si no lo encuentra, retorna un -1.

```
{}
```

```
let saludo = '¡Hola! Estamos programando';

saludo.indexOf('Estamos'); // devuelve 7
saludo.indexOf('vamos'); // devuelve -1, no lo encontró
saludo.indexOf('o'); // encuentra la letra 'o' que está en la posición 2, devuelve 2 y corta la ejecución
```

Ya vimos antes que una función es un bloque de código que nos permite agrupar funcionalidad para usarla muchas veces.

Cuando una **función le pertenece a un objeto**, en este caso nuestro string, la llamamos **método**.

DigitalHouse> Coding School

.slice()

Corta el string y devuelve una parte del string donde se aplica.

- **Recibe** 2 números como parámetros (pueden ser negativos):
  - El índice desde donde inicia el corte.
  - El índice hasta donde hacer el corte (es opcional).
- **Retorna** la parte correspondiente al corte.

```
let frase = 'Breaking Bad Rules!';

frase.slice(9,12); // devuelve 'Bad'
frase.slice(13); // devuelve 'Rules!'
frase.slice(-10); // ¿Qué devuelve? ¡A investigar!
```

.trim()

Elimina los espacios que estén al principio y al final de un string.

- **No recibe** parámetros.
- No quita los espacios del medio.

```
{}
let nombreCompleto = '   Homero Simpson   ';
nombreCompleto.trim(); // devuelve 'Homero Simpson'

let nombreCompleto = '   Homero   J.   Simpson   ';
nombreCompleto.trim(); // devuelve 'Homero   J.   Simpson'
```

.split()

Divide un string en partes.

- **Recibe** un string que usará como separador de las partes.
- **Devuelve un array** con las partes del string.

```
let cancion = 'And bingo was his name, oh!';

cancion.split(' ');
{} // devuelve ['And', 'bingo', 'was', 'his', 'name,' , 'oh!']

cancion.split(', ');
// devuelve ['And bingo was his name', 'oh!']
```

.replace()

Reemplaza una parte del string por otra.

- **Recibe** dos strings como parámetros:
  - El string que queremos buscar.
  - El string que usaremos de reemplazo.
- **Retorna** un nuevo string con el reemplazo.

```
{}
let frase = 'Aguante Phyton!';
frase.replace('Phyton', 'JS'); // devuelve 'Aguante JS!'
frase.replace('Phy', 'JS'); // devuelve 'Aguante JSton!'
```

7 | JSON y objetos literales

El objeto literal y JSON – Estructura

JSON es el acrónimo de JavaScript Object Notation y, como su nombre lo indica, es muy similar al objeto literal que ya conocemos. Veamos las diferencias:

Objeto Literal	JSON
Admite comillas simples y dobles.	Las claves van entre comillas.
Las claves del objeto van sin comillas.	Solo se pueden usar comillas dobles.
Podemos escribir métodos sin problemas.	No admite métodos, solo propiedades y valores.
Se recomienda poner una coma en la última propiedad.	No podemos poner una coma en el último elemento.

El objeto literal y JSON — Código

JSON admite la mayoría de los tipos de datos de JavaScript. Veamos cómo sería la conversión entre ambos formatos.

JavaScript nos proporciona un **objeto nativo JSON** con dos métodos que **nos permiten convertir el formato** de un archivo JSON a objeto literal o array, y viceversa.

JSON.parse()

Convierte un texto con formato JSON al tipo de dato equivalente de JavaScript.

- **Recibe** una cadena de texto con formato **JSON**.
- **Devuelve** el mismo dato que recibió en formato **JavaScript**.

```
{}
```

```
let datosJson = '{"club": "Independiente", "barrio": "Avellaneda"}';
let datosConvertidos = JSON.parse(datosJson);

console.log(datosConvertidos);
// Se verá en consola un objeto literal
// { club: 'Independiente', barrio: 'Avellaneda' }
```

JSON.stringify()

Convierte un tipo de dato de JavaScript en un texto en formato JSON.

- **Recibe** un tipo de dato de **JavaScript**.
- **Devuelve** una cadena de texto con formato **JSON**.

```
{}
```

```
let objetoLiteral = { nombre: 'Carla', pais: 'Argentina' };
let datosConvertidos = JSON.stringify(datosObjeto);

console.log(datosConvertidos);
// Se verán en consola los datos en un string de tipo JSON
// '{ "nombre": "Carla", "pais": "Argentina" }'
```


DigitalHouse>
Coding School