**MÓDULO 5**

Los arreglos o arrays son otro tipo de datos dentro del mundo de Javascript, como los números, los booleanos, los Strings. Puntualmente, los arrays nos permiten guardar y gestionar información de gran utilidad cuando querramos almacenar muchos datos. Por ejemplo, nombres de usuarios, las compras de supermercado o cualquier otra información que se pueda guardar como un conjunto de datos. Podemos pensar los arreglos, como una lista de datos a la que podemos acceder en cualquier momento. Todas las listas tienen distintas posiciones en las que se puede guardar un dato, esto mismo es lo que nos permiten hacer los arreglos, guardar y organizar la información. Dentro de los arreglos, existen dos elementos que pueden parecer muy similares, pero son muy diferentes; por un lado, tenemos lo que se conoce como **elemento**. Los elementos son aquellos datos que hayamos guardado por lo que si tuviéramos esta lista (tomates, lechuga, zanahoria), tendríamos un total de 3 elementos.

Por otro lado, está el concepto de **índice**, los índices no representan el dato en sí mismo, sino, la posición en la que se está guardando. Un detalle muy importante, es que, en los arreglos, siempre se comienza a contabilizar las posiciones desde el número 0, no desde el 1. Por lo que si tenemos esta lista (tomate, lechuga, zanahoria) sabemos que el índice de la lechuga es el 1.

Habiendo entendido la diferencia entre elemento e índice, podamos darnos cuenta que siempre, siempre pero siempre, la cantidad de elementos va a ser un número mayor que la cantidad de índices.

¿Cómo podemos crear un arreglo?

Como en cualquier otro tipo de dato, vamos a utilizar la palabra reservada *var* seguida del nombre con el que queremos identificar nuestro arreglo.

*Var listaDeCompras = [];*

Para agregar productos a esta lista, vamos a asignar valores dentro de un arreglo. Lo primero que tenemos que hacer es definir en qué índice queremos que se guarde, supongamos que queremos guardar el string “tomates” en el índice 3.

*Var listaDeCompras = [];*

*listaDeCompras[3] = “Tomates”;*

*console.log(listaDeCompras); // [ < 3 empty ítems >, “Tomates”]*

Hay 3 espacios vacíos, los índices de esos espacios lógicamente son el 0, 1 y 2; ya que en la posición 3 se encuentran nuestros tomates. Ahora, antes del console log que creamos, vamos a agregar lechuga en la posición número 1:

*Var listaDeCompras = [];*

*listaDeCompras[3] = “Tomates”;*

*listaDeCompras[1] = “Lechuga”;*

*console.log(listaDeCompras); // [ < 1 empty ítem >, “Lechuga”, < 1 empty ítem >, “Tomates”]*

¿Cómo recuperarlos?

Para que podamos acceder al elemento de un arreglo, primero tendremos que saber, cuál es su índice. Siguiendo con nuestro arreglo de la lista de compras, si queremos acceder al elemento que está en el índice 1, tendríamos que escribir:

*Var listaDeCompras = [];*

*listaDeCompras[3] = “Tomates”;*

*listaDeCompras[1] = “Lechuga”;*

*listaDeCompras[1];*

Lo que también podemos hacer, es guardar este valor dentro de una variable, ya hacer un console log para ver el resultado. Para esto, escribamos lo siguiente:

*Var elementoDelArray = listaDeCompras [1];*

*Console.log(elementoDelArray); = // “Lechuga”*

Para **recuperar** el elemento de un arreglo, solo tenemos que escribir, el nombre del arreglo y, entre corchetes, el índice que queremos recuperar.

El método más utilizado en los arreglos es **length**, nos permite saber cuántos elementos tiene un arreglo, es decir, cuál es su extensión. Justamente por eso su traducción del inglés, es longitud.

Si nosotros queremos saber rápidamente cuantos elementos tiene nuestro arreglo dentro, simplemente, le aplicaremos el método length al final.

*Var listaDeCompras = [];*

*listaDeCompras[3] = “Tomates”;*

*listaDeCompras[1] = “Lechuga”;*

*console.log(listaDeCompras.length); = // 4*

Un arreglo funciona muy parecido a lo que es una lista, contiene elementos o datos por un lado e índices o posiciones por otro. Dentro de los arreglos podemos almacenar y acceder a un dato en cualquier posición. El método length nos permite saber rápidamente cuantos elementos contiene un arreglo.

Los arreglos o arrays son lo que se conoce como objeto global dentro de Javascript, y que nos permiten guardar y gestionar información. Esto nos será de gran utilidad cuando queramos almacenar datos.

Dentro de los arreglos existen dos conceptos que pueden parecer similares, pero son muy diferentes.

1. Por un lado, tenemos lo que se conoce como elemento. Los elementos son aquellos datos que hayamos guardado.
2. Por otro lado, está el concepto de índice. Los índices no representan al dato en sí mismo, sino la posición en la que se está guardando. Un detalle muy importante que hay que destacar es que, en los arreglos, siempre se comienza a contabilizar las posiciones desde el número cero, no el uno.

**MÉTODOS DE ARRAYS:**

Los **arreglos**, además de ser un tipo de dato, también son un objeto global dentro de JavaScript. Esto significa que todos los arreglos están asociados a una serie de **métodos**.

Se conoce como método a aquellas funciones nativas o preestablecidas en los lenguajes de programación. Estas funciones son útiles para ahorra líneas de código, y realizar tareas de manera más simple. Una de las tareas más repetitivas en los arreglos, es agregar elementos en ellos.

Existe un método que nos ayuda a agregar elementos de forma rápida:

**Agregar un elemento al último lugar del arreglo: MÉTODO .push**

*Var colores = [“amarillo”, “azul”];*

*Colores.push(“rojo”);*

*Console.log(colores); = // [“amarillo”, “azul”, “rojo”]*

**Agregar un elemento al principio del arreglo: MÉTODO .unshift**

*Var colores = [“amarillo”, “azul”];*

*Colores.push(“rojo”);*

*Colores.unshift(“verde”);*

*Console.log(colores); = // [“verde”, “amarillo”, “azul”, “rojo”]*

**Eliminar el último elemento de nuestro arreglo: MÉTODO .pop**

*Var colores = [“amarillo”, “azul”];*

*Colores.push(“rojo”);*

*Colores.unshift(“verde”);*

*Colores.pop();*

*Console.log(colores); = // [“verde”, “amarillo”, “azul”]*

**Eliminar el primer elemento del arreglo: MÉTODO .shift**

*Var colores = [“amarillo”, “azul”];*

*Colores.push(“rojo”);*

*Colores.unshift(“verde”);*

*Colores.pop();*

*Colores.shift();*

*Console.log(colores); = // [ “amarillo”, “azul”]*

**MÉTODO .includes:** Este método nos sirve para determinar si un arreglo incluye o no un elemento; este método nos va a devolver un valor booleano.

*Var pintores = [“Picasso”, “Velazquez”, “Van Gogh”, “Dali”]*

*Var incluyeDali = pintores.includes(“Dali”);*

*Console.log(incluyeDali); = // true;*

**MÉTODO .every:** Nos va a permitir saber si absolutamente todos los elementos de un arreglo cumplen con una condición. Supongamos que tenemos un arreglo de números, y queremos saber si todos esos números cumplen con la condición de ser mayores a 5.

*Var números = [1, 6, 8, 9];*

*Var cumplenCondicion = Números.every((num) => {*

*Return num > 5;*

*});*

*Console.log(cumplenCondicion); = // false;*

Dentro de los paréntesis vemos una función flecha, y especificaremos el parámetro con el que trabajaremos, en este caso, lo llamaremos *num.* Una vez hecho esto, solo nos queda decirle a la función, cual es la condición que deben cumplir todos los elementos, en este caso, era que sean mayores a 5, así que tendremos que escribir num > 5. Guardemos ahora, este valor en una variable (cumplenCondicion).

**MÉTODOS DE INSERCIÓN**

1. El método push() añade uno o más elementos al final de un arreglo, y devuelve la nueva longitud del array.
2. El método pop() elimina y devuelve el último elemento de un arreglo.
3. El método unshift() agrega uno o más elementos al inicio de un arreglo, y devuelve la nueva longitud del array.
4. El método shift() elimina y devuelve el primer elemento de un arreglo.

**Método split y join:** Split = separar; Join = unir.

Estos métodos nos permiten transformar Strings en arreglos y arreglos en Strings.

Supongamos que tenemos el string “Henri” con la última letra siendo una i latina. Pero Henry, se escribe con la letra “y”, para poder corregir esto, lo primero que vamos a hacer es convertir nuestro string en un arreglo, para eso, usaremos el método split:

*Var palabra = “Henri”*

*Var palabraSeparada = palabra.split (‘‘)*

*Console.log(palabraSeparada); = // [‘H’, ‘e’, ‘n’, ‘r’, ‘i’]*

Dentro de los paréntesis hemos puesto dos comillas juntas, esto indica al método que queremos que separe la palabra por cada carácter. Dentro de los paréntesis hay dos comillas vacías, esto le indica al método que queremos que separe la palabra por cada carácter. Si dejamos, por ejemplo, un espacio vacío entre las comillas, va a separar el string en cada espacio que haya. Si escribiéramos la letra “a”, va a separar la palabra cada vez que encuentre una letra “a”.

Ahora nuestra palabra, se convirtió en un arreglo, donde cada elemento es una de las letras de la palabra Henri.

Nosotros queríamos corregir el problema de la última letra, entonces para esto podríamos, eliminar la última letra y reemplazarla con la nueva. Lo hacemos con los métodos pop y push.

*Var palabra = “Henri”*

*Var palabraSeparada = palabra.split (‘‘)*

*palabraSeparada.pop();*

*palabraSeparada.push(‘y’);*

*Console.log(palabraSeparada); = // [‘H’, ‘e’, ‘n’, ‘r’, ‘y’]*

Ya tenemos la palabra “Henry” bien escrita, ahora solo queda que la convirtamos en un string. Para esto, utilizaremos el método join:

*Var palabra = “Henri”*

*Var palabraSeparada = palabra.split (‘‘)*

*palabraSeparada.pop();*

*palabraSeparada.push(‘y’);*

var palabraArreglada = palabraSeparada.join(‘‘);

al igual que en el método split indicamos dentro de los paréntesis, dos comillas juntas, así junta todos los caracteres y no deja espacios vacíos.

Veamos métodos que nos van a dejar recorrer arreglos, es decir, que nos van a dejar ir elemento por elemento y realizar alguna acción en cada uno. El primero que vamos a ver, se llama forEach que su traducción es “para cada uno”. Si, por ejemplo, tuviéramos un array con números del 1 al 4., y quisiéramos hacer un console log por cada uno de los números por separado. Podríamos escribir: números.forEach por cada número console log num. Si ejecutamos el archivo, vemos que la terminal nos imprime cada uno de los números de manera individual:

*Var números = [1, 2, 3, 4];*

*Números.forEach((num => console.log(num));*

Esto mismo, lo podríamos repetir con cualquier cosa, por ejemplo: si quisiéramos que haga un console log solo con aquellos números que sean iguales a 3 deberíamos escribir lo siguiente:

*Números.forEach(num => {if(num === 3) (console.log(num)})*

Números for each por cada num, if num === 3, console log num

Es importante que entendamos que este método simplemente realiza alguna acción sobre el arreglo que especificamos pero, no hará ningún cambio sobre el mismo.

Si quisiéramos realizar algún cambio sobre el arreglo, podemos utilizar un método llamado **map**. Por ejemplo, si quisiéramos sumarle 1 a todos los números del arreglo, simplemente tenemos que escribir:

*Var números = [1, 2, 3, 4];*

*Var masUno = números.map(num => {return num + 1})*

*Console.log(masUno); = // 1 2 3 4*

Los métodos split y join nos permiten transformar Strings en arreglos y viceversa. Por otro lado, el método forEach, nos permite recorrer elemento por elemento y el método map, lo mismo, pero con una leve diferencia, éste crea una copia del arreglo original y le puede realizar cambios.

**MÉTODOS VARIOS**

1. El método includes() determina si un arreglo incluye o contiene un elemento específico. Devuelve true o false en cada caso.
2. El método every() determina si todos los elementos en un arreglo satisfacen una misma condición.
3. El método split() convierte un string en un arreglo, donde cada elemento contendrá un sub-string, dependiendo del parámetro divisor que indiquemos.
4. El método join() convierte un arreglo en un string, uniendo todos los elementos de este en una misma cadena.

**MÉTODOS DE RECORRIDO**

1. El método forEach() nos permite recorrer un arreglo, realizando alguna acción en para cada elemento.
2. El método map() también nos permite recorrer un arreglo y realizar una acción por cada elemento. La diferencia es que este método devuelve un nuevo arreglo los elementos modificados.

**BUCLES O CILOS EN LOS ARRAYS**

Recordemos que un bucle es una secuencia de instrucciones a la que se le asigna una condición. Este código se ejecutará repetidas veces hasta que la condición asignada deje de cumplirse. Pensemos por ejemplo en lavar los platos.

¿Cuáles serían las instrucciones?: Agarrar el plato sucio, pasarle jabón, mojarlo y secarlo.

¿Cuál sería la condición? Limpiar todos los platos

Por lo tanto, repetiremos estas instrucciones con cada plato, hasta que estén todos los platos limpios.

Ahora, veremos que existen distintos tipos de bucles: aprenderemos dos bucles que nos ofrece Javascript de forma nativa para que trabajemos en nuestro código:

**Bucle FOR:** Como en cualquier otra declaración de JS tendremos una palabra reservada que indicará que estamos trabajando con este tipo de bucles. Esta palabra es **for** por lo que la escribiremos con sus paréntesis y sus llaves.

Recordar: cuando utilizamos una función o un condicional if podríamos recibir datos por parámetros, estos datos nos los pasaban entre paréntesis. Si bien en el bucle for hay paréntesis, no sirven para recibir parámetros sino, que los usaremos para otra cosa.

El paréntesis va a estar compuesto por 3 datos separados por ; Cada dato tendrá una función anidada en específico.

Cuando trabajamos con un bucle for, siempre declararemos una variable de iteración. La variable de iteración nos permitirá tener un nodo para trabajar en cada iteración del bucle, en este caso, podríamos llamar a la variable con la letra i. La primera iteración siempre será la del número 0. Finalmente, escribimos un punto y coma para indicar que a continuación viene el siguiente dato.

Para definir el segundo dato, antes hay que preguntarnos: ¿Qué queremos iterar? Siempre vamos a iterar arreglos, por lo que tenemos que decidir que arreglo será. Una vez que lo hayamos decidido, nos preguntamos: ¿Mientras se cumple qué condición quiero que este bucle se itere? Esto nos permite para una sentencia de cierre a nuestro bucle.

Para ver esto más claramente, tenemos un arreglo con 5 números y queremos iterar sobre este mismo, entonces tendríamos que escribir i < arr.length; De esta manera, sabemos que mientras i sea menor a la cantidad de elementos que tenga el arreglo, éste bucle se estará ejecutando. Pero cuando el número de iteración sea mayor a la cantidad de elementos el bucle se romperá.

Puedes ver que la variable i representa el índice del elemento sobre el cual se está iterando. Si en lugar de querer iterar sobre todos los elementos, queremos iterar solo los primeros 3 elementos, entonces hay que escribir i < 3. De esta forma, el bucle solo se ejecutará en el primer elemento y segundo elemento.

Luego de definir el período de ejecución del bucle tendremos que indicar qué sucederá con la variable de iteración al final de cada iteración. (Nosotros queremos que para recorrer elemento por elemento la variable de iteración sea un número mayor en cada iteración por eso escribiremos i++ para indicar que se sume 1 a la variable i).

Cuando recorremos un arreglo, lo hacemos para realizar algo en cada elemento, para poder realizar una acción sobre cada elemento, necesitamos alguna forma para referirnos a él. Esa forma será el nombre del arreglo que estemos recorriendo más corchetes con la variable de iteración dentro de ellos. Si quisiéramos hacer un console log de todos los elementos de un arreglo, simplemente escribiríamos console.log(arr[i])

*Var arr = [1, 2, 3, 4, 5];*

*For (var i= 0; i < arr.length; i++) {*

*Console.log(arr[i])*

*}; = // 1, 2, 3, 4, 5*

Agregamos un poco más de lógica, supongamos que alguien nos pide que construyamos una función que reciba un string y revises alguna de las letras de este string en la letra “p”. Para esto, declaramos una nueva función llamada encontrar letra p, esta función va a recibir por parámetro un string (así que escribimos string entre los paréntesis). Ahora, ¿Qué es lo primero que deberíamos hacer? Para poder recorrer cada una de las letras y poder ver si son iguales a P. Tendríamos que transformar el string en un arreglo, y esto, lo podríamos hacer con un método split. Así que esta será nuestra primera línea dentro de la función. Ahora que tenemos nuestro arreglo con todas las letras del string, nos toca recorrerlo. Primero, escribimos la palabra reservada for junto con sus paréntesis, ahora declararemos la variable de iteración let i = 0, luego de esto escribimos la función que se debe cumplir para que se ejecute el bucle.

En este caso queremos recorrer un arreglo por lo que la condición es que, aún haya elementos para recorrer, así que escribimos que mientras i sea menor a letras.length (la cantidad de elementos) queremos que éste se siga ejecutando. Ahora solo nos queda aumentar nuestra variable de iteración (i++). YA PODEMOS RECORRER NUESTRO ARREGLO.

Ahora solo nos queda encontrar la letra p. Para esto, vamos a codear un condicional que pregunte si la letra en la que se está iterando es igual a P. En caso de que sea sí, que nos avise.

*Function encontrarLetraP(string) {*

*Var letras = string.split (‘’);*

*For (let i = 0; i < letras.length; i++) {*

*If (letras[i] === ‘p’) {*

*Console.log(‘Si contiene a P’);*

*}*

*}*

*}*

**BUCLE WHILE:** Este bucle utiliza la palabra clave **while** seguido tiene los paréntesis en los que escribiremos la condición que se debe cumplir para que este bucle se ejecute. Esto nos permitirá repetir la misma tarea muchas veces en pocas líneas de código. Por ejemplo, si quisiéramos agregar 5 veces la palabra ‘BOOM’ en un arreglo, podríamos decir que la condición es que el arreglo, tenga una longitud de 5.

*Var arr = [];*

*While (arr.length < 5) {*

*Arr.push(‘BOOM’);*

*}*

*Console.log (arr); // = ‘BOOM’ ‘BOOM’ ‘BOOM’ ‘BOOM’ ‘BOOM’*

Este bucle while fue agregando el string BOOM uno por uno, pero cuando el arreglo, ya tenía 5 elementos, es decir, cuando el arreglo tenía una longitud de 5, la condición se dejó de cumplir, por lo que se dejaron de agregar Strings al arreglo.

Aprendimos a utilizar el bucle while y el for. Y dejamos por sentado la principal diferencia entre ambos, es la expresión de control que le permite a cada uno mantenerse ejecutando. El bucle **for** trabaja con un número determinado de repeticiones, por ejemplo, para recorrer un arreglo sabemos que el bucle se repetirá una vez por cada elemento que el arreglo tenga. Por otro lado, el bucle while trabaja con condiciones, mientras la condición que especifiquemos se cumpla el bucle seguirá ejecutándose.

Bucle For: Crea un bucle que consiste en tres expresiones opcionales, encerradas en paréntesis y separadas por puntos y comas, seguidas de una sentencia ejecutada en un bucle.

Sintaxis: for ([expresion-inicial]; [condicion]; [expresion-final])sentencia

Bucle While: Crea un bucle que ejecuta una sentencia especificada mientras cierta condición se evalúe como verdadera. Dicha condición es evaluada antes de ejecutar la sentencia.

Sintaxis: while (condicion)

Sentencia

Bucle infinito: Esto no es un nuevo tipo de bucle, sino algo que se puede producir en los que ya aprendimos. Un bucle infinito se produce cuando un bucle comienza a ejecutarse, pero jamás termina. Esto se debe a que nunca se llega a la condición de quiebre.

Esto consume muchos recursos de la computadora, por lo que hay que intentar evitarlos.

**MÓDULO 6: INTRODUCCIÓN A LOS OBJETOS**

Los objetos son una estructura en la que se puede guardar todo tipo de información e incluso funcionalidades. Pero la principal diferencia, es la forma en la que organizamos esa información ya que nos permiten ser más específicos. Como se puede ver, los objetos se declaran como cualquier otro tipo de dato, comenzamos por la palabra clave **var**, luego con el nombre que queremos identificar el objeto, y finalmente, igualamos esto a dos llaves vacías. Las llaves vacías son las que nos indican que esto es un objeto.

*Var deportes = {};*

¿Cómo guardar información dentro de los objetos?

Supongamos que tenemos el nombre de muchos deportes distintos, en algunos de estos, se utiliza un balón y en otros, no. Así que vamos a clasificar esta información, dentro de nuestro objeto.

Primero, guardemos los deportes que sí necesitan un balón, para esto, vamos a escribir una **propiedad** que se llame conBalon, esto será igual a un arreglo, y este arreglo escribiremos:

*Var deportes = { conBalon: [ “football”, “Basketball”, Rugby”] }*

Ahora, vamos a crear una nueva **propiedad** que se llame sinBalon, y va a ser igual a otro arreglo:

*Var deportes = { conBalon: [ “football”, “Basketball”, Rugby”] , sinBalon: [“Boxeo”, “surf”, “tracking” ] }*

Aquí podemos identificar que los objetos tienen una estructura con un modo **par clave:valor**. Por un lado, tenemos la clave o propiedad (con/sinBalon), y por el otro lado, su valor (lo que está entre corchetes). Todos los objetos respetarán siempre esta regla. Una característica importante, es que los objetos pueden tener todas las propiedades que nosotros querramos.

Ahora centrémonos en algo específico, los valores. En este caso, los valores de cada propiedad son arreglos, pero podrían ser cualquier otro tipo de dato.

Supongamos que tenemos que reunir la información de una persona: para esto, vamos a crear un objeto llamado persona. Dentro de este nuevo objeto vamos a guardar una nueva propiedad:

*Var persona = { nombre: “Lucas”, edad: 26, estudios: {esProgramador: true} };*

En esta última propiedad, vemos que podemos guardar un objeto, dentro de otro objeto.

A diferencia de las matrices que tienen elementos valorados en índices, los objetos usan un concepto llamado pares de clave:valor.

La clave (key) es el identificador y el valor (value) es el valor que queremos guardar en esa clave. La sintaxis es "clave: valor". Los objetos pueden contener muchos pares de clave-valor, deben estar separados por una coma (importante: sin punto y coma dentro de un objeto).

Las claves son únicas en un objeto, solo puede haber una clave de ese nombre, auqneu, varias claves pueden tener el mismo valor. Los valores pueden ser cualquier tipo de dato de JS; cadena, número, booleano, matriz (array), función o incluso, otro objeto.

**TRABAJANDO CON OBJETOS**

Comencemos por dos conceptos básicos: **Dot-notation** o notación por punto y **bracket-notation** o notación por corchetes. Estas son las dos maneras en las que se pueden trabajar con objetos.

Hasta ahora, sabemos que los objetos se basan en una estructura con forma key:value, por lo que primero vamos a aprender a asignar tanto propiedades como valores:

Supongamos que queremos saber cuál es la edad de la persona, para esto:

*Var persona = { nombre: “Lucas”, edad: 26, estudios: {esProgramador: true} };*

*Console.log(persona.edad);*

Para acceder a las propiedades de un objeto, simplemente, tenemos que escribir el nombre del objeto seguido de un punto y el nombre de la propiedad.

¿Cómo podemos asignar valores a un objeto?

Para esto, supongamos que nos dimos cuenta que esta persona, en realidad, no se llama Lucas, sino que se llama Martín. Para corregir este error simplemente escribimos el nombre del objeto seguido de un punto y el nombre de la propiedad, luego, igualamos este al nuevo nombre. Para cambiar el valor de una propiedad, simplemente, tenemos que acceder a ella e igualarla al nuevo valor.

¿Cómo podemos crear propiedades?

Para esto, tenemos que pensar en qué información necesitamos. Por un lado, tenemos que saber cuál va a ser el nombre de la propiedad y por otro, cual va a ser si valor. Tengamos en cuenta, que no podemos crear una propiedad vacía, siempre debe tener algún valor asignado.

Para entender esto, creemos un objeto vacío llamado autos. Debajo de este objeto, vamos a crear una propiedad llamada marcas que será igual a un arreglo.

*Var autos= {};*

*Autos.marcas = [“Ford”, “Audi”, “Ferrari”];*

Si hacemos un console log de esto, vemos que se creo tanto el valor como la propiedad.

La sintaxis para crear una propiedad es igual a la que usamos para cambiar el valor de una propiedad. La diferencia está en que cuando usamos esta sintaxis y la propiedad ya existe, solo, vamos a reescribir su valor. En cambio, si la propiedad aún no existe, vamos a estar creándola y asignándole un valor.

¿Cómo eliminar propiedades?

Para esta tarea, utilizaremos una palabra reservada llamada **delete**. Debemos escribir la palabra reservada junto al nombre del objeto y la propiedad que queremos eliminar.

*Var autos= {};*

*Delete autos.marcas*

*Autos.marcas = [“Ford”, “Audi”, “Ferrari”]; = // {}*

En este caso, la propiedad marcas ya no existe, ahora el objeto está completamente vacío.

Existe una particularidad que tiene este tipo de datos: es que, dentro de la propiedad de un objeto, podemos guardar una función, de esta forma, podremos acceder a ella más tarde si así lo deseásemos.

*Var misFunciones ={saludar: Function() {console.log(“Hola”)};*

*misFunciones.saludar(); = // “Hola”;*

1. Para acceder a la propiedad de un objeto simplemente tenemos que escribir el nombre del objeto seguido de un punto y el nombre de la propiedad.
2. Para cambiar el valor de una propiedad simplemente tenemos que acceder a ella e igualarla al nuevo valor.
3. Para eliminar propiedades utilizaremos una palabra reservada llamada delete.

**DOT-NOTATION Y BRACKET-NOTATION:**

Aprenderemos los diferentes tipos de sintaxis que existen para trabajar con objetos.

Forma de sintaxis: Dot-notation

Habíamos visto que para acceder a la propiedad de un objeto podemos simplemente, escribir el nombre del objeto seguido de un punto, y el nombre de la propiedad a la que queremos acceder.

*Var atuendos = { manos: [“Guantes”, “Anillos”], pies: [“Zapatos”, “Soquetes”] };*

*Console.log(atuendos.manos); = // “Guantes, “Anillos”*

Se le llama dot-notation porque estamos utilizando un punto para poder acceder a las propiedades. De la misma manera que utilizamos la notación por puntos para que podamos acceder o asignar un valor también podemos utilizar bracket-notation o notación por corchetes. Lo único que cambia es la forma en la que lo escribimos.

Siguiendo con el ejemplo anterior, intentemos crear una nueva propiedad llamada piernas igual a un arreglo con los Strings bermudas y pantalones. Para esto, vamos a escribir el nombre de nuestro objeto seguido de dos corchetes y dentro, irá el nombre de la nueva propiedad que queremos crear. Esto lo igualaremos al arreglo con los Strings.

Atuendos [“piernas”] = [“Bermudas”, “Pantalones”]

Es importante que esta palabra la escribamos como un string, caso contrario, no funcionará.

Es lo mismo que si utilizáramos dot-notation, la diferencia es que la propiedad va entre corchetes y comillas.

Los Brackets-notation tienen una particularidad que dot-notation no tiene:

Crearemos un objeto vacío, luego crearemos una función que reciba dos parámetros (propiedad 1 y propiedad 2), dentro de esta función haremos dos cosas. Primero, y mediante notación por punto crearemos una propiedad en el objeto comidas llamada con el nombre de la propiedad uno recibida por parámetro, a esta la igualaremos un parámetro con el string frutas y vegetales. Debajo de esto, agregaremos otra propiedad del objeto comidas llamada con el nombre de la propiedad dos recibida por parámetro, pero esta vez, con notación por corchetes. Ahora ejecutaremos la función pasándole como parámetro las palabras saludable y no saludable, así estas serán los nombres de las propiedades del objeto.

*Var comidas = {};*

*Var diferenciaDeNotaciones = Function (propUno, propDos) {*

*Comidas.propUno = [“Frutas”, “Vegetales”];*

*Comidas.[“propDos”] = [“Hamburguesa”, “Papas Fritas”];*

*};*

*diferenciaDeNotaciones(“saludable”, “noSaludable”);*

*console.log(comidas); = // propUno = [“Frutas”, “Vegetales”]; “propDos” = [“Hamburguesa”, “Papas Fritas”];*

El nombre de las propiedades de nuestros objetos no son las que pasamos por parámetro, ¿Por qué? La respuesta es muy sencilla, la forma en la que agregamos las propiedades a nuestro objeto está tomando las palabras propiedad uno y propiedad dos literalmente, y no el valor que nosotros planteamos. Cuando vamos a reducir algún tipo de dato externo como una variable o un parámetro tendremos que hacer una pequeña modificación en la forma de declarar esto. Pero para que esto funcione debemos utilizar bracket-notation. Jamás podremos solucionar este problema utilizando dot-notation

Modifiquemos el ejemplo para que funcione. Lo que tenemos que cambiar es la forma en la que hemos escrito el bracket-notation. Lo único que tenemos que hacer, es eliminar las comillas que están dentro de los corchetes. De esta manera, le estaremos indicando a JS que el nombre de la propiedad será la variable propiedad dos, y no esa palabra de forma literal.

*Var comidas = {};*

*Var diferenciaDeNotaciones = Function (propUno, propDos) {*

*Comidas.propUno = [“Frutas”, “Vegetales”];*

*Comidas.[propDos] = [“Hamburguesa”, “Papas Fritas”];*

*};*

*diferenciaDeNotaciones(“saludable”, “noSaludable”);*

*console.log(comidas); = // propUno = [“Frutas”, “Vegetales”]; “noSaludable” = [“Hamburguesa”, “Papas Fritas”];*

De la misma manera que utilizamos la Dot-Notation o notación por puntos para acceder o asignar un valor, también podemos usar Bracket-Notation, o notación por corchetes. Lo único que cambia es la forma en la que lo escribimos.

Muchas veces nos puede suceder que necesitemos utilizar una variable externa para guardarla como propiedad en un objeto. Es importante que en esos casos recordemos utilizar Bracket-Notation sin comillas para que funcione correctamente.

**OBJETOS AVANZADOS**

Al igual que en los arreglos, dentro de JS existe un objeto global llamado **object** que extiende diferentes métodos a los objetos. Habíamos dicho que los métodos son funciones predefinidas de este lenguaje y que nos permite revisar tareas de manera mucho más rápida.

El primer **método** que vamos a ver, se llama **hasOwnProperty**, la traducción del inglés, quiere decir “tiene su propiedad”, este método nos permitirá saber si un objeto tiene una propiedad específica. La respuesta a esta pregunta es un valor booleano.

Supongamos que tenemos un objeto llamado libro, con las propiedades:

*Var libro = {autor: “Borges”, genero: “Policial”, año: 1990};*

*Var tienePropiedad = Libro.hasOwnProperty(“nombre”);*

*Console.log(tienePropiedad); = // false;*

Si queremos saber si “libro” tiene una propiedad “nombre”, simplemente vamos a escribir el nombre del objeto con el método hasOwnProperty y entre paréntesis, la palaba nombre. Esto, lo igualaremos a una variable “tiene propiedad” y le haremos un console.log

Como vemos, el método hasOwnProperty nos puede ser muy útil cuando tengamos que validar este tipo de información.

Otro método muy utilizado, es ***keys***. Si recuerdan, los objetos se componen por un par clave valor o lo que seria en inglés, key value. Lo que hace el método keys, es devolvernos un arreglo con el nombre de todas las propiedades de un objeto. Primero llamamos al objeto global con su método key y le pasamos como parámetro, nuestro objeto. Esto lo guardamos en una variable llamada “todas las propiedades” y le daremos un console log. Al ejecutar, nos devuelve un arreglo solo con las propiedades que tiene el objeto

*Var libro = {autor: “Borges”, genero: “Policial”, año: 1990};*

*Var tienePropiedad = Libro.hasOwnProperty(“nombre”);*

*Var todasLasPropiedades = Object.keys(libro);*

*Console.log(todasLasPropiedades); = // [“autor”, “genero”, “año”]*

Los objetos también se pueden **recorrer**, para esto utilizaremos un bucle especial, llamado **for in**. Veamos como es su estructura, para comenzar, vamos a declarar un nuevo objeto con el nombre mundo, dentro de este, habrá 3 propiedades. Ahora, vamos a utilizar la palabra reservada **for**, seguido de los paréntesis y llaves; y esto indicará que trabajaremos con un bucle. Una vez hecho esto, vamos a ver que dentro de los paréntesis tendremos que escribir algunos valores y palabras clave. Primero, vamos a declarar la variable de iteración con la que podremos acceder a las propiedades, en este caso, podemos llamar **prop** luego escribimos la palabra reservada **in** seguida del objeto al que queremos recorrer. Ya podemos recorrer nuestro objeto, por cada iteración le diremos a este bucle que primero haga un console log de “Esta es la propiedad” más prop. Luego, otro console log de “Este es el valor” + valor (objeto[variable]). Luego de ejecutar el archivo, podemos ver que en cada iteración de la variable prop representa el nombre de una propiedad. Y si utilizamos bracket notation con esta variable, también pudimos acceder al valor de esa propiedad.

*Var mundo = { continentes: 7, países: 195, océanos: 5}*

*For (var prop* ***in*** *mundo) {*

*Console.log(“Esta es la propiedad: “ + prop);*

*Console.log(“Este es el valor: “ + mudno[prop]);*

*}*

Para concluir este tema, vamos a aclarar un tema que puede generar dudas. La diferencia entre el bucle **for** y el **for in**. Recordemos que el bucle **for** sirve **únicamente para recorrer arreglos**, elemento por elemento. En cambio, el bucle **for in** nos sirve **únicamente para recorrer objetos,** propiedad por propiedad. Estos dos bucles, solo tienen en común, un nombre similar.

En JS existe un objeto llamado **this** que tiene una funcionalidad muy particular. Cuando hablamos del objeto this estamos haciendo referencia a un contexto.

Supongamos que tenemos dos casas, una casa tiene chimenea y es de color verde. En cambio, la otra casa tiene una cochera y es de color azul. Si yo digo “chimenea” voy a estar haciendo referencia al contexto de la primera casa, pero si digo “azul” voy a estar haciendo referencia al contexto de la segunda casa. Esto mismo es lo que marca el objeto “this”, el contexto.

Vamos a crear un objeto llamado mascota y este, va a tener una propiedad animal igual a perro, una propiedad raza igual a ovejero alemán y una propiedad amistoso igual a true y una propiedad dueño igual a María López. Ahora lo que haremos es crear una función dentro del objeto, esta función lo que hará será imprimir en la terminal la raza de este perro. Para esto, creemos una nueva propiedad llamada info que ejecuta una función console log, diciendo “mi perro es un “ + raza. Entonces en teoría, esto debería mostrarnos la raza del perro en la terminal.

*Var mascota = {*

*Animal: “perro”,*

*Raza: “Ovejero alemán”,*

*Amistoso: true,*

*Dueña: “María López”,*

*Info: Function () {*

*Console.log(“Mi perro es un “ + this.raza);*

*},*

*};*

*Mascota.info() = //* La terminal nos avisa que la palabra raza no existe, no está definida. Esto sucede porque nuestra función no está reconociendo el contexto en el que queremos que se ejecute. Para que podamos acceder desde esta función a otra propiedad de este objeto primero debemos informarle a la función que el contexto al que hacemos referencia es un objeto, para esto utilizamos el objeto “this” justo antes de la palabra raza.

= “Mi perro es un Ovejero alemán”

Como pueden ver, la palabra raza esta apuntando a la propiedad de este objeto, es decir, que cuando declaramos la palabra “this” estamos indicando que el contexto es el objeto mascota.

**MÉTODOS DE OBJETOS**

1. El método hasOwnProperty() nos permitirá especificar un nombre, y verificar si este existe como una propiedad dentro de un objeto. En cada caso devolverá true o false.
2. El método Object.keys() devuelve todas las propiedades de un objeto guardadas en orden dentro de un arreglo.

**RECORRIDOS EN OBJETOS**

De la misma manera que podíamos recorrer los elementos de un arreglo, podemos recorrer las propiedades de un objeto junto con sus valores.

El bucle For-In nos permite iterar sobre un objeto utilizando dos variables pivot. Una representa el objeto que recorremos, y la otra la propiedad en la que se está realizando la iteración. Utilizando ambos valores podremos acceder al valor de cada propiedad del objeto.

**CONTEXTOS EN JAVASCRIPT**

Dentro de este lenguaje de programación existe un objeto global llamado this. Este nos permite manipular el contexto en el que las funciones y la información está siendo ejecutada. De esta forma podremos tener un alcance más preciso dentro de nuestro código.

**MÓDULO 7: INTRODUCCIÓN A CLASES**

Existe un concepto muy común en la mayoría de los lenguajes de programación, la clase. En esta video clase aprenderemos qué son y qué beneficios nos pueden brindar en la resolución de problemas. Para comprender que es una **clase** tendremos que definir en primer lugar lo que son las **entidades**. Las entidades nacen como la representación de conceptos del mundo real. Por ejemplo, dentro del concepto de Instagram, podemos encontrar entidades como usuarios y publicaciones. Por otro lado, las **clases** son un conjunto de datos que pertenecen a una entidad, podríamos pensar entonces en las clases como una plantilla o un “modelo” de una entidad.

Ejemplo: en el mundo existen cientos de miles de animales, y especies que comparten características similares. Entonces la clase de un animal podría llevar las propiedades ojos, color y hábitat. Todos los animales compartirán estas propiedades, pero sus valores serán diferentes. El hábitat de un pez no es el mismo que el de un tigre. El color de un cocodrilo, no es el mismo que el de un perro. Las gallinas tienen visión, pero los murciélagos no. Esta clase “animal” nos permite tener un concepto base para representar animales del mundo real, para luego poder crear o modelar con código todo tipo de animales a partir de ella. Algunos compartirán las mismas características y otros no.

Existen dos maneras de declarar las clases: Por un lado, tenemos una sintaxis llamada **función constructora** y por otra, tenemos una función llamada **expresión de clase**, pero ¿Cómo podemos reflejar esto en código?

En el comienzo de JS para declarar una clase se utilizaba la sintaxis conocida como función constructora pero antes de continuar, pensemos en una entidad, podríamos crear la plantilla de un auto para esto, utilizaremos la palabra reservada **Function** seguido del nombre de nuestra clase, un detalle importante que hay que tener en cuenta es que, al declarar una clase, la primera letra de su nombre, siempre tiene que ir en mayúscula. Debemos tener en cuenta también algo importante, es que las clases cuentan con dos tipos de datos, por un lado, tienen propiedades y por otro, al igual que los arreglos, podemos definirles una serie de métodos. Ahora, declaremos primero las propiedades: las propiedades de nuestro auto van a ser: puertas, color, marca, año y ruedas. Estos datos serán recibidos como parámetros por lo que los escribiremos entre paréntesis. Para poder declarar estas propiedades, vamos a utilizar la palabra reservada “this”, seguido del nombre de cada propiedad y las igualaremos a su valor recibido por parámetro. Luego de haber hecho esto solo nos queda inicializar nuestra clase auto, es decir, utilizar esta plantilla. Para esto, vamos a declarar una variable “mi primer auto”, esta variable será igual a la palabra reservada **new** que nos permitirá crear una instancia de nuestra clase. Luego, escribimos el nombre de nuestra clase y entre los paréntesis le pasamos los valores de las propiedades que tendrán nuestra nueva instancia. En este caso, los valores serán: dos, rojo, Ferrari, 2018 y 4. Ahora haremos un console log tanto de la instancia como de la propiedad marca y veremos que nos responde la terminal.

*Function Auto (puertas, color, marca, año, ruedas) {*

*This.puertas = puertas;*

*This.color = color;*

*This.marca = marca;*

*This.año = año;*

*This.ruedas = ruedas;*

*}*

*Var miPrimerAuto = new Auto(2, “Rojo”, “Ferrari”, 2018, 4);*

*Console.log(miPrimerAuto);*

*Console.log(miPrimerAuto.marca);*

*= // Auto {*

*Puertas: 2,*

*Color: “RojoR,*

*Marca: “Ferrari”,*

*“año”: 2018,*

*Rudas: 4*

*}*

*Ferrari*

La primera respuesta nos indica el nombre de la clase y luego, entre llaves toda la información de esta instancia, también veremos que efectivamente nos muestra la marca de nuestro primer auto así entonces, es como podríamos reutilizar nuestra clase Auto como plantilla para la creación de muchos otros Autos con diferente cantidad de puertas, marca, año, etc.

Ahora veremos como agregar métodos a una clase: continuando con el ejemplo anterior. Supongamos que queremos que nuestros autos tengan un método que nos de información acerca de la instancia, para esto nuevamente utilizaremos la palabra “this” seguido de un punto y el nombre que nos ayudará a identificar el método (en este caso, información) esto, va a ser igual a una función que para este ejemplo, solo hará un console log de un string “Este es un” concatenado (+) con la propiedad this.marca concatenado con el string “ de color” y por último, concatenado con la propiedad this.color. Ahora en la siguiente línea de código podemos acceder a esta función desde la instancia “mi primer auto” y ejecutar el archivo.

*Function Auto (puertas, color, marca, año, ruedas) {*

*This.puertas = puertas;*

*This.color = color;*

*This.marca = marca;*

*This.año = año;*

*This.ruedas = ruedas;*

*This.informacion = Function () {*

*Console.log(“este es un” + this.marca + “de color” + this.color);*

*);*

*}*

*Var miPrimerAuto = new Auto(2, “Rojo”, “Ferrari”, 2018, 4);*

*miPrimerAuto.informacion(); = // “Este es un Ferrari de color Rojo”*

Se imprimió en la terminal el mensaje que esperábamos, esto quiere decir que dentro de los métodos de una clase, podemos referirnos a la información de esta misma clase mediante la palabra this para indicar un contexto.

En principio, habíamos dicho que existen dos formas de declarar una clase, la que acabamos de ver se llama “función constructora” que es la forma antigua. Cuando se produjo una actualización en 2015, apareció una nueva manera de declarar las clases llamada **expresión de clase**. Vamos a ver que la lógica es muy similar a la función constructora, solo que esta sintaxis es más sencilla a la vista. Ahora transformemos la función constructora Auto en una función de clase.

Ahora, utilizaremos la palabra reservada **class** seguida del nombre Auto, recordemos que la primera letra siempre va a ir en mayúscula, si se fijan, ahora no estamos agregando paréntesis ya que esto no es una función, como sí lo era antes. Recordemos que las clases tienen dos tipos de datos en su interior: propiedades y métodos. Para declarar las propiedades, tendremos que utilizar la palabra reservada **constructor** seguido de paréntesis y llaves. Las propiedades las recibiremos por parámetro, por lo que las escribiremos dentro del paréntesis, y dentro de las llaves, simplemente las declaramos e igualamos a los valores recibidos. Verifiquemos que esto funciona correctamente. Declaremos una variable con el nombre “mi segundo auto”, de la misma forma que antes, la igualaremos a la palabra reservada **new** seguido del nombre auto, seguido de un paréntesis que nos permitirá agregar las propiedades de este auto, las cuales van a ser: 4, blanco, fiat, 2015 y 4. Luego de esto, hagamos un console log de la instancia que acabamos de crear y de su propiedad marca. Ejecutemos el archivo:

*Class Auto {*

*Constructor(puertas, color, marca, año, ruedas) {*

*This.puertas = puertas;*

*This.color = color;*

*This.marca = marca;*

*This.año = año;*

*This.ruedas = ruedas;*

*}*

*}*

*Var miSegundoAuto = new Auto(4, “blanco”, “fiat”, 2015, 4);*

*Console.log(miSegundoAuto);*

*Console.log(miSegundoAuto.marca);*

*= // Auto {*

*Puertas: 4;*

*Color: “blanco”,*

*Marca: “fiat”,*

*“año”: 2015,*

*Ruedas: 4*

*}*

*Fiat*

La terminal nos respondió, por un lado, con la instancia que creamos con la clase auto y toda su información. Por otro lado, también, nos respondió la palabra “Fiat”. Por lo que nuestra clase, está funcionando correctamente.

¿Cómo le asociamos métodos?

Al ser un método y no una propiedad lo haremos por fuera del constructor. Aquí simplemente, escribimos el nombre del método seguido de paréntesis y llaves. Posteriormente explicitamos, cual será la funcionalidad. En este caso, el nombre del método es “información” y le pediremos que haga un console log del string “Este es un “ concatenado con la propiedad this. marca, concatenado con el string “de color” y por último, concatenado con la propiedad this.color. Para comprobar que esto funciona, simplemente accedemos a este método desde la instancia y lo ejecutamos

*Class Auto {*

*Constructor(puertas, color, marca, año, ruedas) {*

*This.puertas = puertas;*

*This.color = color;*

*This.marca = marca;*

*This.año = año;*

*This.ruedas = ruedas;*

*}*

*Información(){*

*Console.log(“Este es un” + this.marca + “de color” + this.color);*

*}*

*}*

*Var miSegundoAuto = new Auto(4, “blanco”, “fiat”, 2015, 4);*

*miSegundoAuto.informacion();*

Cuando vimos que es un objeto, aprendimos un concepto que estamos utilizando también, la palabra reservada **this**. Habíamos dicho que esta palabra nos permite indicarle al código de qué contexto debe obtener información. Esto lo podemos visualizar utilizando clases. Para entenderlo mejor, veamos un ejemplo:

Declaramos una nueva clase llamada fútbol, esta clase tendrá solo una propiedad y un método. La propiedad que va dentro del constructor se llamará “jugador” y el método “obtener nombre”. Este método solo hará un console log de la propiedad “jugador”. Ahora, crearemos dos instancias de esta clase, una instancia llamada “argentina” a la que le pasaremos como argumento el string “Messi” y otra instancia llamada “Brasil”, pasándole como argumento el string “Pele”

A continuación, ejecutaremos el método obtener nombre de ambas instancias y veremos que nos devuelve la terminal:

*Class futbol {*

*Constructor (jugador) {*

*This.jugador = jugador;*

*}*

*obtenerNombre () {*

*console.log(this.jugador);*

*}*

*}*

*Var argentina = new futbol(“Messi”);*

*Var Brasil = new futbol (“Pele”);*

*Argentina.obtenerNombre ();*

*Brasil.obtenerNombre();* = // Messi, Pele

Cada método, nos devuelve un valor distinto. Por más que se esté accediendo a la misma propiedad. Esto se debe a que el contexto de cada variable declarada es diferente, uno es para argentina y el otro, para Brasil.

**PROTOTIPOS Y HERENCIAS**

Dentro de JS existe una forma de declarar plantillas que nos permiten crear bloques de información para representar una entidad en nuestro código. Estas plantillas se denominan clases. También aprendimos que dentro de las clases podemos guardar dos tipos de información: propiedades y métodos. En esta clase, nos centraremos en detallar y entender un nuevo concepto denominado **prototipo**. Este concepto está muy ligado a los métodos que poseen los objetos y las clases.

Los **prototipos** son un mecanismo por el cual todos los objetos o elementos de JS pueden extender sus propiedades y métodos. Por ejemplo, sabemos que dentro de JS existe un tipo de dato llamado **array** que proviene de un objeto global llamado array. Este objeto global le extiende mediante su prototipo propiedades y métodos al tipo de dato array. Lo mismo ocurre con los objetos, los números, los Strings e incluso con las clases que nosotros podemos crear. El proceso en que los objetos globales de JS le extienden métodos y propiedades a cualquier tipo de dato se denomina **HERENCIA**. Explicándolo de manera más sencilla, supongamos que el creador de JS al inventar los arreglos define que este tendrá un método length, entonces este objeto global le heredará a todos los arreglos el método length. Todos los objetos pueden heredar propiedades y métodos por medio de un prototipo. Gracias a estos prototipos podremos acceder al constructor de cualquier objeto para modificarlo.

En este primer ejemplo, vamos a utilizar el prototipo del objeto global array.

Agregaremos un nuevo método a su constructor, lo que hará este método es transformar todos los números mayores a 3 que tenga un arreglo en el valor booleano false. Primero, debemos escribir la palabra “array” para poder ingresar al objeto global, a continuación, escribimos un punto a continuación de la palabra **prototype**. De esta manera indicamos que queremos ingresar a esta propiedad, al prototipo del objeto global array. Ahora tendremos que especificar el nombre que queremos para este método, el nombre podría ser “mayores que tres”, todo esto será igual a una función. Ahora, solo nos queda crear la lógica. Para poder acceder a los valores que tendrá el arreglo que ejecute este método, vamos a utilizar la palabra “this”. De esta forma, definimos el contexto, por lo que en este caso this equivale al arreglo que ejecuta el método mayor que tres, el método que estamos creando. Entonces, como this equivale ahora a un arreglo, vamos a poder utilizar todas las propiedades y métodos de los arreglos.

Dijimos entonces que queremos que este método nos devuelva un arreglo en el que todos los números mayores a 3, se conviertan en el valor false. Para esto, crearemos una variable que se llame arreglo modificado, que por el momento será, un arreglo vacío. Ahora pensemos, necesitamos saber que números de arreglo son mayores a 3, por lo cual tendremos que recorrerlo, para esto, utilizaremos el bucle for. Diremos que la variable de iteración será igual a 0, queremos que este bucle se ejecute mientras que esta variable sea menor a la longitud del arreglo y por último, la incrementaremos de uno en uno en cada nueva iteración. De esta forma, podremos recorrer todos los elementos del arreglo y verificar cuales son mayores a 3 y cuales no. Esta última pregunta, la podremos hacer con un condicional. If this (this equivale al arreglo que está invocando nuestro método) en la posición del iterador es mayor a 3 entonces, pusheamos el valor false dentro de nuestro “arreglo modificado”. Caso contrario, pusheamos el mismo número sin realizar ningún cambio, ahora, solo nos queda retornar el arreglo que tiene los nuevos valores. (Nuestro método ya fue agregado al constructor del objeto global array). Ahora podemos declarar una variable llamada arreglo que contenga los números del 1 al 5. A este arreglo le aplicaremos el nuevo método que acabamos de crear y guardamos el resultado en una variable llamada “nuevo arreglo”. Por último, hagamos un console log de nuevo arreglo.

*Array.prototype.mayoresQueTres = Function () {*

*Var arregloModificado = [];*

*For(let i = 0; i < this.length; i++) {*

*If(this[i] > 3) {*

*arregloModificado.push(false);*

*} else {*

*arregloModificado.push(this[i]);*

*}*

*}*

*Return arregloModificado;*

*};*

*Var arreglo = [1, 2, 3, 4, 5];*

*Var nuevoArreglo = arreglo.mayoresQueTres();*

*Console.log(nuevoArreglo); = // [1, 2, 3, false, false]*

Un nuevo ejemplo: para esto vamos a crear una nueva clase llamada “Latinoamérica”, esta clase tendrá una única propiedad con el nombre “países” dentro de su constructor igual a un arreglo vacío. Lo que haremos ahora, mediante el prototipo de esta clase crear un método que nos permita agregar el nombre de un país dentro de la propiedad llamada “países”. Por lo que la propiedad países será un arreglo de Strings, para esto escribiremos el nombre de la clase seguido de un punto y la palabra prototype, ahora escribiremos el nombre de este método, por ejemplo lo podemos llamar “agregar país”, esto será igual a una función que recibe por parámetro un país y que finalmente accede a la propiedad países utilizando la palabra this y por medio del método push de los arreglos, le pushea a esta propiedad países el valor recibido por parámetro. Ya creamos nuestro método, ahora nos queda probarlo , para esto vamos a crear una variable con el nombre continente que será igual a una nueva instancia de la clase Latinoamérica, ahora a partir de esta instancia, vamos a acceder al método agregar país y le pasaremos por parámetro la palabra “Mexico”, por última haremos un console log de la propiedad países de nuestra instancia continente y ejecutaremos el archivo.

*Class latinoamerica {*

*Constructor () {*

*This.paises = [];*

*}*

*}*

*Latinoamerica.prototype.agregarPais = Function (país) {*

*This.paises.push(país);*

*};*

*Var continente = new latinoamerica ();*

*Continente.agregarPais(“Mexico”);*

*Console.log(continente.paises);*

Se agregó el país que indicamos en nuestra instancia

Los prototipos son un mecanismo por el cual todos los objetos o elementos de Javascript pueden extender sus propiedades y métodos.

El proceso en el que los objetos globales de JavaScript le extienden métodos y propiedades a cualquier tipo de dato se denomina herencia.

Todos los objetos pueden heredar propiedades y métodos por medio de un prototipo. Gracias a estos prototipos podremos acceder al constructor de cualquier objeto para modificarlo.

**EXTENSIÓN DE CLASE**

Las expresiones de clase de JS pueden participar en la composición de otras clases, es decir, que, a partir de una clase con ciertas propiedades y ciertos métodos, podemos crear otras clases que hereden algunas de estas propiedades o métodos. Para que podamos detallar este tema, demos primero, un repaso acerca de cómo se utilizan las clases.

En un principio habíamos dicho que las clases son como plantillas o modelos que nos van a permitir crear un molde de propiedades y métodos para luego ser utilizado como representación de una entidad en nuestro código.

En este ejemplo, crearemos una clase “persona”, tendrá como propiedades nombre y edad, por lo que las vamos a escribir en su constructor. Además, esta clase tendrá un método llamado “saludar” que simplemente hará un console log del string “Hola, mi nombre es” concatenado con la propiedad this.nombre concatenado con el string tengo y por último, concatenado con la propiedad this.edad. Ya tenemos nuestra clase persona, ahora comprobemos que funciona correctamente. Declararemos una variable llamada “martin” que será igual a una nueva instancia de persona, parámetros serán (“martin” y 26). Ahora ejecutemos el método saludar de esta instancia y veamos que nos devuelve la terminal.

*Class persona {*

*Constructor(nombre, edad) {*

*This.nombre = nombre;*

*This.edad = edad;*

*}*

*Saludar() {*

*Console.log(“Hola, mi nombre es “ + this.nombre + “tengo# + this.edad);*

*}*

*}*

*Var martin = new persona(“martin”, 26);*

*Martin.saludar(); = // Hola mi nombre es martin. Tengo 26*

Este ejemplo muestra como una clase nos permite crear una o muchas instancias de una entidad

Ahora vamos a aprender, **extensión de clase**. Extender clases nos va a permitir tener una clase general que les heredará propiedades y métodos a otras clases. De esta forma podremos generar distintas sub entidades que puedan mantener sus propias características pero que compartan algunas otras generales.

Supongamos que queremos crear una clase llamada “programador”, para esto, vamos a utilizar otra vez, la palabra **class** seguido del nombre “programador”, para hacer la extensión de la clase “persona” a la clase “programador” vamos a utilizar la palabra **extends** seguido de “persona”, una vez hecho esto, tendremos que indicar que propiedades queremos que tenga la clase programador. Para esto, abrimos el constructor y diremos que debe tener las propiedades “nombre, edad y años de experiencia”. Como podemos ver las propiedades nombre y edad, ya existen en la clase persona de la cual nos estamos extendiendo. Pero para que estas propiedades sean heredadas, tendremos que utilizar otra palabra reservada más, esta palabra es **super** y va dentro del constructor. Super le permitirá a esta clase no sólo recibir las propiedades heredadas, sino también, decirles a estas que su nuevo contexto será la clase programador. (Si no lo hiciéramos de esta forma, tendríamos que utilizar la palabra this y escribir muchas líneas de código.) Entre los paréntesis de esta nueva palabra vamos a escribir el nombre de estas dos propiedades que queremos heredar. Ahora estas dos propiedades están siendo heredadas gracias a la clase persona, pero aún nos queda “años de experiencia”. Como esta propiedad no es heredada, la tendremos que declarar como una propiedad interna utilizando la palabra this. Para terminar de construir esta clase vamos a agregarle un método que se llamará “codear”, este método simplemente hará un console log del string “soy” concatenado con this.nombre concatenado con “codeo desde hace” concatenado con la propiedad this.añosDeExperiencia y por último, concatenado con el string “años”. Debajo de la declaración de la variable “Martin”, vamos a crear una nueva instancia, pero de la clase programador, esta instancia se puede llamar “programador” y le pasaremos las propiedades: “María”, 37 y 4. Ahora vamos a ejecutar el método saludar de Martín y el método “codear” de “programador”:

*Class persona {*

*Constructor(nombre, edad) {*

*This.nombre = nombre;*

*This.edad = edad;*

*}*

*Saludar() {*

*Console.log(“Hola, mi nombre es “ + this.nombre + “tengo# + this.edad);*

*}*

*}*

*Class Programador extends Persona {*

*Constructor(nombre, edad, añosDeExperiencia) {*

*Super(nombre, edad);*

*This.añosDeExperiencia = añosDeExperienciaM*

*}*

*Codear (){*

*Console.log(*

*“Soy” +*

*This.nombre +*

*“Codeo desde hace “ +*

*This.añosDeExperiencia +*

*“años”*

*);*

*}*

*}*

*Var Martin = new Persona(“martin”, 26);*

*Var Programador = new Programador(“María, 37, 4);*

*Martin.saludar(); = // Hola mi nombre es martin. Tengo 26*

*Programador.codear(); = // “Soy María. Codeo desde hace 4 años”*

Sabemos que el método saludar pertenece a la clase Persona, ¿Qué pasaría sí intentamos ejecutar desde la instancia Programador este método?

*Programador.saludar(); = //Hola, mi nombre es María. Tengo 37.*

La terminal nos responde lo que habíamos instruido en la clase Persona, pero con la instancia de la clase Programador. Esto se debe a que todos los métodos que posee la clase Persona fueron heredados a la clase Programador.

Extender clases nos puede ser de gran utilidad cuando necesitamos que algunas entidades estén relacionadas en nuestro código; podemos extender más de una vez a cualquier clase e incluso podemos extender una clase de otras clases ya extendidas dependiendo de los problemas que queramos resolver y el contexto de los mismos utilizaremos las extensiones para crear un código más consistente

**MÓDULO 8: CALLBACKS**

Las funciones callbacks son un tipo de funciones que se pasan por parámetro a otras funciones. Supongamos que tenemos una función 1 y recibe por parámetro a otra función 2. Cuando la función 1 ejecuta a la función 2 en su interior, se produce un proceso llamado **callback**. Los callback son muy útiles en el mundo de la programación, nos aseguran que una función no se va a ejecutar antes de que se complete una tarea, sino que se ejecutará justo después de que la tarea se haya completado.

Ejemplo: crearemos una variable llamada “devuelvo usuario”, lo único que hará esta función es retornar el string “Camilo”. Debajo de esto, vamos a crear una función llamada “devuelvo saludo”, esta función únicamente retornará el string “hola”. Por último, vamos a crear una función llamada “saludar”, esta función recibirá dos funciones o callbacks por parámetro y las llamaremos “cb1 y cb2”, dentro de su cuerpo simplemente retornará el resultado de ejecutar cada una de las funciones callback cb1 y cb2 concatenados y separados por un espacio. Ahora crearemos una variable llamada “resultado” y la igualaremos a la ejecución de la función “saludar” pasándole por parámetro la definición de las funciones que creamos anteriormente. Cuando hablamos acerca de la *definición* de una función nos referimos a la función sin ser invocada o ejecutada ya que al utilizarse los paréntesis para invocar una función obtendremos el resultado de su invocación, con lo cual estaremos hablando en términos de el resultado de la función es y en caso de las funciones de callback siempre enviaremos por parámetro la definición de la función, no el resultado de la invocación.

*Var devuelvoUsuario = Function () {*

*Return “Camilo”;*

*};*

*Var devuelvoSaludo = Function () {*

*Return “Hola”;*

*};*

*Var saludar = Function (cb1, cb2) {*

*Return cb1 () + “ “ + cb2();*

*};*

*Var resultado = saludar(devuelvoSaludo, devuelvoUsuario);*

*Console.log(resultado); = // Hola Camilo*

Otro ejemplo: vamos a crear una función llamada “devuelvo frase” y recibirá por parámetro una comida, dentro de su cuerpo, simplemente retornará la frase “Hoy quiero comer:” concatenada con comida. Debajo de esto, vamos a crear una nueva función llamada “hablar” esta función recibe por parámetro una comida y un callback al que llamaremos cb. Dentro de su cuerpo, retornará el callback cb pasándole por parámetro la comida recibida. Ahora solo nos queda crear una nueva variable que llamaremos “frase final”, la cual será igual a la ejecución de hablar pasándole como argumento la palabra “pizza” y la función devuelvo frase la cual será nuestro callback. Por último, hacemos un console log de esta variable y ejecutamos el archivo.

*Var devuelvoFrase = Function (comida) {*

*Return “Hoy quiero comer: “ + comida;*

*};*

*Var hablar = Function (comida, cb) {*

*Return cb(comida);*

*};*

*Var fraseFinal = hablar(“Pizza”, devuelvoFrase);*

*Console.log(fraseFinal); = // Hoy quiero comer: Pizza*

En este ejemplo, la función hablar que ejecuta el callback recibe por parámetro una variable llamada comida y además, la función callback llamada cb. A su vez, esta variable comida, se le pasó como argumento al callback cb, por lo que la variable comida fue utilizada como argumento en ambas funciones para obtener el resultado esperado.