Notas JavaScript

En JavaScript no se puede heredar de más de una Clase, por ejemplo, no se puede hacer lo siguiente:

Class Alumno extends Usuario, Admin { }

Lo correcto es que se hereda de una sola Clase;

Class Alumno extends Usuario

Un prototipo es el lugar donde básicamente se almacenan las funciones.

¿cómo se llama al Constructor o a cualquier método de la Clase de la cual estas heredando (alumno sería el hijo, y usuario el padre). Se hace usando la palabra clave Super(); me permite llamar a cualquier atributo o método de la Clase que herede. Y **super()** permite llamar al Constructor. Con super() puedo hacer referencia a todo lo que quiera de la Clase padre.

Método login de Clase Padre Usuario

login(email, password){  
 return this.email === email && this.passwoord === password;  
}

Login es un método de la Clase hija Alumno que llama al método login() de la Clase padre Usuario()

login(email, password){  
 if (!this.activo)  
 return false;  
  
 return super.login(email, password)  
}

**Operador instanceOf()**

Nos sirve para saber de que Clase parte un Objeto, o de que tipo es un Objeto. Es un operador que nos devuelve “true” o “false”

Por ejemplo, con la siguiente línea se haría la pregunta, ¿es Pablo instancia de Usuario?

*console*.log(*pablo* instanceof Usuario);

true

esto devuelve “true”, que nos dice que Pablo es un Usuario.

¿Es jose un Alumno?

*console*.log(*jose* instanceof Alumno);

true

¿Es Pablo un Alumno?

*console*.log(*pablo* instanceof Alumno);

false

¿Es jose un Usuario?

*console*.log(*jose* instanceof Usuario);

true

¿Es jose instancia de Object?

*console*.log(*jose* instanceof *Object*);

true

¿Es pablo instancia de Object?

*console*.log(*pablo* instanceof *Object*);

true

¿Es un objeto vacio instancia de Object?

*console*.log([] instanceof *Object*);

true

**Todos los Objetos son instancia de Object.**

instanceOf se vuelve importante cuando hacemos tratamiento de errores, cuando estamos haciendo de backend.

**Para declarar Objetos en JS prácticamente lo haremos con la palabra reservada “const”**

Hay diferencia entre las “arrow function” (funciones de flecha) y las “funciones normales”. En especial con el uso de “this”, ya que con una función normal, “this” funciona apuntando al propio objeto, no así cuando usas una función de flecha.

Cuando estamos trabajando con Objetos, y necesitamos usar “this”, cuando necesitamos hacer referencias a los datos del propio Objeto, es obligatorio usar “funciones normales (function)”.

Cómo los Objetos son del mismo tipo, el método sería el mismo.

Cuando todos los Objetos de un tipo tengan una serie de métodos por defecto, lo que harán será crear nuevas Clases y demás.

Es decir, esto se podría crear una Clase que fuera Item, que creará Item1 e Item2 basándonos en esa Clase.

Si tenemos una función flecha (function arrow, no podemos usar this, nos dará error). Y tendremos que transformarlo a function. Estas “function” evidentemente podrían recibir parámetros; por ejemplo, recibimos el parámetro “descuento”.

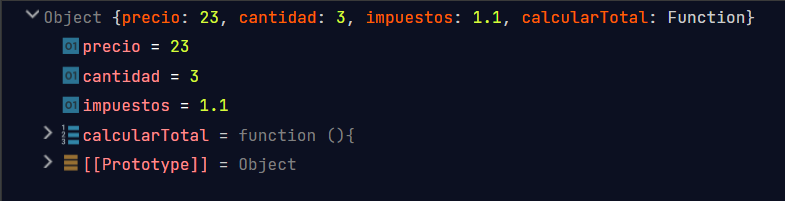
Acá el código completo;

const *IVA\_GENERAL* = 1.21;  
const *IVA\_REDUCIDO* = 1.10;  
  
const *item1* = {  
 precio: 15,  
 cantidad: 2,  
 impuestos: *IVA\_GENERAL*,  
 calcularTotal: function (){  
 return this.precio \* this.cantidad \* this.*impuestos* }  
}  
  
const *item2* = {  
 precio: 25,  
 cantidad: 3,  
 impuestos: *IVA\_REDUCIDO*,  
 calcularTotal: function (){  
 return this.precio \* this.cantidad \* this.*impuestos* }  
}  
  
const *factura* = {  
 *item1*,  
 *item2*,  
 calcularTotal: function(descuento){  
 return (this.*item1*.calcularTotal() + this.*item2*.calcularTotal()) \* descuento;  
 }  
}  
  
*console*.log(*factura*.calcularTotal(0.8))

**Comportamientos extraños con los Objetos que nos pueden llevar a errores**.

Imaginarnos que creamos un 3er Item en el código del ejemplo anterior. De la siguiente manera;

const *item3* = *item2*;

Luego asignamos un nuevo valor al atributo “precio”

*item3*.precio = 23;

si imprimimos el objeto item3 en consola vemos como el precio de item3 es 23, y esto está bien

Pero si luego imprimo en consola el item2, se observará un error en el precio de item2.precio = 23, y este valor no ha sido explícitamente asignado, el valor que tiene asignado en el código es de 25.

¿Porque está pasando esto? Porque item2 e item3 son exactamente el mismo Objeto

Los datos Primitivos, se denomina así no por que sean básicos, si no como se almacenan. Los primitivos almacenan su valor directamente.

Por ejemplo, en un objeto primitivo, si almacenamos un 5, 10, un hola o lo que queramos, lo que almacenamos algo es igual a hola, y el valor que se almacena es hola. Pero un Objeto al ser un tipo de un tamaño que puede ser muy grande, no podemos almacenar en esa memoria el valor directo, lo que almacenamos es una “referencia” una dirección a un puntero que hará referencia a donde estará localizado ese Objeto en memoria.

Entonces cuando copiamos un objeto con igual, lo que estamos copiando es la “referencia”, es decir que tenemos 2 referencias que apuntan al mismo Objeto. Item3 e Item2 apuntan al mismo Objeto en memoría.

const *item3* = *item2*;

Por eso, toda modificación que se haga a un Objeto Item3, se reflejará en Item2.

En resumen; los tipos primitivos se almacenan como “valor”, y los Objetos se almacenan como “referencia”

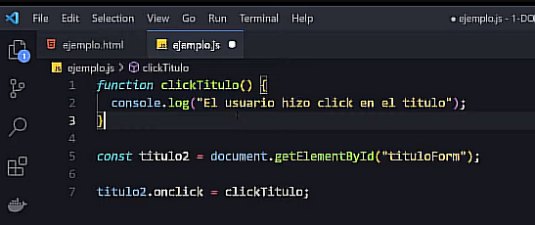
En JS, cuando un Entorno de Ejecución implementa el JS, por definición de lenguaje tiene que haber un Objeto Padre de todos, y en un Navegador, es Window (esto no lo creas en el Código, esto lo hace el NAVEGADOR)

Un Listener es el que escucha, y el handler es el que se ejecuta cuando ocurre ese evento. En JS existen 3 formas de definir estos Listeners y esos handlers.

**CÓMO manejar EVENTOS en Javascript – Event Listeners – DOM**

<https://www.youtube.com/watch?v=03eid8Lc8V8>

Desarrollo útil

Un evento lo podemos llamar desde el mismo código JS, tendríamos un Listener que es el que escucha el evento, y un Handler que es el que ejecuta el método del evento, ver ejemplo a continuación.

2daForma de hacerlo, no usar nunca esta opción;

titulo2.onclick = clickTitulo;

a la izquierda de igual está el “listener es onclick que proviene de extraer el ID de la etiqueta titulo en el HTML” y el “handler” a la derecha del igual es el método programado dentro del código JS.

La desventaja es que no puedes escribir más de una función, porque si haces una 2da, la 1ra función sería ignorada

Otra desventaja, es que si no existe el evento “on…” que buscamos, no tenemos manera de definirlo.

La primera forma era definir la llamada en el código **HTML** del listener desde la etiqueta, y la 2da que es la anterior es definir la llamada a un ID, y luego hacer que se genere un evento asociado y en consecuencia la llamada al handler.

La 3era forma si permite definir o escuchar cualquier evento que exista en el **DOM** (sin límite). De esta forma podría definir un evento, y la forma última y la más recomendada es utilizar el método “addEventListener” (definir escuchador de evento) y esta es una Función que permite pasarle 2 parámetros; el 1ero. El nombre del Evento que quiera escuchar, en el caso del ejemplo, el evento “click”, y cómo 2do parámetro, el “handler o la Función”, podría poner el nombre de la función “clickTitulo” o la función directamente utilizando la función flecha (arrow function) y dentro de ella poner lo que se necesite. Pero esto de utilizar la función arrow, no suele ser recomendado.

Entonces la 3er forma de hacerlo es así, y es la más recomendable;

**titulo2.addEventListener(“click”, clickTitulo);**

Un evento igual que lo añado lo elimino, y una vez lo elimino la acción clickTitulo no se ejecutará;

titulo2.removeEventListener(“click”, clickTitulo);

Pero si la función la defino dentro de el método addEventListener en el 2do parámetro, no podría hacer la eliminación del Evento clic, porque no tengo acceso por el “scope (ámbito)” fuera de ahí.

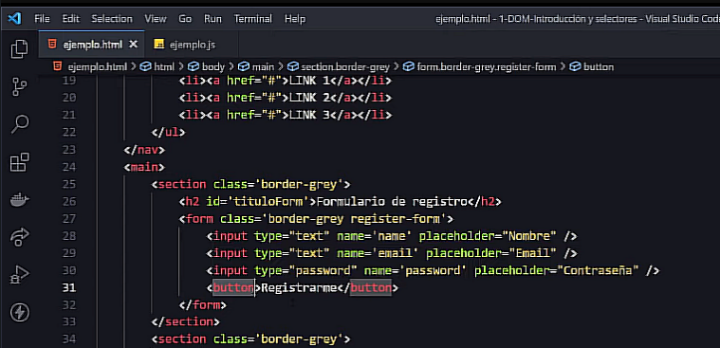
Por ejemplo, si defino una función anónima no tengo acceso a ella, porque además no tengo variable. Tendría que definirla fuera, no como parámetro del método **addEventListener**.

**titulo2.addEventListener(“click”, () => {} );**

Definiendo la función clickTitulo afuera, puedo hacer referencia a ella desde addEventListener, y desde removeEventListener como segundo argumento.

De esta forma podríamos añadir todos los Eventos que necesitemos.

Dirección de documentación relacionada con Eventos JS.

**Introduction to events**

<https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/Events>

Lista de Eventos – Event index

<https://developer.mozilla.org/en-US/docs/Web/Events>

Cómo se define un formulario en una página HTML

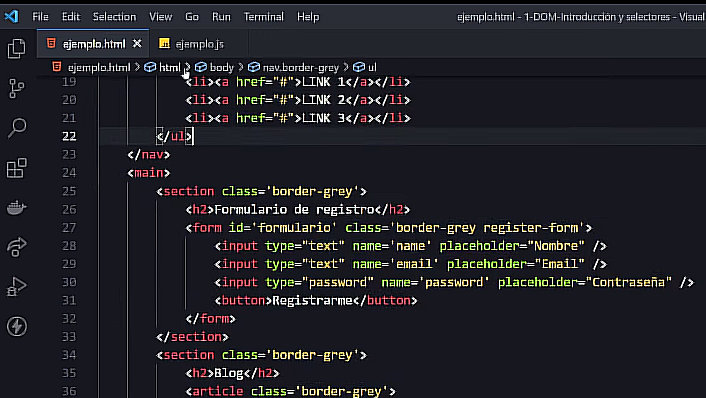
Cuando hacemos “clic” en el botón de un formulario, aparte de provocar ese evento, lo que hacemos también es provocar un evento sobre la etiqueta “form”, un evento “submit”. Un botón por defecto lo que tiene es un evento de tipo “submit” aunque no lo defina en la etiqueta que contiene el botón <button type = ‘submit’> Registrarme </button>

Lo que va a hacer es provocar el evento “submit” del formulario. Es un evento especial de “envío”.

Si en lugar de “submit” utilizamos “button” no ejecutarias esa acción. Si no ponemos nada y el botón está dentro del formulario <button> Registrarme </button> se ejecutará la acción de “submit” por defecto.

Ese formulario se visualiza en el navegador del siguiente modo. Ahí puedes ver el botón de “Registrarme”.

Vamos ahora a detectar ese evento y ver cómo podemos mandar un formulario, cómo podríamos sacar los datos de estos inputs (name, email, password), para una validación previa a enviarlo, etc.

Lo 1ero que tenemos que hacer, es como el evento se origina en “form” el “button”, que lo que hace es provocar un evento “on form”, necesito el elemento “form” para añadir ese ID en la etiqueta “form”, del siguiente modo:

Se necesita el elemento “form” para añadir ese ID listener

Y en el código de JS hacemos lo siguiente;

const form = document.**querySelector**(‘#formulario’)

Asociamos en el 1er argumento el “listener” (evento) submit del botón, y en el 2do argumento (el “handler”) el nombre de una función enviarFormulario que crearemos.

form.addEventListener(‘submit’, enviarFormulario)

Definir esta función enviarFormulario (con una arrow function), y esta función como la vamos a utilizar como un “handler” va a recibir un argumento (evento). De forma que cuando haga clic en el botón del formulario y se ejecuta la acción, se admite el evento “submit”, estoy escuchando ese evento “submit” sobre mi elemento “form” y voy a ejecutar la acción (handler) enviarFormulario. Al ejecutar la página HTML que contiene el formulario y hacer clic en el botón “Registrarme”, en la consola del navegador, se ve que el evento aparece y desaparece, es que se está actualizando la página, es un fallo común.

const enviarFormulario = (event) => {

console.log(event);

}

Cuando un evento sucede, y nosotros escuchamos el evento, también hay funciones nativas del navegador que hacen cosas antes ciertos eventos de forma nativa por el tipo de evento.

Lo que tiene el evento “submit” de los formularios es que cuando se ejecuta, si no le dices nada ni haces nada sobre ese evento, va a enviar la página por decir de una manera, lo que está haciendo es solicitar la misma Web con name, email, y password igual, está como autoenviando por query y param los elementos del HTML, por defecto no quiero que haga esto.

Qué formas tenemos de prevenir que el evento haga cosas nativas del navegador. Seria con la siguiente instrucción;

const event.**prevenDefault**()

si llamamos a esta función sobre el evento que hemos capturado, lo que estamos diciendo es; yo en mi “listener” primero voy a prevenir el comportamiento por defecto que tiene el evento, luego veremos lo que hacemos con el.

También se puede acceder a los valores de los “input” que ha puesto el usuario de una manera sencilla.

Esto se puede hacer a través de la propiedad “event.target()” del siguiente modo, no tengo que hacer getElementById de lo que sea dentro de el console.log

*console*.log(event.target.name, event.target.email, event.target.passwoord);

y dentro de ellos, si ya he accedido al elemento, puedo acceder al valor “value” del “input” de cada uno de ellos

*console*.log(event.target.name.value, event.target.email.value, event.target.passwoord.value);

Este es el Código completo de ejemplo.

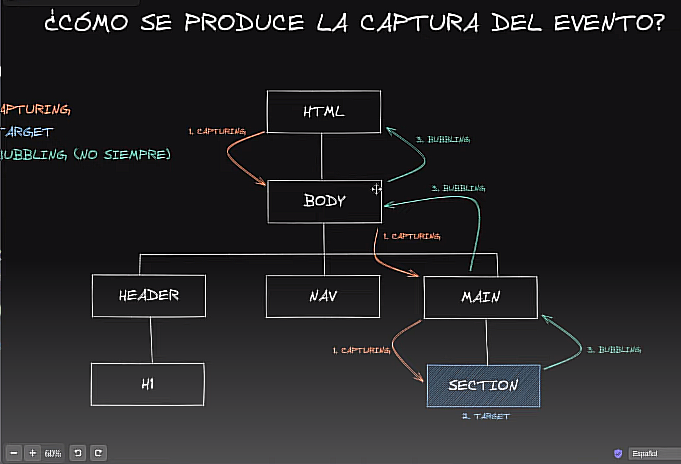
const *form* = *document*.querySelector("#formulario");  
const enviarFormulario = (event) => {  
 event.preventDefault();  
 *console*.log(event.target.name.value, event.target.email.value, event.target.password.value);  
};  
  
*form*.addEventListener("submit", enviarFormulario);

Puedo escribir el Código de una forma más comoda, haciendo un destructuring de event.target; evento tiene un objeto target que a su vez tiene las propiedades; name, email y password que representan los “input”. Y con “value” que introducimos en el formulario.

Podría validar el nombre del siguiente, para verificar si

if(name.value.length === 0) alert("El nombre no es valido");

¿Cómo se produce la captura del Evento?



1. Capturing
2. Target
3. Bubbling (no siempre)

**Capturing**. 1era Fase de Captura cuando se produce un evento, la captura se produce en el elemento HTML desde arriba hacia abajo. Se va buscando el elemento más profundo sobre el cual se ha producido el evento. Si hago un clic en una “sección” la captura va ir bajando a través del HTML, el body, y todos los elementos que tengo en el HTML, pasando por todos los padres de esa sección hasta localizar el elemento del DOM más profundo en el que se haya producido el “evento”.

**Target**. 2da Fase, esta fase lo que hace es ver si el elemento del DOM más profundo sobre el que se ha localizado el evento, tiene un “listener” asociado a ese “evento”, por ejemplo, si he puesto un “addEventListener” de clic en la “sección” si se produce un clic, por ejemplo.

**Bubbling**. 3ra Fase. Si hay un “listener” en “section” y se ejecuta un evento o lo que haya definido para la sección y luego va mirando en los elementos padres también hay un “listener” para ese mismo tipo de evento, es decir, si hago un clic en la “section” y luego tengo un listener para el clic en el “main”, se va a ejecutar en los 2 sitios, hablando en este caso del clic. Luego veremos que no todos los “eventos” tienen esta 3ra Fase de “Bubbling”, depende del tipo de evento, el del clic por ejemplo si que lo tiene. Esto es importante porque si vamos definiendo demasiados eventos y los mismos tipos de eventos podemos tener problemas, porque si hago clic en “section” estoy haciéndolo en “main”, en “body” y en el “HTML”.

El hacer clic en un evento clic que está programado a nivel de sección y luego de Título, hace que si hacemos clic en “Titulo” solamente, hace que se dispare el “clic” de la “section” también, esto es debido al “Bubbling”, pero este comportamiento se puede modificar. Cuando estamos en un “addEventListener” podemos detectar el evento, y este tiene ciertas propiedades, entre ellas “event.bubbles”, si devuelve “true”, lo que haría esto es indicar si el evento que se ha producido hace “bubble”.

*titulo2*.addEventListener("click", (event) =>{  
 *console*.log(event.bubbles);  
 *console*.log("Clic en el Titulo");  
});

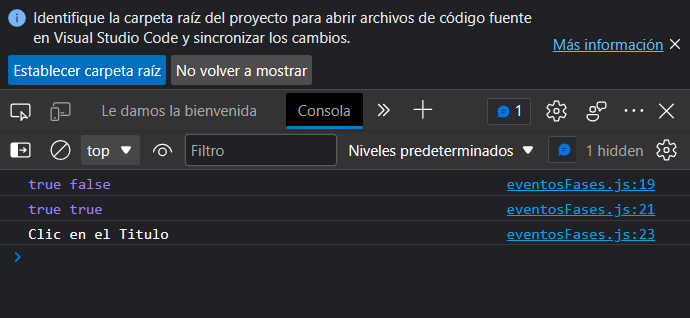
Hay eventos que no producen “bubble” y sea “false”, por ejemplo, comprobemos con el “submit”.

Probemos esto en la función “enviarFormulario”. El resultado es “true” porque tengo una acción dentro de otra, y se ejecutará el “bubble”, porque escucho el evento del “submit” por encima del formulario.

*console*.log(event.bubbles);

Cómo puedo hacer para modificar el comportamiento aunque tenga esa función de “bubbling” y vaya subiendo para arriba el evento, solo me interesa que se ejecute el elemento más profundo de mi jerarquía, en el que realmente he hecho “clic”. En ese caso, igual que hacíamos un “event.preventDefault()”, tenemos una función “**event.stopPropagation()**” que lo que hace es detener la propagación del evento, es decir, que no escale hacia arriba (que no haga Bubbling).

Y además, si hago un “event.bubbles” combinado con “**event.stopPropagation()**” y “event.cancelBubbles” como se vé a continuación:

*titulo2*.addEventListener("click", (event) =>{

*console*.log(event.bubbles, event.cancelBubble);  
 event.stopPropagation();  
 *console*.log(event.bubbles, event.cancelBubble);  
  
 *console*.log("Clic en el Titulo");  
});

Al ejecutar la página que llama al código JS, se observa lo siguiente en la salida de consola.

La propiedad “bubbles” lo que indica es si el evento hace “bubbling” por defecto, y la propiedad “cancelBabble” me indica si como programador, he cancelado la propagación del “bubbling” hacia arriba de ese evento. Por eso, en la consola se muestra el resultado de “event.cancelBubble” en “false” porque aun no se ha ejecutado el método “stopPropagation”

Se observa que solo se ejecuta el “clic” del elemento “titulo”, es decir, el evento de “sección” no se ha ejecutado. De esta forma, si hago clic en Titulo, solo se ejecuta el título, si hago clic en sección, solo se ejecuta la sección. Tengo separado los eventos, la propagación.

**DELEGACIÓN DE EVENTOS**.

Uno de los detalles más interesantes del “bubbling” es que puedo capturar, definir un “handler” a través de un “listener” de un evento para múltiples elementos.

Si tengo en mi documento HTML definidos múltiples artículos, no es práctico definir un “listener” para cada artículo para hacer la misma cosa. En lugar de eso se define a nivel de “section”, de forma que aunque hagas “clic” en el artículo, por el “Bubbling” va a acabar y ejecutar en la sección, y a través de la propiedad “event.target” voy a poder definir en que artículo he hecho clic.

Con esos cambios si hago “clic” en la sección, el console.log, me imprime en los 2 casos, la sección. Pero si hago clic en el Titulo, el Target del Evento no es la “sección” es el “Titulo”. Y “current target” donde se está ejecutando actualmente el evento si es la sección. Pero el objetivo del “clic” no era la “sección” era el “Titulo” que está dentro, con lo cual puedo saber poniendo un evento en un “listener” en un “Padre” a que elemento “hijo” he hecho “clic” y simplificar un montón las acciones de los eventos.

Imaginemos que hacemos clic en un artículo y despliego cada artículo, no hace falta poner un “listener” en cada artículo, lo pongo en la “sección” que contiene esos artículos (el listado entero de esos artículos), y luego a través del “event.target”, puedo decir, yo ya tengo el “target”, se que es este “h2”, le cambio la “clase que sea de CSS” para hacerlo más grande para ampliarlo, le meto una transición o lo que necesite. Es uno de los usos más comunes del “event.bubble”.

Una aclaración de la Fase de Captura, y es que nosotros podemos decidir ejecutar la función en esa Fase. Puede que me interesa si tengo definido varios listeners en la jerarquía, se ejecuten a la inversa, por alguna razón, por ejemplo; que se ejecute 1ero el “main” y luego “section” para efectividad. Podemos hacer que un “listener” este pendiente de la Fase de “captura”.

Si ponemos un 3er parámetro en forma de Objeto “capture: true” y le ponemos a la sección.

const enviarFormulario = (event) => {  
 event.preventDefault();  
 const {name, email, password} = event.target;  
 *console*.log(name.value, email.value, password.value);  
 if(name.value.length === 0) alert("El nombre no es valido");  
};  
  
*form*.addEventListener("submit", enviarFormulario);  
  
const *section* = *document*.querySelector("#seccion");  
const *titulo2* = *document*.querySelector("#titulo2");  
  
*titulo2*.addEventListener("click", () => {  
 *console*.log("Clic en el Titulo");  
});  
*section*.addEventListener("click", () =>{  
 *console*.log("Click en Seccion");  
},  
{  
 capture: true,  
}  
);

COMO CREAR, INSERTAR Y MODIFICAR ELEMENTOS DEL DOM

CREANDO y MODIFICANDO HTML desde Javascript 😄 DOM

<https://www.youtube.com/watch?v=OZfgQGvnmYM>

Veremos una serie de funciones que nos habilitan esos elementos del DOM. Podemos cambiar el HTML desde JS y aportarle dinamismo.

Veremos cómo crear elementos que representen etiquetas HTML. Supongamos que queremos crear otra sección, otro artículo, eso lo podemos hacer con el método “create element”, este método lo tiene el DOM.

Hay 3 elementos,

1. El Objeto node, un node puede ser un document o un element.
2. El Objeto Document y 3- los elementos que son los que podemos crear o podemos obtener con selectores. Los selectores los obtenemos si ya están el DOM

y con estos métodos los vamos a crear. Estos métodos de creación, todos parten del Document. Y el 1ero que veremos es createElement(); este es un método que tiene 2 parámetros, el 1ero es el nombre de la tag que queremos crear; un article, un H2, una section, o lo que sea. Si por ejemplo queremos crear un Título, podríamos crear un H2, el 2do parámetro es un parámetro de Opciones que es un objeto con una única propiedad, que en la mayoría de casos no se usa. Usualmente se utiliza un parámetro y será el nombre de la TAG. El 2do parámetro es opcional.

const *titulo2* = *document*.createElement('h2');

Qué mas elementos tenemos en HTML. A parte de etiquetas con sus propiedades, luego veremos como añadirlas, tenemos texto. Es decir, nuestro HTML a parte de Etiquetas, tiene texto.

Cómo podemos crear un tipo de elemento que represente el texto. Lo podemos hacer del siguiente modo:

const *textoTitulo2* = *document*.createTextNode("Nuevo titulo creado para el video");

Y con esto ya tenemos un texto. Si vamos al navegador, y escribimos textoTitulo2, veremos la salida de texto que pusimos como parámetro en createTextNode.

Tenemos un método para crear “comentarios”. En HTML podemos definir comentarios con la sintaxis, en la que podemos escribir el comentario que no se verá reflejado en la salida del documento HTML porque es un comentario.

<!-- Escribir cualquier cosa -->  
<main id**="main"**>

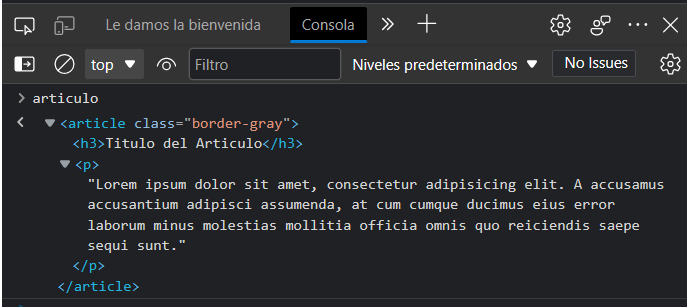
Hay formas de crear comentarios para el DOM desde JS. No se usa nunca porque normalmente los ponemos en el código HTML, no los sueles insertar de forma dinámica. Pero saber que existe una forma de hacerlo. Y si voy a la consola y escribo comentario, aparecerá el comentario que escribimos en el código JS con las marcas de los comentarios.

const *comentario* = *document*.createComment('Comentario random');

<!--Comentario random-->

Estas son las 3 maneras para que podamos crear elementos o nodos que podamos luego insertar en nuestro DOM. Normalmente utilizaremos los 2 primeros. A veces no necesitaremos crear elementos desde 0 porque es demasiado tedioso o ya tenemos un elemento en nuestro DOM.

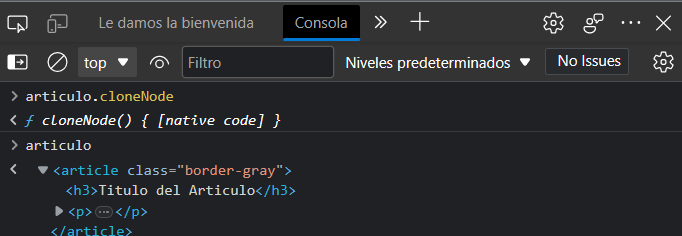
CLONANDO ELEMENTOS.

Primero, hacemos una selección del “article” que deseamos clonar

const *articulo* = *document*.querySelector('article');

lo visualizamos en consola del navegador

Tenemos para clonar el método “clone node” que es un poco diferente a los métodos create element. “Clone node” está por encima de la Clase Node, entonces lo puedo aplicar sobre elementos.

No puedo crear “article.createElement” no existe el método en article, solo existe en el documento. El de Clonar si existe sobre todos los nodos. De modo que puedo aplicar el método “article.cloneNode” a un artículo y lo que estaré haciendo es hacer una copia del elemento “article” en memoria. Esto lo hacemos para probarlo en la consola, con esto no lo hemos introducido en el DOM, solo lo hemos creado en memoria en JS.

Hay que notar una diferencia y es que el nodo original tiene hijos, y el clonado no los tiene. Porque, “cloneNode” admite un parámetro Booleano “Deep” que es “false” por defecto. Si le pongo “true” estoy diciéndole que me copie en profundidad el nodo, es decir el nodo y todos los hijos de hijos que contengan dentro de este.

Tenemos una Propiedad que nos indica si un elemento perteneciente al DOM, en este caso “articulo” está en nuestra Web, porque a lo mejor por CSS no es visual o está oculto, y esa Propiedad es “isConnected”. Lo que indica es que si el está “true” está visualmente o oculto en el documento HTML. Pero un elemento puede solo existir en memoria (false), porque aún no lo hemos insertado en el HTML.

Normalmente para insertar un elemento en la jerarquía de nuestro HTML tenemos que tomar un punto de referencia, saber donde queremos insertar el elemento.

Por ejemplo, si queremos insertar “articulo”, nos interesa insertarlo debajo de “articulo”, lo que nos interesa saber para hacer eso, es saber el “Padre” (el Padre inserta un Hijo), entonces vamos a seleccionar la “sección” y vamos a utilizar el “ID”.

//seleccionamos el elemento article  
const *articulo* = *document*.querySelector('article');  
//clonamos el nodo articulo y ahora tiene una copia en articulo2  
const *articulo2* = *articulo*.cloneNode(true);  
  
const *section* = *document*.getElementById('section');

“articulo2” solo existe en memoria. Para insertarlo podemos utilizar el método “appendChild”, en este caso el método lo que admite es un “nodo” que en este caso es “articulo2”. Lo que hace este método es agregar el elemento nodo que le pasamos como parámetro como último hijo de ese elemento (del elemento Padre que en este ejemplo es el elemento section).

section.appendChild(articulo2)

appendChild solo permite hacer inserción de ese tipo “como último hijo”. Pero a veces, me interesa insertar en otro orden; insertarlo como 1er hijo, como hermano superior (o hermano inferior) o interesa insertarlo después de la “section”.

Podemos insertar un elemento en cualquier sitio del DOM. Lo podemos hacer utilizando el siguiente método: insertadjacentElement().

Element.insertAdjacentElement() El método insertAdjacentElement() inserta un elemento nodo dado en una posición dada con respecto al elemento sobre el que se invoca.

<https://developer.mozilla.org/es/docs/Web/API/Element/insertAdjacentElement>

Sintaxis

elementoObjetivo.insertAdjacentElement(posición, elemento);

Parameters

posición

Un [DOMString](https://developer.mozilla.org/es/docs/conflicting/Web/JavaScript/Reference/Global_Objects/String) representando la posición relativa al elementoObjetivo; debe ser una de las siguientes cadenas:

* 'beforebegin': Antes del elementoObjetivo.
* 'afterbegin': Dentro del elementoObjetivo, antes de su primer hijo.
* 'beforeend': Dentro del elementoObjetivo, después de su último hijo.
* 'afterend': Después del elementoObjetivo.

elemento

El elemento HTML a ser insertado.

Valor devuelto

El elemento insertado o null, si la inserción falla.

Excepciones

|  |  |
| --- | --- |
| **Excepción** | **Explicación** |
| SyntaxError | La posición especificada no tiene un valor reconocido. |
| TypeError | El elemento especificado no es un elemento válido. |

Visualización de los nombres de posición

<!-- beforebegin -->

<p>

<!-- afterbegin -->

foo

<!-- beforeend -->

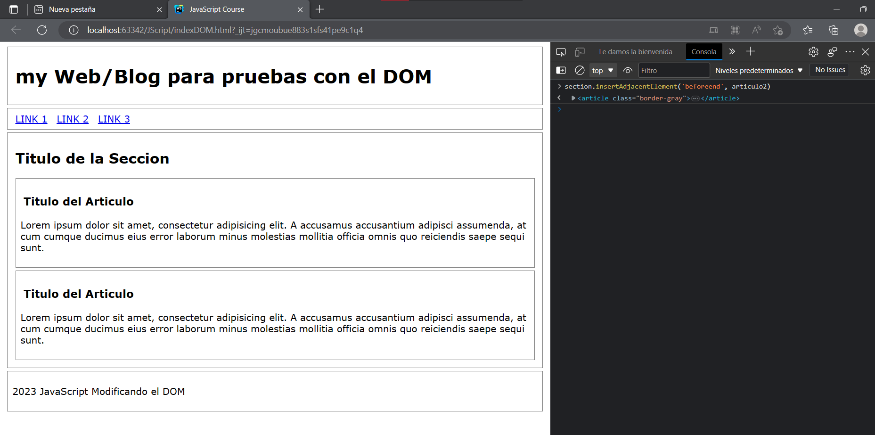
</p>

<!-- afterend -->

**Nota:** Las posiciones **beforebegin** y **afterend** sólo funcionan si el nodo está en un árbol y tiene un padre.

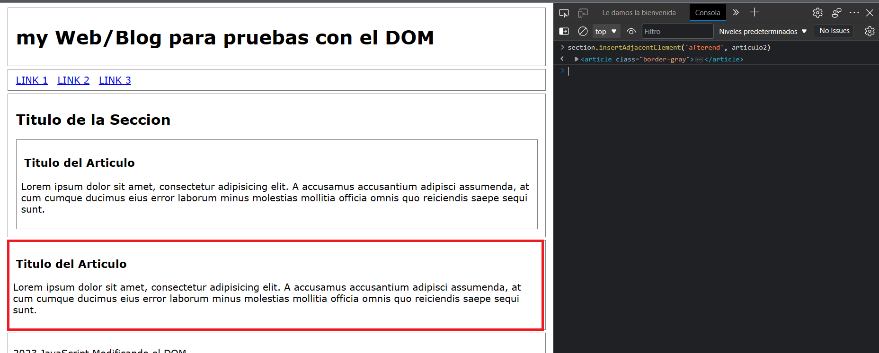
Ejemplo, en este caso hemos seleccionado a partir del elemento “section”.

<!-- Escribir cualquier cosa -->  
<main id**="main"**>  
 <!-- BeforeBegin / Hermano Superior -->  
 <section id**="section"** class**="border-grey"**>  
 <!-- AfterBegin / Primer Hijo -->  
 <h2>**Titulo de la Seccion**</h2>  
  
 <article class**='border-gray'**>  
 </article>  
 <!-- BeforeEnd / Ultimo Hijo -->  
 </section>  
 <!-- AfterEnd / Hermano Inferior -->  
</main>

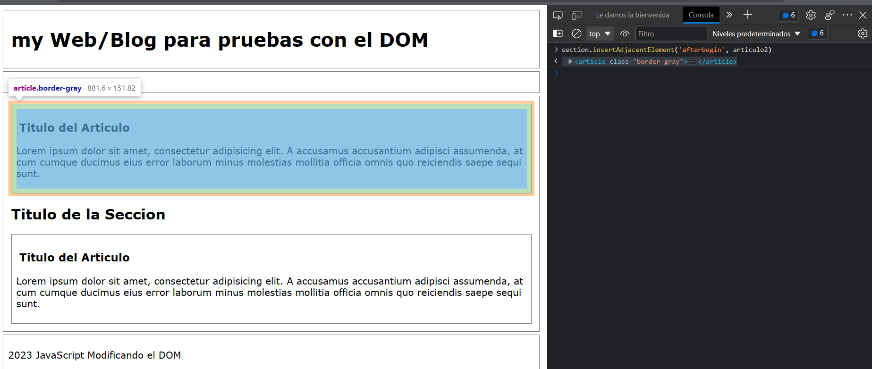
Si quiero replicar el “appendChild” puedo hacerlo con “beforeend”. Haré una emulación con ayuda de la “consola” del navegador con la siguiente instrucción:

>section.**insertAdjacentElement**('**beforeend**', articulo2)

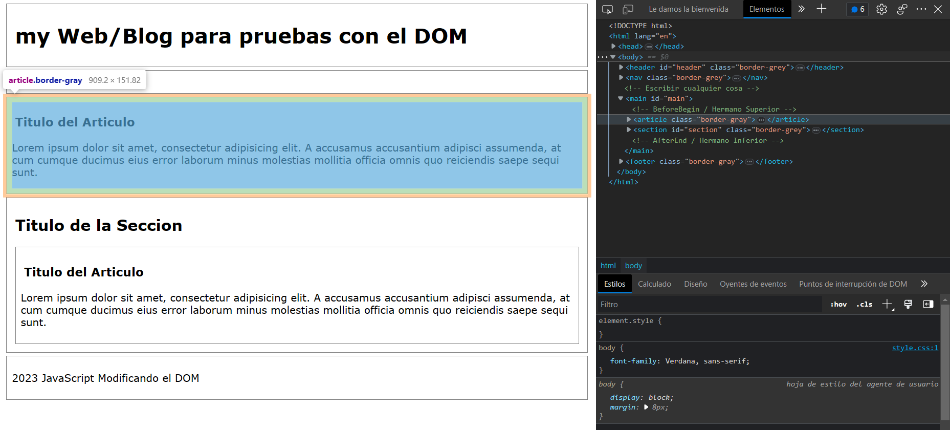
Resultado en consola y en página web en imagen

Si en lugar de eso utilizo ‘afterend’ el resultado sería mostraría el articulo después de la sección, cómo un hermano inferior de “section”

>section.**insertAdjacentElement**('**afterend**', articulo2)

Mostrar como 1er hijo “afterbegin”, iría encima del “Titulo H2” de la section.

>section.**insertAdjacentElement**('**afterbegin**', articulo2)

Luego tendríamos “beforbegin” que sería como hermano superior de “section”

>section.**insertAdjacentElement**('**beforebegin**', articulo2)

Este método **insertAdjacentElement**, nos sirve para agregar todo tipo de elementos, ya sea un elemento clonado, ya sea con “createElement”, con “createNode” o con “créateComment”. Digamos que con “appendChild” o con insertAdjacentElement podemos insertar cualquier tipo de elemento o nodos.

Hay veces que debido a nuestro caso de uso debemos insertar o crear elementos de otras formas.

Veamos como insertar “texto” sin necesidad de crear un “nodoTexto”, como concatenar texto al final y demás.

Esto se puede hacer con el método “**insertAdjacentText**”. Tiene una Sintaxis muy parecida a **insertAdjacentElement**.

section.insertAdjacentText('beforeend', 'Aquí acaba la sección')

El resultado es el siguiente:

Otro ejemplo que nos puede servir de uso es el “**insertAdjacentHTML**”, puedo tener trozos o plantillas de HTML contenidos en JS y meterlos directamente sin necesidad de andar creandolos con document.createElement, si no escribiéndolo como un string e insertarlos directamente

**section**.insertAdjacentHTML('beforeend', '**<h3>**Esto es un Título inventado**</h3>**')

Insertar elemento por elemento quizás no sea lo más adecuado, podría ser que tengamos una especia de plantilla HTML de componente precreado que modificamos para nuestro Caso de Uso, pero lo insertamos como HTML directo.

Ahora veremos como “eliminar” un “Nodo” de nuestro DOM. Lo podemos hacer con el método “remove()”.

section.remove()

el comando anterior ejecutado en consola, elimina una “section” completa, aunque en memoria sigue existiendo, pero no está conectada a nuestro DOM, no está en el HTML

También podemos desear reemplazar el contenido de un elemento.

Puedo tener una Section, y puede ser que, en lugar de agregar contenido, se desea reemplazar contenido

Podría utilizar “innerHTML” que es una propiedad de lectura, pero que la puede sobrescribir de la siguiente manera, desde consola:

section.innerHTML = '<h2>Nuevo título de la sección</h2>'

También tenemos la opción de reemplazar todo el HTML, no solo lo que está contenido; es decir, sección y lo que hay dentro con ayuda de “outerHTML”, esta devuelve la “section” y lo que hay dentro. Con “innerHTML” Me devuelve solo lo que hay dentro. Vea ejemplo a continuación;

section.outerHTML

'<section id="section" class="border-grey"><h2>Nuevo título de la sección</h2></section>'

section.innerHTML

'<h2>Nuevo titulo de la sección</h2>'

Una última opción que manejar con texto; sustituiré el contenido de texto solo de mi elemento “h2” que está dentro de “section”.

section.children[0].textContent = 'otro título más'

'otro título más'

Tenemos la siguiente instrucción que sirve para obtener cualquier “atributo”

section.getAttribute

ƒ getAttribute() { [native code] }

Si le paso el “id” como parámetro, obtendré el id de ese elemento “section”

section.getAttribute('id')

'section'

También se puede utilizar el siguiente atajo:

section.id

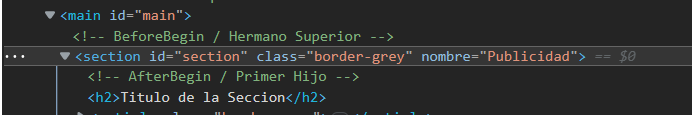
'section'

También puedo obtener el valor del atributo “nombre” de una sección del siguiente modo, al consultar en la consola, me devolverá el valor de la propiedad “nombre” que es ‘mercadeo’

section.getAttribute('nombre')

'mercadeo'

También tengo “setAttribute” que me permite en vez de un solo parámetro con el nombre, permite 2 parámetros; con “name, value” y poner otro valor, modificando el valor de la propiedad en cuestión.

section.setAttribute('nombre', 'Publicidad')

Resultado en el HTML

También podemos obtener el valor del atributo “class” y nos devolvería la clase y podríamos modificarla, pero para el CSS tenemos otras propiedades más idóneas para esto.

section.getAttribute('class')

'border-grey'

Esas propiedades son la propiedad “classList” que nos devuelve un listado de las clases que tenemos en nuestro elemento. En este caso solo una.

section.classList

1. *DOMTokenList ['border-grey', value: 'border-grey']*
   1. **0**: "border-grey"
   2. length: 1
   3. value: "border-grey"
   4. [[Prototype]]: DOMTokenList

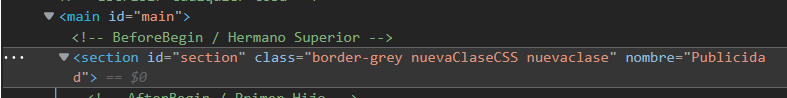
Y tenemos la propiedad “className” que nos devuelve el String de la clase directamente

section.className += ' nuevaClaseCSS' que nos permite concatenar un string al valor de la propiedad class name.

border-grey nuevaClaseCSS

Pero lo adecuado sería lo siguiente; es una manera más cómoda, ya que te añade los espacios en el HTML, el className es para ver en formato String no se suele usar para modificar. Para modificar suele ser mejor el “classList”, porque tiene los métodos; add, remove, esté último te permite eliminar una clase si existe.

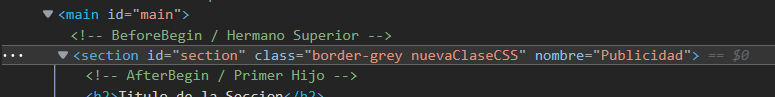
section.classList.add('nuevaclase')



De este modo podríamos eliminar la clase “nuevaclase” que hemos añadido al final

section.classList.remove('nuevaclase')

el HTML ya no tendrá esa clase que habíamos añadido, se observa que ya no aparece la ‘nuevaclase’

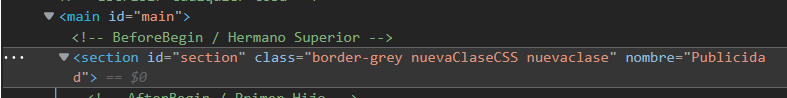


Otro método que no es tan utilizado, pero es más útil de lo que pueda parecer, que es el método “toggle”. Con este método no se sabe si tiene la clase, pero si la tiene que la quite y si no la tiene que la ponga. Es algo típico de un interruptor, que se suelen utilizar para cambiar de valor entre sí. De forma que, si aplico lo siguiente, da igual que lo tenga o no. Funciona como interruptor.

Como no lo tiene agrega un true, y lo pone al final del string de Class.

section.classList.toggle('nuevaclase')

**true**



y si lo vuelvo a ejecutar, devuelve false, y lo quita

section.classList.toggle('nuevaclase')

false

**Node.cloneNode()**

<https://sites.google.com/site/tumentororg/dhtml/clonacion-de-un-nodo-clonenode>

El método Node.cloneNode() devuelve un duplicado del nodo en el que este método fue llamado.

**Sintaxis**

var dupNode = node.cloneNode(deep);

node

El nodo que se desea clonar.

dupNode

El nuevo nodo que será un clon de node

deep *Opcional*

**true** si los hijos del nodo también deben ser clonados, o false para clonar únicamente al nodo.

Si queremos hacer una copia idéntica de un nodo y ubicarlo en otra parte del árbol, podemos utilizar el método:

**cloneNode**(<true o false> )

Si le pasamos como parámetro el valor true, hará una copia del nodo y todos sus descendientes; en caso de pasarle false sólo hará una copia de dicho nodo.

Confeccionaremos un ejemplo que se clone un div que contiene tres hipervínculos:

pagina1.html

<!DOCTYPE html>  
<html lang**="en"**>  
<head>  
 <meta charset**="utf8"**>  
 <title>**Problema**</title>  
 <script src**="funciones.js"** type**="text/javascript"**></script>  
</head>  
<body>  
<div id**="enlaces"**>  
 <a href**="#"**>**Enlace 1**</a><br>  
 <a href**="#"**>**Enlace 2**</a><br>  
 <a href**="#"**>**Enlace 3**</a><br>  
</div>  
<input type**="button"** onclick**="**javascript:clonarNodos()**"** value**="Clonar nodos"**>  
<div id**="enlacesnuevos"**>  
</div>  
</body>  
</html>

var *orden*=1;  
function clonarNodos()  
{  
 var id=*document*.getElementById("enlaces");  
 var nuevos=id.cloneNode(true);  
 nuevos.style.id='enlaces'+*orden*;  
 *orden*++;  
 id=*document*.getElementById("enlacesnuevos");  
 id.appendChild(nuevos);  
}

En la función de clonar nodos debemos tener algunas consideraciones:

Como pasamos true al parámetro del método cloneNode se crea una copia idéntica del nodo y de todos sus descendientes (la llamada del método se hace a partir del objeto que queremos clonar):

**var** id=document.getElementById("enlaces");

**var** nuevos=id.cloneNode(**true**);

Ahora es importante tener en cuenta que los nodos clonados tienen el mismo valor en la propiedad id por lo que debemos modificársela:

nuevos.style.id= 'enlaces' + orden;

orden++;

Por último, obtenemos la referencia del div donde se insertará este subárbol:

id=document.getElementById("enlacesnuevos");

id.appendChild(nuevos);

Desarrollo útil

¿FOR … IN FOR … OF? Cuál debes utilizar – Curso de JavaScript desde 0

https://www.youtube.com/watch?v=FJy8xgEdkNc&t=24s

const *cursoJS* = [  
 "Intro a JS",  
 "Variables y tipos",  
 "Conversion de tipos",  
 "Funciones"  
];  
  
for (let i = 0; i < *cursoJS*.length; i++){  
 *console*.log(`${i + 1}. ${*cursoJS*[i]}`);  
}

Otra forma de hacerlo con el bucle for recorriendo desde el último elemento hacia atrás

const *cursoJS* = [  
 "Intro a JS",  
 "Variables y tipos",  
 "Conversion de tipos",  
 "Funciones"  
];  
  
for (let i = *cursoJS*.length - 1; i >= 0 ; i--){  
 *console*.log(`${i + 1}. ${*cursoJS*[i]}`);  
}

Se usa un “Let” porque la variable “i” que estamos definiendo se va ir modificando constantemente en las iteraciones.

Bucle for .. of

for (const leccion of *cursoJS*){  
 *console*.log(leccion);  
}

Lo que hace el for … of (que se escribe en menos líneas) por nosotros es como el código a continuación;

for (let i = 0; i < *cursoJS*.length; i++) {  
 const leccion = *cursoJS*[i];  
 *console*.log(leccion);  
}

¿Qué ventanas e inconvenientes tiene este for … of?

Uno de ellos es que no tengo acceso a ningún “index”. Es decir, en este tipo de bucle no sabemos en que elemento estamos. Si necesitas saber el índice, es que no necesitas utilizar un “for … of”. Es una mala práctica usar con este bucle un índice o contador dentro del bucle para saber en que elemento estamos, no tiene sentido. Si se necesita un índice se usa un ciclo for convencional.

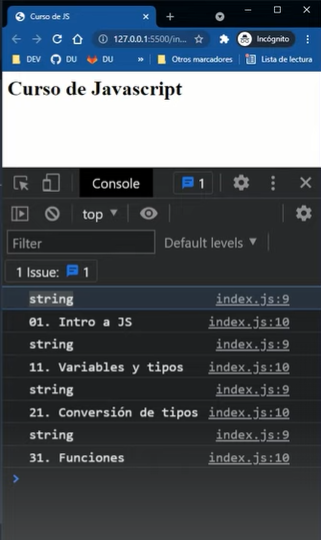
El for … of debe estrictamente hacerse sobre “**objetos iterables**”.

**Bucle For … IN**

Se piensa que este bucle es para Arrays y realmente no lo es ni por asomo.

Para usar este bucle podemos hacer el siguiente código:

for (const leccion in *cursoJS*) {  
 *console*.log(leccion);  
}

Este código genera un índice que algunos piensan que es el “índice” del Array, pero no lo es.

Un array es un objeto.

El bucle FOR … IN nos devuelve todas las claves que tiene el objeto cursoJS

Veamos porque no es posible utilizar FOR … IN y que problemas tiene si lo quieres utilizar con un índice.

for (const i in *cursoJS*) {  
 *console*.log(typeof i);  
 *console*.log((`${i + 1}. ${*cursoJS*[i]}`););  
}

console.log(typeof i); lo que hace es verificar de que tipo es la variable “i”, y devuelve string. Por lo que el código anterior tal como está genera un error de tipo lógico, ya que al tratar de hacer ${i + 1} esto no hace que la suma tenga el resultado esperado. Se ve en la imagen de al lado que el valor de posición en el array dice otra cosa rara para la posición en el array, como 11, 21, 31.

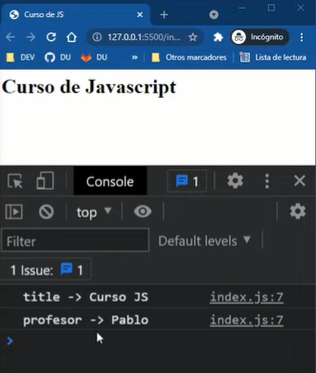
Podrías hacer cursoJS. Profesor = “Pablo”; y el JS lo aceptaría porque lo agrega como un nuevo elemento del objeto array. Esto pasa porque el array no existe cómo tal en JS. Mas bien es una forma que tenemos en JS de representar un objeto especial. Un Array es un objeto como otro cualquiera.

En resumen para que sirve este bucle FOR … IN; para recorrer las claves de un objeto, para iterar cuando queremos iterar sobre las claves de un objeto. Es mejor nunca utilizarlo para Arrays, se puede hacer y te funciona, pero si por ejemplo vienes de otra librería con un objeto que no has creado, en cualquier momento nos pueden colar una propiedad para cualquier tarea y nos puede generar un lío.

Aparte de todo esto, la implementación de FOR … IN en el momento de depurar en Chrome y en Firefox te devuelve los resultados en orden, pero en el estándar de JS dice que no tiene porque devolver los resultados en orden. Por ejemplo, te puede devolver 0 1 2 3 profesor, tal y como los pinta, pero en otros lenguajes perfectamente te podría devolver 3 2 1 7 9 6 4 por qué no está definido en el estándar que FOR … IN todas las implementaciones deban ir en orden, orden alfabético.

Hay que aprender a utilizar más el FOR. Todos los FOR IN en los que estemos iterado sobre Array se tienen que convertir en FOR. Todos los FOR … OF en los que queramos utilizar el índice se tienen que convertir en FOR automáticamente. Y simplemente utilizar los FOR … IN y los FOR … OF para las cosas para las que fueron diseñados. En el caso de FOR … OF para iterar sobre una Colección Ordenada en la que no nos interesa el índice. Y el FOR … IN para cuando estamos buscando Claves en un Objeto, sobre todo en casos de Depuración

En casos en los que queramos pintar un Objeto se usa el FOR .. IN, por ejemplo: convertimos cursoJS en Objeto (se usa llaves en lugar de corchetes) en lugar de array.

const *cursoJS* = {  
 title: "Curso JS",  
 Profesor: "Pablo"  
};  
  
for (const i in *cursoJS*) {  
 *console*.log(`${i} -> ${*cursoJS*[i]}`);  
}

Para estos casos se suele usar, pero es mejor no usarlo para el FOR IN.

Resultado en consola en la imagen a la derecha.

Las CLASES y sus MÉTODOS

CONSTRUCTOR – GETTERS y SETTERS

Curso de JAVASCRIPT desde CERO #12

<https://www.youtube.com/watch?v=M0FfjG4mhZg&t=12s>

JavaScript es un lenguaje de tipado débil, es decir, **no permite al programador definir el tipo de las variables**. Una variable puede almacenar cualquier tipo de datos durante el tiempo de ejecución, y las operaciones asumen el tipo de variable.

En JS los tipos no existen como tal, es tipado dinámico pero no es fuertemente tipado, para eso tenemos TypeScript.

TypeScript es un lenguaje tipado y tiene diferentes elementos. Gracias a esto, es más fácil la **escalabilidad**, hay mayor **confianza** y **se puede verificar que se ejecute de forma correcta**. Mientras JS tiene 8 tipos de datos dinámicos, TS tiene tipos estáticos al código.

Lo bueno y lo malo que tienen los Objetos en JS es que tienen demasiada libertad, y debemos nosotros restringir esa libertad y no esperar que el lenguaje lo haga por nosotros. En JAVA, por ejemplo, defines 3 propiedades y solo puedes crear y modificar esas 3 propiedades. Pero en JS puedes crear lo que tú quieras.

Lo usual y una buena práctica que todos los atributos que vaya a tener una clase se definan en el constructor.

class Curso {  
 constructor(titulo, dificultad) {  
 this.titulo = titulo;  
 this.dificultad = dificultad;  
  
 this.lecciones = [];  
 }  
}

Hemos agregado un array de “lecciones” pero necesitamos tener métodos que nos permitan “agregar o eliminar” lecciones.

Al crear un método también te encuentras con el problema de la definición del tipo de variable o argumento a utilizar por el método, por las características de JS. Por ejemplo, no podemos definir que el argumento “Leccion” sea un string como tal. Se puede comprobar posteriormente y lanzar un error.

Los tratamientos de errores se pueden hacer con el operador “typeof”.

class Curso {  
 constructor(titulo, dificultad) {  
 this.titulo = titulo;  
 this.dificultad = dificultad;  
  
 this.lecciones = [];  
 }  
  
 agregarLeccion(leccion){

//agrega un elemento al final del array  
 this.lecciones.push(leccion);  
 }  
  
 eliminarUltimaLeccion(){

//elimina el último elemento en un array  
 this.lecciones.pop();  
 }  
}

Ahora quiero crear un Objeto de tipo Curso (instanciar una Clase Curso), el new Curso lo que hace es invocar al Constructor de la Clase.

const *cursoJS* = new Curso('JavaScript', 1);  
  
const *cursoTS* = new Curso('TypeScript', 3);

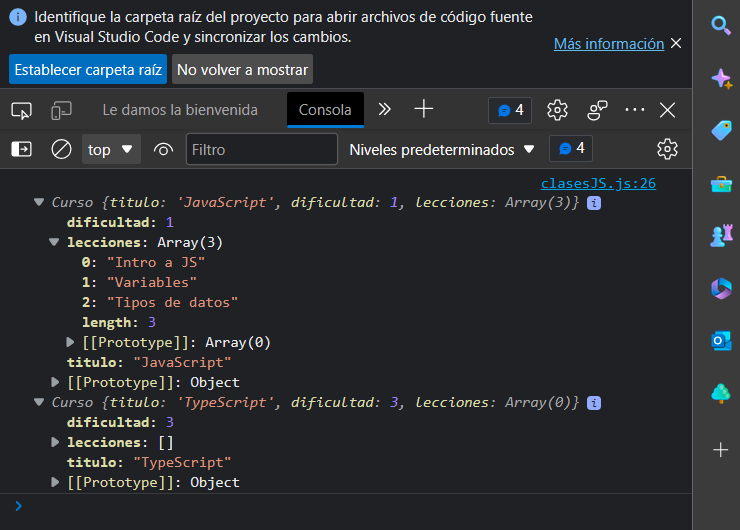
*console*.log(*cursoJS*, *cursoTS*);

Agregando valores al arreglo “lecciones” de la Clase Curso, del siguiente modo

*cursoJS*.agregarLeccion("Intro a JS");  
*cursoJS*.agregarLeccion("Variables");  
*cursoJS*.agregarLeccion("Tipos de datos");

Clase Curso completa hasta ahora.

class Curso {  
 constructor(titulo, dificultad) {  
 this.titulo = titulo;  
 this.dificultad = dificultad;  
  
 this.lecciones = [];  
 }  
  
 agregarLeccion(leccion){  
 this.lecciones.push(leccion);  
 }  
  
 eliminarUltimaLeccion(){  
 this.lecciones.pop();  
 }  
}  
  
const *cursoJS* = new Curso('JavaScript', 1);  
  
const *cursoTS* = new Curso('TypeScript', 3);  
  
*cursoJS*.agregarLeccion("Intro a JS");  
*cursoJS*.agregarLeccion("Variables");  
*cursoJS*.agregarLeccion("Tipos de datos");  
  
*console*.log(*cursoJS*, *cursoTS*);

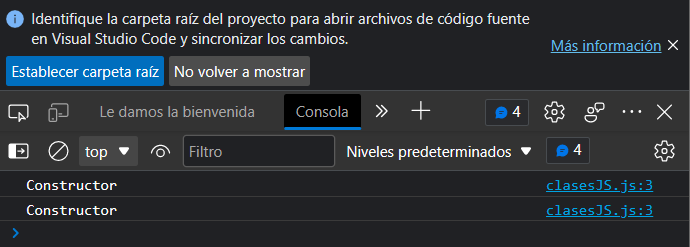


Resultado de la Clase anterior en Pantalla, al llamar al archivo HTML index.html

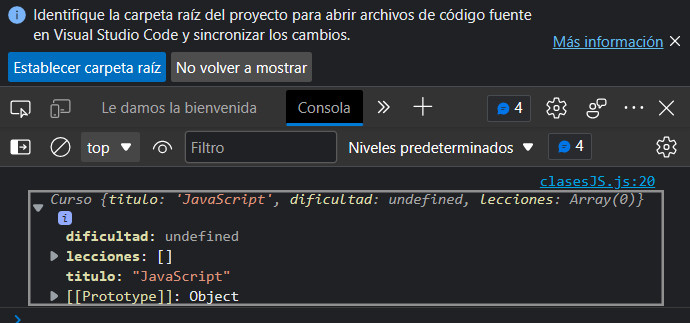
A continuación, otras modificaciones al programa para la Clase Curso.

No se debe llamar dos veces a una constante con el mismo nombre. Es decir, no se puede crear con el mismo nombre.

En el código siguiente se ve que el método “constructor” de la Clase Curso, se ejecuta cada vez que se llama a “new”. Y esta es la razón por la que no puede haber más de un constructor.

class Curso {  
 constructor(titulo, dificultad) {  
 *console*.log("Constructor");  
 this.titulo = titulo;  
 this.dificultad = dificultad;  
  
 this.lecciones = [];  
 }  
  
 agregarLeccion(leccion){  
 this.lecciones.push(leccion);  
 }  
  
 eliminarUltimaLeccion(){  
 this.lecciones.pop();  
 }  
}  
  
const *cursoJS* = new Curso('JavaScript', 1);  
  
const *cursoTS* = new Curso('TypeScript', 3);

Los Lenguajes que tienen tipos y que tienen un poco más de estructurado y se compilan, podemos detectar si estamos llamando a este constructor o a otro en función de los parámetros y del tipo de parámetros que recibamos. Estos constructores son los que llamamos “**sobrecargados**”.

Hacer un constructor por defecto (que inicialice las propiedades) y otro sobrecargado, no es posible en JS, esto generará un error y enviaría el mensaje de error “**A clasesJS.js class may only have one constructor**”. Porque JS, no puede detectar por número de argumentos y por nada. Ya sabemos que no es posible pasarle todos los argumentos, es decir, si por ejemplo, llamásemos al constructor de la clase Curso con un solo parámetro

const cursoJS = new Curso('JavaScript');

console.log(cursoJS);

al hacer eso en la salida de consola puede observarse que el 2do parámetro “dificultad” que no hemos pasado nos saldría “undefined”. Por esta razón no pueden existir constructores sobrecargados.

**Manipulando los Atributos de un Objeto – y que problemas nos puede traer esto.**

Nosotros hasta ahora hemos creado un curso con el atributo dificultad con undefined. Pero podríamos decirle que en cualquier momento que;

cursoJS.dificultad = 1;

consiguiendo lo mismo que si lo hubiésemos llamado con los 2 parámetros al crear la instancia del Curso.

const *cursoJS* = new Curso('JavaScript');  
*cursoJS*.dificultad = 1;  
*console*.log(*cursoJS*);

Es decir, cuando queremos acceder a una propiedad lo haríamos con la sintaxis de punto, como en cualquier objeto. Esto no cambia nada.

**Cosas malas que tiene JS**. Si en algún momento hago malas prácticas, podría algo como lo siguiente:

cursoJS.loquemedelagana = “Loquesea”;

y cuando creamos nuestro Curso nos hemos garantizado que este curso tiene una serie de atributos y de métodos, nada nos impide modificarlos, sobrescribirlos o hacer lo que queramos con ellos.

Por ejemplo, podríamos decir lo siguiente para eliminar un cursoJS.

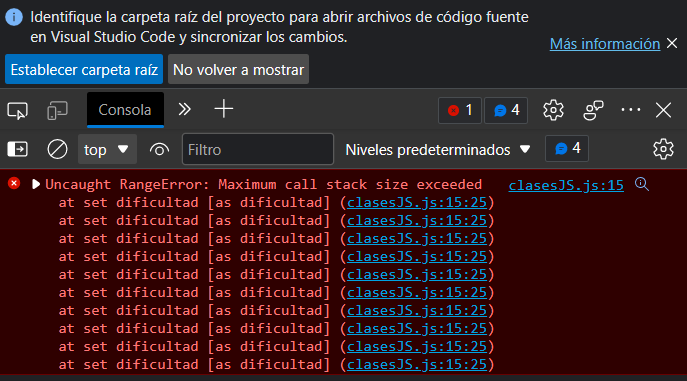
const *cursoJS* = new Curso('JavaScript');  
*cursoJS*.dificultad = 1;  
  
*cursoJS*.eliminarUltimaLeccion = () => *console*.log("No hago nada");  
  
*cursoJS*.eliminarUltimaLeccion();  
  
*console*.log(*cursoJS*);

Esto no es correcto, si queremos programar en JS, tenemos que ser nosotros mismos los que nos pongamos unos límites para ser un buen programador. El hecho que te permitan saltar de un puente no significa que tengas que tirarte. No debemos inventarnos atributos que no existan, ni se modifican los métodos.

Getters y Setters.

Hay una serie de atributos que se pueden modificar de una forma especial.

Si, por ejemplo, quisiera modificar o asignar valores a la propiedad “dificultad”, podría necesitar validar que el valor sea un número y que esté en un rango de dificultad entre 0 y 5, para garantizar ese mínimo. Para eso existen los Getter y los Setter, ¿cómo lo haríamos?

class Curso {  
 constructor(titulo, dificultad) {  
  
 this.titulo = titulo;  
 this.dificultad = dificultad;  
  
 this.lecciones = [];  
 }  
  
 get dificultad(){  
 return this.dificultad;  
 }  
  
 set dificultad(nuevaDificultad){  
 this.dificultad = nuevaDificultad;  
 }  
  
 agregarLeccion(leccion){  
 this.lecciones.push(leccion);  
 }  
  
 eliminarUltimaLeccion(){  
 this.lecciones.pop();  
 }  
}  
  
const *cursoJS* = new Curso('JavaScript');  
  
*console*.log(*cursoJS*.dificultad);

En la forma que llamamos en console.log(cursoJS.dificultad);

Se generaría un bucle infinito porque primero llamaríamos al set, y luego al get, y luego llamaría de nuevo al set y al get, etc. Y esto hace que se exceda el Call Stack.

Cuando tenemos un atributo o una propiedad que se marca para ser utilizada con Getter y Setter, lo que estamos haciendo bajo estos Getter y Setter es esconder otra propiedad distinta a la original. Nosotros estamos exponiendo es una propiedad distinta a la original. Lo que necesitamos exponer por ejemplo es; this.\_dificultad, barra baja “\_” es porque es una sintaxis o convención que tenemos los desarrolladores para indicar que esta variable no se debe tratar directamente. Es una especie de indicativo que esta variable es “privada”.

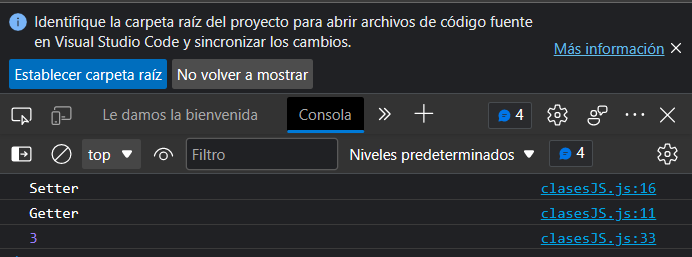
No existen las variables privadas en JS pero esta es una especie de indicativo que nos hace decir a la persona que está viendo el código “oye, esta variable no la debes tocar directamente” es privada.

Cuando llamamos a la siguiente instrucción, enviando el parámetro “dificultad” con valor “1” es invocar el getter.

const *cursoJS* = new Curso('JavaScript', 1);

Y accedemos al “Setter” si usamos la siguiente instrucción;

*cursoJS*.dificultad = 3;

Estas instrucciones generarían un resultado que, al imprimir dentro de cada método por prueba, generaría la siguiente salida:

Primero se ejecuta el Setter y luego el Getter

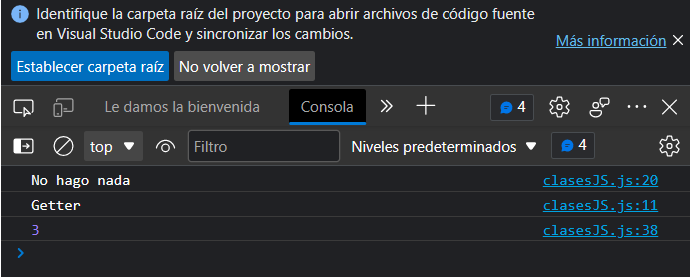
//Setter

cursoJS.dificultad = 3;

//Getter

console.log(cursoJS.dificultad);

Luego de saber este comportamiento, podríamos evaluar el rango de la dificultad en el Setter, del siguiente modo;

class Curso {  
 constructor(titulo, dificultad) {  
  
 this.titulo = titulo;  
 this.\_dificultad = dificultad;  
  
 this.lecciones = [];  
 }  
  
 get dificultad(){  
 *console*.log('Getter');  
 return this.\_dificultad;  
 }  
  
 set dificultad(nuevaDificultad){  
 //console.log('Setter');  
 if (nuevaDificultad >= 0 && nuevaDificultad < 5)  
 this.\_dificultad = nuevaDificultad;  
 else  
 *console*.log("No hago nada");  
 }  
  
 agregarLeccion(leccion){  
 this.lecciones.push(leccion);  
 }  
  
 eliminarUltimaLeccion(){  
 this.lecciones.pop();  
 }  
}  
  
const *cursoJS* = new Curso('JavaScript', 1);  
  
*cursoJS*.dificultad = 3;  
*cursoJS*.dificultad = 7;  
  
*console*.log(*cursoJS*.dificultad);

El código anterior generaría la siguiente salida, vista en la imagen. Lo correcto en el “else” seria hacer un throw new Error(“No hago nada, error de rango”) ;

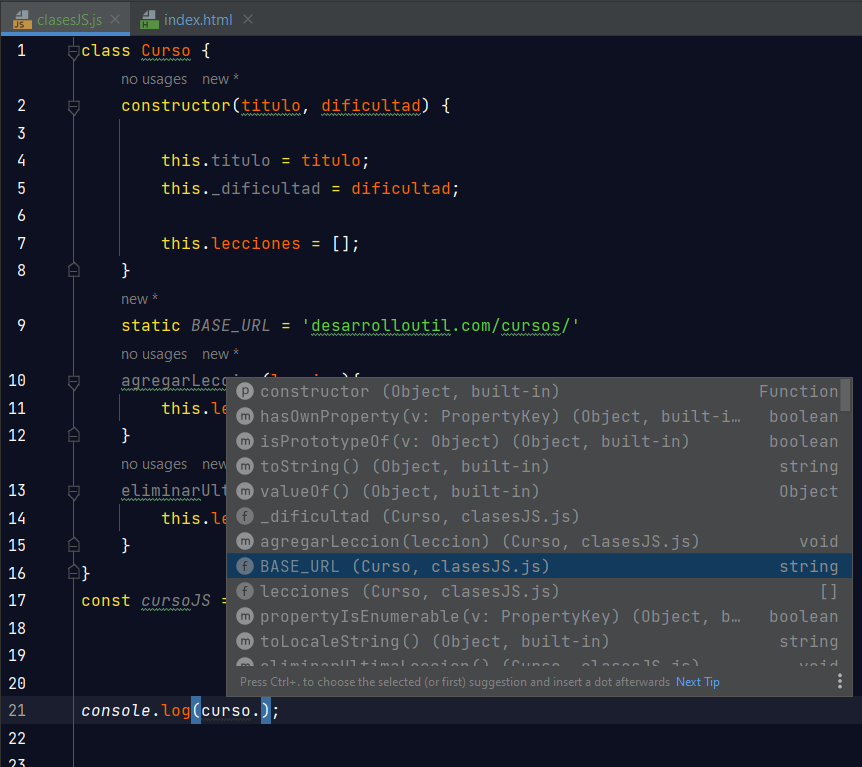
Podemos acceder a la Propiedad como hemos dicho con la barra baja y modificar el valor de la propiedad:

*cursoJS*.\_dificultad = 7;

Una cosa importante que decir, es que en lo Setter y los Getter podemos omitir uno de los 2, convirtiendo una propiedad en “read only” o en “write only”.

Por ejemplo, si omitiésemos el Set dificultad() seria “read only”.

**VARIABLES ESTATICAS**

Las clases pueden tener atributos y métodos estáticos. Pero que hacen referencia es de la propia clase, no son de ningún objeto en cuestión. Es decir, si tenemos un molde de hacer torta. Podríamos hacer una torta de Chocolate, otra será de vainilla, y cada una tendrá sus atributos, sus métodos. Pero hay métodos que vienen puestos directamente para el molde, como ejemplo; limpiar\_molde que no es particularmente de ninguna de las tortas, si no que es específico del molde. En este caso, hay una serie de métodos que normalmente están relacionados con estos “Cursos” pero que no son de un Curso en concreto, sino que son para cualquier Curso. Por ejemplo, podríamos hacer un método para que se calculará una URL. Hacer una URL a partir de este Curso. Podríamos tener una propiedad que fuera la URL base para todos los Cursos. La URL base la tienen que tener absolutamente cada uno de los Cursos. Es decir, pertenece a la Clase.

Esto se hace usando la palabra Clave “static”, por ejemplo;

Pero que pasa cuando quiero hacer referencia a esta variable static “BASE\_URL”

Lo que estamos haciendo es una referencia a la “Clase” no al “Objeto”.

*console*.log(**Curso**.*BASE\_URL*);

Pueden ser Propiedades, pero también pueden ser métodos Static. Por ejemplo;

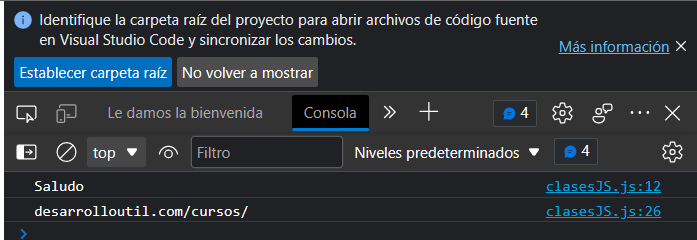
static saludar(){  
 *console*.log("Saludo");  
  
 }

la llamada al método en el código fuera de la clase sería;

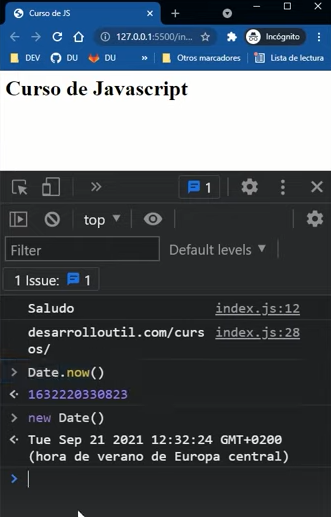
Curso.saludar();

Esto imprimiría lo que tiene el método saludo. No tendríamos que instanciar la clase Curso “const cursoJS = new Curso('JavaScript', 1);” y lo podemos comentar en el código.

class Curso {  
 constructor(titulo, dificultad) {  
  
 this.titulo = titulo;  
 this.\_dificultad = dificultad;  
  
 this.lecciones = [];  
 }  
 static *BASE\_URL* = "desarrolloutil.com/cursos/";  
  
 static saludar(){  
 *console*.log("Saludo");  
  
 }  
 agregarLeccion(leccion){  
 this.lecciones.push(leccion);  
 }  
 eliminarUltimaLeccion(){  
 this.lecciones.pop();  
 }  
}  
//const cursoJS = new Curso('JavaScript', 1);  
  
Curso.saludar();  
  
*console*.log(Curso.*BASE\_URL*);



Resultado del uso de la Propiedad y Método Static.

Podemos usar por ejemplo algo básico como lo siguiente, para calcular el momento actual, es decir, no necesito crear una fecha, solo saber el momento actual:

En esta línea estoy haciendo referencia a la Clase Fecha.

Date.now();

Si quiero crear una fecha para el día de hoy, se puede;

Y en esta línea estoy instanciando un Objeto Fecha.

new Date();

Con las propiedades estáticas.

Por ejemplo tenemos la Clase Number, tiene una serie de constantes. Por ejemplo, Number.Epsilon que muestra la menor diferencia entre dos números.

Number.MAX\_SAFE\_INTEGER es el número más grande que se puede representar en los números de JS.

Number.MAX\_VALUE es el valor más grande que podemos escribir en JS

Number.parseFloat

Number.parseInt

La Cocina del Código

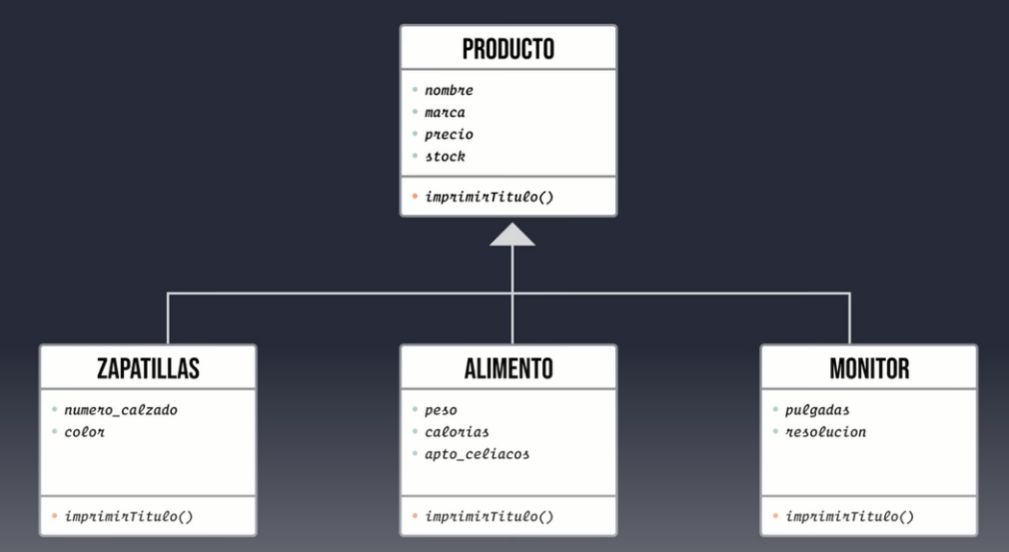
17. HERENCIA POR PROTOTIPOS EN JAVASCRIPT

https://www.youtube.com/watch?v=a2tp64Vtzxs

JavaScript es un Lenguaje Multiparadigma. Significa que podemos programar por medio del paradigma de la POO, o el paradigma de la Programación Funcional. Depende de cual sepamos o nos sintamos más comodos.

Por ahora nos enfocaremos en el Paradigma Funcional. Cómo funciona el modelo de herencia por Prototipos. Al final podrías saber lo que son las “funciones constructoras”, clases y prototipos que existen en JS.

1. Intro
2. Breve Repaso Herencia
3. Delegación de Objetos en JavaScript
4. Los Prototipos son Objetos Dinámicos
5. Ocultamiento de Propiedades
6. La Cadena de Prototipos
7. El Final de la Cadena
8. La Cadena Sólo se Recorre para Leer
9. ¿Cómo saber si un objeto es prototipo de otro?
10. La Cadena de Prototipos Visualmente
11. La Cadena de Prototipos y This
12. 4 Maneras de Usar Prototipos en JS
13. Cierre.

¿Qué es la herencia en programación? Herencia por Clases; es uno de los pilares de la POO.

Este modelo permite que una Clase herede los métodos y propiedades definidos en otra clase.

De esta manera los Objetos de una Sub-Clase van a tener acceso a sus propiedades y métodos, sino también a los que estén definidos en la super clase.

Y podemos agregar más métodos y propiedades en las sub-clases o reescribir los métodos heredados de la super clase. Todo esto con el objetivo de que tengamos menos código, aprovechando que las cosas están definidas una sola vez en las super clases. Y luego gracias a la herencia podemos accederla desde cualquier objeto perteneciente a la sub-clase. Puede que funcione de esta manera en otros Lenguajes de Programación pero en JS la cosa es diferente. El modelo tradicional no dejes que vuelva ahí cuando estás en JS.

No hay forma de escapar a los prototipos en JS.

¿Qué es un Prototipo?

Es como un **delegado**: alguien a quien le delegamos algo, cierta responsabilidad.

Todos los objetos tienen la propiedad oculta “\_\_proto\_\_” y para conectarlo con “cheff” lo haríamos así;

sacha.\_\_proto\_\_ = cheff;

const *cheff* = {  
 tipo: 'Cocina Básica',  
 cocinar: function (plato) {  
 *console*.log(`Cocinando ${plato}`);  
 }  
}  
  
const *sacha* = {  
 nombre: 'sacha',  
 edad: 31,  
 profesion: 'Developer'  
}  
  
//asignamos a la propiedad oculta de sacha el valor de cheff  
*sacha*.\_\_proto\_\_ = *cheff*;

Usar “\_\_Proto\_\_” en una App real no es recomendable, pero con fines educativos estaría bien.

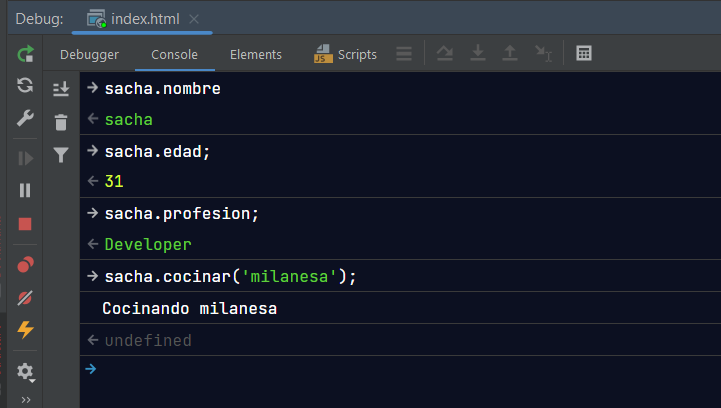
Otra manera de hacer lo mismo, es utilizar la función Object.setPrototypeOf(param1, param2), esta función recibe 2 parametros, el 1ero es el Objeto al cual queremos cambiarle el Prototipo, en este caso es el Objeto “Sasha”, y el 2do, es el Objeto que queremos que sea el Prototipo del 1ero.

Usar este procedimiento tampoco es aconsejable para una App de verdad, pero estamos aprendiendo a usar Prototipos en JS, para estos ejemplos, estas 2 maneras están bien, pero no para Apps reales. Solo son las bases necesarias para trabajar con herramientas avanzadas.

*Object*.setPrototypeOf(*sacha*, *cheff*);

Con esto ya tenemos conectados nuestros objetos. Tenemos conectado a nuestro Objeto “Sasha” y a su Prototipo “Cheff”.

Esto significa que, si ejecutamos el programa, podremos acceder al “nombre” de Sasha, a su “edad” y a su “profesión”. Y no solo eso, le podemos pedir que nos cocine una milanesa por

const *cheff* = {  
 tipo: 'Cocina Básica',  
 cocinar: function (plato) {  
 *console*.log(`Cocinando ${plato}`);  
 }  
}  
  
const *sacha* = {  
 nombre: 'sacha',  
 edad: 31,  
 profesion: 'Developer'  
}  
  
*Object*.setPrototypeOf(*sacha*, *cheff*);

En la imagen de al lado se ve como se accede a las propiedades y al prototipo desde la consola al ejecutar el programa

Cuando accedemos a una propiedad o ejecutamos un método de un objeto, JS primero se fija si esa propiedad o método existen en ese objeto, si existe la accede o lo ejecuta sin problemas. Si no existe se va a fijar al prototipo a ver si existe cómo lo encuentra y lo ejecuta sin dudarlo ese es el secreto que temes que saber y recordar en JS – ~~HERENCIA POR PROTOTIPOS~~, pero es más fácil si la llamamos “**DELEGACIÓN DE OBJETOS**”. Que no se te olvide esto, en JS NO HAY HERENCIA, HAY DELEGACION DE OBJETOS, que le van PIDIENDO COSAS A OTROS OBJETOS, a sus Prototipos. Y los Prototipos son Objetos comunes y corrientes como los que usamos todo el tiempo,

Como todos los objetos, los Prototipos también son Objetos Dinámicos

sí en esta línea de código le agregamos un nuevo método al objeto cheff para cortar un intrediente que recibe por parámetro. Lo podemos llamar luego sin problemas, porque la búsqueda se hace en tiempo de ejecución.

*cheff*.cortar = function (ingrediente) {  
 *console*.log(`Cortando ${ingrediente}`);  
}

luego lo llamamos en tiempo de ejecución, cuando quieras acceder a este método, la búsqueda comienza en sacha y luego continua en su prototipo que en este momento si tiene el método cortar

//Para acceder al nuevo método  
sasha.cortar('Zanahoria');

Pero si llamas al método antes de definirlo, esto generaría un error, porque en ese momento no existe ni en sacha ni en su prototipo.

const *cheff* = {  
 tipo: 'Cocina Básica',  
 cocinar: function (plato) {  
 *console*.log(`Cocinando ${plato}`);  
 }  
}  
  
const *sacha* = {  
 nombre: 'sacha',  
 edad: 31,  
 profesion: 'Developer'  
}  
  
//si en esta línea de código le agregamos un nuevo método al objeto cheff para cortar un intrediente que recibe por parametro  
*cheff*.cortar = function (ingrediente) {  
 *console*.log(`Cortando ${ingrediente}`);  
}  
  
*Object*.setPrototypeOf(*sacha*, *cheff*);  
  
//Para acceder al nuevo método  
sasha.cortar('Zanahoria');

Qué pasaría si un Objeto y su Prototipo tienen una Propiedad que se repite, ¿a cuál accedería JS?

Podríamos agregar una nueva propiedad al objeto sacha, tipo: 'Front-End’, e indicar que tipo de developer es. Ahora cuando queremos acceder a la propiedad tipo de sacha, que valor crees que obtendremos acá, ya que tenemos una propiedad tipo en el objeto, y una propiedad tipo en el prototipo. Si lo ejecutamos, veremos el valor que se encuentra en el Objeto, es el que se mostrará, porque primero busca en el objeto y no en el Prototipo en la búsqueda. Osea que muestra si ponemos en la consola;

sacha.tipo;

Front-End

Pero si no se llega a encontrar ahí entonces continúa buscando en el Prototipo. Hay que tener cuidado con esto que recibe el nombre de ocultamiento de propiedades.

OCULTAMIENTO DE PROPIEDADES (PROPERTY SHADOWING)

**Ocultar las propiedades o métodos** que se encuentran en el prototipo de un objeto porque repiten el mismo nombre. Esto no es ni mas ni menos “ocultar las propiedades” que se encuentran en el Prototipo de un objeto porque repiten el mismo nombre. Quizás eso es lo que queremos, porque queremos modificar el comportamiento de nuestro objeto para cierto método, en vez que se ejecute el método que está en el prototipo.

Toda esta búsqueda a través de objetos nos lleva a lo que es el siguiente tema.

**CADENA DE PROTOTIPOS**

Prototype Chain

Todo el tiempo son los Prototipos en JS, sin darnos cuenta.

Cuando escribimos un Objeto literal estamos haciendo Prototipos

const *pepe* = { nombre: 'Pepe'};

O cuando hacemos un nuevo Array

const *numeros* = [3, 57, 94];

Incluso cuando escribimos una función

const sumar = (a, b) => a + b;

Una expresión regular

const *regexp* = /\d+/gi;

O una fecha también es un Prototipo

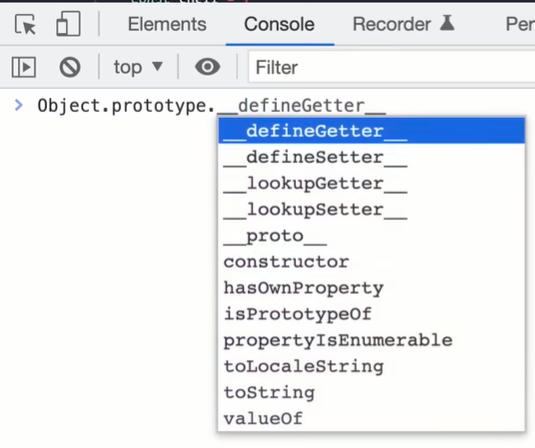
const *ahora* = new *Date*();

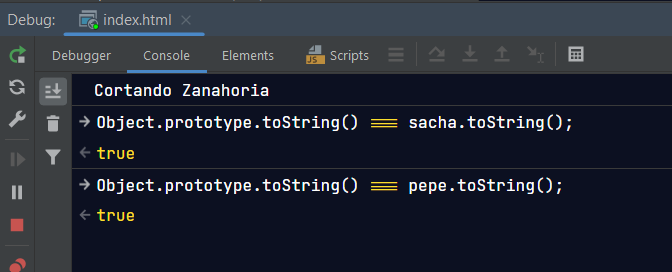
Básicamente en cualquier momento que usemos un objeto (cualquier cosa que no sea un valor primitivo en JS)

Por ejemplo, si hacemos una comparación del método toString() de sacha con el de pepe, como a continuación

sacha.toString() === pepe.toString();

devolverá “true”, vemos que son el mismo método, hacen referencia a la misma función.

En nuestro ejemplo original, el Objeto “cheff” no es el Prototipo final para “sacha”, cada vez que queramos acceder a una “propiedad” de “sacha”, primero se va a buscar a ver si existe en “sacha” si no se encuentra ahí se va a buscar en su prototipo que es “cheff”. Y si no se encuentra ahí tampoco, entonces se va a buscar en un “prototipo” que comparten todos los Objetos en JS. Para los Objetos como “sacha” o “pepe” este prototipo se encuentra en “Object.prototype”. y si abrimos la consola, podemos ver que Object .prototype es un objeto que contiene todos esos métodos y propiedades.



Podemos ver que el método toString() de “sacha” como el de “pepe” hacen referencia al método definido en este prototipo.

**Object.prototype** es el prototipo de todos los Objetos en JS.

Pero por ejemplo los Arrays tienen su propio prototipo que se encuentra en “Array.prototype”. Las fechas los mismo en Date.prototype. También las expresiones regulares RegExp.prototype, Function.prototype, y todos los tipos de datos compleJos que ya vienen en JS, todos los que no sean datos primitivos.

Todo tipo de dato que no es un valor primitivo es un Objeto, realmente es así, porque el prototipo de los array es Array.prototype, pero a su vez su prototipo es Object.prototype, o sea los arrays tienen todas las propiedades y métodos que heredan de los objetos y también definen sus propias propiedades y métodos. Y lo mismo sucede con el resto de los prototipos, todos terminan enlazados con el prototipo de los objetos, y esto viene así por defecto en el lenguaje.

Obviamente se puede reescribir algún método. El objeto string de los arrays no es el mismo que el de los objetos.

Por ejemplo;

numeros.toString() === sacha.toString();

false

Para los “arrays” nos devuelven los elementos que lo componen.

numeros.toString();

3,57,94

Y para los objetos nos devuelven un string que dices object, Object.

sacha.toString();

[object Object]

Si quisiéramos cambiarlo le tendríamos que pasar ese método “toString()” a nuestro objeto (sobreescribiendolo) para evitar que se ejecute el método que se encuentra en el Object.prototype que imprime object, Object.

const *sacha* = {  
 nombre: 'sacha',  
 edad: 31,  
 profesion: 'Developer',  
 tipo: 'Front-End',  
 toString() {  
 *console*.log(this.nombre);  
 }  
};

Resultado de ejecutar en consola; y convierte el nombre en string

sacha.toString(); cadenaDePrototipos.js:14

sacha

AL FINAL DE LA CADENA

¿Hay algo más allá de Object.prototype? no hay nada más, en verdad viene null.

Si preguntamos lo siguiente en la consola: esto nos dará null. Es el final de la cadena de prototipos, no hay nada mas.

Object.prototype.\_\_proto\_\_;

null

Si queremos acceder a una propiedad que no existe sobre un objeto

const pepe = { nombre: ‘Pepe’ };

undefined

pepe.edad;

undefined

vas a recorrer toda la cadena hasta el final y no la encontró, entonces nos va a decir que esa propiedad no está definida “que es undefined”, hasta que vea que no tiene a dónde seguir.

Una cosa importante es la siguiente;

La Cadena de Prototipos siempre se recorre cuando queremos leer una propiedad o ejecutar un método, pero no cuando queremos realizar una asignación.

Es decir, si a nuestro Objeto “sacha” le asignamos un nuevo método “cocinar”.

*sacha*.cocinar = function (ingrediente){  
 *console*.log('Yo no sé cocinar');  
}

Esta función se agrega al objeto “sacha”, pero el prototipo no se ve modificado. Cómo estamos asignando una propiedad no estamos recorriendo la cadena de prototipos, esta solo se recorre cuando queremos “leer una propiedad”.

¿CÓMO SABER SI UN OBJETO ES EL PROTOTIPO DE OTRO?

Si queremos saber si “cheff” es el prototipo de “sacha” podemos preguntar por la propiedad oculta “\_\_proto\_\_”

sacha.\_\_proto\_\_ === cheff

true

como esta propiedad (\_\_proto\_\_) nos apunta directamente al prototipo del Objeto “cheff”, en este caso van a ser iguales. La propiedad \_\_proto\_\_ es una propiedad oculta usada por JS y la usa para ir recorriendo la cadena de prototipos, para ir pasando de prototipo en prototipo. Nosotros no deberíamos usarla en nuestros programas, mucho menos para asignarle un valor.

Así que lo otro que podemos hacer es preguntarle al “cheff” ¿vos sos el prototipo de “sacha”, lo hacemos utilizando el método “isPrototypeOf()”, y está presente en todos los Objetos y nos sirve para ver si ese objeto es el prototipo de otro que le pasamos como parámetro.

cheff.isPrototypeOf(sacha)

true

Lo que tiene de particular este método, es que se fija si ese objeto aparece en algún lugar de la cadena de prototipos del otro objeto que le pasamos por parámetro.

Podemos ver que pasa si reemplazamos “cheff” por “Object.prototype” que es el prototipo al que delegan todos los objetos.

Object.prototype.isPrototypeOf(sacha)

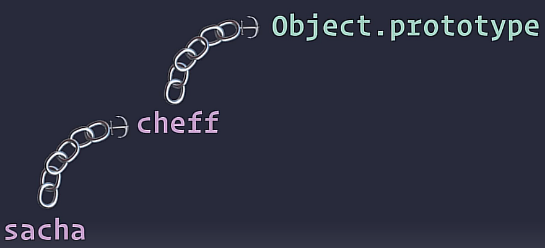
true

Y veremos que igual Object.prototype se encuentra en la cadena de prototipos de “sacha” como en la de todos los Objetos.

Object.prototype.isPrototypeOf(sacha)

true

LA CADENA DE PROTOTIPOS VISUALMENTE



Cuando queremos acceder a una propiedad o métodos del Objeto “sacha”, JavaScript primero la busca en el Objeto mismo, si la encuentra ahí, lee el valor o ejecuta el método. Si no va a buscarlo al siguiente objeto de la cadena “cheff”, al prototipo de la cadena en cuestión y repite lo mismo, si la encuentra, la accede, y si no la encuentra ahí va a buscarla al siguiente eslabón de la cadena “Object.prototype”. Pero recordar que esto lo hace para leer una propiedad o ejecutar algún método pero no para realizar un asignación

sacha.toString();

sacha

Ultimas cosas importantes sobre la cadena de prototipos. ¿qué pasa con THIS cuando usamos prototipos

Podemos escribir un nuevo método en el Prototipo Cheff que sirva para presentarse por consola en que tipo de cocina se especializa.

presentarse: function (){  
 *console*.log(`Hola, soy un cheff especializado en ${this.tipo}`);  
}

Cuando abramos la consola y ejecutemos este método, ¿cuál crees que es el “tipo” que veamos impreso; el de “sacha” o el de “cheff”? (ver video que hizo “La Cocina del Cód”).

Hay que saber cómo se determina el valor que va a tomar “this” cuando se ejecuta el método de un objeto. Hay varios tipos de enlazamiento para “This” y en este caso aplicaremos el lanzamiento implícito, por eso, el valor que va a tomar “This” es el Objeto que está a la izquierda del “punto” o sea el de “sacha”.

Sacha.presentarse();

Resultado:

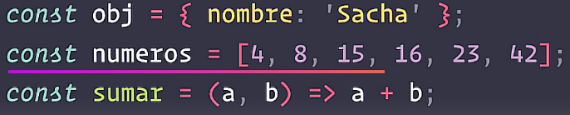
Hola, soy un cheff especializado en Front-End

O sea que toma el valor de la propiedad tipo del objeto “sacha” que es tipo: ‘Front-End’

Y cuando se ejecute la función “presentarse”, por más que está definida en “cheff”, “this” va a ser “sacha”, por eso veremos eso veremos impreso por la consola su tipo que es ‘Front-End’ developer.

**CUATRO (4) MANERAS DE USAR PROTOTIPOS EN JAVASCRIPT**

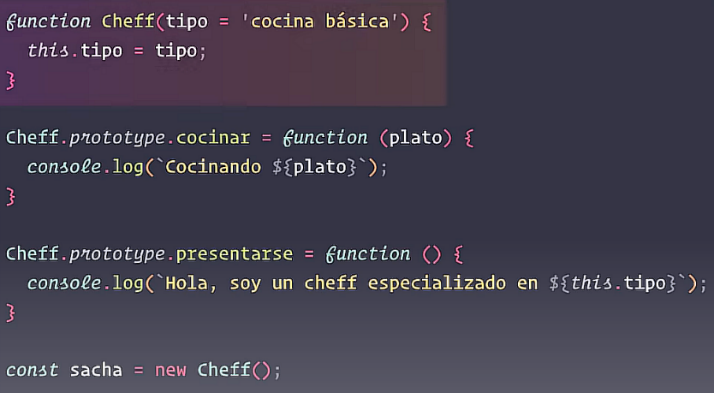
1- La manera más simple que tenemos son los “Objetos literales”.



Cuando creamos un objeto entre llaves, un Array o una función. Cualquier manera que tengamos de crear un Objeto con la propia sintaxis del lenguaje, el lenguaje mismo se encargará de asignar el prototipo correspondiente en cada caso. El Objeto tendrá como prototipo “Object.prototype”, el array tendrá “array prototype” y la función “función prototype”.



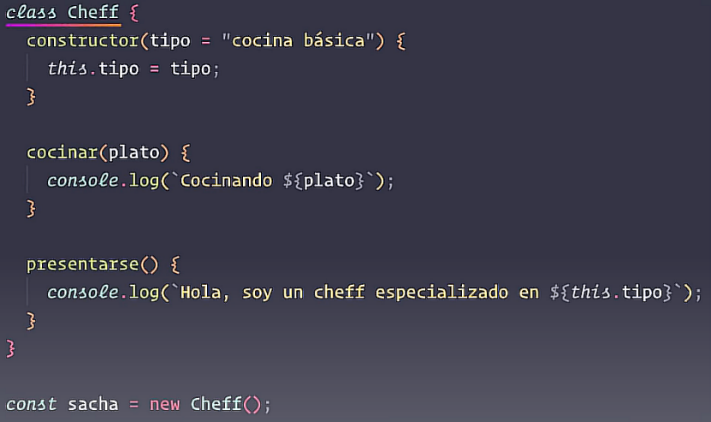
2- Después tenemos el método estático “Object.create(proto)”, este método recibe como parámetro un “prototipo” y nos devuelve un nuevo objeto cuyo prototipo es el que le pasamos.



3- Funciones constructoras y las clases

Podríamos haber definido “Cheff” como una **función** en vez de como **Objeto**. Y podríamos haber creado a sacha con el operador “new”

const **sacha** = **new** **Cheff(**);



Desde ECMA 2015 podemos obtener lo mismo si en vez de escribir una función, escribimos una **clase**. Esta manera puede resultarte familiar de otro lenguaje de programación.

4- La última manera de usar un prototipo es usando; **Object**.**setPrototypeOf**(obj, proto) y \_\_proto\_\_

**Object**.**setPrototypeOf**(sacha, cheff);

En el 1er caso, al objeto “sacha” le vamos a cambiar el prototipo que tiene por el de “cheff”

O directamente se lo asignamos a la propiedad oculta proto con guiones bajos que tienen todos los objetos **JS**

sacha.\_\_proto\_\_ = cheff;

Pero hay que recordar que estas dos maneras no están aconsejadas. En la documentación para los desarrolladores, en ambos casos hay una advertencia que hay que evitar modificar la propiedad “Prototype” de un objeto después que este haya sido creado. Esto es porque el motor de JS optimiza nuestro código a medida que lo va ejecutando y sabe en que posición de la memoria se encuentra el prototipo de cada objeto. Pero si detecta que andamos cambiándolo no va a poder optimizar esa parte del código.

Estas maneras son buenas para enseñar cómo funcionan los prototipos JS, pero no para usarlas en Apps reales.

El SPREAD operator🔴 TODO lo que puedes HACER con TRES PUNTOS ...😎 🟡 Javascript AVANZADO

**Desarrollo Útil**

<https://www.youtube.com/watch?v=_5V6siSlP2k>

<https://www.freecodecamp.org/espanol/news/operadores-rest-spread-de-javascript-cual-es-la-diferencia/>

Este operador, junto con los Arrays y los Objetos, nos permite hacer ciertas operaciones.

Tenemos un arreglo como el siguiente;

const lista1 = [1, 2, 3];

Si quisiéramos copiar el Array anterior. Podría hacerlo del siguiente modo; esta copia seria igual a otro Array. Lista1Copia espera elementos separados por coma, como el “Spread” pone los elementos separados por coma, podría hacer lo siguiente

const *lista1Copia* = [...*list1*];

y es lo mismo que pusiera los elementos separados por coma. Puedo agregar mas elementos al Array, separados por coma.

Recibe un conjunto de elementos separados por coma y los reúne en un Array. En vez de definir parámetro a parámetro lo que defino un operador en la función. Lo que hace es agrupar todos los parámetros que envíes en una llamada; 1, 2, 3,etc y meterlos en un Array.

JavaScript usa tres puntos (...) para ambos operadores, rest y spread. Pero estos dos operadores no son iguales.

La principal diferencia entre rest y spread es que el operador **rest** organiza el resto de algunos valores específicos suministrados por el usuario en un arreglo de JavaScript. Por otro lado, la sintaxis spread expande los iterables en elementos individuales.

Por ejemplo, considere este código que utiliza rest para encerrar algunos valores en un arreglo:

// Usa rest para encerrar el resto de valores específicos proporcionados por el usuario en un arreglo:

**function** **miBio**(primerNombre, apellido, ...otraInfo) {

**return** otraInfo;

}

// Invoca la función miBio pasando cinco argumentos a sus parámetros:

**miBio**(**"Oluwatobi", "Sofela", "CodeSweetly", "Desarrollo Web", "Hombre"**);

// La invocación anterior devolverá:

[**"CodeSweetly", "Desarrollo Web", "Hombre"**]

En el fragmento anterior, utilizamos el parámetro rest **...otraInfo** para colocar **"CodeSweetly", "Desarrollo Web", y "Hombre"** en un arreglo.

Ahora, considere este ejemplo de un operador **spread**:

// Define una función con tres parámetros:

**function** **miBio**(primerNombre, apellido, empresa) {

**return** `${primerNombre} ${apellido} **dirije $**{empresa}`;

}

// Utiliza spread para expandir los elementos de un arreglo en argumentos individuales:

**miBio**(...[**"Oluwatobi", "Sofela", "CodeSweetly"]);**

// La invocación anterior devolverá:

“Oluwatobi Sofela dirije CodeSweetly”

En el fragmento anterior, usamos el operador spread (...) para distribuir el contenido de ["Oluwatobi", "Sofela", "CodeSweetly"] entre los parámetros de miBio().

El código resultante de usar el método “rest”

*console*.clear();  
  
const *lista1* = [1, 2, 3];  
const *lista2* = [4, 5, 6];  
  
function suma(...numeros){  
 const resultado = numeros.reduce((prevValue, nextValue) => prevValue + nextValue, 0);  
 *console*.log(resultado);  
}  
suma(...*lista1*, ...*lista2*);

El resultado del código anterior es el valor de; 21.

Pero puedo pasar 2 Array a la función “suma”. Puedo pasar los parámetros que quiera. Podría pasar directamente una serie de números e igual hace la suma.

suma(1, 2, 3, 4);

el resultado de ejecutar esta llamada a suma sería; 10

OPERACIONES QUE PODEMOS REALIZAR CON LOS OBJETOS

Spread con Objetos

Tenemos el siguiente ejemplo;

*console*.clear();  
  
const *personaje* = {  
 nombre: "Luis",  
 apellido: "Lopez",  
 alergias: {  
 trigo: true,  
 lactosa: false,  
 },  
};  
  
const *direccion* = {  
 calle: "Calle Falsa, 123",  
 localidad: "Paris",  
 pais: "Francia",  
};

Podemos utilizar el Spread ya que un Objeto no deja de ser un conjunto de Propiedades separadas por coma. No es como en un Array nativo en el que podemos tener números, string, lo que sea, aquí tengo en el ejemplo del fragmento de código anterior, Propiedades que son; clave, valor. Con Spread, puedo unir, copiar objetos.

Por ejemplo, puedo tener una unión de los objetos; personaje unidas a la de dirección en un nuevo Objeto Persona utilizando Spread de “Personaje” con “dirección” del siguiente modo;

const *persona* = {  
 ...*personaje*,  
 ...*direccion*}  
  
*console*.log(*persona*);

El resultado es un objeto resultante de la unión de “Personaje” con “dirección” como un único objeto, distribuye todas las propiedades del objeto en un resultante;

{

nombre: 'Luis',

apellido: 'Lopez',

alergias: { trigo: true, lactosa: false },

calle: 'Calle Falsa, 123',

localidad: 'Paris',

pais: 'Francia'

}

También puedo hacer una copia de personaje, y a la vez agregar más elementos e imprimirlo;

const *personajeCopia* = {  
 ...*personaje*,  
 edad: 45  
}  
  
*console*.log(*personajeCopia*);

También puedo unir dos objetos y agregarle más propiedades adicionales.

const *persona* = {  
 ...*personaje*,  
 ...*direccion*

*edad: 45*}  
  
*console*.log(*persona*);

Cuando copiemos con un Spread hay que tener mucho cuidado, porque es una “copia débil”, ya que no es una “copia profunda”, es decir, no nos copia cada uno de los elementos. Si luego los modificamos, se modificarán en el original en función de donde esté. Es decir, cuando hacemos la copia, solo se copian los detalles del 1er nivel de propiedades; nombre, apellido, calle, localidad, país. Cuando me encuentro con otro objeto, si yo cambié el valor de alergias entero no hay problemas, pero si solo cambio el valor de “trigo” que ya va a un nivel por debajo, es una propiedad del objeto “alergias” que a su vez es una propiedad del objeto “personaje”, hay 2 niveles y tengo un problema porque eso va por referencia y no por valor.

Es decir, si hago un cambio luego de haber creado una copia de una unión de un objeto, esto crea una copia de ese objeto. Y luego hago un cambio en una propiedad, ese valor nuevo no se verá registrado en la impresión porque se ha hecho la copia antes de modificar el Objeto. En este caso modificamos la propiedad “nombre” del objeto personaje. Esto porque es en el 1er nivel de las propiedades de un objeto.

const *persona* = {  
 ...*personaje*,  
 ...*direccion*}  
  
*personaje*.nombre = "José";  
*console*.log(*personaje*);  
*console*.log(*persona*);

Pero sin en lugar de hacer el cambio al 1er nivel, lo hago en el subObjeto “alergias” en la propiedad “lactosa” y la pongo a true, esto se verá también reflejado en la copia, porque es un cambio que afecta un valor por referencia.

const *persona* = {  
 ...*personaje*,  
 ...*direccion*}  
  
*personaje*.alergias.lactosa = true;  
*console*.log(*personaje*);  
*console*.log(*persona*);

Resultado;

{

nombre: 'Luis',

apellido: 'Lopez',

alergias: { trigo: true, lactosa: **true** }

}

{

nombre: 'Luis',

apellido: 'Lopez',

alergias: { trigo: true, lactosa: **true** },

calle: 'Calle Falsa, 123',

localidad: 'Paris',

pais: 'Francia'

}

Por ello hay que tener cuidado con estos tipos de objetos, si se tienen objetos sencillos no hay problema. Si no, hay que tener cuidado ya que solo nos permite copiar hasta el primer nivel de profundidad.

El objeto **Promise**.

<https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Using_promises>

PROMESAS y ASYNC - AWAIT 😎 THEN ? ✅ CATCH? ⛔ Curso de JAVASCRIPT desde CERO #16

https://www.youtube.com/watch?v=ksg6SDwllDs

Desarrollo Útil

A continuación, el esqueleto básico de lo que es una “petición” de tipo XHR básica desde un archivo JS.

XMLHttpRequest (XHR) es un objeto especial de JS que permite realizar peticiones HTTP asíncronas (AJAX) de forma nativa desde Javascript. Se trata de la primera implementación que existió en ECMAScript antes de surgir **fetch**, el estándar actual.

const *req* = new XMLHttpRequest();  
*req*.open("GET", "./numero.txt");  
  
*req*.onload = () => {  
 if (*req*.status === 200){  
 *console*.log(*req*.responseText);  
 }  
 else{  
 *console*.log('Error');  
 }  
}  
  
*req*.onerror = () => {  
 *console*.log("Error")  
}  
*req*.send();

El objeto XMLHttpRequest se creó originalmente para realizar peticiones HTTP a ficheros .xml externos desde Javascript. Actualmente, se realizan las mismas operaciones, pero con ficheros JSON, ya que son mucho más habituales en el ecosistema Javascript como almacenamiento ligero de datos.

El mecanismo principal de peticiones HTTP mediante XMLHttpRequest es muy sencillo, aunque se vuelve un poco más complejo a medida que vamos realizando comprobaciones y detalles relacionados, ya que se realiza todo más a bajo nivel que su equivalente moderno fetch.

El archivo de texto que utiliza para leer el archivo JS, es “numero.txt”, el cual solo contiene un número “5”.

Vamos a pasar la respuesta a “número” para hacer una operación de multiplicación, ver ejemplo a continuación, en esto se reutiliza la petición “request” XHR HTTP que estamos asociando, con el uso de “request y la función flecha”:

const toNumber = n => Number(n);  
const multiply = n => n \* 2;  
const print = n => *console*.log(n);  
  
const request = (url) => {  
 const req = new XMLHttpRequest();  
 req.open("GET", url);  
  
 req.onload = () => {  
 if (req.status === 200){  
 let number = toNumber(req.responseText);  
 number = multiply(number);  
 print(number);  
 }  
 else{  
 *console*.log('Error al cargar');  
 }  
 }  
 req.onerror = () => {  
 *console*.log("Error de red")  
 }  
 req.send();  
};  
  
request("./numero.txt");

request("./numero2.txt");

Podemos reutilizar la petición “request” XHR HTTP con otros archivos, por ejemplo, para pedir un 2do archivo:

request("./numero2.txt");

Pero en este último caso del archivo “numero2.txt” como no existe entraríamos por el “else” de “if (req.status === 200){” debido a que se origina un error.

El otro error, el de red, podría ocurrir si llamamos a una dirección web inexistente (el dominio no existe);

request("http://noexiste.com");

Hasta aquí como vá el código que hemos realizado en el archivo “promesas.js”.

const toNumber = n => Number(n);  
const multiply = n => n \* 2;  
const print = n => *console*.log(n);  
  
const request = (url) => {  
 const req = new XMLHttpRequest();  
 req.open("GET", url);  
  
 req.onload = () => {  
 if (req.status === 200){  
 let number = toNumber(req.responseText);  
 number = multiply(number);  
 print(number);  
 }  
 else{  
 *console*.log('Error al cargar');  
 }  
 }  
 req.onerror = () => {  
 *console*.log("Error de red")  
 }  
 req.send();  
};  
  
request("./numero.txt");  
request("./numero2.txt");  
request("http://noexiste.com");

Hay problema de usabilidad, porque quiero cargar un archivo de tipo “txt” pero el comportamiento que quiero hacer a posteriori, es que quiero pasárselo como “función” como “callback”, y pasarle también como “callback” el error.

1ero pasaríamos como “callback” la función “then”, luego utilizamos el “then” en el if donde validamos el status.

Luego tenemos que pasar una función que llamaremos “thenFn” esta recibe como parámetro un número “n” , y le pasamos en la llamada al final del código la función “thenFn”, y además podríamos pasar una función de “catch”, pero esta palabra colisiona con el lenguaje porque es una palabra reservada, entonces la usamos modificada “catchFn”. Entonces necesitaríamos una “catchAndPrint”

Es importante que al capturar errores se haga de la manera correcta, instanciando la clase Error “new Error()”, como se vé a continuación

catchFn(**new Error**('Error al cargar'));

CODIGO COMPLETO A CONTINUACION

const toNumber = n => Number(n);  
const multiply = n => n \* 2;  
const print = n => *console*.log(n);  
  
const thenFn = (n) => {  
 let number = toNumber(n);  
 number = multiply(number);  
 print(number);  
}  
  
const catchAndPrint = (err) => *console*.log(err);  
  
const request = (url, then, catchFn) => {  
 const req = new XMLHttpRequest();  
 req.open("GET", url);  
  
 req.onload = () => {  
 if (req.status === 200){  
 then(req.responseText);  
 }  
 else{  
 catchFn(new Error('Error al cargar'));  
 }  
 }  
 req.onerror = () => {  
 catchFn(new Error("Error de red"));  
 }  
 req.send();  
};  
//pasamos or parametro el archivo y la función CallBack "thenFn"  
request("./numero.txt", thenFn, catchAndPrint);

Podriamos generar errores, cambiando el nombre del archivo de texto que estamos llamando, o poniendo la ruta inexistente

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Pudiera usar un “thenAlt” alternativo, y no necesito cambiar la lógica que hay en el “request”, y de hecho no lo tengo que tocar, simplemente tendría que pasarle un “thenAlt” al “request” y el request no se vería afectado, simplemente lo estaríamos manejando como “callBack” y con esto haríamos que solamente imprima el valor contenido en el archivo de texto;

Normalmente cuando queremos anidar callbacks, callbacks y callbacks, etc. Haciendo lo que se llama un “callback Hell”

¿Qué es el Callback Hell? Son como callbacks anidados.

<https://codearmy.co/que-es-el-callback-hell-y-como-evitarlo-4af418a6ed14>

¿Qué es el Callbackl?

Los callbacks es solo el nombre que se usa para referirse a **ciertas funciones de JS**. No existe algo llamado **callback** en el lenguaje JS, es solo una convención. En lugar de devolver inmediatamente un resultado como la mayoría de las funciones, las funciones que utilizan callbacks toman un tiempo para producir un resultado. La palabra “asíncrono”, también conocido como “async” solo significa “**requiere algo de tiempo**” o “**sucede en el futuro**, **no en este momento**”. Por lo general, los callbacks solo se utilizan al hacer **I/O**, por ejemplo, descargar cosas, leer archivos, hablar con bases de datos, etc.

Dado el desorden que se creaba al principio con los “callbacks”, desarrollaron un mecanismo para estandarizar los procesos con “callbacks” y de ahí que nacen las “promesas”. De modo, que el “catch y el then” que hemos utilizado tienen que ver con las promesas.

VEAMOS COMO CREAR UNA PROMESA

Una [Promise](https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Promise) (promesa en castellano) es un objeto que representa la terminación o el fracaso de una operación asíncrona.

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

Reutilizaremos el código anteriormente creado, para crear una promesa, si queremos hacer una promesa debemos retornar una promesa desde el método “request” que creamos.

La promesa utiliza 2 parametros; el “resolve” y el “reject”, resolve es el valor o el que hemos denominado “then”, y reject es el error capturado. Y con eso en el método “request” ya no necesitaremos el “catchFn” ni el “then” en sus parámetros. De ese modo necesitaremos pasar los “then y los catch” de una manera especial. Un “resolve” en una promesa es como un “return”. En el momento que se ejecuta “resolve” es como si fuera una especie de “return” de la promesa. Y es lo que le llegará a ese “then” que pasaremos en la llamada al request.

Con la instrucción siguiente lo que le llega al método “request” es el “5” (data)

request("./numero.txt").then((data) => *console*.log(data));

Pero podríamos hacer con el “then” lo siguiente.

request("./numero.txt").then(toNumber);

con esta instrucción no se muestra nada, pero lo especial de esto es que lo que devuelve la función then, a su vez, se pasa al siguiente “then”. De modo que podemos seguir agregando “then” y vamos a poder pasar el resultado del then anterior al siguiente then como parámetro de entrada. Es decir, que le puedo decir, que lo que quiero es multiplicar. Y podemos pasarle otro then y decirle aquí que lo que quiero ya, es imprimir. Y vemos que la magia de las promesas hace su efecto.

request("./numero.txt").then(toNumber).then(multiply).then(print);

con la instrucción anterior obtenemos el valor de “10” en la salida de consola. En esta promesa lo que tenemos primero, es un “5”, luego éste se ha pasado al primer then(toNumber), luego este number que ya es un “5” convertido a número lo hemos multiplicado por 2 “then(multiply)” y nos dá “10” como resultado y después lo hemos impreso con “then(print)”. Por lo cual el “thenFn” ya no tiene sentido, y quitamos esa función de este código. Es decir, ya no tenemos que crear funciones complejas, ni gestionar un “callback hell” por ninguna parte, porque podemos anidar estos “then” de forma que nos queda bastante sencillo.

Si el archivo que escribimos no pusimos bien el nombre, lanzará un error no controlado, entonces para complementar el código de los “then” tenemos que agregar “catch” para controlar la excepción. Recordemos que un catch lo que recibe es lo que le pasamos al “reject” en este caso puede ser un “error al cargar” o un “error de red”. Entonces en este catch, podemos hacer cosas como “catchAndPrint” que es una función que captura este error. Y lo imprime “Error: Error al cargar”

request("./numero2.txt").then(toNumber).then(multiply).then(print).catch(catchAndPrint);

Estos “then” y estos “catch” podemos seguir anidándolos tantas veces como queramos.

request("./numero2.txt")  
 .then(toNumber)  
 .then(multiply)  
 .then(print)  
 .catch(catchAndPrint);

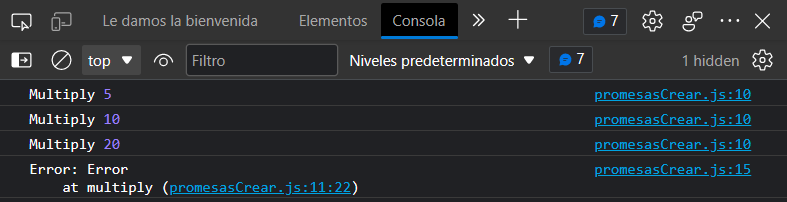
Si, por ejemplo, ahora ya no solo lo quiero multipicar por 2, si no por 4, por 8, etc, puedo meter mas “then(multiply)” anidados uno tras otro.

Normalmente el comportamiento de una promesa, es que el último “then” ya no es necesario que retorne nada.

**Promise catch**

Durante la ejecución los “then” es posible que sucedan errores, que se capturan en el “catch”. Cuando encontramos un error en un “then”, saltamos al “catch”.

//veremos como crear una promesa  
new Promise((resolve, reject) => {  
 resolve(5)  
 reject(new Error())  
});  
  
const toNumber = n => Number(n);  
const multiply = n => {  
 *console*.log("Multiply", n);  
 if(n > 10) throw new Error("Error");  
 return n \* 2;  
}  
const print = n => *console*.log(n);  
const catchAndPrint = (err) => *console*.log(err);  
  
const request = (url) => {  
 return new Promise((resolve, reject) => {  
 const req = new XMLHttpRequest();  
 req.open("GET", url);  
  
 req.onload = () => {  
 if (req.status === 200){  
 resolve(req.responseText);  
 }  
 else{  
 reject(new Error('Error al cargar'));  
 }  
 }  
 req.onerror = () => {  
 reject(new Error("Error de red"));  
 }  
 req.send();  
 })  
};  
//request("./numero.txt").then((data) => console.log(data));  
request("./numero.txt")  
 .then(toNumber)  
 .then(multiply)  
 .then(multiply)  
 .then(multiply)  
 .then(multiply)  
 .then(multiply)  
 .then(print)  
 .catch(catchAndPrint);

Lo primero que sucede en el código anterior, es que convertimos el “5” a número. Luego imprimimos el 5, luego hacemos la 1era llamada al método “multiply”, imprimimos el 5 “**console.log("Multiply", n)**”, luego comprobamos si “**n > 10**”, luego llamamos al 2do “multiply”, imprimos 10, comprobamos si “**n > 10**”, y aún es no, y n es “10\*2” (20), luego llamamos al 3er “multiply”, imprimimos 20, hacemos la comprobación “**n > 10**” (20 > 10) y como es verdadero lanzamos un error “throw new Error("Error")” y aquí se para la cadena de “then”, los siguientes dos “multiply” y el “print” no se llegan a ejecutar nunca. Entonces lo que sucede acá es que encontramos un error en un “then” y entonces saltamos al “catch”. De modo que este “catch” no es solo para el “request” si no para cualquiera de los “then”. Y podemos poner más de un “catch”. Pero hay que quitar el “;” del anterior “catch” porque si no lo interpreta como un “final de línea”. Sólo la última instrucción de la llamada al “request” debe tener el “;”. Pero vemos que en cualquier momento el 1er “catch” se ejecuta y no hace nada más, no funciona como los “.then”.

¿Qué complicaciones podemos entonces tener con los “catch”? ¿porque se pueden poner 2? Porque en este caso lo que podemos hacer con ese 1er “catch” es capturar y devolver un valor, y que entonces el siguiente “then” se ejecute con ese valor. Entonces recordar que estamos en el 3er “then” y que en este ha dado un “error”. En el ejemplo que estamos realizando, agregamos al código del “catchAndPrint” un “return 1“, de forma que ese valor devuelto lo toma el siguiente “then” (el cuarto) e imprime 1, valora si 1 es mayor que 10 y no es así, calcula “1 x 2” y vamos al 5to “then”, imprime el 2 y verifica si 2 es mayor que 10, no se cumple, y calcula 2 x 2 que es 4, luego imprime 4 con la instrucción “print”.

Este tipo de situaciones en los que se dispara un “catch” es posible que sea útil en determinados casos. Para hacer una tarea especial por decir así. Lo que hay que tener mucho cuidado, por que estos “catch” no se ejecutarán en serie, pero cuando se captura un error, se sigue la cadena de “then”. Y no paramos la ejecución, si ponemos “then” por debajo de ese “catch” todo lo que vaya hacia abajo se va a seguir ejecutando. Y si en alguno de esos “then” ocurre un error, se salta al siguiente “catch” y si hay otro “then” después, se va a seguir ejecutando, por ello hay que tener mucho cuidado y vigilar el flujo de ejecución de ejecución de estos “then” y estos “catch”.

//veremos como crear una promesa  
new Promise((resolve, reject) => {  
 resolve(5)  
 reject(new Error())  
});  
  
const toNumber = n => Number(n);  
//const multiply = n => n \* 2;  
const multiply = n => {  
 *console*.log("Multiply", n);  
 if(n > 10) throw new Error("Error");  
 return n \* 2;  
}  
const print = n => *console*.log(n);  
const **catchAndPrint** = (err) => {  
 *console*.log(err);  
 **return 1**;  
}  
  
const request = (url) => { …  
};  
//request("./numero.txt").then((data) => console.log(data));  
request("./numero.txt")  
 .then(toNumber)  
 .then(multiply)  
 .then(multiply)  
 .then(multiply)  
 .catch(**catchAndPrint**)  
 .then(multiply)  
 .then(multiply)  
 .then(print)  
 .catch(catchAndPrint);

INSTRUCCION “finally”.

Hay momentos que independiente mente sea “then” o sea “catch” queremos que realice una acción, y para eso tenemos el método “finally()”. Este método se va a ejecutar, si o sí, independientemente si entra por el “then” o por el “catch”. Y suele ser un método que sea al final. Por ejemplo, tenemos una petición a una BD y hacemos una búsqueda de un “usuario”, lo encuentra en el “then” y me devuelve un “usuario”, no lo encuentras, en el “catch” devolvemos un “not found “, pero al final siempre debe cerrarse la conexión con la BD. Es decir, tanto si no encuentras como si no, la conexión debe cerrarse.

Finally, se ejecuta siempre. Y podemos, tener tantos “finally” como necesitemos, y no habría ningún tipo de problema. Incluso el “finally” podría ejecutarse antes del “catch”.

request("./numero.txt")  
 .then(print)  
 .catch(catchAndPrint)  
 .finally(() => *console*.log("Se acabó la petición"));

o, finally antes del catch. Si tenemos un error en el nombre del archivo, se ejecutará 1ero de todas formas el finally, y luego “Error: Error al cargar”

request("./numero2.txt")  
 .then(print)

.finally(() => *console*.log("Se acabó la petición"));  
 .catch(catchAndPrint);

Archivo de uso en proyecto: promesasAsyncAwait.js

**ASYNC / AWAIT**

Existe una sintaxis especial para trabajar con promesas de una forma más confortable, llamada “async/await”. Es sorprendentemente fácil de entender y usar.

Funciones async

Comencemos con la palabra clave **async**. Puede ser ubicada delante de una función como aquí:

**async function** **f**() {

**return** 1;

}

La palabra “async” ante una función significa: que la función siempre devolverá una promesa. Otros valores serán envueltos y resueltos en una promesa automáticamente.

**async function** f() {

**return** 1;

}

f().then(alert); // 1

…Podríamos explícitamente devolver una promesa, lo cual sería lo mismo:

**async function** f() {

**return** Promise.resolve(1);

}

f().then(alert); // 1

Entonces, **async** se asegura de que la función devuelva una promesa, o envuelve las no promesas y las transforma en una. Hay otra palabra, **await**, que solo trabaja dentro de funciones async y es muy interesante.

Await

La sintaxis:

// funciona solamente dentro de funciones async

**let value** = **await** promise;

**await** hace que JavaScript espere hasta que la **promesa** responda y devuelve su resultado.

Aquí hay un ejemplo con una promesa que resuelve en 1 segundo:

**async function** f() {

**let** promise **= new** **Promise**((resolve, reject) => {

**setTimeout**(() => **resolve**("¡Hecho!"), 1000)

});

***let*** result ***=* *await*** promise*;* *// espera hasta que la promesa se resuelva (\*)*

**alert**(result); // "¡Hecho!"

}

f();

La ejecución de la función es pausada en la línea **(\*)** y se reanuda cuando la promesa responde, con **result** volviéndose su resultado. Entonces el código arriba muestra “¡Hecho!” en un segundo.

Enfaticemos: **await** literalmente suspende la ejecución de la función hasta que se establezca la promesa, y luego la reanuda con el resultado de la promesa. Eso no cuesta ningún recurso de CPU, porque el motor de JavaScript puede hacer otros trabajos mientras tanto: ejecutar otros scripts, manejar eventos, etc.

Es simplemente una sintaxis más elegante para tener el resultado de una promesa que **promise.then**, es más fácil de leer y de escribir.

No se puede usar **await** en funciones comunes

Si tratamos de usar **await** en una función no async, tendremos un error de sintaxis:

**function** f() {

**let** promise = Promise.**resolve**(1);

***let*** *result =* ***await*** *promise; //* ***Syntax error***

}

Es posible que obtengamos este error si olvidamos poner **async** antes de una función. Como se dijo, “**await**” solo funciona dentro de una función **async**.

Resumen

El comando **async** antes de una función tiene dos efectos:

Hace que siempre devuelva una promesa.

Permite que sea usado **await** dentro de ella.

El comando **await** antes de una promesa hace que JavaScript espere hasta que la promesa responda. Entonces:

1. Si es un error, la excepción es generada — lo mismo que si **throw error** fuera llamado en ese mismo lugar.
2. De otro modo, devuelve el resultado.

Juntos proveen un excelente marco para escribir código asincrónico que es fácil de leer y escribir.

Con **async/await** raramente necesitamos escribir **promise.then/catch**, pero aún no deberíamos olvidar que están basados en promesas porque a veces (ej. como en el nivel superior de código) tenemos que usar esos métodos. También **Promise.all** es adecuado cuando esperamos por varias tareas simultáneas.

Usualmente usamos nuestro cerebro y pensamos que las cosas o tareas se ejecutan una detrás de otra y nos cuesta ver todo este tipo de “asincronismo”.

Puede haber promesas anidadas, “then” con otros “then” dentro. Y para eso tenemos las instrucciones “ASYNC” y “AWAIT”.

La primera norma de “Async – await” es que solamente se puede utilizar “await” dentro de una función.

Nosotros no podemos hacer una cosa como la siguiente, porque nos generaría un error:

await request("./numero.txt")  
 .then(print)  
 .catch(catchAndPrint)  
 .finally(() => *console*.log("Se acabó la petición"));

la salida en consola sería la siguiente, con un error indicado en la línea 40, diciendonos que “await” es válida solo en funciones “asyn”:

Uncaught SyntaxError: **await** is only valid in **async functions** and the top level bodies of modules (at promesasAsyncAwait.js:40:1)

Es decir, “await” es válida en el contexto de una función asíncrona.

Para resolver el error, anterior tendríamos que hacer un código como el siguiente, con una llamada a “async”:

const call = **async** (url) => {  
 await request(url)  
 .then(print)  
 .catch(catchAndPrint)  
}  
call('/.numero.txt');

con el código anterior, estamos esperando a que esta promesa termine, con el objetivo de hacer estas cosas en secuencia. Podemos hacer un encadenamiento de promesas. En el siguiente ejemplo encadenaremos 2 promesas.

const call = async (url) => {  
 const response = await request(url);  
 const response2 = await request(url);  
  
 const n1 = toNumber(response);  
 const n2 = toNumber(response2);  
 print(n1 \* n2);  
}  
  
call("./numero.txt");

En el Código anterior obtenemos el resultado de múltiplicar 2 números que se han obtenido de la llamada al método “request” que es visto como una promesa en cada llamada. Luego llamamos al método toNumber() con cada response para obtener el valor de texto y tranformarlo a número. Por último, se realiza una multiplicación por esos 2 números.

En el código anterior, pudimos hacer un tratamiento de errores con el try… catch… en la sintaxis de ASYNC / AWAIT, tal y como se ha hecho de manera convencional, a continuación el código modificado.

const call = async (url) => {  
 try{  
 const response = await request(url);  
 const response2 = await request(url);

const n1 = toNumber(response);  
 const n2 = toNumber(response2);  
 print(n1 \* n2);  
 }catch (error) {  
 catchAndPrint(error);  
 }  
}  
call("./numero.txt");

El código anterior podría tratarse con un try… catch por cada promesa, como vemos a continuación y hacemos una llamada a un nombre de archivo equivocado; call("./numero2.txt") pero al final como usamos dentro del error generado un valor por default, asignado a cada response, obtenemos la multiplicación de 7 x 8 que es 56 en consola.

const call = async (url) => {  
 let response, response2;  
 try {  
 response = await request(url);  
 }catch (error) {  
 catchAndPrint(error);  
 response = 7;  
 }  
  
 try{  
 response2 = await request(url);  
 }catch (error) {  
 catchAndPrint(error);  
 response2 = 8;  
 }  
  
 const n1 = toNumber(response);  
 const n2 = toNumber(response2);  
 print(n1 \* n2);  
}  
call("./numero2.txt");

Al marcar un método con ASYNC / AWAIT lo que haces es convertir un método en una promesa. Por ejemplo, con el método “prom” siguiente, lo que haremos es convertirlo en una promesa.

**const** **prom** = **async** () => {}

simplemente es una sintaxis que nos permite escribir esto de una manera más sencilla.

**Arrays de PROMESAS ¿Cómo las resolvemos? JavaScript AVANZADO**

<https://www.youtube.com/watch?v=FXEjZ41gfNA>

<https://developer.mozilla.org/es/docs/Web/API/Fetch_API/Using_Fetch>

La [API Fetch](https://developer.mozilla.org/es/docs/Web/API/Fetch_API) proporciona una interfaz JavaScript para acceder y manipular partes del canal HTTP, tales como peticiones y respuestas. También provee un método global [fetch() (en-US)](https://developer.mozilla.org/en-US/docs/Web/API/fetch) que proporciona una forma fácil y lógica de obtener recursos de forma asíncrona por la red.

Este tipo de funcionalidad se conseguía previamente haciendo uso de **XMLHttpRequest**. **Fetch** proporciona una alternativa mejor que puede ser empleada fácilmente por otras tecnologías como **Service Workers** (en-US). Fetch también aporta un único lugar lógico en el que definir otros conceptos relacionados con HTTP como **CORS** y extensiones para **HTTP**.

Una petición básica de **fetch** es realmente simple de realizar. Eche un vistazo al siguiente código:

fetch('http://example.com/movies.json')

.then(response => response.json())

.then(data => console.log(data));

Aquí estamos recuperando un archivo **JSON** a través de red e imprimiendo en la consola. El uso de **fetch()** más simple toma un argumento (la ruta del recurso que quieres obtener) y devuelve un objeto **Promise** conteniendo la respuesta, un objeto [**Response**](https://developer.mozilla.org/es/docs/Web/API/Response).

Esto es, por supuesto, una respuesta HTTP no el archivo JSON. Para extraer el contenido en el cuerpo del JSON desde la respuesta, usamos el método [json() (en-US)](https://developer.mozilla.org/en-US/docs/Web/API/Response/json) (definido en el [mixin](https://es.wikipedia.org/wiki/Mixin) de Body, el cual está implementado por los objetos [Request](https://developer.mozilla.org/es/docs/Web/API/Request) y [Response](https://developer.mozilla.org/es/docs/Web/API/Response)).

Peticiones HTTP con fetch

**Fetch** es el nombre de una nueva API para Javascript con la cuál podemos realizar peticiones **HTTP** asíncronas utilizando promesas y de forma que el código sea un poco más sencillo y menos verbose. La forma de realizar una petición es muy sencilla, básicamente se trata de llamar a **fetch** y pasarle por parámetro la URL de la petición a realizar:

// Realizamos la petición y guardamos la promesa

**const** request = **fetch("/robots.txt");**

// Si es resuelta, entonces ejecuta esta función...

request.**then**(**function**(response) { ... });

El **fetch()** devolverá una **PROMISE** que será aceptada cuando reciba una respuesta y sólo será rechazada si hay un fallo de red o si por alguna razón no se pudo completar la petición. El modo más habitual de manejar las promesas es utilizando **.then()**. Esto se suele reescribir de la siguiente forma, que queda mucho más simple:

**fetch("/robots.txt")**

**.then**(**function**(response) {

/\*\* Código que procesa la respuesta \*\*/

});

Al método **.then()** se le pasa una función **callback** donde su parámetro **response** es el objeto de respuesta de la petición que hemos realizado. En su interior realizaremos la lógica que queramos hacer con la respuesta a nuestra petición. A la función **fetch(url, options)** se le pasa por parámetro la **url** de la petición y, de forma opcional, un objeto **options** con opciones de la petición HTTP.

Vamos a examinar un código donde veamos un poco mejor como hacer la petición con **fetch**:

// Opciones de la petición (valores por defecto)

**const** options = {

**method**: **"GET"**

};

// Petición HTTP

**fetch**(**"/robots.txt"**, options)

**.then**(response => response.**text**())

**.then**(data => {

/\*\* Procesar los datos \*\*/

});

En el ejemplo que estamos abordando estamos utilizando el API de Fetch para hacer una solicitud a una Web externa en este caso “rickandmortyapi.com” que es un API que nos permite consultar información de “personajes de esta serie”. Entonces, se ha generado 3 promesas que llaman a la misma API con diferentes personajes. Solicitamos información de 3 personajes. La promesa de “fetch” me devuelve una “respuesta (response)” cuando resuelve. Usando esta respuesta busco el body – el “JSON” y por ello uso el método “.json” pero “.json” es otra promesa. Por lo cual, uso otro “.then”, o sea tengo “.then” por cada “response” y por cada “response” tengo otro “.then”. Cuando la 1era promesa resuelve, es cuando yo tengo la “data”. Es decir, la información del personaje.

//partimos de una Web que hace 3 solicitudes a una misma App. Sera un listado, una agrupación de Solicitudes  
const *promise1* = fetch("https://rickandmortyapi.com/api/character/?name=cowboy%20rick")  
 .then((response) => response.json())  
 .then((data) => *console*.log(data.results[0]) );  
  
const *promise2* = fetch("https://rickandmortyapi.com/api/character/?name=cult%20leader%20morty")  
 .then((response) => response.json())  
 .then((data) => *console*.log(data.results[0]) );  
  
const *promise3* = fetch("https://rickandmortyapi.com/api/character/?name=daron%20jefferson")  
 .then((response) => response.json())  
 .then((data) => *console*.log(data.results[0]) );

Continuando. El 1er método que veremos es el método “promise.all”, del ejemplo anterior, quitaremos los “console.log” porque este “console.log” no retorna nada. De forma que lo que estaremos retornando será la data. Iniciamos por utilizar “Promise.all()”.

Todos los métodos que vamos a ver admiten un “Array de Promesas”, en este caso será utilizando “Promise.All()” que en este caso nos devolverá una promesa “mayor es” que se va a resolver cuando todas las promesas del Array se hayan resuelto, es decir, esperara que se resuelva la promesa1, la promesa 2, la promesa3, y “Promise.All()” de ese array, devolverá una promesa y por ello se usará un “.then” que se va a resolver cuando se resuelvan las anteriores, entonces, “.then(allData)” tendrá toda la “data” de todas las promesas que tenga incluida.

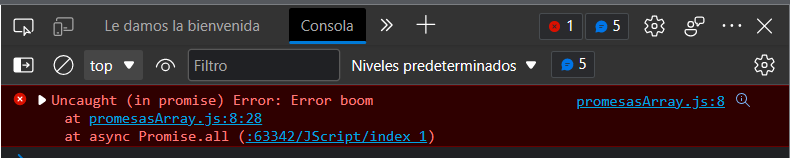
El “Promise.All()” se usa cuando las promesas son dependientes entre sí. O se resuelven todas, o se lanza un error. Si se lanza un error, da igual cuales se hayan resuelto o cuales no, en cuanto se lance un error, se va tomar por saco (todas) el error. Lo que me interesa del “Promise.All()” es que sean dependientes, me interesa que resuelvan todas.

“allData” es un Array con los resultados de cada una de las promesas. Si hay un error, el resultado de las demás se va al carajo. Por esto, este tipo de promesa “Promise.All()” es cuando se necesite un listado de promesas que resuelvan de forma correcta.

//partimos de una Web que hace 3 solicitudes a una misma App. Sera un listado, una agrupación de Solicitudes  
const *promise1* = fetch("https://rickandmortyapi.com/api/character/?name=cowboy%20rick")  
 .then((response) => response.json())  
 .then((data) => data.results[0]);  
  
const *promise2* = fetch("https://rickandmortyapi.com/api/character/?name=cult%20leader%20morty")  
 .then((response) => response.json())  
 .then((data) => data.results[0]);  
  
const *promise3* = fetch("https://rickandmortyapi.com/api/character/?name=daron%20jefferson")  
 .then((response) => response.json())  
 .then((data) => data.results[0]);  
  
*Promise*.all([*promise1*, *promise2*, *promise3*]).then((allData) =>  
 *console*.log(allData)  
);

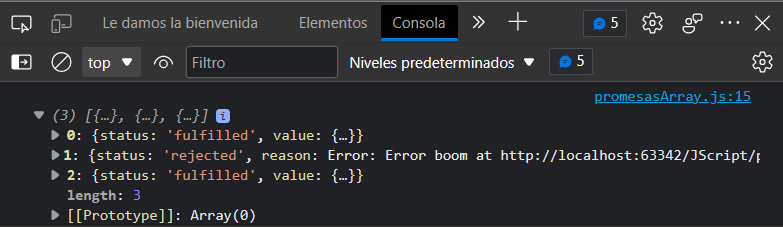
Teniendo “Promise.all()” podría generar un error si hago que falle la 2da promesa al lanzar a propósito un error, como en el siguiente código:

const *promise2* = fetch("https://rickandmortyapi.com/api/character/?name=cult%20leader%20morty")  
 .then((response) => response.json())  
 .then((data) => {throw new Error('Error boom')});

Resultado en consola cuando lanzo un error en la 2da promesa teniendo promise.all programada.

Da un error sin dudar porque no he puesto ningún Try… catch… pero estoy viendo los datos de las otras 2 promesas que si se han resuelto bien. Por lo cual, promise.all es cuando tenga un listado de promesas que resuelvan de forma correcta.

Pero que pasa, si me interesan que se resuelvan, pero me da igual que una falle. Y saber porque ha fallado, que no me dé un error, que se resuelvan todas, y si una falla a lo mejor es salvable, y para eso tenemos otro método que es:

“**Promise.allSettled**” en este caso lo que me retornará es el resultado de la resolución de las promesas actuales “allData”. Pero ese resultado es en un formato específico. No me devuelve un array de objetos directamente como los vistos en el ejemplo anterior, devuelve un “Array de Objetos especial”.

Tiene un valor; Status que indica si la promesa se ha resuelto (fulfilled) o no de forma correcta (rejected). En este caso la promesa 2 no se ha resuelto “rejected” pero no me lanza un “error”, captura el “error” y me notifica que esa promesa ha fallado. Sigue devolviendo el array con los datos.

Entonces devuelve un Array con el Objeto Status y el siguiente valor va a depender de si la promesa resultó bien o mal, si ha resuelto bien, nos devuelve “value” y dentro del “value” obtengo los datos de mi personaje (para este ej.).

Con promise.all lo que obtenía era siempre el value, si no un error.

Con promise.allSettled tengo en el caso de resuellto (fulfilled): Status y Value , en caso de rechazo (rejected): Status y reason, reason es el error por el que ha fallado (puede darte el error capturado y el Stack del error).

El siguiente método a ver es “promise.race” este método no espera a que se resuelvan todas las promesas. Lo que hace es devolvernos el resultado de la promesa que sea más rápido de resolver

Acepta o rechaza dependiendo de la primera promesa que se procese.

El método **Promise.race()** funciona como una «la primera que se procese»: la primera promesa del **Array** que sea procesada, independientemente de que se haya cumplido o rechazado, determinará la devolución de la promesa del **Promise.race()**. Si se cumple, devuelve una promesa cumplida, en caso negativo, devuelve una rechazada.

Ejemplo;

**const** p1 = **fetch**("/robots.txt");

**const** p2 = **fetch**("/index.css");

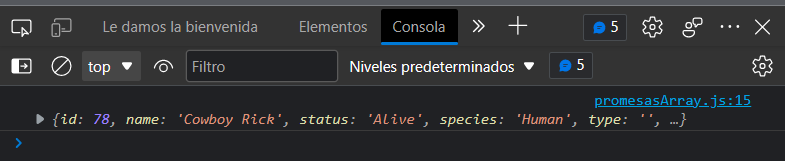
**const** p3 = **fetch**("/index.js");

Promise.**race**([p1, p2, p3])

.**then**(response => console.**log**(response.status, response.url));

De forma muy similar a la anterior, **Promise.race()** devolverá la promesa que se resuelva primero, ya sea cumpliéndose o rechazándose. No importa el orden, sea la 1, la 2 o la 3, por ejemplo. Entonces, la data que obtendré de **Promise.race()** será la 1era que se resuelva “firstToResolve”

*Promise*.race([*promise1*, *promise2*, *promise3*]).then((firstToResolve) =>  
 *console*.log(firstToResolve)  
);



Es la 1era promesa la que resuelve, es “cowboy Rick”.

Para el caso de “errores” tenemos este último método que es el “promise.any”. Actúa de la misma forma que el “race”, pero si la 1era que se resuelve da error, espera la siguiente, si la siguiente da error, espera la siguiente, así hasta que alguna resuelva de manera efectiva.

Funciona como «**la primera que se cumpla**»: Devuelve una promesa con el valor de la primera promesa individual del **Array** que se cumpla. Si todas las promesas se rechazan, entonces devuelve una promesa rechazada.

const p1 = fetch("/robots.txt");

const p2 = fetch("/index.css");

const p3 = fetch("/index.js");

Promise.**any**([p1, p2, p3])

.then(response => console.log(response.status, response.url));

Si ejecutamos el código, solo la 1era y la 3ra promesa podrían devolverse, ya que la 2da devuelve error.

*Promise*.any([*promise1*, *promise2*, *promise3*]).then((firstToResolve) =>  
 *console*.log(firstToResolve)  
);

Este es mucho más útil que el “race”, el 1ero pero que no falle.

DESTRUCTURING en JS con ARRAYS y OBJETOS - Javascript AVANZADO

<https://www.youtube.com/watch?v=30UWn10qbr0&t=17s>

Es una expresión que te permite extraer o “destructurar” datos desde estructuras de datos como arreglos, objetos, mapas y sets y crear nuevas variables con ese dato en particular. Te permite extraer propiedades de un objeto, items de un arreglo de una manera “sencilla” y de una sola vez.

Es una técnica que aplicamos sobre arrays y sobre objetos para asignar cada una de las Propiedades de este tipo de datos a una Variable. En vez de definir cada variable y luego asignarse a la posición de un array o a cada propiedad de un objeto lo haremos con el destructuring.

¿Cómo hacer el código siguiente con destructuring?

//Destructuring Assignment - Asignación de Desestructuración con Arrays  
*console*.clear();  
const *listaNumeros* = [12, 21, 38];  
  
const *posicion1* = *listaNumeros*[0];  
const *posicion2* = *listaNumeros*[1];  
const *posicion3* = *listaNumeros*[2];  
  
*console*.log(*posicion1*, *posicion2*, *posicion3*);

Para comenzar la destructuración, eliminaremos las variables; posicion2, y posicion3, y el siguiente paso es asignar los elementos de “listaNumeros” a un conjunto de variables (posicion1, posicion2, posicion3), para ello lo que hay que hacer es utilizar la misma sintaxis del Array en la parte izquierda de la asignación, es decir, al lado de “const” se define un Array “[ ]” con el conjunto de variables a los que quiero que se asigne cada elemento, como que estuviera definiendo un array con las 3 variables.

Con esta sintaxis defino lo siguiente:

Del Array de la derecha, me asignarás el 1er elemento “12”, a la primera variable “posicion1”, el 2do elemento “21” la asignará a “posicion2”, y el 3er elemento “38” asignarlo a “posicion3”. Estoy haciendo lo mismo que definiéndolo de forma independiente, pero en una sola línea y a través de la sintaxis de Array que seria una forma de aplicar el destructuring en Arrays.

**const** *listaNumeros* = [12, 21, 38];  
  
**const** [*posicion1*, *posicion2*, *posicion3*] = *listaNumeros*;

Pero qué pasa si en nuestro ejercicio solo nos interesa quedarnos con ciertas posiciones del Array. Podría saltarme una posición al momento de asignar los valores del Array. Por ejemplo, si quiero saltarme la “posicion1”, dejo la “coma” huérfana “sin nada a la izquierda” y le estoy diciendo a JS que esa 1era posición no me interesa. Solo me interesa la posicion2 y la posicion3. Y al momento de imprimir, solo imprimo las posiciones 2 y 3.

//Destructuring Assignment - Asignación de Desestructuración con Arrays  
*console*.clear();  
const *listaNumeros* = [12, 21, 38];  
const [, *posicion2*, *posicion3*] = *listaNumeros*;  
  
*console*.log(*posicion2*, *posicion3*);

También podría saltarme las posiciones intermedias. Podría decir que no me interesa la “posicion2”. Y en este caso me mantendría la posición 1era y 3era.

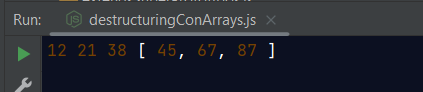
//Destructuring Assignment - Asignación de Desestructuración con Arrays  
*console*.clear();  
const *listaNumeros* = [12, 21, 38];  
const [*posicion1*, , *posicion3*] = *listaNumeros*;  
  
*console*.log(*posicion1, posicion3*);

Entre las comas de la posición 1 y la 3, mientras no ponga ningún carácter o ningún nombre de variable no habría ningún problema, y poner espacio o no da lo mismo.

Tenemos un **caso especial** que es agrupar en los elementos que no he seleccionado con mi “destructuring”. Por ejemplo, tengo un Array con más elementos que las posiciones que estamos asignando (1, 2 y 3).

const *listaNumeros* = [12, 21, 38, 45, 67, 87];

Cómo puedo decirle a JS que tome los elementos de la cola que yo no haya seleccionado. Es decir, yo he seleccionado las 3 primeras posiciones, pero me interesa tener el resto de elementos en el Array. Esto se puede hacer con el “Spread Operator”. Esto lo defino en el último elemento, ahí agregamos una “coma” y el “Spread Operator (…)” y ponemos “…restoElementos”. Con esto le estás diciendo a JS, “oye, el resto de elementos que sobren, que no sean esos 3 primeros, me los agrupas en un Array, y si imprimimos “restoElementos” se observaran ese resto de elementos agrupados en un Array.

//Destructuring Assignment - Asignación de Desestructuración con Arrays  
*console*.clear();  
const *listaNumeros* = [12, 21, 38, 45, 67, 87];  
  
const [*posicion1*, *posicion2*, *posicion3*, ...*restoElementos*] = *listaNumeros*;  
  
*console*.log(*posicion1*, *posicion2*, *posicion3*, *restoElementos*);

Se puede observa en la salida de consola la impresión de las 3 primeras posiciones de forma normal, pero con “**restoElementos**” se ve un grupo de valores agrupados en un Array [**45, 67, 87**] que contiene el resto de elementos que tenía mi Array original por la cola.

Si quito “posicion3”, de la asignación, y la impresión, lo que sucederá que este “38” va a estar en el Array de “restoElementos” y solo posicion1 y posicion2 (12, 21)

//Destructuring Assignment - Asignación de Desestructuración con Arrays  
*console*.clear();  
const *listaNumeros* = [12, 21, 38, 45, 67, 87];  
  
const [*posicion1*, *posicion2*, ...*restoElementos*] = *listaNumeros*;  
  
*console*.log(*posicion1*, *posicion2*, *restoElementos*);

Salida en consola:

12 21 [ 38, 45, 67, 87 ]

En resumen, esta es una técnica para formar un subArray con el resto de elementos.

**Destructuring con Objetos.**

El código a continuación es un código convencional. Es un objeto con 3 propiedades; nombre, apellido y edad. Y 3 definiciones de variables con sus respectivas asignaciones.

//Destructuring Assignment - Asignación de Desestructuración con Objetos  
*console*.clear();  
  
const *persona* = {  
 nombre: "Juan",  
 apellido: "Perez",  
 edad: 30,  
};  
  
**const** ***nombre***= *persona*.nombre;  
**const** ***apellido***= *persona*.apellido;  
**const** ***edad***= *persona*.edad;  
  
*console*.log(*nombre*, *apellido*, *edad*);

Si le aplicamos al código anterior el “destructuring” haríamos algo similar a lo hecho con “Arrays”, pero en vez de utilizar los corchetes “[ ]”, utilizaremos las llaves “{ }” que se usan para definir un objeto. En el caso de los objetos tenemos que acceder por el nombre que tiene la Propiedad en el Objeto. En el caso del ejemplo; {nombre, apellido, edad}.

//Destructuring Assignment - Asignación de Desestructuración con Objetos  
*console*.clear();  
  
const *persona* = {  
 nombre: "Juan",  
 apellido: "Perez",  
 edad: 30,  
};  
  
const {*nombre*, *apellido*, *edad*} = *persona*;  
  
*console*.log(*nombre*, *apellido*, *edad*);

Si pongo el valor de una Propiedad que no existe, entonces al imprimir, el valor será “undefined” porque esa propiedad no existe en el objeto.

Pero pudiera interesar tener la variable con otro nombre que no fuera el de la propiedad puedo renombrar esa propiedad con el propio “destructuring”.

Por ejemplo, supongamos que “nombre” no se llame así, entonces lo que se hace en el “destructuring” una vez puesto nombre (esto no varía) se ponen “:” y el nuevo nombre que le darémos. “nombre: nombreJuan”.

Si en el “console.log” no cambio el nombre de la variable por la nueva variable, me dirá “ReferenceError: nombre is not defined”. Pero si ponemos el valor de la variable renombrada no habrá problema.

//Destructuring Assignment - Asignación de Desestructuración con Objetos  
*console*.clear();  
  
const *persona* = {  
 nombre: "Juan",  
 apellido: "Perez",  
 edad: 30,  
};  
  
const {nombre: *nombreJuan*, *apellido*, *edad*} = *persona*;  
  
*console*.log(*nombreJuan*, *apellido*, *edad*);

Salida en consola: Juan Perez 30

Lo de los “:” lo puedo hacer en todo. Y ya tendría renombrada las 3 variables.

//Destructuring Assignment - Asignación de Desestructuración con Objetos  
*console*.clear();  
  
const *persona* = {  
 nombre: "Juan",  
 apellido: "Perez",  
 edad: 30,  
};  
  
const {nombre: *nombreJuan*, apellido: *apellidoJuan*, edad: *edadJuan*} = *persona*;  
  
*console*.log(*nombreJuan*, *apellidoJuan*, *edadJuan*);

En los Arrays tenemos una forma de saltarnos elementos, en los Objetos no tiene sentido saltarse elementos porque no van por orden. Es decir, si dejo una “coma” sola saltará un error. Porque no me dejará saltarme propiedades.

const {, *apellido*, *edad*} = *persona*;

Pero si puedo elegir las propiedades que necesite.

//Destructuring Assignment - Asignación de Desestructuración con Objetos  
*console*.clear();  
  
const *persona* = {  
 nombre: "Juan",  
 apellido: "Perez",  
 edad: 30,  
};  
  
const {apellido, edad} = *persona*;  
  
*console*.log(*apellido*, *edad*);

Por otro lado, el formato Spread que permite tomar el resto del Objeto y meterlo en un nodo Objeto.

Por ejemplo, podría tener el nombre solo, y tener “apellido y edad” en un solo objeto, podría hacer uso del Objeto Spread. Código a continuación:

//Destructuring Assignment - Asignación de Desestructuración con Objetos  
*console*.clear();  
  
const *persona* = {  
 nombre: "Juan",  
 apellido: "Perez",  
 edad: 30,  
};  
  
const {*nombre*, ...*restoPersona*} = *persona*;  
  
*console*.log(*nombre*, *restoPersona*);

La salida en consola sería;

Juan { apellido: 'Perez', edad: 30 }

Se obtiene el nombre y el restoPersona serán las propiedades que no hemos asignado.

Si en lugar de nombre, hubiese utilizado edad, restoPersona sería el nombre y el apellido

const {*edad*, ...*restoPersona*} = *persona*;  
*console*.log(*edad*, *restoPersona*);

Salida en consola: 30 { nombre: 'Juan', apellido: 'Perez' }

Aquí en los objetos, el orden no aplica como en el caso de los Arrays. Porque son propiedades nombradas. No hay ningún problema en extraer solo las propiedades de mi interés. Pero luego tener un objeto con el formato Spread con el resto de propiedades que no he extraído en mi “destructuring”.

Para finalizar el Spread sobre Objetos, veremos un caso particular.

Supongamos que el objeto Persona tuviese en su interior a parte de las propiedades; nombre, apellido y edad, tuviera además un “sub objeto ***direccion***” que a su vez tiene propiedades; calle, código postal. Y a continuación hago el “destructuring” de nombre y dirección. Y entonces, en el console.log lo que tengo es un nombre que es un string, y dirección que es un Objeto.

const *persona* = {  
 nombre: "Juan",  
 apellido: "Perez",  
 edad: 30,  
 direccion: {  
 calle: 'Avenida Plata, 123',  
 codigoPostal: 48080  
 }  
};  
  
const {*nombre*, *direccion*} = *persona*;  
  
*console*.log(*nombre*, *direccion*);

Salida en consola:

Juan { calle: 'Avenida Plata, 123', codigoPostal: 48080 }

Pero que pasa si ahora quiero hacer destructuring sobre este objeto. Lo haría del siguiente modo, haciendo 1ero un destructuring de “persona”, y luego un destructuring de “direccion”, y así podría acceder a los elementos “calle y código postal”.

const *persona* = {  
 nombre: "Juan",  
 apellido: "Perez",  
 edad: 30,  
 direccion: {  
 calle: 'Avenida Plata, 123',  
 codigoPostal: 48080  
 }  
};  
  
const {*nombre*, *direccion*} = *persona*;  
const {*calle*, *codigoPostal*} = *direccion*;  
*console*.log(*nombre*, *calle*, *codigoPostal*);

Saluda en consola:

Juan Avenida Plata, 123 48080

Esto tiene un límite porque se vuelve ilegible. ¿Cómo lo haría en la misma línea? Con el mismo formato del “rename” que lo que hacíamos es poner la “variable: renombre” en este caso lo que haremos es poner la “variable” que es un objeto “direccion” y otra vez las llaves “{ }” y destructuramos poniendo sus propiedades “calle y código postal”, y es como hacer un “destructuring” dentro del propio “destructuring”. Esto es funcional a un nivel, se puede aplicar a los niveles que quiera, pero con varios niveles se vuelve ilegible.

//Destructuring Assignment - Asignación de Desestructuración con Objetos  
*console*.clear();  
  
const *persona* = {  
 nombre: "Juan",  
 apellido: "Perez",  
 edad: 30,  
 direccion: {  
 calle: 'Avenida Plata, 123',  
 codigoPostal: 48080  
 }  
};  
  
const {  
 *nombre*,  
 direccion: {*calle*, *codigoPostal*}  
} = *persona*;  
  
*console*.log(*nombre*, *calle*, *codigoPostal*);

Finalizando con algunos TRUCOS de uso habitual para cuando deseamos hacer ciertas cosas. En el caso que deseamos hacer un destructuring con algunas tareas ya que esto lo va a simplificar bastante.

Imaginar el caso de un Swap de Variables, en el que tengo, por ejemplo, 2 variables; nombre y apellidos, y lo que quiero es asignar la variable de apellido a nombre y la de nombre a apellido. Para hacer esto necesitaríamos una 3era Variable (auxiliar llamada intermedia) que mantuviera el valor

*console*.clear();  
  
let *nombre* = "Juan";  
let *apellido* = "Perez";  
  
let *intermedia*;  
  
*intermedia* = *nombre*;  
*nombre* = *apellido*;  
*apellido* = *intermedia*;  
  
*console*.log(*nombre*, *apellido*);

¿Cómo aplicamos el **destructuring** al código anterior? Con el uso de Arrays a ambos lados, y como puedo jugar con las posiciones, puedo llegar al lado izquierdo e invertir el orden de las propiedades o elementos del Array; y de este modo lo que se hace es aplicar “nombre” a la 1era posición del Array de lado izquierdo que es “apellido” y “apellido” a la 2da posición que es “nombre” en el array de la izquierda. Y con una sola línea tengo un Swap (intercambio) de variables.

*console*.clear();  
  
let *nombre* = "Juan";  
let *apellido* = "Perez";  
  
[*apellido*, *nombre*] = [*nombre*, *apellido*];  
*console*.log(*nombre*, *apellido*);

Aplicar el “destructuring” en nuestras funciones. Tenemos el objeto “persona” y una función “getFullName” que se encarga de imprimir el nombre completo. La función lo que hace es tomar y concatenar el “nombre y el apellido” de la persona y los retorna. Y el console.log que llama a la función.

*console*.clear();  
  
**const** ***persona***= {  
 nombre: "Juan",  
 apellido: "Perez",  
 edad: 30,  
 direccion: {  
 calle: 'Av Hope, 123',  
 ciudad: "Madrid",  
 pais: "España",  
 },  
};  
  
**function** **getFullName**(**persona**) {  
 **return** `${persona.**nombre**} ${persona.**apellido**}`;  
}  
  
*console*.log(**getFullName**(***persona***));

Puedo aplicar el “destructuring” en los parámetros de la función.

En el ejemplo, lo haríamos eliminando “persona” y cómo sé que recibo un parámetro más, agrego las llaves “{ }” y digo, en el 1er parámetro empezaremos a “destructurar” y pondremos las propiedades “nombre y apellido” y dentro de la función ya tengo acceso a “nombre y apellido” y no necesito agregar el prefijo “persona” a las propiedades.

Si tuviera más parámetros agrego una coma “,” y agrego el siguiente objeto y aplico destructuring con las llaves “{ }” o si tuviera parámetros simples, igual los agrego separándolos por coma “,”, pero yo puedo “destructurar” todos los objetos que recibo como parámetros poniendo el par de llaves y acceder a sus propiedades.

*console*.clear();  
const *persona* = {  
 nombre: "Juan",  
 apellido: "Perez",  
 edad: 30,  
 direccion: {  
 calle: 'Av Hope, 123',  
 ciudad: "Madrid",  
 pais: "España",  
 },  
};  
function getFullName({nombre, apellido}) {  
 return `${nombre} ${apellido}`;  
}  
*console*.log(getFullName(*persona*));

Hasta acá lo hemos visto con los parámetros, ahora lo veremos con el “return”.

Tenemos una función getSalary() que retorna un “Array”. Puede ser un Objeto también.

function getSalary() {  
 return [970, 1020, 980, 1123];  
}  
  
const *salarios* = getSalary();  
  
*console*.log(*salarios*);

Puedo hacer el destructuring directamente en la llamada de la función, a la izquierda del igual.

function getSalary() {  
 return [970, 1020, 980, 1123];  
}  
  
const [*salario1*, *salario2*, *salario3*, *salario4*] = getSalary();  
  
*console*.log(*salario1*, *salario2*, *salario3*, *salario4*);

Salida en consola del 1er código sin hacer destructurign: [970, 1020, 980, 1123]

Salida en consola del 2do código haciendo destructuring: 970 1020 980 1123

En JS el tipado es dinámico. JS no comprueba los tipos de datos al insertarlos en un ARRAY. Entonces, podemos insertar cualquier cosa dentro de un Array. Podemos tener un Array de números y de repente se nos cuela un String, tenemos que tener cuidado con eso. Hay que procurar que los Array sean del mismo tipo. Cuando hacemos operaciones con los Array nos interesa que los Array almacenen información del mismo tipo. Para poder realizar las mismas operaciones.

ARRAYS ✅ MAP, FOREACH, REDUCE y mucho MÁS! 😎 Curso de JAVASCRIPT desde CERO #6

<https://www.youtube.com/watch?v=J9vUZu6edBA>

**Operaciones con Array.**

<https://www.freecodecamp.org/espanol/news/tutorial-de-javascript-arrayforeach-como-iterar-a-traves-de-elementos-en-un-arreglo/>

En JS con frecuencia te verás en necesidad de iterar a través de una colección de arreglos y ejecutar un método callback por cada una de las iteraciones. Y para realizarlo hay un método muy útil que los desarrolladores de JS típicamente utilizan: el método **forEach()**.

El método **forEach()** llama a una función callback específica *una vez por cada elemento sobre el que itera dentro de un arreglo*. Al igual que otros iteradores de arreglos tales como **map** y **filter**, la función callback puede recibir tres parámetros.

* **El elemento actual**: Este es el elemento del arreglo por el cual se está haciendo la iteración.
* **Su índice**: Este es el índice de la posición que tiene el elemento dentro del arreglo.
* **El arreglo objetivo**: Este es el arreglo por el cual se está haciendo la iteración.

El método **forEach** no regresa un nuevo arreglo al igual que lo hacen otros iteradores tal como **filter**, **map** y **sort**. En cambio, el método devuelve el valor undefined por sí solo. Por lo tanto, no se puede encadenar como los otros métodos.

Otra cosa acerca de **forEach** es que no puedes terminar el bucle (con la instrucción **break**) o hacerlo saltarse una iteración (con la sentencia **continue**). En otras palabras, *no lo puedes controlar*.

La única manera de finalizar un bucle **forEach** es incluyendo una excepción dentro de la función callback. No te preocupes, pronto veremos todo esto en la práctica.

**Como usar el método forEach() en JavaScript**

Imagina que un grupo de estudiantes hacen una fila para el pase de lista de rutina. El coordinador de la clase se mueve a través de la fila y dice el nombre de cada uno de los estudiantes mientras marca si están presentes o ausentes.

Es importante hacer mención que el coordinador no cambia el orden de los estudiantes en la fila. También los mantiene en la misma fila después de terminar el pase de lista. Todo lo que realiza es una acción (su inspección) en cada uno de ellos.

En los siguientes ejemplos, teniendo en mente este escenario, veremos cómo puedes utilizar el método **forEach** para resolver problemas de la vida real con JavaScript.

**Como remover el primer número impar en un arreglo con** **forEach()**

Tenemos un arreglo que contiene un número impar en la primera posición y varios números pares después. Pero solo queremos números pares en este arreglo. Así que vamos a remover el número impar del arreglo usando el bucle **forEach()**:

El método **shift** elimina el elemento en el índice cero y desplaza los valores consecutivos hacia abajo, devolviendo el valor eliminado. Si la propiedad length es 0, devuelve undefined. shift es genérico; este método puede utilizarse con call o apply a objetos simliares a arrays.

let *numeros* = [3, 6, 8, 10, 12]  
let *impar* = 3;  
  
*numeros*.forEach(function(numero) {  
 if (numero === *impar*) {  
 *numeros*.shift() // 3 será borrado del arreglo  
 }  
})  
  
*console*.log(*numeros*);  
  
[6, 8, 10, 12] // Todos son pares!

**Como acceder a la propiedad del índice con** **forEach()**

En este ejemplo, vamos a ejecutar la función **pasarLista** por cada uno de los estudiantes por los que se recorrió el bucle dentro del arreglo. La función **pasarLista** solamente registra en la consola una cadena perteneciente a cada uno de los estudiantes.

**nombres** = ["anna", "beth", "chris", "daniel", "ethan"]  
  
function pasarLista(nombre, indice) {  
 *console*.log(`Está el numero de estudiante ${indice + 1} -${nombre} - presente? Sí!`);  
 }  
  
**nombres**.**forEach**((nombre, indice) **=>** pasarLista(nombre, indice));

Salida en consola:

Está el número de estudiante 1 -anna - presente? Sí!

Está el número de estudiante 2 -beth - presente? Sí!

Está el número de estudiante 3 -chris - presente? Sí!

Está el número de estudiante 4 -daniel - presente? Sí!

Está el número de estudiante 5 -ethan - presente? Sí!

En este ejemplo, el **nombre** era la única información que teníamos sobre cada estudiante. Sin embargo, también queremos saber cuáles **pronombres** usa cada estudiante. En otras palabras, queremos definir una propiedad pronombre para cada estudiante.

Entonces vamos a definir a cada estudiante como un objeto con dos propiedades, **nombre** y **pronombre**:

nombres = [  
 {nombre:"anna",pronombre: "ella"},  
 {nombre: "beth",pronombre: "ella"},  
 {nombre:"chris",pronombre: "él"},  
 {nombre: "daniel",pronombre: "él"},  
 {nombre: "ethan",pronombre: "él"}  
]  
  
function pasarLista(estudiante, indice) {  
 *console*.log(`El número de estudiante ${indice + 1} ${estudiante.nombre}. Está ${estudiante.pronombre} presente? Sí!`);  
 }  
  
nombres.forEach((nombre, indice) => pasarLista(nombre, indice));

Salida en consola:

El número de estudiante 1 anna. Está ella presente? Sí!

El número de estudiante 2 beth. Está ella presente? Sí!

El número de estudiante 3 chris. Está él presente? Sí!

El número de estudiante 4 daniel. Está él presente? Sí!

El número de estudiante 5 ethan. Está él presente? Sí!

Estamos registrando en la consola el pase de lista de cada estudiante, luego realizamos una verificación para ver qué pronombre usa cada estudiante, y finalmente pasamos dinámicamente el pronombre correcto como parte de la cadena.

Como copiar un arreglo en un nuevo arreglo con forEach() en JavaScript

Después de tres años de estudio, es tiempo de que cada estudiante se gradúe. En nuestro **JS**, vamos a definir dos arreglos: **aunEstudiante** y **yaGraduado**. Como probablemente adivinaste, **aunEstudiante** guarda el dato de los estudiantes antes de su graduación.

Posteriormente, el bucle **forEach** toma el dato de cada uno de los estudiantes y llama a la función **estudianteGraduado** en él.

En esta función construimos un objeto con dos propiedades: el **nombre de estudiante** y la **posición** en la que se graduó. Luego pasamos el nuevo objeto al arreglo **yaGraduado**. A este punto, el estudiante ya se ha graduado.

Este ejemplo también demuestra cómo puede usar el método **forEach()** para copiar un arreglo dentro de un nuevo arreglo.

let *aunEstudiante* = ["anna", "beth", "chris", "daniel", "ethan"]  
let *yaGraduado* = []  
  
function estudianteGraduado(estudiante, indice) {  
 let objeto = { nombre: estudiante, posicion: indice + 1}  
 *yaGraduado*[indice] = objeto  
}  
  
*aunEstudiante*.forEach((nombre, indice) => estudianteGraduado(nombre, indice));  
  
*console*.log(*yaGraduado*);  
  
/\*  
[  
 { nombre: "anna", posición: 1},  
 { nombre: "beth", posición: 2},  
 { nombre: "chris", posición: 3},  
 { nombre: "daniel", posición: 4},  
 { nombre: "ethan", posicion: 5}]  
]  
\*/

Como revisar el elemento siguiente en un arreglo con el parámetro **arreglo**

En algún momento, el profesor deberá verificar si la lista tiene a continuación un elemento en particular en la lista. En tal caso, el profesor deberá tener una visión amplia de toda la lista. De esa manera, puede saber si hay un próximo estudiante al cual llamar.

En nuestro **JS**, podemos replicar esto porque la función **callback** también puede acceder al parámetro (el tercero) del arreglo. Este parámetro representa el arreglo objetivo, que es nombre.

Verificamos si hay un siguiente elemento (estudiante) en el arreglo. Si lo hay, pasamos la cadena **positivo** a la variable **siguienteElemento**. Si no hay ninguno, pasamos la cadena **negativo** a la variable. Luego para cada iteración, revisamos si ese estudiante es realmente el último.

nombres = ["anna", "beth", "chris", "daniel", "ethan"]  
  
function pasarLista(nombre, indice, arreglo) {  
 let siguienteElemento = indice + 1 < arreglo.length ? "positivo" : "negativo"  
 *console*.log(`Está el número de estudiante ${indice + 1} - ${nombre} presente? Sí!. Hay un próximo estudiante? ${siguienteElemento}!`);  
}  
  
nombres.forEach((nombre, indice, arreglo) => pasarLista(nombre, indice, arreglo))  
  
/\*  
"Está el número de estudiante 1 - anna presente? Sí!. Hay un próximo estudiante? positivo!"  
"Está el número de estudiante 2 - beth presente? Sí!. Hay un próximo estudiante? positivo!"  
"Está el número de estudiante 3 - chris presente? Sí!. Hay un próximo estudiante? positivo!"  
"Está el número de estudiante 4 - daniel presente? Sí!. Hay un próximo estudiante? positivo!"  
"Está el número de estudiante 5 - ethan presente? Sí!. Hay un próximo estudiante? negativo!"  
\*/

Si te deseas salir de un bucle en alguna condición, entonces te recomiendo utilizar cualquiera de los siguientes métodos.

* bucle for
* bucle for…of o for…in
* Array.some()
* Array.every()
* Array.find()

Así es como puedes salirte de un bucle con **Array.every()**:

let *numeros* = [2, 4, 5, 8, 12]  
let *impar* = 5;  
  
*numeros*.every(numero => {  
 if (numero == *impar*) {  
 return false;  
 }  
  
 *console*.log(numero);  
  
 return true;  
});  
  
// 2 4

**Array.map() de JavaScript: Como iterar a través de elementos en un arreglo con map()**

Cuando salió ES6 (EmcaScript 2015), introdujo un nuevo conjunto de métodos para iterar sobre un arreglo. Uno de los más útiles es el método **map()**.

**Array.prototype.map()** es un método incorporado en los arreglos, *para iterar a través de los elementos dentro de una colección de arreglos* en **JS**. Piensa en él, como un bucle para avanzar de un elemento a otro en una lista, manteniendo el orden y la posición de cada elemento.

Este método toma una función **callback**, que se llama *por cada nuevo elemento sobre el que se itera*.

La función callback recibe tres parámetros:

* El valor actual.
* Su índice.
* El arreglo de destino

**Cómo funciona el método map() en JavaScript**

Imagínate esto: hay una cola de personas en la puerta de un hospital esperando a ser vacunadas. Esto significa que aún no han sido vacunados.

Uno por uno, un médico administra la vacuna a todos ellos. El médico lo hace repitiendo la fila. En un extremo, hay un grupo de personas que aún no han sido vacunadas. El médico toma a todos y cada uno de ellos, les administra la vacuna y los devuelve a una nueva línea de personas vacunadas.

En un extremo, hay un **arreglo (A)** sobre el que se quiere operar**. map()** toma todos los elementos de ese **arreglo (A)**, realiza una acción coherente sobre cada uno de esos elementos y los devuelve en un nuevo **arreglo (B)**.

**Cómo utilizar el método map(): Ejemplo de vengadores**

Para ilustrar cómo funciona **map()** en JS, consideremos una lista de nombres de algunos de nuestros Vengadores favoritos. El problema es que los nombres de esta lista no están completos: les falta un **sufijo** importante.

Con **map()**, podemos tomar todos los nombres de la matriz y añadir el **sufijo** "**man**" a todos y cada uno de ellos:

let *prefijos* = ["super", "spider", "ant", "iron"]  
let *sufijo* = "man";  
  
let *nombresCompletos* = *prefijos*.map(prefijo => prefijo + *sufijo*);  
  
*console*.log(*nombresCompletos*);  
  
// ["superman", "spiderman", "antman", "ironman"]

**¿Qué pasa con las mujeres(women)?**

Lo siento, mi error. Me di cuenta de mi error y decidí incluir un personaje femenino en la primera posición del arreglo. Cada elemento dentro de un arreglo se identifica por un valor único, su índice (o posición en el arreglo). El primer elemento tendrá un índice de 0, el segundo un índice de 1, y así sucesivamente.

Dado que ahora hay una superheroína en la lista, tendremos que asegurarnos de añadir el sufijo correcto a la superheroína correspondiente.

Dado que **map()** también toma el índice del elemento sobre el que estamos iterando, podemos hacer esto *comprobando el índice de nuestro héroe* y asegurándonos de que utilizamos el sufijo "woman" *para el primer elemento* de nuestra matriz:

let *prefijos* = ["wonder", "spider", "ant", "iron"]  
let *hombre* = "man";  
let *mujer* = "woman";  
  
let *nombresCompletos* = *prefijos*.map(function(prefijo, indice) {  
 return (indice == 0) ? prefijo + *mujer* : prefijo + *hombre*;  
});  
  
*console*.log(*nombresCompletos*);  
  
// ["wonderwoman", "spiderman", "antman", "ironman"]

**Cómo utilizar el argumento del índice**

Además del valor sobre el que se itera, **map** toma también su *posición de índice*. Esto es muy útil si quieres realizar diferentes tipos de operaciones dependiendo de la posición del índice del elemento.

En el ejemplo anterior, añadimos un sufijo diferente comprobando el índice.

Para conocer la posición del índice de cada uno de nuestros elementos dentro del arreglo, podemos hacer lo siguiente:

//posición de c/u de los elem dentro del array  
let *fullNames* = ["wonderwoman", "spiderman", "antman", "ironman"]  
  
*fullNames*.map(function(nombre, indice) {  
 *console*.log(`${nombre} está en la posición ${indice}`);  
});

wonderwoman está en la posición 0

spiderman está en la posición 1

antman está en la posición 2

ironman está en la posición 3

**Multiplicar todos los elementos del arreglo por 2**

Queremos multiplicar cada número del arreglo de destino por dos y luego devolver sus productos en un nuevo arreglo:

let *numeros* = [10, 20, 30, 40]  
let *multiplicador* = 2;  
  
let *productos* = *numeros*.map(numero => numero \* *multiplicador*);  
  
*console*.log(*productos*);  
  
// [20, 40, 60, 80]

**Redondear al número entero más cercano**

¿Qué pasa si tenemos un arreglo de decimales, pero queremos que cada uno de esos números decimales se redondee al entero más cercano?

let *numeros* = [3.7, 4.9, 6.2]

let *redondeados* = *numeros*.map(function(numero) {  
 return *Math*.round(numero);  
})  
  
*console*.log(*redondeados*);  
  
// [4, 5, 6]

const *totalFactura* = *facturas*.map(i => (i.precio \* i.cantidad \* *IVA*).toFixed(2));

**Cambiar cadenas de texto por números**

Tenemos una lista de números que son de tipo **string** (cadena). Sin embargo, queremos convertir cada uno de ellos al tipo Number (número):

let *textos* = ["10","20","30"]  
  
let *numeros* = *textos*.map(function(texto) {  
 return Number(texto);  
})  
  
*console*.log(*numeros*);  
  
// [10, 20, 30]

**Conseguir nombres reales de los Vengadores**

En este ejemplo, estamos trabajando con objetos. Tenemos cinco vengadores en el arreglo y cada uno tiene tanto un nombre real como un nombre de héroe. Sin embargo, solo queremos recuperar sus nombres reales en el nuevo arreglo.

let *vengadores* = [  
 {nombre: "steve rogers", nombreHeroe: "captain america"},  
 {nombre: "tony stark", nombreHeroe: "iron man"},  
 {nombre: "bruce banner", nombreHeroe: "the hulk"},  
 {nombre: "peter parker", nombreHeroe: "spiderman"},  
 {nombre: "tchalla", nombreHeroe: "black panther"}  
]  
  
let *nombresReales* = *vengadores*.map(vengador => vengador.**nombre**);  
  
*console*.log(*nombresReales*);  
  
// ["steve rogers", "tony stark", "bruce banner", "peter parker", "tchalla"]

**Conseguir los nombres de los héroes de los Vengadores**

Para obtener solo sus nombres de héroe, hacemos casi lo mismo, solo que en este caso accedemos a la propiedad **nombreHeroe**:

let *vengadores* = [  
 {nombre: "steve rogers", nombreHeroe: "captain america"},  
 {nombre: "tony stark", nombreHeroe: "iron man"},  
 {nombre: "bruce banner", nombreHeroe: "the hulk"},  
 {nombre: "peter parker", nombreHeroe: "spiderman"},  
 {nombre: "tchalla", nombreHeroe: "black panther"}  
]  
  
let *nombresReales* = *vengadores*.map(vengador => vengador.**nombreHeroe**);  
  
*console*.log(*nombresReales*);

// ["captain america", " iron man", " the hulk", "spiderman", "black panther"]

**Cómo separar una función para añadir a map**

En lugar de definir una función directamente dentro de **map()**, podemos definir la función fuera y luego llamarla dentro de nuestra función **map()**:

Quitar complejidad al **map** mediante separación de funciones

let *vengadores* = [  
 {nombre: "steve rogers", nombreHeroe: "captain america"},  
 {nombre: "tony stark", nombreHeroe: "iron man"},  
 {nombre: "bruce banner", nombreHeroe: "the hulk"},  
 {nombre: "peter parker", nombreHeroe: "spiderman"},  
 {nombre: "tchalla", nombreHeroe: "black panther"}  
];  
  
let obtenerNombre = vengador => vengador.nombre;  
let *nombresReales* = *vengadores*.map(obtenerNombre);  
  
*console*.log(*nombresReales*);  
  
// ["steve rogers", "tony stark", "bruce banner", "peter parker", "tchalla"]

**Cómo funciona el argumento del arreglo**

En cada iteración, el método **map()** toma el valor sobre el que se itera y también su posición de índice. Hay otro argumento que añadir a esos dos, el del **arreglo**.

El argumento **arr** representa el arreglo de destino sobre el que se hace un bucle, junto con todo su contenido. Con este argumento, se puede buscar en el arreglo completo para encontrar algo.

En este ejemplo, accederemos al parámetro **arr** para buscar y comprobar si el elemento actual es el último de la lista. Si no lo es, accedemos al siguiente elemento y lo restamos del elemento actual. Si es el último, simplemente lo devolvemos.

const *arregloViejo* = [33, 20, 10, 5];  
const *nuevoArreglo* = *arregloViejo*.map((valorActual, indice, arr) => {  
 let siguienteItem = indice + 1 < arr.length ? arr[indice + 1] : 0  
 return valorActual - siguienteItem;  
});  
  
*console*.log(*nuevoArreglo*);  
  
// [13, 10, 5, 5]

**Conclusiones**

El método **map()** fue introducido en **ES6**. Con este método, podemos acceder y realizar una acción consistente en cada elemento dentro de una colección de arreglos.

Toma una función **callback** que llama por cada nuevo elemento sobre el que itera.

Método map()

const *facturas* = [  
 {  
 precio: 23,  
 cantidad: 1,  
 },  
 {  
 precio: 19,  
 cantidad: 2,  
 },  
 {  
 precio: 14,  
 cantidad: 4,  
 },  
 {  
 precio: 22,  
 cantidad: 1,  
 },  
]

const *IVA* = 1.21;

const *totalFactura* = *facturas*.map(i => (i.precio \* i.cantidad \* *IVA*).toFixed(2));

*totalFactura*.forEach((i, index) => *console*.log("Elemento", index, i));

*console*.log(*totalFactura*);

Salida en consola:

Elemento 0 27.83

Elemento 1 45.98

Elemento 2 67.76

Elemento 3 26.62

Map() nos devuelve el valor calculado para cada elemento del array “facturas”, y lo devuelve en la misma posición del Array “facturas” al nuevo Array “totalFactura”.

Con el uso de forEach la salida será la siguiente:

Elemento 0 27.83

Elemento 1 45.98

Elemento 2 67.75999999999999

Elemento 3 26.619999999999997

**Cómo puedo hacer para sumar ahora todos los elementos del Array()**

Por cada uno de ellos ir sumándolos, y creamos un acumulador que inicializamos a cero “total = 0” y en la llamada al método generamos la acumulación.

const *facturas* = [  
 {  
 precio: 23,  
 cantidad: 1,  
 },  
 {  
 precio: 19,  
 cantidad: 2,  
 },  
 {  
 precio: 14,  
 cantidad: 4,  
 },  
 {  
 precio: 22,  
 cantidad: 1,  
 },  
]  
const *IVA* = 1.21;  
  
const *totalFactura* = *facturas*.map(i => i.precio \* i.cantidad \* *IVA*);  
  
let *total* = 0;  
*totalFactura*.forEach((i) => *total* += i)  
*console*.log(*total*);

Salida en consola: **168.19**

Pero tenemos métodos que se pueden ocupar para realizar esta operación.

Metodo Array. Reduce()

El método **reduce()** reduce un arreglo de valores a un solo valor. Para obtener el valor de salida, ejecuta una función reductora en cada elemento del arreglo.

Sintaxis

arr.**reduce**(callback[, valorInicial])

El argumento de **callback** es una función que se llamará una vez por cada elemento del arreglo. Esta función toma cuatro argumentos, pero a menudo solo se usan los dos primeros.

* **acumulador**: el valor devuelto de la iteración anterior
* **valorActual**: el elemento actual del arreglo
* **índice**: el índice del elemento actual
* **arreglo**: el arreglo original en la que se llamó a reduce
* **El argumento valorInicial es opcional**. Si se proporciona, se utilizará como valor acumulador inicial en la primera llamada a la función callback.

**Ejemplos**

El siguiente ejemplo suma todos los números juntos en un arreglo de números.

const *numeros* = [1, 2, 3, 4];  
const *suma* = *numeros*.reduce(function (resultado, elemento) {  
 return resultado + elemento;  
}, 0);  
*console*.log(*suma*); // 10

En el siguiente ejemplo, **reduce()** se usa para transformar un *arreglo de cadenas* en un único objeto que muestra *cuántas veces aparece cada cadena* en el arreglo. Observa que esta llamada para reducir pasa un *objeto vacío {}* como parámetro de **valorInicial**. Esto se usará como el valor inicial del acumulador (el primer argumento) pasado a la función callback.

var *mascotas* = ['perro', 'pollo', 'gato', 'perro', 'pollo', 'pollo', 'conejo'];  
  
var *recuentosDeMascota* = *mascotas*.reduce(function(obj, mascota){  
 if (!obj[mascota]) {  
 obj[mascota] = 1;  
 } else {  
 obj[mascota]++;  
 }  
 return obj;  
}, {});  
  
*console*.log(*recuentosDeMascota*);

salida en consola

>Object {perro: 2, pollo: 3, gato: 1, conejo: 1}

Continuando con el manejo de Arrays con el método reduce() a partir del minuto 25

<https://www.youtube.com/watch?v=J9vUZu6edBA>

Arrays – MAP, FOREACH, **REDUCE** y mucho MÁS!

Transformaremos “totalFactura” en un número.

const *facturas* = [  
 {  
 precio: 23,  
 cantidad: 1,  
 },  
 {  
 precio: 19,  
 cantidad: 2,  
 },  
 {  
 precio: 14,  
 cantidad: 4,  
 },  
 {  
 precio: 22,  
 cantidad: 1,  
 },  
]  
const *IVA* = 1.21;  
  
const *totalFactura* = *facturas*.map(i => i.precio \* i.cantidad \* *IVA*);  
  
const *total* = *totalFactura*.**reduce**((a, b) => a+b)  
  
*console*.log(*total*);

Usando map y reduce combinados puede lograrse el calculo de la suma, con el mismo resultado, y en menos líneas. El método map no es útil solo para transformar Arrays, si no como un Array temporal para poder realizar cálculos, para después realizar otras operaciones con eso.

const ***facturas*** = [  
 {  
 precio: **23,** cantidad: **1,** }**,** {  
 precio: **19,** cantidad: **2,** }**,** {  
 precio: **14,** cantidad: **4,** }**,** {  
 precio: **22,** cantidad: **1,** }**,**]  
const ***IVA*** = **1.21;**//const totalFactura = facturas.map(i => i.precio \* i.cantidad \* IVA);  
  
//const total = totalFactura.reduce((a, b) => a+b)  
  
const ***total*** = ***facturas***.map(i => i.precio \* i.cantidad \* ***IVA***).reduce((a**,**b) => {  
 ***console***.log(a**,**b)  
 return a + b  
})  
***console***.log(***total***)**;**

El método Array.filter() en JavaScript

El método Array.filter() es posiblemente el método más importante y ampliamente utilizado para iterar sobre un arreglo en JS.

Consiste en **filtrar uno o más elementos** (un subconjunto) ***de una colección más grande de elementos*** (un superconjunto) basándose en alguna **condición/preferencia**. Todos hacemos esto todos los días, ya sea leyendo, eligiendo a nuestros amigos o a nuestro cónyuge, eligiendo qué película ver, etc.

El método filter() toma una función callback y llama a esa función para cada elemento sobre el que itera dentro del arreglo de destino. La función callback de llamada puede tomar los siguientes parámetros:

* **valorActual**: Es el elemento del arreglo sobre el que se está iterando actualmente.
* **indice**: Es la posición del índice del **valorActual** dentro del arreglo.
* **arreglo**: Representa el arreglo de destino(final) junto con todos sus elementos.

El método **filter** *crea un nuevo arreglo y devuelve todos los elementos que pasan la condición* especificada en la llamada en el callback.

filter() **Ejemplo 1: Cómo filtrar elementos de un arreglo**

Filtramos todas las personas que son niños pequeños (cuya **edad** está comprendida **entre 0 y 4 años**).

let ***gente*** = [  
 {nombre: "aaron"**,** edad: **65**}**,** {nombre: "beth"**,** edad: **2**}**,** {nombre: "cara"**,** edad: **13**}**,** {nombre: "daniel"**,** edad: **3**}**,** {nombre: "ella"**,** edad: **25**}**,** {nombre: "fin"**,** edad: **1**}**,** {nombre: "george"**,** edad: **43**}**,**]  
  
let ***pequeños*** = ***gente***.filter(persona => persona.edad <= **3**)  
  
***console***.log(***pequeños***)**;**

Salida en Consola:

[

{ nombre: 'beth', edad: 2 },

{ nombre: 'daniel', edad: 3 },

{ nombre: 'fin', edad: 1 }

]

filter() **Ejemplo 2: Cómo filtrar elementos en función de una propiedad concreta**

En este ejemplo, solo filtraremos los miembros del equipo que sean desarrolladores.

***console***.clear()**;**let ***equipo*** = [  
 {  
 nombre: "aaron"**,** posicion: "developer"  
 }**,** {  
 nombre: "beth"**,** posicion: "ui designer"  
 }**,** {  
 nombre: "cara"**,** posicion: "developer"  
 }**,** {  
 nombre: "daniel"**,** posicion: "content manager"  
 }**,** {  
 nombre: "ella"**,** posicion: "cto"  
 }**,** {  
 nombre: "fin"**,** posicion: "backend engineer"  
 }**,** {  
 nombre: "george"**,** posicion: "developer"  
 }**,**]  
  
let ***desarrolladores*** = ***equipo***.filter(miembro => miembro.posicion == "developer")  
  
***console***.log(***desarrolladores***)**;**

Salida en consola:

[

{ nombre: 'aaron', posicion: 'developer' },

{ nombre: 'cara', posicion: 'developer' },

{ nombre: 'george', posicion: 'developer' }

]

En el ejemplo anterior, hemos filtrado a los desarrolladores. Pero, ¿y si queremos filtrar a todos los miembros que no son desarrolladores?

Podríamos hacerlo:

**let *desarrolladores* = *equipo*.filter(miembro => miembro.posicion !== "developer")**

Resultado en consola:

[

{ nombre: 'beth', posicion: 'ui designer' },

{ nombre: 'daniel', posicion: 'content manager' },

{ nombre: 'ella', posicion: 'cto' },

{ nombre: 'fin', posicion: 'backend engineer' }

]

**filter()** **Ejemplo 3: Cómo acceder a la propiedad índice**

Se trata de un concurso. En esta competición, hay tres ganadores. El primero obtendrá la medalla de oro, el segundo la de plata y el tercero, la de bronce.

Utilizando el **filter** y accediendo a la propiedad del **índice** *de cada elemento en cada iteración*, podemos filtrar cada uno de los tres ganadores en diferentes variables.

let *ganadores* = ["Anna", "Beth", "Cara"];  
  
let *oro* = *ganadores*.filter((ganador, indice) => indice == 0);  
let *plata* = *ganadores*.filter((ganador, indice) => indice == 1);  
let *bronce* = *ganadores*.filter((ganador, indice) => indice == 2);  
  
*console*.log(`Oro ganador: ${*oro*}, ganador de plata: ${*plata*}, ganador de bronce: ${*bronce*}`);

Salida en consola:

Oro ganador: Anna, ganador de plata: Beth, ganador de bronce: Cara

**filter()** **Ejemplo 4: Cómo utilizar el parámetro arreglo**

Uno de los usos más comunes del tercer parámetro (arreglo) es inspeccionar el estado del arreglo sobre el que se está iterando. Por ejemplo, podemos comprobar si queda otro elemento en el arreglo. Dependiendo del resultado, podemos especificar que sucedan diferentes cosas.

En este ejemplo, vamos a definir un arreglo de cuatro personas. Sin embargo, como ***solo puede haber tres ganadores, la cuarta persona de la lista tendrá que ser descontada***.

Para poder hacer esto, necesitamos obtener información sobre el arreglo de destino en cada iteración.

let ***competidores*** = ["Anna"**,** "Beth"**,** "Cara"**,** "David"]**;**function mostrarGanadores(nombre**,** indice**,** arreglo) {  
 let esSiguienteItem = indice + **1** < arreglo.length ? true : false**;** if (esSiguienteItem) {  
 ***console***.log(`El ganador #${indice+**1**} es ${nombre}.`)**;** } else {  
 ***console***.log(`Perdón, ${nombre} no es uno de los ganadores.`)**;** }  
}  
  
***competidores***.filter((nombre**,** indice**,** arreglo) => mostrarGanadores(nombre**,** indice**,** arreglo))

Salida en consola:

El ganador #1 es Anna.

El ganador #2 es Beth.

El ganador #3 es Cara.

Perdón, David no es uno de los ganadores.

**Cómo usar el objeto de contexto**

Además de la función de devolución de llamada, el método **filter()** también puede recibir un objeto de contexto.

**filter**(callbackfn, objetoContexto)

Este objeto puede ser referido desde dentro de la función del **callback** usando la referencia de la palabra clave **this**.

**filter()** **Ejemplo 5: Cómo acceder al objeto contexto con this**

Esto va a ser similar al **ejemplo 1**. Vamos a filtrar a todas las personas que tengan entre 13 y 19 años (adolescentes).

Sin embargo, *no codificaremos los valores dentro de la función* ***callback***. En su lugar, definiremos estos valores *13 y 19* como propiedades dentro del objeto **range**, que posteriormente *pasaremos a filter como el objeto contexto* (segundo parámetro).

Uso del objeto contexto

//Filtro por contexto en 2do parametro - Edades entre 13 y 19 años del Array Persona  
  
let ***personas*** = [  
 {nombre: "aaron"**,** edad: **65**}**,** {nombre: "beth"**,** edad: **15**}**,** {nombre: "cara"**,** edad: **13**}**,** {nombre: "daniel"**,** edad: **3**}**,** {nombre: "ella"**,** edad: **25**}**,** {nombre: "fin"**,** edad: **16**}**,** {nombre: "george"**,** edad: **18**}**,**]**;**let ***rango*** = {  
 bajo: **13,** alto: **16**}**;**let ***jovenes*** = ***personas***.filter(function(persona) {  
 return persona.edad >= this.bajo && persona.edad <= this.alto**;**}**, *rango***)  
  
***console***.log(***jovenes***)

Salida en consola:

[

{ nombre: 'beth', edad: 15 },

{ nombre: 'cara', edad: 13 },

{ nombre: 'fin', edad: 16 }

]

Pasamos el objeto rango como segundo argumento a **filter()**. En ese momento, se convirtió en nuestro *objeto de contexto*. En consecuencia, pudimos acceder a nuestros *rangos superior e inferior* en nuestra función de devolución de llamada con la referencia **this.alto** y **this.bajo** respectivamente.

**Conclusiones**

El método **filter()** del arreglo, *filtra los elementos que coinciden con la expresión del callback y los devuelve*.

Además de la función del callback, el método filter también puede tomar un *objeto de contexto como segundo argumento*. Esto te permitirá acceder a cualquiera de sus propiedades desde la función callback usando this.

Obtener los datos para aquellos productos cuyo COSTO TOTAL se mayor a “40”, para ello utilizaremos el método “filter”

Código del programa con uso del método Filter().

const ***facturas*** = [  
 {  
 precio: **23,** cantidad: **1,** }**,** {  
 precio: **19,** cantidad: **2,** }**,** {  
 precio: **14,** cantidad: **4,** }**,** {  
 precio: **22,** cantidad: **1,** }**,**]  
const ***IVA*** = **1.21;**const ***total*** = ***facturas***.map(i => i.precio \* i.cantidad \* ***IVA***).filter(i => i > **40**)  
***console***.log(***total***)**;**

Debe recordarse que el cálculo original de multiplicar el “precio x cantidad x IVA” da los siguientes resultados para los elementos contenidos en el Array de Facturas

Elemento 0 27.83

Elemento 1 45.98

Elemento 2 67.75999999999999

Elemento 3 26.619999999999997

De aquí podemos deducir que los únicos elementos que cumplen con la condición utilizada en el Filter son:

“Elemento 1 que es igual a **45.98**” y el

“Elemento 2 que es igual a **67.76**”

Salida en consola del nuevo Array devuelto por el método Filter

[ 45.98, 67.75999999999999 ]

COMPARADORES

Loose es: == o ¡=

Strict es: === o ¡==

Todo lo que pueda compararse con Loose, lo puedes comparar con Strict. Se recomienda el uso de “Strict”.

Si se requiere comparar un “String” con un “número” lo ideal es que convierta el “String” a “número” antes de compararlo. Si comparamos dos elementos, es importante saber que es mejor comparar esos 2 elementos siendo del mismo tipo.

CONVERSIÓN de TIPOS 📕➡📗 Las REGLAS del juego 😎 Curso de JAVASCRIPT desde CERO #3

<https://www.youtube.com/watch?v=ORpYqsTFxoI&t=77s>

SI QUEREMOS DEJAR UNA VARIABLE SIN DEFINIR, HACERLO CON UN “Let”

Let indefinido;

**String** es el tipo más fuerte cuando estamos sumando.

En el caso de la “suma” de dos valores, todo lo que sumemos con un “String” lo va a convertir siempre a un “String” (concatena los valores). Porque “String” es el tipo más fuerte cuando estamos sumando.

const ***cadena*** = "5"**;**const ***numero*** = **5;**const ***condicion*** = true**;**const ***nulo*** = null**;**let ***indefinido*;  
  
*console***.log(***cadena*** + ***numero***)**;  
*console***.log(***cadena*** + ***condicion***)**;  
*console***.log(***cadena*** + ***nulo***)**;  
*console***.log(***cadena*** + ***indefinido***)**;**

Resultado en consola:

55

5true

5null

5undefined

Este es el tipo de conversión más sencilla de hacer.

El 2do tipo de dato más fuerte es el tipo número.

***console***.log(***numero*** + ***condicion***)**;  
*console***.log(***numero*** + ***nulo***)**;  
*console***.log(***numero*** + ***indefinido***)**;**

Salida en consola:

6

5

NaN

Null toma el valor de 0 en una suma con un número.

NaN: significa “not a number” es un valor que nos indica que el resultado de una operación es de tipo número, pero no es un número.

No podemos sumar, restar, multiplicar o dividir 5 con algo que no existe.

Cuando restamos un número con una cadena, sucede un fenómeno diferente que con el de suma. Por ejemplo, si hacemos:

5 – “5”, la operación de resta en este caso se cumple, porque la operación “menos” no existe en “Strings” con lo cual el resultado sería 0. Igual ocurriría con las operaciones de: multiplicación, división, módulo. No existen con “String” entonces no puede haber un resultado “String”. Pero si resto un número de una cadena como “hola” nos que nos dará como resultado será el tipo “NoN” (not a number), es decir, si queremos restar un número y una cadena, si la cadena no se puede convertir implícitamente a número, entonces el resultado daría un “NoN”.

Las conversiones implícitas cuando son de distinto tipo, solo pueden convertirse a “String” o a “número”. Tenemos que decidir una de las 2.

**OBJETOS PREDIFINIDOS / PROTOTIPADOS DE JS**.

String

Number(valor) convierte el valor a número.

Boolean; la única que vez que en una conversión de cadenas “string” nos dará “false” será cuando la cadena esté vacía “”. Es la única vez que al convertir una “cadena” a un booleano nos dará falso.

boolean(“ ”); //false

Para todo el resto de opciones nos dará “true”. Por ejemplo;

boolean(“false”); //true

boolean(“-1”); //true

todo lo que sea cadena de caracteres nos dará “true”.

Pasa lo mismo con los números, el 0 es el único valor que nos dará “false” y de ahí todo valor numérico nos dará true. Sean negativos, positivos, exponenciales.

Boolean(null); //false

boolean(undefined); //false

Métodos de conversión

Si tengo cadena = “5.5” y quiero la parte entera, JS ha sacado un par de métodos para ello.

parseInt()

parseFloat()

Estos nos sirven para convertir cualquier cadena o cualquier otro elemento en un “número” con su parte entera (parseInt()) o con su parte décimal (parseFloat()).

//conversión de cadena a número, ya sea con decimales o entero.  
const ***cadena2*** = "5.5"**;  
  
*console***.log(parseFloat(***cadena2***))**; //5.5  
*console***.log(parseInt(***cadena2***))**; //5  
*console***.log(parseInt(***numero***))**;** //convierte número decimal 5.25 a numero entero 5

La conversión EXPLICITA, es cuando ordenamos en JS que convierta algo de una forma explícita.

En JS el tipado de las variables es dinámico, es decir, que el tipo de datos que representa una variable vendrá representado por el valor asignado a esa variable. Puedes cambiar el tipo las veces que desees, la variable inicialmente puede contener un número, pero podrías asignarle una cadena “hola” y no hay problema con JS porque su tipado es dinámico.

¿Qué es una IIFE en JavaScript? Ejemplos y explicación Expresión de función ejecutada inmediatamente

IIFE: Immediately Invoked Function Expression

Self Executing Anonymous Function

Self Execution Invoking Function

Es una función que se invoca inmediatamente

El siguiente código no representa una función “IIFE” porque no la estás ejecutando inmediatamente. Primero, guardamos la función en una constante “sayHi” y luego la ejecutamos sayHi().

const sayHi = () => {  
 ***console***.log('Hola')  
}  
sayHi()**;**

La siguiente, si es una función **IIFE**, la envolvemos con 2 paréntesis en “rojo”, y la ejecutamos inmediatamente con estos paréntesis que hemos coloreado en “verde” **()**

//Funcion IIFE  
**(**() => {  
 ***console***.log('Hola')  
}**)()**

Esto quiere decir que nos estamos haciendo la “Expresión” y la estamos invocando inmediatamente.

Se ocupa IIFE por diferentes razones, imaginar que tenemos un solo archivo, y tenemos variables que de alguna forma no contaminen el “scope” global

//Funcion IIFE  
(() => {  
 let b = 'hola' //esta variable no está disponible fuera del ambito de esta funcion IIFE  
 ***console***.log('Hola')  
})()

Para usar el “async await” se usa una IIFE. Se utiliza la IIFE para tener variables que no estén accesibles fuera de esa función.

//Funcion IIFE  
(() => {  
 ***console***.log('Hola')  
})()

La primera parte del código y la función IIFE hacen los mismo. La diferencia es que una no se ejecuta de inmediato, y la IIFE si se ejecuta de inmediato.

Lo anterior se puede hacer también con un “Arrow Function” o con una función normal. Esto se hace desde hace tiempo atrás, no es nada nuevo. Se utilizaba mucho para hacer “librerías”.

(function **jQuery**() {  
 ***console***.log('Hola')  
})()

NO utilices SWITCH ⛔ ¿QUÉ ALTERNATIVAS hay? 🟡 Javascript AVANZADO

<https://www.youtube.com/watch?v=7JOVJv0EcbE>

Desarrollo Útil

El switch se suele utilizar para simplificar las típicas cadenas de if… else… else… if... anidadas, que a nivel estructural son un caos y a nivel de complejidad no escalan bien. Pero cuando los sustituimos por un switch acabamos teniendo el mismo problema cuando eso escala. Ya que no deja de ser una estructura parecida a un if… else… con la diferencia que aquí tenemos los típicos case con los breaks, etc. Haciendo el código bastante ilegible y complejo.

Para ver este caso, usaremos el ejemplo de piedra, papel o tijera.

//No utilizar Switch - Qué alternativas hay - JS Avanzado  
*console*.clear();  
const *GAME\_OPTIONS* = ['PIEDRA', 'PAPEL', 'TIJERA'];  
  
/\*\*  
 \* Función que ejecuta el juego de piedra papel y tijera  
 \* @param player1 Opción del jugador1  
 \* @param player2 Opción del jugador2  
 \*/  
function game(player1, player2){  
 if(!*GAME\_OPTIONS*.includes(player1) || !*GAME\_OPTIONS*.includes(player2))  
 throw new Error('Opcióon de juego no válida');  
 if(player1 === player2) return "Empate";  
  
 switch (player1){  
 case 'PIEDRA':  
 switch(player2){  
 case 'PAPEL':  
 return 'Player 2 gana';  
 default:  
 return 'Player 1 gana';  
 }  
 case 'PAPEL':  
 switch(player2){  
 case 'TIJERA':  
 return 'Player 2 gana';  
 default:  
 return 'Player 1 gana';  
 }  
 case 'TIJERA':  
 switch(player2){  
 case 'PIEDRA':  
 return 'Player 2 gana';  
 default:  
 return 'Player 1 gana';  
 }  
 }  
}  
*console*.log(game('PIEDRA', 'TIJERA'));

El Switch anterior no se entiende tanto, así que veremos una forma de optimizar el código anterior. Sin necesidad de utilizar Switch. Utilizaremos un Objeto con Propiedades que a su vez cada propiedad tiene una sub-propiedad.

//No utilizar Switch - Qué alternativas hay - JS Avanzado  
*console*.clear();  
const *GAME\_OPTIONS* = ['PIEDRA', 'PAPEL', 'TIJERA'];  
  
//Objeto GAME\_RULES dirá en función del valor del jugador1 si gana (propiedad es true) o pierde (propiedad es false)  
//Cómo "false" vale igual que "undefined" podemos ahorrarnos el valor de la 2da sup-propiedad  
const *GAME\_RULES* = {  
 PIEDRA: {  
 TIJERA: true,  
 //PAPEL: false,  
 },  
 PAPEL: {  
 PIEDRA: true,  
 //TIJERA: false,  
 },  
 TIJERA: {  
 PAPEL: true,  
 //PIEDRA: false  
 }  
}  
  
/\*\*  
 \* Función que ejecuta el juego de piedra papel y tijera  
 \* @param player1 Opción del jugador1  
 \* @param player2 Opción del jugador2  
 \*/  
function game(player1, player2){  
 if(!*GAME\_OPTIONS*.includes(player1) || !*GAME\_OPTIONS*.includes(player2))  
 throw new Error('Opcióon de juego no válida');  
 if(player1 === player2) return "Empate";  
  
 if(*GAME\_RULES*[player1][player2]) return 'Player1 gana';  
 else return 'Player2 gana';  
}  
*console*.log(game('PAPEL', 'PIEDRA'));

VER VIDEOS DE YOUTUBER SIGUIENTE, RECOMENDADOS EN TYPESCRIPT Y REACT

Carlos Azaustre en español y Jack Herrington