JAVA SCRIPT

🤯LEVEL 3🤯

DISCLAIMER

**Todo el código existente en este material es pensando con fines didácticos y no necesariamente propone la forma idónea de implementarse en proyectos reales.**

ESTRUCTURA DEL RESUMEN:

El resumen pretende brindar un pantallazo general de

JavaScript

¡NO SE ABARCAN TODOS LOS CONCEPTOS EXHAUSTIVAMENTE!

a lo largo del resumen se usan 4 tipos de títulos:

# TITULO 1

## TITULO 2

### TITULO 3

#### TITULO 4

los cuales se siguen de sus respectivas tipografías:

“Siendo esta la letra por defecto para los títulos 1 y 2”

“Y siendo esta la letra por defecto para los títulos 3 y 4”

Cuando se trata de referencias externas o documentación particular

Se usa esta tipografía

Cuando se quiere hacer un comentario informal con el fondo blanco se usa esta tipografía

Cuando se quiere hacer un comentario informal con el fondo negro se usa esta tipografía

Cuando se quiere hacer un comentario importante o más técnico se usa este color y esta tipografía.

Después de cada tema suele haber una muestra tentativa de la sintaxis que describe *grosso modo* la idea de cómo se organiza/estructura determinada palabra reservada y su funcionamiento.

Y después de la sintaxis *suele haber* un ejemplo

Y dos etiquetas: una html y una javascript con sus respectivos colores:

de este modo:

Sintaxis:

EJEMPLO:

HTML:

JavaScript:

*LA ETIQUETA HTML:*

muestra lo que contendría el archivo html pero ignora las etiquetas fundamentales de todo documento html

A ESCLARECER:

<!DOCTYPE html>

<html></html>

<head></head>

<body></body>

teniendo en cuenta que el lector tiene una comprensión de

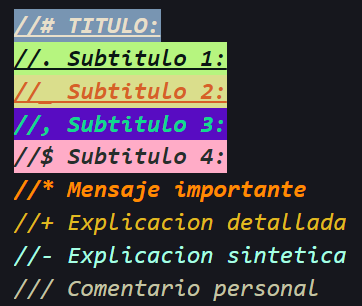
html y css

*LA ETIQUETA JAVASCRIPT:*

se remite a mostrar cada caso de uso particular, para que el lector pueda ver y razonar las diferentes formas de trabajar en los diferentes temas con javascript.

*RESPECTO A LOS COMENTARIOS:*

los códigos están comentados con colores para que sea mas practico y ameno poder leerlos, además estos comentarios tienen una jerarquía particular que siguen todos.



para que esto sea posible, se utiliza la siguiente extensión…



Gracias a esto la interacción entre el resumen y el código es más intuitiva y menos tediosa de relacionar.

por defecto la extensión viene con una configuración que brinda otros colores en los comentarios.

para poder tener exactamente los mismos colores que los vistos en la imagen anterior es necesario cambiar la configuración de ajustes personales del editor de texto.

el Json que permite la configuración de color de estos comentarios es el siguiente:

  "better-comments.tags": [

    {

*//#comentario*

      "tag": "#",

      "color": "#E8DFCA",

      "strikethrough": false,

      "underline": true,

      "backgroundColor": "#7895B2",

      "bold": true,

      "italic": false

    },

    {

*//.comentario*

      "tag": ".",

      "color": "#0A1714",

      "strikethrough": false,

      "underline": true,

      "backgroundColor": "#B6F57F",

      "bold": true,

      "italic": true

    },

    {

*//\_comentario*

      "tag": "\_",

      "color": "#d66127",

      "strikethrough": false,

      "underline": true,

      "backgroundColor": "#dade8c",

      "bold": true,

      "italic": true

    },

    {

*//,comentario*

      "tag": ",",

      "color": "#0fdb8d",

      "strikethrough": false,

      "underline": false,

      "backgroundColor": "#580cc2",

      "bold": true,

      "italic": false

    },

    {

*//$comentario*

      "tag": "$",

      "color": "#262e2b",

      "strikethrough": false,

      "underline": false,

      "backgroundColor": "#FFACC7",

      "bold": true,

      "italic": true

    },

    {

*//\*comentario*

      "tag": "\*",

      "color": "#FF8C00",

      "strikethrough": false,

      "underline": false,

      "backgroundColor": "transparent",

      "bold": true,

      "italic": false

    },

    {

*//+comentario*

      "tag": "+",

      "color": "#E0BB20",

      "strikethrough": false,

      "underline": false,

      "backgroundColor": "transparent",

      "bold": false,

      "italic": false

    },

    {

*//-comentario*

      "tag": "-",

      "color": "#A7FFE4",

      "strikethrough": false,

      "underline": false,

      "backgroundColor": "transparent",

      "bold": false,

      "italic": false

    },

    {

*///comentario*

      "tag": "/",

      "color": "#c4c4a5",

      "strikethrough": false,

      "underline": false,

      "backgroundColor": "transparent",

      "bold": false,

      "italic": true

    }

ES POSIBLE QUE A LO LARGO DEL RESUMEN CAMBIE EL COLOR DE LAS PALABRAS RESERVADAS, YA QUE AL LLEVAR TANTO TIEMPO HACERLO SE HAN USADO DIFERENTES PERSONALIZACIONES, NO OBSTANTE, EL CODIGO FUNCIONA IGUAL.

### **GLOSARIO ESCENCIAL:**

##### SCRIPT:

Se le llama script a cualquier porción de código, literalmente significa “guion”

##### ELEMENTO:

Cada etiqueta **HTML** es un elemento customizable en **CSS** y dinamizable en **JavaScript**.

##### ATRIBUTO O PROPIEDAD:

Una **propiedad** o un **atributo** es una característica que un elemento de cierto tipo puede poseer.

##### VALOR:

Los valores sirven para darle un comportamiento diferente a cada atributo de cada elemento.

##### FUNCION:

Una función es una porción de código que puede recibir parámetros y devolver un resultado

##### METODO:

Un método es una función específica de un objeto.

##### JSON:

*Java script object notation*, es un tipo de escritura, útil para transportar datos.

##### API:

*Application programming interface*, es un grupo de funcionalidades o datos disponibles para importarlas en otro software.

##### SINCRONIA Y ASINCRONIA:

La **sincronía** hace referencia a código que se ejecuta secuencialmente en un orden.

La **asincronía** hace referencia a código que se ejecuta independientemente y sin un orden fijo.

##### PROMESA:

Una promesa es una función asíncrona que devuelve el resultado de una evaluación

##### OBJETO:

En JS todo excepto los datos primitos representan un objeto, estos tienen una estructura formada por: propiedades con valores y métodos.

##### CLASE:

Una clase representa una estructura básica que pueden compartir varios objetos

##### HERENCIA:

Gracias a la herencia los objetos y las clases pueden heredar partes deseadas de la estructura que los conforman

### **SIMBOLOS:**

Estos mismos pueden aparecer después de los títulos para indicar lo que simbolizan.



Simboliza que la propiedad, función o método es un **shorthand.**Es decir una propiedad que engloba varias propiedades.



Simboliza que la propiedad, función o método o se usa para obtener un valor u otorgar un valor.

“**GET** AND **SET**”



Simboliza que la sintaxis de escritura lleva paréntesis al final por ser un método.

### Características de Java Script:

#### Es interpretado:

***JavaScript*** es un lenguaje interpretado porque utiliza un intérprete que permite convertir las líneas de código en el lenguaje de la máquina. Esto tiene un gran número de ventajas como la reducción del procesamiento en servidores web al ejecutarse directamente en el navegador del usuario, o que es apto para múltiples plataformas permitiendo usar el mismo código.

#### Orientado a objetos:

***JavaScript*** es un lenguaje orientado a objetos. Que un lenguaje esté orientado a objetos quiere decir que utiliza clases y objetos como estructuras que permiten organizarse de forma simple y son reutilizables durante todo el desarrollo.

#### Imperativo:

Todas las instrucciones se ejecutan de línea en línea en comparación de otros lenguajes donde se ejecuta todo en conjunto de forma exclusiva.

#### Case Sensitive:

Es sensible a MAYUSCULAS y minúsculas

#### Tipado débil:

El valor de las variables puede cambiar durante la ejecución y no es necesario especificar el tipo de dato al crear variables

#### Dinamico:

La variable no se ajusta al dato el dato se ajusta a la variable.

#### De alto nivel:

Que ***JavaScript*** sea un lenguaje de alto nivel significa que su sintaxis es fácilmente comprensible por su similitud al lenguaje de las personas. Se le llama de “alto nivel” porque su sintaxis se encuentra alejada del nivel máquina, es decir, del código que procesa una computadora para ejecutar lo que nosotros programamos.

### Usos de JavaScipt:

* Dinamismo en Sitios web: Para poder lograr una interacción dinámica del lado del cliente.

#### Otras posibilidades:

* Inteligencia artificial: Como la librería tensorflow desarrollada por Google.
* Placas electrónicas: Johnny five con Arduino.
* Mobile apps: Es posible crear aplicaciones móviles
* Desktop apps: Es posible crear aplicaciones para escritorio

Aunque no necesariamente **JavaScript** sea el lenguaje más óptimo para los tipos de desarrollos mencionados previamente, es interesante conocer sus potenciales posibilidades.

# CALLBACKS:

Si ya leíste todos los resúmenes anteriores de JavaScript, has visto y posiblemente usado callbacks, montonazo de veces

Leyendo la documentación de JavaScript vas a ver varias veces la palabra “callback”

La definición que a menudo vas a escuchar de un callback es:

“una función que se pasa como parámetro a otra función”

Y es fácil escucharlo y decirlo, pero sus usos pueden ir de los más simple a lo más complejo, así que adentrémonos en la definición técnica:

En [programación informática](https://es.wikipedia.org/wiki/Programaci%C3%B3n), una **retrollamada** o devolución de llamada (en inglés: ***callback***), también denominada «posllamada» (o «*call-after*» en inglés), es una [función ejecutable](https://es.wikipedia.org/wiki/Ejecutable) «A» que se usa como [argumento](https://es.wikipedia.org/wiki/Argumento_(Ciencias_de_la_computaci%C3%B3n)) de otra función «B». De esta forma, al llamar a «B», esta ejecutará «A». Esta acción puede ser inmediata, lo que se denominará **retrollamada sincronizada** o puede producirse en un punto posterior, lo que se denominaría **retrollamada asíncrona**.

Sintaxis:

function funcionQueLlama(FuncionComoParametro){FuncionLlamada()};

EJEMPLO:

Empecemos por entender que una función puede ejecutar otra función.

***JavaScript:***

// Una función ejecutando otra función:

function funcion\_1() {funcion\_2()};

function funcion\_2() {console.log ("Hola, soy codigo de la funcion\_2")};

funcion\_1();

******

¿Esto claramente no tendría sentido no?

Para lograr lo mismo podríamos poner el código de la función\_2 en la función\_1 y nos ahorraríamos una funcion.

Pero que pasa si quisiéramos que nuestra función ejecute cualquier función, no solo una especifica.

Supongamos que tenemos las siguientes funciones:

function funcion\_2() {console.log ("Hola, soy codigo de la funcion\_2")};

function funcion\_3() {console.log ("Hola, soy codigo de la funcion\_3")};

function funcion\_4() {console.log ("Hola, soy codigo de la funcion\_4")};

function funcion\_5() {console.log ("Hola, soy codigo de la funcion\_5")};

Una opción seria poner en su bloque de ejecución el nombre de la función que quisiéramos que se ejecute…

Alguna de estas opciones:

function funcion\_1(){funcion\_2()};

function funcion\_1(){funcion\_3()};

function funcion\_1(){funcion\_4()};

function funcion\_1(){funcion\_5()};

o poniéndolas todas juntas:

function funcion\_1(){funcion\_2(),funcion\_3(),funcion\_4(),funcion\_5()};

Pero lamentablemente esta practica es horrenda y no es la manera correcta.

Que tal si… proponemos que la funcion\_1 reciba un parámetro… y que ese parámetro sea una funcion

Puede ser cualquier tipo de funcion…

//# CALLBACK SINCRONO:

function funcion\_1(funcionComoParametro){};

function funcion\_2(){console.log("Hola, soy una  funcion call Back")};

funcion\_1(funcion\_2());



Donde: funcionComoParametro hace referencia a cualquier función que le pasemos.

Y esa función que le pasemos es una función CallBack

Esta función callback, no es exclusiva como en el primer ejemplo, es un argumento que representa cualquier función existente.

function funcion\_3() {console.log ("Pero pude haber sido yo... la funcion\_3")};

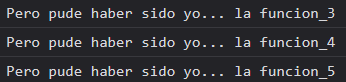
function funcion\_4() {console.log ("Pero pude haber sido yo... la funcion\_4")};

function funcion\_5() {console.log ("Pero pude haber sido yo... la funcion\_5")};

funcion\_1(funcion\_3());

funcion\_1(funcion\_4());

funcion\_1(funcion\_5());



\*Dependiendo de los requerimientos de cada función cada callback suple las necesidades previstas\*

Lo que se intenta explicar en ese mensaje resaltado es que si una función\_A necesita otra funcion\_B callback que sume dos números (por poner un ejemplo random) entonces esta funcion\_B callback no debe ser cualquier función.

De modo que cuando se usa una función callback se lo hace intencionalmente para un uso especifico. ☺♥

Como vimos en los ejemplos anteriores, estas funciones que llaman a otras funciones están intencionalmente definidas para que ocurran de forma SECUENCIAL o SINCRONIZADA

Pero otro uso de los callbacks es para que ocurran de forma no SECUENCUAL y ASINCRONA.

Un ejemplo de eso es la funcion setTimeout(Funcion\_name, time\_value); de Java Script, que ejecuta una funcion después de que pasa cierto periodo de tiempo

//# CALLBACK ASINCRONO:

function callbackFunction() {console.log("Buenas, soy una funcion callback")};

setTimeout(callbackFunction, 1000);



Después de 1000 milisegundos = 1 segundo, se ejecuta una función, que en este caso mostraba un mensaje por consola.

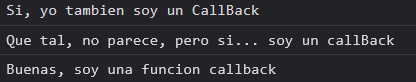
setTimeout(function(){console.log("Que tal, no parece, pero si... soy un callBack")}, 500)



setTimeout(() => {console.log("Si, yo tambien soy un CallBack")}, 200)



Si ejecutamos las tres al mismo tiempo en el orden anterior, veremos esto:



Claramente se ejecutaron al revés de como se esperaba, y esto es porque sus tiempos de espera estaban intencionalmente escritos así, pero estos tiempos de espera usualmente pueden ser tiempos de espera reales que se generen porque una pagina tiene que terminar de cargar, o porque un usuario esta validando su mail para registrarse, etc…

¿Y como podría hacerse si necesitamos que un callback, espere a otro callbacks antes de ser ejecutado?

Ya podrán deducir que se necesitaría que uno este adentro de otro, y este adentro de otro, y este adentro de otro… AD INFINITUM.

Es lo que dentro del nicho informático se lo conoce como CALLBACK HELL

Y para solucionar este problemón de tener un código totalmente ilegible, se crearon las promesas…

# Programación orientada a objetos (POO)

La programación orientada a objetos emula las características que tendrían los objetos en la vida real, pero abstrayéndolas a código

Pudiendo así atribuirle a un “objeto virtual” cualidades, propiedades/atributos que se pueden heredar; así como también los objetos pueden ser parte de una clase.

Es una de las mayores virtudes de **JS** y tiene mucho potencial.

Los objetos pueden contener cualquier tipo de dato:

* numéricos
* alfanuméricos
* strings
* booleanos
* arrays
* funciones
* otros objetos

## Creación de objetos:

Los objetos contienen duplas ***llaves: valor*** o ***key:*** ***value***

Es decir, los objetos ***tienen propiedades que tienen un valor***

Estas **propiedades** de las conoce como **llaves** ya que llevan un **nombre**

Y las llaves hacen referencia a un valor

En otras palabras, llamando a la llave, obtenemos su valor.

Sintaxis**:**

*const* object\_name = {

   key\_name1: value1,

   key\_name2: value2,

   key\_name3: value3,

   key\_name4: value4

}

Cada **KEY** tiene asignado un **VALOR** después de **“ : ”**

Cada **dupla** es un **CAMPO** del objeto

Cada **CAMPO** se separa por **“ , ”**

El ultimo **CAMPO** no es necesario que tenga

EJEMPLO:

JavaScript:

**EJEMPLO 1:**

*const* persona = {

    nombre: "carlos",

    apellido: "pergolini",

    edad: 29.8,

    nacionalidad: "marciano",

    existe: true

}

EJEMPLO 2:

*const* frutas = {

    1: "banana",

    2: "manzana",

    3: "pera",

    4: "morron"

}

EJEMPLO 3:

*const* numero = {

    "numero 4": 4,

    "numero 20": 20,

    "numero 30": 30

}

Lo primero interesante a ver en estos ejemplos es que

hay mucha libertad tanto a la hora de definir las key : value

Recordemos que en JS todo son objetos, y en/con ellos podemos hacer muchas cosas.

### Objetos dentro de objetos:

Dentro de lo que podemos incluir en un objeto están los objetos mismos

EJEMPLO:

*const* saludos = {

    comun: "hola!",

    animado : "Hola que bueno verte",

    excentrico: "Me exalto ante su presencia"

}

*const* nombres = {

    1: "Miguel Angel" ,

    2: "Clarita",

    3: "Socrates"

}

*const* saludarNombres = {

    saludos: saludos,

    nombres : nombres

}

console.log(saludarNombres.saludos);

console.log(saludarNombres.nombres);

**

\*Si tenemos **lo mismo** en ambos lados del **“:”** podemos escribirlo **una sola vez\***

*const* saludarNombres = {

    saludos,

    nombres

}

Funciona exactamente igual.

## Interaccion con el objeto

Podemos interactuar con un objeto y sus características de varias formas.

Ya sea creando, mostrando, modificando o eliminando.

### **mostrar el objeto**

Sintaxis***:***

console.log(object\_name); *//mostraría el objeto*

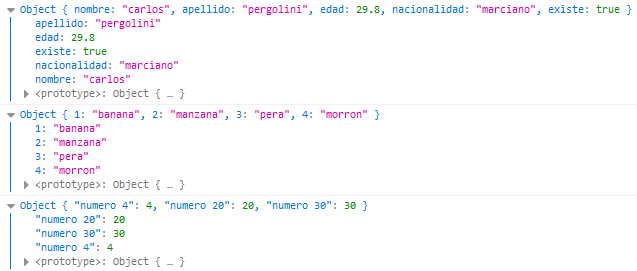
*EJEMPLO:*

console.log(persona);

console.log(frutas);

console.log(numero);



**

\*\*\*Nótese que las propiedades se han ordenado alfabéticamente\*\*\*

### **mostrar sus propiedades:**

Sintaxis***:***

console.log(object\_name.poperty\_name); *//mostraría una propiedad de un objeto*

console.log(object\_name.poperty\_name1.poperty\_nameN°);

EJEMPLO:

console.log(persona.nombre);

console.log(frutas[1]);

console.log(numero["numero 4"]);



### Modificar propiedades:

Las propiedades de un objeto son variables, y es posible modificar su valor desde afuera.

Sintaxis**:**

Object\_name.property\_name = new\_value

*EJEMPLO:*

*// antes:*

console.log(persona.nombre);

console.log(frutas[1]);

console.log(numero["numero 4"]);

persona.nombre =  "Rigoberto"

frutas[1] = "melocoton"

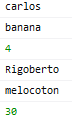
numero["numero 4"] = 15+15;

*// despues:*

console.log(persona.nombre);

console.log(frutas[1]);

console.log(numero["numero 4"]);



### Eliminar propiedades:

Podemos eliminar las propiedades de un objeto.

Sintaxis:

delete Object\_name.property\_name ;

EJEMPLO:

console.log(persona);*// antes:*

delete persona.apellido;

delete persona.nacionalidad;

console.log(persona); *// después:*



### Buscar propiedades:

Permite corroborar si existe una propiedad en un objeto.

Sintaxis:

Object\_name.hasOwnProperty('property\_name'));

EJEMPLO:

*const* persona = {

    nombre: "carlos",

    apellido: "pergolini",

    edad: 29.8,

    nacionalidad: "marciano",

    existe: true

}

*const* frutas = {

    1: "banana",

    2: "manzana",

    3: "pera",

    4: "morron"

}

console.log(persona.hasOwnProperty('nombre'));

console.log(persona.hasOwnProperty('peso'));

console.log(frutas.hasOwnProperty('1'));

console.log(frutas.hasOwnProperty('7'));



### Frizar un objeto:

Se le llama frezar un objeto a aplicarle un método existente en JS

Para no poder alterar las propiedades ni los valores del objeto, de esta forma es estático y podemos garantizar que no se puede alterar desde afuera.

Sintaxis:

Object.freeze(object\_name);

*EJEMPLO:*

const ubicacion = {

    EjeX : "10",

    EjeY : "42",

    EjeZ : "99"

}

//- objeto orginal:

console.log(ubicacion)

//- objeto mutable antes del friz:

ubicacion.EjeY = "987654";

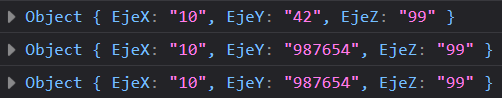
console.log(ubicacion)

//- objeto inmutable despues del friz:

Object.freeze(ubicacion);

ubicacion.EjeY = "0"

console.log(ubicacion);



Dependiendo del navegador, puede que tengamos un mensaje por consola que nos indica la inmutabilidad del objeto, o que simplemente no muestre nada, pero tampoco lo deje editar

## DESESTRUCTURACION:

La sintaxis de desestructuración es una forma de expresión en JS que permite desempacar valores de arreglos o propiedades de objetos en distintas variables.

Se basa en una forma mucho más legible y practica

### Destructuring objects:

Se puede usar la sintaxis de desestructuración para poder obtener de forma legible y practica los valores de las propiedades de un objeto.

Sintaxis:

Let/var/const {keyName\_1,…,keyName\_N} = object\_name;

EJEMPLO:

Ejemplo 1:

Con lo que sabemos acerca de objetos, en caso de tener un objeto como este:

const persona = {

    nombre:"pepe" ,

    apellido:"romano" ,

    edad:"24" ,

}

Para mostrar los valores de sus campos deberíamos haber recurrido a escribir algo asi:

const nom = persona.nombre;

const ape = persona.apel;

const eda = persona.edad;

gracias a la sintaxis de desestructuración podemos reemplazarlo por esto:

const {nombre,apellido,edad} = persona;

lo cual es mucho mas legible y practico.

console.log(`${nombre} ${apellido} ${edad} años`);



Ejemplo 2:

*const* cartuchera = {

    lapices: 30,

    fibrones : ["azul","amarillo","verde"],

    cierre: true,

    alto : "6cm",

    largo : "17cm"

}

*const* {lapices, fibrones, cierre} = cartuchera *//destructuracion del objeto*

console.log(lapices,cierre,fibrones);



Si quisiéramos acceder a objetos anidados, es decir objeto dentro de objetos:

//$ Ejemplo 3:

const comida = {

    frutas: {

        manzana: "roja",

        pera: "verde",

        morron: "amarillo",

    }

}

const {frutas: {manzana,pera,morron}} = comida;

console.log(`manzana: ${manzana} pera: ${pera} morron: ${morron}`);



Pero si tuviermos dos objetos, tendríamos un problema ya que no podemos tener constantes con el mismo nombre.

Ahí es donde aparece para salvarnos el alias.

#### Alias:

Permite asignar un segundo nombre a las claves de la sintaxis de desestructuración.

Además si por alguna razón tenemos una variable con un scope global que coincide con el nombre de nuestra key podemos asignarle un alias.

Sintaxis:

Let/var/const {key\_objectNAme: {keyNameObject: aliasNameObject} } = object\_name;

EJEMPLO:

Supongamos este objeto:

const libros = {

    homo\_deus: {

        autor: "Yuval Noah Harari",

        genero: "historia",

        publicacion: "2015",

    },

    guerra\_y\_paz: {

        autor: " Leon Tolstói",

        genero: "novela belica",

        publicacion: "1867",

    }

}

Si quisiéramos hacer esto:

const {homo\_deus:{autor,genero,publicacion}} = libros;

const {guerra\_y\_paz:{autor,genero,publicacion}} = libros;

tendríamos un problema porque estaríamos redeclarando constantes.

Por consola:

**

En VS Code:

**

Podríamos pensar que una solución sería cambiar el tipo de variable constante:

var {homo\_deus:{autor,genero,publicacion}} = libros;

var {guerra\_y\_paz:{autor,genero,publicacion}} = libros;

console.log(autor,genero,publicacion);

pero obtendríamos solo uno de los resultados, puntualmente el ultimo que se asigno

**

La solución es ponerle un alias a cada variable de este modo:

const {

  homo\_deus: {

    autor: autor\_Hom,

    genero: genero\_Hom,

    publicacion: publicacion\_Hom,

  },} = libros;

const {

  guerra\_y\_paz: {

    autor: autor\_Tol,

    genero: genero\_Tol,

    publicacion: publicacion\_Tol,

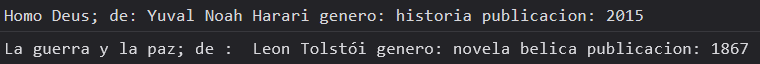
  },} = libros;

console.log(

  `Homo Deus; de: ${autor\_Hom} genero: ${genero\_Hom} publicacion: ${publicacion\_Hom}`);

console.log(

  `La guerra y la paz; de : ${autor\_Tol} genero: ${genero\_Tol} publicacion: ${publicacion\_Tol}`);



#### Objetos como argumento:

Podemos usar la desestructuración para pasar un objeto o sus propiedades como argumentos:

Sintaxis:

const objectName = {}

functionName\_1 = (argName)=>{CODIGO}

functionName\_2 = ({key\_1,…,Key\_N°})=>{ CODIGO }

obtenerElementos(objectName);

*EJEMPLO:*

Supongamos el siguiente objeto:

const elementos = {

  Cl:"Cloro",

  Ar:"Argón",

  K:"Potasio",

  Ca:"Calcio"

}

Podemos usar la desestructuración para mostrar las propiedades del objeto.

const obtenerElementos = (objetoNombre)=>{

  const {Cl,Ar,K,Ca} = objetoNombre;

  console.log(Cl);

  console.log(Ar);

  console.log(K);

  console.log(Ca);

}

obtenerElementos(elementos);

pero esto no cambiar mucho a trabajar con el objeto directamente.



En cambio podemos usar una desestructuración un poco más legible y práctica.

const obtenerElementos = ({Cl,Ar,K,Ca})=>{

  console.log(Cl);

  console.log(Ar);

  console.log(K);

  console.log(Ca);

}

obtenerElementos(elementos);



Obtendríamos exactamente lo mismo.

De esta forma podemos trabajar con las propiedades necesarias de un objeto.

const mostrarDosElementos = ({Cl,Ca})=>{console.log(`Cl significa ${Cl}: Ca significa: ${Ca}`);}

mostrarDosElementos(elementos)



#### Consision en la creacion de objetos:

Podemos crear objetos de forma muchísimo más legible y concisa gracias a la sintaxis de desestructuración.

Sintaxis:

const objectName = (argumento\_1,…,arguemento\_N°)=>({argumento\_1,…,argumento\_N°});

*EJEMPLO:*

const criptoMoneda = (cantidad,valor,nombre,minable)=>({cantidad,valor,nombre,minable});

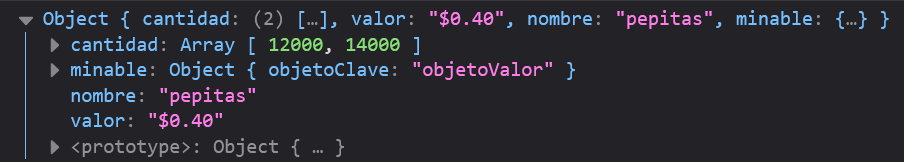
console.log(criptoMoneda(1692,"$0.89","criptonita",true));

**

Claramente podemos tambien asignarlo a una variable y pasar cualquier tipo de dato:

const pepe = criptoMoneda([12000,14000],"$0.40","pepitas",{objetoClave:"objetoValor"});

console.log(pepe);

**

### Destructuring arrays:

También se pueden desestructurar los arreglos para trabajar con sus valores de una forma particular.

Sintaxis:

*Let/var/const* [ key\_name1,…,key\_nameN°] = array\_name

EJEMPLO:

Ejemplo 1:

var a;

var b;

[a,b] = [1,2];

console.log(a);

console.log(b);

**

Ejemplo 2:

en caso de que queramos usar un numero definido de variables, pero queramos acceder a posiciones relativamente lejanas del arreglo, no necesitamos crear variables hasta alcanzar el índice:

var a;

var b;

var c;

[a, b,,,,,c] = [1, 2,3,4,5,6,7,8];

console.log(a);

console.log(b);

console.log(c);

usamos tantas comas de más como elementos queramos saltear



Ejemplo 3:

const te = ["menta","jengibre","rosa mosqueta","verde"];

const [te1,te2,te3,te4] = te;

console.log(te1,te2)

**

En caso de que tengamos una variable que no existe:

const [te1,te2,te3,te4,te5] = te;

console.log(te4,te3,te5)



También podemos usar valores default.

const [te1,te2,te3,te4,te5="rojo"] = te;

console.log(te4,te3,te5)

**

#### Operador rest y destructuring arrays:

Podemos combinar el poder del operador rest, con la sintaxis de desestructuración.

Sintaxis:

[element\_1,…,element\_N°,...nombreArgumento ] = [array\_elements]

EJEMPLO:

var a;

var b;

var c;

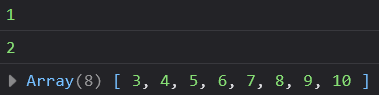
\*El operador **rest** siempre tiene que ser el último argumento que se escriba\*

[a,b,...c ] = [1,2,3,4,5,6,7,8,9,10]

console.log(a);

console.log(b);

console.log(c);



## Métodos:

Los métodos dentro de un objeto equivalen a las funcionalidades que el mismo puede tener definidas como propiedades.

Ósea, lo que puede hacer el objeto en sí mismo, es decir, que funcionalidades tiene implícitas para serle útil como tal.

Como **JS** es de tipado dinámico lo que hace que una propiedad pueda ser identificada como método es que su valor corresponda a una función.

Sintaxis:

*//CON FUNCIONES TRADICIONALES:*

*const* oject\_name = {method\_name: *function* (*parameters*) {return codigo} }

*//CON FUNCIONES FLECHA:*

*const* oject\_name = {method\_name: () *=>* {return codigo}}

\* las funciones flecha tienen ciertas limitaciones en objetos. \*

EJEMPLO:

con funciones tradicionales:

const saludos = {

  saludo\_1 : function (){return "hola"},

  saludo\_2 : function (){return "Que tal"},

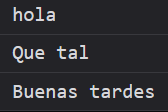
  saludo\_3 : function (){return "Buenas tardes"},

}

console.log(saludos.saludo\_1());

console.log(saludos.saludo\_2());

console.log(saludos.saludo\_3());



con funciones flecha:

const sumas = {

  sumaDos : (n1,n2) => n1+n2,

  sumaTres : (n1,n2,n3) => n1+n2+n3

}

console.log(sumas.sumaDos(2,3));

console.log(sumas.sumaTres(3,6,9));



### Objects value:

Es un método propio de todos los objetos y nos devuelve las propiedades numerables de un objeto dentro de un array.

Sintaxis:

Object.values(objectName));

EJEMPLO:

const reinosSeresVivos = {

  1: "Animal",

  2: "Vegetal",

  3: "Fungi",

  4: "Protocista",

  5: "monera",

}

console.log(Object.values(reinosSeresVivos));

const nombresApellidos = {

  "Manuel":"Garcia",

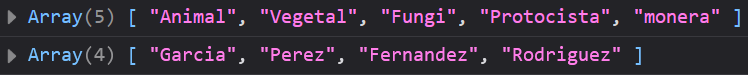
  "Pedro":"Perez",

  "Jorje":"Fernandez",

  "Nahuel":"Rodriguez",

}

console.log(Object.values(nombresApellidos));



Tenemos una única limitación aquí, es que no podemos usar métodos que ocupen las mismas propiedades del objeto sin la palabra reservada *THIS.*

### this:

Si quisiéramos ocupar la **key** de un objeto dentro del mismo objeto debemos usar el método ***this***

\*\***no** se puede operar con **arrow functions** en un objeto al hacerse referencia a si mismo\*\*

Es importante que usemos *funciones tradicionales* para trabajar con el metodo ***this*** y no *arrow functions* ya que al usar *arrow functions* el método ***this*** hace referencia al objeto ***window*** y no al objeto deseado.

Sintaxis:

const object\_name = {

  keyName1: "value1",

  keyNameN°: "valueN°",

  method\_name : function () {return this.keyNameN°}

}

EJEMPLO:

ejemplo 1:

Supongamos este objeto:

const saludos = {

  saludo1: "Hola",

  saludo2: "Buenas",

  saludo3: "Tardes",

}

Si quisiéramos ocupar los valores de sus keys podríamos hacer esto:

const saludos = {

  saludo1: "Hola",

  saludo2: "Buenas",

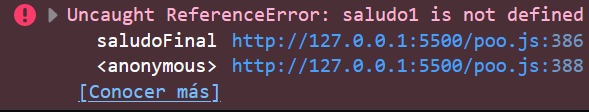
  saludo3: "Tardes",

  saludoFinal : function () {return `${saludo1} ${saludo2} ${saludo3} `}

}

console.log(saludos.saludoFinal());

a simple paracer debería funcionar…

**

pero esto no ocurre, sin embargo, al usar el metodo this…

const saludos = {

  saludo1: "Hola",

  saludo2: "Buenas",

  saludo3: "Tardes",

  saludoFinal : function () {return `${this.saludo1} ${this.saludo2} ${this.saludo3} `}

}

console.log(saludos.saludoFinal());

**

Probémoslo con una función flecha:

const saludos = {

  saludo1: "Hola",

  saludo2: "Buenas",

  saludo3: "Tardes",

  saludoFinal :  ()=> {return `${this.saludo1} ${this.saludo2} ${this.saludo3} `}

}

console.log(saludos.saludoFinal());



Ocurre lo advertido anteriormente

ejemplo 2:

const operaciones = {

  1: 2+2,

  2: 3+3,

  3: 4+4,

  calculo: function (operacion) {return this[operacion] }

}

console.log(operaciones.calculo(1));

console.log(operaciones.calculo(2));

console.log(operaciones.calculo(3));



## Clases:

La clase de un objeto se pueden entender como el “grupo" o “tipo” a donde pertenece este.

Por ejemplo: auto, humano, taza, celular; podrían ser clases.

A partir de una clase podemos crear varios objetos.

La creación de la clase pasa por:

Definir sus propiedades/atributos y valores, Construirla y finalmente Instanciarla.

Gracias a esto, varios objetos adquieren un sentido semántico a la vez que se puede establecer una relación más practica entre objetos debido a la *herencia*.

Las clases pueden tener **atributos** o métodos ***estáticos*** que forman parte de la propia clase y no de algún objeto en cuestión.

La sintaxis de las clases es similar a la de los objetos solo que tiene una ***palabra reservada***:

### Constructor:

Es una función obligatoria que construye las propiedades del objeto y es parte esencial de las clases.

Todas las clases tienen un constructor, aunque no este explicito en el código.

\*\*\*Cada clase puede tener solo UN constructor. \*\*\*

Lo primero que se ejecuta al crear un objeto es el constructor.

Dicho constructor puede recibir N parámetros y también tener dentro N duplas key:value

Sintaxis:

class Class\_name {

  constructor(parámetro\_1,…, parámetro\_n){

  CODIGO

  }

}

Por convención se suele escribir el nombre de las clases empezando con una mayúscula.

EJEMPLO:

ejemplo 1:

\*\*\* Los valores y los parámetros deben llevar el mismo nombre. \*\*\*

class humano {

  constructor(nombre,apellido){

      this.nombre = nombre;

      this.apellido = apellido;

  }

}

ejemplo 2:

class humano {

  constructor(nombr,apel){

      this.nombre = nombr;

      this.apellido = apel;

  }

}

ejemplo 3:

class Ukelele {

  constructor(color,modelo){

    this.color = color;

    this.modelo = modelo;

  }

console.log(Ukelele);



### Instanciación:

Luego de haber escrito una clase con sus características deseadas, podemos crear un objeto que pertenezca a la misma, el objeto creado a partir de esta clase heredara las características de la clase.

De esta forma estamos instanciando una clase, y también creando un objeto o varios.

Sintaxis:

const object\_name = new class\_name (parameters)

EJEMPLO:

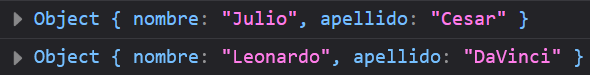
Recordando las clases anteriores:

const emperador = new Humano("Julio","Cesar");

console.log(emperador);

const genio = new Humano("Leonardo","DaVinci");

console.log(genio);

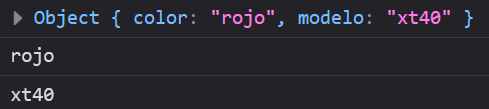


primerukelele =  new Ukelele("rojo", "xt40");

console.log(primerukelele);

console.log(primerukelele.color);

console.log(primerukelele.modelo);



#### Vulnerabilidad de objetos en JS:

Una consideración a tener en cuenta en **JS** es su libertad, ya que nosotros podemos modificar las propiedades, valores y métodos de un objeto a gusto siendo ideal que estos preserven su contenido.

Otros lenguajes de programación tienen una forma diferente de trabajar con objetos donde se pueden establecer propiedades y métodos privados o públicos siendo una forma segura de operar con objetos.

EJEMPLO:

const objetoGenerico = {

  propiedad1: "valor1",

  metodo1: (n1,n2)=>{return n1+n2},

  metodo2: (frase)=>{return `Dijiste: ${frase}`},

}

console.log(objetoGenerico);

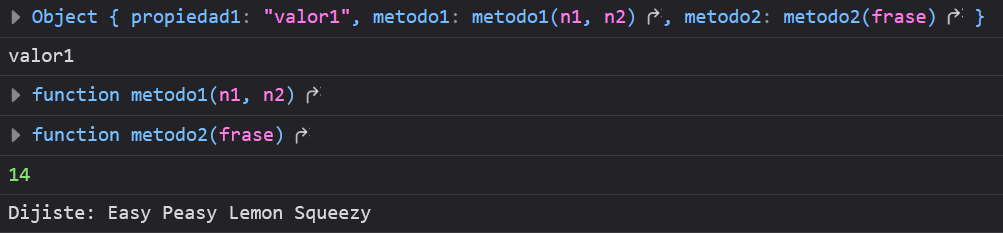
console.log(objetoGenerico.propiedad1);

console.log(objetoGenerico.metodo1);

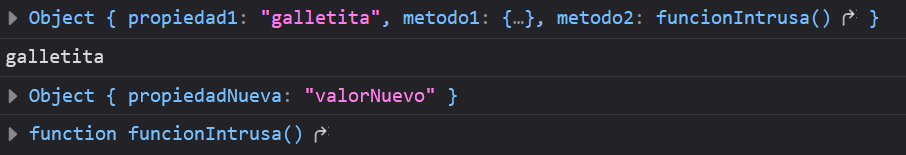
console.log(objetoGenerico.metodo2);

console.log(objetoGenerico.metodo1(9,5));

console.log(objetoGenerico.metodo2("Easy Peasy Lemon Squeezy"));



Hasta aquí todo bien, pero, intentemos alterar el objeto.



Como se ve, es posible modificar prácticamente todo el objeto, y no solo sus valores sino tambien sus tipos de dato, una posible solución seria “frizar” el objeto.

Object.freeze(objetoGenerico);

objetoGenerico.propiedad1 = "galletita"

objetoGenerico.metodo1 = {propiedadNueva: "valorNuevo"}

objetoGenerico.metodo2 = function funcionIntrusa(){}

console.log(objetoGenerico);

console.log(objetoGenerico.propiedad1);

console.log(objetoGenerico.metodo1);

console.log(objetoGenerico.metodo2);



como vemos el objeto preserva su estado inicial, pero ahora también es inmutable para nosotros mismos.

Para poder evitar mal entendidos en JS existe una serie de medidas para implementar en los objetos estas indican a los otros programadores que lean el código que algo es privado o público, veamos:

### GETTERS Y SETTERS:

Los Getters y Setters son funciones cuya ejecución se anticipa a una propiedad en particular en la que los usemos.

***GET: OBTENER***, retorna algo

***SET: ESTABLECER***, Permite darle un nuevo valor a algo

**GET:**

cuando llamemos propiedades de un objeto se invocará antes al **get**

**SET:**

cuando definamos o redefinamos una propiedad se invocará antes al **set**.

**TIP:**

Por convención se suele agregar un guion bajo “\_” antes del nombre de propiedad que debe ser considerada como privada. Esto es conocido entre la comunidad de programadores de JS que se han dignado a estudiar

Si bien en JS no existen las variables privadas, (porque las mismas se pueden modificar de forzosa e intencionalmente) al menos así en el ámbito de la programación, se entiende que los atributos que tengan “\_” previamente a su nombre deber ser considerados privados y por esto, no se debe alterar su valor desde afuera.

Sintaxis:

get propertyName(parameters){CODIGO}

set propertyName(parameters){CODIGO}

EJEMPLO:

Ejemplo 1:

Entendiendo el funcionamiento…

class humano {

  constructor(nombre, edad) {

    this.\_nombre = nombre;

    this.\_edad = edad;

  }

  get nombre() {

    console.log("llamaste al getter");

  }

  set nombre(w) {

    console.log("llamaste al setter");

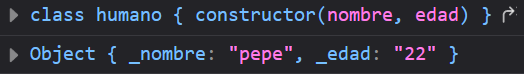
  }

}

console.log(humano);

humano1 = new humano("pepe", "22"); ///Creamos el humano con su clase

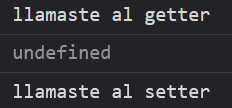
console.log(humano1);

**

Vemoas que pasa cuando intentamos obtener o modificar la propiedad que tienen get y set

console.log(humano1.nombre);

humano1.nombre = "algo";



*Conclusión:*

Cuando intentemos una acción de obtención o modificación de la propiedad, primero se invocaran estas funciones. Porque es importante? , porque de este modo nosotros podemos garantizar que si alguien quiere obtener o modificar estos valores vamos a poder realizar acciones deseadas como: arrojar un error diciendole, no puede modificar este valor, o un mensaje , etc… más allá de que pueda igualmente alterar el valor.

Ejemplo 2:

Un caso práctico podría ser el siguiente

class humano {

  constructor(nombre, edad) {

    this.\_nombre = nombre;

    this.\_edad = edad;

  }

  get nombre() {return `El nombre del humano es: ${this.\_nombre}`}

  set nombre(nuevoNombre) {this.\_nombre =nuevoNombre}

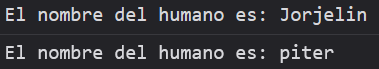
}

const humanoNuevo = new humano("Jorjelin","99 años");

console.log(humanoNuevo.nombre);

humanoNuevo.nombre = "piter";

console.log(humanoNuevo.nombre);



Parece que usamos una asignación vulgar para las propiedades del objeto, pero de fondo se llamaron a las funciones get y set.

Cuando usamos getters y setters podemos omitir alguno de ellos y se va a convertir en una propiedad de “read only” o “write only”

### Variables estáticas:

Sirven para que una característica este siempre presente en nuestro objeto, ya sea un método o un atributo en particular.

Sintaxis:

static name= CODIGO;

EJEMPLO:

class animal {

    constructor(nombre,especie){

        this.nombre = nombre;

        this.especie = especie;

    }

    static caracteristica = "organismo pluricelular";

    static saludo= ()=> {return "Buongiorno!"}

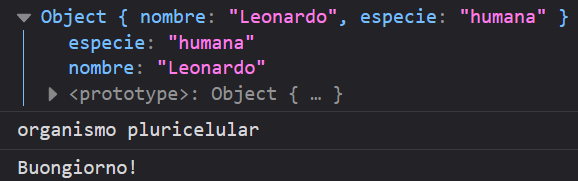
}

const leo = new animal("Leonardo","humana");

console.log(leo);

console.log(animal.caracteristica)

console.log(animal.saludo());



Notese que la variable y la funcion está ejecutándose desde la CLASE y no desde el objeto.

Además, estas no aparecen en el objeto.

De esta forma nosotros no tenemos que instanciar la clase previamente; por defecto contiene esta característica.

Demas esta decir que una variable estática tiene sentido cuando una clase realmente necesita una.

# Herencia:

La herencia de propiedades o clases, se da cuando necesitamos extraer solo determinadas partes de algo sin adquirir todas sus características.

En **JS** solo se puede heredar una sola clase, si quisiéramos heredar varias deberíamos hacer que **una** clase herede a **otra** y la **ultima** a la **ante ultima.**

\*Todos los objetos en **JS** heredan la clase OBJECT por defecto.\*

### EXTENDS:

Sirve para extender las funcionalidades y características de una clase a otra.

Sintaxis:

class clasName1 extends clasName2 {CODIGO};

clasName1: es la clase que se manda a clasName2.

clasName2: Es la clase hereda de clasName1.

EJEMPLO:

Ejemplo 1:

class fisica{

  constructor(espacio, tiempo){

      this.espacio = espacio;

      this.tiempo = tiempo;

  }

}

class materia extends fisica{}

const atomo = new materia("cuantico","atemporal");

console.log(atomo);



como vemos el objeto atomo tiene las propiedades espacio y tiempo que son heredadas desde la clase fisica a la clase materia

Ejemplo 2:

class animal{

  constructor(nombre, especie){

      this.nombre = nombre;

      this.especie = especie;

  }

  mostrarEspecie(){return this.especie;}

}

class humano extends animal{};

const animal\_1 = new animal("Cabra","Mamifero");

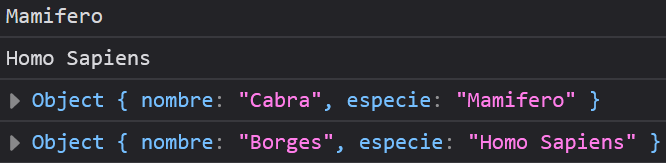
const humano\_1 = new humano("Borges","Homo Sapiens");

console.log(animal\_1.mostrarEspecie());

console.log(humano\_1.mostrarEspecie());

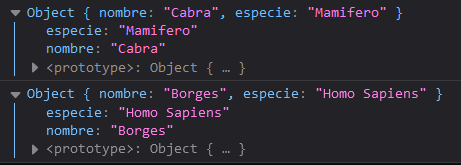
console.log(animal\_1);

console.log(humano\_1);

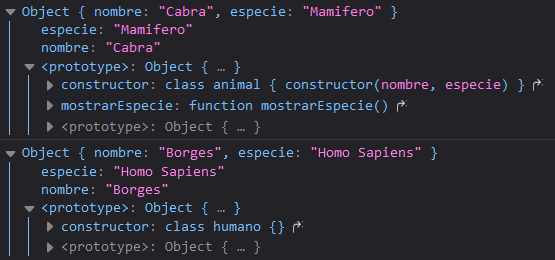


Como se puede notar ambos objetos tienen el método mostrarEspecie incluido, siendo que la clase animal lo tiene por definición, y la clase humano lo tiene por extensión.

Quizás lo mas importante a destacar es lo siguiente:



Hasta aquí el contenido de los objetos es el mismo, pero ampliemos para ver la diferencia:



\*Nótese la diferencia de niveles que hay entre la primera clase y la segunda\*

Esto se debe a cómo funcionan los objetos en JS, que es mediante prototipos.

Es un tema un poco más complejo de entender y es necesario saber que lo que vamos a usar aquí es un AZUCAR SINTACTICO para lograr lo que en realidad lograría la herencia por prototipos en JS.

“Los prototipos son un mecanismo mediante el cual los objetos en JavaScript heredan características entre sí”

Todos los objetos en JS son prototipados excepto los datos primitivos.

**Para mas información consultar:**

[**https://developer.mozilla.org/es/docs/Learn/JavaScript/Objects/Object\_prototypes#:~:text=Los%20prototipos%20son%20un%20mecanismo,m%C3%A9todos%20a%20los%20contructores%20existentes**](https://developer.mozilla.org/es/docs/Learn/JavaScript/Objects/Object_prototypes#:~:text=Los%20prototipos%20son%20un%20mecanismo,m%C3%A9todos%20a%20los%20contructores%20existentes)

### SUPER:

Para poder utilizar el constructor de la clase padre de la que heredamos necesitamos llamar al SUPER CONSTRUCTOR.

La palabra clave ***super*** es usada para acceder y llamar atributos o métodos de una clase padre de un objeto.

Sintaxis:

super(parameter\_1,…,parameter\_N°);

*EJEMPLO:*

*SUPER CON PROPIEDADES:*

Ejemplo 1:

Supongamos la siguiente clase:

class SerVivo {

  constructor(dominio,reino,genero,especie){

    this.dominio = dominio;

    this.reino = reino;

    this.genero = genero;

    this.especie = especie;

  }

}

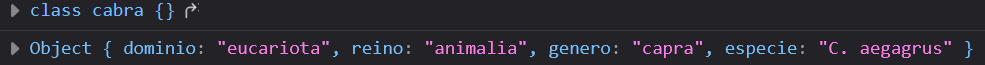
**desglose 1:**

class cabra extends SerVivo{};

console.log(cabra);

const Capra\_falconeri = new cabra("eucariota","animalia","capra","C. aegagrus");

console.log(Capra\_falconeri);



Hasta aquí todo bien, vemos que podemos usar todas las propiedades adquiridas.

Pero… ¿Qué pasaría si quisiéramos además de las propiedades adquiridas agregarle una que sea propia de nuestra nueva clase?

**desglose 2:**

class cabra extends SerVivo{

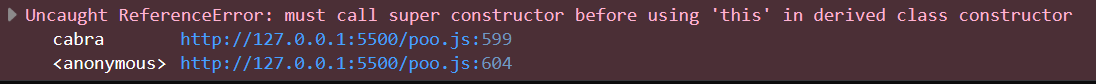
  constructor(colorPelaje){

    this.colorPelaje = colorPelaje;

  }

};

const Capra\_falconeri = new cabra();



Veriamos este mensaje, aquí es donde entra super.

Para poder agregar nuevas propiedades ADEMAS de las propiedades heredadas, necesitamos invocar a este SUPER constructor.

Dentro de nuestro constructor tenemos que invocar a super, y dentro de super, pasarle como argumento todas las propiedades que queremos heredar de nuestra clase padre.

**desglose 3:**

class cabra extends SerVivo{

  constructor(colorPelaje){

    super(dominio,reino,genero,especie)

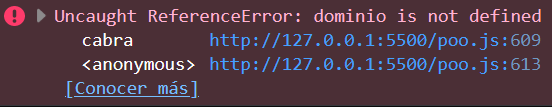
    this.colorPelaje = colorPelaje;

  }

};

const Capra\_falconeri = new cabra();

pero aquí surge un nuevo problema y este es el ultimo 😊



Y es que dentro de nuestra clase nueva, no estamos construyendo lo que recibimos en super.

Así que la solución final es esta:

**desglose 4:**

class cabra extends SerVivo{

  constructor(colorPelaje,dominio,reino,genero,especie){

    super(dominio,reino,genero,especie)

    this.colorPelaje = colorPelaje;

  }

};

const Capra\_falconeri = new cabra();

console.log(Capra\_falconeri);



Y ahora si tenemos nuestra clase , que tiene todas las propiedades heredadas de la clase padre MAS una propiedad nueva que es propia de esa clase nada mas ♥

Y con esta forma de heredar propiedes no tenemos que volver a definirlas con this

Ejemplo 2:

class humano extends SerVivo{

  constructor(nombre,ocupacion,dominio,reino,genero,especie){

    super(reino,dominio,genero,especie);

    this.nombre = nombre;

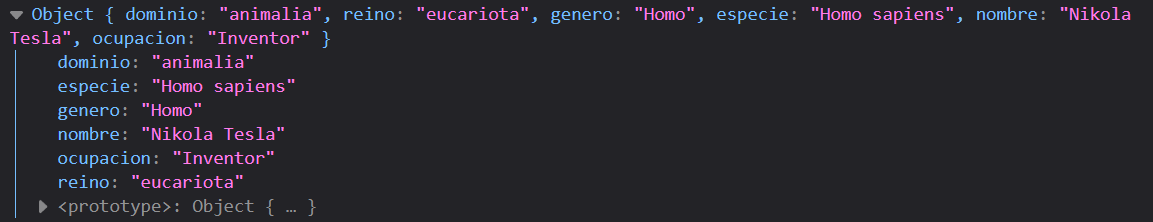
    this.ocupacion = ocupacion;

  }

};

const Tesla = new humano("Nikola Tesla","Inventor","eucariota","animalia","Homo" ,"Homo sapiens");

console.log(Tesla);



*SUPER CON METODOS:*

supongamos la siguiente clase con 3 propiedades y 3 métodos

class presentacion{

  constructor(nombre,oficio,meta){

    this.nombre = nombre;

    this.oficio = oficio;

    this.meta = meta;

  }

  presentacion1(){console.log(`Me llamo ${this.nombre}`);}

  presentacion2(){console.log(`Mi oficio  es: ${this.oficio}`);}

  presentacion3(){console.log(`Mi meta  es: ${this.meta}`);}

}

Ahora heredemos sus propiedades con esta otra clase y además creemos un método propio de esta clase

class humano extends presentacion{

  constructor(nombre,oficio,meta,edad){

    super(nombre,oficio,meta)

    this.edad = edad

  }

  decirEdad(){console.log(`Mi edad  es: ${this.edad}`);}

}

const pepe  = new humano("pepe","verdulero","Crear una verdura","86 años");

pepe.decirEdad();

console.log(pepe.nombre);

console.log(pepe.oficio);

console.log(pepe.meta);

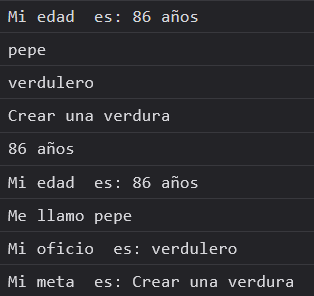
console.log(pepe.edad);

pepe.decirEdad()

pepe.presentacion1();

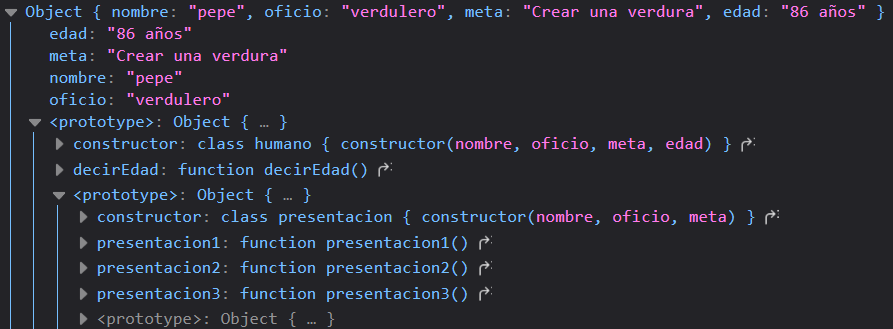
pepe.presentacion2();

pepe.presentacion3();

**

Como se puede observar funcionan todos los métodos de ambas clases.

pero tenemos que tener en cuenta que los métodos presentación 1,2 y 3 no son propios de la clase humano ni del objeto pepe

**

, sino que vienen por defecto arrastrados en al cadena de prototipos. si nosotros quisieramos que los metodos sean propios de esta clase, deberiamos llamar tambien al super en los metodos, de la siguiente manera:

class humano extends presentacion{

  constructor(nombre,oficio,meta,edad){

    super(nombre,oficio,meta)

    this.edad = edad

  }

  presentacion1(){

    super.presentacion1()

  }

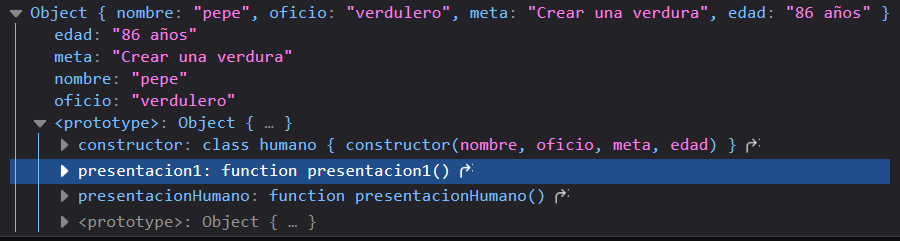
  presentacionHumano(){console.log(`Mi edad  es: ${this.edad}`);}

}

const pepe  = new humano("pepe","verdulero","Crear una verdura","86 años");

console.log(pepe);

y ahora si…

**

solo el método presentacion1 es oficialmente parte de la clase humano

### INSTANCE OF:

Es un operador que nos permite saber a ***qué clase pertenece un objeto***, recordemos que en **JS** todo (excepto los tipos de datos primitivos) es un objeto y como tales todos heredan propiedades y métodos de clases superiores, puntualmente heredan prototipos de la cadena de prototipos.

Lo que corrobora ***instanceof***; se puede entender con la pregunta: ¿Es “tal objeto” parte de “tal clase”?

Sintaxis**:**

objectName instanceof className

EJEMPLO:

Ejemplo 1:

class continente{}

class color{}

class liquido{}

const america = new continente;

const amarillo = new color;

const jugo = new liquido;

console.log(america instanceof continente);

console.log(amarillo instanceof color);

console.log(jugo instanceof liquido);



Ejemplo 2:

const cualquierObjeto\_1  = []

const cualquierObjeto\_2  = function name(params) {}

const cualquierObjeto\_3  = ()=>{}

const cualquierObjeto\_4  = {}

console.log(cualquierObjeto\_1 instanceof Object);

console.log(cualquierObjeto\_2 instanceof Object);

console.log(cualquierObjeto\_3 instanceof Object);

console.log(cualquierObjeto\_4 instanceof Object);



Ejemplo 3:

var cualquiercosa;

console.log(cualquiercosa instanceof Object);



Aquí podemos ver que no todo es un objeto en JS

# PROMESAS:

Son objetos de JavaScript que representan un resultado futuro.

Para divertirnos un poco podríamos comprar el funcionamiento de las promesas con el comportamiento de los meeseeks.

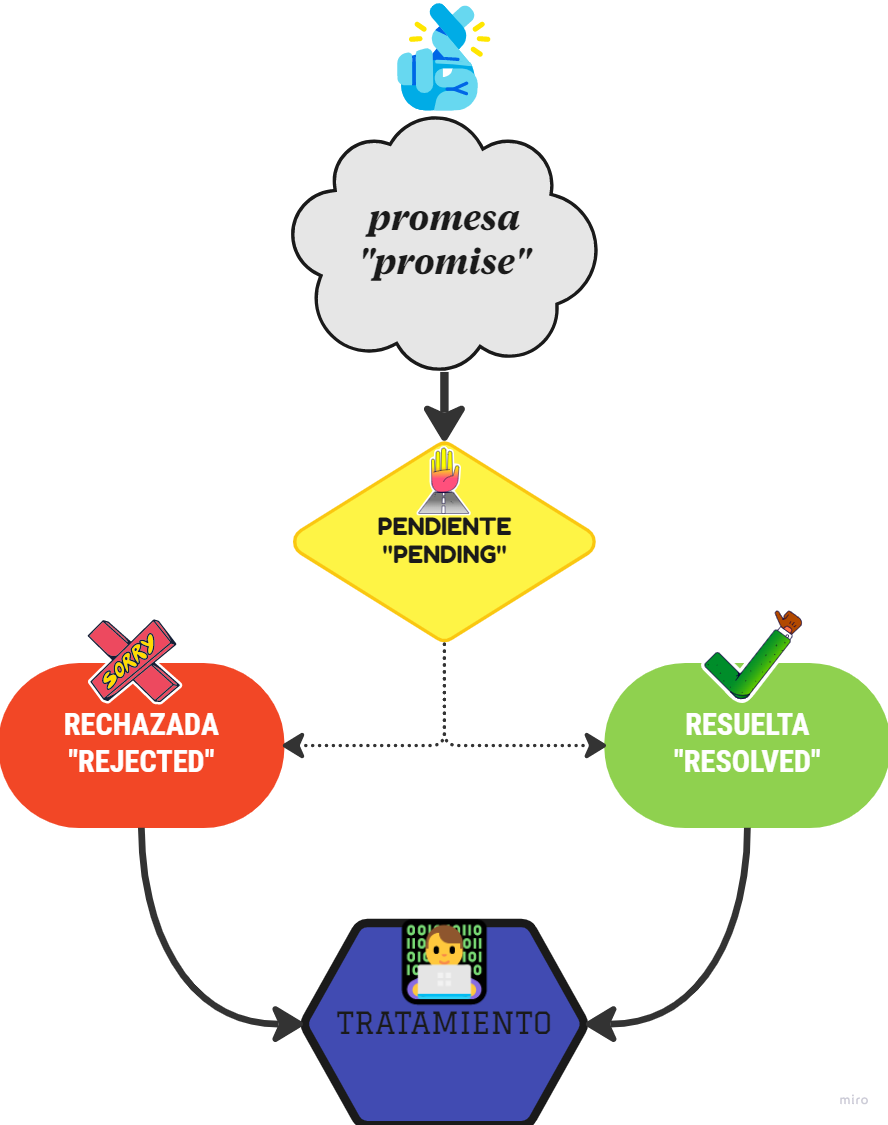
una analogía un poco exótica pero bastante certera.



Los Meeseeks son criaturas que se crean para servir a un propósito singular para el que se esforzarán por cumplir. Después de que cumplen su propósito, expiran y desaparecen en el aire.

Las promesas generalmente vienen anticipadas por una función, la cual debería tener un propósito singular, dependiendo del resultado de la función, la promesa va poder “expirar” o no.

*Los tres Estados de una promesa:*



Una promesa puede quedar ***pendiente***, ***cumplirse*** o ***no cumplirse***

En caso de quedar pendiente, simplemente no pasar nada…

Sería como cuando los meeseeks no pueden colaborar con jerry y quedan atrapados en la existencia



Pero una vez que la promesa adquiere alguno de los otros dos estados; osea si se cumplio o no (lo que se conoce como **resolve** y **reject**) se da un tratamiento especifico. Para determinar que hacer a partir del resultado de una promesa.

Veremos dos formas de dar este tratamiento:

Con el metodo **then**.

Con bloques **try-catch-finally**

Primero jugemos un poco con el contenido de una promesa en **JS**.

Sintaxis:

// sintaxis 1:

const promesaName = new Promise(() => {});

// sintaxis 2:

const promesaName = new Promise(function nameFunction(parameters){});

*EJEMPLO:*

***JavaScript:***

const promesa\_1 = new Promise(() => {});

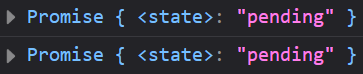
const promesa\_2 = new Promise(function nameFunction(parameters){});

console.log(promesa\_1);

console.log(promesa\_2);

funciona tanto con funciones tradicionales y funciones flecha

Esto es lo que vemos por consola:



Como se puede observar, las promesas son un tipo de objeto puntual dentro de JS.

Las mismas tienen un estado:

Claramente en este caso están en un estado pendiente eterno, ya que dentro no hay instrucciones que puedan generar un cambio.

Las promesas como objetos en JavaScript reciben dos funciones callbacks adjuntas.

Estas funciones callbacks, son los parametros que puede recibir cada promesa y van en orden: **1° el éxito(resolve)** y **2° el rechazo(reject).**

Para explayar mejor este funcionamiento veamos un ejemplo más didáctico de ver y leer:

const promesa = new Promise((exito, rechazo) => {

  const estado = true;

  if(estado){exito("La promesa fue exitosa ");}

  else{rechazo("La promesa fue rechazada");}

});

console.log(promesa);



…Si cambiamos el valor de estado…

  const estado = false;

veremos esto:



Donde empieza a haber un comportamiento un poco más serio en caso de que una promesa “falle”.

Aquí hay dos cosas que importan mucho, y es que más allá del estado final de la promesa, ósea:

Si es cumplida-resolve-fulfilled

O

Si es rechazada-reject-rejected

Cuando se cumple, devuelve el valor (value)



Pero cuando no, muestra la razón. (reason)



Esto es importante porque es propio de la estructura misma de las promesas en J S.

Es decir, Nosotros como programadores, “DEBERIAMOS” devolver un valor cuando se cumple la promesa

Y “DEBERIAMOS” devolver el error en caso de que no se cumpla.

¿Por qué? No hay porque, en tanto a obligación, ya que JS nos da muchísima libertad.

Tranquilamente podríamos pasar cualquier cosa en vez de un error, o poner un valor indefinido.

Pero si hay una obligación en tanto a buenas prácticas, ya que como programadores deberíamos de tener un indicio de porque esta fallando una promesa y este indicio podría ser un error bien pensado y tratado, y en caso de que se cumpla deberíamos obtener un valor para poder continuar con la ejecución, de lo contrario,

¿Para qué querríamos usar promesas no?

Veamos este comportamiento con más detalle:

const promesa = new Promise((exito, rechazo) => {

  const estado = true;

  if(estado){exito()}

  else{rechazo()}

});

console.log(promesa);



Si cambiamos el valor de estado…

const estado = false;



Vemos lo de antes, pero con la diferencia de que ahora, <value> y <reason> están indefinidos.

Okey, entonces podríamos poner dentro de estos, algo que se devuelva

en caso de que la promesa se cumpla o no.

Probemos pasarle alguno de estos parámetros:

\*\*\*

function funcion() {console.log("Soy una funcion")}

const objeto = {

1: "valor1",

2: "valor2",

3: "valor3",

}

var numero = 237;

let palabra = "Anacoreta";

\*\*\*

Tenemos: una función, un objeto, y dos variables:

empecemos con la función tradicional:

const promesa = new Promise((exito, rechazo) => {

  if(estado){exito(funcion)}

  else{rechazo(funcion)}

});

console.log(promesa);

si const estado = true;



si const estado = false;



Y ambas apuntan aca:



Probemos ahora con una función flecha:

const promesa = new Promise((exito, rechazo) => {

  if(estado){exito(()=>{console.log("algo1");})}

  else{rechazo(()=>{console.log("algo2");})}

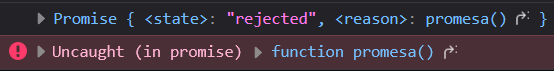
});

console.log(promesa);

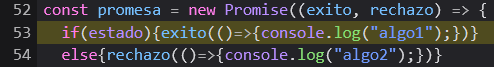
si const estado = true;

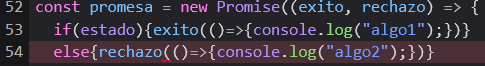


si const estado = false;



Y ambas apuntan a las funciones dentro de la promesa misma:





Probemos ahora con el objeto:

const promesa = new Promise((exito, rechazo) => {

  if(estado){exito(objeto)}

  else{rechazo(objeto)}

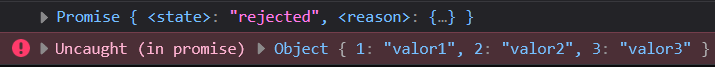
});

console.log(promesa);

si const estado = true;



si const estado = false;



Probemos ahora con el la primer variable que contiene un numero:

const promesa = new Promise((exito, rechazo) => {

  if(estado){exito(numero)}

  else{rechazo(numero)}

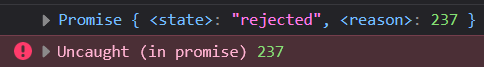
});

console.log(promesa);

si const estado = true;



si const estado = false;



Probemos ahora con la segunda variable que contiene un string:

const promesa = new Promise((exito, rechazo) => {

  if(estado){exito(palabra)}

  else{rechazo(palabra)}

});

console.log(promesa);

si const estado = true;



si const estado = false;



Okey, muchos ejemplos… es suficiente, resumen y conclusión:

Las promesas representan un posible valor o no.

Las promesas reciben como parámetros dos funciones callbacks adjuntas

1° cumplida/resolved/fulfilled como quieras llamarle

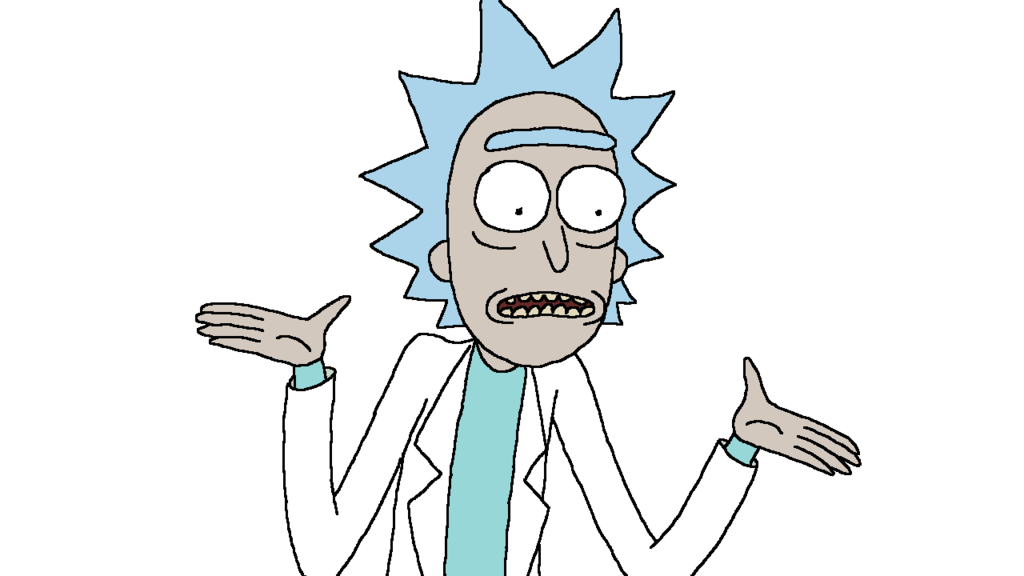
2° fallida/rechazada/rejected como quieras llamarle.

Si la promesa es resolved devuelve un value

Si la promesa es rejected devuelve un error.

¡!!y es responsabilidad puramente nuestra!!!

¡!!!especificar qué error va a devolver, como porque cuando en donde y con quien!!!!



Todo lo visto anteriormente es una práctica horrorosamente desastrosa y solo se usó el código con fines didácticos.

Ahora sí, mas allá de todo lo visto anteriormente, existen formas específicas de tratar una promesa, que ayudan a que sea más legible y que mejoran las practicas con las mismas.

Veámoslas.

## Then:

EL método then devuelve una promesa, por lo tanto, puede recibir también dos funciones callbacks,

Una se ejecuta si la promesa se cumple

La otra se ejecuta si la promesa no se cumple

Tiene dos formas de escribirse, una con una sintaxis bastante rebuscada, y la otra un poco más entendible, revisémoslas simplemente por curiosidad.

Sintaxis:

promesaName.then(function\_Sucess,function\_Failure);

EJEMPLO:

***JavaScript:***

SINTAXIS AMABLE:

FORMA 1:

Con las funciones afuera

function seCumplio() {console.log("Me cumpli!");  }

function noSeCumplio() {console.log("No me cumpli!");  }

const promesa = new Promise((resolve, reject) => {

    if (estado) {resolve()}

    else{reject()}

});

promesa.then(seCumplio,noSeCumplio);

también podemos escribirlo así:

promesa

  .then(seCumplio,noSeCumplio);

const estado = true;



const estado = false;



FORMA 2:

Con las funciones adentro:

const promesa = new Promise((resolve, reject) => {

    if (estado) {resolve('Esto se muestra si la promesa se cumple');}

    else{reject("Esto se muestra si la promesa no se cumple");}

});

promesa.then(value =>{console.log(value)} , er => { console.log(er)});

const estado = true;



const estado = false;



FORMA 3:

Un then para cada función callback:

esto se conoce como encadenamiento de then:

const promesa = new Promise((resolve, reject) => {

    if (estado) {resolve('Esto se muestra si la promesa se cumple');}

    else{reject("Esto se muestra si la promesa no se cumple");}

});

promesa

  .then(value => { console.log(value) })

  .then(er => { console.log(er) })

const estado = true;



const estado = false;



Aquí podemos observar una diferencia importante entre las formas 1 y 2, ya que el error es tratado de una forma puntual en este caso.

SINTAXIS MAS REBUSCADA:

const promesa = new Promise((resolve, reject) => {

    if (estado) {resolve(‘Esto se muestra si la promesa se cumple’);}

    else{reject(“Esto se muestra si la promesa no se cumple”);}

});

const estado = true;

FORMA 1:

promesa.then(value=>{console.log(value);},[null,er=>{console.log(er);}]);

FORMA 2:

promesa.then(value=>{console.log(value);},[undefined,er=>{console.log(er);}]);

FORMA 3:

promesa.then(value=>{console.log(value);},[0,er=>{console.log(er);}]);

FORMA 4:

promesa.then(value=>{console.log(value);},[NaN,er=>{console.log(er);}]);

FORMA 5:

promesa.then(value=>{console.log(value);},[false,er=>{console.log(er);}]);

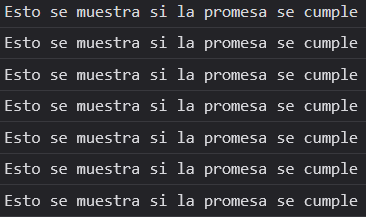
FORMA 6:

promesa.then(value=>{console.log(value);},["",er=>{console.log(er);}]);

FORMA 7:

promesa.then(value=>{console.log(value);},[er=>{console.log(er);}]);

Cuando estado = true:



Cuando estado = false:



Lo interesante que tenemos que notar aquí, es que el segundo parámetro del metodo then tambien debería de recibir dos parámetros

y es que recordemos que then, devuelve una promesa

y como las promesas reciben dos parámetros en orden: 1° éxito 2° rechazo

Para tratar los errores es necesario que 1° no se haya cumplido

Lo que significa que la funcion 1° debe ser un valor FALSEY dentro de JS .

De echo si nosotros al segundo parámetro de then, lo sacamos de los corchetes, no nos mostrara un error:

Nos mostrara la instrucción de una función, en este caso un mensaje.

promesa.then(value=>{console.log(value);},er=>{console.log(er);});



Okey genial hasta aquí todo, ahora vamos a ir un paso mas, vimos por consola un error que empieza diciendo 

Lo que literalmente significa: “no atrapado”

Es decir, nosotros obtuvimos un error, pero no le dimos tratamiento, no lo “dirigimos”, no lo “manipulamos”, no lo “atrapamos”.

Para “atrapar” errores es que se usan los bloques try-catch-finally

A continuación, los veremos, pero antes veamos un poco más sobre errores y excepciones en JS.

## Promise All

Sirve para corroborar que se cumplan varias promesas.

Si se cumplen todas se resuelve y si no se cumple una sola entonces se rechaza devolviendo el error de la primera promesa que no se cumplió.

Sintaxis:

Promise.all([promesa1,…, promesaN°]).then(CODIGO);

*EJEMPLO:*

Ejemplo 1

const estado1 = true;

const estado2 = true;

const estado3 = true;

const promesa1 = new Promise((resolve, reject) => {

    if (estado1) {resolve('Esto se muestra si la promesa se cumple');}

    else{reject("Esto se muestra si la promesa no se cumple");}

});

const promesa2 = new Promise((resolve, reject) => {

    if (estado2) {resolve('Esto se muestra si la promesa se cumple');}

    else{reject("Esto se muestra si la promesa no se cumple");}

});

const promesa3 = new Promise((resolve, reject) => {

    if (estado3) {resolve('Esto se muestra si la promesa se cumple');}

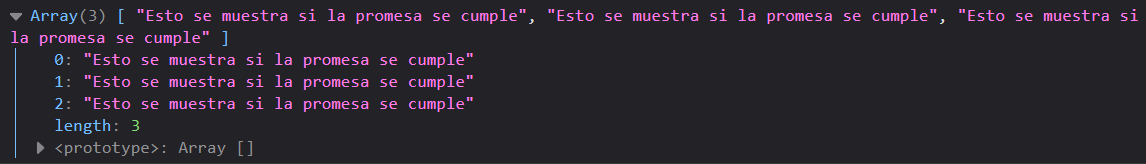
    else{reject("Esto se muestra si la promesa no se cumple");}

});

Promise.all([promesa1, promesa2, promesa3]).then(values => {

  console.log(values);

});



Ejemplo 2:

const estado1 = true;

const estado2 = false;

const estado3 = true;

const promesa1 = new Promise((resolve, reject) => {

  if (estado1) {resolve('Esto se muestra si la promesa se cumple');}

  else{reject("Esto se muestra si la promesa 1 no se cumple");}

});

const promesa2 = new Promise((resolve, reject) => {

  if (estado2) {resolve('Esto se muestra si la promesa se cumple');}

  else{reject("Esto se muestra si la promesa 2 no se cumple");}

});

const promesa3 = new Promise((resolve, reject) => {

  if (estado3) {resolve('Esto se muestra si la promesa se cumple');}

  else{reject("Esto se muestra si la promesa 3 no se cumple");}

});

Promise.all([promesa1, promesa2, promesa3]).then(values => {

console.log(values);

});



Como vemos se interrumpe automáticamente ni bien una promesa no se cumple.

## Errores y excepciones:

Son situaciones anómalas que se producen en nuestro código y que nos obligan a desviar la ejecución desde el flujo principal hacia un flujo alternativo.

Por ejemplo, cuando en la división de números se nos presenta un “0” “cero”, y tenemos que buscar un tratamiento especial para esa situación.

#### throw:

Permite retornar una excepción deseada.

Sintaxis:

throw expresion

EJEMPLO:

normalmente en una excepción suele devolver un mensaje de error

pero en JS se puede devolver cualquier objeto.

throw "ERROR SUPREMO"; 

throw 5;  

throw 2+2;  

throw function(){} ; 

throw true;  

throw 2<5;  

throw 5<2;  

throw []; 

Uncaught significa literalmente no atrapado, hace referencia al error.

\*Existen claramente buenas prácticas al momento de lanzar un error…\*

\*La excepción throw rompe con el flujo del programa. \*

console.log("antes");

throw "ERROR SUPREMO";

console.log("Despues");



console.log("Despues"); nunca se ejecutó.

De hecho el mismo editor de código lo reconoce y aplica una opacidad a esa línea de código…



Supongamos una situación donde tenemos que testear que un arreglo contenga solo impares.

Y mandar un mensaje de error cuando encuentre un numero par.

const impares = [1,3,5,7,9,11,12,55,77];

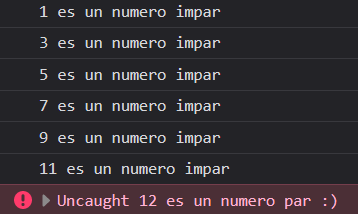
for (let i = 0; i < impares.length; i++) {

if ((impares[i]%2) === 0) {

    throw `${impares[i]} es un numero par :)`}

    else{console.log(`${impares[i]} es un numero impar`);}

}



Notemos que después del error se interrumpe el programa como se mencionaba antes.

Pero nosotros no siempre buscamos que se interrumpa todo el proceso, sino solo informar un error y continuar

para esto debemos darle un tratamiento especial a esta excepción

y aquí es donde entran dos bloques particulares:

#### Try y catch:

La declaración **try...catch** señala un bloque de instrucciones a intentar (**try**), y especifica una respuesta si se produce una excepción (**catch**).

Sintaxis:

try{CODIGO}

catch(error\_name) {CODIGO}

EJEMPLO:

function palabraApta(palabra) {

  try {

    if(palabra == "")  throw "No ingresaste nada";

    if(!isNaN(palabra)) throw "No ingresaste una palabra";

    if(palabra.length == 1)   throw "Ingresaste solo una letra";

    if(palabra.length > 15)  throw "Palabra demasiado larga";

    else{console.log(palabra + " me parece una muy buena palabra");}

  }

  catch(err) {console.log(`ERROR!! ${err} `)}

}

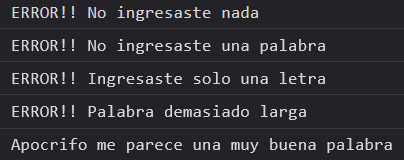
palabraApta("");

palabraApta(85);

palabraApta("z");

palabraApta("aasdasdasdasdassd");

palabraApta("Apocrifo");



#### Finally:

Permite dar un tratamiento final a nuestros bloques **try-catch**

Es la alternativa así captemos un error o no.

**Finally** se ejecuta SIEMPRE, más allá de que todo haya salido correctamente o no.

Sintaxis:

try{CODIGO}

catch(error\_name) {CODIGO}

finally {CODIGO}

EJEMPLO:

ejemplo 1:

function coloresPrimarios(color) {

  var colores = ["rojo","amarillo","azul"]

  try{

    if (colores.indexOf(color) === -1) {throw `${color} no es un color primario`}

    else{console.log(`${color} es un color primario`);}}

  catch(er){console.log(er);}

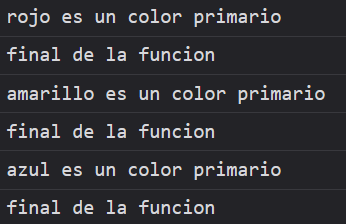
  finally{ console.log("final de la funcion");}

}

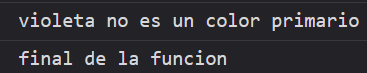
coloresPrimarios("rojo");

coloresPrimarios("amarillo");

coloresPrimarios("azul");



coloresPrimarios("violeta");



ejemplo 2:

También es posible realizar esto sin el catch

function coloresPrimarios(color) {

  var colores = ["rojo","amarillo","azul"]

  try{

    if (colores.indexOf(color) === -1) {console.log(`${color} no es un color primario`); }

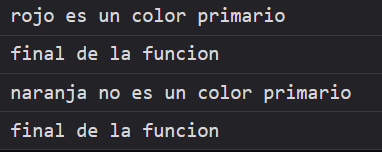
    else{console.log(`${color} es un color primario`);}}

  finally{ console.log("final de la funcion");}

}

coloresPrimarios("rojo");

coloresPrimarios("naranja");



En resumen los bloques:

TRY-CATCH-FINALLY

Nos permiten ejecutar códigos en los que no poseemos la certeza de que sean correctos siempre.

Las combinaciones posibles son:

Try-catch

Try-catch-finally

Try-finally

# ASINCRONISMO:

El diseño de software asíncrono amplía el concepto al construir un código que permite a un programa realizar una o varias tareas al mismo tiempo, sin detenerse a esperar que las tareas previas se haya completado.

Vamos a requerir que parte de nuestro código se ejecute de manera asíncrona cuando, no dependamos de el para seguir cargando todo el resto.

***Por ejemplo:***

Ingresamos a una página web donde podemos encargar comida y está por x razón tiene un juego interactivo.

No necesitamos que se termine de cargar el juego para cargar la página completa, podemos cargar la página, y que el juego cargue cuando pueda, de manera asíncrona, si termina de cargar antes o después va a depender de otros factores, como que tan pesado es el juego o la página; el punto es que su carga fue totalmente independiente.

## async:

Permite definir una función como asíncrona.

y que mejor manera de mostrar esto que con una función nativa de JS

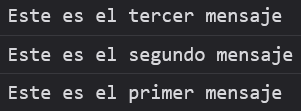
El método setTimeout() establece un temporizador que ejecuta una función o una porción de código después de que transcurre un tiempo establecido.

Veamos su funcionamiento:

setTimeout(() => {console.log("Este es el primer mensaje")}, 5000);

setTimeout(() => {console.log("Este es el segundo mensaje")}, 3000);

setTimeout(() => {console.log("Este es el tercer mensaje")}, 1000);



Como se puede observar si bien están escrito en un orden particular, se ejecutaron totalmente opuestos a como se esperaba; claro que esto fue intencional, pero cuando se trate de consultar información externa que no dependa de nosotros sino de la conexión a internet, la cantidad de información que queramos traer y demás, este retraso en mili segundos, va a ser real e impredecible, y por esto mismo debe tener un tratamiento particular.

Usaremos este temporizador para simular que nuestras funciones tardar un tiempo en procesar la información y así poder trabajar prácticamente con el concepto de asincronía.

## await:

Permite esperar a que se termine de ejecutar una función para poder pasar a la siguiente instrucción.

\*No hay await sin async\*

Async y await se usan en conjunto cuando trabajamos con promesas, no siempre son necesarias, pero suelen ser muy útiles cuando lo son.

Sintaxis:

Con funciones tradicionales:

async function functionName () {  {CODIGO} }

Con funciones flecha:

let/var/const/ functionName = async () {  {CODIGO} }

await functionName ();

*EJEMPLO:* respuesta true

supongamos que le preguntamos a alguien para que valla al cine con nosotros:

si la respuesta = true; significa que acepto la propuesta.

si la respuesta = false; significa que rechazo la propuesta.

const respuesta = true;

a continuación, tenemos lo que representaría el tiempo que tarda alguien en pensar la respuesta.

function pensandoRespuesta() {

  return new Promise ((resolve, reject) => {

    setTimeout(() => {

      if (respuesta) {resolve ('Acepto ir al cine con tigo.')}

      else {reject ("Lo siento! no han aceptado tu propuesta.")}

    }, 2000);

  });

}

a continuación, tenemos el veredicto final, donde sabremos la respuesta fue sí o no.

async function recibiendoRespuesta() {

  console.log('recibiendo la respuesta...');

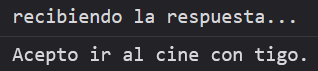
  const respuestaFinal = await pensandoRespuesta();

  console.log(respuestaFinal);

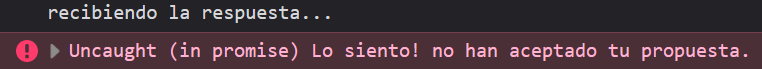
}

recibiendoRespuesta();

respuesta = true



respuesta = false



Lo que tenemos que sacar de aquí es que este await propone una espera, es decir, llamamos a la función recibiendoRespuesta() y dentro de esta hay un await que no va permitir continuar la ejecucion mientras este esperando la respuesta

# Orden de carga en scripts:

Como sabemos, es diferente si un script esta, al principio, a medio o al final de nuestro documento HTML.

También existen formas de garantizar que se carguen solo cuando termina de cargar el DOM o también de forma asíncrona, es decir, independientemente de que el DOM haya o no terminado de cargar.

#### script defer:

El atributo defer indica al navegador que no espere por el script. En lugar de ello, debe seguir procesando el HTML, construir el DOM. El script carga “en segundo plano” y se ejecuta cuando el DOM esta completo.

#### script async:

El atributo async vuelve al script completamente independiente:

El navegador no se bloquea con scripts async (como defer).

Otros scripts no esperan por scripts async, y scripts async no espera por ellos.

DOMContentLoaded y los scripts asincrónicos no se esperan entre sí:

DOMContentLoaded puede suceder antes que un script asincrónico (si un script asincrónico termina de cargar una vez la página está completa)

…o después de un script asincrónico (si tal script asincrónico es pequeño o está en cache)

En otras palabras, los scripts async cargan en segundo plano y se ejecutan cuando están listos. El DOM y otros scripts no esperan por ellos, y ellos no esperan por nada. Un script totalmente independiente que se ejecuta en cuanto se ha cargado. Tan simple como es posible, ¿cierto?

#### script dinámicos:

Hay otra manera importante de agregar un script a la página.

Podemos crear un script y agregarlo dinámicamente al documento usando JavaScript

Fuente:

<https://es.javascript.info/script-async-defer>

Y ya que mencionamos los scripts dinámicos hay otras formas de importar scripts a nuestro documento y esto es mediante módulos:

# MODULOS:

Para evitar que un código muy grande quede acumulado en un solo archivo o muy pocos archivos, es conveniente fragmentarlo, y para esto se utilizan los módulos.

Que es otra cosa que código que se pueden exportar e importar en otros archivos.

Al exportar elementos y crear un módulo de exportación, lo que realmente creamos es un objeto donde las propiedades son los nombres de los elementos (constantes, variables, funciones, clases...)

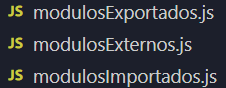
Es importante que usemos el atributo **TYPE** con el valor **MODULE** en nuestro script de **HTML** para evitar este error:



Simplemente así:

<script type="module" src="fileName.js"></script>

Para probar y explicar el funcionamiento de exportar e importar módulos utilizaremos estos archivos:



La etiqueta de nuestro script en el archivo HTML es la siguiente:

<script type="module" src="modulosImportados.js"></script>

### Export:

La palabra reservada **export** permite exportar uno o varios elementos, con su nombre propio o con un nombre agregado.

Esta misma tiene sentido en conjunto con la palabra ***import***  que se verá luego.

#### Exportación por elemento:

Permite exportar elementos declarados en la misma línea.

Sintaxis:

export let/var/const variableName= value;

*EJEMPLO:*

export const numero = 77;

export const texto = "Buenas!";

export const arregloFrutas = ["manzana","morron","zapallo"];

export const objetoPersona = {nombre:"value1",nombre2:"value2"};

export const funcion = () => "Que tal!";

export class humano { };

#### Exportación por variable:

Permite exportar una variable puntual.

Sintaxis:

export { variableName };

EJEMPLO:

const numero = 99;

const palabra1 = "improperio";

const palabra2 = "eufemismo";

const bienvenir = () => "Bievenido al mundo virtual";

const despedir = () => "Adi0s 1os 0 y 1 lo esperan pronto";

class humano { };

export { numero };

export { palabra1 , palabra2 };

export { bienvenir };

export { despedir };

export { humano };

#### Exportación renombrada:

Permite renombrar una variable para ser exportada.

Sintaxis:

export { variableNAme as variableNewName};

EJEMPLO:

const numero = 99;

const palabra1 = "improperio";

const palabra2 = "eufemismo";

const bienvenir = () => "Bievenido al mundo virtual";

const despedir = () => "Adi0s 1os 0 y 1 lo esperan pronto";

class humano { };

export { numero as numeroPerfecto };

export { palabra1 as insulto , palabra2 as alusion };

export { bienvenir as saludo1 };

export { despedir as saludo2 };

export { bienvenir, despedir as chau };

export { bienvenir as hola ,despedir  };

export { humano as homosapie };

#### Exportación multiple:

Permite exportar un grupo de variables en un solo bloque.

Sintaxis:

export {variableName1,…,variableNameN°};

EJEMPLO:

const numero = 99;

const palabra1 = "improperio";

const palabra2 = "eufemismo";

const bienvenir = () => "Bievenido al mundo virtual";

const despedir = () => "Adi0s 1os 0 y 1 lo esperan pronto";

class humano { };

export {

  numero,

  palabra1,palabra2,

  bienvenir as alo,

  despedir as byebye,

  humano

};

#### Exportación externa:

Permite exportar módulos de otro archivo JS diferente al que estamos trabajando actualmente.

Sintaxis:

Para todos los módulos del archivo:

export \* from "./relativePath.js";

para variables específicas del módulo:

export {variableName1,…,variableNameN°}; from "./relativePath.js";

EJEMPLO:

Exportar todo el contenido de un módulo externo.

Ejemplo 1:

export \* from "./modulosExternos.js";

Exportar variables específicas de un módulo externo.

Ejemplo 2:

export { n1,n2 } from "./modulosExternos.js";

Ejemplo 3:

export { n1, n2 as pepe } from "./modulosExternos.js";

Exportar todo el contenido de un módulo externo dentro de un objeto con un nombre especifico.

Ejemplo 4:

export \* as nuevoNombre from "./modulosExternos.js";

#### Exportación por default:

Permite realizar una exportación por default que tendrá una sintaxis diferente al importarse.

\*SOLO PUEDE HABER UNA EXPORTACION POR DEFAULT\*

Sintaxis:

Una sola variable o valor:

export default value;

Un objeto que contenga varios valores o variables:

export default {

  key1:"value1",

  key2:"value2",

  key3:"value3",

}

EJEMPLO:

Ejemplo 1:

export default "cadena de texto exportada por default";

Ejemplo 2:

export default {

  nombre: "pedro",

  edad: 23,

  ideosincracia: "nihilista",

};

Ejemplo 3:

let agamenon = {

  hito:"heroe",

  mitologia:"griega",

  creador:"homero",

}

export default agamenon;

### Import:

La palabra reservada **import** permite importar uno o varios elementos, con su nombre propio o con un nombre agregado.

Esta misma tiene sentido en conjunto con la palabra ***export*** vista antes.

\*SE RECOMIENDA IMPORTAR AL PRINCIPIO DE NUESTRO CODIGO JS\*

\*NO PUEDE HACER USO DE IMPORT DENTRO DE BLOQUES COMO BUCLES O FUNCIONES\*

Trabajaremos con todo lo exportado previamente en el orden en el que se fueron mostrando los diferentes tipos de exportación

#### Importación por variable o elemento:

Sintaxis:

import {variableName} from "./relativePath.js";

EJEMPLO:

import {numero} from "./modulosExportados.js";

console.log(numero);

import {texto} from "./modulosExportados.js";

console.log(texto);

import {arregloFrutas} from "./modulosExportados.js";

console.log(arregloFrutas);

import {objetoPersona} from "./modulosExportados.js";

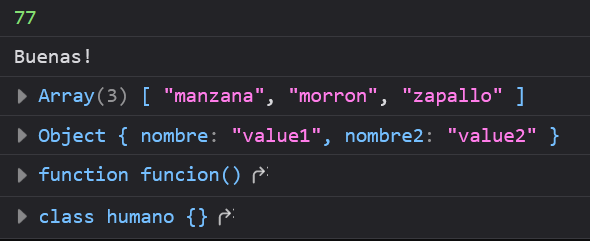
console.log(objetoPersona);

import {funcion} from "./modulosExportados.js";

console.log(funcion);

import {humano} from "./modulosExportados.js";

console.log(humano);



#### Importacion multiple:

SINTAXIS:

import {variableName1,…,variableNameN°} from "./relativePath.js";

EJEMPLO:

import {numero,texto,arregloFrutas,objetoPersona,funcion,humano} from "./modulosExportados.js";

console.log(numero);

console.log(texto);

console.log(arregloFrutas);

console.log(objetoPersona);

console.log(funcion);

console.log(humano);



#### Importacion renombrada:

Seguimos con la misma lógica anteriormente usada:

Sintaxis:

import {variableName as variableNewname} from "./relativePath.js";

import {variableName1 as variableNewname1,…,variableNameN° as variableNewnameN° } from "./relativePath.js";

EJEMPLO:

ejemplo 1:

import {numero as numeraso} from "./modulosExportados.js";

console.log(numeraso);

import {texto as textaso} from "./modulosExportados.js";

console.log(textaso);

import {arregloFrutas as arreglaso} from "./modulosExportados.js";

console.log(arreglaso);

import {objetoPersona as objetaso} from "./modulosExportados.js";

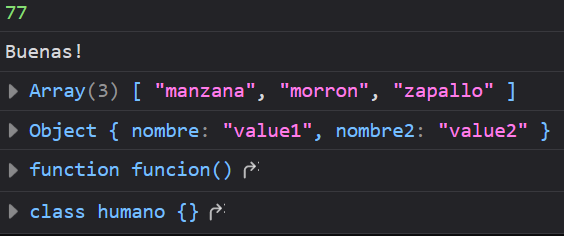
console.log(objetaso);

import {funcion as funcionasa} from "./modulosExportados.js";

console.log(funcionasa);

import {humano as humanaso} from "./modulosExportados.js";

console.log(humanaso);



ejemplo 2:

import {numero as n,texto as t,arregloFrutas as a ,objetoPersona as o,funcion as f,humano as h} from "./modulosExportados.js";

console.log(n);

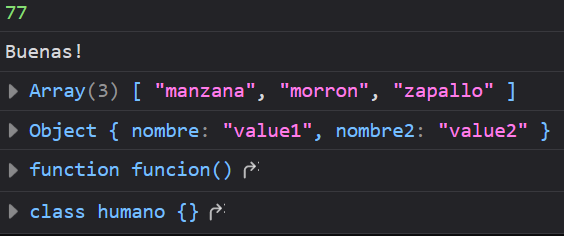
console.log(t);

console.log(a);

console.log(o);

console.log(f);

console.log(h);



#### importación por default:

Podemos importar por default en caso de que sea necesario.

Sintaxis:

import defaultName from "./relativePath.js";

EJEMPLO:

import loquesea from "./modulosExportados.js";

console.log(loquesea);

cómo se puede observar no son necesarias las “llaves” “{}” en la importación por default.

\*Se trabajo con los elementos de la exportación por default usada anteriormente\*







#### Importación general:

Si quisiéramos ser mas generales e importar todas las exportaciones de un archivo es posible.

El único requisito es darle un nombre a esta importación ya que ese nombre será del objeto que contenga todo.

Sintaxis:

import  \* as newAlias from "./relativePath.js";

EJEMPLO:

import  \* as aliasPiola from "./modulosExportados.js";

console.log(aliasPiola);

aquí estaríamos accediendo a todo el objeto:

****

si quisiéramos acceder a propiedades puntuales de este nuevo objeto renombrado lo hacemos con la misma sintaxis que los objetos.

console.log(aliasPiola.numero);

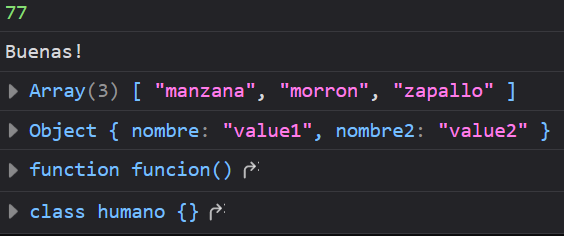
console.log(aliasPiola.texto);

console.log(aliasPiola.arregloFrutas);

console.log(aliasPiola.objetoPersona);

console.log(aliasPiola.funcion);

console.log(aliasPiola.humano);



#### importacion total:

Permite importar todo un archivo JS sin usar la palabra reservada ***export.***

Sintaxis:

import "./relativePath.js";

EJEMPLO:

Supongamos el siguiente código

JavaScript: 

console.log("Buenas, esta es una importacion local");

const f1 =  ()=>{ console.log( "nosotras somos funciones");}

const f2 =  ()=>{console.log("que se ejecutaran automaticamente al importar el archivo"); }

const f3 =  (a,b)=>{console.log(a+b); }

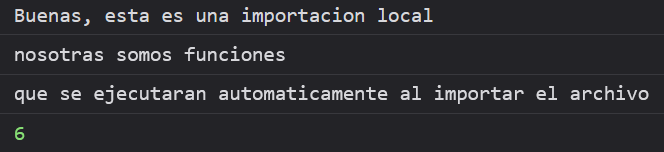
f1();

f2();

f3(2,4);

JavaScript: 

import "./modulosExportados.js";



Es como llamar a un script directamente en el archivo HTML pero haciendolo desde JS

#### Importación remota:

También podemos importar código de forma remota, a través de internet.

Basta con conocer el link donde se aloja el archivo **JS** y que el mismo haya exportado contenido.

Sintaxis:

import {variableName} from "URL";

import {variableName1,…,variableNameN°} from "URL";

# Json:

Json significa JavaScript Object Notation, es decir

Notación de objeto de JavaScript.

Sirve para estructurar datos en formato de texto plano con la sintaxis de objetos de **JS**, de esta forma podemos intercambiar información entre aplicaciones de manera sencilla, liviana y rápida.

Nos sirve para comunicarnos entre archivos locales, y también para comunicarnos en internet con modelos como el cliente servidor.

Json no propone la única forma de realizar esta comunicación, pero conserva hasta ahora, la estructura más intuitiva y sintácticamente simple de leer y escribir, además de tener una gran inclusividad con respecto a la diversidad de lenguajes que son compatibles con el formato Json.

Un objeto en **JS** se ve así:

const objetoJS = {

  propiedad1: "valor1",

  propiedad2: "valor2",

  propiedad3: "valor3",

}

Un objeto Json se ve así:

const objetoJson = {

  "propiedad1": "valor1",

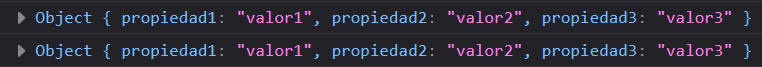
  "propiedad2": "valor2",

  "propiedad3": "valor3",

}

console.log(objetoJS);

console.log(objetoJson);



Como podemos observar, ambos son objetos

Pero su principal diferencia sintáctica como se puede observar es simplemente sus comillas tanto es sus llaves como en sus valores.

Esto parece irrelevante, pero veremos mas adelante porque es importante

Como se describió previamente, un JSON es una cadena cuyo formato recuerda al de los objetos literales JavaScript. Es posible incluir los mismos tipos de datos básicos dentro de un JSON que en un objeto estándar de JavaScript - cadenas, números, arreglos, booleanos, etc...

## Serialización/deserialización

Para poder **enviar** un archivo Json a un servidor debemos ***serializar*** el json

Y para poder ***recibir*** datos desde un servidor debemos ***deserializar*** el formato json.

El proceso de serialización: consiste simplemente en convertir todo el objeto Json en texto plano es decir en una cadena de texto entera, y la deserialización en lo contrario: en convertir una cadena de texto plana en un archivo de formato Json, afortunadamente Json cuenta con métodos propios para poder hacer esto de forma sencilla.

Sintaxis:

JSON.stringify(ObjectName); /// para serializar

JSON.parse(ObjectName); /// para deserializar

EJEMPLO:

Supongamos el siguiente Objeto en formato Json

const objetoJson = {

  "propiedad1": "valor1",

  "propiedad2": "valor2",

  "propiedad3": "valor3",

}

//. Serializar:

Json\_serializado =  JSON.stringify(objetoJson);

console.log(Json\_serializado);

//. Deserializar:

Json\_deserializado =  JSON.parse(Json\_serializado);

console.log(Json\_deserializado);



console.log(typeof Json\_serializado);

console.log(typeof Json\_deserializado);



Como vemos, pasaron por un proceso de conversión de objeto a string y luego de string a objeto.

Podemos serializar o deserializar cualquier cosa, no solo un objeto con el formato Json

Pero es nuestra responsabilidad saber a qué aplicamos estos métodos

Cabe mencionar que hoy en día un archivo Json puede estar totalmente aparte de un archivo JS

Y que el archivo JS puede importar o hacer uso de un archivo JSON externo.

# FETCH API:

Para entender lo que es fetch api tenemos que tener una idea de algunos conceptos.

Primero, ¿qué es un api?

La interfaz de programación de aplicaciones, conocida también por la sigla API, en inglés, application programming interface, ​ es un conjunto de subrutinas, funciones y procedimientos que ofrece cierta biblioteca para ser utilizada por otro software como una capa de abstracción

Es decir, es un conjunto de utilidades que puede ser usada por algo externo.

Nosotros vamos a necesitar a veces comunicarnos con aplicaciones, paginas, o bases de datos, que no son nuestras.

Gracias al internet, esto es posible, y nosotros podemos consultar una api, para obtener datos deseados.

Una api puede contener material muy diverso.

Por ejemplo, puede ser una API que tenga una colección de imágenes de animales

O una API que todas las conversiones de Unidades de medida.

O una API de algún software conocido como Google map.

Si nosotros quisiéramos agregar a Google map en nuestra pagina ¿como podríamos hacerlo?

La respuesta es comunicándonos y usando con la API Google maps

lo que se conoce como “consumir” una API

pero esta comunicación se puede dar de varias maneras,

una de estas formas de comunicarse es mediante fetch

**fetch** proporciona una interfaz para recuperar recursos de manera sencilla.

Mediante la API de fetch nosotros podemos realizar diferentes solicitudes http

las cuales se conocen como verbos o métodos:

##### Métodos de petición HTTP

HTTP define un conjunto de **métodos de petición** para indicar la acción que se desea realizar para un recurso determinado. Aunque estos también pueden ser sustantivos, estos métodos de solicitud a veces son llamados *HTTP verbs*.

GET

El método GET solicita una representación de un recurso específico. Las peticiones que usan el método GET sólo deben recuperar datos.

HEAD

El método HEAD pide una respuesta idéntica a la de una petición GET, pero sin el cuerpo de la respuesta.

POST

El método POST se utiliza para enviar una entidad a un recurso en específico, causando a menudo un cambio en el estado o efectos secundarios en el servidor.

PUT

El modo PUT reemplaza todas las representaciones actuales del recurso de destino con la carga útil de la petición.

DELETE

El método DELETE borra un recurso en específico.

CONNECT

El método CONNECT establece un túnel hacia el servidor identificado por el recurso.

OPTIONS

El método OPTIONS es utilizado para describir las opciones de comunicación para el recurso de destino.

TRACE

El método TRACE realiza una prueba de bucle de retorno de mensaje a lo largo de la ruta al recurso de destino.

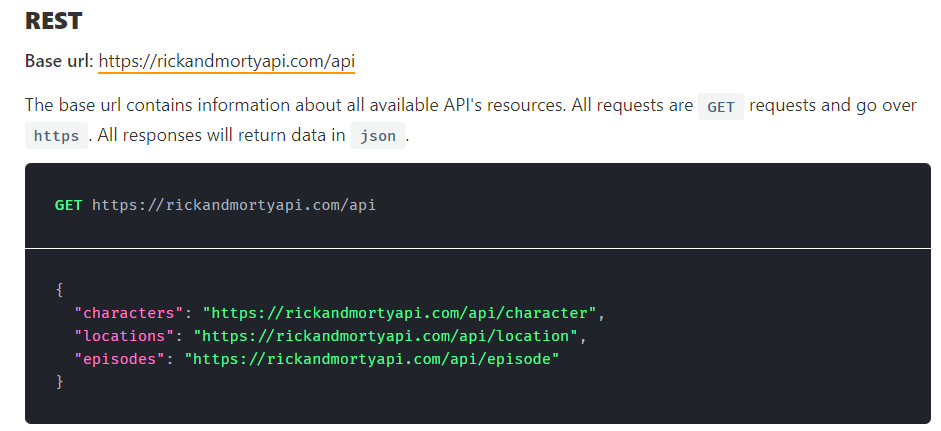
PATCH

El método PATCH es utilizado para aplicar modificaciones parciales a un recurso.

Supongamos que vamos a trabajar con la api de Rick and morty.

Si… existe…

<https://rickandmortyapi.com/documentation/#introduction>



Veamos que contienen las url.

Sintaxis:

fetch(url)

*EJEMPLO:*

Recordemos que fetch por defecto usa el verbo GET y también recordemos que devuelve una promesa, por lo tanto tenemos que tratarla, definirla que pasa si es exitosa o no.

\*para este ejemplo debemos tener una conexión a internet\*

const url\_1 = "https://rickandmortyapi.com/api";

fetch(url\_1)

    .then((respuesta) => respuesta)

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

const url\_2 = "https://rickandmortyapi.com/api/character";

fetch(url\_2)

    .then((respuesta) => respuesta)

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

const url\_3 = "https://rickandmortyapi.com/api/location";

fetch(url\_3)

    .then((respuesta) => respuesta)

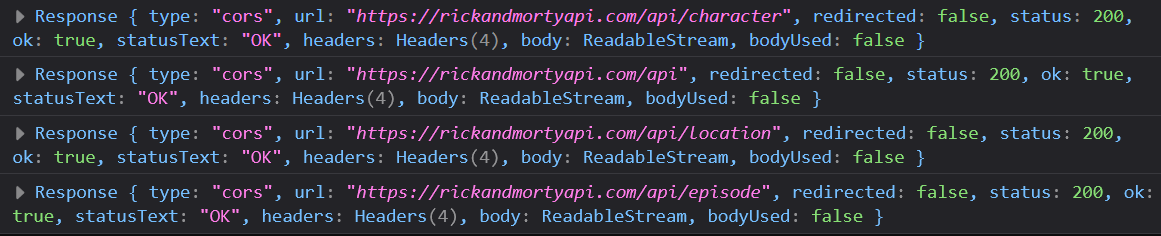
    .then((data) => console.log(data));

const url\_4 = "https://rickandmortyapi.com/api/episode";

fetch(url\_4)

    .then((respuesta) => respuesta)

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



Podemos observar que obtuvimos una respuesta, pero no estamos obteniendo información puntual aún.

Estamos obteniendo una respuesta HTTP para saber si nuestra solicitud salió bien o salió mal.

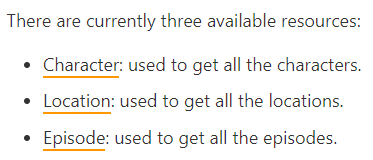
Si quisiéramos trabajar con la información que hay en la API tenemos que:

1° Conocer la estructura de la API.

2° Extraer el contenido convirtiéndolo en formato Json.

Hay formas de obtener esta información con fetch pero antes tenemos que tener en cuenta como está estructurada la API externa con la que vamos a trabajar,

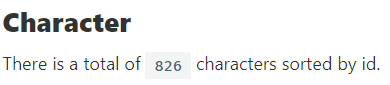
Es decir:



La API de Rick and morty nos dice QUE podemos obtener, pero también nos tiene que decir COMO.

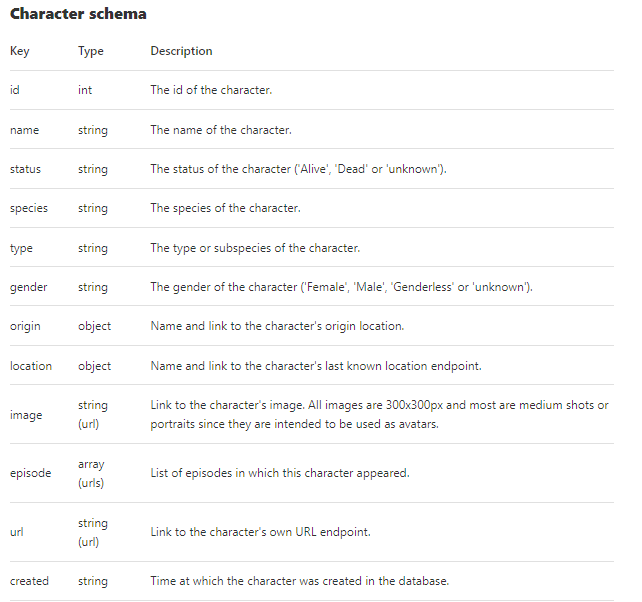
¡Todas las APIs deben tener su documentación para poder consumirlas correctamente!!

Por ejemplo, veamos los personajes:

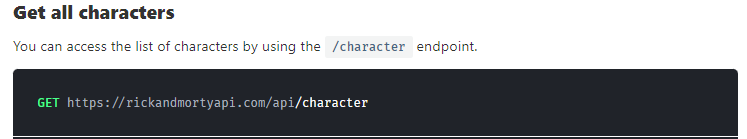


Dice que hay en total 826 personajes ordenados por id

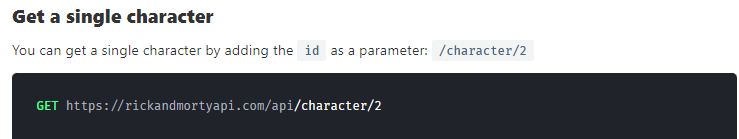
Y que este es su esquema:



Nos dice como acceder a todos los personajes:



Y como acceder a un solo personaje.



Y todo esto es importante, porque no hay una manera específica de organizar una API todo va a depender del creador, y su sintaxis de consulta está totalmente pensada por quien la cree, por eso, si nosotros queremos consultar una API y acceder a las propiedades de sus objetos, tenemos que conocer los nombres de esas propiedades y la sintaxis que esta API tiene.

##### **Obtener datos con el fetch:**

Para obtener el contenido de las repuestas que consultamos con la fetch API debemos hacer uso del método Json.

.then((respuesta) => respuesta.json())

const url\_1 = "https://rickandmortyapi.com/api";

fetch(url\_1)

    .then((respuesta) => respuesta.json())

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

const url\_2 = "https://rickandmortyapi.com/api/character";

fetch(url\_2)

    .then((respuesta) => respuesta.json())

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

const url\_3 = "https://rickandmortyapi.com/api/location";

fetch(url\_3)

    .then((respuesta) => respuesta.json())

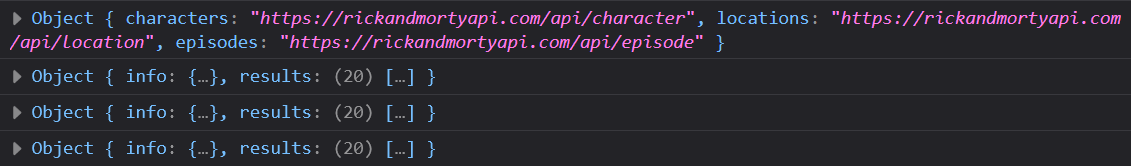
    .then((data) => console.log(data));

const url\_4 = "https://rickandmortyapi.com/api/episode";

fetch(url\_4)

    .then((respuesta) => respuesta.json())

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



Vemos que ahora por consola obtuvimos algo muy diferente a lo anterior.

Ahora probemos la sintaxis de los personajes:

const url\_1 = "https://rickandmortyapi.com/api/character/1";

fetch(url\_1)

    .then((respuesta) => respuesta.json())

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

const url\_2 = "https://rickandmortyapi.com/api/character/23";

fetch(url\_2)

    .then((respuesta) => respuesta.json())

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

const url\_3 = "https://rickandmortyapi.com/api/character/415";

fetch(url\_3)

    .then((respuesta) => respuesta.json())

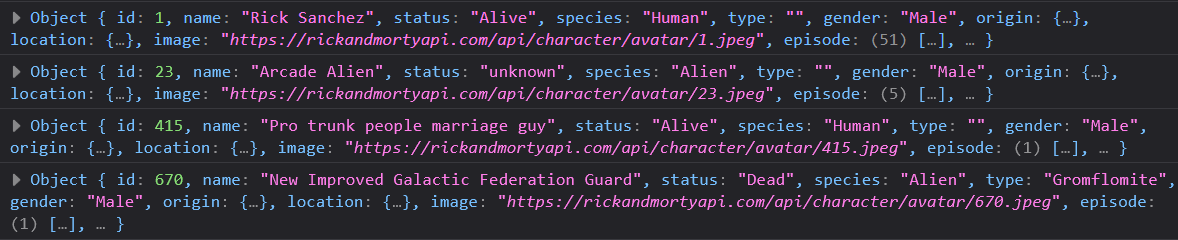
    .then((data) => console.log(data));

const url\_4 = "https://rickandmortyapi.com/api/character/670";

fetch(url\_4)

    .then((respuesta) => respuesta.json())

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



Ahora todo empieza a ser mucho más divertido.♥

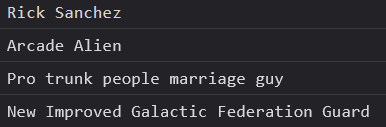
Si cambiásemos esto:

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

Por esto:

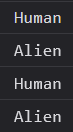
.then((data) => console.log(data.name));

Obtendríamos esto:



Y lo mismo pasaría con todos los atributos vistos antes en el esquema de personajes:

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



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

