DWC

(Desarrollo Web en entorno cliente)

Índice

1.-Funciones

Las funciones en JavaScript son bloques de código reutilizables que realizan una tarea específica. Permiten organizar el código, hacerlo más modular y evitar la repetición de código. Las funciones pueden recibir entradas (parámetros) y devolver valores (resultados).

El concepto básico de funciones es similar al de otros lenguajes, aunque en JavaScript tenemos algunas peculiaridades como parámetros rest, expresión de funciones, funciones anónimas y fat arrow

Estructura básica de una función

```
function nombreFuncion(parametro1, parametro2, ...) {
  // Cuerpo de la función
  // Instrucciones que se ejecutan cuando se llama a la función
 return valor
}
```

Partes de una función:

- **function:** Palabra clave que indica el inicio de la definición de la función.
- **nombreFuncion:** Nombre identificativo de la función. Debe ser un nombre válido en JavaScript.
- **parametro1, parametro2, ...:** Parámetros opcionales de la función. Se utilizan para pasar datos a la función cuando se llama.
- **{}:** Llaves que delimitan el cuerpo de la función.
- **instrucciones:** Instrucciones JavaScript que se ejecutan cuando se llama a la función.
- **return:** Las funciones pueden devolver valores utilizando la palabra clave return.

Tipos de funciones:

- **Funciones declaradas:** Se definen utilizando la palabra clave function seguida del nombre y los parámetros.
- **Funciones de expresión:** Se definen utilizando la sintaxis de expresión de funciones.

- **Funciones anónimas:** También conocidas como **IIFE** (Immediately Invoked Function Expression), se ejecutan inmediatamente después de ser definidas.

1.1.- Parámetros

Los parámetros son los datos que le pasamos a la función, en JavaScript tenemos las siguientes alternativas:

- **Pasar el mismo número de parámetros** a la función que el número definido en la cabecera de la función. Esta sería la forma tradicional de los demás lenguajes
- **Utilizar valores predeterminados** para los parámetros, esto quiere decir que podemos definir en la cabecera de la función valores predeterminados, si en la llamada a la función no se incluye ese parámetro, el valor que tomara la función será el que hayamos definido como predeterminado.

```
function muestraMensaje(msg = "No hay texto...") {
  alert(msg);
}

muestraMensaje(); // Llamamos a la funcion sin parametro
```

- **Utilizar el objeto arguments**, es un objeto **similar a un array** que está **disponible dentro de todas las funciones** (excepto las fat arrow) y contiene los valores de los argumentos pasados a esa función. Se puede utilizar para acceder a los argumentos de la función dentro del cuerpo de la misma. Nos permite llamar a la función con los parámetros que queramos, sin tener que definir parámetros en la función. Actualmente está en desuso

```
function sumar() {
  let total = 0;
  for (let i = 0; i < arguments.length; i++) {
    total += arguments[i];
  }
  return total;
}

const resultado = sumar(10, 5, 3, 2);
```

```
console.log('La suma es:', resultado); // Imprime: La suma es:
20
```

- **Parámetros Rest y Operador Spread**, son unas características introducida en ECMAScript 6 (ES6).

  los parámetros rest permiten a las funciones aceptar un número indeterminado de argumentos. Estos argumentos se recopilan en un array dentro del cuerpo de la función.

```
function nombreFuncion(...parametroRest) {
 // Cuerpo de la función
}
```

Los parámetros rest se diferencian del objeto arguments en que realmente son un array y podemos aplicar todos los métodos del objeto array

```
function sumar(...numeros) {
  let total = 0; // Se inicializa la variable `total` para
almacenar la suma

  // El bucle forEach recorre cada número en el array `numeros`
  numeros.forEach(function(numero) {
    total += numero; // Se suma cada número al valor de `total`
  });

  // Se devuelve el valor de `total`, que representa la suma de
todos los números
  return total;
}
```

El operador spread, también conocido como sintaxis spread, nos permite **expandir elementos iterables** como arrays y objetos en argumentos individuales o elementos de un array literal.

```
function sumar(...numeros) {
  // Cuerpo de la función
}

const numeros = [10, 5, 3, 2];
sumar(...numeros); // Equivalente a sumar(10, 5, 3, 2)
```

Utilizado también para crear arrays sin modificar arrays existentes, problema de **tratar los arrays como referencias**.

```
const arrayOriginal = [1, 2, 3, 4, 5];
```

```
const arrayCopia = [...arrayOriginal];
console.log(arrayCopia === arrayOriginal); // Imprime: false (son
diferentes referencias)
console.log(arrayCopia); // Imprime: [1, 2, 3, 4, 5] (contenido
igual)
```

1.2.- Expresiones de funciones

En JavaScript, una función es un tipo especial de valor. La sintaxis que usamos hasta ahora se llama ***declaración de función***. Hay otra sintaxis para crear una función que se llama ***expresión de función***.

Las expresiones de funciones en JavaScript son una forma **concisa** de definir funciones que se puede utilizar para crear **funciones anónimas** o asignar **funciones a variables**.

Una función anónima es aquella que se define sin un nombre y que no se reutiliza, ya que se asigna directamente a una variable o a la propiedad de un objeto. Un caso de ello podría ser la asignación de eventos desde código

Declaración de función

```
function hola() {
  alert( "Hola..." );
}
```

Expresión de función

```
let hola = function() {
  alert( "Hola..." );
};
```

Aquí, la función se crea y se asigna a la variable explícitamente, como cualquier otro valor. No importa cómo se defina la función, es solo un valor almacenado en la variable hola. **Muy importante**, al ser una expresión que se asigna a la variable, esta instrucción **deberá acabar con un punto y coma**

Se utilizan ampliamente como **callbacks** para definir funciones que se pasan como argumentos a otras funciones, como en eventos, APIs o bibliotecas.

1.3.- Sintaxis fat arrow

La sintaxis fat arrow, también conocida como **expresión de función de flecha** o **función lambda**, es una forma concisa de escribir funciones JavaScript introducida en ECMAScript 6 (ES6). Se caracteriza por usar la flecha (**=>**) en lugar de la palabra clave function y por omitir las llaves (**{}**) y la palabra clave **return** en casos simples.

En esencia es una forma sintáctica diferente de definir las funciones, que nos ofrece una mayor sencillez a la hora de trabajar, principalmente cuando utilicemos programación funcional. Un ejemplo claro, pueden ser las funciones de callback utilizadas como parámetros en otras funciones.

Sintaxis básica:

```
(parametro1, parametro2, ...) => {
  // Cuerpo de la función
}
```

Explicación:

- **Parámetros:** Los parámetros de la función se definen entre paréntesis, separados por comas. **Si no hay parámetros, se utilizan paréntesis vacíos ().**
- **Flecha:** La flecha (=>) separa los parámetros del cuerpo de la función.
- **Cuerpo de la función:** El cuerpo de la función se define entre llaves {}. **Si la función solo tiene una instrucción, las llaves y la palabra clave return se pueden omitir**.

```
const sumar = (a, b) => a + b; // Función que suma dos números

const resultado = sumar(5, 3); // Se llama a la función
console.log('La suma es:', resultado); // Imprime: La suma es: 8
```

2.- Iterables

En JavaScript, un **iterable** es un objeto que se puede recorrer utilizando un bucle. Los iterables permiten acceder a los elementos de una colección de forma secuencial.

Los iterables más importantes en JavaScript son:

- **Arrays:** Los arrays son el tipo iterable más común. Permiten almacenar y acceder a una colección de valores ordenados.
- **Sets:** Los sets son colecciones de valores únicos que no tienen orden. Se pueden iterar para acceder a cada valor del set.
- **Maps:** Los mapas son colecciones de pares clave-valor. Se pueden iterar para acceder a las claves o a los valores del mapa.
- **Objetos:** Algunos objetos en JavaScript son iterables, como los objetos que contienen propiedades con valores iterables (por ejemplo, arrays o strings).

2.1.- Arrays

Los arrays en JavaScript son estructuras de datos que permiten almacenar una colección de valores. Son una herramienta fundamental para organizar y manipular datos en aplicaciones JavaScript.

La idea básica es la misma en todos los lenguajes, una variable que permite almacenar más de un valor y poder iterar sobre ella para acceder a todos ellos.

En el ES6 se incorporaron nuevos métodos que hacen uso de funciones de callback como parámetros, que junto a la sintaxis fat arrow y el uso de programación funcional simplifican mucho las operaciones

2.1.1.- Declaración y acceso a elementos

Existen dos formas principales de crear arrays en JavaScript:

1.-Literal de array: La forma más común de crear un array es utilizando un literal de array, que consiste en una lista de valores encerrada entre corchetes []. Los valores pueden separarse por comas.

```
const frutas = ["manzana", "naranja", "plátano"];
```

```
const numeros = [10, 5, 3, 2];
const mixto = ["valor1", 20, true, { nombre: "Objeto" }];
```

2. Constructor de Array:

El constructor de Array (Array()) también se puede utilizar para crear arrays. Se puede pasar una lista de valores como argumento al constructor.

```
const frutas = new Array("manzana", "naranja", "plátano");
const numeros = new Array(10, 5, 3, 2);
```

Se puede acceder a los elementos de un array utilizando su índice entre corchetes []. El índice comienza en 0, por lo que el primer elemento tiene el índice 0, el segundo elemento tiene el índice 1, y así sucesivamente.

```
const frutas = ["manzana", "naranja", "plátano"];

console.log(frutas[0]); // Imprime: manzana
console.log(frutas[1]); // Imprime: naranja
console.log(frutas[2]); // Imprime: plátano
```

2.1.2.- Propiedades y métodos de arrays

Los arrays disponen de la **propiedad length** de un array que contiene el número de elementos que tiene el array.

```
const frutas = ["manzana", "naranja", "plátano"];

console.log(frutas.length); // Imprime: 3
```

Los arrays en JavaScript proporcionan una serie de **métodos** para manipular sus elementos y realizar las operaciones más habituales.

- **Agregar, eliminar y extraer elementos**

  push(): Agrega un nuevo elemento al final del array.

  pop(): Elimina el último elemento del array y lo devuelve.

  shift(): Elimina el primer elemento del array y lo devuelve.

  unshift(): Agrega un nuevo elemento al principio del array.

  splice(:): Elimina elementos del array y/o agrega nuevos elementos.

  slice(): Extrae una subsección del array.

  concat(): Concatena dos o más arrays.

- **Iterar para cada elemento**

forEach(): Recorre cada elemento del array y ejecuta una función de callback.

- **Buscar en arrays**

  indexOf(), lastIndexOf(): Devuelven el índice donde se encuentra la primera o última aparición de un elemento en el array

  includes(): Devuelve boolenao indicando si un elemento existe o no

  find() and findIndex(): Buscan un elemento o su posición en el array

  filter(): Filtra elementos en nuevo arrayr

- **Transformar un array**

  map(): Transforma el contenido de un array

  reduce(): Obtiene un valor del array aplicando una formula

- **Ordenar arrays**

  sort(): Ordena un array

  reverse(): Invierte un array

- **String – Array**

  split(): Convierte una cadena en array

  join(): Convierte un array en cadena

Para poder consultar todas las propiedades y métodos del objeto array consultar la guía de MDN
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array

2.1.3.- Funciones de callback

En el ES6 se incorporaron una serie de métodos para el objeto array que trabajan con **funciones de callback**, esto simplifica mucho las operaciones sobre arrays y nos permiten utilizar programación funcional que nos evita tener que recurrir a los bucles habituales que recorrían el array e iban realizando las operaciones que queríamos, ahora realizaremos las operaciones como métodos del objeto array y usaremos como parámetro una función que ya va integrada en el método

Por ejemplo, para mostrar los elementos de un array en pantalla, la forma tradicional seria:

```
let  lista=[1,2,3,4,5]
for(let i=0;i<lista.length;i++){
  console.log(lista[i])
}
```

Utilizando el método forEach

```
lista.forEach(function(e){console.log(e)})
```

El método **forEach()** ejecuta la función indicada una vez por cada elemento del array, esa función es la función de callback, que en este caso recibe como parámetro cada elemento del array, también recibe como parámetros la posición actual y el array completo. Generalmente sólo nos suele interesar utilizar el parámetro del elemento actual, aunque hay caso en los que se utiliza la posición

```
array.forEach(function callback(currentValue, index, array) {
                código de nuestra función })
```

Lo más habitual es utilizar las funciones de callback con sintaxis fat arrow, de esta manera se simplifica mucho más el código.

```
lista.forEach(e=>console.log(e))
```

Repaso de los métodos más usados en MDN

2.1.4.- El tipo Array

Trabajando con arrays (***o que pensamos que son arrays***) pueden surginos un par de dudas:

Hay ocasiones en las que necesitamos saber si un objeto es realmente un array, para ello deberíamos comprobar si el tipo es exactamente un array. En JavaScript existen otro tipo de objetos iterables similares a los arrays (nodeList y HTMLCollection) pero no son realmente arrays, con lo cual no se pueden aplicar los métodos de arrays.

Si nos encontráramos en esa situación sería interesante poder convertir esos objetos en arrays para poder utilizar los métodos vistos.

Los arrays no forman un tipo del lenguaje aparte, se basan en objetos, por lo tanto, typeof no ayuda a distinguir un objeto de un array:

```
alert(typeof {}); // object
alert(typeof []); // object
```

Pero los arrays se usan con tanta frecuencia que hay un método especial para comprobar si un elemento es un array, ese método es **Array.isArray(elemento)** que devuelve true si value es una matriz, y de lo contrario false.

```
alert(Array.isArray({})); // false
alert(Array.isArray([])); // true
```

El método **Array.from(object)** permite convertir en array objetct. Esto es muy interesante cuando trabajamos con objetos que aparentemente son Arrays pero realmente no lo son. Esto lo veremos en el DOM.

Los objetos **NodeList** son colecciones de nodos como los devueltos por propiedades de Node.childNodes y métodos como document.querySelectorAll (), document.getElementsBy…

Aunque NodeList no es un Array, es posible iterar sobre él utilizando forEach().

```
<div id="contenedor">
  <p>Parrafo</p>
  <button>Boton</button>
  <h1>Encabezado</h1>
</div>
<script>
  // NodeList
  nodosHijos=document.getElementById("contenedor").childNodes
  console.log(nodosHijos)
  nodosHijos.forEach(n=>console.log(n))
</script>
```

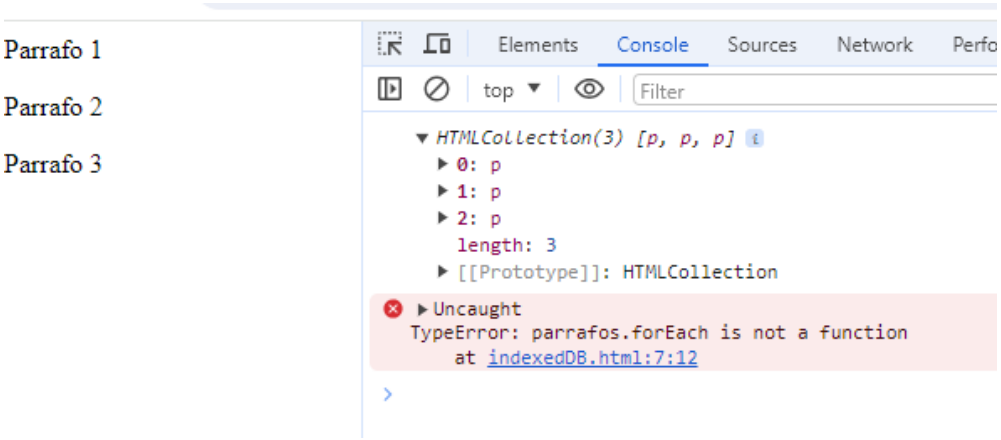
Un **HTMLCollection** es un objeto similar a un array que representa una colección de elementos HTML extraídos del documento. Se puede tener acceso a los

elementos de la colección mediante un índice numérico, un nombre o un identificador

```
document.getElementsByTagName('p')[2]
```

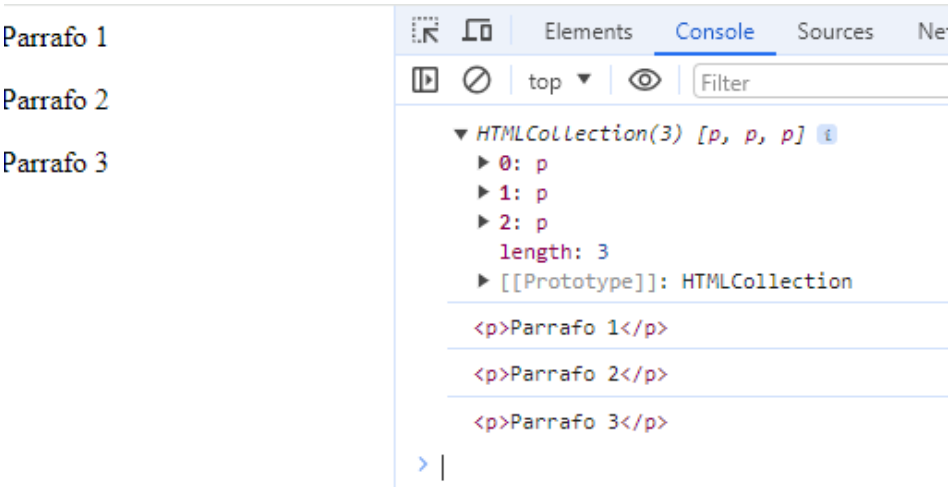
A diferencia del NodeList, **no es compatible con el método forEach().**

```
<p>Parrafo 1</p>
<p>Parrafo 2</p>
<p>Parrafo 3</p>
<script>
  const parrafos = document.getElementsByTagName('p');
  console.log(parrafos)
  parrafos.forEach(p=>console.log(p))
</script>
```



Sin embargo, se puede utilizar el método **Array.from(HTMLCollection)** para convertir la colección a un array normal y, a continuación, utilizar foreach() para iterar sobre él.

```
Array.form(parrafos).forEach(p=>console.log(p))
```



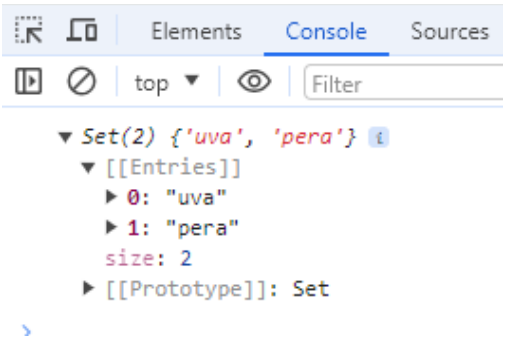
2.2.- Sets

En JavaScript, el objeto Set es una estructura de datos que representa un **conjunto de valores únicos**. A diferencia de los arrays, los sets no permiten valores duplicados y mantienen el orden de inserción de los elementos. El objeto Set se introdujo en la especificación ES6 (2015) y se ha convertido en una herramienta valiosa para trabajar con conjuntos de datos en JavaScript.

Creación de un Set:

La forma más común de crear un Set es utilizando un literal de Set, que consiste en una lista de valores encerrada entre corchetes [] con el prefijo new Set().

```
// Solo se agregan "uva" y "pera" una vez
const frutas = new Set(["uva", "pera", "uva", "pera"]);
console.log(frutas); // Imprime: Set { "uva", "pera" }
```



Métodos de Set:

El objeto Set proporciona una serie de métodos para manipular sus elementos. Algunos de los métodos más comunes incluyen:

- **add(valor):** Agrega un nuevo valor al set. Si el valor ya existe, no se agrega de nuevo.
- **delete(valor):** Elimina un valor del set.
- **has(valor):** Devuelve true si el valor existe en el set, y false si no.
- **size:** Devuelve la cantidad de elementos en el set.
- **clear():** Elimina todos los elementos del set.
- **values():** Devuelve un iterador que permite recorrer los valores del set.
- **forEach(callback):** Recorre cada valor del set y ejecuta una función de callback.

2.3 Maps

En JavaScript, un **map** (también conocido como **objeto mapa** o **diccionario ordenado**) es una estructura de datos que almacena pares de **clave-valor**

Características:

- **Claves únicas:** Las claves en un mapa deben ser únicas. Si se intenta agregar una nueva clave que ya existe, se sobrescribirá el valor asociado con la clave existente.
- **Claves de cualquier tipo:** Las claves en un mapa pueden ser de cualquier tipo de dato válido en JavaScript, incluyendo cadenas, números, objetos e incluso otros mapas.
- **Valores de cualquier tipo:** Los valores en un mapa también pueden ser de cualquier tipo de dato válido en JavaScript, incluyendo estructuras de datos complejas como arrays y objetos.

Creación de un mapa:

La forma más común de crear un mapa en JavaScript es utilizando el constructor Map(). Se puede pasar un iterable de pares de clave-valor como argumento al constructor o crearlo vacío e ir rellenándolo. Generalmente se suele utilizar con objetos como valor.

Vamos a crear un map para almacenar clientes con un id como clave y como valor un objeto con su ciudad y su edad

```
const clientes = new Map([
      [1, {nombre: "Juan", ciudad: "Madrid", edad: 30}],
      [2, {nombre: "Ana", ciudad:" Barna", edad:25}]
    ]);

clientes.set(3, {nombre: "Carlos", ciudad: "Valencia", edad: 40});
```

Métodos de mapas:

Los map en JavaScript proporcionan algunos métodos útiles para manejar sus claves y valores. Algunos de los métodos más comunes incluyen:

- **has(clave)**: Devuelve true si la clave existe en el mapa, y false si no.
- **get(clave):** Devuelve el valor asociado con la clave especificada. Si la clave no existe, devuelve undefined.

- **set(clave, valor):** Agrega o actualiza un par de clave-valor en el mapa. Si la clave ya existe, se sobrescribe el valor asociado.
- **delete(clave):** Elimina el par de clave-valor asociado con la clave especificada.
- **keys():** Devuelve un iterador que permite recorrer las claves del mapa.
- **values():** Devuelve un iterador que permite recorrer los valores del mapa.
- **entries():** Devuelve un iterador que permite recorrer los pares de clave-valor del mapa.
- **forEach(callback):** Recorre cada par de clave-valor del mapa y ejecuta una función de callback.

```
console.log(`Cliente 1: ${clientes.get(1).nombre}
             de ${clientes.get(1).ciudad}
             (${clientes.get(1).edad} años)`);

console.log(`Cliente 4: ${clientes.get(4) || "No existe el cliente
4."}`);

clientes.forEach((cliente, id) => {
                  console.log(`Cliente ${id}: ${cliente.nombre}
                                de ${cliente.ciudad}
                                (${cliente.edad} años)`);
});
```

3.- Objetos

En JavaScript, los **objetos** son estructuras de datos fundamentales que nos permiten almacenar, organizar y acceder a información de forma flexible y eficiente. Se utilizan en prácticamente todas las áreas del desarrollo web moderno, desde la representación de datos en el front-end hasta la gestión de lógica compleja en el back-end.

Un objeto en JavaScript **es una colección de pares clave-valor**. Cada par consta de una **clave** (que identifica al valor) y un **valor** (que puede ser de cualquier tipo de dato en JavaScript).

Los objetos en si son arrays asociativos, es decir son arrays en los que la información no se guarda en posiciones accesibles por índices númericos, si no que se almacena en posiciones accesibles por nombres.

3.1.- Creación de objetos en JavaScript

Existen dos formas principales de crear objetos en JavaScript:

1. **Literal de objeto:**

   El método más común es utilizar la notación literal de objetos. Se utilizan llaves {} para definir el objeto y se separan los pares clave-valor con comas.

```
const persona = {
  nombre: "Juan",
  apellido: "Pérez",
  edad: 30,
  ciudad: "Valencia"
};
```

2. **Constructor de objetos:**

   También puedes crear objetos utilizando el constructor Object(). Sin embargo, este método se utiliza con menos frecuencia, ya que la sintaxis literal es más concisa y fácil de leer.

```
const persona = new Object();
persona.nombre = "Juan";
persona.apellido = "Pérez";
persona.edad = 30;
persona.ciudad = "Valencia";
```

3.2.-Acceso a las propiedades de un objeto

Para acceder a las propiedades de un objeto, se utilizan dos sintaxis principales:

1. **Notación de punto:**

   La notación de punto utiliza el nombre de la propiedad después del nombre del objeto, separado por un punto ..

```
const nombrePersona = persona.nombre;
```

2. **Notación de corchetes:**

   La notación de corchetes utiliza el nombre de la propiedad entre corchetes [] después del nombre del objeto. Es útil cuando el nombre de la propiedad es dinámico o no se conoce de antemano.

```
const propiedad = "edad";
const edadPersona = persona[propiedad];
```

Una característica notable de los objetos en JavaScript, en comparación con muchos otros lenguajes, es que es posible acceder a cualquier propiedad. ¡No habrá error si la propiedad no existe!

La lectura de **una propiedad no existente solo devuelve undefined**. Entonces podemos probar fácilmente si la propiedad existe:

```
alert(persona.cp)
```

También hay un operador especial "**in**"para comprobar si existe una propiedad en un objeto. La sintaxis es: "key" in object

```
alert("nombre" in persona)
alert("cp" in persona)
```

¿Por qué existe el operador in? ¿No es suficiente para comparar undefined? Bueno, la mayoría de las veces la comparación undefined funciona bien. Pero hay un caso especial cuando falla, pero "in" funciona correctamente.

Es cuando existe una propiedad de objeto almacena undefined:

```
persona.cp=undefined
console.log(persona.cp)
```

En este caso el acceso a la propiedad cp nos devuelve undefined, no porque no exista, sino porque vale undefined. En este caso el operador in nos resuelve la duda.

También podemos crear propiedades nuevas únicamente asignado un valor a esa propiedad.

```
console.log(persona.cp)
persona.cp="46026"
console.log(persona.cp)
```

El bucle "for... in"

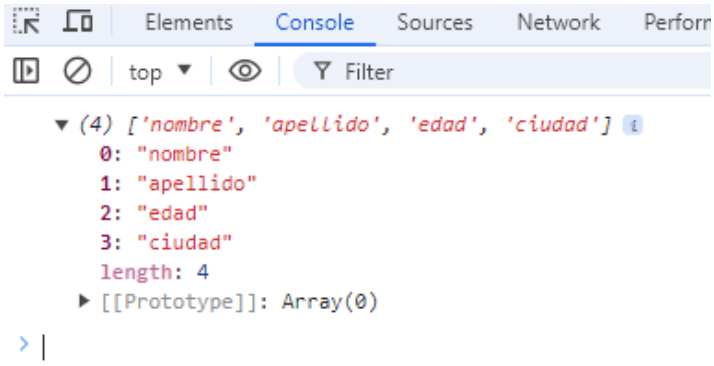
Para recorrer todas las propiedades de un objeto, existe una forma que es utilizando el operador in dentro de un bucle for.

```
for (key in persona){
    console.log(`${key}: ${persona[key]}`)
}
```

El método Object.keys(object)

El método Object.keys(object) nos permite obtener todas las propiedades de un objeto. Devuelve un array con todas las propiedades del objeto que le pasamos

```
let propiedades=Object.keys(persona)
console.log(propiedades)
```



La comparación de objetos no es una cosa trivial como en otras variables, en los objetos con que cambie el orden de una propiedad ya nos va a devolver que los dos objetos son diferentes.

```
const persona1 = {
  nombre: "Juan",
  apellido: "Pérez",
  edad: 30,
  ciudad: "Valencia"
};
const persona2 = {
  ciudad: "Valencia",
  nombre: "Juan",
  apellido: "Pérez",
  edad: 30
};

console.log(persona1==persona2)
```

Cuando queremos trabajar con comparación de objetos, realmente lo que queremos para que dos objetos sean iguales es que tengan las mismas propiedades y los mismos valores para cada propiedad.

3.3.- Métodos de los objetos

Los objetos en JavaScript también pueden tener **métodos**, que son funciones asociadas al objeto. Los métodos se definen utilizando la palabra clave function y se llaman utilizando la misma sintaxis que las propiedades.

```
const persona = {
  nombre: "Juan",
  apellido: "Pérez",
  edad: 30,
  ciudad: "Madrid",
  // Método "obtenerNombreCompleto"
  obtenerNombreCompleto: function() {
    return this.nombre + " " + this.apellido;
  }
};

const nombreCompleto = persona.obtenerNombreCompleto();
```

This en JavaScript

En JavaScript, la palabra clave se this se comporta diferente de la mayoría de los otros lenguajes de programación. Se puede usar en cualquier función.

```
function hola(){
    alert (this)
}
hola()
```

El valor de this se evalúa durante el tiempo de ejecución, según el contexto. Por ejemplo, en JavaScript podemos utilizar una función para gestionar el evento click de una serie de botones de nuestra web, es decir no vamos a crear una función para cada botón, sino que todos tendrán la misma función. This dentro de esa función permitirá saber cuál es el botón que la ha llamado y en función de eso realizar las acciones necesarias.

3.4.- JSON (JavaScript Object Notation)

Es un formato ligero de datos basado en texto para la representación de datos estructurados. Se utiliza ampliamente para el intercambio de datos entre aplicaciones web, especialmente en el contexto de las APIs y las solicitudes AJAX. Antiguamente se utilizaba XML, pero con el desarrollo de las aplicaciones web modernas fue reemplazado por “la complejidad” de la representación de los datos y “el peso” de los datos para la comunicación con el servidor

Características principales de JSON

- **Ligero y fácil de leer:** La sintaxis de JSON es muy similar a la de los objetos literales de JavaScript, lo que la hace fácil de leer y escribir tanto para humanos como para máquinas.
- **Independiente del lenguaje:** no está vinculado a ningún lenguaje de programación específico, por lo que puede ser utilizado por una amplia variedad de lenguajes, incluyendo JavaScript, Python, Java, C++, etc.
- **Estructura flexible:** admite estructuras de datos jerárquicas, como objetos y arrays, lo que lo hace adecuado para representar una amplia gama de información.
- **Fácil de analizar y generar:** Existen bibliotecas y módulos para analizar y generar JSON en la mayoría de los lenguajes de programación

Sintaxis de JSON

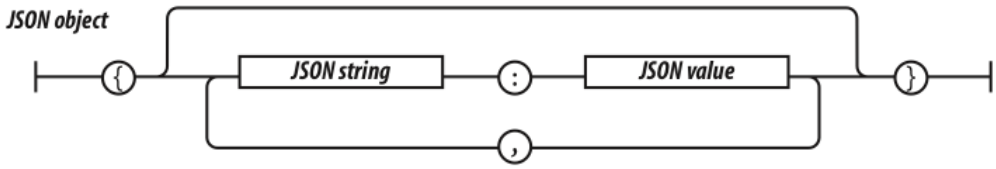
Un objeto JSON es un contenedor, no ordenado de parejas clave/valor. Una clave puede ser un string, y un valor puede ser un valor JSON (tanto un array como un objeto). Los objetos JSON se pueden anidar hasta cualquier profundidad.

En esencia los caracteres que definen un objeto JSON son:

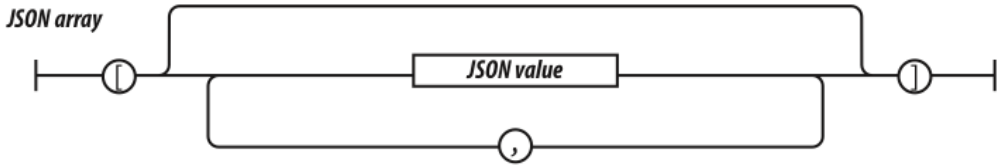
- {} representan un objeto
- [] representan un array
- “” se utilizan para el nombre de las propiedades
- : se utilizan para separar el par propiedad y valor
- , se utilizan para separar los pares propiedad y valor dentro de un objeto, y para separar las propiedades del objeto

La sintaxis de los valores JSON es la siguiente:

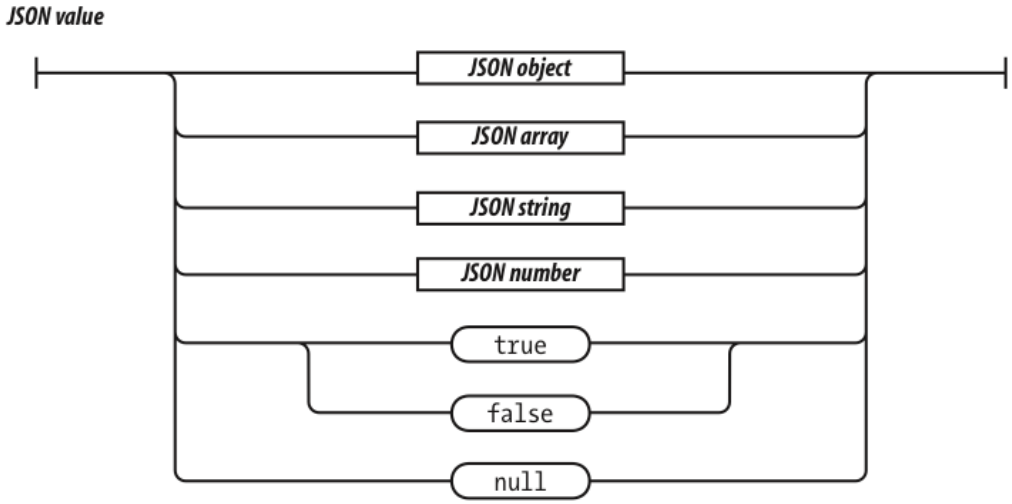
Para definir un objeto



Para definir un array



Tipos de valores admitidos en JSON



Ejemplo objeto sencillo en notación JSON

```
let estudiante={
  "nombre": "Juan Pérez",
  "edad": 30,
  "ciudad": "Madrid",
  "hobbies": ["leer", "cine", "deportes"],
  "mascota": {
    "tipo": "perro",
    "nombre": "Toby"
  }
}
estudiante.hobbies.forEach(h=>console.log(h))
console.log(Object.keys(estudiante.mascota))
```

Ejemplo de un array de objetos en notación JSON

```
let estudiantes = [
  {
    nombre: "Juan Pérez",
    edad: 30,
    ciudad: "Madrid",
    hobbies: ["leer", "cine", "deportes"],
    mascota: {
      tipo: "perro",
      nombre: "Toby"
    }
  },
  {
    nombre: "Ana López",
    edad: 25,
    ciudad: "Barcelona",
    hobbies: ["bailar", "música", "viajar"],
    mascota: {
      tipo: "gato",
      nombre: "Misha"
    }
  },
  {
    nombre: "Carlos García",
    edad: 40,
    ciudad: "Valencia",
    hobbies: ["fotografía", "senderismo", "cocina"],
    mascota: {
      tipo: "pez",
      nombre: "Nemo"
    }
  }
];
```

Métodos JSON

A la hora de trabajar nos interesa utilizar toda la potencia de los objetos para poder acceder a sus propiedades y métodos, pero a su vez a la hora de mandar la información al servidor o de almacenarla internamente en una **cookie** o usando **localSorage** nos interesa que ese objeto sea convertido a una cadena de texto. A su vez también cuando recibamos información desde el servidor o leamos una cookie estaremos leyendo cadenas de texto, pero después de leerlas las querremos utilizar en nuestros scripts como objetos.

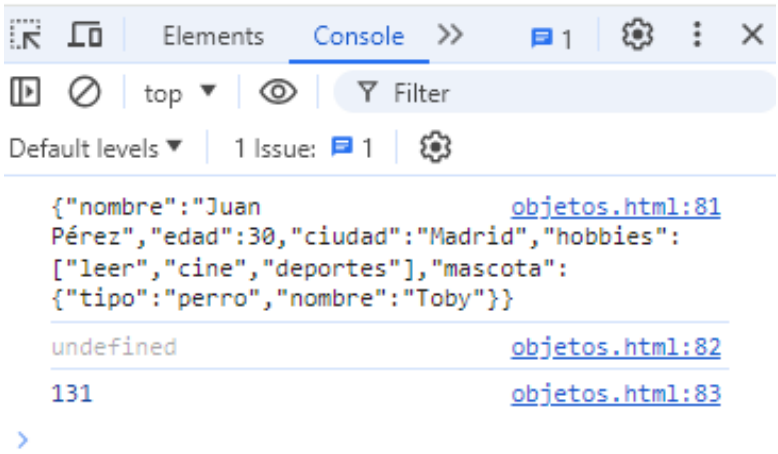
Una vez visto esto se ve claramente que vamos a tener que **transformar nuestros objetos JSON en cadenas de texto y viceversa**. Para ello JavaScript dispone de métodos adecuados.

JSON.stringify

JSON.stringify convierte un objeto en un JSON (cadena de texto con la sintaxis JSON)

```
let estudiante={
  "nombre": "Juan Pérez",
  "edad": 30,
  "ciudad": "Madrid",
  "hobbies": ["leer", "cine", "deportes"],
  "mascota": {
    "tipo": "perro",
    "nombre": "Toby"
  }
}
estudianteString=JSON.stringify(estudiante)
console.log(estudianteString)
console.log(estudianteString.nombre)
```

Hemos convertido el objeto estudiante en una cadena de texto que representa el objeto en notación JSON, pero es un string



JSON es una especificación independiente del lenguaje de **solo datos**, por lo que se omiten algunas propiedades de objeto específicas de JavaScript.

Se eliminan en la conversión a string:

- Propiedades de la función (métodos).
- Propiedades simbólicas.
- Propiedades que almacenan undefined.

JSON.parse

Para decodificar una cadena JSON, necesitamos otro método llamado JSON.parse. Este método lo que realiza es la conversión de una cadena de texto en formato JSON a un objeto.

Después de aplicar este método podremos acceder al objeto como si fuera un objeto creado con propiedades y su acceso será como el visto para objetos utilizando la notación del punto o los corchetes.

```
estudianteTexto='{"nombre":"Pepe","edad":25, "ciudad":"Valencia"}'
estudianteObjeto=JSON.parse(estudianteTexto)
console.log(estudianteObjeto.nombre)
```

4.- Clases

La **Programación Orientada a Objetos (POO)** es un paradigma de programación que organiza el código en torno a **objetos**, que son entidades que encapsulan datos (propiedades) y comportamiento (métodos). La POO con clases en JavaScript nos permite crear objetos reutilizables y modulares, lo que facilita el desarrollo y mantenimiento de aplicaciones complejas.

Las clases fueron introducidas en **ECMAScript 2015**, también conocido como **ES6**. Esta versión del estándar JavaScript fue lanzada en junio de 2015 e introdujo varias características nuevas, incluyendo las clases como una forma sintáctica para definir objetos orientados a objetos.
En versiones anteriores de JavaScript, la POO se implementaba principalmente utilizando **prototipos** y **funciones constructoras**. Si bien era posible crear objetos con propiedades y métodos, la sintaxis era menos intuitiva y no ofrecía las mismas ventajas de modularidad y encapsulamiento que las clases.

Conceptos básicos de la POO con clases en JavaScript

- **Clases:** Las clases son como planos o plantillas que definen la estructura y el comportamiento de los objetos. Contienen la declaración de las propiedades y métodos que todos los objetos de esa clase tendrán en común.

- **Objetos:** Las instancias de una clase se llaman objetos. Cada objeto tiene un conjunto de datos (propiedades) y un conjunto de comportamientos (métodos) definidos por la clase a la que pertenece.
- **Propiedades:** Las propiedades son variables asociadas a un objeto que almacenan datos. Se definen dentro de la clase y se pueden acceder y modificar desde los métodos del objeto.
- **Métodos:** Los métodos son funciones asociadas a un objeto que definen su comportamiento. Se definen dentro de la clase y se pueden llamar desde el propio objeto o desde otros objetos.
- **Herencia:** La herencia permite a una clase heredar las propiedades y métodos de otra clase. Esto te permite crear clases jerárquicas donde las clases hijas pueden especializarse y agregar nuevas funcionalidades a las clases base.
- **Encapsulamiento:** La encapsulación oculta la implementación interna de un objeto y solo expone una interfaz pública a través de sus métodos. Esto promueve la modularidad y protege los datos internos del objeto.
- **Polimorfismo:** El polimorfismo permite que objetos de diferentes clases respondan al mismo mensaje de manera diferente. Esto se logra mediante la sobreescritura de métodos en clases derivadas.

Actualmente para crear una clase en JavaScript, se utiliza la palabra clave **class** seguida del nombre de la clase y las llaves que engloban las propiedades y métodos de la clase que estamos definiendo.

```
class Persona {
  constructor(nombre, apellido, edad) {
    this.nombre = nombre;
    this.apellido = apellido;
    this.edad = edad;
  }

  obtenerNombreCompleto() {
    return `${this.nombre} ${this.apellido}`;
  }

  calcularEdadSiguiente() {
    return this.edad + 1;
  }
}
```

Getters y Setters en Clases de JavaScript

Los **getters** y **setters** son métodos especiales que permiten acceder y modificar propiedades de un objeto de forma controlada y segura. Se utilizan principalmente en clases de JavaScript para encapsular el acceso a datos y definir cómo se pueden leer y modificar desde fuera del objeto.

- Un **getter** es un método que simula el comportamiento de una propiedad al ser leída. Se define utilizando la palabra clave get seguida por el nombre de la propiedad.
- Un **setter** es un método que simula el comportamiento de una propiedad al ser asignada. Se define utilizando la palabra clave set seguida por el nombre de la propiedad

Las **propiedades privadas** en clases de JavaScript son aquellas que solo pueden ser accedidas desde dentro de la clase en la que se definen. Esto permite encapsular datos y protegerlos de modificaciones no deseadas desde fuera del objeto.

Para definir propiedades privadas en JavaScript hay dos sintaxis admitidas:

1 Utilizando convenciones de nomenclatura:

La sintaxis más habitual y extendida para definir una propiedad privada en JavaScript es utilizar el prefijo **-** delante del nombre de la propiedad, se utiliza como prefijo no oficial para indicar que una propiedad no debe ser accedida desde fuera de la clase

```
class Persona {
  _nombre; // Propiedad privada
  _ciudad; // Propiedad privada

  constructor(nombre, ciudad) {
    this._nombre = nombre;
    this._ciudad=ciudad
  }
  get nombre(){
    return this._nombre
  }
  set nombre(nombre){
    this._nombre=nombre
  }
```

```
}
unaPersona=new Persona('Juan','Valencia')
console.log(unaPersona.nombre) // Juan por el getter
unaPersona.nombre='Pepe'       // Cambia nombre por el setter
console.log(unaPersona.nombre)
// devuelve undefined por ser privada y no tener getter
console.log(unaPersona.ciudad)
// deberia generar error por no tener setter
unaPersona.ciudad='Alicante'
console.log(unaPersona.ciudad)
```

¿Por qué asigna el valor de Alicante y además luego puede acceder a él, si no tiene getter ni setter la propiedad ciudad?

2 Utilizando la notación #:

A partir de ECMAScript 2020 (ES2020), se introdujo la notación # para definir propiedades privadas.

```
class Persona {
  #nombre; // Propiedad privada
  #ciudad; // Propiedad privada

  constructor(nombre, ciudad) {
    this.#nombre = nombre;
    this.#ciudad=ciudad
  }

  get nombre(){
    return this.#nombre
  }
}
```

Herencia y Polimorfismo

La herencia en las clases de JavaScript es un mecanismo fundamental para la **reutilización de código** y la **organización de estructuras jerárquicas** de objetos. Permite a una clase (clase hija) heredar las propiedades y métodos de otra clase (clase padre). Esto facilita la creación de clases con características comunes y la extensión de funcionalidades a partir de clases base existentes. La herencia en JavaScript se implementa utilizando la palabra clave extends al momento de definir una clase hija

Las propiedades y métodos estáticos son aquellas que pertenecen a la clase en sí, no a las instancias de la clase. Se definen utilizando la palabra clave static antes de la declaración de la propiedad o método:

El polimorfismo en las clases de JavaScript es la capacidad de que las instancias de diferentes clases respondan al mismo mensaje o método de manera diferente. Esto permite crear estructuras de código más flexibles y adaptables, donde el comportamiento específico se define en cada clase derivada.

```
 class Persona {
  static info(){
    console.log('Metodo de la clase no de una instancia')
  }
  constructor(nombre, apellido) {
    this.nombre = nombre;
    this.apellido = apellido;
  }

  saludar() {
    console.log(`Hola, mi nombre es ${this.nombre} ${this.apellido}`);
  }
}

class Estudiante extends Persona {
  constructor(nombre, apellido, curso) {
    super(nombre, apellido); // Llama al constructor de la clase padre
    this.curso = curso;
  }

  presentarse() {
    super.saludar(); // Llama al método de la clase padre
    console.log(`Soy estudiante del curso ${this.curso}`);
  }

   saludar(){
    console.log(`Me presento como hijo ${this.nombre}`)
  }

}

Persona.info()
const estudiante = new Estudiante("Juan", "Pérez", "Matemáticas");
```

```
estudiante.presentarse(); // Imprime: "Hola, mi nombre es Juan Pérez.
Soy estudiante del curso Matemáticas"
estudiante.saludar()
```