SYMBOL

Los Symbols se utilizan a menudo para añadir claves de propiedades únicas a un objeto que no sean iguales a las claves que cualquier otro código pueda añadir al objeto, y que están ocultas de cualquier mecanismo que otro código utilice normalmente para acceder al objeto.

ejemplo:

let sym1 = Symbol();

let sym2 = Symbol("foo");

let sym3 = Symbol("foo");

Symbol("foo") === Symbol("foo"); // false

A la hora de usar el typeof en un Symbol nos devuelve tipo symbol, esto nos ayuda a identificar estos valores.

typeof Symbol() === "symbol";

typeof Symbol("foo") === "symbol";

typeof Symbol.iterator === "symbol";

SET

Los objetos Set son colecciones de valores. Puede iterar a través de los elementos de un conjunto en orden de inserción. Un valor en un Set solo puede ocurrir una vez; es único en la colección del Set.

Debido a que cada valor en el Set tiene que ser único, se verificará la igualdad de valores. En una versión anterior de la especificación ECMAScript, esto no se basaba en el mismo algoritmo que el utilizado en el operador ===.

//para usar el objeto set primero debemos instanciarlo

const mySet1 = new Set();

//ahora para agregar valores dentro de el usamos el metodo add

mySet1.add(1); // Set [ 1 ]

Para usar este objeto debemos instanciarlo primeramente ya despues podremos adicionar valores dentro de el

mySet1.has(1); // true

Podemos usar “.has” para buscar rápidamente un valor en el set este puede ser usado con otro valor junto a una coma (1, 2) lo que va hacer esto es que va a buscar “1” de la posición “2” en adelante y si este se encuentra retorna true de lo contrario sera false.

mySet1.delete(5); // elimina 5 del conjunto

mySet1.has(5); // false, 5 ha sido eliminado

Si deseamos borrar un registro del set vamos a usar “.delete” como se ve en el ejemplo una vez usado el delete el registro no se encontrara dentro del Set.

mySet1.size; //

Usando “.Size” vamos a poder saber cuantos registros hay dentro del Set

Iteracion de Objeto Set

//La sentencia sentencia for...of ejecuta un bloque de código para cada elemento de un objeto iterable (en-US), como lo son: String, Array, objetos similares a array (por ejemplo, arguments or NodeList), TypedArray, Map (en-US), Set e iterables definidos por el usuario.

for (let item of mySet1) console.log(item)

//For of que recorre Set e imprime cada uno de sus elementos

for (let item of mySet1.keys()) console.log(item)

El método estático Object.keys() devuelve un arreglo de propiedades enumerables propias de un objeto dado.

for (let item of mySet1.values()) console.log(item)

for (let [key, value] of mySet1.entries()) console.log(key)

El método Object.entries() devuelve una matriz de pares propios de una propiedad enumerable [key, value]

//Cada uno de estos metodos sirven para iterar set

Podemos iterar sobre los elementos de set usando for of como se ve en el ejemplo donde vamos a imprimir en pantalla cada uno de estos elementos .

Convertir a Array

const myArr = Array.from(mySet1)

Podemos convertir un set en un array de la siguiente manera

let array = [1, 3, 6, 8, 0, 6, 9, 9, 12, 34, 4, 5, 2, 1]

const set2 = new Set(array);

console.log(set2)

Y para convertir un array en set lo hacemos como el ejemplo, esto no va a servir por ejemplo para sacar o omitir números repetidos.

Incluso se puede usar con Strings

let text = "India";

const mySet3 = new Set(text);

mySet3.size;

// mayúsculas, minúsculas y omisión duplicada

console.log(mySet3)

Por ultimo podemos limpiar el set con el metodo clear().

mySet1.clear();

console.log(mySet1)

Operaciones de conjuntos

Unión: Es la suma de todos los elementos del conjunto A y el conjunto B.

Intersección: Es la parte común de los elementos del conjunto A y el conjunto B.

Diferencia: Son los elementos del conjunto A quitándole los comunes del conjunto B.

Exclusión: Son los elementos del conjunto A y el conjunto B que no están en ambos.

//Set para operaciones de conjuntos

const set1 = new Set([1, 2, 3, 4]);

const set2 = new Set([3, 4, 5, 6, 7]);

//union

const union = new Set([...set1, ...set2])

console.log(union);

//intersección

const intersection = [...set1].filter(element => set2.has(element));

const inter = new Set(intersection)

console.log(inter);

//diferencia

const diferencia = [...set1].filter(element => !set2.has(element));

const dif = new Set(diferencia);

console.log(dif);

//exclusión

const excl1 = [...set1].filter(element => !set2.has(element));

const excl2 = [...set2].filter(element => !set1.has(element));

const exclusion = new Set([...excl1, ...excl2]);

console.log(exclusion)

Se operar con datos complejos

const firstSet = new Set([

{ name: "Javascript" },

{ name: "CSS" },

{ name: "HTML" },

{ name: "SVG" },

{ name: "JSON" }

]);

const secondSet = new Set([

{ name: "Javascript", type: "Node" },

{ name: "PHP" },

{ name: "Go" },

{ name: "Javascript", type: "Deno" },

{ name: "JSON" }

]);

MAPS

Los Map en Javascript son estructuras de datos nativas que permiten implementar una estructura de tipo mapa, es decir, una estructuras donde tiene valores guardados a través de una clave para identificarlos. Comúnmente, esto se denomina pares clave-valor. La principal diferencia es que Map permite claves de cualquier tipo.

son objetos que sirven para almacenar valores asociados a manera de objeto(Es como un objeto primitivo)

El método .set() fija un par clave-valor en el mapa. Se diferencia de add que añade set fija la clave valor en el map

Si usamos .set() para una clave que no existe, se añade al mapa.

Si usamos .set() para una clave que ya existe, la sobreescribe.

const map = new Map();

map.set(5, "cinco");

map.set("A", "letra A");

map.set(5, "five");

//se reescribe el anterior

0:{5 => "five"}

1:{"A" => "letra A"}

Para comprobar si un elemento existe en un mapa, se debe hacer a través de su clave, y se utiliza el método .has().Este método devuelve un , por lo que si existe la clave, nos devolverá true, y en caso contrario, nos devolverá false.

const map = new Map([[1, "uno"], [2, "dos"], [3, "tres"]]);

map.has(2); // true

map.has(34); // false

para eliminar un registro del map podemos usar el método delete este devuelve un valor booleano en caso de eliminarlo devuelve true de lo contrario false.

map.delete(5); //devuelve true

map.delete(1); //devuelve false

vaciar el conjunto map

El método clear nos permite vaciar todos los valores existentes en el conjunto map para dejarlo limpio… a diferencia del delete este método no devuelve nada

map.clear();

map.size; //0 🙂

Convertir en array

(segun lei es mejor usar structured Clone para clonar los elementos anidados y hacer un clonado a profundidad)

const arr = [...structuredClone(map)];

console.log(arr)

Incluso podemos convertirlo en un objeto usando el método Object.from Entries() que transforma una lista de pares con [clave-valor] en un objeto.

const object = Object.fromEntries(map);

WeakSet

Los WeakSet son otro tipo de estructura de conjuntos, muy similar a Set que comparte la similitud de no poder introducir elementos duplicados, sin embargo, tiene algunos matices y diferencias.

(set de baja frecuencia)

Los Weakset a diferencia de los los set no permite usar datos primitivos (como string, number, boolean)

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Set

const set = new Set([1, "A", true]); // OK

const set2 = new Set([{ name: "Manz" }, [2, 30]]); // OK

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ WeakSet

const setw = new WeakSet([1, "A", true]);

// ERROR: Uncaught TypeError: Invalid value used in weak set

const setw2 = new WeakSet([{ name: "Manz" }, [2, 30]]); // OK

Además los weak set usan referencias débiles para los objetos, por lo cual si no se usa sera eliminado.

let element = { name: "Manz" };

const set = new WeakSet([element]);

set; // WeakSet({ { name: "Manz" } })

element = null;

set; // WeakSet({})

El objeto puede que no se elimine de inmediato y la información permanezca. Esto ocurre porque la recolección de basura es un proceso que no ocurre instantáneamente,

WeakMap

Al igual que el set y weakset el map maneja una estructura muy similar llamada WeakMap, la cual también posee algunas diferencias con respecto al map.

la diferencia principal de los Map con los WeakMap es que estos últimos, no permiten utilizar tipos primitivos (number, String, boolean ) como clave, mientras que el Map si lo permite:

// \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Map

const map1 = new Map([[1, "uno"]]); // OK

const map2 = new Map([[{ id: 1, type: "number" }, "uno"]]); // OK

// \_\_\_\_\_\_\_\_\_\_\_\_WeakMap

const mapw1 = new WeakMap([[1, "uno"]]);

// ERROR: Uncaught TypeError: Invalid value used in weak map key

const mapw2 = new WeakMap([[{ id: 1, type: "number" }, "uno"]]); // OK

Iterables

Los objetos iterables son una generalización de arrays. Es un concepto que permite que cualquier objeto pueda ser utilizado en un bucle for..of.

Por supuesto, las matrices o arrays son iterables. Pero hay muchos otros objetos integrados que también lo son. Por ejemplo, las cadenas o strings son iterables también. Como veremos, muchos operadores y métodos se basan en la iterabilidad.

const arr = [1, 2, 3, 4, 5];

for (const num of arr) {

console.log(num);

}

iterators

En JavaScript, un iterador es un objeto que permite recorrer una colección y devolver un valor al terminar. Específicamente, un iterador es un objeto que implementa el protocolo de iteración a través del método next(), el cual devuelve un objeto con dos propiedades:

value

El siguiente valor en la secuencia de iteración.

done

Es true si el último valor en la secuencia ya ha sido consumido. Si value está presente junto con done, es el valor de retorno del iterador. Un iterador se considera ya terminado/finalizado cuando la invocación de next() regresa un objeto donde la propiedad done es verdadero.

Una vez creado, un objeto iterador puede utilizarse explícitamente llamando repetidamente al método next().

function crearIterador(arreglo) {

var siguienteIndice = 0;

return {

next: function () {

return siguienteIndice < arreglo.length

? { value: arreglo[siguienteIndice++], done: false }

: { done: true };

},

};

}

Proxies

Un Proxy es un objeto que envuelve a otro objeto e intercepta operaciones básicas en él, como acceder, asignar y eliminar propiedades. El proxy es un aspecto crucial de JavaScript que permite a los desarrolladores escribir código más versátil y robusto.

Proxy de JavaScript es una capacidad que permite la creación de objetos capaces de modificar y personalizar las operaciones básicas realizadas en otros objetos.

Para establecer un objeto Proxy, se necesitan dos componentes: un objeto de destino y un objeto controlador. El objeto de destino es aquel en el que se van a interceptar las operaciones, mientras que el objeto controlador es responsable de contener las trampas, o métodos, utilizados para capturar estas operaciones.

const target = { name: 'John', age: 25, };

const handler = { get: function(target, prop) { console.log(`Getting property ${prop}`);return target[prop]; }, };

const proxy = new Proxy(target, handler);

console.log(proxy.name); // Getting property name // John