**Symbols**

Los Symbols son un tipo de dato único y primitivo introducido en JavaScript en ECMAScript 6 (también conocido como ES2015). Son utilizados para crear identificadores únicos que no colisionan con ningún otro valor, incluyendo otros Symbols. Aquí tienes un ejemplo simple de cómo se usan los Symbols en JavaScript:

**Ejemplo**

// Crear un Symbol

const simbolo = Symbol();

// Los Symbols pueden tener una descripción opcional

const otroSimbolo = Symbol('Mi descripción');

// Los Symbols siempre son únicos, incluso si tienen la misma descripción

const simbolo1 = Symbol('Mi descripción');

const simbolo2 = Symbol('Mi descripción');

console.log(simbolo1 === simbolo2); // false

// Los Symbols pueden ser utilizados como claves de propiedades de objetos

const miObjeto = {};

// Asignar una propiedad utilizando un Symbol como clave

miObjeto[simbolo] = 'Valor del Symbol';

// Acceder a la propiedad utilizando el Symbol como clave

console.log(miObjeto[simbolo]); // 'Valor del Symbol'

// Los Symbols no son enumerables

console.log(Object.keys(miObjeto)); // []

// Pero se pueden obtener todas las propiedades Symbols de un objeto

console.log(Object.getOwnPropertySymbols(miObjeto)); // [Symbol()]

// También se pueden obtener todas las propiedades (incluyendo Symbols)

console.log(Reflect.ownKeys(miObjeto)); // [Symbol()]

**Sets**

En JavaScript, un "set" es una estructura de datos que permite almacenar colecciones de valores únicos, lo que significa que no puede haber elementos duplicados dentro de un set. Aquí tienes algunas características clave de los sets en JavaScript:

1. **Elementos únicos:** Los sets no permiten elementos duplicados. Si intentas agregar un elemento que ya existe en el set, se ignorará y el set no se modificará.
2. **Iteración en orden de inserción:** A diferencia de los objetos en JavaScript, los sets mantienen el orden de inserción de los elementos. Esto significa que puedes iterar sobre los elementos en el mismo orden en que se agregaron al set.
3. **Tipos de datos permitidos:** Los sets pueden contener cualquier tipo de datos, como números, cadenas, objetos e incluso otras estructuras de datos como arrays y sets.
4. **Operaciones comunes:** Los sets admiten operaciones comunes como agregar elementos, eliminar elementos, verificar si un elemento existe en el set, obtener el tamaño del set y borrar todos los elementos del set.

/ Crear un nuevo set

let miSet = new Set();

// Agregar elementos al set

miSet.add(1);

miSet.add(2);

miSet.add(3);

miSet.add(1); // Este elemento se ignorará, ya que 1 ya está en el set

// Verificar si un elemento existe en el set

console.log(miSet.has(2)); // true

console.log(miSet.has(4)); // false

// Eliminar un elemento del set

miSet.delete(2);

// Obtener el tamaño del set

console.log(miSet.size); // 2

// Iterar sobre los elementos del set

miSet.forEach(function(valor) {

console.log(valor);

});

**Maps**

se refiere a una estructura de datos que permite almacenar pares clave-valor, donde las claves pueden ser de cualquier tipo, incluidos objetos y funciones, y los valores pueden ser de cualquier tipo también. A diferencia de los objetos regulares en JavaScript, los Maps mantienen el orden de inserción de sus elementos, lo que significa que los elementos se recuperan en el orden en que se insertaron

// Crear un nuevo Map

let miMapa = new Map();

// Agregar elementos al Map

miMapa.set('clave1', 'valor1');

miMapa.set('clave2', 'valor2');

// Obtener el valor asociado a una clave

console.log(miMapa.get('clave1')); // Imprimirá "valor1"

// Verificar si una clave está presente en el Map

console.log(miMapa.has('clave2')); // Imprimirá true

// Obtener el tamaño del Map

console.log(miMapa.size); // Imprimirá 2

// Eliminar un elemento del Map

miMapa.delete('clave1');

// Verificar si una clave está presente en el Map después de eliminarla

console.log(miMapa.has('clave1')); // Imprimirá false

**Weaksets**

Los WeakSets en JavaScript son estructuras de datos que te permiten almacenar conjuntos de objetos de manera que no impidan que los objetos que contienen sean eliminados por el recolector de basura si ya no son referenciados en ningún otro lugar de tu código. Esto significa que los objetos almacenados en un WeakSet no impiden que sean eliminados de la memoria si no hay ninguna otra referencia a ellos en tu programa.

Los WeakSets solo pueden contener objetos y no aceptan ningún otro tipo de dato. Los objetos almacenados en un WeakSet deben ser referenciados desde otro lugar en tu código, ya que si el único lugar donde se referencia un objeto es a través de un WeakSet, ese objeto puede ser eliminado por el recolector de basura.

ejemplo básico de cómo crear y usar un WeakSet en JavaScript:

// Crear un nuevo WeakSet

let miWeakSet = new WeakSet();

// Crear algunos objetos

let objeto1 = {};

let objeto2 = {};

let objeto3 = {};

// Agregar objetos al WeakSet

miWeakSet.add(objeto1);

miWeakSet.add(objeto2);

// Verificar si un objeto está presente en el WeakSet

console.log(miWeakSet.has(objeto1)); // Imprimirá true

console.log(miWeakSet.has(objeto3)); // Imprimirá false, ya que objeto3 no está en el WeakSet

// Eliminar un objeto del WeakSet

miWeakSet.delete(objeto1);

// Verificar si un objeto está presente en el WeakSet después de eliminarlo

console.log(miWeakSet.has(objeto1)); // Imprimirá false

Los WeakSets son útiles cuando necesitas almacenar conjuntos de objetos y quieres asegurarte de que los objetos almacenados no impidan que sean eliminados de la memoria si no son necesarios en tu programa. Son especialmente útiles cuando estás trabajando con estructuras de datos donde los objetos almacenados pueden ser eliminados dinámicamente durante la ejecución del programa

**Weakmaps**

Los **WeakMaps** en JavaScript son estructuras de datos que permiten crear mapas de pares clave-valor donde las claves son objetos y los valores pueden ser de cualquier tipo. Al igual que los **WeakSets**, los **WeakMaps** tienen la particularidad de que no impiden que los objetos utilizados como claves sean eliminados por el recolector de basura si ya no son referenciados en ningún otro lugar de tu código.

Esto significa que los objetos utilizados como claves en un **WeakMap** pueden ser eliminados de la memoria si no hay ninguna otra referencia a ellos en tu programa, incluso si todavía existen como claves en el **WeakMap**.

Los **WeakMaps** tienen una serie de ventajas y casos de uso específicos:

1. **Privacidad de datos**: Al utilizar objetos como claves, puedes asociar datos privados a esos objetos sin preocuparte de que estos datos permanezcan en memoria una vez que los objetos dejan de ser utilizados.
2. **Evitar fugas de memoria**: Los **WeakMaps** pueden ser útiles para evitar fugas de memoria al asociar datos con objetos temporales o objetos que pueden ser eliminados durante la vida útil de tu aplicación.
3. **Seguridad**: Al utilizar **WeakMaps**, puedes asegurarte de que ciertos datos solo estén disponibles mientras el objeto clave asociado está presente, lo que puede mejorar la seguridad de tu aplicación.

ejemplo básico de cómo crear y usar un **WeakMap** en JavaScript:

// Crear un nuevo WeakMap

let miWeakMap = new WeakMap();

// Crear algunos objetos

let clave1 = {};

let clave2 = {};

// Asociar datos con las claves en el WeakMap

miWeakMap.set(clave1, "Datos relacionados con clave1");

miWeakMap.set(clave2, "Datos relacionados con clave2");

// Obtener datos asociados con las claves

console.log(miWeakMap.get(clave1)); // Imprimirá "Datos relacionados con clave1"

console.log(miWeakMap.get(clave2)); // Imprimirá "Datos relacionados con clave2"

// Eliminar datos asociados con una clave

miWeakMap.delete(clave1);

// Verificar si una clave está presente en el WeakMap después de eliminarla

console.log(miWeakMap.has(clave1)); // Imprimirá false

**Iterables**

En JavaScript, un "iterable" es un objeto que implementa el protocolo de iteración, lo que significa que puede ser recorrido iterativamente utilizando un bucle **for...of** u otras construcciones que esperan un conjunto de elementos iterables, como **Array.from()** o el constructor de **Set**. Este protocolo de iteración permite que los objetos sean tratados como secuencias de elementos que pueden ser recorridos uno a uno.

Los iterables son ampliamente utilizados en JavaScript para trabajar con colecciones de datos, como matrices, conjuntos y objetos de tipo **Map**. También puedes crear tus propios iterables utilizando el método **Symbol.iterator**, que devuelve un iterador para el objeto.

ejemplo básico de cómo trabajar con iterables en JavaScript:

// Crear un iterable personalizado

let miIterable = {

[Symbol.iterator]: function() {

let i = 0;

let elementos = [1, 2, 3, 4, 5];

return {

next: function() {

return {

value: elementos[i++],

done: i > elementos.length

};

}

};

}

};

// Utilizar el iterable con un bucle for...of

for (let elemento of miIterable) {

console.log(elemento);

}

En este ejemplo, **miIterable** es un objeto que implementa el protocolo de iteración mediante el método **Symbol.iterator**, que devuelve un iterador. El iterador devuelve un objeto con dos propiedades: **value**, que representa el valor del elemento actual, y **done**, que indica si se ha alcanzado el final del iterable.

Los iterables son muy útiles en JavaScript porque permiten una forma genérica y flexible de trabajar con colecciones de datos, lo que facilita la escritura de código más legible y mantenible. Puedes utilizarlos para recorrer arrays, sets, maps y cualquier otro tipo de objeto que implemente el protocolo de iteración.

**Iterators**

Los iteradores en JavaScript son objetos que implementan un protocolo específico que les permite definir una secuencia de elementos y proporcionar un método para iterar sobre esta secuencia. El protocolo de iteración consiste en que el objeto debe tener un método llamado next() que devuelve un objeto con dos propiedades: value, que representa el siguiente valor en la secuencia, y done, que indica si se ha alcanzado el final de la secuencia.

Los iteradores son esenciales para trabajar con estructuras de datos iterables, como arrays, sets, maps y objetos personalizados que implementan el protocolo de iteración. Al utilizar iteradores, puedes recorrer secuencialmente los elementos de una colección de datos de manera eficiente y flexible.

**// Definir un iterable personalizado**

**let miIterable = {**

**elementos: [1, 2, 3, 4, 5],**

**indice: 0,**

**next: function() {**

**return {**

**value: this.elementos[this.indice++],**

**done: this.indice > this.elementos.length**

**};**

**}**

**};**

**// Crear un iterador a partir del iterable**

**let iterador = miIterable[Symbol.iterator]();**

**// Utilizar el iterador para recorrer los elementos del iterable**

**let resultado = iterador.next();**

**while (!resultado.done) {**

**console.log(resultado.value);**

**resultado = iterador.next();**

**}**

En este ejemplo, miIterable es un objeto que implementa el protocolo de iteración mediante la implementación de un método next(). El método Symbol.iterator se utiliza para obtener un iterador a partir del iterable. Luego, el iterador se utiliza para recorrer los elementos del iterable mediante un bucle while, extrayendo cada valor con el método next() y verificando si se ha alcanzado el final de la secuencia con la propiedad done.

**Principio del formulario**

**Proxies**

Los Proxies en JavaScript son objetos que permiten la creación de un intermediario entre un objeto objetivo y el código que interactúa con ese objeto. Este intermediario, el proxy, puede interceptar y controlar el acceso a las propiedades y métodos del objeto objetivo, lo que proporciona una manera flexible de personalizar el comportamiento de los objetos en JavaScript.

Los Proxies son útiles para una variedad de propósitos, incluyendo:

1. **Validación de datos:** Puedes usar un Proxy para validar los valores que se asignan a las propiedades de un objeto, asegurándote de que cumplan ciertos criterios antes de ser almacenados.
2. **Control de acceso:** Puedes restringir o permitir el acceso a ciertas propiedades de un objeto dependiendo de ciertas condiciones, lo que puede ser útil para implementar reglas de seguridad o control de acceso en tu aplicación.
3. **Logging y debugging:** Los Proxies pueden ser utilizados para registrar o interceptar las operaciones realizadas en un objeto, lo que puede ser útil para propósitos de debugging o para realizar un seguimiento del comportamiento de tu aplicación.

**/ Objeto objetivo**

**let persona = {**

**nombre: "Juan",**

**edad: 30**

**};**

**// Crear un Proxy para el objeto persona**

**let proxyPersona = new Proxy(persona, {**

**// Intercepta la lectura de propiedades**

**get: function(target, propiedad, receptor) {**

**console.log(`Acceso a la propiedad "${propiedad}"`);**

**return target[propiedad];**

**},**

**// Intercepta la escritura de propiedades**

**set: function(target, propiedad, valor, receptor) {**

**console.log(`Asignación a la propiedad "${propiedad}"`);**

**if (propiedad === 'edad' && typeof valor !== 'number') {**

**throw new TypeError('La edad debe ser un número');**

**}**

**target[propiedad] = valor;**

**}**

**});**

**// Acceso y modificación de propiedades a través del proxy**

**console.log(proxyPersona.nombre); // Acceso a la propiedad "nombre" - Imprimirá "Juan"**

**proxyPersona.edad = 35; // Asignación a la propiedad "edad"**

**console.log(proxyPersona.edad); // Acceso a la propiedad "edad" - Imprimirá 35**

**proxyPersona.edad = "treinta y cinco"; // Asignación a la propiedad "edad" - Generará un error TypeError**

En este ejemplo, creamos un Proxy para el objeto persona, donde interceptamos las operaciones de lectura y escritura de propiedades. Cada vez que se accede o se asigna una propiedad en el proxy, se imprime un mensaje en la consola, y en el caso de la propiedad 'edad', se valida que el valor sea un número. Esto muestra cómo los Proxies pueden ser utilizados para personalizar y controlar el comportamiento de los objetos en JavaScript.

**Propiedades dinámicas de los objetos**

Las propiedades dinámicas de los objetos en JavaScript son aquellas propiedades que pueden ser añadidas, modificadas o eliminadas durante la ejecución del programa. Esto significa que no estás limitado a definir todas las propiedades de un objeto en el momento de su creación; puedes manipular su estructura y contenido en tiempo de ejecución.

Estas propiedades son útiles en situaciones donde necesitas manejar datos cuya estructura puede variar o evolucionar dinámicamente. Algunos casos de uso incluyen:

1. **Datos dinámicos:** Cuando recibes datos de una fuente externa, como una API, puedes agregar propiedades dinámicamente a un objeto para almacenar esa información de manera flexible.
2. **Personalización de objetos:** Puedes permitir que los usuarios agreguen o eliminen propiedades de un objeto según sus necesidades o preferencias.
3. **Procesamiento de datos dinámicos:** Al manipular datos en tiempo de ejecución, puedes adaptar la estructura de los objetos para realizar operaciones específicas en ellos.

**// Crear un objeto vacío**

**let persona = {};**

**// Agregar propiedades dinámicamente**

**persona.nombre = "Juan";**

**persona.edad = 30;**

**// Modificar una propiedad existente**

**persona.edad = 35;**

**// Eliminar una propiedad**

**delete persona.edad;**

**// Iterar sobre las propiedades del objeto**

**for (let clave in persona) {**

**console.log(clave + ": " + persona[clave]);**

**}**

En este ejemplo, nombre y edad son propiedades dinámicas que se agregan al objeto persona durante la ejecución del programa. Esto proporciona flexibilidad para adaptar la estructura del objeto según sea necesario durante la ejecución del programa.

**Principio del formulario**