Map en JavaScript

El objeto `Map` en JavaScript, introducido en ES6, es una estructura de datos diseñada para almacenar pares clave-valor con mayor flexibilidad y eficiencia que los objetos tradicionales. Su capacidad para aceptar cualquier tipo de clave y su rendimiento consistente lo convierten en una herramienta esencial para proyectos modernos.

En este artículo, exploraremos cómo crear y utilizar un `Map`, profundizando en sus métodos principales y aplicaciones prácticas.

¿Qué es un Map en JavaScript?

Un `Map` es una colección que permite almacenar pares clave-valor, donde cada clave es única. A diferencia de los objetos, que solo permiten claves de tipo cadena o símbolo, un `Map` puede usar cualquier tipo de dato como clave, incluidos objetos y funciones.

1. Permitir claves de cualquier tipo, no solo cadenas.
2. Mantener el orden de inserción, lo cual es crucial en muchas aplicaciones.
3. Proporcionar métodos nativos específicos para la gestión de claves y valores.

¿Cuándo usar Map en lugar de Object?

- Si necesitas claves que no sean cadenas (por ejemplo, números u objetos), `Map` es ideal.
- Si planeas iterar frecuentemente sobre las claves o valores, `Map` es más eficiente y directo.

Cómo crear un Map

```javascript
const mapa = new Map([
  ['nombre', 'Carlos'],
  ['edad', 30],
]);

// Acceder a los valores del Map
const nombre = mapa.get('nombre');
const edad = mapa.get('edad');

console.log(nombre);

console.log(edad);
```

En este ejemplo, el `Map` se inicializa con dos pares clave-valor. Puedes usar el método `get` para recuperar los valores asociados a cada clave.

Métodos principales de Map

Los métodos de `Map` están diseñados para facilitar la manipulación de datos de manera eficiente. Aquí detallamos los más importantes:

1. Método set(key, value)

Este método agrega un nuevo par clave-valor al `Map` o actualiza el valor si la clave ya existe. Es especialmente útil cuando necesitas construir dinámicamente una colección de datos.

```javascript
const mapa = new Map();
mapa.set('clave', 'valor inicial');
mapa.set('clave', 'valor actualizado'); // Sobrescribe el valor anterior

console.log(mapa.get('clave'));
```

Detalle adicional: `set` devuelve el propio `Map`, lo que permite el encadenamiento de métodos:

```
javascript

mapa.set('a', 1).set('b', 2).set('c', 3);
```

2. Método `get(key)`

Permite recuperar el valor asociado a una clave específica. Si la clave no existe en el `Map`, devuelve `undefined`.

```
javascript

const profesion = mapa.get('profesión'); // "Desarrollador"
```

A diferencia de los objetos, no se requiere manipular estructuras complejas para buscar valores; `get` simplifica la consulta directa.

3. Método `has(key)`

Este método verifica si una clave específica existe en el `Map`. Es útil cuando necesitas validar la presencia de un dato antes de usarlo.

```
javascript

const mapa = new Map();
mapa.set('clave', 'valor inicial');

console.log(mapa.has('clave'));

console.log(mapa.has('otraClave'));
```

Con `has`, puedes evitar errores al intentar acceder a claves inexistentes.

4. Método `delete(key)`

El método `delete` elimina un par clave-valor de la colección. Devuelve `true` si la clave existía y fue eliminada, o `false` si no se encontró.

```
javascript
```

```
const mapa = new Map();
mapa.set('clave', 'valor inicial');
mapa.delete('clave');
```

```
console.log(mapa.has('clave'));
```

Ventaja sobre los objetos: En un objeto, eliminar propiedades implica usar la propiedad `delete`, lo que puede ser más propenso a errores en comparación con la claridad de `delete()` en `Map`.

5. Método `clear()`

El método `clear` elimina todos los pares clave-valor de un `Map`. Esto reinicia la colección, dejándola vacía.

```
javascript
```

```
const mapa = new Map();
mapa.set('clave', 'valor inicial');
mapa.clear();
```

```
console.log(mapa.size);
```

¿Cuándo usarlo? Ideal para reiniciar datos temporales o limpiar cachés en aplicaciones.

Propiedades de `Map`

La propiedad `size` devuelve el número total de pares clave-valor en el `Map`. Esto resulta útil para verificar rápidamente cuántos elementos contiene la colección.

```
javascript
```

```
const mapa = new Map([
  ['clave1', 'valor1'],
  ['clave2', 'valor2']
]);
```

```
console.log(mapa.size);
```

A diferencia de los objetos, no es necesario usar métodos adicionales como `Object.keys()` para contar las propiedades. Con `Map`, la propiedad `size` está directamente disponible.

Iteración sobre `Map`

Una de las grandes ventajas de `Map` es su compatibilidad nativa con métodos de iteración.

– Método `keys()`: Devuelve un iterador con todas las claves del `Map`.

```
javascript
```

```
const mapa = new Map([
  ['clave1', 'valor1'],
  ['clave2', 'valor2']
]);

for (const clave of mapa.keys()) {
  console.log(clave);
```

```
}
```

– Método `values()`: Devuelve un iterador con todos los valores del `Map`.

```javascript
const mapa = new Map([
  ['clave1', 'valor1'],
  ['clave2', 'valor2']
]);

for (const valor of mapa.values()) {
  console.log(valor);

}
```

– Método `entries()`: Devuelve un iterador con los pares clave-valor en forma de arrays.

```javascript
const mapa = new Map([
  ['clave1', 'valor1'],
  ['clave2', 'valor2']
]);

for (const [clave, valor] of mapa.entries()) {
  console.log(`${clave}: ${valor}`);

}
```

– Uso con `forEach`: `Map` también soporta el método `forEach` para iterar sobre los pares clave-valor.

```javascript
const mapa = new Map([
  ['clave1', 'valor1'],
  ['clave2', 'valor2']
]);

mapa.forEach((valor, clave) => {
  console.log(`${clave}: ${valor}`);

});
```

Ejemplo Básico: Contador de Palabras

Un `Map` es ideal para contar la cantidad de veces que aparece un elemento en una colección, como palabras en un texto.

```
javascript
```

```javascript
const frase = 'hola mundo hola javascript mundo javascript javascript';
const palabras = frase.split(' ');

const contador = new Map();

palabras.forEach((palabra) => {
  if (contador.has(palabra)) {
    contador.set(palabra, contador.get(palabra) + 1);
  } else {
    contador.set(palabra, 1);
  }
});
```

```
console.log(contador);
```

Casos de Uso de `Map`

1. Almacenar configuraciones: `Map` es ideal para guardar configuraciones con claves dinámicas.
2. Seguimiento de referencias a objetos: Dado que las claves pueden ser objetos, es útil para almacenar datos asociados a instancias específicas.
3. Reemplazo de `Object` cuando se necesita iterabilidad: `Map` facilita la iteración directa sobre claves y valores sin métodos adicionales.

ANEXO:

1. **Convertir a Arrays**
2. **WeakMAP**
3. **Comparativa**

Convertir a Arrays

Si tenemos claro el proceso de desestructuración, podemos convertir los `Map` en array o incluso en object de forma muy sencilla.

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

map.size;                   // 3 (Contiene 3 elementos)

map.constructor.name;       // "Map"

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

entries.constructor.name;   // "Array"

entries;                    // [[1, "uno"], [2, "dos"], [3, "tres"]]
```

Al igual que ocurre con los `Set` y los `WeakSet`, con los `Map` tenemos una estructura denominada `WeakMap`. La idea es la misma: se trata de una estructura derivada, muy similar a los `Map`, pero con algunas diferencias.

Diferencias con los Map

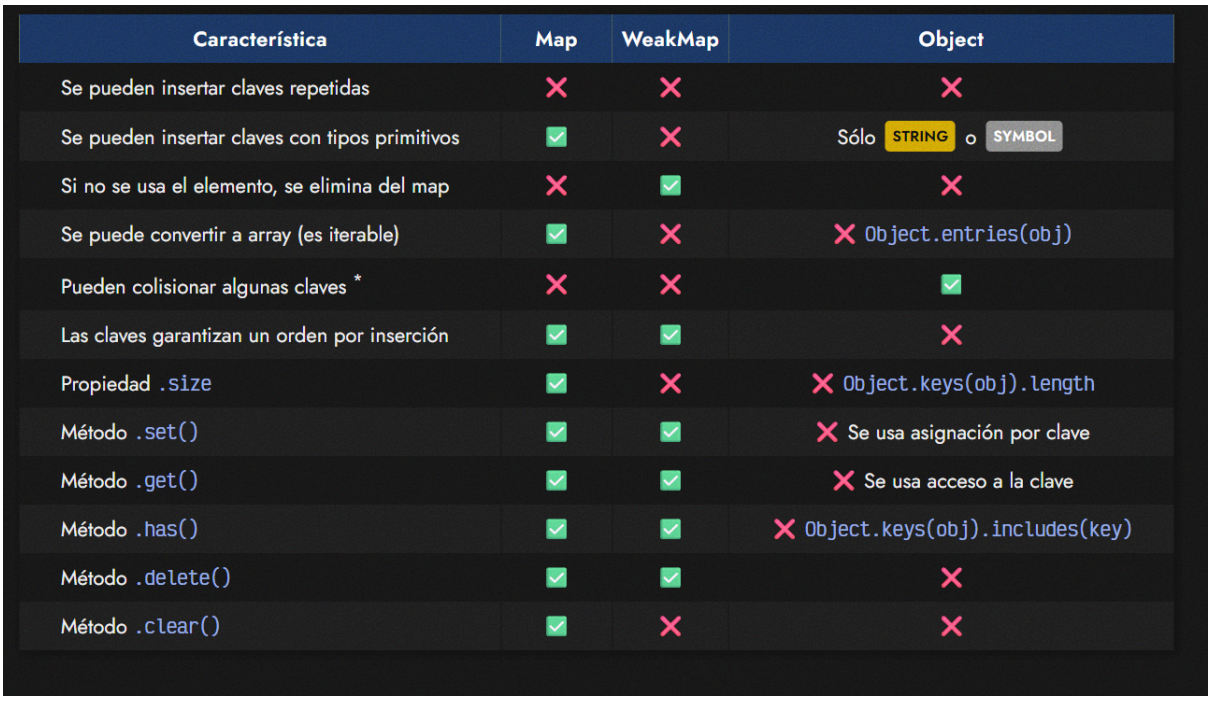
Al margen de algunas diferencias que detallaremos más adelante, la diferencia principal de los `Map` con los `WeakMap` es que estos últimos, no

permiten utilizar tipos primitivos (, ,) como **clave**, mientras que el `Map` si lo permite:

```
// *** Map

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

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

// *** WeakMap

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

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

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

---

3. Comparativa



Característica	Map	WeakMap	Object
Se pueden insertar claves repetidas	✗	✗	✗
Se pueden insertar claves con tipos primitivos	✓	✗	Sólo STRING o SYMBOL
Si no se usa el elemento, se elimina del map	✗	✓	✗
Se puede convertir a array (es iterable)	✓	✗	✗ Object.entries(obj)
Pueden colisionar algunas claves *	✗	✗	✓
Las claves garantizan un orden por inserción	✓	✓	✗
Propiedad .size	✓	✗	✗ Object.keys(obj).length
Método .set()	✓	✓	✗ Se usa asignación por clave
Método .get()	✓	✓	✗ Se usa acceso a la clave
Método .has()	✓	✓	✗ Object.keys(obj).includes(key)
Método .delete()	✓	✓	✗
Método .clear()	✓	✗	✗

Conclusión

El objeto `Map` es una herramienta esencial para gestionar colecciones clave-valor en JavaScript de manera eficiente y ordenada. Su flexibilidad para aceptar cualquier tipo de clave, junto con sus métodos optimizados, lo convierte en una alternativa superior a los objetos tradicionales en muchos casos. Desde manejar datos dinámicos hasta eliminar redundancias, `Map` es una solución versátil para desarrolladores modernos.

Ejercicio tipo examen

Enunciado:

Crea una función llamada `analizarTexto(texto)` que:

1. **Reciba una cadena de texto.**
2. **Cuente cuántas veces aparece cada palabra usando un `Map`.**
3. **Devuelva el `Map` resultante.**
4. **Ignora mayúsculas/minúsculas.**
5. **Elimina signos de puntuación como `.,;:!?`.**