5.6 MAP Y SET

Contenido

1 Map: creación y métodos

Map es una colección de datos identificados por claves, dichas pueden ser de cualquier tipo.

Los métodos y propiedades son:

- `new Map()` – crea el mapa.
- `map.set(clave, valor)` – almacena el valor asociado a la clave.
- `map.get(clave)` – devuelve el valor de la clave. Será `undefined` si la `clave` no existe en map.
- `map.has(clave)` – devuelve `true` si la `clave` existe en map, `false` si no existe.
- `map.delete(clave)` – elimina el valor de la clave.
- `map.clear()` – elimina todo de map.
- `map.size` – tamaño, devuelve la cantidad actual de elementos.

Por ejemplo:

```
let map = new Map();

map.set('1', 'str1');   // un string como clave
map.set(1, 'num1');     // un número como clave
map.set(true, 'bool1'); // un booleano como clave

alert( map.get(1)   ); // 'num1'
alert( map.get('1') ); // 'str1'
```

```
alert( map.size ); // 3
```

Cualquier tipo de clave es posible en un Map.

map[clave] no es la forma correcta de usar Map
Aunque `map[clave]` también funciona (por ejemplo, podemos establecer `map[clave] = 2`), esto es tratar a `map` como un objeto JavaScript simple, lo que implica tener todas las limitaciones correspondientes (que solo se permita string/symbol como clave, etc.).

Por lo tanto, debemos usar los métodos de `Map`: `set`, `get` y demás.

También podemos usar objetos como claves. (todavía no hemos visto los objetos)

De momento veamos un objeto como colección de pares clave:valor (propiedades) donde la clave siempre es un String.

Por ejemplo:

```
let john = { name: "John" };//es un objeto

// para cada usuario, almacenemos el recuento de visitas
let visitsCountMap = new Map();

// john es la clave para el Map
visitsCountMap.set(john, 123);

alert( visitsCountMap.get(john) ); // 123
```

Cómo `Map` compara las claves
Para probar la equivalencia de claves, `Map` utiliza el algoritmo SameValueZero. Es aproximadamente lo mismo que la igualdad estricta `===`, pero la diferencia es que `NaN` se considera igual a `NaN`. Por lo tanto, `NaN` también se puede usar como clave.

Este algoritmo no se puede cambiar ni personalizar.

Encadenamiento
Cada llamada a `map.set` devuelve map en sí, así que podemos "encadenar" las llamadas:

```
map.set('1', 'str1')
   .set(1, 'num1')
   .set(true, 'bool1

');
```

2 Uso de Arrays para crear Mapas

También podemos utilizar un Array donde cada elemento es otro array, en el que el primer elemento es la clave y el segundo el valor de esa clave. Podemos, a partir de dicho array, crear un mapa con las claves y valores del array:

```
const personas=new Map([[1,”Jose”], [2,”María”], [3,”Elena”], [4,”Paco”]]);
```

3 Iteración sobre Map

Para recorrer un `map`, hay 3 métodos:

- `map.keys()` –- devuelve un iterable para las claves.
- `map.values()` -– devuelve un iterable para los valores.
- `map.entries()` -– devuelve un iterable para las entradas `[clave, valor]`. Es el que usa por defecto en `for..of`.

Por ejemplo:

```
let recipeMap = new Map([
  ['pepino', 500],
  ['tomates', 350],
  ['cebollas',50]
]);

// iterando sobre las claves (verduras)
for (let vegetable of recipeMap.keys()) {
  alert(vegetable); // pepino, tomates, cebollas
}

// iterando sobre los valores (precios)
for (let amount of recipeMap.values()) {
  alert(amount); // 500, 350, 50
}

// iterando sobre las entradas [clave, valor]
for (let entry of recipeMap) { // lo mismo que
recipeMap.entries()
  alert(entry); // pepino,500 (etc)
}
```

Se utiliza el orden de inserción.
La iteración va en el mismo orden en que se insertaron los valores. `Map` conserva este orden, a diferencia de un `Objeto` normal.

Además, `Map` tiene un método `forEach` incorporado, similar al de `Array`:

```
// recorre la función para cada par (clave, valor)
recipeMap.forEach( (value, key, map) => {
  alert(`${key}: ${value}`); // pepino: 500 etc
});
```

4 Object.entries: Map desde Objeto

EJEMPLO MAP DESDE ARRAY

Al crear un `Map`, podemos pasarle un array (u otro iterable) con pares clave/valor para la inicialización:

```
// array de [clave, valor]
let map = new Map([
  ['1','str1'],
  [1,'num1'],
  [true,'bool1']
]);
```

```
alert( map.get('1') ); // str1
```

Si tenemos un objeto plano y queremos crear un `Map` a partir de él, podemos usar el método incorporado Object.entries(obj) que devuelve un array de pares clave/valor para un objeto exactamente en ese formato.

Entonces podemos inicializar un map desde un objeto:

```
let obj = {
  name: "John",
  age: 30
};

let map = new Map(Object.entries(obj));

alert( map.get('name') ); // John
```

Aquí, `Object.entries` devuelve el array de pares clave/valor: `[ ["name","John"], ["age", 30]]`. Es lo que necesita `Map`.

5 Object.fromEntries: Objeto desde Map

Acabamos de ver cómo crear un `Map` a partir de un objeto simple con `Object.entries(obj)`.

Existe el método `Object.fromEntries` que hace lo contrario: dado un array de pares [clave, valor], crea un objeto a partir de ellos:

```
let prices = Object.fromEntries([
  ['banana', 1],
  ['orange', 2],
  ['meat', 4]
]);

// ahora prices = { banana: 1, orange: 2, meat: 4 }

alert(prices.orange); // 2
```

Podemos usar `Object.fromEntries` para obtener un objeto desde `Map`.

Ejemplo: almacenamos los datos en un `Map`, pero necesitamos pasarlos a un código de terceros que espera un objeto simple.

Aquí vamos:

```
let map = new Map();
map.set('banana', 1);
map.set('orange', 2);
map.set('meat', 4);

let obj = Object.fromEntries(map.entries()); // hace un objeto
simple (*)

// Hecho!
// obj = { banana: 1, orange: 2, meat: 4 }

alert(obj.orange); // 2
```

Una llamada a `map.entries()` devuelve un array de pares clave/valor, exactamente en el formato correcto para `Object.fromEntries`.

También podríamos acortar la línea `(*)`:

```
let obj = Object.fromEntries(map); // omitimos .entries()
```

Es lo mismo, porque `Object.fromEntries` espera un objeto iterable como argumento. No necesariamente un array. Y la iteración estándar para el `Map` devuelve los mismos pares clave/valor que `map.entries()`. Entonces obtenemos un objeto simple con las mismas claves/valores que `Map`.

6 Set

Un `Set` es una colección de tipo especial: "conjunto de valores" (sin claves), donde cada valor puede aparecer solo una vez.

Sus principales métodos son:

- `new Set(iterable)` – crea el set. El argumento opcional es un objeto iterable (generalmente un array) con valores para inicializarlo.
- `set.add(valor)` – agrega un valor, y devuelve el set en sí.
- `set.delete(valor)` – elimina el valor, y devuelve `true` si el `valor` existía al momento de la llamada; si no, devuelve `false`.
- `set.has(valor)` – devuelve `true` si el valor existe en el set, si no, devuelve `false`.
- `set.clear()` – elimina todo el continido del set.
- `set.size` – es la cantidad de elementos.

La característica principal es que llamadas repetidas de `set.add(valor)` con el mismo valor no hacen nada. Esa es la razón por la cual cada valor aparece en `Set` solo una vez.

Por ejemplo, vienen visitantes y queremos recordarlos a todos. Pero las visitas repetidas no deberían llevar a duplicados. Un visitante debe ser "contado" solo una vez.

`Set` es lo correcto para eso:

```
let set = new Set();

let john = { name: "John" };
let pete = { name: "Pete" };
let mary = { name: "Mary" };

// visitas, algunos usuarios lo hacen varias veces
set.add(john);
set.add(pete);
set.add(mary);
set.add(john);
set.add(mary);

// set solo guarda valores únicos
alert( set.size ); // 3

for (let user of set) {
  alert(user.name); // John (luego Pete y Mary)
}
```

La alternativa a `Set` podría ser un array de usuarios y el código para verificar si hay duplicados en cada inserción usando arr.find. Pero el rendimiento sería mucho peor,

porque este método recorre el array completo comprobando cada elemento. `Set` está optimizado internamente para verificar unicidad.

7 Iteración sobre Set

Podemos recorrer `Set` con `for..of` o usando `forEach`:

```
let set = new Set(["oranges", "apples", "bananas"]);

for (let value of set) alert(value);

// lo mismo que forEach:
set.forEach((value, valueAgain, set) => {
  alert(value);
});
```

Tenga en cuenta algo peculiar: la función callback pasada en forEach tiene 3 argumentos: un valor, luego el mismo valor "valueAgain" y luego el objeto de destino que es set. El mismo valor aparece en los argumentos dos veces.

Eso es por compatibilidad con `Map` donde la función callback tiene tres argumentos. Parece un poco extraño, seguro. Pero en ciertos casos puede ayudar a reemplazar `Map` con `Set` y viceversa con facilidad.

También soporta los mismos métodos que `Map` tiene para los iteradores:

- `set.keys()` – devuelve un iterable para las claves.
- `set.values()` – lo mismo que `set.keys()`, por su compatibilidad con `Map`.
- `set.entries()` – devuelve un iterable para las entradas `[clave, valor]`, por su compatibilidad con `Map`.

8 Resumen

`Map`: es una colección de valores con clave.

Métodos y propiedades:

- `new Map()` – crea el mapa.
- `map.set(clave, valor)` – almacena el valor para la clave.
- `map.get(clave)` – devuelve el valor de la clave: será `undefined` si la `clave` no existe en Map.
- `map.has(clave)` – devuelve`true` si la `clave` existe, y `false` si no existe.
- `map.delete(clave)` – elimina el valor de esa clave.
- `map.clear()` – limpia el Map.
- `map.size` – devuelve la cantidad de elementos en el Map.

La diferencia con un `Objeto` regular:

- Cualquier clave. Los objetos también pueden ser claves.
- Métodos adicionales convenientes, y la propiedad `size`.

`Set`: es una colección de valores únicos.

Métodos y propiedades:

- `new Set(iterable)` – crea el set. Tiene un argumento opcional, un objeto iterable (generalmente un array) de valores para inicializarlo.
- `set.add(valor)` – agrega un valor, devuelve el set en sí.
- `set.delete(valor)` – elimina el valor, devuelve `true` si `valor` existe al momento de la llamada; si no, devuelve `false`.
- `set.has(valor)` – devuelve `true` si el valor existe en el set, si no, devuelve `false`.
- `set.clear()` – elimina todo del set.
- `set.size` – es la cantidad de elementos.

La iteración sobre `Map` y `Set` siempre está en el orden de inserción, por lo que no podemos decir que estas colecciones están desordenadas, pero no podemos reordenar elementos u obtener un elemento directamente por su número.

Tareas

Filtrar miembros únicos del array

Digamos que `arr` es un array.

Cree una función `unique(arr)` que debería devolver un array con elementos únicos de `arr`, sin elementos repetidos.

Por ejemplo:

```
function unique(arr) {
  /* tu código */
}

let values = ["Buba", "Hugo", "Lisa", "Buba",
  "Hugo", "Lisa", "Lisa", "Hugo", "Cora"];

alert( unique(values) ); // Buba, Hugo, Lisa, Cora
```

P.D. Aquí se usan strings, pero pueden ser valores de cualquier tipo.

Pista: Use `Set` para almacenar valores únicos.

Filtrar anagramas

Anagramas son palabras que tienen el mismo número de letras, pero en diferente orden.

Por ejemplo:

```
nap - pan
ear - are - era
cheaters - hectares - teachers
```

Escriba una función `aclean(arr)` que devuelva un array limpio de anagramas.

Por ejemplo:

```
let arr = ["nap", "teachers", "cheaters", "PAN", "ear", "era",
"hectares"];

alert( aclean(arr) ); // "nap,teachers,ear" o "PAN,cheaters,era"
```

Es decir, de cada grupo de anagramas debe quedar solo una palabra, sin importar cual.

Pista: dividir cada palabra en letras, ordenarla y luego utilizarla como clave en un map, si la clave existe no se añadirá de nuevo al map.

Claves iterables

Nos gustaría obtener un array de `map.keys()` en una variable y luego aplicarle métodos específicos de array, ej. `.push`.

Este código no funciona:

```
let map = new Map();

map.set("name", "Cora");

let keys = map.keys();

// Error: keys.push no es una función
keys.push("Morfero");
```

¿Por qué? ¿Cómo podemos arreglar el código para que funcione `keys.push`?