UT03.6: Colecciones

Contenido

1. Introducción

Una vez aprendido los conceptos básicos de arrays vamos a profundizar en los métodos más avanzados como son `find()` o `filter()`, junto con nuevas formas de iteración sobre ellos. La versión ES6 dispone de nuevos objetos para colección datos como son `set` y `map`. Estudiaremos sus métodos específicos.

1.1. Proyecto registro de ventas

Para aplicar todos los conceptos que vamos a ver sobre colecciones, vamos a crear una página que muestre el registro de ventas de nuestra tienda mediante gráficas. En nuestra web tenemos algunos archivos para implementar este proyecto:

- En el directorio *sales* hemos creado la página *sales.html* destinada a contener el registro de ventas. Esta página dispone del esqueleto HTML necesario para implementar el registro.
- En *css* tenemos *graph.css* con el estilo personalizado de la página.
- En *js* tenemos *pageChart.js* donde ubicaremos el código para implementar la funcionalidad. De momento está vacío.

Para la gestión de gráficos vamos a utilizar el framework **chart.js** disponible en https://www.chartjs.org/. Para importar el framework lo hacemos mediante su ubicación en **cdnjs** al final del `body` de la página.

```
<script
src="https://cdnjs.cloudflare.com/ajax/libs/Chart.js/2.9.4/Chart.bundle.m
in.js"  integrity="sha512-
SuxO9djzjML6b9w9/I07IWnLnQhgyYVSpHZx0JV97kGBfTIsUYlWflyuW4ypnvhBrslz1yJ3R
+S14fdCWmSmSA=="  crossorigin="anonymous"></script>
<script src="./js/RegistroVentas/pageChart.js"></script>
```

2. Array

Partimos de un array de objetos literales sobre el cual realizaremos gran parte de los ejemplos propuestos.

```
const computers = [
  {
    computerID: 134,
    brand: 'HP',
    model: 'EliteBook',
    memory: 16,
  },
  {
    computerID: 14,
    brand: 'HP',
    model: 'EliteBook',
    memory: 32,
  },
  {
    computerID: 456,
    brand: 'HP',
    model: 'Pavilion',
    memory: 16,
  },
];
```

2.1. Uso de indexOf()

Este método nos permite obtener la posición en un *array* de un dato primitivo, o devuelve -1 en caso de no encontrase en el *array*. Podemos invocar el método con un segundo argumento indicando la posición a partir de la que queremos buscar.

La función `findNumberPositions()` recibe como argumento un array y un elemento para buscar en el array. Mantenemos un bucle *while* para continuar buscando elementos mientras nos devuelva posiciones el método, y las guardamos en el array de retorno.

```
function findNumberPositions(array, elem) {
  const indexes = [];
  let index = array.indexOf(elem);
  while (index != -1) {
    indexes.push(index);
    index = array.indexOf(elem, index + 1);
  }
  return indexes;
}
```

Pasamos a invocar la función pasando un array de ejemplo. Obtenemos las posiciones donde se encuentra el número 32.

```
let numbers = [32, -5, 66, 32, 23, 14, 32, 16];
$$result.log(numbers.toString()); // 32,-5,66,32,23,14,32,16
$$result.log(findNumberPositions(numbers, 32).toString()); // 0,3,6
```

El método `lastIndexOf()` funciona exactamente igual que el anterior, pero la búsqueda comienza desde el final del array.

Veamos otro ejemplo. La siguiente función permite añadir un elemento nuevo en el array, solo si no existía previamente el elemento, por lo que funcionará como un conjunto. Hacemos el chequeo con `indexOf()`.

```
function addNumberCollection(array, elem) {
  if (array.indexOf(elem) === -1) {
    array.push(elem);
  }
  return array;
}
```

Comprobamos la función intentando añadir el número 32, al existir no es añadido, mientras que el 1 no pertenece al array y si que es añadido.

```
numbers = [32, -5, 66, 23, 14, 16];
$$result.log(numbers.toString()); // 32,-5,66,23,14,16
addNumberCollection(numbers, 32);
$$result.log(numbers.toString()); // 32,-5,66,23,14,16
addNumberCollection(numbers, 1);
$$result.log(numbers.toString()); // 32,-5,66,23,14,16,1
```

Por último, creamos una función con doble funcionalidad según el número de argumentos recibidos:

- Con dos argumentos, eliminamos el segundo argumento del array que actúa como colección con `splice()`.
- Con tres argumentos, sustituimos el segundo argumento en el array por el tercero.

La comprobación de si tenemos tercer argumento la hacemos con un ternario, teniendo en cuenta que el valor 0 devolvería false, por lo que hay que comprobarlo también.

```
function updateNumberCollection(array, elem, newElement) {
  const index = array.indexOf(elem);
  newElement = Number(newElement);
  if (index >= 0) {
    (newElement || newElement === 0)
      ? array.splice(index, 1, newElement)
      : array.splice(index, 1);
  }
  return array;
}
```

Debemos notar que si en el parámetro de la posición en el método `splice()` tenemos un valor negativo, la sustitución o el borrado se hace comenzando por el final.

Actualizamos el ejemplo con el borrado del número 66 en primer lugar, pasamos a intentar sustituir el 66 por un 1 pero ya no existe, por lo que el resultado es el mismo, y por último intercambiamos el -5 por un 0.

```
updateNumberCollection(numbers, 66);
$$result.log(numbers.toString()); // 32,-5,23,14,16,1
updateNumberCollection(numbers, 23, 2);
$$result.log(numbers.toString()); // 32,-5,2,14,16,1
updateNumberCollection(numbers, -5, 0);
$$result.log(numbers.toString()); // 32,0,23,14,16,1
```

Si queremos rediseñar la función anterior para que modifique o borre todas las ocurrencias de un elemento concreto del array, necesitamos un bucle *while*.

```
function updateAllNumbersCollection(array, elem, newElement) {
  let index = array.indexOf(elem);
  newElement = Number(newElement);
  while (index > -1) {
    (newElement || newElement === 0)
      ? array.splice(index, 1, newElement)
      : array.splice(index, 1);
    index = array.indexOf(elem);
  }
}
```

Testeamos el código anterior modificando el número 32 por el 15 y lo volvemos a restaurar.

```
numbers = [32, -5, 66, 32, 23, 14, 32, 16];
$$result.log(numbers.toString()); // 32,-5,66,32,23,14,32,16
updateAllNumbersCollection(numbers, 32, 15);
$$result.log(numbers.toString()); // 15,-5,66,15,23,14,15,16
updateAllNumbersCollection(numbers, 15, 32);
$$result.log(numbers.toString()); // 32,-5,66,32,23,14,32,16
```

2.2. Función *callback* en métodos de array

Una función *callback* es una función que es pasada como argumento a otra función externa o método. Este mecanismo se utiliza principalmente para que código de terceros, en este caso los métodos de array, puedan invocar nuestro código. Otra razón para el uso de funciones *callback* es con motivo de sincronización como por ejemplo en llamada AJAX. Para más información https://developer.mozilla.org/es/docs/Glossary/Callback_function.

Los métodos `find()` y `findIndex()` son ejemplo de métodos que utilizan funciones *callback*. Su función es muy similar y devuelven el primer elemento o su posición que cumplan una condición determinada por la función *callback*. En ambos métodos, se invocará la función de *callback* pasando como parámetro cada uno de los elementos del array hasta que la función devuelva *true*. Veamos un ejemplo.

El método `indexOf()` solo lo podemos utilizar para buscar tipos primitivos. Con un array de objetos, la comparación se haría por direcciones de memoria, pero nos podría interesar comparar por el contenido de los objetos.

Partimos del array de objetos creado en la introducción.

Vamos a buscar el objeto *computer* que tenga como propiedad *memory* mayor de 16. Como vemos en la invocación de find(), la función de *callback* devolverá un *booleano* en cada

invocación para determinar si el objeto cumple con nuestra condición, no implicando que tenga que ser una condición de igualdad y pudiendo ser una condición más compleja.

```
function findHighPerformanceComputer() {
  const computer = computers.find(function(elem){
    return elem.memory > 16;
  });
  $$result.log(computer); // ID: 14 Brand: HP Memory: 32
}
```

Las funciones de *callback* pueden invocarse con dos argumentos optativos que nos permiten alterar el contenido del array.

- La posición del elemento en segundo lugar.
- El propio array en tercer lugar.

Creamos una función que dado el array de *computers* modifique cada equipo del array con memoria de 16GB por un nuevo objeto *computer* con las prestaciones indicadas, pero conservando el identificador, la marca y el modelo. Además, queremos conocer los identificadores de los equipos modificados.

Como vemos en la función, iteramos sobre *localComputers* con un `forEach()`. La función *callback* será invocada con los tres argumentos porque queremos modificar el array en su interior con los elementos que cumplan la condición. En este ejemplo jugamos con el ámbito de las funciones, ya que la función *callback* al ser interna tiene acceso las variables definidas en la función contenedora.

```
function updateMemoryV1() {
  const localComputers = [...computers]; // Copia del array
  const updatedIds = []; // Array para obtener Ids modificados
  const memory = 16;
  // Objeto sustituto para intercambiar en el array original
  const newComputer = {
    memory: 24,
    SSD: 1024,
  };
  // Iteración sobre el array original
  localComputers.forEach((elem, index, array) => {
    // Modificamos el array origininal si se cumple la condición
    if (elem.memory === memory) { // callback puede acceder a memory por
ser una función interna
      // El parámetro array referencia el array original
      array[index] = { // Creamos un objeto nuevo
        computerID: elem.computerID,
        brand: elem.brand,
        model: elem.model,
        memory: newComputer.memory, // callback puede acceder por ser una
función interna
        SSD: newComputer.SSD,
      };
      // Guardamos el array.
      // callback puede acceder por ser una función interna
      updatedIds.push(elem.computerID);
    }
  });
  $$result.log(updatedIds); // 134,456
}
```

El ejemplo anterior es la forma más habitual de utilizar una función de *callback* y es utilizando una función anónima instanciada en el momento de la invocación del método `forEach()`. Como función de *callback* podemos pasar una función externa. En este caso tendremos un nuevo problema, que la función de *callback* no tendrá acceso a las variables locales por estar en ámbitos diferentes.

Los métodos que utilizan funciones de *callback* pueden opcionalmente pasar un segundo argumento denominado **argumento this**, el cual es referenciado en la función de *callback* como this, y es pasado a la función en cada una de las invocaciones que hace forEach(). Hemos rediseñado el ejemplo anterior con una función externa changeComputer(), la cual hace uso de la referencia this pasada como argumento. Al hacerlo como un objeto literal, podemos pasar varios datos a la función de *callback*, siendo uno de ellos una referencia, por lo que cualquier cambio realizado en la función será permanente.

```
function changeComputer(elem, index, array) {
  // El argumento this puede ser utilizado en la invocación del callback
  if (elem.memory === this.memory) {
    array[index] = {
      computerID: elem.computerID,
      brand: elem.brand,
      model: elem.model,
      memory: this.newComputer.memory,
      SSD: this.newComputer.SSD,
    };
    // this.updatedIds es una referencia
    this.updatedIds.push(elem.computerID);
  }
}
function updateMemoryV2() {
  const localComputers = [...computers];
  const updatedIds = [];
  const memory = 16;
  const newComputer = {
    memory: 24,
    SSD: 1024,
  };
  localComputers.forEach(changeComputer, { // Argumento this
    updatedIds, // El array es una referencia, puede ser modificado
    memory,
    newComputer,
  });
  $$result.log(updatedIds); // 134,456
}
```

2.3. Mapear un array

El método `map()` lo utilizamos para iterar sobre un array utilizando una función *callback* para crear un nuevo array basado en el anterior. Este nuevo array no altera el array original, a no ser que lo deseemos. En este caso, la función *callback* devolverá cada objeto que compondrá el nuevo array.

Vamos a repetir el ejercicio anterior, pero mapeando el array a un nuevo array. Para simplificar, no almacenaremos los identificadores de los equipos modificados. Debemos notar, que el array original *computers* no será modificado, por lo que no necesitamos hacer una copia del mismo, y por la misma razón no tenemos que pasar loas argumentos opcionales en la función *callback*.

```
function mapArray() {
  const updatedIds = [];
  const memory = 16;
  const newComputer = {
    memory: 24,
    SSD: 1024,
  };
  const newComputers = computers.map((elem) => {
    if (elem.memory === memory) {
      return {
        computerID: elem.computerID,
        brand: elem.brand,
        model: elem.model,
        memory: newComputer.memory,
        SSD: newComputer.SSD,
      };
    }
    return elem;
  });
  newComputers.forEach((elem) => {
    $$result.log(elem);
  }); // 134, 14, 456
}
```

2.4. Ordenar un array

El método `sort()` ordena un array utilizando una función *callback* que permita comparar dos elementos del array. La función devuelve un *number* para indicar qué elemento es mayor. Sea *elemA* el primer parámetro de la función y *elemB* el segundo parámetro:

- Si la función devuelve un *number* menor que 0, *elemA* es más pequeño e irá primero en el array.
- Si el valor es 0, los elementos serán iguales y no se intercambiarán las posiciones.
- Si el valor mayor que 0, indica que *elemB* será el más pequeño e irá primero en el array.

Ordenamos el array *computers* utilizando el identificador, para ello utilizamos una función de comparación restando los identificadores.

```
function sortComputers() {
  const localComputers = [...computers];
  localComputers.sort((elemA, elemB) => elemA.computerID -
elemB.computerID);
  localComputers.forEach((elem) => {
    $$result.log(elem.computerID);
  });// 14,134,456
}
```

Prueba a ordenar de mayor a menor por el identificador.

En el siguiente ejemplo vemos como ordenar por la propiedad *model*. El método `localeCompare()` permite comparar *string* teniendo en cuenta los caracteres locales, en caso en español, permite comparar vocales con y sin tilde, así como el carácter ñ.

```
function sortComputersByModel(){
  let localComputers = [...computers];
  localComputers.sort(function(elemA,elemB){
    return elemA.model.localeCompare(elemB.model);
  });
  localComputers.forEach(function(elem){
    $$result.log(elem.model);
  });//Pavilion,EliteBook,EliteBook
}
```

2.5. Filtrado de elementos

El método `filter()` permite seleccionar los objetos de un array en base a una función *callback*. La función devuelve un *true* o *false* para indicar si seleccionamos el elemento o no.

En la siguiente función, a partir de un array de enteros, se filtran los pares y los impares.

```
function evenAndOddNumbers() {
  const numbers = [32, -5, 66, 32, 23, 14, 32, 16];
  const evenNumbers = numbers.filter((elem) => !(elem % 2));
  const oddNumbers = numbers.filter((elem) => (elem % 2));

  $$result.log(evenNumbers.toString()); // 32,66,32,14,32,16
  $$result.log(oddNumbers.toString()); // -5,23
}
```

Podemos utilizar el método `filter()` para eliminar no deseados en el array. En el ejemplo tenemos dos arrays predefinidos, uno con elementos falsos como por ejemplo undefined, NaN o 0 y otro array con una mezcla de elementos finitos e infinitos.

En el primer array podemos filtrar los elementos no deseados aplicando la función `Boolean()` que realiza el casting de cada elemento a un booleano. En el segundo aplicamos `Number.isFinite()` para obtener solo los elementos finitos del array.

```
function removeArrayElements() {
  let falsyElementsArray = ['text', 0, '', undefined, 'green', null, 5,
false, NaN];
  falsyElementsArray = falsyElementsArray.filter(Boolean);

  let finitesNumbersArray = [1, 2, , 3, , 3, , , 0, , , 4, , 4, , 5, , 6,
NaN];
  finitesNumbersArray = finitesNumbersArray.filter(Number.isFinite);

  $$result.log(falsyElementsArray.toString()); // text,green,5
  $$result.log(finitesNumbersArray.toString()); // 1,2,3,3,0,4,4,5,6
}
```

2.6. Todos o algunos de los elementos

Podemos chequear si todos los elementos de un array o algunos elementos del array cumplen una condición determinada a través de los métodos `every()` y `some()` respectivamente.

Disponemos de un array de usuarios con su nombre y su fecha de nacimiento. Vamos a comprobar si todos o algunos son mayores de edad. La función isOlderLegalAge() recibe un objeto *usuario* para comparar su fecha de nacimiento con la de hoy hace 18 años, la cual es utilizada como función *callback* por los métodos `every()` y `some()`.

```
function isOlderLegalAge(user) {
  const today = new Date();
  today.setFullYear(today.getFullYear() - 18);
  return (user.birth < today);
}

function everyAndSomeElements() {
  const users = [
    { username: 'user1', birth: new Date(1999, 7, 3) },
    { username: 'user2', birth: new Date(1998, 2, 23) },
    { username: 'user3', birth: new Date(2010, 6, 19) },
    { username: 'user4', birth: new Date(1995, 10, 12) },
  ];
  $$result.log(users.every(isOlderLegalAge)); // false
  $$result.log(users.some(isOlderLegalAge)); // true
}
```

2.7. Asignación desestructurando un array

Podemos asignar valores a un conjunto de variables desestructurando un array. Creamos un array con los códigos postales de cada provincia, el cual vamos a asignar a un array con los nombres de las provincias en forma de variable. Cada variable queda asignada con su correspondiente código postal, y podemos utilizarlas independientemente del array.

```
function destructuringArray() {
  const postalCodeArray = ['01', '02', '03', '04', '05', '06', '07',
'08', '09', '10', '11', '12', '13', '14', '15', '16', '17', '18', '19',
'20', '21', '22', '23', '24', '25', '26', '27', '28', '29', '30', '31',
'32', '33', '34', '35', '36', '37', '38', '39', '40', '41', '42', '43',
'44', '45', '46', '47', '48', '49', '50', '51', '52'];
  const [Alava, Albacete, Alicante, Almeria, Avila, Badajoz, Baleares,
Barcelona, Burgos, Caceres, Cadiz, Castellon, CiudadReal, Cordoba,
Coruna, Cuenca, Gerona, Granada, Guadalajara, Guipuzcoa, Huelva, Huesca,
Jaen, Leon, Lerida, LaRioja, Lugo, Madrid, Malaga, Murcia, Navarra,
Orense, Asturias, Palencia, LasPalmas, Pontevedra, Salamanca,
SantaCruzdeTenerife, Cantabria, Segovia, Sevilla, Soria, Tarragona,
Teruel, Toledo, Valencia, Valladolid, Vizcaya, Zamora, Zaragoza, Ceuta,
Melilla] = postalCodeArray;
  $$result.log(Albacete); // 02
  $$result.log(CiudadReal); // 13
  $$result.log(Cuenca); // 16
  $$result.log(Guadalajara); // 19
  $$result.log(Toledo); // 45
}
```

3. Trabajando con Array

Empezamos a implementar nuestro proyecto. En *sales.html* tenemos dos elementos `canvas` identificados con un id que nos permitirán generar los gráficos, uno para las ventas mensuales y el otro por departamentos.

```
<!-- Charts -->
<div class="row mt-3 mb-3">
    <div class="col-12 col-md-6 mb-4 mb-md-0">
        <div class="graphSec px-2">
            <h4 class="text-center py-3">Ventas mensuales</h4>
            <canvas id="monthlySales" width="400" height="300"></canvas>
        </div>
    </div>
    <div class="col-12 col-md-6">
        <div class="graphSec px-2">
            <h4 class="text-center py-3">Departamento de ventas</h4>
            <canvas id="deptSales" width="400" height="300"></canvas>
        </div>
    </div>
</div>
```

3.1. Primer gráfico

En primer lugar, vamos a crear las constantes con los objetos DOM que vamos a necesitar. Necesitamos acceder a los contextos de los dos elementos `canvas` en formato 2D para poder dibujar los gráficos. También tenemos un elemento `h1` destinado a mostrar los ingresos anuales.

```
const monthCtx =
document.getElementById('monthlySales').getContext('2d');
const deptCtx = document.getElementById('deptSales').getContext('2d');
const yearlyLabel = document.getElementById('yearlyTotal');
```

Definimos las variables con los datos que vamos a mostrar en cada gráfico, básicamente, por cada gráfico tenemos un array con los datos y otro con las etiquetas a mostrar. Además, tendremos una variable *number* para el total de ventas anuales.

```
//Variables
const monthSales = Array.of(6500, 3250, 4000);
const monthLabels = Array.of('Octubre', 'Noviembre', 'Diciembre');
const deptSales = Array.of(12, 9, 7, 3);
const deptLabels = Array.of('Cámara', 'Móvil', 'Portátil', 'Tablet');
const yearlyTotal = 0;
```

Creamos un gráfico de barras para las ventas mensuales. A través del operador `new` creamos un objeto `Chart`. El constructor recibe dos argumentos, por un lado, el contexto del elemento `canvas` donde mostrar el gráfico, y un objeto literal con la configuración del gráfico. Entre las propiedades del objeto literal tenemos:

- `type`: con el tipo de gráfico. En este caso de barras.

- `data`: con la información a mostrar. Esta propiedad es un objeto con las propiedades:
  - `labels`: Recibe el array con las etiquetas a mostrar.
  - `datasets`: Array con el conjunto de datos a mostrar. Los gráficos de barras admiten varias variables. Su contenido son objetos con:
    - `label`: Etiqueta de la variable.
    - `data`: array con los datos de la variable.
    - `backgroundColor`: Colores de cada variable.

```
// Gráfico de Barras
// Bar
let monthlySalesChart = new Chart(monthCtx, {
  type: 'bar',
  data: {
    labels: monthLabels,
    datasets: [{
      label: 'Número de ventas',
      data: monthSales,
      backgroundColor: [
        'rgba(238, 184, 104, 1)',
        'rgba(75, 166, 223, 1)',
        'rgba(239, 118, 122, 1)',
      ],
      borderWidth: 0
    }
  ]},
  options: {
    scales: { yAxes: [{ ticks: { beginAtZero: true }}] }
  }
});
```

Creamos un gráfico de sectores para las ventas por departamentos, aplicando los arrays de datos y etiquetas correspondientes.

```
// Pie
const deptSalesChart = new Chart(deptCtx, {
  type: 'pie',
  data: {
    labels: deptLabels,
    datasets: [{
      label: 'Número de ventas',
      data: deptSales,
      backgroundColor: [
        'rgba(238, 184, 104, 1)',
        'rgba(75, 166, 223, 1)',
        'rgba(239, 118, 122, 1)',
        'rgba(40, 167, 69, 1)',
      ],
      borderWidth: 0,
    }],
  },
  options: {},
});
```

3.2. Cálculo de totales con operador de propagación

Vamos a crear una función para calcular la suma de facturación de cada mes a partir de 3 parámetros.

```
/* Calculo de totales */
function addYearlyTotal(a, b, c){
  return a + b + c;
}
```

Si queremos realizar la suma total a partir de los valores recogidos en el array de ventas mensuales podemos utilizar el operador de propagación en la invocación de la función. Asignamos el valor recogido en la etiqueta de totales.

```
yearlyTotal = addYearlyTotal(...monthSales);
yearlyLabel.innerHTML = yearlyTotal + "€";
```

3.3. Encontrar un elemento en un array

Queremos buscar la primera venta del array que esté por encima de cinco mil y su posición, para ello podemos utilizar los métodos `find()` o `findIndex()`, aunque lo vamos a hacer con una única invocación.

Definimos un botón en la página para que podamos invocar la funcionalidad.

```
<button id="bSalesOver5000" type="button" class="btn">
    <i class="bi bi-search"></i> Ventas > 5000
</button>
```

Declaramos la variable con el objeto DOM del botón.

```
const bSalesOver5000 = document.getElementById('bSalesOver5000');
```

Definimos una función con dos variables, *position* con un valor -1, y *quantity* que le asignaremos con el elemento de array que cumpla la condición de la función de *callback*. Para devolver la posición del elemento, jugamos con el ámbito de las funciones, ya que desde *callback* tenemos acceso a las variables de la función que la contiene. Por último, asignamos la función al evento *click* del botón para que actúe como manejador de eventos.

```
/* Ventas por encima de 5000 */
function findOver5000(){
  let position = -1;
  let quantity = monthSales.find(function(elem, index){
    if (elem > 5000){
      position = index;
      return true;
    }
    return false;
  });
  alert("Cantidad: " + quantity + " Posición: " + position);
}
bSalesOver5000.addEventListener('click', findOver5000);
```

3.4. Reducir un array a un solo valor

Vamos a rehacer el cálculo de ventas totales para que no dependan de un número estático de parámetros. Eliminamos la función con el operador de propagación para calcular la suma del total de ventas, así como su invocación y asignación a la variable *yearlyTotal*.

Creamos una función que itere sobre cada elemento del array de ventas mensuales con un método `reduce()`, el cual utiliza como primer parámetro una función *callback* para simplificar un array a un solo valor. El método tiene dos parámetros obligatorios, el valor acumulado creado en cada llamada, y el valor actual. La función devuelve el valor acumulado.

En el segundo parámetro del método `reduce()` asignamos el valor inicial del valor acumulado, en este caso 0 porque vamos a hacer una suma de sus valores.

```
function initMonthlyTotalSales() {
  yearlyLabel.innerHTML = `${monthSales.reduce((count, value) => count +
value, 0)}€`;
}
initMonthlyTotalSales();
```

3.5. Reinicializar un array

Necesitamos restaurar un array a sus valores por defecto. En primer lugar creamos un botón en la página para invocar la funcionalidad.

```
<button id="bReset" type="button" class="btn">
    <i class="bi bi-x-square"></i> Resetear
</button>
```

Creamos la constante con el objeto DOM que hace referencia al elemento.

```
const bReset = document.getElementById('bReset');
```

Disponemos del método `fill()` que asigna un valor estático a todos los elementos de un array, aunque podemos configurar por parámetro en qué posición empezar y en qué posición terminar. En la siguiente función invocamos el método sobre el array de ventas mensuales, pasando como argumento el valor 0, con lo que todos sus elementos lo tomarán.

Para aplicar los cambios al gráfico, utilizamos la variable con sus instancia e invocamos el método `update()` y a `initMonthlyTotalSales()` para inicializar el total de ventas mensuales.

```
function resetMonthlySales(){
  monthSales.fill(0);
  monthlySalesChart.update();
  initMonthlyTotalSales();
}
```

Para finalizar asignamos el manejador de eventos al objeto DOM.

```
bReset.addEventListener('click',resetMonthlySales);
```

4. Set

Una colección almacena y estructura una gran cantidad de tipos de datos para un fácil acceso, proporcionando métodos para recuperar esa información.

Los objetos `Set` son colecciones de valores sobre las que podemos iterar para recuperar la información que almacena, en el orden de su inserción. En este sentido funcionan como un array, la diferencia es que un elemento de `Set` no puede estar repetido dentro de la misma colección.

La propiedad `size` nos devuelve el número de elemento almacenados en un `Set`. En cuanto sus métodos tenemos los siguientes:

- `add()`: Añade un nuevo elemento.
- `clear()`: Vacía la colección.
- `delete()`: Elimina un elemento de la colección.
- `entries()`: Devuelve un objeto **Iterador** que contiene un array de tuplas [valor, valor] para hacerlo compatible con otro tipo de colecciones como veremos más adelante.
- `forEach()`: Nos permite iterar sobre la colección.
- `has()`: Comprueba si un elemento pertenece a la colección.
- `values()`: Nos devuelve un objeto **Iterador** con los valores de la colección.

4.1. Utilizando objetos `Set`

En este ejemplo estamos instanciando un nuevo `Set`, y añadimos diferentes tipos de elementos con `add()`. Aunque intentamos añadir dos objetos con datos similares, se trata de dos referencias diferentes, y por tanto podemos añadir a la colección.

Si intentamos borrar un elemento de la colección que no existe, el intérprete no genera ningún error.

```
function workingWithSets() {
  // Nuevo objeto set.
  const mySet = new Set();

  // Añadimos elementos heterogéneos al set
  mySet.add(1);
  mySet.add(5);
  mySet.add('text');
  const o = { a: 1, b: 2 };
  mySet.add(o);
  // Dos objetos diferentes con referencias diferentes
  mySet.add({ a: 1, b: 2 });

  // Tamaño del set
  $$result.log(mySet.size); // 5

  // Existencia de elementos en el set
  $$result.log(mySet.has(1)); // true
  $$result.log(mySet.has(3)); // false
  $$result.log(mySet.has('TEXT'.toLocaleLowerCase())); // true
  $$result.log(mySet.has(o)); // true

  // Eliminar elementos
  mySet.delete(5);
  $$result.log(mySet.has(5)); // false
  $$result.log(mySet.size); // 4
  // No genera error eliminar un elemento no existente
  mySet.delete(11111);
}
```

4.2. Añadir elementos a un `Set`

El constructor Set permite inicializar la colección a partir de un iterable, por ejemplo, un array. Además, los métodos de un objeto Set devuelve una instancia de la colección, lo que nos permite encadenar llamadas sobre la instancia, simplificando la codificación.

```
function addElementsInSet() {
  // Crear un set a partir de un iterable
  const mySet = new Set([1, 2, 3]);
  $$result.log(mySet.size); // 3
  $$result.log(mySet.has(1)); // true

  // Encadenado de llamadas a add()
  mySet.add(4).add(5).add(6);
  $$result.log(mySet.size); // 6
  $$result.log(mySet.has(6)); // true

  // Vaciar set
  mySet.clear();
  $$result.log(mySet.size); // 0
}
```

4.3. Iterar sobre un objeto Set

Iterar sobre un objeto `Set` es igual que si lo hacemos con un array. Lo podemos hacer con un bucle `for/of`.

```
function iterateOverSetV1() {
  const mySet = new Set([1, 2, 3]);
  // Al iterar recuperamos los elementos en orden de inserción
  for (const item of mySet) $$result.log(item); // 1,2,3
}
```

El método `values()` devuelve un objeto *iterador* sobre el cual podemos recuperar cada uno de los elementos de la colección.

```
function iterateOverSetV2() {
  const mySet = new Set([1, 2, 3]);
  // Al iterar recuperamos los elementos en orden de inserción
  for (const item of mySet.values()) $$result.log(item); // 1,2,3
}
```

Por último, tenemos el método `forEach()` que lo utilizamos con una función *callback* con lo mismo parámetros al hacerlo con en un array.

```
function iterateOverSetV3() {
  const mySet = new Set([1, 2, 3]);
  // Iteración utilizando forEach
  mySet.forEach((item) => {
    $$result.log(item); // 1,2,3
  });
}
```

5. Trabajando con `Set`

Vamos a integrar en nuestro proyecto de registro de ventas colecciones `Set`.

5.1. Inicialización de variables

Declaramos los objetos DOM para recoger los valores del formulario en el modal de la página.

```
// Valores del formulario
const newAmount = document.getElementById('itemAmount');
const newMonth = document.getElementById('monthId');
```

Al gráfico de barras lo definimos sin datos iniciales. En la propiedad *data* asignamos valores vacíos a *labels* y *datasets/data*.

```
data: {
  labels: [],
  datasets: [{
    label: 'Número de ventas',
    data: [],
    backgroundColor: [
      'rgba(238, 184, 104, 1)',
      'rgba(75, 166, 223, 1)',
      'rgba(239, 118, 122, 1)',
    ],
    borderWidth: 0,
  }],
},
```

5.2. Añadir ventas

Implementamos esta función por pasos.

5.2.1. Modal de nueva venta

Debemos identificar el botón que abre el modal de nueva venta, por lo que le asignamos el atributo `id` para que quede de la siguiente forma.

```
<button id="bAddSaleModal" type="button" class="btn">
    <i class="bi bi-file-earmark-plus"></i> Añadir venta
</button>
```

Creamos el objeto DOM para el botón.

```
const bAddSaleModal = document.getElementById('bAddSaleModal');
```

Si comprobamos el código del formulario veremos que tenemos identificadores para cada uno de los `inputs`:

- `monthId`: Recoger el mes. Es de tipo `month`.
- `itemAmount`: Recoge la cantidad. Es de tipo `number`.
- `inlineRadioOptions`: Aunque seleccionado por `name`, lo utilizaremos más adelante para diferenciar el tipo de venta.

Declaramos dos objetos para almacenar datos mensuales y las etiquetas que serán utilizados en los gráficos. Las etiquetas estarán en una colección para evitar repeticiones y las ventas en un array.

```
//Colecciones para mostrar en gráficos.
const monthlyLabelsSet = new Set();
const monthlySalesArray = [];
```

5.2.2. Añadir elementos a la colección

Definimos una función que nos permita añadir ventas recogidas desde el formulario. Accediendo a los objetos DOM del formulario, su atributo `value` nos permite recoger el valor. Debemos tener en cuenta que los elementos `input` recoger los datos introducidos por el usuario en formato *string*, por lo que la cantidad la debemos traducir a *number*. Accedemos a la estructura de la colección desde la consola para comprobar los cambios.

```
//Añadir ventas al gráfico
function addSale(){
  monthlySalesArray.push(Number.parseInt(newAmount.value));
  monthlyLabelsSet.add(newMonth.value);
  alert("Número de ventas: " + monthlyLabelsSet.size);
  console.log(monthlyLabelsSet);
}
```

Vinculamos la función con el evento *click* del botón.

```
bAddSaleModal.addEventListener('click', addSale);
```

5.2.3. Tratamiento de valores repetidos

Vamos a realizar un chequeo de si el *mes* existe en el conjunto de datos de la gráfica. Hacemos un chequeo de los datos introducidos y los tratamos con una excepción. El uso de excepciones en este caso nos permite validar los datos de entrada para tratar valores no válidos.

El cheque lo hacemos con el método `has()` y en caso de existir el mes lanzamos una excepción personalizada. El bloque `finally` nos permite resetear los valores del formulario independientemente de que hayamos podido registrar la venta o no. Los pasos intermedios serían recalcular los totales y actualizar el gráfico con los nuevos datos.

```
function addSale() {
  try {
    // Validación de datos de entrada
    if (monthlyLabelsSet.has(newMonth.value)) {
      throw {
        name: 'MonthError',
        message: 'El mes ya está incluido en la gráfica.',
      };
    }
    monthlySalesArray.push(Number.parseInt(newAmount.value));
    monthlyLabelsSet.add(newMonth.value);
    // Recuento de totales
    initMonthlyTotalSales();
    // Actualizar gráfico
    monthlySalesChart.data.datasets[0].data = monthlySalesArray;
    monthlySalesChart.data.labels = Array.from(monthlyLabelsSet);
    monthlySalesChart.update();
  } catch (error) {
    // Tratamiento de excepciones
    alert(error.message);
  } finally {
    // Reseteo de formulario
    cleanAddSaleForm();
  }
}
```

Añadimos el método que permite resetear el formulario.

```
function cleanAddSaleForm() {
  newMonth.value = '';
  newAmount.value = '';
}
```

5.2.4. Recuento de totales

La función de recuento de totales debe utilizar el nuevo array que estamos utilizando.

```
function initMonthlyTotalSales() {
  yearlyLabel.innerHTML = `${monthlySalesArray.reduce((count, value) =>
count + value, 0)}€`;
}
```

5.2.5. Resetear los datos del gráfico

Necesitamos actualizar la función `resetMonthlySales()` para inicializar los gráficos para empezar de cero con los datos. El método `clear()` elimina todos los datos de un objeto Set, para el *array* asignamos la longitud a 0 y actualizamos el gráfico.

```
//Resetear datos en los gráficos
function resetMonthlySales(){
  monthlySalesArray.length = 0;
  monthlyLabelsSet.clear();
  monthlySalesChart.update();
  initMonthlyTotalSales();
}
```

5.2.6. Recuento de ventas

Creamos una función que recorra el conjunto de etiquetas de ventas mostrando los meses que hemos añadido en una función `alert()`.

```
function getSalesMonths(){
  monthlyLabelsSet.forEach(function (month){
    console.dir(month);
    alert(month);
  });
}
```

Asignamos la función como manejador al botón de búsqueda de ventas.

6. Map

Los objetos `Map` son colecciones de pares **clave/valor**, es decir, podemos localizar un valor en la colección en base a la clave con la que está relacionada.

La propiedad `size` nos devuelve el número de elemento almacenados en un `Map`. En cuanto sus métodos tenemos los siguientes:

- `clear()`: Vacía la colección.
- `delete()`: Elimina un elemento de la colección en función de la clave.
- `entries()`: Devuelve un objeto **Iterator** que contiene un array de tuplas [clave, valor] para hacerlo compatible con otro tipo de colecciones como veremos más adelante.
- `forEach()`: Nos permite iterar sobre la colección. Le pasamos una función de *callback* y opcionalmente el argumento `this`.
- `get()`: Devuelve el valor asociado a una clave concreta.
- `has()`: Comprueba si una clave pertenece a la colección.
- `keys()`: Obtenemos un **Iterator** con las claves de la colección.
- `set()`: Añade un nuevo para clave/valor a la colección.
- `values()`: Nos devuelve un objeto **Iterator** con los valores de la colección.

6.1. Utilizando objetos Map

En este ejemplo estamos instanciando un nuevo `Map`, y añadimos diferentes pares clave/valor con `set()`. La clave puede ser tanto tipos primitivos como objetos. Si la clave ya existe en la colección, el método actualizará el valor asociado a dicha clave.

```
function workingWithMaps() {
  const myMap = new Map();
  const obj = {};
  const f = function () {};
  const str = 'Text';

  // Asignar valores a un Map
  myMap.set(obj, 'Valor con Object');
  myMap.set(f, 'Valor con function');
  myMap.set(str, 'Valor con string');
  $$result.log(myMap.size); // 3
  // Obtener valores
  $$result.log(myMap.get(obj)); // Valor con Object
  $$result.log(myMap.get(f)); // Valor con function
  $$result.log(myMap.get({})); // undefined
  // Borrar elementos
  myMap.delete(obj);
  $$result.log(myMap.size); // 2
}
```

6.2. Iterar sobre un objeto `Map`

Podemos iterar sobre un objeto `Map` podemos utilizar un bucle `for/of`.

```
function iterateOverMapV1() {
  const myMap = new Map([[0, 'cero'], [1, 'uno']]);
  // Iterando con for/of
  for (const [key, value] of myMap) {
    $$result.log(`Clave: ${key} Valor: ${value}`);
  }
}
```

También podemos iterar sobre las claves y los valores de la colección.

```
function iterateOverMapV2() {
  const myMap = new Map([[0, 'cero'], [1, 'uno']]);
  // Iterando sobre arrays de claves y valores
  for (const key of myMap.keys()) {
    $$result.log(`Clave: ${key}`);
  }
  for (const value of myMap.values()) {
    $$result.log(`Valor: ${value}`);
  }
}
```

El método `entries()` devuelve un iterador con un array con la clave y el valor de cada elemento.

```
function iterateOverMapV3() {
  const myMap = new Map([[0, 'cero'], [1, 'uno']]);
  // Iteramos sobre un array con las entradas.
  for (const [key, value] of myMap.entries()) {
    $$result.log(`Clave: ${key} Valor: ${value}`);
  }
}
```

Por último, podemos utilizar una función *callback* en el método `forEach()`, siendo sus parámetros el valor, la clave y el objeto por ese orden.

```
function iterateOverMapV4() {
  const myMap = new Map([[0, 'cero'], [1, 'uno']]);
  // Iteramos sobre el método forEach
  myMap.forEach((value, key, m) => {
    $$result.log(`Clave: ${key} Valor: ${value}`);
  });
}
```

7. Trabajando con `Map`

En este apartado transformamos el registro de ventas utilizando colecciones `Map`.

7.1. Inicialización de variables

Cambiamos el tipo de colección para utilizar `Map`. La idea es que las claves del objeto sean las etiquetas del gráfico, y los valores pasen a ser las cantidades mensuales.

```
const monthlySalesMap = new Map();
```

7.2. Añadir ventas

Rediseñamos la funcionalidad con la nueva colección. En primer lugar, necesitamos validar si el mes introducido forma parte de la colección. Utilizamos el método `has()`. Si ya pertenece lanzamos una excepción para que sea tratada para informar al usuario desde el bloque `catch`.

Tenemos que añadir el mes como clave y la cantidad como valor a la colección con `set()`. Posteriormente tenemos que hacer el recuento de totales y actualizamos el gráfico utilizando el mapa. El método `Array.from()` nos genera un array a partir de un iterador, el cual conseguimos con los métodos `values()` y `keys()` de `Map`.

Hay que finalizar la función limpiando el formulario para la siguiente inserción.

```
function addSale() {
  try {
    // Validación de datos de entrada
    if (monthlySalesMap.has(newMonth.value)) {
      throw {
        name: 'MonthError',
        message: 'El mes ya está incluido en la gráfica.',
      };
    }
   monthlySalesMap.set(newMonth.value, Number.parseInt(newAmount.value));
    // Recuento de totales
    initMonthlyTotalSales();
    // Actualizar gráfico
    monthlySalesChart.data.datasets[0].data =
Array.from(monthlySalesMap.values());
    monthlySalesChart.data.labels = Array.from(monthlySalesMap.keys());
    monthlySalesChart.update();
  } catch (error) {
    // Tratamiento de excepciones
    alert(error.message);
  } finally {
    // Reseteo de formulario
    cleanAddSaleForm();
  }
}
```

Para el recuento de totales utilizamos la misma técnica anterior, generamos un array a partir de los valores de la colección, para reducirlo a un único valor a través de una función de *callback*.

```
function initMonthlyTotalSales(){
  yearlyLabel.innerHTML = Array.from(monthlySalesMap.values()).reduce(
function (count, value){ return count + value; }, 0) + "€";
}
```

7.3. Resetear los datos del gráfico

Modificamos la función para utilizar la nueva colección.

```
function resetMonthlySales(){
  monthlySalesMap.clear();
  monthlySalesChart.reset();
  monthlySalesChart.render();
  initMonthlyTotalSales();
}
```

7.4. Recuento de ventas

Para el recuento de ventas recorremos la colección con el método `forEach()` cuya función de *callback* tiene dos parámetros obligatorios. A diferencia de los conjuntos

- **currentValue**: El valor del elemento actual que estamos procesando.
- **currentKey**: Key por lo que obtenemos el valor actual.

```
function getSalesMonths() {
  monthlySalesMap.forEach((products, month) => {
    products.forEach((amount, product) => {
      alert(`${month}: ${product} ${amount}`);
    });
  });
}
```

7.5. Eliminar meses del gráfico

Para eliminar un mes del gráfico tenemos que hacerlo nuevamente por pasos.

7.5.1. Botón del modal

Añadimos un identificador al botón que abre el modal.

```
<button id="bRemoveSale" type="button" class="btn">
    <i class="bi bi-file-earmark-x"></i> Eliminar venta
</button>
```

Recogemos el objeto en una constate.

```
const bRemoveSale = document.getElementById('bRemoveSale');
```

7.5.2. Seleccionar el mes a eliminar

Creamos una función que permita crear las opciones de un `select` a partir de los datos de la colección, lo hacemos en la función `drawSelectMontlySales()`.

En esta función vamos a tener la primera toma de contacto con el framework **jQuery**. Para seleccionar elementos con el framework utilizamos el prefijo "$", y el carácter "#" indica el uso de un identificador. La sentencia `$("#removeSales")` hace referencia al elemento con el identificador indicado de la página. Una vez seleccionado el elemento con *jQuery* podemos eliminar su contenido con el método `empty()`, por lo que borramos todos los elementos `option` del `select`.

Iterando sobre la colección, creamos un elemento `option`, y con el método `val()` le asignamos un atributo `value` con la clave del item, y con `text()` fijamos el contenido del

elemento. Por último, añadimos cada nuevo `option` creado al `select` utilizando el método `append()`.

Podríamos haber utilizado el API de DOM para crear esta función, pero es extremadamente más complejo hacer que ese API nativa.

```
function drawSelectMontlySales(){
  // Seleccionamos elemento usando id con jQuery
  let removeSales = $("#removeSales");
  // Eliminamos option del select.
  removeSales.empty();
  for (let [month, amount] of monthlySalesMap.entries()){
    // Creamos elemento option con jQuery
    let opt = $("<option>").val(month).text(month + ": " + amount);
    // Añadimos elemento al select.
    removeSales.append(opt);
  }
}
```

Asignamos al evento *click* del botón la función.

```
bDeleteSale.addEventListener('click',drawSelectMontlySales);
```

7.5.3. Eliminar el mes del gráfico

La función recoge el valor que hemos seleccionado en el select, el cual es una clave de la colección, y con el método `delete()` lo eliminamos. Tenemos que actualizar las etiquetas y los datos en el gráfico, así como modificar la vista con el total de ventas, así como reconstruir el `select` del formulario, ya que el mes no debe de estar disponible.

```
// Borrar meses de la colección
function removeMonthlySale(){
  let removeSales = document.getElementById("removeSales");
  // Borramos de la colección la venta.
  monthlySalesMap.delete(removeSales.value);
  // Actualizamos colección en el gráfico
  monthlySalesChart.data.datasets[0].data =
Array.from(monthlySalesMap.values());
  monthlySalesChart.data.labels = Array.from(monthlySalesMap.keys());
  monthlySalesChart.update();
  // Actualizasmos la vista
  initMonthlyTotalSales();
  drawSelectMontlySales();
}
```

Tenemos que asociar la función anterior al botón dentro del modal para que se ejecuta al cliquearse. Añadimos un identificador.

```
<button id="bRemoveSaleModal" type="button" class="btn">
    <i class="bi bi-file-earmark-x"></i> Eliminar venta
</button>
```

Y asociamos la función como manejador de eventos al objeto.

```
const bRemoveSaleModal = document.getElementById('bRemoveSaleModal');
bRemoveSaleModal.addEventListener('click',removeMonthlySale);
```

8. WeakSet

Los objetos `WeakSet` con colecciones de objetos. Su funcionamiento es exactamente igual que las colecciones `Set`, donde un elemento solo puede encontrarse una única vez almacenado en la colección, por tanto, sus elementos serán únicos. La diferencia entre ambos es:

- Una colección `WeakSet` solo admite objetos entre sus elementos. No podemos añadir valores primitivos.
- La colección es "débil", ya que los objetos que añadidos a la colección y se haya perdido la referencia, por ejemplo, por haber finalizado el bloque donde se han declarado, estos objetos pueden ser eliminados automáticamente por el *Garbage Collector*. Los no son `WeakSet` enumerables.

Los métodos se reducen a:

- `add()`
- `delete()`
- `has()`

Como podemos observar, este tipo de colección se limita a almacenar objetos, y no podemos iterar sobre su contenido. Su función principal es la de **almacenamiento de datos adicional**.

En un proyecto es muy habitual trabajar con diferentes tipos de librerías o framework, por lo que en ocasiones podemos necesitar almacenar datos asociados de terceros. Cuando la librería que ha creado el objeto deja de referenciarlo, nuestra colección deberían también hacerlo. Esta función es lo que precisamente hace `WeakSet`, por el contrario, con una colección `Set`, el elemento permanecería en la colección a pesar de no tener referencias activas hacía él.

8.1. Uso de colecciones `WeakSet`

En este ejemplo añadimos objetos usuario a una colección para saber si nos ha visitado. Cuando estas referencias dejan de existir, por ejemplo, al terminar un bloque, automáticamente dejan de estar accesibles en la colección también.

```
  function weakSetExample() {
    const visitedSet = new WeakSet();
    const john = { name: 'John' };
    const pete = { name: 'Pete' };

    visitedSet.add(john); // John nos visita
    visitedSet.add(pete); // luego Pete
    // visitedSet tiene 2 usuarios
    {
      const mary = { name: 'Mary' };
      visitedSet.add(mary);
      $$result.log(visitedSet.has(mary)); // true
      // visitedSet tiene 3 usuarios
    }
    // Mary desaparece de la colección por acabar su bloque y perderse su
referencia.
    $$result.log(visitedSet.has(john)); // true
    $$result.log(visitedSet.has(pete)); // true
  }
```

9. WeakMap

La relación entre una colección `Map` y una `WeakMap` es exactamente igual que una relación en `Set` y `WeakSet`. En `WeakMap` solamente podemos tener como claves objetos, y si la referencia al objeto es eliminada, automáticamente también es eliminado de la colección. Nuevamente no podemos iterar sobre este tipo de colección y no es enumerable.

Los métodos que tenemos son:

- `delete()`
- `get()`
- `has()`
- `set()`

9.1. Uso de colecciones `WeakMap`

En este ejemplo tenemos una función que incremente un contador para mantener el número de visitas que ha realizado un usuario. La función tiene en cuenta si la clave ya se encuentra en la colección, o lo tiene que inicializar a 0.

Nuevamente mientras que mantenemos la referencia al objeto, básicamente en el bloque que hemos creado, la entrada está almacenada en la colección. En el momento que perdemos la referencia al terminar el bloque, automáticamente la entrada es eliminada por el Garbage Collector.

```
function weakMapExample() {
  // incrementar el recuento de visitas
  function countUser(user) {
    const count = visitsCountMap.get(user) || 0;
    visitsCountMap.set(user, count + 1);
  }

  let visitsCountMap = new WeakMap();
  {
    const john = { name: 'John' };
    countUser(john); // cuenta sus visitas
    countUser(john); // cuenta sus visitas
    countUser(john); // cuenta sus visitas
    $$result.log(visitsCountMap.get(john)); // 3
  }
  // John deja de pertenecer automáticamente a la colección.
}
```

9.2. Cache con `WeakMap`

Una aplicación interesante de este tipo de colección es crear una caché para almacenar cálculos complejos en función de un objeto, que nos permita recuperar dicho valor sin tener que recalcularlo. Cuando el objeto deja de existir, la entrada en la colección se elimina automáticamente.

En este ejemplo creamos una función interna que en base a un objeto de entrada realiza un cálculo, el cual estamos simulando con un bucle `for`. Si el objeto no está todavía en la colección realizamos el cálculo y lo guardamos siendo la clave el propio objeto. Si el objeto ya existe en la colección, el dato calculado es recuperado de la colección.

Como vemos, la primera vez que se ejecuta el método `process()` calculamos el dato. En la segunda invocación ya tenemos el dato y lo recuperamos de la colección.

```
function cache() {
  // calcular y recordad el resultado
  function process(obj) {
    if (!cache.has(obj)) {
      let result = 0;
      for (let i = 1; i < 64; i++) result += 2 ** i;
      cache.set(obj, result);
    }
    return cache.get(obj);
  }
  let cache = new WeakMap();

  const obj = {/* Objeto cualquiera */};

  $$result.log(process(obj)); // 18446744073709552000
  $$result.log(process(obj)); // 18446744073709552000
}
```