TEMA 6

OBJETOS

INTRODUCCIÓN

Los objetos son estructuras de información capaces de contener variables (denominadas propiedades), así como funciones (llamadas métodos). Como los objetos almacenan valores junto con funciones, son como programas independientes que se comunican entre sí para realizar tareas comunes.

La idea detrás de los objetos en programación es la de simular el rol de los objetos en la vida real. Un objeto real tiene propiedades y realiza acciones. Por ejemplo, una persona tiene nombre, apellidos y una dirección postal, pero también puede caminar y hablar. Las características y la funcionalidad son parte de la persona y es la persona la que define cómo va a caminar y lo que va a decir. Organizando nuestro código de esta manera, podemos crear unidades de procesamiento independientes capaces de realizar tareas y que cuentan con toda la información que necesitan para hacerlo. Por ejemplo, podemos crear un objeto que controla un botón, muestra su título, y realiza una tarea cuando se pulsa el botón. Como toda la información necesaria para presentar y controlar el botón se almacena dentro del objeto, el resto del código no necesita saber cómo hacerlo. Cuando conocemos los métodos provistos por el objeto y los valores devueltos, el código dentro del objeto se puede actualizar o reemplazar por completo sin afectar el resto del programa.

Las ventajas de los objetos es que podemos crear unidades de procesamiento independientes, duplicar esas unidades tantas veces como sea necesario, y modificar sus valores para adaptarlos a las circunstancias actuales.

OBJETOS BASADO EN PROTOTIPOS

1. DECLARACIÓN DE OBJETOS

Existen diferentes formas de declarar objetos en JavaScript, la más sencilla es la notación literal. El objeto se declara como cualquier otra variable usando la palabra clave var, let o const (como las variables), y las propiedades y métodos que definen el objeto se declaran entre llaves usando dos puntos después del nombre y una coma para separar cada declaración **(key: valor).**

```
<script>
let usuario = {
 nombre: "Maite",
 edad: 25
}

</script>
```

En el ejemplo anterior podíamos haber utilizado también las palabras clave var y const, y el objeto usuario obtenido hubiera sido el mismo. En este caso este objeto tiene dos propiedades **nombre** y **edad** y los valores con “Maite” y 25 respectivamente.

A diferencia de las variables, no podemos acceder a los valores de las propiedades de un objeto usando solo sus nombres, se debe también especificar el nombre del objeto al que pertenecen usando notación de puntos o corchetes. Veamos cómo:

```
<script>
let usuario = {
 nombre: "Maite",
 edad: 25
};
let mensaje = "Hola soy "+usuario.nombre + "y tengo "+usuario["edad"]+ "años"
alert(mensaje)

</script>
```

Como vemos en el ejemplo anterior hemos usado las dos opciones, el nombre del objeto y la clave del valor que queremos recuperar separados por un punto (**usuario.nombre**) y en el segundo caso el nombre del objeto y entre corchetes la clave del valor que queremos recuperar **(usuario[nombre]).**

Utilizar una opción u otra es irrelevante, aunque cuando el nombre de una propiedad utiliza un nombre que el intérprete puede considerar no válido, es obligatorio utilizar corchete. En el siguiente ejemplo hemos utilizado un espacio en blanco al definir una clave y en este caso obligatoriamente debemos utilizar corchetes.

```
<script>
let usuario = {
 nombre: "Maite",
 'mi edad': 25
};
let mensaje = "Hola soy "+usuario.nombre + "y tengo "+usuario["mi edad"]+ "años"
alert(mensaje)

</script>
```

En este caso obligatoriamente debemos utilizar corchetes al utilizar la clave “mi edad”, ya que si utilizáramos la opción del punto nos daría error.

```
<script>
let usuario = {
 nombre: "Maite",
 'mi edad': 25,
 hijos: ["Jorge", "Laura"]
};
let mensaje = "Hola soy "+usuario.nombre + "y tengo "+usuario["mi edad"]+ "años"+
" además tengo dos hijos: "+ usuario.hijos[0]+ " y "+usuario.hijos[1]
alert(mensaje)
</script>
```

Además de leer los valores de las propiedades, también podemos asignar nuevas propiedades al objeto o modificarlas usando notación de puntos. En el siguiente ejemplo, modificaremos el nombre y añadimos una nueva propiedad llamada trabajo.

```
let usuario = {
 nombre: "Maite",
 'mi edad': 25,
 hijos: ["Jorge", "Laura"]
};
usuario.nombre="Maite García"
usuario.trabajo = "Profesor"
```

Los objetos también pueden contener otros objetos. En el siguiente ejemplo, asignamos un objeto a la propiedad de otro objeto.

```
<script>
let usuario = {
 nombre: "Maite",
 'mi edad': 25,
 direccion : {
    calle: "Conde Aranda",
    ciudad: "Zaragoza"
 }
};
let mensaje = "Hola soy "+usuario.nombre + " y tengo "+usuario["mi edad"]+ "años"+
" además vivo en " + usuario.direccion.ciudad
alert(mensaje)
</script>
```

En este caso para indicar que algo es un objeto, como podemos ver se indica con los dos puntos, añadiendo las claves y los valores entre llaves. En este caso para acceder al valor de la propiedad tenemos que ir accediendo a cada uno de los objetos separando los valores con puntos.

2. METODOS

Los objetos también pueden incluir funciones, denominadas dentro de los objetos métodos. Los métodos tienen la misma sintaxis que las propiedades: requieren dos puntos después del nombre y una coma para separar cada declaración, pero en lugar de valores, debemos asignarles funciones anónimas.

```
<script>
let usuario = {
 nombre: "Maite",
 edad: 25,
 mostrarDatos: function(){
    let mensaje = "Nombre: "+usuario.nombre + "\n"+
    "Y tengo "+usuario.edad
    alert(mensaje)
    },
    cambiarNombre: function(nuevoNombre){
        usuario.nombre=nuevoNombre
    }   }
    usuario.mostrarDatos()
    usuario.cambiarNombre("Juan")
    usuario.mostrarDatos()
</script>
```

En este ejemplo, agregamos dos métodos al objeto: mostrarDatos() y cambiarNombre(). El primero muestra una ventana emergente con los valores de las propiedades nombre y edad, y el segundo asigna el valor recibido en el parámetro a la clave nombre. Estos dos métodos

independientes que trabajan sobre las mismas propiedades uno lee sus valores y el otro los modifica. Para ejecutar los métodos, usamos la notación y puntos y paréntesis después del nombre, como hacemos con funciones.

Igual que en las funciones, los métodos pueden devolver valores, para ello utilizaremos la función return. Veamos cómo:

```
<script>
let usuario = {
 nombre: "Maite",
 edad: 25,
 mostrarDatos: function(){
    let mensaje = "Nombre: "+usuario.nombre + "\n"+
    "Y tengo "+usuario.edad
    alert(mensaje)
    },
    cambiarNombre: function(nuevoNombre){
        let nombreViejo = usuario.nombre
        usuario.nombre=nuevoNombre
        return nombreViejo
    }   }
    usuario.mostrarDatos()
    alert(usuario.cambiarNombre("Juan"))
    </script>
```

El nuevo método cambiarNombre() almacena el valor actual de la propiedad nombre en una variable temporal llamada nombreViejo para poder devolver el valor anterior después de que el nuevo se asigna a la propiedad.

3. LA PALABRA CLAVE THIS

En los últimos ejemplos, mencionamos el nombre del objeto cada vez que queríamos modificar sus propiedades desde los métodos. Aunque esta técnica funciona, no es una práctica recomendada. Cuando el nombre del objeto queda determinado por el nombre de la variable al que se asigna el objeto, el mismo se puede modificar sin advertirlo. Como veremos, JavaScript nos permite crear múltiples objetos desde la misma definición o crear nuevos objetos a partir de otros, lo que produce diferentes objetos que comparten la misma definición. Para asegurarnos de que siempre referenciamos al objeto con el que estamos trabajando, JavaScript incluye la palabra clave **this**. Usaremos esta palabra en lugar del nombre del objeto para referenciar el objeto al que la instrucción pertenece. Veamos cómo modificar el ejemplo anterior utilizando this.

```
<script>
let usuario = {
 nombre: "Maite",
 edad: 25,
 mostrarDatos: function(){
    let mensaje = "Nombre: "+this.nombre + "\n"+
    "Y tengo "+this.edad
    alert(mensaje)
     },
    cambiarNombre: function(nuevoNombre){
        let nombreViejo = this.nombre
        this.nombre=nuevoNombre
        return nombreViejo
    }   }
    usuario.mostrarDatos()
    alert(usuario.cambiarNombre("Juan"))
    </script>
```

4. CONSTRUCTORES

Usando notación literal podemos crear objetos individuales, pero si queremos crear copias de estos objetos con las mismas propiedades y métodos, usaremos constructores. Un constructor es una función anónima que define un nuevo objeto y lo devuelve, creando copias del objeto (o instancias), cada una con sus propias propiedades, métodos y valores.

```
<script>
let constructor = function(nombreUsuario){
  let nuevoUsuario = {
 nombre: nombreUsuario,
 mostrarDatos: function(){
      alert(this.nombre)
      }
       }
       return nuevoUsuario;
      }
      let usuario1 = constructor("Maite")
      usuario1.mostrarDatos();
      </script>
```

El propósito de un constructor es el de funcionar como una fábrica de objetos. Podemos crear usuario2, usuario3, etc. de la misma forma que hemos creado usuario1 en el ejemplo anterior.

Aunque los objetos creados desde un constructor comparten las mismas propiedades y métodos, se almacenan en diferentes espacios de memoria y, por tanto, manipulan valores diferentes.

OBJETOS ESTANDAR

Los objetos son como envoltorios de código y JavaScript se aprovecha de esta característica extensamente. Es más, casi todo en JavaScript es un objeto, por ejemplo: los números, las cadenas se convierten automáticamente en objetos por el intérprete JavaScript. Cada vez que asignamos un nuevo valor a una variable, en realidad estamos asignando un objeto que contiene ese valor.

Como los valores que almacenamos en variables son de tipos diferentes, existen distintos tipos de objetos disponibles para representarlos. Por ejemplo, si el valor es una cadena de caracteres,

el objeto almacenado es de tipo String. Cuando asignamos un texto a una variable, JavaScript crea un objeto String, almacena la cadena de caracteres en el objeto y asigna ese objeto a la variable. Existen distitnos constructores disponibles dependiendo del tipo de valor que queremos almacenar. Los siguientes son los más usados.

- **Number (valor).** Este constructor crea objetos para almacenar valores numéricos. Acepta números y también cadenas de caracteres con números. Si el valor especificado por el atributo no se puede convertir en un número, el constructor devuelve el valor **NaN** (No es un número).

  let numero1 = new Number(15)

- **String(valor).** Este constructo crea objetos para almacenar cadenas de caracteres. Acepta una cadena o cualquier valor que pueda convertirse en una cadena de caracteres.
  let nombre = new String("Maite")
- **Boolean(valor).** Este constructor crea objetos para almacenar valores booleanos. Acepta los valores true y false. Si el valor se omite o es igual a 0, NaN, null, undefined, o una cadena de caracteres vacía, el valor almacenado en el objeto es false, en caso contrario true.
- **Arrays(valores).** Este constructor crea objetos para almacenar arrays. Si se proveen múltiples valores, el constructor crea un array con esos valores, pero si solo se provee un valor, y ese valor es un número entero, el constructor crea un array con la cantidad de elementos que indica el valor del atributo y almacena el valor undefined en cada índice.
  let lista = new Array(12, 35, 57, 8);
  Si lo que especificamos es un único valor y ese valor es un número entero, el constructor lo usa para determinar el tamaño del array, crea un array con esa cantidad de elementos y asigna el valor undefined a cada uno de ellos.
  let lista = new Array(2);
  crea un objeto con dos elementos.

EL OBJETO ARRAY

El objeto de JavaScript es un objeto global que es usado en la construcción de arrays, que son objetos tipo lista de alto nivel. Los arrays son objetos similares a una lista cuyo prototipo proporciona métodos para realizar operaciones de recorrido y mutación. Tanto la longitud como el tipo de los elementos de un array son variables. Dado que la longitud de un arreglo puede cambiar en cualquier momento, y lo datos se pueden almacenar en posiciones no contiguas, no se manejan de la misma forma que en otros lenguajes de programación como Java.

1. CÓMO RECORRER UN ARRAY

Ya hemos visto que una de las formas en que podemos recorrer un array, es mediante el uso de la sentencia for, asociándolo a los índices que acompañan a los elementos de un array, y también vimos que existe otra forma de recorrer un array con la opción for… of, recordemos cómo se utiliza esta sentencia.

El uso del bucle for … of, simplifica al for tradicional, ya que no hay que declarar una variable para recorrer el array, ni especificar de antemano el número de veces que se ejecutará, ni definir un incremento.

Veamos cómo utilizar este bucle con un ejemplo concreto, para ello partiremos de un array ciudades con 5 ciudades, y veremos cómo recorrerlo con esta nueva estructura de bucle.

let ciudades = [“Zaragoza”, “Madrid”, “Valencia”,”Barcelona”,”Sevilla]

ahora lo que tenemos que hacer es declarar una variable, donde se guardarán cada uno de los valores incluidos en el array, de la siguiente forma:

```
for(let city of ciudades){
        console.log(city);
}
```

En este caso la variable city va tomando cada uno de los valores del array, **city=”Zaragoza”** en el primer momento, **city=”Madrid”**, y así sucesivamente. El nombre de la variable puede ser cualquiera, pero por convenio se suele utilizar el nombre del array en plural y el nombre de la variable en singular.

Esta misma instrucción utilizando el bucle **for… in**, nos devolverá los índices asociados a cada uno de los valores, así que si queremos crear con este bucle algo similar a lo anterior, deberemos escribir lo siguiente:

```
for(let city in ciudades){
        console.log(ciudades[city]);
        }
```

En este caso city recorre cada uno de los índices del array, y si accedemos al vector en cada una de esas posiciones nos encontraremos con los valores incluidos en el array, como antes. Por eso este bucle for…in se utiliza para recorrer objetos y no arrays.

2. FUNCIONES PREDEFINIDAS ASOCIADAS A LOS ARRAYS

Veremos algunas de las funciones que podemos utilizar asociadas a los arrays, y su propiedad más importante que el length.

2.1 PROPIEDAD length

La propiedad length de un array en JavaScript indica el número de elementos que nos encontramos en un array. Recordar que cuando queramos acceder a los elementos de un array mediante su índice, hay que tener en cuenta que los índices empiezan en cero, y por tanto el último estará en la posición length-1.

2.2 FUNCIÓN unshift

La función unshift permite añadir un elemento al principio, es decir el nuevo elemento ocupará la posición asociada al índice cero.

```
<script>
let frutas =  ["pera", "manzana"]
alert(frutas)
frutas.unshift("fresa")
alert(frutas)

    </script>
```

En este caso el array frutas será [“fresa”,”pera”,”manzana”].

2.3 FUNCIÓN push

La función **push** añade un elemento, pero en este caso al final de la lista. La forma de manejarla es exactamente igual que la anterior.

```
frutas.push("fresa")
alert(frutas)
```

2.4 FUNCIÓN pop

La función pop elimina el último elemento de un array. Como podemos ver en el siguiente ejemplo, la función no necesita parámetros.

```
frutas.pop()
alert(frutas)
```

2.5 FUNCIÓN shift

La función shift elimina el primer elemento, tampoco necesita que pasemos ningún parámetro.

```
frutas.shift()
alert(frutas)
```

2.6 FUNCIÓN indexOf

La función indexOf nos permite conocer el primer índice que ocupa dentro del array un elemento, por eso necesitaremos un parámetro que se pasará a esta función. El método devuelve ese primer índice donde se encuentra un elemento determinado en un array o String, y devolverá -1 si el elemento que se buscaba no existe dentro del array.

Veamos un ejemplo:

let frutas = [‘pera’,’manzana’,’naranja’,’melocotón’,’plátano’]

console.log (frutas.indexOf(‘manzana’));

este ejercicio devolverá el valor 1, que es la posición que ocupa este elemento dentro del array.

2.7 FUNCIÓN from

La función **from()** lo que hace es convertir en array el elemento iterable. Un elemento iterable es todo aquel que se puede recorrer, como un String. Esta función la utilizaremos mucho cuando utilicemos el DOM, y podamos interactuar con los elementos de una página html.

Veamos un ejemplo:

let word = ‘murcielago’;

console.log (Array.from(word));

esta función nos mostrará la palabra murciélago como si fuera un array de letras.

2.8 FUNCIÓN split

El método split() coge una cadena y devuelve una matriz de cadenas, dividiendo la cadena original en función de un carácter separador proporcionado. Este método devuelve las subcadenas en forma de matriz. Veamos un ejemplo:

let frase = ‘Hola mundo’;

let vector = frase.split(“ “);

Esta última instrucción busca un espacio en blanco y crea un array donde cada palabra dentro de la frase, se convierte en un elemento del array. De esta forma el resultado será el siguiente:

vector = [“Hola”,”mundo”]

esto es un array con dos elementos y cada uno incluye la palabra de la frase.

Si lo que pasamos es una cadena vacía “”, lo que obtendremos como resultado será un array formado por cada uno de los caracteres de la frase, de la siguiente forma:

vector = [“H”,”o”,”l”,”a”,” “,”m”,”u”,”n”,”d”,”o”]

2.9 FUNCIÓN splice

El método splice() cambia el contenido de un array eliminando elementos existentes y/o agregando nuevos elementos. Veamos la sintaxis de esta función:

array.splice(start[, deleteCount[, item1[, intem2[, …]]]]);

Los parámetros asociados a esta función son los siguientes, donde solo el primero es obligatorio, el resto son opcionales:

- **start**: es el índice donde comenzará a cambiar el array (con 0 como origen). Si es mayor que la longitud del array, el punto inicial será la longitud del array. Si es negativo, emperzará esa cantidad de elementos contando desde el final.
- **defaultCount**: es un entero indicando el número de elementos a eliminar del array antiguo. Si este elemento se omite, o su valor es mayor que el número de elementos restantes desde start, entonces todos los elementos desde start hasta el final del array serán eliminados. Si ese elemento es igual a 0 o negativo, no se eliminará ningún elemento, en este caso se debe especificar al menos un nuevo elemento.
- **Item1, item2,** …: Estos son los elementos que se agregarán al arry, empezando en el índice start. Si no se especifica, solamente se eliminarán elementos del array.

Veamos ejemplos asociados al uso de este método:

let elementos = [‘angel’, ‘clown’, ‘mandarin’, ‘arlequin’];

let borrado = elementos.splice(2,0,’sirena’);

en este caso no se eliminan elementos, se añade el elemento ‘sirena’ desde la posición 2. Esto es, el resultado de esta instrucción será **elementos = [‘angel’, ‘clown’, ‘sirena’,‘mandarin’, ‘arlequin’],** y **borrado = [].**

Con la siguiente instrucción, se elimina un elemento desde el índice 3.

let elementos = [‘angel’, ‘clown’, ‘mandarin’, ‘arlequin’];

let borrado = elementos.splice(3,1);

en este caso **borrado = [‘arlequin’]**, y **elementos = [‘angel’,’clown’,’mandarin’].**

Y por último con la siguiente instrucción se elimina un elemento desde el índice 2 y se añaden dos elementos.

let elementos = [‘angel’, ‘clown’, ‘mandarin’, ‘arlequin’];

let borrado = elementos.splice(2,1,’sirena’,’delfín’);

en este caso **borrado = [‘mandarin’]**, y **elementos = [‘angel’,’clown’,’sirena’,’delfín’,’arlequin’].**

2.10 FUNCIÓN sort

El método sort() lo que hace es ordenar alfabéticamente los elementos de un array. Como ya vimos cuando trabajamos con valores numéricos esto no funciona, con lo que debemos ordenar a través de los índices como ya vimos. Sin embargo, teniendo en cuenta que en muchas ocasiones necesitaremos ordenar arrays numéricos, a la función sort se le puede pasar un parámetro que será una función y que permitirá ordenar valores numéricos. A estas funciones que se ejecutan dentro de otra función se les denomina callback. Como podemos ver es similar al uso de las interfaces compare y compareTo que usamos en Java.

Veamos cómo podemos hacer que un array de valores numéricos se ordene de mayor a menor y de menor a mayor. Para ello lo que haremos será comparar dos valores que estén contiguos y para ello los restaremos, en función de si el resultado es positivo, negativo o cero; los valores se intercambiarán o no. Veamos cómo implementarlo en los siguientes ejemplos:

```
<script>
let numeros =  [8,3,5,4,10,22]
alert(numeros.sort())
alert(numeros.sort((a,b)=>a-b))

    </script>
```

Si queremos ordenarlos de menor a mayor, bastará cambiar la condición que queremos comprobar.

```
<script>
let numeros =  [8,3,5,4,10,22]
alert(numeros.sort())
alert(numeros.sort((a,b)=>b-a)

    </script>
```

2.11 FUNCIÓN foreach

El método foreach, sirve para recorrer los elementos de un array, y podemos decir mediante un callback que es lo que queremos hacer en ese recorrido. Veamos un ejemplo, donde simplemente vamos a mostrar los elementos del array por pantalla. Como podemos ver funciona de forma similar a los stream que se utilizan en otros lenguajes de programación. Si nos paramos a pensar, realmente en JavaScript no tienen las características de los arrays en Java, más bien funcionan como listas o stream a los que se pueden aplicar funciones.

```
<script>
let numeros =  [8,3,5,4,10,22]
numeros.forEach(x=>alert(x))


    </script>
```

También podemos utilizar un segundo parámetro que estará asociado al índice del array, para ello lo utilizaremos de la siguiente manera.

```
<script>
let numeros =  [8,3,5,4,10,22]
numeros.forEach((x,indice)=>alert(`El número ${x} ocupa la posicion ${indice}`))

</script>
```

Como vemos es otra forma de recorrer un array, sin tener que añadir bucles, ni contadores.

2.12 FUNCIÓN some

El método some, lo que hacer es comprobar si alguno de los elementos de un array cumple una determinada condición. Así devolverá true si encuentra algún elemento que lo cumple y false si ninguno de ellos lo cumple.

```
<script>
let numeros =  [8,3,5,4,10,22]
alert(numeros.some(x=>x>40))
alert(numeros.some(x=>x<40))
</script>
```

Como vemos la primera instrucción nos devuelve false porque no hay ningún elemento del array que sea mayor de 40, y la siguiente instrucción devuelve true porque todos los elementos son menores que 40, si alguno no cumpliera la condición nos devolvería false de nuevo.

2.13 FUNCIÓN every

El método every, lo que hacer es comprobar si todos de los elementos de un array cumple una determinada condición. Así devolverá true si todos los elementos los cumplen y false si alguno de ellos no lo cumple.

En este caso como podemos ver la primera sentencia nos devuelve true porque todos los elementos del array cumplen que son menores que 40, y la segunda nos da false, porque hay valores que no cumplen la condición y son menores de 15.

```
<body>
    <script>
    let numeros =  [10,8,36,29,18]
    alert(numeros.every(x=>x>40))
    alert(numeros.every(x=>x>15))
    </script>
</body>
```

2.14 Función map

El método map, crea un array con los resultados de la llamada a la función indicada aplicados a cada uno de sus elementos. Veamos un ejemplo en el que lo que necesitamos es multiplicar por dos todos los elementos de un array.

```
<body>
    <script>
   let numeros =  [10,8,36,29,18]
   let doble = numeros.map(x=>x*2)
   alert(doble)
       </script>
</body>
```

2.15 FUNCIÓN filter

El método filter, crea un array con los elementos que cumplen una determinada condición. Veamos un ejemplo práctico, donde se creará un array con los valores que son mayores que 10.

```
    <script>
   let numeros =  [10,8,36,29,18]
   let otro = numeros.filter(x=>x>15)
   alert(otro)
       </script>
```

2.16 FUNCIÓN reduce

El método reduce, aplica una función a un acumulador y a cada valor de un array (de izquierda a derecha) para reducirlo a un único valor. Es decir, la función recorrerá el array de izquierda a derecha pasando por todos los elementos acumulativamente. Esta función nos puede venir bien cuando lo que queremos es sumar todos los elementos de un array.

```
    <script>
   let numeros =  [10,8,36,29,18]
   let suma = numeros.reduce((a,b)=>a+b)
   alert(suma)
       </script>
```

Veamos otro ejemplo de uso cuando se realiza sobre objetos. Imaginemos que tenemos un array de objetos, con los nombres de los usuarios y un atributo asociado a si están o no conectados, si queremos contar el número de usuarios que están conectados, podemos utilizar reduce de la siguiente forma.

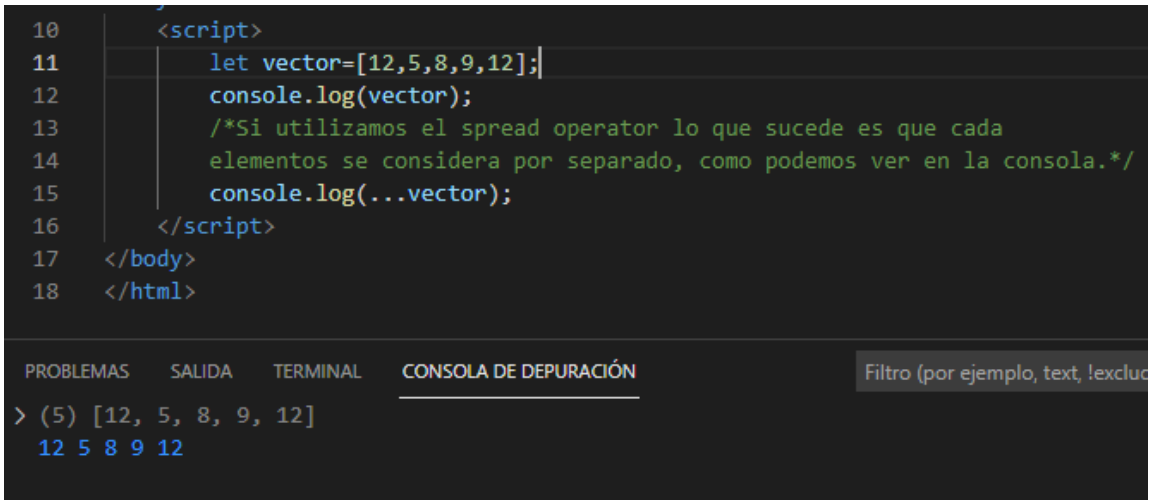
```
    <script>
   let usuarios =  [
    {name:"Manolo",
    online: true},
    {name:"Marta",
    online: true},
    {name:"Ana",
    online: false},
    {name:"Alvaro",
    online: true}]
    const usuariosConectados = usuarios.reduce((contador,x)=>
    {if(x.online)contador++
        return contador},0)
   alert(usuariosConectados)
       </script>
```

Como vemos en este caso, al utilizar una variable nueva dentro de reduce, el contador, debemos inicializarlo para ello, lo que debemos hacer es añadir un nuevo parámetro a reduce que en este caso es un cero, ya que lo que queremos es inicializarlo a cero.

En este caso reduce, recorre el array y guarda sus elementos en la variable user, lo que hace la función creada dentro de reduce es comprobar si el argumento online es true, y si se cumple esta condición, lo que hace es incrementar en uno el contador.

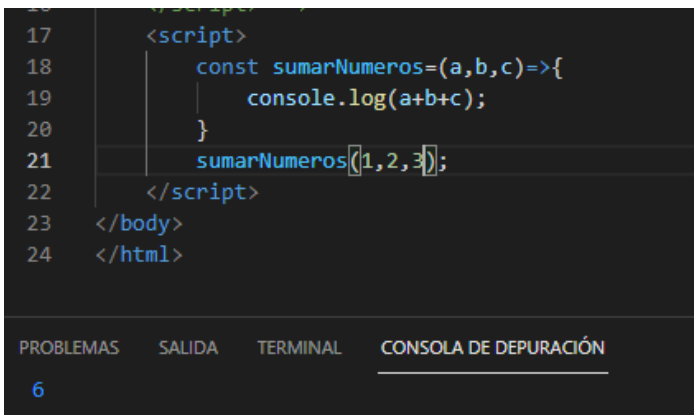
SPREAD OPERATOR (OPERADOR DE EXPANSIÓN)

El operador de expansión se utiliza simplemente utilizando tres puntos, y lo que hace básicamente es expandir los elementos de un array, es decir, considera cada uno de los elementos del vector como un valor separado. Vamos a ver distintos ejemplos de dónde y cómo podemos utilizar el **spread operator**.

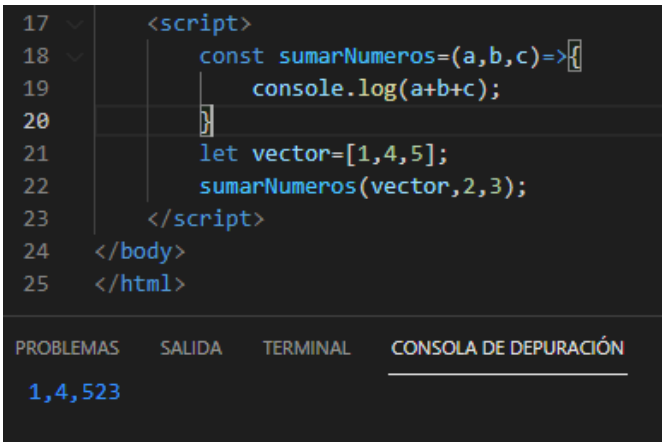


Como podemos ver en el ejemplo anterior, cuando escribimos el array directamente en la consola, nos aparece con los paréntesis y separados por comas, lo que indica que se considera un array. Sin embargo, en el segundo console, cuando utilizamos los tres puntos delante del nombre del array, nos considera cada elemento por separado.

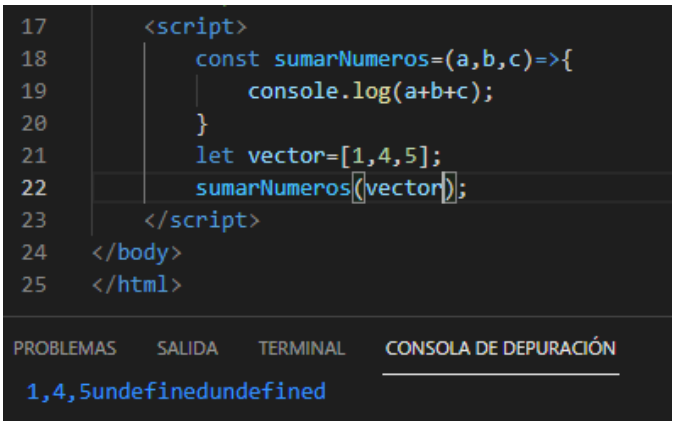
Veamos otro ejemplo donde usarlo. Supongamos que tenemos una función como la siguiente que nos suma tres valores que le pasamos como argumento.



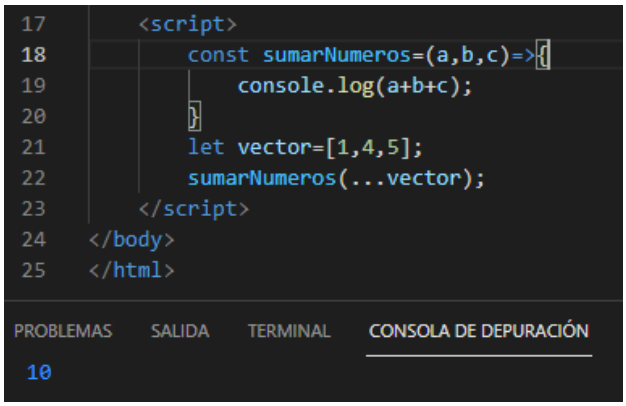
Como vemos en este caso, la suma se realiza correctamente cuando le pasamos tres valores numéricos, pero ¿qué sucede si uno de los elementos que le pasamos es un array? Veamos.



Como vemos aquí lo que hace es concatenar esos valores, pero si lo que le pasamos directamente es el valor, que tiene tres elementos, este es el resultado.

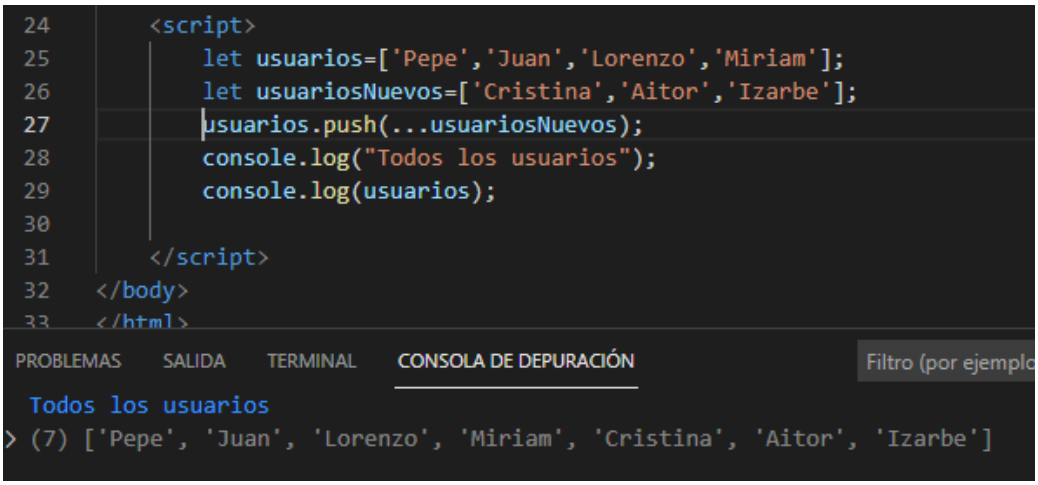


En este caso nos aparece el vector, que se asocia al primer argumento de la variable a, y los otros dos nos aparecen como indefinidos, ya que no pasamos tres argumentos. Si lo que nos interesa es sumar los tres elementos del array, lo que tenemos que hacer es lo siguiente.



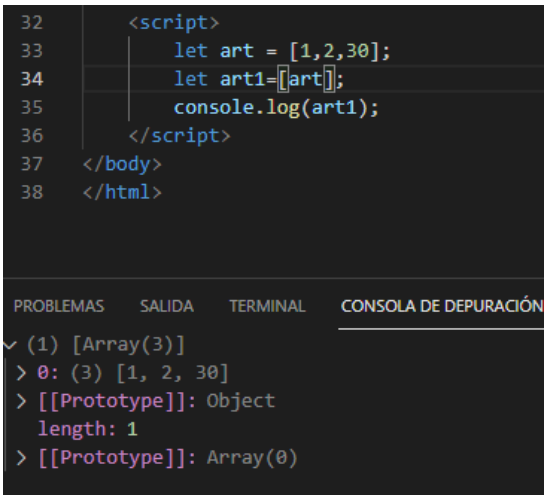
Como podemos ver, ahora nos hace la suma de esos tres elementos, ya que el spread operator lo que hace es pasar los tres valores del array como si fueran tres elementos por separado.

Veamos otro ejemplo donde utilizar este operador, en este caso lo que haremos será añadir un array a otro array. En el primer array tenemos los usuarios que ya son miembros de nuestro centro, y tenemos un segundo array donde nos encontramos los nuevos alumnos que nos interesa añadir. Para añadir los nuevos usuarios al array inicial, podemos utilizar la función push para cada uno de ellos, o un bucle for que los añada, esto suele ser bastante pesado de escribir. Veamos cómo podemos utilizar el spread operator para hacer esto.



Esta forma como podemos ver es muy rápida y funciona correctamente. Estos elementos del array se pueden añadir también en una posición determinada, para ello utilizaremos la función splice, como hemos visto antes.

Otro de los usos que tiene este operador es para copiar arrays, es decir, para crear un segundo array con los mismos elementos que el primero. Si creamos como vemos abajo, un array asociado al array que ya existe, lo que nos encontramos es un array de un array, que no es lo que pretendíamos.



Para solucionar esto, podemos utilizar el spread operator de la siguiente forma, donde como vemos que sí que nos aparece un array idéntido al primero.

```
32      <script>
33          let art = [1,2,30];
34          let art1=[...art];
35          console.log(art1);
36      </script>
37  </body>
38  </html>
```

```
PROBLEMAS  SALIDA  TERMINAL  CONSOLA DE DEPURACIÓN
> (3) [1, 2, 30]
```

Veamos cómo utilizarlo también para concatenar arrays y guardarlos en un nuevo array, podemos utilizar el método concat, es un método que nos sirve para concatenar arrays, veámoslo.

```
32      <script>
33          let art = [1,2,30,10,24];
34          let art1=[4,6,7];
35          let arrayConcatenado=art.concat(art1);
36          console.log(arrayConcatenado);
37      </script>
38  </body>
```

```
PROBLEMAS  SALIDA  TERMINAL  CONSOLA DE DEPURACIÓN
> (8) [1, 2, 30, 10, 24, 4, 6, 7]
```

Otra forma de hacerlo sería la siguiente, donde como vemos obtenemos el mismo resultado.

```
32      <script>
33          let art = [1,2,30,10,24];
34          let art1=[4,6,7];
35          //  let arrayConcatenado=art.concat(art1);
36          let arrayConcatenado=[...art,...art1];
37          console.log(arrayConcatenado);
38      </script>
```

```
PROBLEMAS  SALIDA  TERMINAL  CONSOLA DE DEPURACIÓN
> (8) [1, 2, 30, 10, 24, 4, 6, 7]
```

Otra función del spread operator es cuando queremos enviar parámetros a una función, pero no sabemos exactamente cuántos parámetros se van a pasar. Para ello utilizaríamos los parámetros denominados rest. En este caso, la función lo que hace es generar un array vacío si no pasamos parámetros, y si pasamos parámetros se genera un array cuyos elementos son cada uno de los parámetros pasados.

```
42          const restParametros = (...numbers)=>{
43              console.log(numbers);
44          }
45          restParametros(); //Podemos no pasar parámetos y tendremos un array vacío
46          //Si enviamos varios parámetros, obtendríamos un array con esos parámetros
47          restParametros(1,2,4,5,6);
48      </script>
49  </body>
```

PROBLEMAS SALIDA TERMINAL CONSOLA DE DEPURACIÓN Filtro (por ejemplo, text, !exclude)

```
> (0) []
> (5) [1, 2, 4, 5, 6]
```

Si utilizamos el spread operator dentro de la función, lo que nos sucede es que obtenemos cada uno de los parámetros pasados, sin que estén incluidos en el array.

```
40      <script>
41
42          const restParametros = (...numbers)=>{
43              console.log(...numbers);
44
45          }
46          restParametros(); //Podemos no pasar parámetos y tendremos un array vacío
47          //Si enviamos varios parámetros, obtendríamos un array con esos parámetros
48          restParametros(1,2,4,5,6);
49      </script>
```

PROBLEMAS SALIDA TERMINAL CONSOLA DE DEPURACIÓN Filtro (por ejemplo, text, !exclude)

```
1 2 4 5 6
```

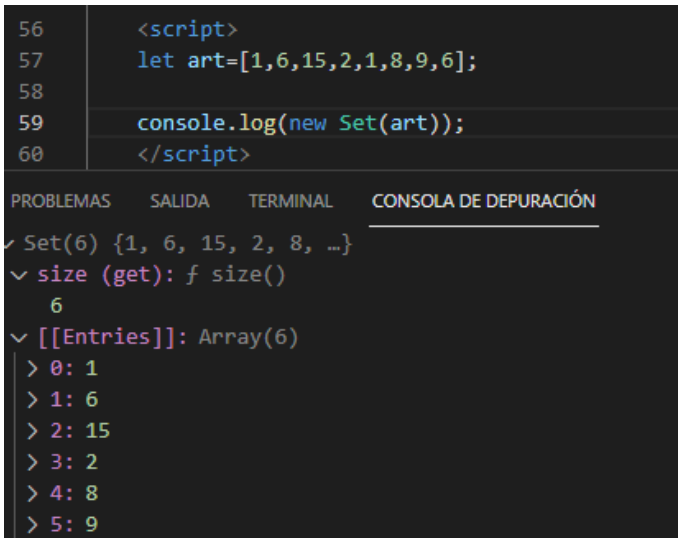
Veamos cómo utilizarlo en la librería Math, con las funciones max y min; que calculan respectivamente el máximo y el mínimo de un listado que pasamos. En este caso lo que vamos a hacer es utilizar el spread operator para calcular el máximo y el mínimo de un array.

```
51          let art = [1,2,30,10,24];
52          console.log(Math.max(...art));
53          console.log(Math.min(...art));
54      </script>
55  </body>
56  </html>
```

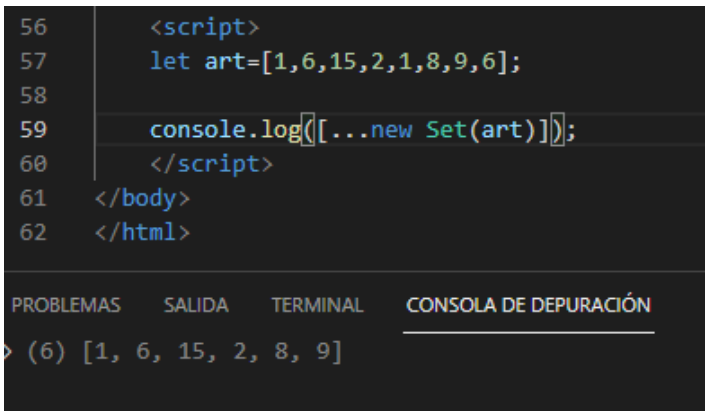
PROBLEMAS SALIDA TERMINAL CONSOLA DE DEPURACIÓN

```
30
1
```

Veamos ahora cómo eliminar elementos que estén repetidos en un array, para ello utilizaremos la clase Set, que no permite que existen elementos repetidos en su interior. Una forma de hacerlo sería lo siguiente:



Como vemos en este caso en el Set no tenemos valores repetidos porque los objetos de tipo Set no permiten que existan valores positivos. Los valores 1 y 6 que estaban repetidos se han eliminado. Para crear un array con los elementos de otro array que no estén repetidos lo que haremos será lo siguiente:



Todos estos métodos y el uso de los arrays nos van a servir cuando veamos el DOM y empecemos a utilizar los objetos en los archivos html.

.