

Unidad 4.- Programación con arrays, funciones y objetos definidos por el usuario:

- a) Funciones predefinidas del lenguaje.
- b) Llamadas a funciones. Definición de funciones.
- c) Arrays.
- d) Creación de objetos. Definición de métodos y propiedades.

a) Funciones predefinidas del lenguaje.

Las funciones predefinidas vienen dadas por el lenguaje. Vamos a ver algunas de estas funciones:

parseInt(*cadena* [, *base*]) devuelve un número entero resultante de convertir el número representado por la cadena a entero. Con base se indica la base en la que se expresa el número, si no se indica la base, tomará ésta en función de los primeros caracteres, si empieza por cero la base será 8, octal, si empieza por 0x la base será 16, hexadecimal, y por cualquier otro dígito la base es 10, decimal. Si la cadena empieza por un/os dígito/s y a continuación encuentra un carácter que no es dígito la conversión la realiza con el valor de la cadena hasta el primer carácter no dígito.

parseFloat(*cadena*) devuelve un número en coma flotante, que es el valor representado por la cadena.

isNaN(*valor*) devuelve un valor lógico que indica si el valor es NaN.

eval(*expresión*) devuelve el valor de la expresión, si realizamos la concatenación de cadenas y ésta representa una variable u objeto, va a devolver la referencia a la variable u objeto.

Number(*cadena*) devuelve un número con el valor de la cadena.

String(*valor*) devuelve una cadena con el valor indicado.

isFinite(*valor*) devuelve un valor lógico que nos indica si el valor es finito, devuelve false cuando el valor es infinito, -infinito o NaN.

escape(*cadena*) devuelve una cadena que es una copia de la original, en la cual los caracteres no ASCII aparecen escapados, con \xx.

unescape(*cadena*) devuelve una cadena que es una copia de la original en la cual los caracteres escapados aparecen con su valor.

b) Definición de funciones. Llamadas a funciones.
Para realizar la definición de una función deberemos poner:

```
function nombre-función( [lista-parámetros] ){  
    instrucciones  
}
```

Nos permite definir una función en la cual la lista de parámetros va a ser el nombre de los mismos. Dentro de las instrucciones nos vamos a encontrar con la instrucción **return**(en las funciones vamos a poner una única instrucción **return**) que seguida de un valor devolverá dicho valor, si no se pone el valor no va a devolver nada, o también se puede poner en este caso para que no devuelva nada **return null**. Admite recursividad. Los **parámetros siempre se van a pasar por valor**. Una forma de pasar parámetros por referencia es pasar una matriz o un objeto.

001	function sumando(primero, segundo){
002	var suma;
003	suma = primero + segundo;
004	return suma;
005	}

factorial.js

001	function factorial(numero){
002	if (numero==0){
003	return 1;
004	} else {
005	return numero*factorial(numero-1);
006	}

007 }

Para realizar una llamada a una función deberemos poner

nombre-función([valores-parámetros])

La llamada a la función la podemos poner sola en una línea, si no devuelve valores o si los queremos ignorar o bien en una expresión del mismo tipo que el valor devuelto por la función.

001 result=mifuncion(indice,suma) ;

Otra forma de declarar una función es:

```
var nombre-variable=function() {  
    cuerpo-función  
}
```

Declarar una función que es asignada a una variable. La diferencia entre la primera y la segunda declaración está en el tratamiento. Mientras que la primera declaración de la función se compila al inicio y se mantiene hasta que se necesita la segunda es compilada y ejecutada según se va leyendo.

```
001 var mia =function() {  
002     console.log(arguments.length) ;  
003     console.log(arguments) ;  
004     for(var i=0; i< arguments.length ; i++){  
005         console.log(arguments[i]) ;  
006     }
```

007	<code>return;</code>
008	<code>}</code>
009	<code>mia("hola","prueba",13,45);</code>

Otra forma de definir una función es:

**window[nombre-función]=new Function(lista-argumento,
cuerpo-función);**

De esta forma se pueden crear funciones de manera dinámica, ya que tanto el nombre de la función, como los parámetros y el cuerpo de la función pueden estar contenidas en variables.

001	<code>var nombreFuncion='cuadrado';</code>
002	<code>var argumentoFuncion='x';</code>
003	<code>var codigoFuncion='return x * x;'</code>
004	<code>window[nombreFuncion]=new Function(argumentoFuncion,codigoFuncion);</code>
005	<code>alert(window[nombreFuncion](3));</code>

Existe la posibilidad de declarar una función sin parámetros, aunque luego se la pueda pasar un número indeterminado de parámetros. En este caso dentro de la función vamos a tener una variable llamada **arguments**, que va a ser un array.

001	<code>function mia() {</code>
002	<code> console.log(arguments.length);</code>
003	<code> console.log(arguments);</code>
004	<code>for(var i=0; i< arguments.length ; i++){</code>
005	<code> console.log(arguments[i]);</code>
006	<code>}</code>
007	<code>return;</code>
008	<code>}</code>
009	<code>mia("hola","prueba",13,45);</code>

Parámetros opcionales

```
function nombre-función(parámetro1=valor1, ..) {  
    cuerpo  
}
```

Se deben poner los parámetros opcionales al final

ejemplo-04-06.js

001	function opera(first, second=0, thersty=2) {
002	return first + second + thersty;
003	}
004	document.writeln(opera(45)+" ");

Podemos poner que a partir de un determinado parámetro vamos a poder tener un número indeterminado de parámetros más, que se van a agrupar en un parámetro que va a ser tratado como un array (estamos declarando un número mínimo de parámetros, que son los que van al inicio y luego un número indeterminado de parámetros), al poner ese parámetro le vamos a poner un prefijo de tres puntos seguidos. La forma de declararlo será

```
function nombre-función(parámetro1, ...parámetro2) {  
    cuerpo  
}
```

El parámetro2 recibe todos los parámetros que se le pasan a la función a partir del segundo y dentro del cuerpo de la función se trata como un array.

ejemplo-04-05.js

001	function operaciones(one, two,...other) {
002	var sumar = one + two;
003	for (var i=0; i < other.length;i++)
004	sumar += other[i];
005	return sumar;
006	}
007	document.writeln(operaciones (2,4,6,8)+" ");

Funciones simples

function (parámetros)instrucción-del-return

ejemplo-04-03.js

001	function sumar(primer,segundo) {
002	var suma = primero + segundo;
003	return suma
004	}
005	var uno = sumar(12,24);
006	function sumando(primer,segundo) primero+segundo;
007	var dos = sumando(24,12);
008	document.writeln(uno +" ");
009	document.writeln(dos +" ");

Funciones que devuelven varios valores

return [lista-valores]

Los valores los puede recibir un array o bien varias variables, en este caso se deben poner los nombres de las variables encerradas entre corchetes.

ejemplo-04-02.js

001	function operaciones(primer,segundo) {
002	var suma = primero + segundo;
003	var resta = primero - segundo;
004	var multi = primero * segundo;

005	<code>var divi = primero / segundo;</code>
006	<code>var poten = primero ** segundo;</code>
007	<code>return[suma,resta, multi, divi, poten]</code>
008	<code>}</code>
009	<code>var todos=new Array();</code>
010	<code>todos=operaciones(8,2);</code>
011	<code>[uno,dos,tres, cuatro,</code> <code>cinco]=operaciones(4,2);</code>
012	<code>document.writeln(uno+"
");</code>
013	<code>document.writeln(dos+"
");</code>
014	<code>document.writeln(tres+"
");</code>
015	<code>document.writeln(cuatro+"
");</code>
016	<code>document.writeln(cinco+"
");</code>
017	<code>for(var i=0; i < todos.length ; i++){</code>
018	<code>document.writeln(todos[i]+"
");</code>
019	<code>}</code>

Funciones flecha.

Si tenemos una función del tipo

```
function nombre-función([parámetros]) {
    cuerpo-función;
    return valor;
}
```

Se puede transformar en una función anónima, haciéndola una función flecha, para lo cual deberemos realizar la siguiente transformación

```
var nombre-función= ([parámetros]) => {
    cuerpo-función;
    return valor;
}
```

Estas funciones admiten cualquier tipo de parámetros que hemos visto anteriormente, a excepción de **arguments** que no va a existir de por si en la función.

Si la función solamente tiene una instrucción **return** no es necesario poner las llaves ni poner la palabra clave **return**.

```
var nombre-función= ([parámetros]) =>expresión;
```

Si solamente se tiene un parámetro no es necesario poner los paréntesis.

```
var nombre-función= parámetro => {  
    cuerpo-función;  
    return valor;  
}
```

o bien

```
var nombre-función= parámetro =>expresión;
```

ejemplo-04-04.js

001	function suma (primero, segundo){
002	return primero + segundo;
003	}
004	var suma1 =(primero, segundo)=>{
005	return primero + segundo;
006	};
007	var suma2 =(primero,segundo)=> primero + segundo;
008	function doble(unos) {
009	return uno *2;
010	}
011	var doble1 =(unos)=>{
012	return uno *2;
013	};
014	var doble2 = uno =>{
015	return uno *2;
016	};
017	var doble3 =(unos)=> uno *2;
018	var doble4 = uno => uno *2;
019	
020	function operaciones(one, two,...other){
021	var sumar = one + two;

022	<code>for(var i=0; i < other.length;i++)</code>
023	<code>sumar += other[i];</code>
024	<code>return sumar;</code>
025	<code>}</code>
026	<code>var operaciones1 =(one, two,...other)=>{</code>
027	<code>var sumar = one + two;</code>
028	<code>for(var i=0; i < other.length;i++)</code>
029	<code>sumar += other[i];</code>
030	<code>return sumar;</code>
031	<code>}</code>
032	<code>function opera(first, second=0, thersty=2){</code>
033	<code>var res1=first + second + thersty;</code>
034	<code>var res2=first * thersty;</code>
035	<code>var res3=first / thersty;</code>
036	<code>return[res1,res2,res3];</code>
037	<code>}</code>
038	<code>var operal=(first, second=0, thersty=2)=>{</code>
039	<code>var res1=first + second + thersty;</code>
040	<code>var res2=first * thersty;</code>
041	<code>var res3=first / thersty;</code>
042	<code>return[res1,res2,res3];</code>
043	<code>}</code>
044	<code>document.writeln(suma(2,4)+"
");</code>
045	<code>document.writeln(suma1(2,4)+"
");</code>
046	<code>document.writeln(suma2(2,4)+"
");</code>
047	<code>document.writeln(doble(5)+"
");</code>
048	<code>document.writeln(doble1(5)+"
");</code>
049	<code>document.writeln(doble2(5)+"
");</code>
050	<code>document.writeln(doble3(5)+"
");</code>
051	<code>document.writeln(doble4(5)+"
");</code>
052	<code>document.writeln(operaciones(2,4,6,8)+"
");</code>
053	<code>document.writeln(operaciones1(2,4,6,8)+"
");</code>
054	<code>[ope1,ope2,ope3]= opera(45);</code>
055	<code>document.writeln(ope1+"
");</code>
056	<code>document.writeln(ope2+"
");</code>
057	<code>document.writeln(ope3+"
");</code>
058	<code>[ope1,ope2,ope3]= operal(45);</code>
059	<code>document.writeln(ope1+"
");</code>
060	<code>document.writeln(ope2+"
");</code>

```
061 document.writeln( ope3+"<br />" );
```

Función de generador

```
function *nombre-función([lista-parámetros]) {  
    cuerpo-función  
}
```

La forma de hacer referencia a la función es

```
nombre-variable=nombre-función([lista-valores])
```

Para que se ejecute la función y obtener el valor devuelto deberemos poner

```
nombre-variable.next().value
```

El método **next()** hace que se ejecute la función y con la propiedad **value** obtener el valor devuelto.

ejemplo-04-50.js

001	function * sumatorio() {
002	var suma=0;
003	var i;
004	for (i=0;i < arguments.length;i++) {
005	suma+=arguments[i];
006	}
007	return suma;
008	}
009	
010	var sumando=sumatorio(12,23,13,45,25,56,37,53,74, 83,16,94,84);
011	document.writeln(sumando.next().value);

Dentro de la función podemos poner la instrucción **yield**

```
yield expresión
```

Detiene la ejecución de la función hasta que se vuelva a llamar y devuelve un objeto de tipo **yield** compuesto por dos propiedades, que

son: **value** que se corresponde con el valor de la expresión y **done** que nos indica si se ha terminado la función a través de un valor lógico.

Para mandar ejecutar la función y que se vaya reanudando la función tenemos el método **next()**, que además devuelve el objeto **yield** de la función.

ejemplo-04-51.js

001	function* sumatorio() {
002	var suma=0;
003	var i;
004	for (i=0;i < arguments.length;i++) {
005	suma+=arguments[i];
006	yield suma;
007	}
008	return suma;
009	}
010	var sumando=sumatorio(12,23,13,45,25,56,37,53,74, 83,16,94,84);
011	var dato=sumando.next();
012	while (!dato.done) {
013	document.writeln(dato.value);
014	dato=sumando.next();
015	}

Además también podemos utilizar la instrucción

yield *nombre-función-generadora(lista-valores)

Realiza una llamada a la función generadora con el valor del parámetro

ejemplo-04-52.js

001	function* duplicado(numero) {
002	yield numero*2;
003	}
004	function* sumatorio() {
005	var suma=0;
006	var i;
007	for (i=0;i < arguments.length;i++) {

008	<code>suma+=arguments[i];</code>
009	<code>yield* duplicado(suma);</code>
010	<code>}</code>
011	<code>return suma;</code>
012	<code>}</code>
013	<code>var</code> <code>sumando=sumatorio(12,23,13,45,25,56,37,53,74,</code> <code>83,16,94,84);</code>
014	<code>var dato=sumando.next();</code>
015	<code>while(!dato.done){</code>
016	<code>document.writeln(dato.value);</code>
017	<code>dato=sumando.next();</code>
018	<code>}</code>

c) Arrays.

Un array es un conjunto de celdas, que almacenan diversos valores y que son nombrados mediante un nombre y la posición que ocupan dentro de la estructura.

En JavaScript los arrays se empiezan a numerar por el 0. Un array puede contener valores de diferentes tipos localizados en distintas posiciones.

En los arrays la dimensión no es importante, ya que en cualquier momento se puede modificar añadiendo un nuevo elemento al array.

Para realizar la declaración de un array podemos utilizar diversos formatos como son:

```
var nombre-array= new Array()
```

Nos declaramos un array sin dimensión.

001	<code>var tabla=newArray();</code>
-----	------------------------------------

```
var nombre-array= new Array(lista-valores)
```

Nos declaramos un array, que va a tener tantos elementos como valores se indican, los valores están separados por comas.

```
001 var matriz=newArray("Juan","Pedro",13,true);
```

var nombre-array = new Array(número-elementos)

Nos declara un array con tantos elementos como se indican.

```
001 var arreglo=newArray(9);
```

var nombre-array=[]

Nos declaramos un array sin dimensión.

```
001 var datos=[];
```

var nombre-array=[lista-valores]

Nos declaramos un array, que va a tener tantos elementos como valores se indican, los valores están separados por comas.

```
001 var conjunto=[13,56,78,"Luis"];
```

Para acceder a un elemento del array deberemos poner

nombre-array[posición]

```
001 conjunto[2]
```

Para añadir un elemento bastará con asignar valor a un elemento que ocupa una posición posterior al último elemento.

```
001 conjunto[9]="Leonor";
```

Los arrays disponen de las siguientes propiedades:

♦ **length**: contiene el número de elementos del array.

Los arrays disponen de los siguientes métodos:

- ◆ **shift()**: devuelve el valor del primer elemento del array y le elimina.
- ◆ **pop()**: devuelve el valor del último elemento del array y le elimina.
- ◆ **push(lista-valores)**: añade los valores indicados al final del array, cada uno de ellos en una nueva posición.
- ◆ **unshift(lista-valores)**: añade los valores indicados al inicio del array, cada uno de ellos en una nueva posición, desplazando los que había en esas posiciones.
- ◆ **splice(inicio, nºelemento[, lista-valores])**: elimina a partir de la posición indicada por inicio tanto elementos como se indican, al mismo tiempo se pueden añadir los valores indicados, cada uno en un elemento, a partir de la posición indicada.
- ◆ **reverse()**: devuelve un array con los elementos en orden inverso.
- ◆ **sort()**: ordena el array y devuelve una copia del array ordenado.
- ◆ **slice(inicio[, último])**: devuelve un array con los elementos existen entre el inicio y el final o bien hasta el último, excluido este último.
- ◆ **indexOf(valor[, inicio])**: devuelve la posición que ocupa la primera aparición del valor indicado dentro del array, empezando la búsqueda por el primer elemento o por la posición de inicio; la búsqueda se realiza de inicio a fin. Si no encuentra el valor en el array devuelve -1.

- ◆ **lastIndexOf(valor[,inicio]):** devuelve la posición que ocupa la primera aparición del valor indicado dentro del array, empezando la búsqueda por el último elemento o por la posición de inicio; la búsqueda se realiza del final al inicio. Si no encuentra el valor en el array devuelve el valor-1.
- ◆ **includes(valor[,inicio]):** devuelve un valor lógico que nos indica si el valor se encuentra en el array.
- ◆ **concat(array):** devuelve un array que es la concatenación del array del objeto con el array suministrado.
- ◆ **join(caracter):** devuelve una cadena con todos los elementos del array separados por el carácter indicado.
- ◆ **forEach(función):** para cada elemento del array llama a la función con tres parámetros, que son: el valor, la posición y el array completo.
- ◆ **fill(valor [, inicio [, final]]):** devuelve un nuevo array en el que se han rellenados todos los elementos que tiene el array con el valor indicado, si indicamos inicio se indica a partir de qué posición se inicia el relleno y si indicamos final se indica en qué posición se para el relleno, en esa posición no se produce el relleno.
- ◆ **find(nombre-función):** se ejecuta la función indicada para cada uno de los elementos de la función, esta función va a tener tres parámetros que se corresponden con el valor, la posición y el array. Esta función va a devolver el valor del primer elemento encontrado, si devuelve false ese valor no es tenido en cuenta.

- ◆ **findIndex(*nombre-función*)**: se ejecuta la función indicada para cada uno de los elementos de la función, esta función va a tener tres parámetros que se corresponden con el valor, la posición y el array. Esta función va a devolver la posición del primer elemento encontrado, si devuelve false ese valor no es tenido en cuenta.
- ◆ **entries()**: devuelve un nuevo array bidimensional que va a tener en cada fila la referencia a los elementos del array inicial y en las columnas va a tener la posición del elemento y el valor del elemento.
- ◆ **keys()**: devuelve un nuevo array que va a contener las posiciones de todos los elementos del array inicial.
- ◆ **copyWithin(*posición* [, *inicio* [, *final*]])**: devuelve un nuevo array, con el mismo número de elementos que tiene el array inicial, en el cual se van a copiar elementos a partir de la posición indicada (positivo se empieza a contar desde el principio, y negativo se empieza a contar desde el final. El primer elemento empezando por la izquierda es cero y el primer elemento empezando por la derecha es -1) e inicialmente los elementos se toman a partir del primer elemento a no ser que se indique la posición en la que se empiezan a copiar. También se puede indicar en qué posición se dejan de coger elementos para copiar, esa posición no se incluye.
- ◆ **some(*nombre-función* | **function** ([*parámetros*]){*cuerpo*})**: devuelve un array que se obtiene al ejecutar la función para

cada uno de los valores del array inicial(se ejecuta una vez por cada elemento del array.

- ◆ **every(nombre-función):** devuelve un valor lógico que nos indica si todos los elementos del array cumplen la condición establecida en la función indicada en la instrucción return. Esa función tiene un parámetro que hace referencia a cada uno de los valores del array. Se llama a la función una vez por cada elemento del array.
- ◆ **filter(nombre-función):** devuelve un array con los elementos del array que cumplen la condición establecida en la función indicada en la instrucción return. Esa función tiene un parámetro que hace referencia a cada uno de los valores del array. Se llama a la función una vez por cada elemento del array.
- ◆ **flat([nivel]):** devuelve un array con los elementos del array (array multidimensional) teniendo este nuevo array una dimensión inferior o tantas dimensiones inferiores como indique nivel.
- ◆ **reduce(nombre-función [, valor-inicial]):** devuelve un valor correspondiente a realizar las operaciones que se indican en la función, esta función tiene dos parámetros, que son un acumulador y el valor de un elemento del array, a esta función se la llama una vez por cada elemento que tenga el array. El valor inicial es el valor que se le asigna de forma inicial al acumulador, si no se le asigna un valor inicial va a

tomar el valor del primer elemento del array como valor inicial y se ejecuta la función a partir del segundo elemento.

- ♦ **reduceRight(*nombre-función* [, *valor-inicial*])**: devuelve un valor correspondiente a realizar las operaciones que se indican en la función, esta función tiene dos parámetros, que son un acumulador y el valor de un elemento del array, a esta función se la llama una vez por cada elemento que tenga el array, empezando por el último hacia el primero. El valor inicial es el valor que se le asigna de forma inicial al acumulador, si no se le asigna un valor inicial va a tomar el valor del último elemento del array como valor inicial y se ejecuta la función a partir del elemento anterior.
- ♦ **toString()**: devuelve una cadena con los valores de los elementos del array separados por comas.
- ♦ **toLocaleString([*código-país*])**: devuelve los elementos del array con el formato del país indicado, o el establecido por defecto, separados por comas. (cuidado con el separador de los números reales, ya que en español es la coma y coincide con el separador de elementos).

Métodos aplicados a Array.

- ♦ **of(lista-valores)**: crea un nuevo array con tantos valores como se indican, y cada uno de los elementos del array es uno de los valores indicados, los valores son números. Se diferencia de crear un array con `new Array` en que cuando utilizamos un único parámetro numérico, en este caso se

crea un array con el número de elementos indicados y con of
se crea un array con un elemento que tiene ese valor.

- ♦ **from(objeto-map|objeto-set):** convierto los objetos indicados en un array.

001	<code>var numeros=newArray(12,23,25,14);</code>
002	<code>var cadena="";</code>
003	<code>numeros.forEach(function(valor,indice, arreglo){</code>
004	<code>cadena+="valor: "+valor.toString()+"</code> <code>Indice: "+indice.toString()+" \n";</code>
005	<code>});</code>
006	<code>alert(cadena);</code>

001	<code>var numeros=newArray(12,23,25,14);</code>
002	<code>var cadena="";</code>
003	<code>numeros.forEach(manejo);</code>
004	<code>function manejo(valor,indice, arreglo){</code>
005	<code>cadena+="valor: "+valor.toString()+"</code> <code>Indice: "+indice.toString()+" \n";</code>
006	<code>}</code>
007	<code>alert(cadena);</code>

Dentro de las cadenas tenemos el siguiente método relacionado con los arrays.

- ♦ **split(caracter):** devuelve un array cuyos elementos están constituidos por los caracteres de la cadena que están separados por el carácter indicado.

Arrays multidimensional

Si queremos tener arrays con más de una dimensión (lo normal es tener arrays bidimensionales) vamos a tener varias posibilidades que vamos a ir viendo una a una.

En la **primera posibilidad** vamos a inicializar el array incluyendo otros arrays dentro, los valores de un array se incluyen entre corchetes.

```
001 var nuevo = [ ["Juan", "Pedro"],  
    ["Antonio", "Felix"] ];
```

Para acceder a los elementos del array vamos a poner nombre del array entre corchetes la fila y entre otros corchetes la columna.

nombre-array[fila][columna]

```
001 document.writeln(nuevo[0][0]+"<br />");
```

En la **segunda posibilidad** vamos a inicializar el array incluyendo otros arrays vacíos y luego asignamos valores a los elementos del array

```
001 var nuevo=[[], []];  
002 nuevo[0][0]="Juan";  
003 nuevo[0][1]="Pedro";  
004 nuevo[1][0]="Antonio";  
005 nuevo[1][1]="Felix";
```

En la **tercera posibilidad** vamos a declarar un array y luego cada uno de los elementos del mismo va a ser un nuevo array.

```
001 var mitabla =new Array();  
002 mitabla[0]=new  
    Array("Juan", "Pedro", "Antonio");  
003 mitabla[1]=new Array("Felix", "Luis", "Ana");  
004 mitabla[2]=new Array("Rosa", "Laura", "Rocio");  
005 for(var i=0; i < mitabla.length; i++) {  
006     for(var j=0; j < mitabla[i].length; j++) {  
007         document.writeln(mitabla[i][j]+"<br />")  
008     }  
009 }
```

Bucles para arrays

Para obtener todos los valores del array

```
for (nombre of nombre-array)instrucción;
```

Se va a ejecutar una vez por cada uno de los elementos del array y en cada ejecución de la instrucción nombre va a ir tomando cada uno de los valores del array.

Para obtener todos los índices del array

```
for (nombre in nombre-array)instrucción;
```

Se va a ejecutar una vez por cada uno de los elementos del array y en cada ejecución de la instrucción nombre va a ir tomando cada uno de los índices del array

El **objeto Map** nos va a permitir tener un array cuyo índice es un valor de tipo alfanumérico.

Método constructor

◆ **new()**

```
var novedad = new Map();
```

Propiedades

◆ **size**: nos indica el número de elementos que tiene el map.

Métodos

◆ **get(clave)**: devuelve el elemento que tiene esa clave.

◆ **set(clave , valor)**: incluye el valor en el map asociado a la clave indicada.

◆ **has(clave)**: devuelve un valor lógico que nos indica si existe un elemento con esa clave.

- ◆ **delete(*clave*)**: borra el elemento del map que tiene la clave indicada.
- ◆ **clear()**: borra todos los elementos del map.
- ◆ **entries()**: devuelve un objeto iterator que va a tener en cada posición un array con la clave y el valor del elemento del objeto Map inicial.
- ◆ **keys()**: devuelve un objeto iterator con todas las claves del objeto Map.
- ◆ **values()**: devuelve un objeto iterator con todos los valores del objeto Map.
- ◆ **toString()**: devuelve el elemento como una cadena. “[Object Map]”
- ◆ **valueOf()**: devuelve el objeto Map.
- ◆ **forEach(función (*valor, clave, objeto*) { *cuerpo-función* })**: realiza la acción indicada para cada elemento del Map.

ejemplo-04-020.js

001	<code>var nuevo=new Map();</code>
002	<code>function anadir(){</code>
003	<code>var valor=prompt("Introduce un valor");</code>
004	<code>var clave=prompt("Introduce su clave");</code>
005	<code>if(nuevo.has(clave))</code>
006	<code>alert("Ya existe esa clave en el array");</code>
007	<code>else</code>
008	<code> nuevo.set(clave, valor);</code>
009	<code>}</code>
010	<code>function consulta(){</code>
011	<code>var clave=prompt("Introduce su clave");</code>
012	<code>if(nuevo.has(clave))</code>
013	<code> alert("El valor correspondiente a la clave "+</code> <code> clave +" es "+nuevo.get(clave));</code>
014	<code>else</code>

015	<code>alert("NO existe esa clave en el array");</code>
016	<code>}</code>
017	<code>function borrar() {</code>
018	<code>var clave=prompt("Introduce su clave");</code>
019	<code>if(nuevo.has(clave)){</code>
020	<code> nuevo.delete(clave);</code>
021	<code>alert("Valor borrado del array");</code>
022	<code>}else</code>
023	<code>alert("NO existe esa clave en el array");</code>
024	<code>}</code>
025	<code>function numero() {</code>
026	<code>alert("El número de elementos del array es "+</code> <code>nuevo.size.toString());</code>
027	<code>}</code>
028	<code>function todos() {</code>
029	<code> nuevo.clear()</code>
030	<code>alert("Todos los elementos han sido borrados</code> <code>");</code>
031	<code>}</code>
032	<code>function valores() {</code>
033	<code>var todosValores="";</code>
034	<code>var todasClaves="";</code>
035	<code>var conjunto="";</code>
036	<code> nuevo.forEach(function(valor, clave ,</code> <code> mismo) {</code>
037	<code> todosValores+=valor +" \n";</code>
038	<code> todasClaves+=clave +" \n";</code>
039	<code> conjunto+=" clave: "+ clave +"</code> <code>valor; "+ valor +" \n";</code>
040	<code>});</code>
041	<code>alert("Valores \n"+ todosValores);</code>
042	<code>alert("Claves \n "+ todasClaves);</code>
043	<code>alert("todos \n "+ conjunto);</code>
044	<code>}</code>
045	<code>function valor() {</code>
046	<code>alert("valueOf() \n"+ nuevo.valueOf());</code>
047	<code>}</code>
048	<code>function cadena() {</code>
049	<code>alert("toString() \n "+ nuevo.toString());</code>
050	<code>}</code>

Bucle for para el Objeto Map

for (*nombre of objeto-map*) instrucción;

Se ejecuta la instrucción tantas veces como elementos tiene el objeto map, en cada una de las ejecuciones nombre toma la dupla clave, valor en un array.

for ([clave, valor] of *objeto-map*) instrucción;

Se ejecuta la instrucción tantas veces como elementos tiene el objeto map, en cada una de las ejecuciones clave y valor toman los valores del elemento del objeto map.

Esto que hemos visto con el objeto Map también se puede hacer con un array normal, como se muestra en el siguiente ejemplo, no podremos acceder al array mediante un índice, sino mediante un valor alfanumérico. En este caso si consultamos la propiedad length siempre nos va a devolver el valor 0, aunque tenga elementos. Si accedemos al array con una clave que no existe vamos a obtener el valor null, no da error de ejecución.

001	<code>var nombres =new Array();</code>
002	<code>nombres["primero"]="Juan";</code>
003	<code>nombres["segundo"]="Pedro";</code>
004	<code>nombres["tercero"]="Antonio";</code>
005	<code>nombres["cuarto"]="Felix";</code>
006	<code>document.writeln(nombres['primero']+"
");</code>
007	<code>document.writeln(nombres.segundo+"
");</code>

En este caso el bucle **for ..of** no funciona.

d) Creación de objetos. Definición de métodos y propiedades.

Vamos a ver diferentes formas de crear objetos, en concreto vamos a ver cuatro formas diferentes de crear objetos.

Primera Forma

Para la creación de objetos vamos a utilizar el objeto **Object** y su método constructor. El objeto Object es un objeto genérico de datos.

```
var nombre-variable = new Object()
```

Crea un objeto genérico con el nombre indicado.

```
001 var personal=new Object ();
```

La forma de declarar las propiedades es asignando valor a las mismas a continuación de la creación del objeto, poniendo

```
nombre-objeto.nombre-propiedad=valor
```

También podemos utilizar:

```
nombre-objeto[nombre-propiedad]= valor
```

En este caso el nombre de la propiedad puede venir representada por una variable o una constante de tipo cadena.

La declaración de los métodos se realiza:

```
nombre-objeto.nombre-método= function([parámetros]) {  
    cuerpo-método  
}
```

y también podemos utilizar la siguiente forma

```
nombre-objeto[nombre-método]=function ([parámetros]) {  
    cuerpo-método  
}
```

en este caso como en el caso anterior el nombre del método puede venir expresado como una variable o una constante de tipo cadena.

Para acceder desde los métodos a las propiedades deberemos poner:

nombre-objeto.nombre-propiedad

ejemplo-4-030.js

001	<code>var coche =newObject ();</code>
002	<code>coche.marca=vmarca;</code>
003	<code>coche.modelo=vmodelo;</code>
004	<code>coche.precio=parseFloat(vprecio);</code>
005	<code>coche.potencia=parseInt(vpotencia);</code>
006	<code>coche.cilindrada=parseInt(vcilindrada);</code>
007	<code>coche.consumo=parseFloat(vconsumo);</code>
008	<code>coche.precioKm=function(precioCombustible){</code>
009	<code>var elprecio= coche.consumo *</code> <code>precioCombustible /100;</code>
010	<code>return elprecio;</code>
011	<code>}</code>
012	<code>coche.precioCil=function(){</code>
013	<code>var valor= coche.precio / coche.cilindrada;</code>
014	<code>return valor;</code>
015	<code>}</code>
016	<code>coche.incrementoPrecio=function(incremento){</code>
017	<code>var incre=(coche.precio * incremento /100);</code>
018	<code>coche.precio +=incre;</code>
019	<code>}</code>

Segunda Forma

También podemos declarar un objeto a través de un método constructor que es una función, de la siguiente forma:

```
function nombre-pseudoclase(lista-parámetros) {
    cuerpo
}
```

Luego nos declaramos un objeto de esa clase a través de:

```
var nombre-objeto= new nombre-pseudoclase(
    valores-parámetros)
```

Para definir propiedades usaremos dentro del cuerpo:

```
this.nombre-propiedad=valor
```

Para declara propiedades de solo lectura desde dentro usaremos

```
this.__defineGetter__(nombre-propiedad,  
function(parámetro) { cuerpo}
```

Para declara propiedades de solo escritura desde dentro usaremos

```
this.__defineSetter__(nombre-propiedad,  
function(parámetro) { cuerpo}
```

Para declarar métodos usaremos dentro del cuerpo:

```
this.nombre-método=function ([parámetros]) {  
cuerpo-método  
}
```

Dentro de los métodos para poder acceder a las propiedades deberemos poner:

```
this.nombre-propiedad
```

Si deseamos añadir alguna propiedad desde fuera deberemos usar:

```
nombre-pseudoclase.prototype.nombre-propiedad=valor
```

Para declara propiedades de solo lectura desde fuera usaremos

```
nombre-objeto.__defineGetter__(nombre-propiedad,  
function(parámetro) { cuerpo}
```

Para declara propiedades de solo lectura desde fuera usaremos

```
nombre-pseudoclase.prototype.__defineGetter__(  
nombre-propiedad, function(parámetro) { cuerpo}
```

Para declara propiedades de solo escritura desde fuera usaremos

```
nombre-objeto.__defineSetter__(nombre-propiedad,  
function(parámetro) { cuerpo}
```

Para declara propiedades de solo escritura desde fuera usaremos

```
nombre-pseudoclase.prototype.__defineSetter__(
nombre-propiedad, function(parámetro) { cuerpo}
```

Si deseamos añadir algún método desde fuera pondremos

```
nombre-pseudoclase.prototype.nombre-método =
function([parámetros]) {
    cuerpo
}
```

ejemplo-04-031.js

001	function tipoVehiculo(pmarca, pmodelo, pprecio, pcilindrada, ppotencia, pconsumo, pfechaCompra) {
002	var tipoCombustible="Gasolina";
003	this .marca=pmarca;
004	this .modelo=pmodelo;
005	this .precio=pprecio;
006	this .cilindrada=pcilindrada;
007	this .potencia=ppotencia;
008	this .consumo=pconsumo;
009	this .fechaCompra=pfechaCompra;
010	this .precioKm= function (precioCombustible) {
011	var elprecio= this .consumo * precioCombustible /100;
012	return elprecio;
013	}
014	this .precioCil= function () {
015	var valor= this .precio / this .cilindrada;
016	return valor;
017	}
018	this .incrementoPrecio= function (incremento) {
019	var incre=(this .precio * incremento /100);
020	this .precio +=incre;
021	}
022	this .__defineGetter__("añoCompra", function () {
023	return this .fechaCompra.getFullYear();
024	});
025	this .__defineSetter__("añoCompra", function (anyo) {
026	this .fechaCompra.setFullYear(anyo);

027	});
028	this.__defineGetter__("Combustible",function()
029	{
030	return tipoCombustible;
031	});
032	this.__defineSetter__("Combustible",function(
033	combus){
034	tipoCombustible=combus;
035	});
036	};
037	tipoVehiculo.prototype.precioMetalizado=1000;
038	tipoVehiculo.prototype.precioCompleto=function(
039	complemento){
040	returnthis.precio + complemento;
041	}
042	tipoVehiculo.prototype.__defineGetter__("nomb
043	reCompleto",function(){
044	returnthis.marca + " "+this.modelo;
	});
	tipoVehiculo.prototype.__defineSetter__("mesC
	ompra",function(vmes){
	this.fechaCompra.setMonth(vmes -1);
	});

Si queremos aplicar herencia utilizando esta segunda forma, deberemos crearnos la clase padre y luego dentro de la clase hija para heredar el comportamiento de la clase padre deberemos poner.

nombre-clase-padre.call(this, parámetros)

ejemplo-04-032.js

001	function tipoCoche(pmar, pmod){
002	console.log(pmar + " "+ pmod)
003	this.marca=pmar;
004	this.modelo=pmod;
005	}
006	function tipoVehiculo(pmarca, pmodelo,
	pprecio, pcilindrada, ppotencia, pconsumo,
	pfechaCompra){

007	<code>var tipoCombustible="Gasolina";</code>
008	<code>console.log(pmarca+" "+ pmodelo);</code>
009	<code>tipoCoche.call(this, pmarca, pmodelo);</code>
010	<code>this.precio=pprecio;</code>
011	<code>this.cilindrada=pcilindrada;</code>
012	<code>this.potencia=ppotencia;</code>
013	<code>this.consumo=pconsumo;</code>
014	<code>this.fechaCompra=pfechaCompra;</code>
015	<code>this.precioKm=function(precioCombustible) {</code>
016	<code>var elprecio=this.consumo * precioCombustible</code> <code>/100;</code>
017	<code>return elprecio;</code>
018	<code>}</code>
019	<code>this.precioCil=function() {</code>
020	<code>var valor=this.precio</code> <code>/this.cilindrada;</code>
021	<code>return valor;</code>
022	<code>}</code>
023	<code>this.incrementoPrecio=function(incremento) {</code>
024	<code>var incre=(this.precio * incremento /100);</code>
025	<code>this.precio +=incre;</code>
026	<code>}</code>
027	<code>this.__defineGetter__("añoCompra",function(){</code>
028	<code>returnhis.fechaCompra.getFullYear();</code>
029	<code>});</code>
030	<code>this.__defineSetter__("añoCompra",function(an</code> <code>yo){</code>
031	<code>this.fechaCompra.setFullYear(anyo);</code>
032	<code>});</code>
033	<code>this.__defineGetter__("Combustible",function(</code> <code>) {</code>
034	<code>return tipoCombustible;</code>
035	<code>});</code>
036	<code>this.__defineSetter__("Combustible",function(</code> <code>combus){</code>
037	<code>tipoCombustible=combus;</code>
038	<code>});</code>
039	<code>}</code>

Una clase puede tener herencia múltiple, es decir que herede el comportamiento de varias clases, para lo cual deberemos poner la

instrucción anterior tantas veces como veces herede el comportamiento de otras clases.

ejemplo-04-033.js

001	<code>function tipoCoche(pmar, pmod){</code>
002	<code> this.marca=pmar;</code>
003	<code> this.modelo=pmod;</code>
004	<code>}</code>
005	<code>function tecnicos(pcilin,ppoten,pcons){</code>
006	<code> this.cilindrada=pcilin;</code>
007	<code> this.potencia=ppoten;</code>
008	<code> this.consumo=pcons;</code>
009	<code>}</code>
010	<code>function tipoVehiculo(pmarca, pmodelo, pprecio, pcilindrada, ppotencia, pconsumo, pfechaCompra){</code>
011	<code> var tipoCombustible="Gasolina";</code>
012	<code> console.log(pmarca+" "+ pmodelo);</code>
013	<code> tipoCoche.call(this, pmarca, pmodelo);</code>
014	<code> this.precio=pprecio;</code>
015	<code> tecnicos.call(this, pcilindrada,ppotencia, pconsumo);</code>
016	<code> this.fechaCompra=pfechaCompra;</code>
017	<code> this.precioKm=function(precioCombustible){</code>
018	<code> var elprecio=this.consumo * precioCombustible /100;</code>
019	<code> return elprecio;</code>
020	<code>}</code>
021	<code> this.precioCil=function(){</code>
022	<code> var valor=this.precio /this.cilindrada;</code>
023	<code> return valor;</code>
024	<code>}</code>
025	<code> this.incrementoPrecio=function(incremento){</code>
026	<code> var incre=(this.precio * incremento /100);</code>
027	<code> this.precio +=incre;</code>
028	<code>}</code>
029	<code> this.__defineGetter__("añoCompra",function(){</code>
030	<code> returnthis.fechaCompra.getFullYear();</code>
031	<code> });</code>
032	<code> this.__defineSetter__("añoCompra",function(an</code>

	yo) {
033	this.fechaCompra.setFullYear(anho) ;
034	});
035	this.__defineGetter__("Combustible",function(
) {
036	return tipoCombustible;
037	});
038	this.__defineSetter__("Combustible",function(
	combust) {
039	tipoCombustible=combust;
040	});
041	}

Para acceder desde la clase hija a un elemento de la clase padre, deberemos poner **this.elemento-padre**.

Tercera forma

Nos declaramos una clase

```
class nombre-clase{
    [ constructor ([parámetros]) {
        instrucciones
    }]
    [ [static] nombre-método(parámetros) {
        instrucciones}]
}
```

Mediante la palabra constructor nos estamos declarando el método constructor de la clase y en el cual vamos a inicializar todas las propiedades de la clase, que van a llevar siempre el prefijo **this**. También se pueden declarar variables cuyo ámbito será el constructor y se pueden declarar así mismo, métodos.

Mediante **static** nos estamos declarando un método llamado estático, método que puede ser llamado sin ser estanciado, esto es, que se puede llamar a ese método utilizando la clase y no el objeto de la clase.

ejemplo-04-053.js

001	class coches {
002	constructor (pmarca,pmodelo,pprecio) {
003	this .marca=pmarca;
004	this .modelo=pmodelo;
005	this .precio=pprecio;
006	}
007	cuotamensual(meses) {
008	let valor=(this .precio *1.20) / meses;
009	return valor;
010	}
011	}
012	var mio= new coches("seat","arosa",12450);
013	document .writeln(mio.marca + " ");
014	document .writeln(mio.modelo + " ");
015	document .writeln(mio.precio + " ");
016	document .writeln(mio.cuotamensual(12)+ " ");

ejemplo-04-055.js

001	class coches {
002	constructor (pmarca,pmodelo,pprecio) {
003	this .marca=pmarca;
004	this .modelo=pmodelo;
005	this .precio=pprecio;
006	let dolar=0;
007	this .valor_dolar= function (pvalor) {
008	dolar=pvalor;
009	}
010	this .precio_dolar= function () {
011	return this .precio / dolar;
012	}
013	};
014	cuotamensual(meses) {
015	let valor=(this .precio *1.20) / meses;
016	return valor;
017	}

018	
019	}
020	var mio=new coches("seat","arosa",12450);
021	document.writeln(mio.marca + " ");
022	document.writeln(mio.modelo + " ");
023	document.writeln(mio.precio + " ");
024	document.writeln(mio.cuotamensual(12) + " ");
025	mio.valor_dolar(0.87);
026	document.writeln(mio.precio_dolar() + " ");

Dentro de la clase y fuera del constructor nos podemos declarar propiedades de solo lectura a través de:

```
get nombre-propiedad(){
    cuerpo
    return expresión;
}
```

También dentro de la clase y fuera del constructor nos podemos declarar propiedades de solo escritura mediante:

```
set nombre-propiedad(parámetro){
    cuerpo
}
```

001	class coches {
002	constructor(pmarca,pmodelo,pprecio) {
003	this.marca=pmarca;
004	this.modelo=pmodelo;
005	this.precio=pprecio;
006	this.dolar=0;
007	};
008	cuotamensual(meses) {
009	let valor=(this.precio *1.20)/ meses;
010	return valor;
011	}
012	set valor_dolar(pvalor) {
013	this.dolar=pvalor;
014	}
015	get precio_dolar() {

016	<code>returnthis.precio /this.dolar;</code>
017	<code>}</code>
018	<code>}</code>
019	<code>var mio=new coches("seat","arosa",12450);</code>
020	<code>document.writeln(mio.marca + "
");</code>
021	<code>document.writeln(mio.modelo + "
");</code>
022	<code>document.writeln(mio.precio + "
");</code>
023	<code>document.writeln(mio.cuotamensual(12)+ "
");</code>
024	<code>mio.valor_dolar=0.87;</code>
025	<code>document.writeln(mio.precio_dolar + "
");</code>
026	<code>mio.dolar=0.93;</code>
027	<code>document.writeln(mio.precio_dolar + "
");</code>

Dentro de la declaración de una clase también nos podemos declarar variables y métodos privados, para ello bastara con anteponer el símbolo de la almohadilla “#” delante del nombre de la variable o del método (también se puede aplicar a elementos de solo lectura o solo escritura). Las variables las podemos declarar delante del constructor. Cuando queramos hacer referencia a estos elementos privados deberemos poner **this.#nombre_variable** o **this#nombre_método()**.

001	<code>class persona {</code>
002	<code> #nom="";</code>
003	<code> constructor(pnombre,papellidos) {</code>
004	<code> this.#nom=pnombre;</code>
005	<code> this.apellidos=papellidos;</code>
006	<code> };</code>
007	<code> set nombre(pvalor) {</code>
008	<code> this.#nom=pvalor;</code>
009	<code> }</code>
010	<code> get nombre() {</code>
011	<code> returnthis.#nom;</code>
012	<code> }</code>
013	<code> #completo() {</code>
014	<code> returnthis.#nom + " " +this.apellidos;</code>
015	<code> }</code>
016	<code> total() {</code>

017	<code>return this.#completo();</code>
018	<code>}</code>
019	<code>}</code>
020	<code>if(document.addEventListener)</code>
021	<code>window.addEventListener("load", inicio)</code>
022	<code>elseif(document.attachEvent)</code>
023	<code>window.attachEvent("onload", inicio);</code>
024	<code>function inicio() {</code>
025	<code>let</code> <code>boton=document.getElementById("resultado");</code>
026	<code>if(document.addEventListener)</code>
027	<code>boton.addEventListener("click", tratar)</code>
028	<code>else if(document.attachEvent)</code>
029	<code>boton.attachEvent("onclick", tratar);</code>
030	<code>}</code>
031	<code>function tratar() {</code>
032	<code>let</code> <code>ape=document.getElementById("apellidos").value;</code>
033	<code>let</code> <code>nom=document.getElementById("nombre").value;</code>
034	<code>let nuevo =new persona(nom,ape);</code>
035	<code>console.log(nuevo.nombre);</code>
036	<code>console.log(nuevo.apellidos);</code>
037	<code>document.getElementById("completo").value=nuevo.total();</code>
038	<code>}</code>

Para aplicar herencia utilizaremos

```
class nombre-clase extends clase-padre{
    [ constructor ([parámetros]) {
        super([parámetros]);
        instrucciones
    }]
    [ [static] nombre-método(parámetros) {
        instrucciones}]
}
```

Para llamar al método constructor de la clase padre utilizamos dentro del constructor de la clase hija **super** con sus correspondientes parámetros.

Si desde la clase hija queremos llamar a algún método de la clase padre deberemos poner **super.nombre-método**

ejemplo-04-056.js

001	class coches {
002	constructor (pmarca,pmodelo,pprecio) {
003	this .marca=pmarca;
004	this .modelo=pmodelo;
005	this .precio=pprecio;
006	this .dolar=0;
007	};
008	cuotamensual(meses) {
009	let valor=(this .precio *1.20) / meses;
010	return valor;
011	}
012	set valor dolar(pvalor) {
013	this .dolar=pvalor;
014	}
015	get precio dolar() {
016	return this .precio / this .dolar;
017	}
018	completo() {
019	return this .marca +" "+ this .modelo;
020	}
021	}
022	}
023	class vehiculos extends coches {
024	constructor (pmarca,pmodelo,pacabado,pprecio,p cilin,ppoten) {
025	super (pmarca,pmodelo,pprecio) ;
026	this .acabado=pacabado;
027	this .cilindrada=pcilin;
028	this .potencia=ppoten;
029	}
030	completo() {
031	return super .completo()+" "+ this .acabado;
032	}

033	}
034	var mio= new coches("seat","arosa",12450);
035	document .writeln(mio.marca + " ");
036	document .writeln(mio.modelo + " ");
037	document .writeln(mio.precio + " ");
038	document .writeln(mio.cuotamensual(12)+" ");
039	document .writeln(mio.completo()+" ");
040	mio.valor_dolar=0.87;
041	document .writeln(mio.precio_dolar + " ");
042	mio.dolar=0.93;
043	document .writeln(mio.precio_dolar + " ");
044	var nuestro = new vehiculos("opel","vectra","alto",19850,2000,150);
045	document .writeln(nuestro.marca + " ");
046	document .writeln(nuestro.modelo + " ");
047	document .writeln(nuestro.acabado + " ");
048	document .writeln(nuestro.potencia + " ");
049	document .writeln(nuestro.cilindrada + " ");
050	document .writeln(nuestro.precio + " ");
051	document .writeln(nuestro.cuotamensual(24)+" ");
052	document .writeln(nuestro.completo()+" ");
053	document .writeln(mio.cuotamensual(36)+" ");

Cuarta Forma

Declararnos un objeto de forma implícita

```
var nombre-objeto = {  
    cuerpo  
}
```

Para declarar propiedades pondremos

```
nombre-propiedad :valor,
```

Para declarar propiedades de solo lectura pondremos

```
get nombre-propiedad() { cuerpo},
```

En el cuerpo va a actuar como una función, con lo cual debe devolver un valor.

Para declarar propiedades de solo escritura pondremos

```
set nombre-propiedad(parámetro) { cuerpo},
```

También podemos declararnos propiedades a través del set y de get y que no dependan de ninguna otra propiedad, en este caso se necesita una variable auxiliar que se debe declarar dentro de la función donde se crea el objeto y que se puede utilizar en el set y en el get.

Para declarar métodos según esta cuarta forma usaremos

```
nombre-método : function ([parámetros]) {  
  cuerpo  
}
```

Dentro de los métodos, del set y del get para poder acceder a las propiedades deberemos poner:

```
this.nombre-propiedad
```

001	var coche={
002	marca:vmarca,
003	modelo:vmodelo,
004	precio: parseFloat (vprecio),
005	potencia: parseInt (vpotencia),
006	cilindrada: parseInt (vcilindrada),
007	consumo: parseFloat (vconsumo),
008	fechaCompra:vfecha,
009	precioKm: function (precioCombustible) {
010	var elprecio= this .consumo*precioCombustible / 100;
011	return elprecio;
012	},
013	precioCil: function () {

014	<code>var valor=this.precio/this.cilindrada;</code>
015	<code>return valor;</code>
016	<code>},</code>
017	<code>incrementoPrecio:function(incremento) {</code>
018	<code>var incre=(this.precio*incremento/100);</code>
019	<code>this.precio+=incre;</code>
020	<code>},</code>
021	<code>getañoCompra() {</code>
022	<code>return this.fechaCompra.getFullYear();</code>
023	<code>},</code>
024	<code>setañoCompra(anyo) {</code>
025	<code>this.fechaCompra.setFullYear(anyo);</code>
026	<code>}</code>
027	<code>}</code>

Con los objetos podemos utilizar las siguientes instrucciones:

```
for (variable in objeto) {  
    cuerpo  
}
```

Se va a ejecutar una vez por cada elemento del objeto ya bien sea propiedad o método y en donde variable va a tomar el nombre de los elementos del objeto.

Ejecutas el cuerpo de las instrucciones por cada uno de los valores del objeto, solo para objeto iterables, los objetos que nos creamos no lo son.

```
for (variable of objeto) {  
    cuerpo  
}
```

Se puede hacer referencia a las propiedades y métodos del objeto sin hacer referencia al mismo ya que se indica al principio.

```
with (objeto) {  
    instrucciones}
```

001	class coches {
002	constructor (pmarca,pmodelo,pprecio) {
003	this .marca=pmarca;
004	this .modelo=pmodelo;
005	this .precio=pprecio;
006	this .dolar=0;
007	};
008	cuotamensual(meses) {
009	let valor=(this .precio *1.20)/ meses;
010	return valor;
011	}
012	set valor_dolar(pvalor) {
013	this .dolar=pvalor;
014	}
015	get precio_dolar() {
016	return this .precio / this .dolar;
017	}
018	}
019	var mio= new coches("seat","arosa",12450);
020	with (mio) {
021	marca="Volkswagen";
022	modelo="Golf";
023	precio=35000;
024	dolar=0.98;
025	}
026	for (var dato in mio) {
027	document .writeln(dato+" "+eval("mio."+dato)+" ");
028	}

Para saber si un objeto es de una clase tenemos:

nombre-objeto.**constructor.name** === "nombre-clase"

nombre-objeto.**constructor** === nombre-clase

Object.getPrototypeOf(nombre-objeto) === nombre-clase.**prototype**

También podemos utilizar **instanceof**, pero deberemos tener en cuenta que si la clase es una clase hija, nos va a decir que es de la clase propia y de la clase padre.

nombre-objeto **instanceof** nombre-clase

Quinta Forma

En esta forma va a ser a través del objeto **Object**, sus métodos y propiedades.

El Objeto Object.

Características de Object y de los objetos.

Propiedad **constructor**

Nombre-objeto.constructor → tiene una referencia al constructor del objeto.

001	<code>var misDatos =newObject();</code>
002	<code>if(misDatos.constructor==Object){</code>
003	<code> console.log("El constructor de misDatos es Object");</code>
004	<code>}</code>

001	<code>function coches () {</code>
002	<code> this.marca ="";</code>
003	<code> this.modelo="";</code>
004	<code> this.precio =0;</code>
005	<code> this.precioComplementos=0;</code>
006	<code> this.nombreCompleto=function(){return(this.marca +" "+this.modelo);}</code>
007	<code> this.incrementoPrecio=function(porcentaje){this.precio *=(1+(porcentaje/100));}</code>
008	<code> this.incrementoComplementos=function(){this.precioComplementos *=1.05;}</code>
009	<code>}</code>
010	<code>var miCoche =new coches;</code>

011	<code>if(miCoche.constructor== coches){</code>
012	<code>alert("El constructor de miCoche es coches");</code>
013	<code>}</code>

001	<code>var coches = {</code>
002	<code> marca : "",</code>
003	<code> modelo : "",</code>
004	<code> precio : 0,</code>
005	<code> precioComplementos : 0,</code>
006	<code> nombreCompleto</code> <code> :function() {return(this.marca + "</code>
	<code> "+this.modelo) },</code>
007	<code> incrementoPrecio</code> <code> :function(porcentaje) {this.precio</code>
	<code> *= (1+(porcentaje/100)); },</code>
008	<code> incrementoComplementos</code> <code> :function() {this.precioComplementos *=1.05; }</code>
009	<code>}</code>
010	<code>if(coches.constructor==Object) {</code>
011	<code> alert("El constructor de coches es Object");</code>
012	<code>}</code>

create crear un objeto a partir de un prototipo con unas propiedades. El prototipo puede ser **null** bien **Object.prototype** o bien otro objeto o bien una clase o bien **clase.prototype**.

Object.create(nombre-objeto, {definición propiedades})

001	<code>var misDatos =newObject ();</code>
002	<code>misDatos.nombre="pedro";</code>
003	<code>var miObjeto=Object.create (misDatos, {</code>
004	<code> apellidos: {</code>
005	<code> value: "Garcia",</code>
006	<code> writable: true,</code>
007	<code> enumerable: true,</code>
008	<code> configurable: true</code>
009	<code> },</code>
010	<code>});</code>
011	<code>if(miObjeto.constructor==Object) {</code>
012	<code> alert("El constructor de miObjeto es Object");</code>
013	<code>}</code>

Para definir una propiedad vamos a poner:

```
nombre-propiedad : {  
    value:valor,  
    writable:true|false ,  
    enumerable:true|false,  
    configurable:true|false  
}
```

- En este caso ponemos **value** para asignar un valor. El resto de opciones se pueden poner o bien omitir y tienen el siguiente significado: Con **writable** nos indica si en la propiedad se puede escribir (true) o bien no se puede (false).
- Con **enumerable** nos indica si la propiedad la podemos utilizar en un bucle for in, si se puede (true) y si no se puede (false).
- Con **configurable** nos indica si la propiedad se puede configurar mediante otros métodos de la clase Object, con true se puede y con false no se puede.

También se pueden declarar propiedades utilizando el **set** si es de solo escritura, utilizando el **get** si es de solo lectura y también se puede declarar utilizando el **set** y **get**. Si la propiedad no depende de ninguna otra propiedad se puede poner una variable auxiliar que se declara en la función que crea el objeto y que se puede utilizar. La forma de declarar las propiedades de esta forma es:

```
nombre-propiedad : {
    get : function([parametros]) { cuerpo-función },
    set : function(parámetro[, parametros]) { cuerpo-función },
    enumerable:true|false ,
    configurable:true|false
}
```

Se puede poner todo o bien solo el **set** y/o el **get** el resto de las opciones se pueden poner o bien omitir.

001	<code>var coche =Object.create(null,{</code>
002	<code>marca:{value:"", writable:true,</code>
	<code>configurable:true, enumerable:true},</code>
003	<code>modelo:{value:"", writable:true,</code>
	<code>configurable:true, enumerable:true},</code>
004	<code>precio:{value:1.0, writable:true,</code>
	<code>configurable:true, enumerable:true},</code>
005	<code>potencia:{value:1, writable:true,</code>
	<code>configurable:true, enumerable:true},</code>
006	<code>cilindrada:{value:1, writable:true,</code>
	<code>configurable:true, enumerable:true},</code>
007	<code>consumo:{value:1.0, writable:true,</code>
	<code>configurable:true, enumerable:true},</code>
008	<code>fechaCompra:{value:newDate(), writable:true,</code>
	<code>configurable:true, enumerable:true},</code>
009	<code>añoCompra:{</code>
010	<code>get:function() {</code>
011	<code>returnthis.fechaCompra.getFullYear()</code>
012	<code>},</code>
013	<code>set:function(anyo) {</code>
014	<code>this.fechaCompra.setFullYear(anyo)</code>
015	<code>}</code>
016	<code>},</code>
017	<code>precioCilindrada:{</code>
018	<code>get:function() {</code>
019	<code>returnthis.precio /this.cilindrada;</code>
020	<code>}</code>
021	<code>},</code>
022	<code>});</code>

001	function tipoCoche(pmarca,ppre) {
002	this .marca=pmarca;
003	this .precio=ppre;
004	}
005	var nuevo = new tipoCoche(vmarca,vpre);
006	var coche = Object .create(tipoCoche. prototype , {
007	modelo:{value:"", writable: true ,
008	potencia:{value:1, writable: true ,
009	cilindrada:{value:1, writable: true ,
010	consumo:{value:1.0, writable: true ,
011	fechaCompra:{value: new Date(),
012	añoCompra:{
013	get: function () {
014	return this .fechaCompra.getFullYear()
015	},
016	set: function (anyo) {
017	this .fechaCompra.setFullYear(anyo)
018	}
019	},
020	precioCilindrada:{
021	get: function () {
022	return this .precio / this .cilindrada;
023	}
024	},
025	});

001	var nuevo = new Object();
002	nuevo.marca=vmarca;
003	nuevo.precio=vpre;
004	var coche = Object .create(nuevo ,{
005	modelo:{value:"", writable: true ,
006	potencia:{value:1, writable: true ,

007	cilindrada:{value:1, writable:true, configurable:true, enumerable:true},
008	consumo:{value:1.0, writable:true, configurable:true, enumerable:true},
009	fechaCompra:{value:new Date(), writable:true, configurable:true, enumerable:true},
010	añoCompra:{
011	get:function(){
012	return this.fechaCompra.getFullYear();
013	},
014	set:function(anyo){
015	this.fechaCompra.setFullYear(anyo)
016	}
017	},
018	precioCilindrada:{
019	get:function(){
020	return this.precio /this.cilindrada;
021	}
022	},
023	});

Todas las declaraciones de las propiedades van a estar separadas por comas.

defineProperty → añade una propiedad a un objeto

Object.defineProperty(*nombre-objeto, nombre-propiedad, descriptor-propiedad*)

001	Object.defineProperty (miObjeto,"edad",{value:33, writable:true});
001	Object.defineProperty(coche,"color",{value:color, writable:true, configurable:true, enumerable:true});

defineProperties → añade propiedades a un objeto

Object.defineProperty(objeto, descriptores-propiedades)

001	<code>Object.defineProperty(miObjeto,{</code>
002	<code> localidad:{value:"Madrid", writable:true},</code>
003	<code> estadoCivil:{value:"Soltero", writable:true}</code>
	<code>});</code>

001	<code>Object.defineProperty(coche,{</code>
	<code> matricula:{value:vmatricula, writable:true,</code>
	<code> configurable:true, enumerable:true},</code>
002	<code> bastidor:{value:vbastidor, writable:true,</code>
	<code> configurable:true, enumerable:true}));</code>

Para ver si dos objetos son iguales

Object.is(objeto-1,objeto-2)

Devuelve un valor lógico que nos indica si los dos objetos son iguales.

Para copiar una serie de objetos a otro

Object.assign(destino, lista-objetos)

Copia la lista de objetos sobre el destino y devuelve una copia del mismo.

freeze → impide añadir propiedades, modificar propiedades o atributos.

Object.freeze(objeto)

001	<code>Object.freeze(miObjeto);</code>
-----	---------------------------------------

isExtensible → indica si se pueden añadir nuevas propiedades al objeto

Object.isExtensible(objeto)

```
001 | Object.isExtensible(miObjeto) ;
```

isFrozen → indica si NO se pueden modificar propiedad, atributos ni añadir nuevas propiedades.

Object.isFrozen(objeto)

```
001 | Object.isFrozen(miObjeto) ;
```

isSealed → indica si no se pueden modificar atributos de propiedades no se pueden añadir nuevas propiedades.

Object.isSealed(objeto)

```
001 | Object.isSealed(miObjeto) ;
```

seal → impide modificar atributos de propiedades y añadir nuevas propiedades.

Object.seal(objeto)

```
001 | Object.seal(miObjeto) ;
```

getOwnPropertyNames → devuelve un array con el nombre de las propiedades y métodos de un objeto.

array=Object.getOwnPropertyNames(objeto)

```
001 | var nombres  
    =Object.getOwnPropertyNames(miObjeto) ;
```

getOwnPropertyDescriptor → devuelve el descriptor de una propiedad de un objeto.

Object.getOwnPropertyDescriptor(objeto, nombre-propiedad)

nombre-objeto.toString() → devuelve el objeto como una cadena.

nombre-objeto.propertyIsEnumerable(nombre-propiedad)→indica si la propiedad es enumerable (indica si puede estar en un bucle for each).

nombre-objeto-1.isPrototypeOf(objeto-2)→indica si el objeto 2 tiene objeto 1 en su cadena de prototipos.

nombre-objeto.hasOwnProperty(nombre-propiedad)→ indica si el objeto tiene la propiedad indicada.

Object.preventExtensions(objeto)→ impide que se puedan añadir más propiedades

Object.keys(objeto)→ devuelve un array con los nombres de los métodos y propiedades.

objeto.watch(propiedad, función)→ función que se ejecuta cuando se asigna valor a la propiedad.

objeto.unwatch(propiedad)→deja de ejecutarse la función.

objeto.__lookupGetter__(propiedad) →referencia a la función de un getter para la propiedad.

objeto.__LookupSetter__(propiedad)→referencia a la función de un setter para la propiedad.

El método **create** también se puede utilizar para cambiar el comportamiento de un objeto existente si ponemos

objeto.prototype = objeto.create(clase-padre.prototype, {declaración-propiedades});