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.

isFinity(*valor***)** devuelve un valor lógico que nos indica si el valor es finito, devuelve false cuando el valor en in finito, -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 escapdos aparecen con su valor..

b) Llamadas a funciones. Definición de 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        return;
008 }
009 mia("hola","prueba",13,45);
```

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

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

```
001  function mia() {
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       return;
008  }
009  mia("hola","prueba",13,45);
```

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)+"<br />" );
```

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, al poner ese parámetro le vamos a poner un prefijo de tres puntos seguidos. La forma de declararlo será

function *nombre-función*(*parámetro1*, *parámetro2*, …*parámetro3*) **{**
cuerpo
}

El parámetro3 recibe todos los parámetros que se le pasan a la función a partir del tercero 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) + "<br />");
```

Funciones simples

function(*parámetros*) *instrucción-del-return*

ejemplo-04-03.js

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

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

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(uno) {
009     return uno * 2;
010 }
011 var doble1 = (uno) => {
012     return uno * 2;
013 };
014 var doble2 = uno => {
015     return uno * 2;
016 };
017 var doble3 = (uno) => uno * 2;
018 var doble4 = uno => uno * 2;
019
020 function operaciones(one, two, ...other) {
021     var sumar = one + two;
022     for (var i=0; i < other.length;i++)
023         sumar += other[i];
024     return sumar;
025 }
026 var operaciones1 =(one, two, ...other) => {
027     var sumar = one + two;
028     for (var i=0; i < other.length;i++)
029         sumar += other[i];
030     return sumar;
031 }
032 function opera(first, second=0, thersty=2) {
033     var res1=first + second + thersty;
034     var res2=first * thersty;
035     var res3=first / thersty;
036     return [res1,res2,res3];
037 }
038 var opera1=(first, second=0, thersty=2) => {
039     var res1=first + second + thersty;
040     var res2=first * thersty;
041     var res3=first / thersty;
042     return [res1,res2,res3];
043 }
044 document.writeln( suma( 2, 4 )+"<br />" );
045 document.writeln( suma1( 2, 4 )+"<br />" );
046 document.writeln( suma2( 2, 4 )+"<br />" );
047 document.writeln( doble( 5 ) +"<br />");
048 document.writeln( doble1( 5 ) +"<br />");
049 document.writeln( doble2( 5 ) +"<br />");
050 document.writeln( doble3( 5 ) +"<br />");
```

```
051 document.writeln( doble4( 5 ) +"<br />");
052 document.writeln( operaciones(2,4,6,8) + "<br />");
053 document.writeln( operaciones1(2,4,6,8) + "<br />");
054 [ope1,ope2,ope3] = opera(45);
055 document.writeln( ope1+"<br />" );
056 document.writeln( ope2+"<br />" );
057 document.writeln( ope3+"<br />" );
058 [ope1,ope2,ope3] = opera1(45);
059 document.writeln( ope1+"<br />" );
060 document.writeln( ope2+"<br />" );
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 llama 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         suma+=arguments[i];
009         yield* duplicado(suma);
010     }
011     return suma;
012 }
013 var sumando=sumatorio(12,23,13,45,25,56,37,53,74,83,16,94,84);
014 var dato=sumando.next();
015 while (!dato.done) {
016     document.writeln(dato.value);
017     dato=sumando.next();
018 }
```

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 array 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 var tabla = new Array();
```

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=new Array("Juan","Pedro",13,true);
```

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

Nos declara un array con tantos elementos cono se indican.

```
001 var arreglo = new Array(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 elemento en orden inverso.
- **sort()** :devuelve un array con los elementos ordenados.
- **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 rellenado y si indicamos final se indica en qué posición se para el rellenado, en esa posición no se produce el rellenado.
- **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 incicial.
- **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.

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

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

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

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 vacios 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 array.

Métodos

- **get(***clave***)**: devuelve el elemento que tiene esa clave.
- **set(***clave* **,** *valor* **)**: incluye el valor en el array 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 array que tiene la clave indicada.

- **clear()**: borra todos los elementos del array.
- **entries()**:devuelve un array bidimensional que va a tener en cada fila la referencia a los elementos del objeto Map inicial y en las columnas va a tener la clave del elemento y el valor del elemento.
- **keys()**:devuelve un array con todas las claves del objeto Map.
- **values()**: devuelve unarray con todos los valores del objeto Map.
- **toString()**: devuelve el elemento como una cadena.
- **valueOf()**: devuelve un valor.
- **forEach(función (*valor*, *clave*, *objeto*) { *cuerpo-función*}):** realiza la acción indicada para cada elemento del array.

ejemplo-04-020.js

```
001 var nuevo= new Map();
002 function anadir() {
003     var valor=prompt("Introduce un valor");
004     var clave=prompt("Introduce su clave");
005     if (nuevo.has(clave))
006         alert("Ya existe esa clave en el array");
007     else
008         nuevo.set(clave, valor);
009 }
010 function consulta() {
011     var clave=prompt("Introduce su clave");
012     if (nuevo.has(clave))
013         alert("El valor correspondiente a la clave " + clave + " es  "
+nuevo.get(clave));
014     else
015         alert("NO existe esa clave en el array");
016 }
017 function borrar() {
018     var clave=prompt("Introduce su clave");
019     if (nuevo.has(clave)) {
020         nuevo.delete(clave);
021         alert("Valor borrado del array");
022     } else
023         alert("NO existe esa clave en el array");
024 }
025 function numero() {
026     alert("El número de elementos del array es  " + nuevo.size.toString());
027 }
028 function todos() {
029     nuevo.clear()
030     alert("Todos los elementos han sido borrados " );
031 }
032 function valores() {
033     var todosValores="";
034     var todasClaves="";
035     var conjunto="";
036     nuevo.forEach(function (valor, clave , mismo) {
037         todosValores+=valor + " \n";
038         todasClaves+=clave + "\n";
039         conjunto+=" clave: " + clave  +"    valor; " + valor +  "\n";
040     });
041     alert("Valores \n" + todosValores);
042     alert("Claves \n " + todasClaves);
043     alert("todos \n " + conjunto );
044 }
045 function valor() {
046     alert("valueOf() \n" + nuevo.valueOf());
047 }
048 function cadena() {
049     alert("toString() \n " + nuevo.toString());
050 }
```

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.

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

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

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-pseudo*clase* **.prototype.***nombre-métod***=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         return tipoCombustible;
030     });
031     this.__defineSetter__("Combustible", function (combus) {
032         tipoCombustible=combus;
033     });
034 }
035 tipoVehiculo.prototype.precioMetalizado=1000;
036 tipoVehiculo.prototype.precioCompleto=function (complemento) {
037     return this.precio + complemento;
038 }
039 tipoVehiculo.prototype.__defineGetter__("nombreCompleto", function () {
040     return this.marca + "  " + this.modelo;
041 });
042 tipoVehiculo.prototype.__defineSetter__("mesCompra", function (vmes) {
043     this.fechaCompra.setMonth(vmes - 1);
044 });
```

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

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 clase.

ejemplo-04-033.js

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

```
026             var incre= (this.precio * incremento / 100) ;
027             this.precio +=incre;
028         }
029         this.__defineGetter__("añoCompra", function() {
030             return this.fechaCompra.getFullYear();
031         });
032         this.__defineSetter__("añoCompra", function (anyo) {
033             this.fechaCompra.setFullYear(anyo);
034         });
035         this.__defineGetter__("Combustible", function() {
036             return tipoCombustible;
037         });
038         this.__defineSetter__("Combustible", function (combus) {
039             tipoCombustible=combus;
040         });
041     }
```

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 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 + "<br />");
014 document.writeln(mio.modelo + "<br />");
015 document.writeln(mio.precio + "<br />");
016 document.writeln(mio.cuotamensual(12) + "<br />");
```

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 + "<br />");
022 document.writeln(mio.modelo + "<br />");
023 document.writeln(mio.precio + "<br />");
024 document.writeln(mio.cuotamensual(12) + "<br />");
025 mio.valor_dolar(0.87);
026 document.writeln(mio.precio_dolar() + "<br />");
```

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

```
026 mio.dolar=0.93;
027 document.writeln(mio.precio_dolar + "<br />");
```

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,pcilin,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 + "<br />");
036 document.writeln(mio.modelo + "<br />");
037 document.writeln(mio.precio + "<br />");
038 document.writeln(mio.cuotamensual(12) + "<br />");
039 document.writeln(mio.completo() + "<br />");
040 mio.valor_dolar=0.87;
041 document.writeln(mio.precio_dolar + "<br />");
042 mio.dolar=0.93;
```

```
043 document.writeln(mio.precio_dolar + "<br />");
044 var nuestro = new vehiculos("opel","vectra","alto",19850,2000,150);
045 document.writeln(nuestro.marca + "<br />");
046 document.writeln(nuestro.modelo + "<br />");
047 document.writeln(nuestro.acabado + "<br />");
048 document.writeln(nuestro.potencia + "<br />");
049 document.writeln(nuestro.cilindrada + "<br />");
050 document.writeln(nuestro.precio + "<br />");
051 document.writeln(nuestro.cuotamensual(24) + "<br />");
052 document.writeln(nuestro.completo() + "<br />");
053 document.writeln(mio.cuotamensual(36) + "<br />");
```

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 tercera 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             var valor= this.precio / this.cilindrada;
015             return valor;
016         },
017         incrementoPrecio:function(incremento) {
018             var incre= (this.precio * incremento / 100) ;
019             this.precio +=incre;
020         } ,
021         get añoCompra () {
022             return this.fechaCompra.getFullYear();
023         } ,
024         set añoCompra (anyo) {
025             this.fechaCompra.setFullYear(anyo);
026         }
027     }
```

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)+"<br />");
028 }
```

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 var misDatos = new Object();
002 if (misDatos.constructor==Object) {
003     console.log("El constructor de misDatos es Object");
004 }
```

```
001 function coches () {
002     this.marca = "";
003     this.modelo= "";
004     this.precio = 0;
005     this.precioComplementos=0;
006     this.nombreCompleto= function () { return (this.marca + " " + this.modelo);}
007     this.incrementoPrecio=function (porcentaje) { this.precio *= (1 +
    (porcentaje/100));}
008     this.incrementoComplementos=function() {this.precioComplementos *= 1.05;}
009 }
010 var miCoche = new coches;
011 if (miCoche.constructor == coches) {
012     alert("El constructor de miCoche es coches");
013 }
```

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

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

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

```
001 var misDatos = new Object();
002 misDatos.nombre="pedro";
```

```
003  var miObjeto= Object.create(misDatos,{
004          apellidos:
005              {    value:"Garcia",
006                   writable:true,
007                   enumerable:true,
008                   configurable:true
009              } ,
010          });
011  if (miObjeto.constructor == Object) {
012       alert("El constructor de miObjeto es Object");
013  }
```

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  var coche = Object.create(null , {
002  marca:{value:"", writable:true, configurable:true, enumerable:true},
003  modelo:{value:"", writable:true, configurable:true, enumerable:true},
004  precio:{value:1.0, writable:true, configurable:true, enumerable:true},
005  potencia:{value:1, writable:true, configurable:true, enumerable:true},
006  cilindrada:{value:1, writable:true, configurable:true, enumerable:true},
007  consumo:{value:1.0, writable:true, configurable:true, enumerable:true},
008  fechaCompra:{value:new Date(), writable:true, configurable:true, enumerable:true},
009  añoCompra: {
010      get: function() {
```

```
011             return this.fechaCompra.getFullYear()
012         } ,
013         set: function(anyo) {
014             this.fechaCompra.setFullYear(anyo)
015         }
016     }     ,
017     precioCilindrada: {
018         get:function () {
019             return this.precio / this.cilindrada;
020         }
021     },
022     });
```

```
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, configurable:true, enumerable:true},
008     potencia:{value:1, writable:true, configurable:true, enumerable:true},
009     cilindrada:{value:1, writable:true, configurable:true, enumerable:true},
010     consumo:{value:1.0, writable:true, configurable:true, enumerable:true},
011     fechaCompra:{value:new Date(), writable:true, configurable:true,
    enumerable:true},
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, configurable:true, enumerable:true},
006     potencia:{value:1, writable:true, configurable:true, enumerable: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:vcolor, writable:true,
     configurable:true, enumerable:true});
```

defineProperties → añade propiedades a un objeto

Objet.defineProperties(*objeto*, *descriptores-propiedades***)**

```
001  Object.defineProperties(miObjeto, { localidad: {value:"Madrid", writable:true},
002                                       estadoCivil: {value:"Soltero", writable:true}
003                                     });
```

```
001  Object.defineProperties(coche,{ matricula: {value:vmatricula, writable:true,
     configurable:true, enumerable:true},
002                                   bastidor:{value:vbastidor, writable:true,
     configurable:true, enumerable:true}});
```

Para ver si dos objetos son iguales

Object.is(*objeto-1,onbjeto-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  Object.freeze(miObjeto);
```

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-objecto***.toString()** -> devuelve el objeto como una cadena.

*nombre-objecto***.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***.unatch(***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});