|  |
| --- |
| **Ciclo superior:** |
| **Diseño de aplicaciones web** |
|  |
| **MÓDULO:**  **Desarrollo Web en Entorno Cliente** |
|  |
|  |
|  |
|  |



Material realizado por Rafael Veiga para el MEC

Adaptación y ampliación realizada por Beatriz Buyo

Índices

[Índices i](#_Toc528152984)

[Índice de Contenidos i](#_Toc528152985)

[Índice de Tablas ii](#_Toc528152986)

[Índice de Imágenes ii](#_Toc528152987)

[Índice de Saber Más ii](#_Toc528152988)

[Índice de Recomendación ii](#_Toc528152989)

[Índice de Ejemplos iii](#_Toc528152990)

[Unidad didáctica 3 1](#_Toc528152991)

[Mapa conceptual 26](#_Toc528152992)

[GLOSARIO 27](#_Toc528152993)

Índice de Contenidos

[1. Estructuras de datos. 2](#_Toc528152994)

[1.1 Objeto Array. 3](#_Toc528152995)

[1.1.1 Creación de un array e introducción de datos. 4](#_Toc528152996)

[1.1.2 Recorrido de un array. 6](#_Toc528152997)

[1.1.3 Borrado de elementos en un array. 7](#_Toc528152998)

[1.1.4 Propiedades y métodos. 9](#_Toc528152999)

[1.2 Arrays paralelos. 11](#_Toc528153000)

[1.3 Arrays multidimensionales. 12](#_Toc528153001)

[2. Funciones. 13](#_Toc528153002)

[2.1 Creación y uso de funciones 14](#_Toc528153003)

[2.2 Parámetros 15](#_Toc528153004)

[2.3 Ámbito de las variables. 16](#_Toc528153005)

[2.4 Funciones anidadas. 17](#_Toc528153006)

[2.5 Funciones anónimas. 18](#_Toc528153007)

[2.6 Funciones predefinidas del lenguaje. 19](#_Toc528153008)

[2.7 Buenas prácticas en el diseño y codificación de funciones. 20](#_Toc528153009)

[3. Creación de objetos a medida. 21](#_Toc528153010)

[3.1 Definición de propiedades. 22](#_Toc528153011)

[3.2 Definición de métodos. 23](#_Toc528153012)

[3.3 Definición de objetos literales. 25](#_Toc528153013)

Índice de Tablas

[Tabla 1 Propiedades del objeto Array 9](#_Toc528149326)

[Tabla 2 Métodos del objeto Array 10](#_Toc528149327)

[Tabla 3 Propiedades globales de JavaScript 19](#_Toc528149328)

[Tabla 4 Métodos globales de JavaScript 19](#_Toc528149329)

Índice de Imágenes

[Ilustración 1 Representación de la sintaxis de la definición literal de un objeto 25](#_Toc528149320)

Índice de Saber Más

[PARA SABER MÁS 1 Estructuras de datos 2](#_Toc528149314)

[PARA SABER MÁS 2 Orientación a Objetos en JavaScript 21](#_Toc528149315)

Índice de Recomendación

[Recomendación 1 Propiedades y funciones predefinidas 19](#_Toc528149310)

Índice de Ejemplos

[Ejemplo 1 Creación de array sin tamaño determinado 4](#_Toc528149338)

[Ejemplo 2 Creación de array con tamaño definido 4](#_Toc528149339)

[Ejemplo 3 Asignación de datos a un array 4](#_Toc528149340)

[Ejemplo 4 Acceso a un elemento del array 4](#_Toc528149341)

[Ejemplo 5 Crear un array denso 5](#_Toc528149342)

[Ejemplo 6 Creación de array de forma literal 5](#_Toc528149343)

[Ejemplo 7 Sintaxis de creación de un array mixto 5](#_Toc528149344)

[Ejemplo 8 Creación de array de tipo mixto 5](#_Toc528149345)

[Ejemplo 9 Recorrido de un array con for 6](#_Toc528149346)

[Ejemplo 10 Recorrido de un array con while 6](#_Toc528149347)

[Ejemplo 11 Sintaxis sentencia for ... in 6](#_Toc528149348)

[Ejemplo 12 Recorrido del array con for ... in 6](#_Toc528149349)

[Ejemplo 13 Borrado con delete 7](#_Toc528149350)

[Ejemplo 14 Métodos del objeto Array 10](#_Toc528149351)

[Ejemplo 15 Arrays paralelos 11](#_Toc528149352)

[Ejemplo 16 Creación de array bidimensional 12](#_Toc528149353)

[Ejemplo 17 Creación de Array bidimensional. Definición breve. 12](#_Toc528149354)

[Ejemplo 18 Recorrido array bidimensional 12](#_Toc528149355)

[Ejemplo 19 Sintaxis de definición de una función 14](#_Toc528149356)

[Ejemplo 20 Sintaxis de la llamada a una función 14](#_Toc528149357)

[Ejemplo 21 Programa que devuelve el mayor de dos números 15](#_Toc528149358)

[Ejemplo 22 Sintaxis de las Funciones anidadas 17](#_Toc528149359)

[Ejemplo 23 Ejemplo de funciones anidadas 17](#_Toc528149360)

[Ejemplo 24 Uso de funciones anónimas 18](#_Toc528149361)

[Ejemplo 25 Uso de la función global eval() 19](#_Toc528149362)

[Ejemplo 26 Nombres correctos e incorrectos de funciones 20](#_Toc528149363)

[Ejemplo 27 Nombres correctos e incorrectos de funciones 20](#_Toc528149364)

[Ejemplo 28 Constructor de objeto 21](#_Toc528149365)

[Ejemplo 29 Propiedades de un objeto 22](#_Toc528149366)

[Ejemplo 30 Constructor de coches 22](#_Toc528149367)

[Ejemplo 31 Acceso y Modificación de las propiedades de un objeto 22](#_Toc528149368)

[Ejemplo 32 Definición y uso de métodos de objetos 23](#_Toc528149369)

[Ejemplo 33 Encapsulamiento 24](#_Toc528149370)

[Ejemplo 34 Definición de objetos literales 25](#_Toc528149371)

[Ejemplo 35 Acceso a la propiedad del objeto 25](#_Toc528149372)

[Ejemplo 36 Definición y recorrido de un array de objetos 25](#_Toc528149373)

Unidad didáctica 3

Estructuras definidas por el usuario en JavaScript.

# Estructuras de datos.

En los lenguajes de programación existen estructuras de datos especiales que sirven para guardar información más compleja que lo que permite una variable simple. En la segunda unidad vimos como crear variables y cómo almacenar valores dentro de esas variables. Hay otros tipos de estructuras de datos (listas, pilas, colas, árboles, conjuntos,…) que se pueden utilizar para almacenar cualquier tipo de dato pero, sin duda, una de las más utilizadas en todos los lenguajes de programación es el **array**. El array es una variable o zona de almacenamiento continuo donde podemos almacenar más de un valor.

Los arrays también se suelen denominar **matrices** o **vectores**. Desde el punto de vista lógico, una matriz se puede ver como una colección de elementos ordenados en filas (o filas y columnas si tuviera dos o más dimensiones).

Se puede considerar que todos los arrays son de una dimensión: la dimensión principal, pero los elementos de dicha fila pueden a su vez contener otros arrays o matrices, lo que nos permitiría hablar de **arrays multidimensionales** (los más fáciles de imaginar son los de una, dos y tres dimensiones).

Los arrays son una estructura de datos, adecuada para situaciones en las que el acceso a los datos se realiza de forma aleatoria e impredecible. Por el contrario, si los elementos pueden estar ordenados y el acceso a ellos se realiza de forma secuencial, sería más adecuado usar una lista ya que esta estructura puede cambiar de tamaño fácilmente durante la ejecución de un programa.

Los arrays nos permiten guardar un gran número de elementos y acceder a ellos de forma independiente. Cada elemento es referenciado por la posición que ocupa dentro del array. Dichas posiciones se llaman índices y siempre son correlativos. Existen tres formas de indexar los elementos de un array:

* Indexación **base-cero(0)**: en este modo, el primer elemento del array será el que ocupe la posición o índice 0.
* **Indexación base-uno(1)**: en este modo, el primer elemento tiene el índice 1.
* **Indexación base-n(n)**: este modo, es un modo versátil de indexación, en el que el índice del primer elemento puede ser elegido libremente.

En JavaScript utilizaremos la indexación base-cero(0) siempre que trabajamos con índices numéricos.

Los arrays se introdujeron en JavaScript a partir de la versión 1.1, lo que significa que no tendrás ningún tipo de problema para poder usar arrays en cualquier navegador disponible actualmente.

|  |
| --- |
| PARA SABER MÁS  En el siguiente enlace puedes consultar la información sobre los tipos de datos estructurados que se han mencionado en este apartado.  [**Estructuras de datos**](http://es.wikipedia.org/wiki/Estructura_de_datos) |

PARA SABER MÁS 1 Estructuras de datos

# Objeto Array.

***Un array es una de las mayores estructuras de datos proporcionadas para almacenar y manipular colecciones de datos.***

A diferencia de otros lenguajes de programación, los arrays de JavaScript son muy versátiles. Esto se debe a los diferentes tipos de datos que podemos almacenar en cada posición del array. Se permite, por ejemplo, tener un array de arrays, proporcionando el equivalente de los arrays multidimensionales pero adaptado a los tipos de datos que necesite nuestra aplicación.

**En programación,** **un array se define como una colección ordenada de datos**. Puedes pensar en un array como si fuera una tabla en la que cada celda contiene datos.

JavaScript emplea los arrays internamente para gestionar los objetos HTML del documento, las propiedades del navegador, etcétera.

Por ejemplo, si tu documento (objeto document) contiene 10 enlaces (links), el navegador mantiene un array con los 10 enlaces. De esta forma, puedes acceder a esos enlaces por su número de enlace (donde el 0 es el primer enlace); es decir, el nombre del array seguido del número índice (o número que indica la posición del enlace en el documento) entre corchetes. Para acceder al enlace que aparece por primera vez dentro del cuerpo de un documento utilizaremos document.links[0].

En la unidad siguiente veremos las colecciones que podemos encontrar en el objeto document (anchors[], forms[], links[] e images[]). Cada una de esas colecciones será un array que contendrá las referencias de todas las anclas, formularios, enlaces e imágenes del documento.

A medida que vayas diseñando tu aplicación, tendrás que identificar el momento más propicio para utilizar los arrays para almacenar datos.

Por ejemplo, imagina que quieres almacenar un conjunto de coordenadas geográficas de una ruta que vas a hacer a caballo: éste sería un buen momento para emplear una estructura de datos de tipo array ya que podríamos asignar nombres a cada posición del array, realizar cálculos, ordenar los puntos, etc. Siempre que veas similitudes con un formato tabular, será una buena opción para usar un array.

Imagina que tuvieras las siguientes variables:

|  |
| --- |
| var coche1="Seat";  var coche2="BMW";  var coche3="Audi";  var coche4="Toyota"; |

El ejemplo anterior sería un buen candidato a convertirlo en un array ya que permitiría introducir más marcas de coche sin tener que crear por ello nuevas variables. Lo haríamos del siguiente modo:

|  |
| --- |
| var misCoches=new Array();  misCoches[0]="Seat";  misCoches[1]="BMW";  misCoches[2]="Audi";  misCoches[3]="Toyota"; |

En la primera línea del ejemplo anterior se ha creado el array y a continuación se ha asignado a cada posición un nombre de coche.

La creación de arrays la veremos en el siguiente apartado.

# Creación de un array e introducción de datos.

**Creación de un array con el constructor new Array().**

Para crear un objeto array, usaremos el constructor new Array(). Por ejemplo:

|  |
| --- |
| var miarray= new Array(); |

Ejemplo 1 Creación de array sin tamaño determinado

Un objeto array dispone de una propiedad length que nos indica su longitud. Ésta será 0 para un array vacío. Si queremos precisar el tamaño del array durante la inicialización podríamos hacerlo pasándole un parámetro al constructor. De esta forma todos los elementos del array serían undefined. En el ejemplo siguiente, se crea un array para almacenar la información de 40 personas:

|  |
| --- |
| var personas = new Array(40); |

Ejemplo 2 Creación de array con tamaño definido

A diferencia de otros lenguajes de programación en los que el hacer el dimensionamiento previo del array puede ofrecer ventajas, en JavaScript no aporta ningún tipo de ventaja especial ya que podremos asignar un valor a cualquier posición del array en cualquier momento, esté o no definido previamente. La propiedad length se ajustará automáticamente al nuevo tamaño del array.

Por ejemplo: podríamos hacer una asignación a personas[53] siendo nuestro array de 40 posiciones y no ocurrirá ningún problema ya que la longitud del array se ajustará automáticamente para poder almacenar la posición 53, con lo que la nueva longitud del array será 54. *(Recuerda que la primera posición del array es la [0] y, si la última es la [53], tendremos 54 elementos en el array.)*

|  |
| --- |
| personas[53] = "Irene Sáinz Veiga";  longitud = personas.length; // longitud toma el valor 54 |

**Introducir datos en un array.**

Introducir datos en un array es tan simple como crear una serie de sentencias de asignación (una por cada elemento del array). En el siguiente ejemplo creamos un array y almacenamos en él los nombres de los planetas del sistema solar:

|  |
| --- |
| sistemaSolar = new Array();  sistemaSolar[0] = "Mercurio";  sistemaSolar[1] = "Venus";  sistemaSolar[2] = "Tierra";  sistemaSolar[3] = "Marte";  sistemaSolar[4] = "Júpiter";  sistemaSolar[5] = "Saturno";  sistemaSolar[6] = "Urano";  sistemaSolar[7] = "Neptuno"; |

Ejemplo 3 Asignación de datos a un array

Aunque esta forma es un poco tediosa a la hora de escribir el código, una vez que las posiciones del array están cubiertas con los datos, acceder a esa información nos resultará muy fácil:

|  |
| --- |
| unPlaneta = sistemaSolar[2]; // unPlaneta tiene el valor de "Tierra" |

Ejemplo 4 Acceso a un elemento del array

**Creación de un array denso.**

Otra forma de crear el array puede ser a través del propio constructor creando lo que se denomina un "**array denso**", aportando al constructor Array() como argumento los datos de cada una de las posiciones del array separados estos datos por comas tal y como puedes observar en el siguiente ejemplo:

|  |
| --- |
| sistemaSolar=new Array("Mercurio","Venus","Tierra","Marte","Jupiter","Saturno","Urano","Neptuno"); |

Ejemplo 5 Crear un array denso

El término de "array denso" quiere decir que los datos están empaquetados dentro del array, sin espacios y comenzando en la posición 0.

**Creación de un array literal.**

Otra forma permitida desde la versión de JavaScript 1.2 es aquella en la que no se emplea el constructor y se definen los **arrays de forma literal**:

|  |
| --- |
| sistemaSolar=["Mercurio","Venus","Tierra","Marte","Jupiter","Saturno","Urano","Neptuno"]; |

Ejemplo 6 Creación de array de forma literal

Los corchetes sustituyen a la llamada al constructor new Array().

Hay que tener cuidado con esta forma de creación que quizás no funcione en navegadores antiguos.

**Creación de un array mixto u objeto literal.**

Existe también otra forma de creación del array: el **array mixto** (o también denominado **objeto literal**), en el que las posiciones son referenciadas con índices de tipo texto o números, pudiéndose mezclar de cualquier forma. Si las posiciones o índices están definidas por un texto, accederemos a ellas utilizando su nombre y no su número (posición relativa) ya que si usamos el número nos daría una posición undefined.

El formato de creación tendrá la sintaxis siguiente:

|  |
| --- |
| nombrearray = { "indice1" : valor , indice2 : "valor" , …} |

Ejemplo 7 Sintaxis de creación de un array mixto

Observa que para definir el array de tipo mixto tenemos que hacerlo comenzando y terminando con llaves { } .

|  |
| --- |
| var datos = { "numero": 42, "mes" : "Junio", "hola" : "mundo", 69 : "96" };  alert(datos["numero"]+" -- "+datos["mes"]+" -- "+datos["hola"]+" -- "+datos[69]); |

Ejemplo 8 Creación de array de tipo mixto

# Recorrido de un array.

Existen múltiples formas de recorrer un array para mostrar todos sus datos. Veamos algunos ejemplos con el array del sistema Solar:

|  |
| --- |
| sistemaSolar = new Array();  sistemaSolar[0] = "Mercurio";  sistemaSolar[1] = "Venus";  sistemaSolar[2] = "Tierra";  sistemaSolar[3] = "Marte";  sistemaSolar[4] = "Júpiter";  sistemaSolar[5] = "Saturno";  sistemaSolar[6] = "Urano";  sistemaSolar[7] = "Neptuno"; |

**Empleando un bucle for:**

|  |
| --- |
| for (i=0;i<sistemaSolar.length;i++)  alert(sistemaSolar[i]); |

Ejemplo 9 Recorrido de un array con for

**Empleando un bucle while:**

|  |
| --- |
| var i=0;  while (i < sistemaSolar.length) {  alert(sistemaSolar[i]);  i++;  } |

Ejemplo 10 Recorrido de un array con while

**Empleando la sentencia for in** (disponible en versiones de JavaScript 1.6 o superiores) cuya sintaxis se muestra a continuación**:**

|  |
| --- |
| for (var variable in objeto) {  //sentencias  } |

Ejemplo 11 Sintaxis sentencia for ... in

|  |
| --- |
| for (var planeta in sistemaSolar) {  alert(sistemaSolar[planeta] |

Ejemplo 12 Recorrido del array con for ... in

***Los arrays de tipo mixto hay que recorrerlos obligatoriamente con FOR ... IN ya que no tienen propiedad length.***

***La propiedad length es siempre el índice cuyo valor es más alto.***

# Borrado de elementos en un array.

Para borrar cualquier dato almacenado en un elemento del array, lo podrás hacer ajustando su valor a null o a una cadena vacía "".

Hasta que apareció el operador delete en las versiones más modernas de los navegadores, no se podía eliminar completamente una posición del array.

***Al borrar un elemento del array, se eliminará su índice de la lista de índices del array, pero no se reducirá la longitud del array.***

Veamos un ejemplo en el código siguiente:

|  |
| --- |
| elarray.length; // resultado: 8  delete elarray[5];  elarray.length; // resultado: 8  elarray[5]; // resultado: undefined |

Ejemplo 13 Borrado con delete

El proceso de borrar una entrada del array no libera la memoria ocupada por esos datos necesariamente. El intérprete de JavaScript, encargado de gestionar la colección de basura en memoria, se encargará de liberar memoria ocupada cuando la necesite.

Si quieres tener un mayor control sobre la eliminación de elementos de un array, deberías considerar usar el método splice(índice, número\_de\_elementos\_a\_eliminar), que está soportado por la mayoría de los navegadores. Este método se puede usar en cualquier array y permite eliminar un elemento o una secuencia de elementos de un array, provocando que la longitud del array se ajuste al nuevo número de elementos.

El operador delete es compatible a partir de estas versiones: WinIE4 +, MacIE4 +, NN4 +, Moz +, Safari +, Opera +, Chrome +.

**Ejemplo de uso del operador** delete**:**

Si consideramos el siguiente array denso:

|  |
| --- |
| var oceanos = new Array("Atlántico","Pacífico","Ártico","Índico"); |

Esta clase de array asigna automáticamente índices numéricos a sus entradas. Para poder acceder posteriormente a los datos podemos hacerlo con un bucle for:

|  |
| --- |
| for (var i=0; i < oceanos.length; i++) {  if (oceanos[i] == "Atlantico") {  // instrucciones a realizar..  }  } |

Si ejecutamos la instrucción:

|  |
| --- |
| delete oceanos[2]; |

Se producirán los siguientes cambios:

* en primer lugar, el tercer elemento del array ("Ártico"), será eliminado del mismo, pero la longitud del array seguirá siendo la misma, y el array resultante será:

|  |
| --- |
| oceanos[0] = "Atlántico";  oceanos[1] = "Pacífico";  oceanos[3] = "Índico"; |

Si intentamos referenciar oceanos[2] nos devolverá un resultado indefinido (undefined).

Si queremos eliminar la tercera posición y que el array reduzca su tamaño podríamos hacerlo con la instrucción:

|  |
| --- |
| oceanos.splice(2,1); // las posiciones del array resultante serán 0, 1 y 2.  oceanos[0] = "Atlántico";  oceanos[1] = "Pacífico";  oceanos[2] = "Índico"; |

***El operador*** *delete****, se recomienda para arrays que usen texto como índices del array, ya que de esta forma se producirán menos confusiones a la hora de borrar los elementos.***

# Propiedades y métodos.

|  |  |
| --- | --- |
| Propiedades del objeto Array | |
| **Propiedad** | **Descripción** |
| constructor | Devuelve la función que creó el prototipo del objeto array. |
| length | Establece o retorna el número de elementos en un array. |
| prototype | Permite añadir propiedades y métodos a un objeto. |

Tabla 1 Propiedades del objeto Array

|  |  |
| --- | --- |
| Métodos del objeto Array | |
| **Método** | **Descripción** |
| [concat()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/concat) | Une dos o más arrays y devuelve el resultado de la unión. |
| [copyWithin()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/copyWithin) | Copia elementos del array , desde y hasta unas posiciones especificadas, sin modificar su longitud. **arr.copyWithin(pos[,ini[,fin]])** |
| [entries()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/entries) | Retorna un nuevo objeto Array Iterator que contiene los pares clave/valor para cada índice de la matriz. |
| [every()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/every) | Chequea si todos los elementos de un array cumplen una condición implementada en una función. (Y) |
| [fill()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/fill) | Rellena los elementos de un array con un contenido concreto desde una determinada posición sin modificar su longitud. **arr.fill(contenido[,ini=0[,fin=this.length]])** |
| [filter()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/filter) | Crea un nuevo array con los elementos del array que cumplen una condición implementada en una función o un array vacío. |
| [find()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/find) | Devuelve el valor del primer elemento del array que cumple una condición implementada en una función o undefined. |
| [findIndex()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/findIndex) | Devuelve el índice del primer elemento del array que cumple una condición implementada en una función o -1. |
| [forEach()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/forEach) | Ejecuta una función con cada uno de los elementos de un array. |
| [includes()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/includes) | Comprueba si un array contiene o no un elemento. |
| [indexOf()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/indexOf) | Retorna la posición en la que se encuentra un elemento del array o -1. |
| [isArray()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/isArray) | Chequea si un objeto es un array. |
| [join()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/join) | Devuelve la cadena de texto formada por todos los elementos de un array concatenados. **join([carácter\_de\_unión])** |
| [keys()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/keys) | Devuelve un objeto Array Iterator que contiene las claves de los índices con los que se puede acceder a cada elemento del array. |
| [lastIndexOf()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/lastIndexOf) | Devuelve la posición en la que se encuentra un elemento en un array pero buscándolo desde el final. |
| [map()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/map) | Crea un array con el resultado de llamar a una función con cada elemento del array. |
| [pop()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/pop) | Elimina y devuelve el último elemento de un array. |
| [push()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/push) | Añade nuevos elementos al final de un array y devuelve la nueva longitud del array. |
| [reduce()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/reduce)  [reduceRight()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/reduce) | Reduce a un único valor el array aplicando una función con cada elemento y con el valor acumulado hasta el momento. |
| [reverse()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/reverse) | Invierte el orden de los elementos en un array. |
| [shift()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/shift) | Elimina el primer elemento de un array y devuelve ese elemento. |
| [slice()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/slice) | Selecciona una parte de un array y devuelve un nuevo array con la parte seleccionada.  a representa la posición del primer elemento a seleccionar.  b representa la posición del último elemento a seleccionar sin estar éste incluído.  Si los elementos son objetos se copia la referencia al objeto. **slice([a[,b]])** |
| [some()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/some) | Chequea si algún elemento del array cumple la condición implementada en una función. (lo mismo que el método **every()** pero un O) |
| [sort()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/sort) | Ordena alfabéticamente los elementos de un array.  Ordenación numérica ascendente: **array.sort(function(a,b){return a-b;});** Ordenación numérica descendente: **array.sort(function(a,b){return b-a;});** |
| [splice()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/splice) | Añade/elimina elementos de un array modificando el array.  **splice(comienzo[,num\_elementos\_a\_borrar[,elemInsert1[,elemInsertX]…]])** |
| [toLocaleString()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/toLocaleString) | Retorna un string representando los elementos del array convirtiendo cada elemento usando las especificaciones locales.  **arr.toLocaleString([**[**locales**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Intl)**[, options]]);** |
| [toString()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/toString) | Retorna un string con el resultado de concatenar todos los elementos del array. |
| [unshift()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/unshift) | Añade nuevos elementos al comienzo de un array y devuelve la nueva longitud. |
| [values()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/values) | Devuelve un objeto Array Iterator que contiene los valores de cada uno de los índices del array. |

Tabla 2 Métodos del objeto Array

En el siguiente ejemplo se muestra el funcionamiento de algunas de los métodos anteriores:

|  |
| --- |
| var frutas = ["Plátano", "Naranja", "Manzana", "Melocotón"];  alert(frutas.slice(0,1)); // resultado: Plátano  alert(frutas.slice(1)); // resultado: Naranja,Manzana,Melocotón  alert(frutas.slice(-2)); // resultado: Manzana, Melocotón  alert(frutas); // resultado: Plátano,Naranja,Manzana,Melocotón  alert(frutas.reverse()); // resultado: Melocotón,Manzana,Naranja,Plátano  alert(frutas.splice(1,2,"Pera") // resultado: Melocotón,Pera,Plátano |

Ejemplo 14 Métodos del objeto Array

|  |
| --- |
| HERRAMIENTA  Desde el siguiente enlace podrás acceder a la página de w3schools donde podrás comprobar el funcionamiento de todas las propiedades y métodos del objeto Array con ejemplos sencillos en los que podrás variar lo que quieras.  [Arrays](http://www.w3schools.com/jsref/jsref_obj_array.asp) |

Herramienta 1 Comprobación del funcionamiento de los arrays

# Arrays paralelos.

El uso de los arrays para almacenar información proporciona gran versatilidad a las aplicaciones, facilitando la gestión de los datos. En algunos casos, resulta de gran utilidad hacer la búsqueda en varios arrays a la vez, de tal forma que podamos almacenar diferentes tipos de datos relacionados entre sí en arrays que se comportan de forma sincronizada.

Cuando tenemos dos o más arrays que utilizan el mismo índice para referirse a datos distintos pero que guardan una relación entre ellos, se denominan arrays paralelos.

Veamos un ejemplo:

Si queremos almacenar los datos: nombre de profesor, asignatura que imparte, número de alumnos que cursan la asignatura y suponiendo que un profesor sólo da una asignatura y dicha asignatura sólo puede darla un profesor, podríamos crear tres arrays de la siguiente forma:

|  |
| --- |
| var profesores = ["Cristina","Catalina","Vieites","Benjamin"];  var modulos=["Seguridad","Bases de Datos","Sistemas Informáticos","Redes"];  var alumnos=[24,17,28,26]; |

Usando los tres arrays de forma sincronizada y haciendo referencia al índice=3 podríamos saber que Benjamín es profesor de Redes y que tiene 26 alumnos en clase.

Veamos un ejemplo de código que recorrería todos los profesores que tengamos en el array imprimiendo la información sobre el módulo que imparten y el número de alumnos que tiene matriculados en el módulo:

|  |
| --- |
| function imprimeDatos(indice) {  alert(profesores[indice]+" imparte el módulo de "+modulos[indice]+" que tiene "+alumnos[indice]+" alumnos en clase.");  }  for (i=0;i<profesores.length;i++) {  imprimeDatos(i);  } |

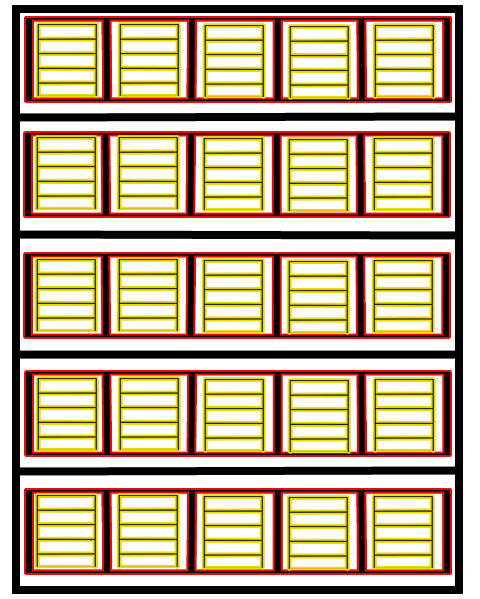
Ejemplo 15 Arrays paralelos

Para que los arrays paralelos sean homogéneos, éstos tendrán que tener la misma longitud, ya que de esta forma se mantendrá la consistencia de la estructura lógica creada.

Entre las ventajas del uso de arrays paralelos tenemos:

* Se pueden usar en lenguajes que soporten solamente arrays como tipos primitivos y no registros (como puede ser JavaScript).
* Son fáciles de entender y utilizar.
* Pueden ahorrar una gran cantidad de espacio evitando, en algunos casos, complicaciones de sincronización.
* El recorrido secuencial de cada posición del array es extremadamente rápido en las máquinas actuales.

# Arrays multidimensionales.

Una alternativa a los arrays paralelos es la simulación de un array multidimensional. Si bien es cierto que en JavaScript los arrays son unidimensionales, podemos crear arrays que cuyos elementos sean, a su vez, arrays u otros objetos. Podemos crear de esta forma arrays bidimen­sionales, tridimensionales, etc.

Podemos realizar el ejemplo anterior (arrays paralelos) creando un array bidimensional de la siguiente forma:

|  |
| --- |
| var datos = new Array();  datos[0] = new Array("Cristina","Seguridad",24);  datos[1] = new Array("Catalina","Bases de Datos",17);  datos[2] = new Array("Vieites","Sistemas Informáticos",28);  datos[3] = new Array("Benjamin","Redes",26); |

Ejemplo 16 Creación de array bidimensional

También podemos usar una definición más breve y literal del array:

|  |
| --- |
| var datos = [ ["Cristina","Seguridad",24],  ["Catalina","Bases de Datos",17],  ["Vieites","Sistemas Informáticos",28],  ["Benjamin","Redes",26] ]; |

Ejemplo 17 Creación de Array bidimensional. Definición breve.

Para acceder a un dato en particular del array anterior (**datos**) se requiere que hagamos una doble referencia. La primera referencia (**datos[1]**) devuelve un array con todos los datos (nombre, módulo y nº de alumnos) del profesor que ocupa la posición 1 (Catalina). La segunda referencia (**datos[1][1]**) cogería sólo uno de los datos (Bases de Datos) de dicho profesor.

|  |
| --- |
| for (i=0;i<datos.length;i++) {  alert(datos[i][0]+" da el módulo "+datos[i][1]+" con "+datos[i][2]+" alumnos");  }  /\*tendría el siguiente resultado  Cristina da el módulo Seguridad con 24 alumnos  Catalina da el módulo Bases de Datos con 17 alumnos  Vieites da el módulo Sistemas Informáticos con 28 alumnos  Benjamin da el módulo Redes con 26 alumnos  \*/ |

Ejemplo 18 Recorrido array bidimensional

# Funciones.

En unidades anteriores ya has utilizado alguna función. **Una función es la definición de un conjunto de acciones pre-programadas**. Las funciones se llaman a través de eventos o bien mediante una orden de nuestro script.

Siempre que sea posible, tienes que diseñar funciones que puedas reutilizar en otras aplicaciones, de esta forma, tus funciones se convertirán en pequeños bloques constructivos que te permitirán ir más rápido en el desarrollo de nuevos programas.

Si conoces otros lenguajes de programación, quizás te suene el término de *subrutina* o *procedimiento*. En JavaScript no vamos a distinguir entre **procedimientos** (que ejecutan acciones), o **funciones** (que ejecutan acciones y devuelven un valor). En JavaScript siempre se llamarán funciones.

# Creación y uso de funciones

Lo primero que hay que conocer es la sintaxis de definición de una función, que puedes ver a continuación:

|  |
| --- |
| function nombreFunción ( [parámetro1]....[, parámetroN] ) {  // instrucciones  [ return valor; ]  } |

Ejemplo 19 Sintaxis de definición de una función

Como se desprende de la sintaxis anterior, la definición de una función consta de:

* La declaración de la función mediante la palabra reservada **function** seguida de un nombre de función escogido por el/la creador/a del código y unos paréntesis que pueden contener o no, uno o más argumentos o parámetros, separados (en el caso de ser más de uno) por comas.
* El código que ejecutará la función en el momento en que se llame. Este código va encerrado entre llaves a continuación de la declaración.
* La última instrucción de una función debe ser (en caso de existir) la de retornar el valor al programa principal desde donde fue llamada.

Una función es capaz de devolver un valor a la instrucción que la invocó, pero esto no es un requisito obligatorio en JavaScript. Cuando una función devuelve un valor, la instrucción que llamó a esa función, la tratará como si fuera una expresión.

A la hora de definir una función, ten en cuenta que:

* Los nombres que puedes asignar a una función tendrán las mismas restricciones que tienen los elementos HTML y las variables en JavaScript. Deberías asignarle un nombre que realmente la identifique, o que indique qué tipo de acción realiza. Puedes usar palabras compuestas como **chequearMail** o **calcularFecha**. Fíjate que el nombre de las funciones suele incluir un verbo, puesto que las funciones son elementos del lenguaje que realizan acciones.
* Las funciones deben ser muy específicas, es decir, que no realicen tareas adicionales a aquellas para la que fueron pensadas. En el ejemplo anterior, si tenemos una función chequearMail, su código debe servir única y exclusivamente a ese propósito y debería devolver un valor que identifique si el mail es o no correcto. En ningún caso debería valer para hacer otro tipo de cálculo o validación de otro dato.

Las funciones se llaman en cualquier momento del programa. La sintaxis de esta llamada dependerá de si la función tiene o no un valor de retorno. El siguiente ejemplo muestra las dos formas que tenemos de llamar a la función:

|  |
| --- |
| nombreFuncion([argumento1]…[,argumentoN]); // 1. sin valor de retorno  variable=nombreFuncion([argumento1]…[,argumentoN]); // 2. con valor de retorno  if (nombreFuncion([argumento1]…[,argumentoN]))==valor) .... |

Ejemplo 20 Sintaxis de la llamada a una función

# Parámetros

Como habrás visto en el apartado anterior, una función puede definirse con o sin parámetros.

A la hora de definir una función que recibe parámetros, lo que haremos es, escribir los nombres de las variables que recibirán esos parámetros entre los paréntesis de la función.

Las variables que usamos como parámetros en la definición de la función no usan la palabra reservada **var** para su inicialización

***Los parámetros son variables locales a la función que se inicializan automáticamente en el momento de llamar a la función con los valores que se le pasan en la llamada.***

Al realizar la llamada a la función se escribirán los valores (o variables que tiene dichos valores) en el mismo orden en el que figuran en la definición.

***El número y posición de los argumentos en la definición y en la llamada a la función debe ser el mismo.***

El siguiente ejemplo muestra un pequeño programa que devuelve el mayor de dos números introducidos por el usuario. El programa utiliza una función para devolver el mayor de los dos.

|  |
| --- |
| function devolverMayor(a,b) {  if (a>b) {  return a;  } else {  return b;  }  }  var numero1=parseInt(prompt("Introduce un primer número"));  var numero2=parseInt(prompt("Introduce un segundo número"));  if (numero1!=numero2) {  alert("El mayor de ambos números es "+devolverMayor(numero1,numero2));  } else {  alert("Ambos números son iguales");  } |
|  |

Ejemplo 21 Programa que devuelve el mayor de dos números

***Cuando se pasan arrays y/o objetos de usuario en la llamada a una función, los parámetros reciben la referencia (dirección de memoria) al array o al objeto, por lo que cualquier cambio realizado en el parámetro en el código de la función afecta al propio array u objeto global.***

# Ámbito de las variables.

Vimos, ya en el primer tema, que una variable declarada mediante la palabra reservada **var** es local al lugar donde se define, siendo global en caso de no utilizar dicha palabra reservada.

En el apartado anterior vimos que las variables utilizadas como parámetros en la declaración de una función son variables locales a la función y se crean en el momento de la llamada a la misma sin necesidad de utilizar la palabra reservada **var**.

Cuando hablamos de **local** y **global** nos estamos refiriendo al ámbito de las variables, su alcance o zona del programa donde estas variables son conocidas.

Una variable global en JavaScript tiene una connotación un poco diferente respecto a otros lenguajes de programación. Para un script de JavaScript, el alcance de una variable global, se limita al documento actual que está cargado en la ventana del navegador o en un **frame**. Cuando se inicializa una variable como variable global, todas las instrucciones del script (incluidas las instrucciones que están dentro de las funciones) tendrán acceso directo al valor de esa variable. Todas las instrucciones podrán leer y modificar el valor de esa variable global.

En el momento que una página se cierra, todas las variables definidas en esa página se eliminarán de la memoria para siempre. Si necesitas que el valor de una variable persista de una página a otra, tendrás que utilizar técnicas que te permitan almacenar esa variable (como las cookies, etc.).

Aunque el uso de la palabra reservada **var** para inicializar variables es opcional, te recomiendo que la uses ya que así te protegerás de futuros cambios en las próximas versiones de JavaScript.

Antes recordamos que podemos definir variables en los parámetros de una función sin usar la palabra reservada **var**, pero es posible que necesitemos definir nuevas variables dentro del código de la función. En este caso, tendremos que usar la palabra reservada **var** para garantizar que el ámbito de la variable es local a la función y que fuera de la función no será conocida. Es decir, ninguna otra función o instrucción que esté fuera de la función donde está definida podrá acceder al valor de dicha variable.

Reutilizar el nombre de una variable global como local es uno de los bugs más sutiles y por consiguiente más difíciles de encontrar en el código de JavaScript. La variable local en momentos puntuales ocultará el valor de la variable global, sin avisarnos de ello. Como recomendación, no reutilices un nombre de variable global como local en una función, y, sobretodo, no declares una variable global dentro de una función, ya que podrás crear fallos que te resultarán difíciles de solucionar.

# Funciones anidadas.

Los navegadores más modernos nos proporcionan la opción de anidar unas funciones dentro de otras. Es decir, podemos programar una función dentro de otra función.

Las funciones que hemos visto hasta ahora son funciones globales, accesibles desde cualquier parte del código. Con las funciones anidadas, podemos encapsular la accesibilidad de una función dentro de otra y hacer que esa función sea privada o local a la función en la que está contenida. La estructura de las funciones anidadas la puedes ver en el siguiente ejemplo:

|  |
| --- |
| function globalA() {  Una buena opción para aplicar las funciones anidadas, es cuando tenemos una secuencia de instrucciones que necesitan ser llamadas desde múltiples sitios dentro de una función, y esas instrucciones sólo tienen significado dentro del contexto de esa función principal. En otras palabras: en lugar de romper la secuencia de una función muy larga en varias funciones globales, haremos lo mismo, pero utilizando funciones locales.  function localA() {  // instrucciones  }  // instrucciones  }  function globalB() {  function localB1() {  // instrucciones  }  function localB2() {  // instrucciones  }  // instrucciones  } |

Ejemplo 22 Sintaxis de las Funciones anidadas

Ejemplo de una función anidada:

|  |
| --- |
| function totalVentaArticulo(articulo,cantidad) {  function obtenerPrecio(articulo) {  // instrucciones  return precio;  }  function obtenerStock(articulo) {  // instrucciones  return stock;  }  function obtenerIva(articulo) {  // instrucciones  return iva;  }  if (obtenerStock(articulo)>=cantidad) {  return obtenerPrecio(articulo)\*cantidad\*(1+obtenerIva(articulo));  } else {  return 0;  }  **//...............instrucciones del programa principal**  **if (totalVentaArticulo("000123",20)>0) {**  **alert("El precio total de la venta es" + totalVentaArticulo("000123",20)+"€");**  **} else {**  **alert("No hay suficiente stock del articulo 000123");**  **}**  **//...............instrucciones** |

Ejemplo 23 Ejemplo de funciones anidadas

# Funciones anónimas.

A estas alturas, después de consultar los ejemplos de los métodos de los arrays habrás visto que en muchos de ellos se han empleado este tipo de funciones.

Las funciones anónimas son funciones que no tienen un nombre por lo que no se declaran como el resto de las funciones.

Son funciones que son creadas mediante una expresión que se asigna a una variable. Será esta variable la que utilicemos después para llamar a la función.

En el ejemplo siguiente podemos ver la diferencia en el uso de las funciones con nombre y las anónimas:

|  |
| --- |
| **//...............función declarada con nombre**  function por2(numero) {  return numero\*2;  }  **// uso de la función**  **var num=por2(4);**  **alert(num); //enseñaría un 8**  **//. . . . . . .hacemos uso de una función anónima**  **//. . . . . . .le asignamos a num la expresión de la función**  **var num=function(numero) {return numero\*2;};**  **//aquí después de la llave hay un ; porque estamos haciendo una asignación**  **//usamos la función**  **alert(num(4));**  **//4 es el argumento de la función anónima** |

Ejemplo 24 Uso de funciones anónimas

# Funciones predefinidas del lenguaje.

Después de ver los arrays habrás podido observar que todos los métodos en realidad son funciones (que usan o no parámetros dentro de sus paréntesis). Pues bien, en JavaScript disponemos de algunos elementos que necesitan ser tratados a escala global y que no pertenecen a ningún objeto en particular (o que se pueden aplicar a cualquier objeto).

A continuación, se muestran algunos de las propiedades y métodos que se pueden utilizar a nivel global en cualquier parte de tu código de JavaScript.

|  |  |
| --- | --- |
| Propiedades globales de JavaScript | |
| **Propiedad** | **Descripción** |
| [Infinity](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Infinity) | Un valor numérico que representa el infinito positivo/negativo. |
| [NaN](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/NaN) | Valor que no es numérico "Not a Number". |
| [undefined](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/undefined) | Indica que a esa variable no le ha sido asignado un valor. |
| [null](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/null) | Literal de Javascript que representa un valor nulo o "vacío". |

Tabla 3 Propiedades globales de JavaScript

|  |  |
| --- | --- |
| Métodos globales de JavaScript | |
| **Método** | **Descripción** |
| [decodeURI()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/decodeURI) | Decodifica los caracteres especiales de una URL excepto:  , / ? : @ & = + $ # |
| [decodeURIComponent()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/decodeURIComponent) | Decodifica todos los caracteres especiales de una URL codificados previamente con encodeURIComponent() |
| [encodeURI()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/encodeURI) | Codifica los caracteres especiales de una URL excepto:  , / ? : @ & = + $ # |
| [encodeURIComponent()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/encodeURIComponent) | Codifica todos los caracteres especiales de una URL. |
| [eval()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/eval) | Evalúa una cadena y la ejecuta si contiene código u operaciones. |
| [isFinite()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/isFinite) | Determina si un valor es un número finito válido. |
| [isNaN()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/isNaN) | Determina cuándo un valor no es un número. |
| [parseFloat()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/parseFloat) | Convierte una cadena a un número real. |
| [parseInt()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/parseInt) | Convierte una cadena a un entero. **parseInt(*string*, *base*)** |

Tabla 4 Métodos globales de JavaScript

|  |
| --- |
| eval("x=50;y=30;alert(x\*y)"); // Visualiza 1500  alert(eval("8+6")); // Visualiza 14  alert(eval(x+30)); // Visualiza 80 |

Ejemplo 25 Uso de la función global eval()

|  |
| --- |
| RECOMENDACIÓN  Además de los enlaces en cada uno de los métodos y los propiedades, desde el siguiente enlace puedes comprobar el funcionamiento de estas propiedades y funciones predefinidas del JavaScript con la ayuda de w3schools.  [Propiedades y funciones predefinidas](http://www.w3schools.com/jsref/jsref_obj_global.asp) |

Recomendación 1 Propiedades y funciones predefinidas

# Buenas prácticas en el diseño y codificación de funciones.

Al igual que ya vimos en la unidad 2 respecto a la forma adecuada de nombrar las variables, con las funciones debemos seguir manteniendo la misma rutina: **los nombres de las variables deben ser descriptivos de la labor que realizan.**

|  |
| --- |
| //El siguiente nombre es correcto porque indica que la función nos dirá si es o no es válido un DNI  function esValidoDNI(dni) {  //Aquí iría el código de la función  //En alguna parte del código se encontraría con un return false o return true;  }  var dni="36750893G";  if (esValidoDNI(dni) {  alert("El DNI Nº"+dni+"es correcto";  } else {  alert("El DNI Nº"+dni+"es incorrecto";  }  // nombre de función incorrecto. No sabemos lo que hace  function DNI(dni) {  //Aquí iría el código de la función  } |

Ejemplo 26 Nombres correctos e incorrectos de funciones

Cuando se emplea una función para agrupar un trozo de código que se va a ejecutar en diferentes partes del programa (actuando, así como un procedimiento), se suele emplear un verbo como parte del nombre de la función.

|  |
| --- |
| //El siguiente nombre es correcto porque indica su función  function visualizarDNIs(listaDNI) {  //Aquí iría el código de la función que recorrería la lista y visualizaría  }  var listaDNIs=["36750893G","36660893H","36750553I","36711193A","36700093P"];  visualizarDNIs(listaDNIs); |

Ejemplo 27 Nombres correctos e incorrectos de funciones

***La filosofía del diseño de funciones es aplicar el método DRY (don´t repeat yourself) agrupando código que se va a utilizar en diferentes partes del documento. Esto contribuye a un fácil mantenimiento de la aplicación.***

A la hora de escribir un programa podemos optar por agrupar las funciones que definamos en un archivo separado del propio programa que las usa. De esta forma, podemos crear nuestras propias librerías de funciones que podemos usar en nuestros programas. Eso sí, no hay que olvidar que en la cabecera del html deberemos enlazar todos los archivos JavaScript que necesitemos en cada momento.

Por legibilidad y en beneficio de un fácil mantenimiento de la aplicación:

* No debemos intercalar definición de funciones con el programa principal.
* La longitud (en nº de líneas) de la función no debe sobrepasar la altura del espacio de trabajo. Se trata de evitar el uso del scroll cuando reviso el código de la función.

***La filosofía de las funciones es “divide y vencerás” por lo que una función se diseña para realizar alguna tarea concreta.***

# Creación de objetos a medida.

En JavaScript puedes crear tus propios objetos en memoria, objetos con propiedades y métodos que podrás definir a tu manera. Estos objetos no serán elementos de la página de interfaz de usuario, pero sí que serán objetos que podrán contener datos (propiedades) y funciones (métodos), cuyos resultados sí que se podrán mostrar en el navegador. Tú controlarás la estructura del objeto, sus datos y su comportamiento.

También hay que dejar claro que, JavaScript no es un lenguaje orientado a objetos en sentido estricto. JavaScript es un lenguaje basado en prototipos que no contiene ninguna declaración de clase, como se encuentra, por ejemplo, en C + + o Java. Esto es a veces confuso para los programadores acostumbrados a los lenguajes con una declaración de clase. En su lugar, JavaScript utiliza funciones como clases. Definir una clase es tan fácil como definir una función.

Un objeto en JavaScript es realmente una colección de propiedades. Las propiedades pueden tener forma de datos, tipos, funciones (métodos) o incluso otros objetos. De hecho, sería más fácil de entender un objeto de JavaScript como un array de valores, cada uno de los cuales está asociado a una propiedad (un tipo de datos, método u objeto). Un momento: ¿un método puede ser una propiedad de un objeto? Pues en JavaScript parece que sí.

Una función contenida en un objeto se conoce como un método. Los métodos no son diferentes de las funciones que has visto anteriormente, excepto que han sido diseñados para ser utilizados en el contexto de un objeto y, por lo tanto, tendrán acceso a las propiedades de ese objeto. Esta conexión entre propiedades y métodos es uno de los ejes centrales de la orientación a objetos.

Los objetos se crean empleando una función especial denominada constructor, determinada por el nombre del objeto. Ejemplo de una función constructor:

|  |
| --- |
| function Coche( ){  // propiedades y métodos  } |

Ejemplo 28 Constructor de objeto

Esta función, que por el momento no contiene código, es la base para crear objetos de tipo Coche. Puedes pensar en un constructor como un molde o plantilla, que será utilizada para crear objetos. Por convención, los nombres de los constructores se ponen generalmente con las iniciales de cada palabra en mayúscula y cuando creamos un objeto con ese constructor (instancia de ese objeto), lo haremos empleando minúsculas al principio. Por ejemplo: **var unCoche = new Coche();**

La palabra reservada **new** se emplea para crear objetos en JavaScript. Al crear la variable **unCoche** de esta forma, técnicamente podríamos decir que hemos creado una instancia de la clase **Coche**, o que hemos instanciado el objeto **Coche**, o que hemos creado un objeto **Coche**, etc. Es decir, hay varias formas de expresarlo, pero todas ellas quieren decir lo mismo.

|  |
| --- |
| PARA SABER MÁS  En el siguiente enlace puedes consultar un artículo que habla de la Orientación a Objetos en JavaScript.  [**Orientación a Objetos en JavaScript**](https://developer.mozilla.org/es/docs/Web/JavaScript/Introducci%C3%B3n_a_JavaScript_orientado_a_objetos) |

PARA SABER MÁS 2 Orientación a Objetos en JavaScript

# Definición de propiedades.

Una vez que ya sabemos cómo crear un constructor para un objeto, vamos a ver cómo podemos crear una propiedad específica para ese objeto. Las propiedades para nuestro objeto se crearán dentro del constructor empleando para ello la palabra reservada **this**. Observa el siguiente ejemplo:

|  |
| --- |
| function Coche(){  // Propiedades  this.marca = "Audi A6";  this.combustible = "diesel";  this.cantidad = 0; // Cantidad de combustible en el depósito.  } |

Ejemplo 29 Propiedades de un objeto

La palabra reservada **this**, se utiliza para hacer referencia al objeto actual, que en este caso será el objeto que está siendo creado por el constructor. Por lo tanto, usarás **this**, para crear nuevas propiedades para el objeto. El único problema del ejemplo anterior es que todos los coches que hagamos del tipo Coche serán siempre Audi A6, diésel y sin combustible en el depósito. No parece muy lógico ¿no crees?

|  |
| --- |
| var cocheDeMartin = new Coche();  var cocheDeSilvia = new Coche(); |

El ejemplo anterior estaría muy bien si tanto Martín como Silvia tuvieran el mismo modelo de coche y ambos tuvieran el depósito vacío. Esto no es lo que suele ocurrir en el mundo real por lo que nunca tendremos un constructor de objetos totalmente iguales. Los valores de las propiedades de los objetos deberían ser configurables en el momento de la creación del objeto. Para ello, hacemos uso de los argumentos o parámetros en el constructor. El siguiente ejemplo es una variación del constructor Coche para que permita crear cualquier tipo de coche. También se muestra en el ejemplo la forma en la que se instanciaría cada uno de los coches (dando un valor concreto a cada uno de los argumentos).

|  |
| --- |
| function Coche(propietario,marca,combustible,litros){  this.propietario = propietario;  this.marca = marca;  this.combustible = combustible;  this.cantidad = litros;  }  var coche1 = new Coche("Martín","Volkswagen Golf","gasolina",45);  var coche2 = new Coche("Silvia","Mercedes SLK","diesel",55); |

Ejemplo 30 Constructor de coches

Una vez creados dos objetos de tipo Coche, podemos consultar y/o modificar sus propiedades tal y como puedes ver en el siguiente ejemplo.

|  |
| --- |
| alert(coche1.propietario+" tiene un "+coche1.marca);  alert(coche2.propietario+" tiene un "+coche2.marca);  coche1.marca = "BMW X5";  coche1.combustible = "diesel";  alert(coche1.propietario+" tiene ahora un "+coche1.marca);  //lo que obtendríamos sería:  // Martín tiene un Volkswagen Golf  // Silvia tiene un Mercedes SLK  // Martín tiene ahora un BMW X5 |

Ejemplo 31 Acceso y Modificación de las propiedades de un objeto

# Definición de métodos.

Las propiedades son solamente la mitad de la ecuación de la orientación a objetos en JavaScript. La otra mitad son **los métodos o funciones asociadas a los objetos**.

¿Te acuerdas de los que decíamos al principio de este apartado?

*"Un objeto en JavaScript es realmente una colección de propiedades. Las propiedades pueden tener forma de datos, tipos, funciones (métodos) o incluso otros objetos. De hecho, sería más fácil de entender un objeto de JavaScript como un array de valores, cada uno de los cuales está asociado a una propiedad (un tipo de datos, método u objeto). Un momento: ¿un método puede ser una propiedad de un objeto? Pues en JavaScript parece que sí."*

Vamos a verlo con un ejemplo. Supón que queremos tener una forma de rellenar el depósito de cualquier coche. Podemos definir una propiedad **rellenarDepósito** que permita modificar la **cantidad** de litros del depósito.

Esta operación la podemos realizar de dos formas:

1. Usando una función anónima (sin nombre) declarada dentro del constructor.
2. Asignando como valor a una de las propiedades del constructor una función externa.

El siguiente ejemplo muestra las dos formas de hacerlo:

|  |
| --- |
| *//Forma 1*  *Fíjate que las funciones anónimas llevan un punto y coma después de la llave de cierre.*  *Esto se debe a que terminamos la definición del valor asignado a una propiedad.*  function Coche(propietario,marca,combustible){  this.propietario = propietario;  this.marca = marca;  this.combustible = combustible;  this.cantidad=0; //todos los coches se inicializan con 0 litros en el depósito  this.rellenarDeposito = function (litros){ //función anónima  this.cantidad=litros;  };  }  *//Forma 2*  function rellenarDeposito(litros){  this.cantidad = litros;  }  function Coche(propietario,marca,combustible){  this.propietario = propietario;  this.marca = marca;  this.combustible = combustible;  this.cantidad=0;  this.rellenarDeposito = rellenarDeposito; //función externa  }  //En ambos casos el método o función se usa de la misma manera  var coche1 = new Coche("Martín","Volkswagen Golf","gasolina");  coche1.rellenardeposito(45); //rellenamos el depósito con 45 litros |

Ejemplo 32 Definición y uso de métodos de objetos

La segunda forma define el método **rellenarDeposito** a nivel global. Ésta no es la mejor práctica en la programación orientada a objetos. La primera forma es una mejor aproximación a la orientación a objetos donde los métodos definidos dentro del constructor aportan mayor privacidad y seguridad al objeto.

Siguiendo un poco con el tema de las mejores prácticas, podemos decir que el acceso a las propiedades que hacíamos en el subapartado anterior no es el más adecuado en la orientación a objetos, ya que realmente se suele acceder a estas propiedades a través de los métodos definidos en el propio objeto. Es lo que se conoce como Encapsulamiento en la Programación Orientada a Objetos.

Entonces, sin son las funciones o métodos de los objetos lo que nos permite acceder a las propiedades de un objeto, ¿cómo podríamos saber el coche que tiene Martín o Silvia? La respuesta es simple: definiendo un método que devuelva la marca del coche de un determinado propietario. El siguiente ejemplo muestra la forma en la que haríamos el método y cómo lo emplearíamos para hacer la consulta.

|  |
| --- |
| function Coche(propietario,marca,combustible){  this.propietario = propietario;  this.marca = marca;  this.combustible = combustible;  this.cantidad=0; //todos los coches se inicializan con 0 litros en el depósito  this.imprimeMarca = function(){  return this.marca;  };  this.imprimePropietario = function(){  return this.propietario;  };  this.cambiarCoche = function(marca,combustible) {  this.marca = marca;  this.combustible = combustible;  };  this.venderCoche = function(nuevoPropietario) {  this.propietario = nuevoPropietario;  };    }  var c = new Coche("Martín","Volkswagen Golf","gasolina");  alert(c.imprimePropietario()+" conduce un "+c.imprimeMarca());  c.cambiarCoche("Toyota","Diesel");  alert(c.imprimePropietario()+" conduce ahora un "+c.imprimeMarca());  c.venderCoche("Enrique");  alert(c.imprimePropietario()+" es ahora el nuevo propietario del "+c.imprimeMarca()); |

Ejemplo 33 Encapsulamiento

# Definición de objetos literales.

Otra forma de definir objetos es hacerlo de forma literal.

***Los literales de objeto se utilizan para almacenar información en parejas nombre-valor. Un literal de objeto se define mediante llaves ({ y }) dentro las cuales podemos colocar cualquier número de parejas nombre-valor, definida mediante una cadena, un símbolo de dos puntos y el valor. Cada pareja nombre-valor deben estar separadas por coma.***

**cadena**

**valor**

Ilustración 1 Representación de la sintaxis de la definición literal de un objeto

Al igual que ocurría con los arrays, tenemos varias formas de definir los objetos de forma literal. Lo podemos ver en el siguiente ejemplo:

|  |
| --- |
| Fotografía de un avión Boeing 747 en carrera de despegue.*//Forma 1*  avion = { "marca":"Boeing","modelo":"747","pasajeros":450 };  *//Forma 2*  var avion = new Object();  avion.marca = "Boeing";  avion.modelo = "747";  avion.pasajeros = 450; |

Ejemplo 34 Definición de objetos literales

Para referirnos desde JavaScript a una propiedad del objeto **avion** podríamos hacerlo, también, de dos formas:

|  |
| --- |
| *//Forma 1 (a través de su propiedad)*  alert(avion.marca);  *//Forma 2*  alert(avion["marca"]); //usando el nombre de la propiedad como índice |

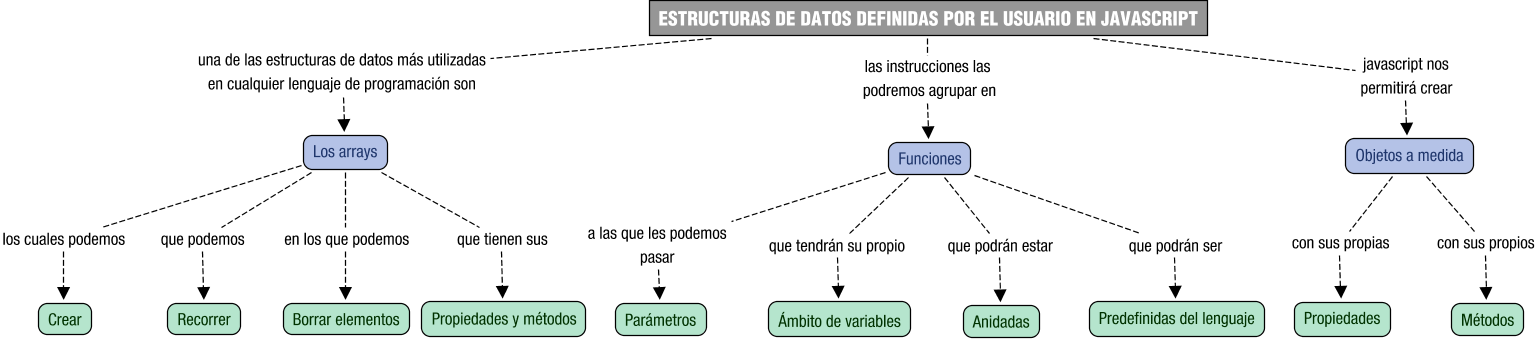
Ejemplo 35 Acceso a la propiedad del objeto

Incluso podríamos tener un conjunto de objetos literales simplemente creando un array que contenga en cada posición una definición de un objeto literal. El siguiente ejemplo es una muestra de ello:

|  |
| --- |
| var datos=[  {"id":"2","nombrecentro":"IES A Piringalla" ,"localidad":"Lugo","provincia":"Lugo"},  {"id":"10","nombrecentro":"IES As Fontiñas","localidad":"Santiago","provincia":"A Coruña"},  {"id":"9","nombrecentro":"IES A Carballeira","localidad":"Ourense","provincia":"Ourense"},  {"id":"4","nombrecentro":"IES de Teis","localidad":"Vigo","provincia":"Pontevedra"}];  **for (var i=0; i< datos.length; i++){**  **alert("El centro "+datos[i].nombrecentro+" con ID nº "+datos[i].id + " está en ");**  **alert(datos[i].localidad+" en la provincia de " +datos[i].provincia);**  **}** |

Ejemplo 36 Definición y recorrido de un array de objetos

Mapa conceptual



GLOSARIO

Bug. Es un típico fallo en programación. Generalmente se refiere a fallos o errores de tipo lógico, que suelen ser más difíciles de detectar que los errores sintácticos, los cuáles sí que son mostrados por el analizador sintáctico del lenguaje.

Indexación.Es el proceso que consiste en registrar ordenadamente la información para elaborar un índice, con la finalidad de obtener resultados de forma más rápida y relevante a la hora de realizar búsquedas.