

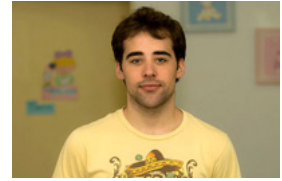
Caso práctico

Antonio ha avanzado mucho en su estudio de JavaScript, viendo los principales objetos con los que puede trabajar, sus métodos y propiedades.

Ha llegado el momento, de ver cómo puede organizar los datos que genera en su aplicación de JavaScript, de forma que le permita gestionarlos más eficientemente.

También verá cómo crear funciones, el alcance de las variables y cómo utilizarlas en su código. Y para terminar **Juan**, le va a explicar un poco más en detalle, cómo podrá crear nuevos objetos y asignarles propiedades y métodos en JavaScript.

Antonio ha comenzado a analizar más a fondo parte del trabajo que tiene que realizar, y comenta con su directora **Ada**, la posibilidad de hacer algunos bocetos de las innovaciones y mejoras que quiere aportar al proyecto. **Ada** está muy contenta con los progresos de **Antonio** y está deseando ver los bocetos.



Elaboración Propia

1.- Estructuras de datos.

Caso práctico



[Ricardo Luque](#)

Antonio comienza a estudiar las estructuras de datos que le permitirán almacenar la información que gestionará en sus scripts, de una forma mucho más organizada. Existen muchos tipos de estructuras de datos, pero en este caso **Antonio** profundizará en una de ellas "los arrays".

Verá cómo crearlos, recorrerlos para consultar su información, borrar elementos del array, y las propiedades y métodos disponibles para su gestión.

Terminará estudiando lo que son arrays paralelos y arrays multidimensionales, los cuales le permitirán ampliar la capacidad y ventajas que aporta un array estándar.

En los lenguajes de programación existen estructuras de datos especiales, que nos sirven para guardar información más compleja que una variable simple. En la segunda unidad vimos como crear variables y cómo almacenar valores (simples), dentro de esas variables.

Hay un montón de tipos de estructuras de datos (listas, pilas, colas, árboles, conjuntos,...), que se pueden utilizar para almacenar datos, pero una estructura de las más utilizadas en todos los lenguajes es el array. El **array**, es como una variable o zona de almacenamiento continuo, donde podemos introducir varios valores en lugar de solamente uno, como ocurre con las variables normales.

Los arrays también se suelen denominar **matrices** o **vectores**. Desde el punto de vista lógico, una matriz se puede ver como un conjunto de elementos ordenados en fila (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 se va a utilizar acceso 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 montón de elementos y acceder a ellos de manera 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 la componente 0, es decir tendrá el 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 cuando trabajamos con índices numéricos utilizaremos la *indexación base-cero(0)*.

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

Para saber más

[Más información y tipos de estructuras de datos.](#)

Citas para pensar

"Lo que el estilo es a la persona, la estructura es a la obra."

Goytisolo, Luis.

1.1.- Objeto Array.

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



[Aaron Gustafson](#)

otros lenguajes de programación, los arrays en JavaScript son muy versátiles, en el sentido de los diferentes tipos de datos que podemos almacenar en cada posición del array. Esto nos permite por ejemplo, tener un array de arrays, proporcionando la equivalencia de 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**. Lo mejor es que pienses en un array como si fuera una tabla que contiene datos, o también como si fuera una hoja de cálculo.

JavaScript emplea un montón de arrays internamente para gestionar los objetos HTML en el documento, propiedades del navegador, etc. Por ejemplo, si tu documento contiene 10 enlaces, el navegador mantiene una tabla con todos esos enlaces. Tú podrás acceder a esos enlaces por su número de enlace (comenzando en el 0 como primer enlace).

Si empleamos la sentencia de array para acceder a esos enlaces, el nombre del array estará seguido del número índice (número del enlace) entre corchetes, como por ejemplo: `document.links[0]`, representará el primer enlace en el documento.

En la unidad anterior si recuerdas, teníamos colecciones dentro del objeto `document`. Pues bien, cada una de esas colecciones (`anchors[]`, `forms[]`, `links[]`, e `images[]`) 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 las pistas que te permitan utilizar arrays para almacenar datos. Por ejemplo, imagínate que tienes que almacenar un montón de coordenadas geográficas de una ruta a caballo: ésto si que sería un buen candidato a 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 de tabla, será una buena opción para usar un array.

Por ejemplo si tenemos las siguientes variables:

```
1 | var coche1="Seat";
2 | var coche2="BMW";
3 | var coche3="Audi";
4 | var coche4="Toyota";
```

Este ejemplo sería un buen candidato a convertirlo en un array, ya que te permitiría introducir más marcas de coche, sin que tengas que crear nuevas variables para ello. Lo haríamos del siguiente modo:

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

Te explicaremos más en detalle las formas de creación de un array en el siguiente apartado.

Autoevaluación

A la hora de referenciar las posiciones de un array en JavaScript, que estén definidas de forma numérica, la primera posición de ese array comenzará con el índice 0.

- ☐ Verdadero.
- ☐ Falso.

Es correcto, la primera posición en un array será la posición 0.

Es incorrecto, la primera posición de un array se referencia con el índice 0.

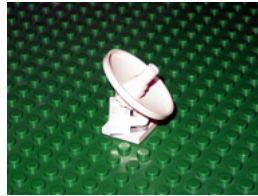
Solución

1. Opción correcta
2. Incorrecto

1.1.1.- Creación de un array.

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

```
1 | var miarray= new Array();
```



[kreezzalee_Very Large Array Module](#)

Un objeto array dispone de una propiedad que nos indica su longitud. Esta propiedad es `length` (longitud, que será 0 para un array vacío). Si queremos precisar el tamaño del array durante la inicialización (por ejemplo para que se cargue con valores `<null`), podríamos hacerlo pasándole un parámetro al constructor. Por ejemplo, aquí puedes ver como crear un nuevo array que almacene la información de 40 personas:

```
1 | var personas = new Array(40);
```

A diferencia de otros lenguajes de programación, en los que hacer el dimensionamiento previo del array tiene ventajas, en JavaScript no nos dará ningún tipo de ventaja especial, ya que podremos asignar un valor a cualquier posición del array en cualquier momento, esté o no definida previamente. La propiedad `length` se ajustará automáticamente al nuevo tamaño del array. Por ejemplo podríamos hacer una asignación tal como `personas[53]` y eso que nuestro array es de 40 posiciones. En este caso no ocurrirá ningún problema, ya que la propiedad `length` del array se ajustará automáticamente para poder almacenar la posición 53, con lo que la nueva longitud del array será 54, ya que la primera posición del array es la [0] y la última es la [53], tendremos por lo tanto 54 elementos en el array.

```
1 | personas[53] = "Irene Sáinz Veiga";
2 | longitud = personas.length;      // asignará a la variable longitud 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. Ejemplo de un array que contiene el nombre de los planetas del sistema solar:

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

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

```
1 | unPlaneta = sistemaSolar[2];      // almacenará en unPlaneta la cadena "Tierra".
```

Otra forma de crear el array puede ser mediante el constructor. En lugar de escribir cada sentencia de asignación para cada elemento, lo podemos hacer creando lo que se denomina un **"array denso"**, aportando al **constructor Array()**, los datos a cubrir separados por comas:

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

El término de **"array denso"** quiere decir que los datos están empaquetados dentro del array, sin espacios y comenzando en la posición 0. Otra forma permitida a partir de la versión de JavaScript 1.2+, sería aquella en la que no se emplea el constructor y se definen los arrays de forma literal:

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

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.

Y para terminar vamos a ver otra forma de creación de un **array mixto** (o también denominado objeto literal), en el que las posiciones son referenciadas con índices de tipo texto o números, mezclándolos de forma aleatoria. Si las posiciones índice están definidas por un texto, para acceder a ellas lo haremos utilizando su nombre y no su número (en este caso si usamos el número nos daría una posición **undefined**). El formato de creación sería: `nombrearray = { "indice1" : valor , indice2 : "valor" , ...}`

(fijate que en este caso para definir el array de tipo mixto tendremos que hacerlo comenzando y terminando con **llaves { }**). Por ejemplo:

```
1 | var datos = { "numero": 42, "mes" : "Junio", "hola" : "mundo", 69 : "96" };
2 | document.write("<br/>" + datos["numero"] + " -- " + datos["mes"] + " -- " + datos["hola"] + " -- " + datos[69] + "<br/>");
```

También podemos hacer combinaciones aún más extrañas.

```
1 | var datos=[]; // Se crea una matriz vacía de otra forma distinta.
2 | datos[-124]="valor"; // Es una posición asociativa.
3 | datos[1.000]="Contenido"; //No es asociativo ya que redondea a la posición 1. No es asociativo o disperso el índice.
4 | datos["tres"]=""; // Es asociativo
5 | console.log(datos);
```

Por supuesto no es recomendable mezclarlas, pero se pueden utilizar.

1.1.2.- Recorrido de un array.

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



[Kabsik Park](#)

```
1 | var sistemaSolar = new Array();
2 | sistemaSolar[0] = "Mercurio";
3 | sistemaSolar[1] = "Venus";
4 | sistemaSolar[2] = "Tierra";
5 | sistemaSolar[3] = "Marte";
6 | sistemaSolar[4] = "Jupiter";
7 | sistemaSolar[5] = "Saturno";
8 | sistemaSolar[6] = "Urano";
9 | sistemaSolar[7] = "Neptuno";
```

Empleando un bucle `for`, por ejemplo:

```
1 | for (i=0;i<sistemaSolar.length;i++)
2 |     document.write(sistemaSolar[i] + "<br/>");
```

Empleando un bucle `while`, por ejemplo:

```
1 | var i=0;
2 | while (i < sistemaSolar.length)
3 | {
4 |     document.write(sistemaSolar[i] + "<br/>");
5 |     i++;
6 | }
```

Empleando la sentencia `for in`

```
1 | for (var i in objeto)
2 |     sentencias
```

Esta sentencia repite la variable indicada, sobre todos los valores de las propiedades del objeto. Dentro de las sentencias `i` contiene el índice con el que acceder.

Es realmente útil ya que nos permite conocer las propiedades y métodos de cualquier objeto de JavaScript.

Ejemplo:

```
1 | for (var i in screen)
2 |     document.write("Propiedad " + i + " y valor " + screen[i] + "<br />");
```

Citas para pensar

"Es curioso, que podamos predecir con siglos de antelación el recorrido de las estrellas más lejanas, y en cambio, no seamos capaces de saber cómo soplará mañana el viento en nuestro pequeño planeta." *SPOERL, Heinrich.*

Autoevaluación

Si accedemos a una posición de un array que no está definida, obtendremos el valor:

- ☐ Error en tiempo de ejecución y se detiene la aplicación.
- ☐ Undefined.
- ☐ No devuelve ningún valor.

No es correcta, ya que la aplicación sigue funcionando sin problemas.

Correcto. Muy bien, ya que esa posición no está definida y ese es el mensaje que devolverá el intérprete de JavaScript al intentar acceder a esa posición.

Incorrecto, ya que devuelve un mensaje al acceder a esa posición.

Solución

1. Incorrecto
2. Opción correcta
3. Incorrecto

1.1.3.- 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** "".



[Daniel Novia](#)

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

Al borrar un elemento del array, se eliminará su índice en la lista de índices del array, pero no se reducirá la longitud del array. Por ejemplo en las siguientes instrucciones:

```
1 | elarray.length;           // resultado: 8
2 | delete elarray[5];
3 | elarray.length;           // resultado: 8
4 | elarray[5];                // resultado: undefined
```

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 se quiere 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 te 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**:

```
1 | var oceanos = new array("Atlantico","Pacifico","Artico","Indico");
```

Esta clase de array asigna automáticamente índices numéricos a sus entradas, para poder acceder posteriormente a los datos, como por ejemplo con un bucle:

```
1 | for (var i=0; i < oceanos.length; i++)
2 | {
3 |     if (oceanos[i] == "Atlantico")
4 |     {
5 |         // instrucciones a realizar..
6 |     }
7 | }
```

Si ejecutamos la instrucción:

```
1 | delete oceanos[2];
```

Se producirán los siguientes cambios: en primer lugar el tercer elemento del array ("Artico"), será eliminado del mismo, pero la longitud del array seguirá siendo la misma, y el array quedará tal y como:

```
1 | oceanos[0] = "Atlantico";
2 | oceanos[1] = "Pacifico";
3 | oceanos[3] = "Indico";
```

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:

```
1 | oceanos.splice(2,1); // las posiciones del array resultante serán 0, 1 y 2.
```


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.

1.1.4.- Propiedades y métodos.

Vamos a ver a continuación, las propiedades y métodos que podemos usar con cualquier array que utilicemos en Javascript.



[Les Chatfield](#)

Propiedades del objeto array:

Propiedades del objeto Array

Propiedad	Descripción
constructor	Devuelve la función que creó el prototipo del objeto array.
length	Ajusta o devuelve el número de elementos en un array.
prototype	Te permite añadir propiedades y métodos a un objeto.

Métodos del objeto array:

Métodos del objeto Array

Métodos	Descripción
concat()	Une dos o más arrays, y devuelve una copia de los arrays unidos.
join()	Une todos los elementos de un array en una cadena de texto.
pop()	Elimina el último elemento de un array y devuelve ese elemento.
push()	Añade nuevos elementos al final de un array, y devuelve la nueva longitud.
reverse()	Invierte el orden de los elementos en un array.
shift()	Elimina el primer elemento de un array, y devuelve ese elemento.
slice()	Selecciona una parte de un array y devuelve el nuevo array.
sort()	Ordena los elementos de un array.
splice()	Añade/elimina elementos a un array.
toString()	Convierte un array a una cadena y devuelve el resultado.
unshift()	Añade nuevos elementos al comienzo de un array, y devuelve la nueva longitud.
valueOf()	Devuelve el valor primitivo de un array.

Ejemplo de algunos métodos del objeto array:

Método reverse():

```
1 <script>
2   var frutas = ["Plátano", "Naranja", "Manzana", "Melocotón"];
3   document.write(frutas.reverse());           // Imprimirá: Melocotón,Manzana,Naranja,Plátano
4 </script>
```

Método slice():

```
1 <script>
2   var frutas = ["Plátano", "Naranja", "Manzana", "Melocotón"];
3   document.write(frutas.slice(0,1) + "<br />"); // imprimirá: Plátano
4   document.write(frutas.slice(1) + "<br />");   // imprimirá: Naranja,Manzana,Melocotón
5   document.write(frutas.slice(-2) + "<br />");  // imprimirá: Manzana, Melocotón
6
```

```
7 | document.write(frutas + "<br />"); // imprimirá: Plátano,Naranja,Manzana,Melocotón
   </script>
```

Métodos `push()` y `pop()`

```
1 // Con la estructura array se pueden implementar pilas (o incluso colas con el uso de shift y unshift)
2 // Las pilas son arrays LIFO donde se añaden elementos a continuación de la máxima posición del array.
3 <script>
4   var miPila=["elemento 1","elemento 2"];
5   var valor1=prompt("Dime un dato");
6   var valor2=prompt("Dime otro dato");
7   miPila.push(valor1);
8   miPila.push(valor2);
9   console.log("La pila después de añadir"+miPila);
10  // Ahora se elimina el elemento que está en la posición final
11  var cabecera=miPila.pop();
12  console.log("Se saca la cabecera "+cabecera); // Se puede ver que se guarda la cabecera y ya se puede hacer algo con ese dato.
13  console.log("La pila después de eliminar"+miPila);
14 </script>
```

Debes conocer

[Más información y ejemplos sobre el objeto array.](#)

1.2.- Arrays paralelos.

El usar arrays para almacenar información, facilita una gran versatilidad en las aplicaciones, a la hora de realizar búsquedas de elementos dentro del array. Pero en algunos casos, podría ser muy útil hacer las búsquedas en varios arrays a la vez, de tal forma que podamos almacenar diferentes tipos de información, en arrays que estén sincronizados.

Cuando tenemos dos o más arrays, que utilizan el mismo índice para referirse a términos homólogos, se denominan **arrays paralelos**.

Por ejemplo consideremos los siguientes arrays:

```
1 | var profesores = ["Cristina", "Catalina", "Vieites", "Benjamin"];
2 | var asignaturas=["Seguridad", "Bases de Datos", "Sistemas Informáticos", "Redes"];
3 | var alumnos=[24,17,28,26];
```



[Dominic Alves](#)

Usando estos tres arrays de forma sincronizada, y haciendo referencia a un mismo índice (por ejemplo índice=3), podríamos saber que Benjamín es profesor de Redes y tiene 26 alumnos en clase.

Veamos un ejemplo de código que recorrería todos los profesores que tengamos en el array imprimiendo la información sobre la asignatura que imparten y cuantos alumnos tienen en clase:

```
1 | <script>
2 |   var profesores = ["Cristina", "Catalina", "Vieites", "Benjamin"];
3 |   var asignaturas=["Seguridad", "Bases de Datos", "Sistemas
4 |   Informáticos", "Redes"];
5 |   var alumnos=[24,17,28,26];
6 |   function imprimeDatos(indice)
7 |   {
8 |       document.write("<br/>" + profesores[indice] + " del módulo de " + asignaturas[indice] + ", tiene " + alumnos[indice] + " alumnos en clase.");
9 |   }
10 |   for (i=0; i<profesores.length; i++)
11 |   {
12 |       imprimeDatos(i);
13 |   }
14 | </script>
```

Éste será el resultado que obtendremos de la aplicación anterior:

```
1 | Cristina del módulo de Seguridad, tiene 24 alumnos en clase.
2 | Catalina del módulo de Bases de Datos, tiene 17 alumnos en clase.
3 | Vieites del módulo de Sistemas Informáticos, tiene 28 alumnos en clase.
4 | Benjamin del módulo de Redes, tiene 26 alumnos en clase.
```

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, en algunos casos evitando complicaciones de sincronización.
- ✓ El recorrido secuencial de cada posición del array, es extremadamente rápido en las máquinas actuales.

Para saber más

[Ejemplo de arrays paralelos.](#)

1.3.- 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 en sus posiciones contengan otros arrays u otros objetos. Podemos crear de esta forma *arrays bidimensionales*, *tridimensionales*, etc.



[Dan Zen](#)

Por ejemplo podemos realizar el ejemplo anterior creando un **array bidimensional** de la siguiente forma:

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

ó bien usando una definición más breve y literal del array:

```
1 | var datos = [
2 |     ["Cristina", "Seguridad", 24],
3 |     ["Catalina", "Bases de Datos", 17],
4 |     ["Vieites", "Sistemas Informáticos", 28],
5 |     ["Benjamin", "Redes", 26]
6 | ];
```

Para acceder a un dato en particular, de un array de arrays, se requiere que hagamos una doble referencia. La primera referencia, será a una posición del array principal, y la segunda referencia, a una posición del array almacenado dentro de esa casilla del array principal. Esto se hará escribiendo el nombre del array, y entre corchetes cada una de las referencias: `nombrearray[indice1][indice2][indice3]` (nos permitiría acceder a una posición determinada en un array tridimensional).

Por ejemplo:

```
1 | document.write("<br/>Quien imparte Bases de Datos? "+datos[1][0]);           // Catalina
2 | document.write("<br/>Asignatura de Vieites: "+datos[2][1]);                 // Sistemas Informaticos
3 | document.write("<br/>Alumnos de Benjamin: "+datos[3][2]);                 // 26
```

Si queremos imprimir toda la información del array multidimensional, tal y como hicimos en el apartado anterior podríamos hacerlo con un bucle `for`:

```
1 | for (i=0;i<datos.length;i++)
2 | {
3 |     document.write("<br/>" + datos[i][0] + " del módulo de " + datos[i][1] + ", tiene " + datos[i][2] + " alumnos en clase.");
4 | }
```

Obtendríamos como resultado:

```
1 | Cristina del módulo de Seguridad, tiene 24 alumnos en clase.
2 | Catalina del módulo de Bases de Datos, tiene 17 alumnos en clase.
3 | Vieites del módulo de Sistemas Informáticos, tiene 28 alumnos en clase.
4 | Benjamin del módulo de Redes, tiene 26 alumnos en clase.
```

Para crear un array multidimensional 10x10 e inicializarlo podríamos utilizar el siguiente código:

```
1  var datos = new Array(10);
2  for ( i = 0; i < datos.length; i++)
3  {
4      datos[i]=new Array(10);
5      for (j = 0; j < datos[i].length ; j++)
6      {
7          datos[i][j]=j;
8          document.write(datos[i][j]+" ");
9      }
10     document.write("<br />");
11 }
```

Se puede asignar un objeto en la línea `datos[i][j]=new Objeto();` y de esa forma crear un array con objetos. En el punto 3 se verá como crear objetos en JavaScript.

También podríamos imprimir todos los datos de los arrays dentro de una tabla:

```
1  document.write("<table border=1>");
2  for (i = 0; i< datos.length; i++)
3  {
4      document.write("<tr>");
5      for (j = 0; j< datos[i].length; j++)
6      {
7          document.write("<td>"+datos[i][j]+"</td>");
8      }
9      document.write("</tr>");
10 }
11 document.write("</table>")
```

2.- Creación de funciones.

Caso práctico



Elaboración Propia.

Juan está siguiendo los progresos de **Antonio**, y le recomienda empezar a ver funciones. Las funciones son una herramienta muy potente en los lenguajes de programación, ya que le van a permitir realizar tareas de una manera mucho más organizada, y además le permitirán reutilizar un montón de código en sus aplicaciones.

Antonio verá como crear funciones, cómo pasar parámetros, el ámbito de las variables dentro de las funciones, cómo anidar funciones y las funciones predefinidas en JavaScript. **Antonio** está muy ilusionado con este tema, ya que a lo largo de las unidades anteriores ha estado utilizando funciones y es ahora el momento en el que realmente verá toda la potencia que le pueden aportar a sus aplicaciones.

En unidades anteriores ya has visto y utilizado alguna vez funciones. **Una función es la definición de un conjunto de acciones pre-programadas.** Las funciones se llaman a través de eventos o bien mediante comandos desde 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 valores). **En JavaScript siempre se llamarán funciones.**

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.

Te mostraré algunos ejemplos en un momento, pero antes de nada, vamos a ver la sintaxis formal de una función:

```
1 | function nombreFunción ( [parámetro1]...[parámetroN] )
2 | {
3 |     // instrucciones
4 | }
```

Si nuestra función va a **devolver algún valor** emplearemos la palabra reservada **return**, para hacerlo. Ejemplo:

```
1 | function nombreFunción ( [parámetro1]...[parámetroN] )
2 | {
3 |     // instrucciones
4 |     return valor;
5 | }
```

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*, y fíjate que las funciones suelen llevar un verbo, puesto que las funciones son elementos que realizan acciones.

Una recomendación que te hacemos, es la de que las funciones sean muy específicas, es decir que no realicen tareas adicionales a las inicialmente propuestas en esa función.

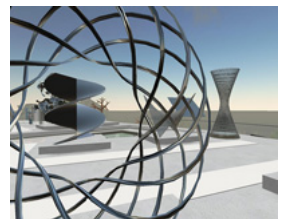
Para realizar una llamada a una función lo podemos hacer con:

```
1 | nombreFuncion( ); // Esta llamada ejecutaría las instrucciones programadas dentro de la función.
```

Otro ejemplo de uso de una función en una asignación:

```
1 | variable=nombreFuncion( ); // En este caso la función devolvería un valor que se asigna a la variable.
```

Las funciones en JavaScript también son objetos, y como tal tienen métodos y propiedades. Un método, aplicable a cualquier función puede ser `toString()`, el cuál nos devolverá el código fuente de esa función.



[NMC Virtual Worlds](#)

2.1.- Parámetros.

Cuando se realiza una llamada a una función, muchas veces es necesario pasar parámetros (también conocidos como argumentos). Este mecanismo nos va a permitir enviar datos entre instrucciones.

Para pasar parámetros a una función, tendremos que escribir dichos parámetros entre paréntesis y separados por comas.

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.

Veamos el siguiente ejemplo:



[Matteo Piatto](#)

```
1 | function saludar(a,b)
2 | {
3 |     alert("Hola " + a + " y " + b + ".");
4 | }
```

Si llamamos a esa función desde el código:

```
1 | saludar("Martin","Silvia"); //Mostraría una alerta con el texto: Hola Martin y Silvia.
```

Los parámetros que usamos en la definición de la función a y b, no usan la palabra reservada `var` para inicializar dichas variables. Esos parámetros a y b serán variables locales a la función, y se inicializarán automáticamente en el momento de llamar a la función, con los valores que le pasemos en la llamada. En el siguiente apartado entraremos más en profundidad en lo que son las variables locales y globales.

Otro ejemplo de función que devuelve un valor:

```
1 | function devolverMayor(a,b)
2 | {
3 |     var devolver;
4 |     if (a > b)
5 |         devolver=a;
6 |     else
7 |         devolver=b;
8 |
9 |     return (devolver);
10 |
11 | }
```

Ejemplo de utilización de la función anterior:

```
1 | document.write ("El número mayor entre 35 y 21 es el: " + devolverMayor(35,21) + ".");
```


Citas para pensar

"La inteligencia es la función que adapta los medios a los fines."

HARTMANN, N.

Para saber más

[Más información sobre funciones.](#)

[Ejemplo de funciones con parámetros.](#)

2.2.- Ámbito de las variables.

Ha llegado la hora de distinguir entre las variables que se definen fuera de una función, y las que se definen dentro de las funciones.

Las variables que se definen fuera de las funciones se llaman **variables globales**. Las **variables** que se definen dentro de las funciones, con la palabra reservada `var`, se llaman variables locales.

Handwritten mathematical formulas illustrating variable scope. It shows $C_{TOTAL} = C_D + A \cdot C_D$, then $C_D = 4 \cdot C_D$, and finally $C_{TOTAL} = (4 \cdot C_D) + A \cdot C_D$. Annotations include 'CONSTANTE' for C_D and A , and 'VARIABLE' for C_{TOTAL} . A small box highlights the '4' in the second equation.

[Leonardo Parada](#)

Una **variable global** en JavaScript tiene una connotación un poco diferente, comparando con 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. Sin embargo cuando inicializas una variable como variable global, quiere decir que todas las instrucciones de tu 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, o bien poner esa variable en el documento frameset, 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. En el modo estricto de JavaScript, es obligatorio. Siempre es una buena práctica: **declarar las variables con `var`**.

En contraste a las variables globales, una **variable local será definida dentro de una función**. Antes viste que podemos definir variables en los parámetros de una función (sin usar `var`), pero también podrás definir nuevas variables dentro del código de la función. En este caso, **si que se requiere el uso de la palabra reservada `var` cuando definimos una variable local**, ya que de otro modo, esta variable será reconocida como una variable global.

El alcance de una variable local está solamente dentro del ámbito de la función. Ninguna otra función o instrucciones fuera de la función podrán acceder al valor de esa 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 tampoco declares una variable global dentro de una función, ya que podrás crear fallos que te resultarán difíciles de solucionar.

Ejemplo de variables locales y globales:

```
1  // Uso de variables locales y globales no muy recomendable, ya que estamos empleando el mismo nombre de variable en global y en local.
2  var chica = "Aurora";           // variable global
3  var perros = "Lucky, Samba y Ronda"; // variable global
4  function demo()
5  {
6      // Definimos una variable local (fíjate que es obligatorio para las variables locales usar var)
7      // Esta variable local tendrá el mismo nombre que otra variable global pero con distinto contenido.
8      // Si no usáramos var estaríamos modificando la variable global chica.
9      var chica = "Raquel";        // variable local
10     document.write( "<br/>" + perros + " no pertenecen a " + chica + ".");
11 }
12 // Llamamos a la función para que use las variables locales.
13 demo();
14 // Utilizamos las variables globales definidas al comienzo.
15 document.write( "<br/>" + perros + " pertenecen a " + chica + ".");
```

Como resultado obtenemos:

```
1 | Lucky, Samba y Ronda no pertenecen a Raquel.
2 | Lucky, Samba y Ronda pertenecen a Aurora.
```

Para saber más

En unos años el modo estricto será el más utilizado. Pero de momento no lo vamos a usar. De cualquier forma es interesante que lo busquéis.

[Modo estricto.](#)

2.3.- 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.

Cuando no tenemos funciones anidadas, cada función que definamos será accesible por todo el código, es decir serán funciones globales. 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 principal. Tampoco te recomiendo el reutilizar nombres de funciones con esta técnica, para evitar problemas o confusiones posteriores.

La estructura de las funciones anidadas será algo así:



[timlewisnm](#)

```
1 function principalA()  
2 {  
3   // instrucciones  
4   function internaA1()  
5   {  
6     // instrucciones  
7   }  
8   // instrucciones  
9 }  
10 function principalB()  
11 {  
12   // instrucciones  
13   function internaB1()  
14   {  
15     // instrucciones  
16   }  
17   function internaB2()  
18   {  
19     // instrucciones  
20   }  
21   // instrucciones  
22 }
```

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.

Ejemplo de una función anidada:

```
1 function hipotenusa(a, b)  
2 {  
3   function cuadrado(x)  
4   {  
5     return x*x;  
6   }  
7   return Math.sqrt(cuadrado(a) + cuadrado(b));  
8 }  
9 document.write("<br/>La hipotenusa de 1 y 2 es: "+hipotenusa(1,2));  
10 // Imprimirá: La hipotenusa de 1 y 2 es: 2.23606797749979
```

Citas para pensar

"La función última de la crítica es que satisfaga la función natural de desdeñar, lo que conviene a la buena higiene del espíritu."
de PESSOA, Fernando.

2.4.- Funciones predefinidas del lenguaje.

Has visto en unidades anteriores un montón de objetos con sus propiedades y métodos. Si te das cuenta todos los métodos en realidad son funciones (llevan siempre paréntesis con o sin parámetros). 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).



[Austin King](#)

Propiedades globales en JavaScript:

Propiedades globales en JavaScript

Propiedad	Descripción
Infinity	Un valor numérico que representa el infinito positivo/negativo.
NaN	Valor que no es numérico "Not a Number".
undefined()	Indica que a esa variable no le ha sido asignado un valor.

Te mostraremos aquí una lista de funciones predefinidas, que se pueden utilizar a nivel global en cualquier parte de tu código de JavaScript. Estas funciones no están asociadas a ningún objeto en particular. Típicamente, estas funciones te permiten convertir datos de un tipo a otro tipo.

Funciones globales o predefinidas en JavaScript:

Funciones globales en JavaScript.

Función	Descripción
decodeURI()	Decodifica los caracteres especiales de una <u>URL</u> excepto: , / ? : @ & = + \$ #
decodeURIComponent()	Decodifica todos los caracteres especiales de una URL.
encodeURI()	Codifica los caracteres especiales de una URL excepto: , / ? : @ & = + \$ #
encodeURIComponent()	Codifica todos los caracteres especiales de una URL.
escape()	Codifica caracteres especiales en una cadena, excepto: * @ - _ + . /
eval()	Evalúa una cadena y la ejecuta si contiene código u operaciones.
isFinite()	Determina si un valor es un número finito válido.
isNaN()	Determina cuando un valor no es un número.
Number()	Convierte el valor de un objeto a un número.
parseFloat()	Convierte una cadena a un número real.
parseInt()	Convierte una cadena a un entero.
unescape()	Decodifica caracteres especiales en una cadena, excepto: * @ - _ + . /

Ejemplo de la **función eval()**:

```
1 <script type="text/javascript">
2   eval("x=50;y=30;document.write(x*y)"); // Imprime 1500
3   document.write("<br />" + eval("8+6")); // Imprime 14
4   document.write("<br />" + eval(x+30)); // Imprime 80
5 </script>
```

Debes conocer

El siguiente enlace amplía información sobre las funciones predefinidas en JavaScript.

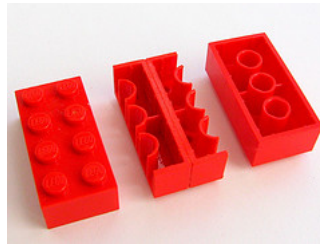
[Más información y ejemplos sobre propiedades y funciones predefinidas en JavaScript.](#)

3.- Creación de objetos a medida.

Caso práctico

Antonio ha hecho una pausa para tomar café, y charla con **Juan** de todo lo que ha aprendido sobre las funciones. Comenta lo interesante que ha sido, pero le surge una duda, ya que cuando estudió los objetos en JavaScript, vio que tenían propiedades y métodos, y los métodos tienen el mismo formato que las funciones que acaba de estudiar. **Juan** le dice que efectivamente eso es así, porque en realidad los métodos son funciones que se asignan a un objeto determinado, y que a parte de los objetos que ya ha estudiado en unidades anteriores, en JavaScript podrá crear sus propios objetos, con los métodos (funciones) y propiedades que él quiera.

Terminan su café y los dos vuelven al despacho para buscar documentación sobre cómo crear nuevos objetos en JavaScript.



[Windell Oskay](#)

En unidades anteriores has visto, como toda la información que proviene del navegador o del documento está organizada en un modelo de objetos, con propiedades y métodos. Pues bien, JavaScript también te da la oportunidad de crear tus propios objetos en memoria, objetos con propiedades y métodos que tú puedes definir a tu antojo. 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 si que se podrán mostrar en el navegador. El definir tus propios objetos, te permitirá enlazar a cualquier número de propiedades o métodos que tú hayas creado para ese objeto. Es decir, 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 de verdad en sentido estricto**. Se considera que, JavaScript **es un lenguaje basado en objetos**. La diferencia entre orientado a objetos y basado en objetos es significativa, y tiene que ver sobre todo en cómo los objetos se pueden extender. Quizás en un futuro no muy lejano podamos ver que JavaScript soporte clases, interfaces, herencia, etc.

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 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:

```
1 function Coche( )
2 {
3     // propiedades y métodos
4 }
```

Aunque esta función no contiene código, es sin embargo la base para crear objetos de tipo Coche. Puedes pensar en un constructor como un anteproyecto 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`, 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 quieren decir lo mismo.

Para saber más

[Más información sobre creación de Objetos de forma clásica \(la nueva la vemos en el punto 4\).](#)



3.1.- Definición de propiedades.

Una vez que ya sabemos como 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`. Véase el siguiente ejemplo:



[Fotos GOVBA](#)

```
1 function Coche( )
2 {
3     // Propiedades
4     this.marca = "Audi A6";
5     this.combustible = "diesel";
6     this.cantidad = 0;           // Cantidad de combustible en el depósito.
7 }
```

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 con el ejemplo anterior es, que todos los coches que hagamos del tipo `Coche` será siempre Audi A6, diésel, y sin litros de combustible en el depósito. Por ejemplo;

```
1 var cocheDeMartin = new Coche( );
2 var cocheDeSilvia = new Coche( );
```

A partir de ahora, si no modificamos las propiedades del coche de Martin y de Silvia, en el momento de instanciarlos tendrán ambos un Audi A6 a diésel y sin combustible en el depósito.

Lo ideal sería por lo tanto que en el momento de instanciar un objeto de tipo `Coche`, que le digamos al menos la marca de coche y el tipo de combustible que utiliza. Para ello tenemos que modificar el constructor, que quedará de la siguiente forma:

```
1 function Coche(marca, combustible)
2 {
3     // Propiedades
4     this.marca = marca;
5     this.combustible = combustible;
6     this.cantidad = 0;           // Cantidad de combustible inicial por defecto en el depósito.
7 }
```

Ahora sí que podríamos crear dos tipos diferentes de coche:

```
1 var cocheDeMartin = new Coche("Volkswagen Golf", "gasolina");
2 var cocheDeSilvia = new Coche("Mercedes SLK", "diesel");
```

Y también podemos acceder a las propiedades de esos objetos, consultar sus valores o modificarlos. Por ejemplo:

```
1 document.write("<br>El coche de Martin es un: "+cocheDeMartin.marca+" a "+cocheDeMartin.combustible);
2 document.write("<br>El coche de Silvia es un: "+cocheDeSilvia.marca+" a "+cocheDeSilvia.combustible);
3 // Imprimirá:
4 // El coche de Martin es un: Volkswagen Golf a gasolina
5 // El coche de Silvia es un: Mercedes SLK a diesel
6 // Ahora modificamos la marca y el combustible del coche de Martin:
7 cocheDeMartin.marca = "BMW X5";
8 cocheDeMartin.combustible = "diesel";
9 document.write("<br>El coche de Martin es un: " + cocheDeMartin.marca + " a " + cocheDeMartin.combustible);
10 // Imprimirá: El coche de Martin es un: BMW X5 a diesel
```


3.2.- 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, que serán funciones que se enlazarán a los objetos**, para que dichas funciones puedan acceder a las propiedades de los mismos.



[Josh Smith](#)

Nos definimos un ejemplo de método, que se podría utilizar en la clase Coche:

```
1 function rellenarDeposito (litros)
2 {
3     // Modificamos el valor de la propiedad cantidad de combustible
4     this.cantidad = litros;
5 }
```

Fíjate que el método `rellenarDeposito`, que estamos programando a nivel global, hace referencia a la propiedad `this.cantidad` para indicar cuantos litros de combustible le vamos a echar al coche. Lo único que faltaría aquí es realizar la conexión entre el método `rellenarDeposito` y el objeto de tipo `Coche` (recuerda que los objetos podrán tener propiedades y métodos y hasta este momento sólo hemos definido propiedades dentro del constructor). Sin esta conexión la palabra reservada `this` no tiene sentido en esa función, ya que no sabría cuál es el objeto actual. Veamos cómo realizar la conexión de ese método, con el objeto dentro del constructor:

```
1 function Coche(marca, combustible)
2 {
3     // Propiedades
4     this.marca = marca;
5     this.combustible = combustible;
6     this.cantidad = 0;           // Cantidad de combustible inicial por defecto en el depósito.
7     // Métodos
8     this.rellenarDeposito = rellenarDeposito;
9 }
```

Aquí se ve de forma ilustrada, que los métodos son en realidad propiedades: se declaran igual que las propiedades, por lo que son enmascarados como propiedades en JavaScript. Hemos creado una nueva propiedad llamada `rellenarDeposito` y se le ha asociado el método `rellenarDeposito`. Es muy importante destacar que el método `rellenarDeposito()` se referencia sin paréntesis dentro del constructor, `this.rellenarDeposito = rellenarDeposito`.

Ejemplo de uso del método anterior:

```
1 cocheDeMartin.rellenarDeposito(35);
2 document.write("<br>El coche de Martin tiene "+cocheDeMartin.cantidad+ " litros de " + cocheDeMartin.combustible+ " en el depósito.");
3 // Imprimirá
4 // El coche de Martin tiene 35 litros de diesel en el depósito.
```

La forma en la que hemos definido el método `rellenarDeposito` a nivel global, no es la mejor práctica en la programación orientada a objetos. Una mejor aproximación sería definir el contenido de la función `rellenarDeposito` dentro del constructor, ya que de esta forma los métodos al estar programados a nivel local aportan mayor privacidad y seguridad al objeto en general, por ejemplo:

```
1 function Coche(marca, combustible)
2 {
3     // Propiedades
4     this.marca = marca;
5     this.combustible = combustible;
6     this.cantidad = 0;
7     // Métodos
8     this.rellenarDeposito = function (litros)
9     {
10         this.cantidad=litros;
11     };
12 }
```

JavaScript es muy flexible y podríamos añadir más propiedades y métodos después de la definición. Aunque esto no es algo recomendado en el código va a servir para comprender la necesidad de **prototype**.

Lo primero que haremos será crear un nuevo coche y después le añadiremos el método que es llamado desde **document.write** para poder escribir los contenidos de un objeto: **toString**.

```
1 | var micoche= new Coche("Honda", "120");
2 |     document.write(micoche); // Sale Object.
3 |     micoche.toString = function () {
4 |         return ( this.marca + " " + this.combustible + " " + this.cantidad );
5 |     }
6 | document.write(micoche); // Sale el contenido.
```

Ahora bien, se presenta otro problema, para cada objeto que creamos hay que añadir esta función. De hecho cada objeto usa memoria para cada función repitiéndose una y otra vez. Para solucionarlo se puede utilizar el objeto **prototype**.

El objeto **prototype** se encuentra en cualquier objeto de JavaScript y con él se puede añadir métodos nuevos a cualquier objeto e incluso implementar herencia.

Veamos un ejemplo de uso en el que se añade el método **toString**.

```
1 |     micoche= new Coche("Honda", "120");
2 |     document.write(micoche); // Sale Object.
3 |     Coche.prototype.toString = function () {
4 |         return ( this.marca + " " + this.combustible + " " + this.cantidad );
5 |     }
6 |     document.write(micoche); // Sale el contenido.
```

Para saber más

[La herencia en JavaScript se realiza a través del objeto prototype.](#)

3.3.- Definición de objetos literales.

Otra forma de definir objetos es hacerlo de forma literal.

Un literal es un valor fijo que se especifica en JavaScript. Un objeto literal será un conjunto, de cero o más parejas del tipo **nombre:valor**.



[planegezeer](#)

Por ejemplo:

```
1 | avion = { marca:"Boeing" , modelo:"747" , pasajeros:"450" };
```

Es equivalente a:

```
1 | var avion = new Object();
2 | avion.marca = "Boeing";
3 | avion.modelo = "747";
4 | avion.pasajeros = "450";
```

Para referirnos desde JavaScript a una propiedad del objeto avión podríamos hacerlo con:

```
1 | document.write(avion.marca); // o también se podría hacer con:
2 | document.write(avion["modelo"]);
```

Podríamos tener un conjunto de objetos literales simplemente creando un array que contenga en cada posición una definición de objeto literal:

```
1 | var datos=[
2 |   {"id":"2","nombrecentro":"IES A Piringalla" , "localidad":"Lugo", "provincia":"Lugo"},
3 |   {"id":"10","nombrecentro":"IES As Fontiñas", "localidad":"Santiago", "provincia":"A Coruña"},
4 |   {"id":"9","nombrecentro":"IES As Lagoas", "localidad":"Ourense", "provincia":"Ourense"},
5 |   {"id":"8","nombrecentro":"IES Cruceiro Baleares", "localidad":"Culleredo", "provincia":"A
6 |   Coruña"},
7 |   {"id":"6","nombrecentro":"IES Cruceiro Baleares", "localidad":"Culleredo", "provincia":"A
8 |   Coruña"}, {"id":"4","nombrecentro":"IES de Teis", "localidad":"Vigo", "provincia":"Pontevedra"},
9 |   {"id":"5","nombrecentro":"IES Leliadoura", "localidad":"Ribeira", "provincia":"A Coruña"},
10 |  {"id":"7","nombrecentro":"IES Leliadoura", "localidad":"Ribeira", "provincia":"A Coruña"},
11 |  {"id":"1","nombrecentro":"IES Ramon Aller
12 |  Ulloa", "localidad":"Lalin", "provincia":"Pontevedra"},
13 |  {"id":"3","nombrecentro":"IES San Clemente", "localidad":"Santiago de
14 |  Compostela", "provincia":"A Coruña"}
15 | ];
```

De la siguiente forma se podría recorrer el array de datos para mostrar su contenido:

```
1 | for (var i=0; i< datos.length; i++)
2 | {
3 |     document.write("Centro ID: "+datos[i].id+ " - ");
4 |     document.write("Nombre: "+datos[i].nombrecentro+ " - ");
5 |     document.write("Localidad: "+datos[i].localidad+ " - ");
6 |     document.write("Provincia: "+datos[i].provincia+"<br/>");
7 | }
```

Y obtendríamos como resultado:

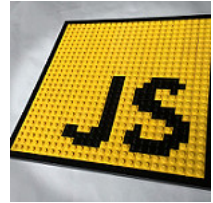
1	Centro ID: 2 - Nombre: IES A Piringalla - Localidad: Lugo - Provincia: Lugo
2	Centro ID: 10 - Nombre: IES As Fontiñas - Localidad: Santiago - Provincia: A Coruña
3	Centro ID: 9 - Nombre: IES As Lagoas - Localidad: Ourense - Provincia: Ourense
4	Centro ID: 8 - Nombre: IES Cruceiro Baleares - Localidad: Culleredo - Provincia: A Coruña
5	Centro ID: 6 - Nombre: IES Cruceiro Baleares - Localidad: Culleredo - Provincia: A Coruña
6	Centro ID: 4 - Nombre: IES de Teis - Localidad: Vigo - Provincia: Pontevedra
7	Centro ID: 5 - Nombre: IES Leliadoura - Localidad: Ribeira - Provincia: A Coruña
8	Centro ID: 7 - Nombre: IES Leliadoura - Localidad: Ribeira - Provincia: A Coruña
9	Centro ID: 1 - Nombre: IES Ramon Aller Ulloa - Localidad: Lalin - Provincia: Pontevedra
10	Centro ID: 3 - Nombre: IES San Clemente - Localidad: Santiago de Compostela - Provincia: A Coruña

4.- EcmaScript 6.

Caso práctico

Antonio ha visto por internet que el Lenguaje JavaScript está evolucionando constantemente y añadiendo funcionalidad para adaptar el lenguaje de programación a los tiempos.

Juan se había adelantado y viendo que IE ya no se utiliza casi nada ha llegado el momento de utilizar las ventajas, ateniéndose a lo que funciona en los navegadores de los últimos años. Antonio le advierte a Juan que puede no funcionar todo en todos los sitios así que deciden investigar más profundamente.



[Christopher Hiller, JavaScript Logo](#)
(CC BY-NC-SA)

Lo primero es decir que todo lo visto hasta ahora en esta unidad se sigue utilizando. Sin embargo, las empresas también están utilizando las nuevas tecnologías añadidas a Java y hay que conocerlas.

Pero antes de entrar a explicar centrémonos un poco en la historia. Originalmente había distintas implementaciones del lenguaje JavaScript. Incluso tenía distintos nombres; Microsoft lo llamaba `jscript`. A partir de 1996 se le encargó a ECMA la estandarización del lenguaje y ya lleva realizados cambios en 9 versiones a fecha de diciembre de 2018. La mayoría de las versiones han añadido características pero nunca tan revolucionarias para el programador con la versión 6.

Otros lenguajes de programación se han derivado de este estándar; el más famoso es `actionScript` usado en Adobe Flash.

Lo que vamos a tratar de la versión 6 está implementado y bastante estable en los navegadores Firefox, Chrome y Edge. Y si no funciona podemos utilizar [babel.js](#), un framework que veremos en la última unidad (por ahora no te metas).

Para saber más

[Web de ECMA Internacional.](#)

4.1- Clases.

Hasta la implementación completa de EcmaScript 6 los navegadores habían tenido diversos problemas a la hora de realizar una aplicación web con clases.

Algunas cosas no funcionaban en Firefox, otras en Chrome, otras en Edge y ninguna en Internet Explorer. Ahora si que son usables.

En el siguiente código vemos lo sencillo y similar que es al Java el modelo de clases.

```
1  class clase {  
2      metodo() {  
3          // .....  
4      }  
5      toString() { // Este método devuelve una propiedad.  
6          return ("El valor de la propiedad es" + this.propiedad);  
7      }  
8  }  
9  
10 // A la hora de crear el objeto vamos a tener el mismo funcionamiento que en las formas anteriores  
11 miclase = new Clase();  
12 document.write(miclase)
```

Si por algún motivo quieres utilizar estas tecnologías en navegadores antiguos, te recomiendo que uses babeljs (se encuentra explicado en la unidad 6 del curso). Así podrás utilizar todas las ventajas de las versiones ES6 y superiores sin tener que perder la compatibilidad.

No existen ni métodos privados, ni podemos realizar sobrecarga. Si que se permite la herencia y se añaden constructores y un intento de encapsulamiento que veremos en los siguientes puntos.

Para saber más

[Como saber si un navegador soporta clases ES6.](#)

4.1.1.- Constructor.



[OpenClipart-Vectors, Albaril \(CC0\)](#)

En esta versión de JavaScript ya podemos disponer de un constructor definido. En las anteriores versiones de los objetos escribíamos dentro de la función de la cual colgaban los métodos el código para la asignación inicial de variables, un tanto limitado pero funcional.

A partir de ES6 se soluciona permitiendo incluir todo el código de inicialización en un método llamado **constructor**.

```
1 | class serVivo {
2 |
3 |     this.peso = peso;
4 | }
5 | toString() {
6 |     return "El peso es " + this.peso + "<br>";
7 | }
8 | }
9 |
10 | // A la hora de crear el objeto vamos a tener el mismo funcionamiento que en las formas anteriores.
11 | bicho = new serVivo(1);
12 | document.write(bicho); // forma correcta de mostrar el objeto con toString().
```

También podemos usar el objeto **arguments** para pasar un número no determinado de argumentos. Veámoslo con un ejemplo:

```
1 | class listaElementos {
2 |     constructor(lista) {
3 |         this.miArray=new Array();
4 |
5 |         this.miArray.push(arguments[i]); // Se añaden los argumentos al final
6 |     }
7 | }
8 | toString() {
9 |     let devolver="";
10 |    for (let i=0;i<this.miArray.length;i++) {
11 |        devolver+=this.miArray[i];
12 |    }
13 |    return (devolver);
14 | }
15 | }
16 |
17 | coleccion= new listaElementos("perro","gato","canario");
18 | document.write (coleccion); // forma correcta de mostrar el objeto con toString().
```

Este objeto ya existía anteriormente a ES6, pero ahora se puede utilizar con más sentido, ya que las clases tienen pinta de clases y con este objeto podemos simular una sobrecarga de métodos.

4.1.2.- Modelo de Setters y Getters.



Abinoam Jr., [Pills](#) (CC BY-SA)

ES6 añade la capacidad de crear getters y setters usando las palabras clave `get` y `set` antes del método. Este método será la forma de acceder a la propiedad, creando una especie de encapsulamiento limitado. Es limitado ya que si se conoce el nombre de la propiedad puede ser accedido. Recordemos que JavaScript no dispone de clases privadas ni métodos privados.

Si queremos que se acceda a una propiedad con este tipo de setters o getters tenemos que darle un nombre distinto a la propiedad a la que nos vamos a referir. Por ejemplo si queremos que se acceda al peso de un objeto debemos realizar un `set` y un `get` con `peso`, y la propiedad original se llamará de otra forma. Una buena práctica para identificarlas es ponerle un guión bajo delante. Así pues en este caso al peso lo vamos a llamar `_peso`. Por supuesto hay otras formas un tanto arcanas que podréis consultar en el para saber más.

```
1  class serVivo {
2      constructor(peso) {
3          this._peso = peso;
4      }
5
6      return this._peso;
7  }
8
9      if (nuevoPeso >= 0)
10         this._peso = nuevoPeso;
11     else this.peso = 0; // no puede haber peso inferior a 0.
12 }
13 toString() {
14     return "El peso es " + this.peso + "<br>";
15 }
16 }
17
18 var bicho = new serVivo(12);
19 bicho.peso = 33; // Es realmente un setPeso(33)
```

Como se puede comprobar aunque es accesible `bicho._peso`, es mucho mejor usar `bicho.peso` ya que te permite controlar que no se asigne un peso inferior a 0.

Para saber más

[La guía definitiva para trabajar con setters y getters.](#)

4.1.3.- Herencia con extends.

La palabra reservada de Java es la misma que en JavaScript: `extends`. Por dentro, el lenguaje sigue siendo gestionado por `prototype`, pero es más sencillo para un programador de Java y similares. Nos permite aislarnos de las complejidades que puede causar utilizar `prototype` y su sistema de creación de objetos con herencia.

Al igual que en Java podremos acceder a los métodos del padre usando `super`.

En el siguiente ejemplo podremos ver el uso de la herencia con `super` heredando de la clase `serVivo`:

```
2   constructor(peso, altura) {
3       super(peso); // El padre asigna el peso.
4       this._altura = altura; // La clase hija asigna la altura.
5   }
6   get altura() {
7       return this._altura;
8   }
9   set altura(nuevaAltura) {
10      this._altura = nuevaAltura;
11  }
12  toString() {
13      let devolver = super.toString();
14      devolver = devolver + "La altura es " + this._altura + "<br>";
15      return (devolver);
16  }
17
18  }
19  var planta = new vegetal(13, 12);
20  document.write(planta);
```



Pedro Vélez (1665). [Testamento](#)
[La Gaceta](#) (Dominio público)

Para saber más

[Prototype vs ES6](#)

4.1.4.- Métodos estáticos.

La palabra clave `static` nos permite crear un método accesible aunque la clase no haya sido instanciada.

Una de las clases que te van a servir para entender este sistema es `Math` ya nos permite usar su métodos sin crear ninguna instancia. Por ejemplo `Math.pow(2,2)`.

`static` se escribe justo antes de la declaración del método. Veamos un ejemplo.

```
1 | class Mates {  
3 |     return(lado*lado);  
4 | }  
5 | static areaRectángulo (base, altura) {  
  
7 | }  
8 | }  
9 |  
10 | console.log(Mates.areaCuadrado(2));
```

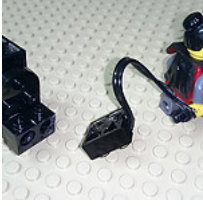
Para terminar `static` no se puede utilizar en variables directamente. Aunque podemos utilizarla con `get` en los métodos.

```
1 | static get PI() {  
2 |     return 3.14;  
3 | }
```

Para saber más

[Uso de static en ES6.](#)

4.2.- Módulos.



[O Falcão de LEGO, Dojo04 \(CC BY-NC-ND\)](#)

ES6 ha creado nuevos tipos de forma de declarar variables.

const variable=valor. Asigna un valor a una variable y **no permite reasignarlo**. Se suele utilizar para cuando lees un valor de teclado y quieres asegurarte de que no sea modificado. El ámbito es el del bloque.

let variable. Crear una variable pero teniendo el ámbito de esa variable el bloque donde se ha creado. No pasa como con **var** que lo creamos en un bucle y se puede utilizar en cualquier parte produciendo el fenómeno llamado hoisting que nos permite utilizar una variable antes de ser declarada.

Además de lo anterior, ES6 añade más funcionalidades referidas al ámbito de las variables, funciones y clases: los módulos. Son lo más parecido a los paquetes de Java de lo que podemos disponer en JavaScript.

Los módulos son útiles como medio de aislar un archivo de código de otro. Imaginar dos desarrolladores que realizan funcionalidades distintas pero que por casualidad usan una misma variable, o método o clase. A la hora de usarlo como hasta ahora había un problema y la última de las definiciones realizadas era la que quedaba como utilizable. Los módulos en ES6 nos van a permitir separar este código y exportar con el nombre que necesitemos la funcionalidad que queremos que sea visible desde otros sitios. El resto permanecerá oculto y no dará problemas.

El siguiente código importa una funcionalidad a tu archivo principal html.

```
2 | // Se importa desde el archivo html que usa la funcionalidad.
4 |     comprobarDNI
5 | } from 'momodulo.js';
6 |     funcionPublica("Haz algo");
7 | </script>
```

Este segundo trozo de código es el archivo que se va a importar.

```
2 | export function funcionPublica(valor) {
3 |     let devolver=funcionPrivada(valor); // Aquí compruebas que la letra es correcta.
4 |     return (devolver);
5 | }
6 | // La siguiente función no puede ser llamada desde el archivo que la importa ya que no tiene import.
7 | function funcionaPrivada(valor) {
8 |     let devolver; // variable para devolver.
9 |     // Aquí el calculo
10 |    // .....
11 |    return (devolver);
12 | }
```

Así como puede verse, en vez de usar `<script src="fuente.js">` , se utiliza **import** para añadir la funcionalidad. Eso si, no te olvides de poner en el archivo que importa los módulos `<script type="module">` . Si no lo haces tendrás muchos problemas.

Podemos exportar tanto variables, como funciones o clases.

Si quisiésemos importar varias funciones o clases deberíamos separarlas por comas.

```
1 | import { area, perimetro } from 'circulo';
```

Con **import** se puede cambiar el nombre a una función evitando de esa forma colisiones del módulo con funciones del tuyo.

```
1 | import {
3 |     } from 'miOtroModulo.js';
```

Los módulos deben tener extensión **js**.

Para saber más

[var, let, const. Diferencias.](#)

[Para saber más sobre como trabajar con módulos.](#)

4.3.- Otras novedades.

ES6 añadió un nuevo tipo de bucle: `for of`.

A diferencia del `for in` permite obtener el contenido de un array , cadena, mapa o conjunto sin tener el índice del mismo.

```
1 | miArray=new Array("Casa","Coche","Piscina");
2 | for (contenido of miArray) {
3 |     document.write ( contenido + " ");
4 | }
5 | // Al final podrás ver en la pantalla "Casa Coche Piscina".
```

Esto se puede extender a los objetos personales utilizando iteradores, iterables y generadores, pero que aquí no vamos a ver porque su uso es limitado y son bastante complejos para un curso como éste. Además la filosofía tiende a ser de un lenguaje funcional y no de un lenguaje con objetos como el que estamos viendo en este módulo.

Los objetos `set` (conjuntos) y `map` (mapas) fueron introducidos en la versión ES5, pero han dado el salto de usabilidad en la versión 6.



Otra de las novedades de ES6 que tampoco vamos a tratar son las funciones flecha (arrow también conocidas como lambdas en Java y otros lenguajes). Sin meternos en complicaciones, son funciones declaradas de esta forma curiosa: `argumentos => { sentencias }` Veamos un ejemplo sencillo para realizar el cuadrado de un número.

```
1 | var cuadrado = (x) => { return x*x; };
2 | document.write("El cuadrado de 2 es:" + cuadrado(2));
```

[DrTrigonBot, Flecha \(CC BY-SA\)](#)

Las funciones flecha se deriva de los lenguajes funcionales. Pero últimamente se están añadiendo a todos los lenguajes orientados a objetos (incluso C++).

Seguro que las habéis visto alguna vez en Internet y aunque son útiles, no es necesario conocerlas para programar en JavaScript siendo realmente útiles en el lenguaje derivado TypeScript (pero eso es otra cuestión).

Para saber más

[Mapas.](#)

[Conjuntos.](#)

[Iteradores, iterables y generadores.](#)

[Funciones Flecha.](#)







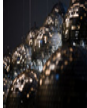

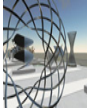


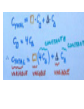






5.- Enlaces de refuerzo y ampliación.

Para saber más

- ✓ Arrays en JavaScript :
 - ✦ [Arrays](#)
- ✓ Metodos de Array en JavaScript:
 - ✦ [Métodos del array.](#)
 - ✦ [Más de Arrays.](#)
- ✓ JavaScript Functions:
 - ✦ http://www.w3schools.com/js/js_functions.asp
- ✓ Detalles del modo estricto:
 - ✦ [Modo estricto.](#)
- ✓ JSON: JavaScript Object Notation.:
 - ✦ [Introducción a la notación JSON.](#)
- ✓ Ámbito de las funciones (Funciones anidadas):
 - ✦ [Closures.](#)
- ✓ Propiedades y funciones globales JavaScript:
 - ✦ [Más Funciones.](#)

Anexo.- Licencia de recursos.

Licencias de recursos utilizados en la Unidad de Trabajo.

Recurso (1)	Datos del recurso (1)	Recurso (2)	Datos
	Autoría: Ricardo Luque. Licencia: CC BY-SA 2.0. Procedencia: http://www.flickr.com/photos/23235581@N02/4639535768/sizes/z/in/photostream/		Autoría: Aaron Gustafson. Licencia: CC BY-SA 2.0. Procedencia: http://www.flickr.com/photos/aarongi
	Autoría: Repoort. Licencia: CC BY-2.0. Procedencia: http://www.flickr.com/photos/repoort/2645497916/		Autoría: Kabsik Park. Licencia: CC BY-2.0. Procedencia: images/145118335/sizes/z/in/photos
	Autoría: Daniel Novta. Licencia: CC BY-2.0. Procedencia: http://www.flickr.com/photos/vanf/5006945413/sizes/z/in/photostream/		Autoría: Elsie esq. Licencia: CC BY 2.0. Procedencia: http://www.flickr.com/photos/elsie/52
	Autoría: Dominic Alves. Licencia: CC BY 2.0. Procedencia: http://www.flickr.com/photos/dominicspics/4625651369/sizes/z/in/photostream/		Autoría: Dan Zen. Licencia: CC BY 2.0. Procedencia: http://www.flickr.com/photos/danzen
	Autoría: NMC Virtual Worlds. Licencia: CC BY 2.0. Procedencia: http://www.flickr.com/photos/nmcvirtualworlds/353867999/sizes/z/in/photostream/		Autoría: Matteo Piotto. Licencia: CC BY-SA 2.0. Procedencia: http://www.flickr.com/photos/namuit/
	Autoría: Rafael Veiga Cid. Licencia: CC BY-NC-SA 2.0. Procedencia: Elaboración Propia.		Autoría: leonardoparada. Licencia: CC BY 2.0. Procedencia: http://www.flickr.com/photos/leonard
	Autoría: timlewisnm. Licencia: CC BY-SA 2.0. Procedencia: http://www.flickr.com/photos/gozalewis/3249937230/sizes/z/in/photostream/		Autoría: Austin King. Licencia: CC BY-SA 2.0. Procedencia: http://www.flickr.com/photos/ozten/4
	Autoría: Windell Oskay Licencia: CC BY 2.0. Procedencia: https://www.flickr.com/photos/oskay/265899784/sizes/sq/		Autoría: Fotos GOVBA. Licencia: CC BY 2.0. Procedencia: http://www.flickr.com/photos/agecon
	Autoría: Josh Smith. Licencia: CC BY-SA 2.0. Procedencia: http://www.flickr.com/photos/joshsmith/28687097/sizes/z/in/photostream/		Autoría: planegeezer. Licencia: CC BY 2.0. Procedencia: http://www.flickr.com/photos/102615