¿Qué es un array? **Fuente: lenguajejs.com // Mozilla Developer**

Un array es una colección o agrupación de elementos en una misma variable, cada uno de ellos ubicado por la posición que ocupa en el array. En Javascript, se pueden definir de varias formas:

Constructor	Descripción
new Array(len)	Crea un array de **len** elementos .
new Array(e1, e2...)	Crea un array con ninguno o varios elementos.
[e1, e2...]	Simplemente, los elementos dentro de corchetes: **[]**. **Notación preferida**.

Por ejemplo, podríamos tener un array que en su primera posición tenemos el 'a', en la segunda el 'b' y en la tercera el 'c'. En Javascript, esto se crearía de esta forma:

```
// Forma tradicional
const array = new Array("a", "b", "c");

// Mediante literales (preferida)
const array = ["a", "b", "c"]; // Array con 3 elementos
const empty = []; // Array vacío (0 elementos)
const mixto = ["a", 5, true]; // Array mixto (string, number, boolean)
```

Al contrario que muchos otros lenguajes de programación, Javascript permite que se puedan realizar arrays de **tipo mixto**, no siendo obligatorio que todos los elementos sean del mismo tipo de dato (*en el ejemplo anterior,*).

> **OJO**: Al crear un array con **new Array(num)**, si solo indicamos un parámetro y **num** es un número, Javascript creará un array de **num** elementos sin definir. Es decir, **a = new Array(3)** sería equivalente a **a = [undefined, undefined, undefined]**. Esto no ocurre con su equivalente, **a = [3]**, donde estamos creando un array con un único elemento: **3**.

Acceso a elementos

Al igual que las Strings , saber el número elementos que tiene un array es muy sencillo. Sólo hay que acceder a la propiedad **.length**, que nos devolverá el número de elementos existentes en un array:

Método	Descripción
.length	Devuelve el número de elementos del array.
[pos]	Operador que devuelve el elemento número **pos** del array.

Por otro lado, si lo que queremos es acceder a un elemento específico del array, no hay más que utilizar el operador **[]**, al igual que hacemos con las Strings para acceder a un carácter concreto. En este caso, accedemos a la posición del elemento que queremos recuperar sobre el array:

```
const array = ["a", "b", "c"];

array[0]; // 'a'
array[2]; // 'c'
array[5]; // undefined
```

Recuerda que las posiciones empiezan a contar desde **0** y que si intentamos acceder a una posición que no existe (*mayor del tamaño del array*), nos devolverá un undefined.

Añadir o eliminar elementos

Existen varias formas de añadir elementos a un array existente. Veamos los métodos que podemos usar para ello:

Método	Descripción
.push(obj1, obj2...)	Añade uno o varios elementos al final del array. Devuelve tamaño del array.
.pop()	Elimina y devuelve el último elemento del array.
.unshift(obj1, obj2...)	Añade uno o varios elementos al inicio del array. Devuelve tamaño del array.
.shift()	Elimina y devuelve el primer elemento del array.

.concat(obj1, obj2...)	Concatena los elementos (o elementos de los arrays) pasados por parámetro.

En los arrays, Javascript proporciona métodos tanto para insertar o eliminar elementos **por el final** del array: **push()** y **pop()**, como para insertar o eliminar elementos **por el principio** del array: **unshift()** y **shift()**. Salvo por esto, funcionan exactamente igual.

El método de inserción, **push()** o **unshift()** inserta los elementos pasados por parámetro en el array y devuelve el tamaño actual que tiene el array después de la inserción. Por otro lado, los métodos de extracción, **pop()** o **shift()**, extraen y devuelven el elemento.

```
const array = ["a", "b", "c"]; // Array inicial

array.push("d"); // Devuelve 4. Ahora array = ['a', 'b', 'c', 'd']
array.pop(); // Devuelve 'd'. Ahora array = ['a', 'b', 'c']

array.unshift("Z"); // Devuelve 4. Ahora array = ['Z', 'a', 'b', 'c']
array.shift(); // Devuelve 'Z'. Ahora array = ['a', 'b', 'c']
```

Además, al igual que en las Strings tenemos el método **concat()**, que nos permite concatenar los elementos pasados por parámetro en un array. Se podría pensar que los métodos **.push()** y **concat()** funcionan de la misma forma, pero no es exactamente así. Veamos un ejemplo:

```
const array = [1, 2, 3];
array.push(4, 5, 6); // Devuelve 6. Ahora array = [1, 2, 3, 4, 5, 6]
array.push([7, 8, 9]); // Devuelve 7. Ahora array = [1, 2, 3, 4, 5, 6, [7, 8, 9]]

const array = [1, 2, 3];
array = array.concat(4, 5, 6); // Devuelve 6. Ahora array = [1, 2, 3, 4, 5, 6]
array = array.concat([7, 8, 9]); // Devuelve 9. Ahora array = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Observa un detalle muy importante. El método **concat()**, a diferencia de **push()**, no modifica el array sobre el cuál trabajamos y al que le añadimos los elementos, sino que simplemente lo devuelve. Al margen de esto, observa que en el caso de pasar un array por parámetro, **push()** lo inserta como un array, mientras que **concat()** inserta cada uno de sus elementos.

> También hay que tener cuidado al utilizar el operador **+** con los arrays. A diferencia de lo que quizás puede parecer intuitivo, utilizando este operador no se añaden los elementos al array, sino que se convierten los arrays en string y luego se concatenan. Veremos más sobre estas **conversiones implícitas** en temas posteriores.

Creación de arrays

Existen métodos para crear pequeños arrays derivados de otras variables u objetos. Es el caso de los métodos **slice()** y **splice()**. Luego, también hablaremos del método **join()** y el método estático **Array.from()**:

Método	Descripción
.slice(ini, end)	Devuelve los elementos desde posición **ini** hasta **end** (excluído).
.splice(ini, num)	Elimina y devuelve **num** elementos desde posición **ini**.
.splice(ini, num, o1, o2...)	Idem. Además inserta **o1**, **o2**... en la posición **ini**.
.join(sep)	Une los elementos del array por **sep** en un .
Array.from(o, f, thisVal)	Crea un array a partir de **o** (algo similar a un array).

El método **slice()** devuelve los elementos del array desde la posición **ini** hasta la posición **end**, permitiendo crear un nuevo array más pequeño con ese grupo de elementos. Recuerda que las posiciones empiezan a contar desde **0**. En el caso de que no se proporcione el parámetro **end**, se devuelven todos los elementos desde la posición **ini** hasta el final del array.

Por otro lado, ten en cuenta que el array sobre el que realizamos el método **slice()** no sufre ninguna modificación, sólo se devuelve por parámetro el array creado. Diferente es el caso del método **splice()**, el cuál realiza algo parecido a **slice()** pero con una gran diferencia: **modifica el array original**. En el método **splice()** el segundo parámetro **num** no es la posición final del subarray, sino el tamaño del array final, es decir, el número de elementos que se van a obtener desde la posición **ini**.

Por lo tanto, con el método **splice()**, devolvemos un array con los elementos desde la posición **ini** hasta la posición **ini**+**num**. El array original es modificado, ya que se eliminan los elementos desde la posición **ini** hasta la posición **ini**+**num**. Es posible también indicar una serie de parámetros opcionales después de los mencionados, que permitirán además de la extracción de elementos, **insertar dichos elementos** justo donde hicimos la extracción.

Veamos un ejemplo ilustrativo:

```
const array = ["a", "b", "c", "d", "e"];

// .slice() no modifica el array
array.slice(2, 4); // Devuelve ['c', 'd']. El array no se modifica.

// .splice() si modifica el array
array.splice(2, 2); // Devuelve ['c', 'd']. Ahora array = ['a', 'b', 'e']
array.splice(1, 0, "z", "x"); // Devuelve []. Ahora array = ['a', 'z', 'x', 'b', 'e']
```

> En ciertos casos, nos podría interesar reducir el tamaño de un array para quedarnos con sus primeros elementos y descartar el resto. Hay una forma muy sencilla y eficiente que es modificar directamente el tamaño del array mediante **.length**.

Por ejemplo, hacer un **a.length = 4** en un array de 8 elementos, reducirá el array a los primeros 4 elementos de una forma eficiente, ya que no crea un nuevo array, sino que reduce el tamaño del actual y descarta el resto de elementos.

Además, también tenemos otro método con el que es posible crear un a partir de un . Se trata del método **split()** que vimos en el tema de los . En este caso, el método **join()** es su contrapartida. Con **join()** podemos crear un con todos los elementos del array, separándolo por el texto que le pasemos por parámetro:

```
const array = ["a", "b", "c"];

array.join("->"); // Devuelve 'a->b->c'
array.join("."); // Devuelve 'a.b.c'
"a.b.c".split("."); // Devuelve ['a', 'b', 'c']
"5.4.3.2.1".split("."); // Devuelve ['5', '4', '3', '2', '1']
```

Ten en cuenta que, como se puede ver en el último ejemplo, **split()** siempre devolverá los elementos como .

Por último, mencionar también el método estático **Array.from()**. Aunque ahora no le encontraremos mucha utilidad, nos resultará muy interesante más adelante. Este método se suele utilizar para convertir variables «parecidas» a los arrays (*pero que no son arrays*) en arrays reales.

```
Array.from("hola"); // ['h', 'o', 'l', 'a']
```

Búsqueda y comprobación

Existen varios métodos para realizar ciertas comprobaciones con arrays:

Método	Descripción
Array.isArray(obj)	Comprueba si **obj** es un array. Devuelve **true** o **false**.
.includes(obj, from)	Comprueba si **obj** es uno de los elementos incluidos en el array.
.indexOf(obj, from)	Devuelve la posición de la primera aparición de **obj** desde **from**.
.lastIndexOf(obj, from)	Devuelve la posición de la última aparición de **obj** desde **from**.

El primero de ellos, **Array.isArray(obj)** se utiliza para comprobar si **obj** es un array o no, devolviendo un booleano. Los otros tres métodos funcionan exactamente igual que sus equivalentes en las Strings. El método **includes()** comprueba si el elemento **obj** pasado por parámetro es uno de los elementos que incluye el array, partiendo desde la posición **from**. Si se omite **from**, se parte desde **0**.

```
const arreglo = [5, 10, 15, 20, 25];

Array.isArray(arreglo); // true
arreglo.includes(10); // true
arreglo.includes(10, 2); // false
```

```
arreglo.indexOf(25); // 4
arreglo.lastIndexOf(10, 0); // -1
```

Por otro lado, tenemos **indexOf()** y **lastIndexOf()** dos funciones que se utilizan para devolver la posición del elemento **obj** pasado por parámetro, empezando a buscar en la posición **from** (*o **0** si se omite*). El primer método, devuelve la primera aparición, mientras que el segundo método devuelve la última aparición.

En el último caso, **lastIndexOf()**, busca el número 10 de derecha a izquierda comenzando en la posición 0, lo cual haría que nunca encuentre el número 10, devolviendo -1.

Modificación de arrays

Es posible que tengamos un array específico al que queremos hacer ciertas modificaciones donde **slice()** y **splice()** se quedan cortos (*o resulta más cómodo utilizar los siguientes métodos*). Existen algunos métodos introducidos en **ECMAScript 6** que nos permiten crear una versión modificada de un array, mediante métodos como **copyWithin()** o **fill()**:

Método	Descripción
.copyWithin(pos, ini, end)	Devuelve , copiando en **pos** los ítems desde **ini** a **end**.
.fill(obj, ini, end)	Devuelve un relleno de **obj** desde **ini** hasta **end**.

El primero de ellos, **copyWithin(pos, ini, end)** nos permite crear una copia del array que alteraremos de la siguiente forma: en la posición **pos** copiaremos los elementos del propio array que aparecen desde la posición **ini** hasta la posición **end**. Es decir, desde la posición **0** hasta **pos** será exactamente igual, y de ahí en adelante, será una copia de los valores de la posición **ini** a la posición **end**. Veamos algunos ejemplos:

```
const array = ["a", "b", "c", "d", "e", "f"];

// Estos métodos modifican el array original
array.copyWithin(5, 0, 1); // Devuelve ['a', 'b', 'c', 'd', 'e', 'a']
array.copyWithin(3, 0, 3); // Devuelve ['a', 'b', 'c', 'a', 'b', 'c']
array.fill("Z", 0, 5); // Devuelve ['Z', 'Z', 'Z', 'Z', 'Z', 'c']
```

Por otro lado, el método **fill(obj, ini, end)** es mucho más sencillo. Se encarga de devolver una versión del array, rellenando con el elemento **obj** desde la posición **ini** hasta la posición **end**.

Ordenaciones

En JavaScript, es muy habitual que tengamos arrays y queramos ordenar su contenido por diferentes criterios. En este apartado, vamos a ver los métodos **reverse()** y **sort()**, útiles para ordenar un array:

Método	Descripción
.reverse()	Invierte el orden de elementos del array.
.sort()	Ordena los elementos del array bajo un criterio de **ordenación alfabética.**
.sort(func)	Ordena los elementos del array bajo un criterio de ordenación **func**.

En primer lugar, el método **reverse()** cambia los elementos del array en orden inverso, es decir, si tenemos **[5, 4, 3]** lo modifica de modo que ahora tenemos **[3, 4, 5]**. Por otro lado, el método **sort()** realiza una ordenación (*por orden alfabético*) de los elementos del array:

```
const array = ["Alberto", "Ana", "Mauricio", "Bernardo", "Zoe"];

// Ojo, cada línea está modificando el array original
array.sort(); // ['Alberto', 'Ana', 'Bernardo', 'Mauricio', 'Zoe']
array.reverse(); // ['Zoe', 'Mauricio', 'Bernardo', 'Ana', 'Alberto']
```

Un detalle muy importante es que estos dos métodos **modifican el array original**, además de devolver el array modificado. Si no quieres que el array original cambie, asegúrate de crear primero una copia del array, para así realizar la ordenación sobre esa copia y no sobre el original.

Sin embargo, la ordenación anterior se realizó sobre Strings y todo fue bien. Veamos que ocurre si intentamos ordenar un array de números:

```
const array = [1, 8, 2, 32, 9, 7, 4];

array.sort(); // Devuelve [1, 2, 32, 4, 7, 8, 9], que NO es el resultado deseado
```

Esto ocurre porque, al igual que en el ejemplo anterior, el tipo de ordenación que realiza **sort()** por defecto es una ordenación alfabética, mientras que en esta ocasión buscamos una **ordenación natural**, que es la que se suele utilizar con números. Esto se puede hacer en JavaScript, pero requiere pasarle por parámetro al **sort()** lo que se llama una **función de comparación**.

Función de comparación

Como hemos visto, la ordenación que realiza **sort()** por defecto es siempre una ordenación alfabética. Sin embargo, podemos pasarle por parámetro lo que se conoce con los nombres de **función de ordenación** o **función de comparación**. Dicha función, lo que hace es establecer otro criterio de ordenación, en lugar del que tiene por defecto:

```
const array = [1, 8, 2, 32, 9, 7, 4];

// Función de comparación para ordenación natural
const fc = function (a, b) {
  return a - b;
};
```

```
array.sort(fc); // Devuelve [1, 2, 4, 7, 8, 9, 32], que SÍ es el resultado deseado
```

Como se puede ver en el ejemplo anterior, creando la función de ordenación **fc** y pasándola por parámetro a **sort()**, le indicamos cómo debe hacer la ordenación y ahora si la realiza correctamente.

¿Qué son las Array functions?

Básicamente, son métodos que tiene cualquier variable que sea de tipo Array y que permite realizar una operación con todos los elementos de dicho array para conseguir un objetivo concreto, dependiendo del método. En general, a dichos métodos se les pasa por parámetro una **función callback** y unos parámetros opcionales.

Estas son las **Array functions** que podemos encontrarnos en JavaScript:

Método	Descripción
.forEach(cb, arg)	Realiza la operación definida en **cb** por cada elemento del array.
.every(cb, arg)	Comprueba si **todos** los elementos del array cumplen la condición de **cb**.
.some(cb, arg)	Comprueba si **al menos** un elem. del array cumple la condición de **cb**.

.map(cb, arg)	Construye un array con lo que devuelve **cb** por cada elemento del array.
.filter(cb, arg)	Construye un array con los elementos que cumplen el filtro de **cb**.
.findIndex(cb, arg)	Devuelve la posición del elemento que cumple la condición de **cb**.
.find(cb, arg)	Devuelve el elemento que cumple la condición de **cb**.
.reduce(cb, arg)	Ejecuta **cb** con cada elemento (de izq a der), acumulando el resultado.
.reduceRight(cb, arg)	Idem al anterior, pero en orden de derecha a izquierda.

A grandes rasgos, a cada uno de estos métodos se les pasa una función **callback** que se ejecutará por cada uno de los elementos que contiene el array. Empecemos por **forEach()**, que es quizás el más sencillo de todos.

forEach (Cada uno)

Como se puede ver, el método **forEach()** no devuelve nada y espera que se le pase por parámetro una funcion que se ejecutará por cada elemento del array. Esa función, puede ser

pasada en cualquiera de los formatos que hemos visto: como función tradicional o como función flecha:

```
const arr = ["a", "b", "c", "d"];

// Con funciones por expresión
const f = function () {
  console.log("Un elemento.");
};
arr.forEach(f);

// Con funciones anónimas
arr.forEach(function () {
  console.log("Un elemento.");
});

// Con funciones flecha
arr.forEach(() => console.log("Un elemento."));
```

Sin embargo, este ejemplo no tiene demasiada utilidad. A la función **callback** se le pueden pasar varios parámetros opcionales:

- Si se le pasa un **primer parámetro**, este será el elemento del array.
- Si se le pasa un **segundo parámetro**, este será la posición en el array.
- Si se le pasa un **tercer parámetro**, este será el array en cuestión.

Veamos un ejemplo:

```
const arr = ["a", "b", "c", "d"];

arr.forEach((e) => console.log(e)); // Devuelve 'a' / 'b' / 'c' / 'd'
arr.forEach((e, i) => console.log(e, i)); // Devuelve 'a' 0 / 'b' 1 / 'c' 2 / 'd' 3
arr.forEach((e, i, a) => console.log(a[0])); // Devuelve 'a' / 'a' / 'a' / 'a'
```

En este ejemplo, he nombrado **e** al parámetro que hará referencia al **elemento**, **i** al parámetro que hará referencia al índice (*posición del array*) y **a** al parámetro que hará referencia al **array** en cuestión. Aún así, el usuario puede ponerles a estos parámetros el nombre que prefiera. Como se puede ver, realmente **forEach()** es otra forma de hacer un bucle (*sobre un array*), sin tener que recurrir a bucles tradicionales como **for** o **while**.

every (Todos)

El método **every()** permite comprobar si **todos y cada uno** de los elementos de un array cumplen la condición que se especifique en la **callback**:

```
const arr = ["a", "b", "c", "d"];
arr.every((e) => e.length == 1); // true
```

En este caso, la magia está en el **callback**. La condición es que la longitud de cada elemento del array sea **1**. Si dicha función devuelve **true**, significa que cumple la condición, si devuelve **false**, no la cumple. Por lo tanto, si todos los elementos del array devuelven **true**, entonces **every()** devolverá **true**.

Si expandimos el ejemplo anterior a un código más detallado, tendríamos el siguiente ejemplo equivalente, que quizás sea más comprensible para entenderlo:

```
const arr = ["a", "b", "c", "d"];

// Esta función se ejecuta por cada elemento del array
const todos = function (e) {
  // Si el tamaño del string es igual a 1
  if (e.length == 1) return true;
  else return false;
};

arr.every(todos); // Le pasamos la función callback todos() a every
```

some (Al menos uno)

De la misma forma que el método anterior sirve para comprobar si todos los elementos del array cumplen una determinada condición, con **some()** podemos comprobar si **al menos uno** de los elementos del array, cumplen dicha condición definida por el **callback**.

```
const arr = ["a", "bb", "c", "d"];
arr.some((e) => e.length == 2); // true
```

Observa que en este ejemplo, el método **some()** devuelve **true** porque existe al menos un elemento del array con una longitud de **2** carácteres.

map (Transformaciones)

El método **map()** es un método muy potente y útil para trabajar con arrays, puesto que su objetivo es devolver un nuevo array donde cada uno de sus elementos será lo que devuelva la función **callback** por cada uno de los elementos del array original:

```
const arr = ["Ana", "Pablo", "Pedro", "Pancracio", "Heriberto"];
const nuevoArr = arr.map((e) => e.length);

nuevoArr; // Devuelve [3, 5, 5, 9, 9]
```

Observa que el array devuelto por **map()** es **nuevoArr**, y cada uno de los elementos que lo componente, es el número devuelto por el **callback** (e.length), que no es otra cosa sino el tamaño de cada .

Este método nos permite hacer multitud de operaciones, ya que donde devolvemos **e.length** podriamos devolver el propio modificado o cualquier otra cosa.

filter (Filtrado)

El método **filter()** nos permite filtrar los elementos de un array y devolver un nuevo array con sólo los elementos que queramos. Para ello, utilizaremos la función **callback** para establecer una condición que devuelve **true** sólo en los elementos que nos interesen:

```
const arr = ["Ana", "Pablo", "Pedro", "Pancracio", "Heriberto"];
const nuevoArr = arr.filter((e) => e[0] == "P");

nuevoArr; // Devuelve ['Pablo', 'Pedro', 'Pancracio']
```

En este ejemplo, filtramos sólo los elementos en los que su primera letra sea **P**. Por lo tanto, la variable **nuevoArr** será un array con sólo esos elementos.

Ten en cuenta que si ningún elemento cumple la condición, **filter()** devuelve un vacío.

find (Búsqueda)

En **ECMAScript 6** se introducen dos nuevos métodos dentro de las **Array functions**: **find()** y **findIndex()**. Ambos se utilizan para buscar elementos de un array mediante una condición, la diferencia es que el primero devuelve el elemento mientras que el segundo devuelve su posición en el array original. Veamos como funcionan:

```
const arr = ["Ana", "Pablo", "Pedro", "Pancracio", "Heriberto"];

arr.find((e) => e.length == 5); // 'Pablo'
arr.findIndex((e) => e.length == 5); // 1
```

La condición que hemos utilizado en este ejemplo es buscar el elemento que tiene **5** carácteres de longitud. Al buscarlo en el array original, el primero que encontramos es **Pablo**, puesto que **find()** devolverá 'Pablo' y **findIndex()** devolverá **1**, que es la segunda posición del array donde se encuentra.

En el caso de no encontrar ningún elemento que cumpla la condición, **find()** devolverá , mientras que **findIndex()**, que debe devolver un , devolverá **-1**.

reduce (Acumuladores)

Por último, nos encontramos con una pareja de métodos denominados **reduce()** y **reduceRight()**. Ambos métodos se encargan de recorrer todos los elementos del array, e ir acumulando sus valores (*o alguna operación diferente*) y sumarlo todo, para devolver su resultado final.

En este par de métodos, encontraremos una primera diferencia en su función **callback**, puesto que en lugar de tener los clásicos parámetros opcionales **(e, i, a)** que hemos utilizado hasta ahora, tiene **(p, e, i, a)**, donde vemos que aparece un primer parámetro extra inicial: **p**.

En la primera iteración, **p** contiene el valor del primer elemento del array y **e** del segundo. En siguientes iteraciones, **p** es el acumulador que contiene lo que devolvió el **callback** en la iteración anterior, mientras que **e** es el siguiente elemento del array, y así sucesivamente. Veamos un ejemplo para entenderlo:

```
const arr = [95, 5, 25, 10, 25];
arr.reduce((p, e) => {
  console.log(`P=${p} e=${e}`);
  return p + e;
});

// P=95 e=5     (1ª iteración: elemento 1: 95 + elemento 2: 5) = 100
// P=100 e=25   (2ª iteración: 100 + elemento 3: 25) = 125
// P=125 e=10   (3ª iteración: 125 + elemento 4: 10) = 135
// P=135 e=25   (4ª iteración: 135 + elemento 5: 25) = 160

// Finalmente, devuelve 160
```

Gracias a esto, podemos utilizar el método **reduce()** como acumulador de elementos de izquierda a derecha y **reduceRight()** como acumulador de elementos de derecha a izquierda. Veamos un ejemplo de cada uno, realizando una resta en lugar de una suma:

```
const arr = [95, 5, 25, 10, 25];
arr.reduce((p, e) => p - e); // 95 - 5 - 25 - 10 - 25. Devuelve 30
arr.reduceRight((p, e) => p - e); // 25 - 10 - 25 - 5 - 95. Devuelve -110
```

localStorage y sessionStorage: ¿qué son?

El objeto Storage (API de almacenamiento web) nos permite almacenar datos de manera local en el navegador y sin necesidad de realizar alguna conexión a una base de datos.

localStorage y **sessionStorage** son propiedades que acceden al objeto Storage y tienen la función de almacenar datos de manera local, la diferencia entre éstas dos es que localStorage almacena la información de forma indefinida o hasta que se decida limpiar los datos del navegador y sessionStorage almacena información mientras la pestaña donde se esté utilizando siga abierta, una vez cerrada, la información se elimina.

Validar objeto Storage en el navegador

Aunque gran parte de los navegadores hoy en día son compatibles con el objeto Storage, no está de más hacer una pequeña validación para rectificar que realmente podemos utilizar dicho objeto, para ello utilizaremos el siguiente código:

```
if (typeof(Storage) !== 'undefined') {
  // Código cuando Storage es compatible
} else {
 // Código cuando Storage NO es compatible
}
```

Guardar datos en Storage

Existen dos formas de guardar datos en Storage, que son las siguientes:

localStorage

```
// Opción 1 ->  localStorage.setItem(name, content)
// Opción 2 ->localStorage.name = content
// name = nombre del elemento
// content = Contenido del elemento

localStorage.setItem('Nombre', 'Miguel Antonio')
localStorage.Apellido = 'Márquez Montoya'
```

sessionStorage

```
// Opción 1 ->  sessionStorage.setItem(name, content)
// Opción 2 ->sessionStorage.name = content
// name = nombre del elemento
// content = Contenido del elemento

sessionStorage.setItem('Nombre', 'Miguel Antonio')
sessionStorage.Apellido = 'Márquez Montoya'
```

Recuperar datos de Storage

Al igual que para agregar información, para recuperarla tenemos dos maneras de hacerlo

localStorage

```
// Opción 1 -> localStorage.getItem(name, content)
// Opción 2 -> localStorage.name

// Obtenemos los datos y los almacenamos en variables
let firstName = localStorage.getItem('Nombre'),
    lastName  = localStorage.Apellido

console.log(`Hola, mi nombre es ${firstName} ${lastName}`)
// Imprime: Hola, mi nombre es Miguel Antonio Márquez Montoya
```

sessionStorage

```
// Opción 1 -> sessionStorage.getItem(name, content)
// Opción 2 -> sessionStorage.name

// Obtenemos los datos y los almacenamos en variables
let firstName = sessionStorage.getItem('Nombre'),
    lastName  = sessionStorage.Apellido

console.log(`Hola, mi nombre es ${firstName} ${lastName}`)
// Imprime: Hola, mi nombre es Miguel Antonio Márquez Montoya
```

Eliminar datos de Storage

Para eliminar un elemento dentro de Storage haremos lo siguiente:

localStorage

```
// localStorage.removeItem(name)
localStorage.removeItem(Apellido)
```

sessionStorage

```
// sessionStorage.removeItem(name)
sessionStorage.removeItem(Apellido)
```

Limpiar todo el Storage

Ya para finalizar veremos la forma para eliminar todos los datos del Storage y dejarlo completamente limpio

localStorage

localStorage.clear()

sessionStorage

sessionStorage.clear()

Guardar y mostrar múltiples datos desde local/session Storage

La clave está en que **localStorage solo nos permite guardar un *string***. Así que primero debemos convertir nuestro objeto a string con *JSON.stringify()* como en el siguiente ejemplo:

```
let datos = ['html5','css3','js'];
// Guardo el objeto como un string
localStorage.setItem('datos', JSON.stringify(datos));
```

```
// Obtener el string previamente salvado
let datos = localStorage.getItem('datos');
console.log('Datos: ', JSON.parse(datos));
```

Las características de *Local Storage* y *Session Storage* son:

- Permiten almacenar entre 5MB y 10MB de información; incluyendo texto y multimedia
- La información está almacenada en la computadora del cliente y NO es enviada en cada petición del servidor, a diferencia de las cookies
- Utilizan un número mínimo de peticiones al servidor para reducir el tráfico de la red
- Previenen pérdidas de información cuando se desconecta de la red
- La información es guardada por domino web (incluye todas las páginas del dominio)