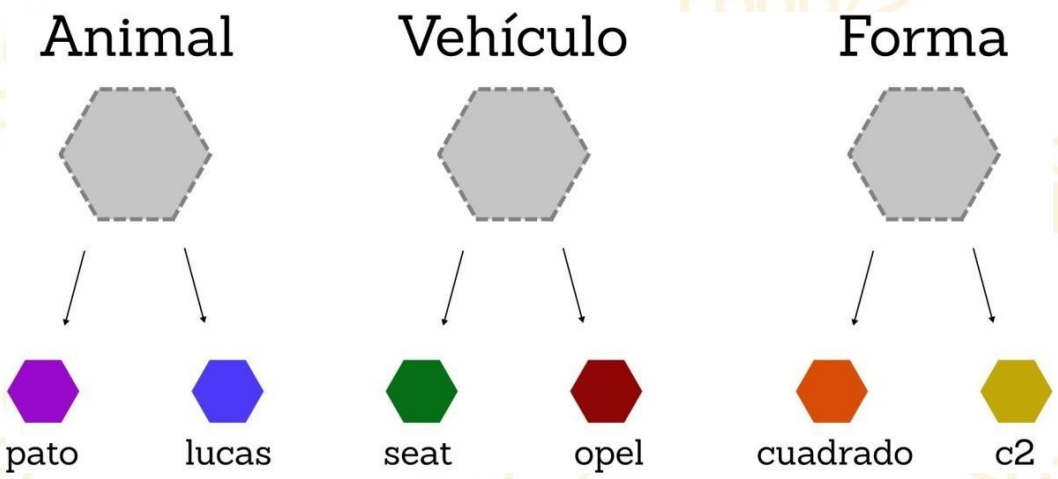
¿Qué es una clase?

Una **clase** es una forma de organizar código de forma entendible con el objetivo de simplificar el funcionamiento de nuestro programa. Además, hay que tener en cuenta quelas clases son «conceptos abstractos» de los que se pueden crear objetos de programación, cada uno con sus características concretas.

Esto puede ser complicado de entender con palabras, pero se ve muy claro con ejemplos:



En primer lugar, tenemos la **clase**. La clase es el **concepto abstracto** de un objeto, mientras que el **objeto** es el elemento final que se basa en la clase. En la imagen anteriortenemos varios ejemplos:

- En el **primer ejemplo** tenemos dos variables: **pato** y **lucas**. Ambos son animales,por lo que son objetos que están basados en la clase **Animal**. Tanto **pato** como **lucas** tienen las características que estarán definidas en la clase **Animal**: color, sonido que emiten, nombre, etc...
- En el **segundo ejemplo** tenemos dos variables **seat** y **opel**. Se trata de dos coches,que son vehículos, puesto que están basados en la clase **Vehículo**. Cada uno tendrá las características de su clase: color del vehículo, número de ruedas, marca, modelo, etc...
- En el **tercer ejemplo** tenemos dos variables **cuadrado** y **c2**. Se trata de dos formas geométricas, que al igual que los ejemplos anteriores tendrán sus propias características, como por ejemplo el tamaño de sus lados. El elemento **cuadrado** puede tener un lado de **3** cm y el elemento **c2** puede tener un lado de **6** cm.

En JavaScript se utiliza una sintaxis muy similar a otros lenguajes como, por ejemplo, Java.Declarar una clase es tan sencillo como escribir lo siguiente:

```
// Declaración de una claseclass Animal {}
```

```
// Crear o instanciar un objetoconst pato = new Animal();
```

El nombre elegido debería hacer referencia a la información que va a contener dicha clase. Piensa que el objetivo de las clases es almacenar en ella todo lo que tenga relación (*en esteejemplo, con los animales*).

Observa que luego creamos una variable donde hacemos un **new Animal()**. Estamos creando una variable **pato** (*un objeto*) que es de tipo **Animal**, y que contendrá todas lascaracterísticas definidas dentro de la clase **Animal** (*de momento, vacía*).

Una norma de estilo en el mundo de la programación es que las clases deben siempre empezar en mayúsculas. Esto nos ayud

Elementos de una clase

Una clase tiene diferentes características que la forman, vamos a ir explicándolas todas detalladamente. Pero primero, una tabla general para verlas en conjunto:

Elemento	Descripción

Propiedad	Variable que existe dentro de una clase. Puede ser pública o privada.
Propiedad pública	Propiedad a la que se puede acceder desde fuera de la clase.
Propiedad computada	Función para acceder a una propiedad con modificaciones (getter/setter).
Método	Función que existe dentro de una clase. Puede ser pública o privada.
Método público	Método que se puede ejecutar desde dentro y fuera de laclase.
Método estático	Método que se ejecuta directamente desde la clase, no desdela instancia.

Constructor	Método que se ejecuta automáticamente cuando se crea una instancia.

Como vemos, todas estas características se dividen en dos grupos: las **propiedades** (*a grandes rasgos, variables dentro de clases*) y los **métodos** (*a grandes rasgos, funciones dentro de clases*). Veamos cada una de ellas en detalle, pero empecemos por los **métodos**.

¿Qué es un método?

Hasta ahora habíamos visto que los **métodos** eran funciones que viven dentro de una variable, más concretamente de un objeto. Los objetos de tipo String tienen varios métodos,los objetos de tipo Number tienen otros métodos, etc... Justo eso es lo que definimos en el interior de una clase.

Si añadimos un método a la clase **Animal**, al crear cualquier variable haciendo un **new Animal()**, tendrá automáticamente ese método disponible. Ten en cuenta que podemoscrear varias variables de tipo **Animal** y serán totalmente independientes cada una:

```
// Declaración de claseclass Animal {

  // Métodos
```

```
  hablar() {
    return "Cuak";
  }
}

// Creación de una instancia u objetoconst pato = new Animal(); pato.hablar(); // 'Cuak'
const donald = new Animal();donald.hablar(); // 'Cuak'
```

Observa que el método **hablar()**, que se encuentra dentro de la clase **Animal**, existe en lasvariables **pato** y **donald** porque realmente son de tipo **Animal**. Al igual que con las funciones, se le pueden pasar varios parámetros al método y trabajar con ellos como venimos haciendo normalmente con las funciones.

¿Qué es un método estático?

En el caso anterior, para usar un método de una clase, como por ejemplo **hablar()**, debemos crear el objeto basado en la clase haciendo un **new** de la clase. Lo que se denomina crear un objeto o una instancia de la clase. En algunos casos, nos puede interesar crear **métodos estáticos** en una clase porque para utilizarlos no hace falta crearese objeto, sino que se pueden ejecutar directamente sobre la clase directamente:

```
class Animal {
  static despedirse() {
    return "Adiós";
```

```
  }

  hablar() {
    return "Cuak";
  }
}

Animal.despedirse(); // 'Adiós'
```

Como veremos más adelante, lo habitual suele ser utilizar métodos normales (*no estáticos*),porque normalmente nos suele interesar crear varios objetos y guardar información diferente en cada uno de ellos, y para eso tendríamos que instanciar un objeto.

Una de las limitaciones de los **métodos estáticos** es que en su interior sólo podremoshacer referencia a elementos que también sean estáticos. No podremos acceder a propiedades o métodos no estáticos, ya que necesitaríamos instanciar un objeto para hacerlo.

Los métodos estáticos se suelen utilizar para crear funciones de apoyo que realicen tareas concretas o genéricas, porque está

¿Qué es un constructor?

Se le llama **constructor** a un tipo especial de método de una clase, que se ejecuta automáticamente a la hora de hacer un **new** de dicha clase. Una clase **solo puede tener un constructor**, y en el caso de que no se especifique un constructor a una clase, tendráuno vacío de forma implícita.

```
// Declaración de claseclass Animal {
  // Método que se ejecuta al hacer un new
  constructor() {
    console.warn("Ha nacido un pato.");
  }
  // Métodos
  hablar() {
    return "Cuak";
  }
}

// Creación de una instancia u objeto
```

```
const pato = new Animal(); // 'Ha nacido un pato'
```

El **constructor** es un mecanismo muy interesante y utilizado para tareas de inicialización oque quieres realizar tras haber creado el nuevo objeto.

¿Qué es una propiedad?

Las clases, siendo estructuras para guardar información, pueden guardar variables con su correspondiente información. Dicho concepto se denomina **propiedades** y en Javascript serealiza en el interior del constructor, precedido de la palabra clave **this** (*que hace referenciaa «este» elemento, es decir, la clase*), como puedes ver en el siguiente ejemplo:

```
class Animal {
  constructor(n = "pato") {
    this.nombre = n;
  }

  hablar() {
    return "Cuak";
  }
```

```
  quienSoy() {
    return "Hola, soy " + this.nombre;
  }
}

// Creación de objetos const pato = new Animal();
pato.quienSoy(); // 'Hola, soy pato'

const donald = new Animal("Donald");pato.quienSoy(); // 'Hola, soy Donald'
```

Como se puede ver, estas **propiedades** existen en la clase, y se puede establecer de formaque todos los objetos tengan el mismo valor, o como en el ejemplo anterior, tengan valores diferentes dependiendo del objeto en cuestión, pasándole los valores específicos por parámetro.

Observa que, las propiedades de la clase podrán ser modificadas externamente, ya que pordefecto son **propiedades públicas**:

```
const pato = new Animal("Donald"); pato.quienSoy(); // 'Hola, soy Donald'

pato.nombre = "Paco"; pato.quienSoy(); // 'Hola, soy Paco'
```

Los ámbitos en una clase

Dentro de una clase tenemos dos tipos de ámbitos: **ámbito de método** y **ámbito de clase**:

En primer lugar, veamos el **ámbito dentro de un método**. Si declaramos variables o funciones dentro de un método con **var**, **let** o **const**, estos elementos existirán sólo en elmétodo en cuestión. Además, no serán accesibles desde fuera del método:

```
class Clase {
  constructor() {
    const name = "Manz";
    console.log("Constructor: " + name);
  }

  metodo() {
    console.log("Método: " + name);
  }
}

const c = new Clase(); // 'Constructor: Manz'
```

```
c.name; // undefined c.metodo(); // 'Método: '
```

Observa que la variable **name** solo se muestra cuando se hace referencia a ella dentro del **constructor()** que es donde se creó y donde existe.

En segundo lugar, tenemos el **ámbito de clase**. Podemos crear propiedades precedidas por **this.** (*desde dentro del constructor*) y desde **ES2020** desde la parte superior de la clase,lo que significa que estas propiedades tendrán alcance en toda la clase, tanto desde el constructor, como desde otros métodos del mismo:

```
class Clase {
  role = "Teacher”; // ES2020+

  constructor() {
    this.name = "Manz";
    console.log("Constructor: " + this.name);
  }

  metodo() {
    console.log("Método: " + this.name);
  }
```

```
}

const c = new Clase(); // 'Constructor: Manz'

c.name;                // 'Manz'   metodo                    // 'Teacher'
```

Ojo, estas propiedades también pueden ser modificadas desde fuera de la clase, simplemente asignándole otro valor. Si quieres evitarlo, añade el **#** antes del nombre de lapropiedad al declararla.

La palabra clave this

Como te habrás fijado en ejemplos anteriores, hemos introducido la palabra clave **this**, quehace referencia al **elemento padre** que la contiene. Así pues, si escribimos **this.nombre** dentro de un método, estaremos haciendo referencia a la propiedad **nombre** que existe dentro de ese objeto. De la misma forma, si escribimos **this.hablar()** estaremos ejecutandoel método **hablar()** de ese objeto.

Veamos el siguiente ejemplo, volviendo al símil de los animales:

```
class Animal {
  constructor(n = "pato") {
    this.nombre = n;
  }

  hablar() {
    return "Cuak";
  }
  quienSoy() {
    return "Hola, soy " + this.nombre + ". ~" + this.hablar();
  }
}

const pato = new Animal("Donald");

pato.quienSoy(); // 'Hola, soy Donald. ~Cuak'
```

Ten en cuenta que si usas **this** en contextos concretos, como por ejemplo fuera de una clase te devolverá el objeto **Window**, que no es más que una referencia al objeto global dela pestaña actual donde nos encontramos y tenemos cargada la página web.

Es importante tener mucho cuidado con la palabra clave this, ya que en muchas situaciones creeremos que devolverá una ref

¿Qué es un getter?

Los **getters** son la forma de definir propiedades computadas de lectura en una clase.Veamos un ejemplo sobre el ejemplo anterior de la clase **Animal**:

```
class Animal {
  constructor(n) {
    this._nombre = n;
  }

  get nombre() {
    return "Sr. " + this._nombre;
  }

  hablar() {
    return "Cuak";
  }
  quienSoy() {
    return "Hola, soy " + this.nombre;
```

```
  }
}

// Creación de objetos
const pato = new Animal("Donald");

pato.nombre; // 'Sr. Donald'
pato.nombre = "Pancracio"; // 'Pancracio'pato.nombre; // 'Sr. Donald'
```

Si observas los resultados de este último ejemplo, puedes comprobar que la diferencia al utilizar **getters** es que las propiedades con **get** no se pueden cambiar, son de sólo lectura.

¿Qué es un setter?

De la misma forma que tenemos un **getter** para obtener información mediante **propiedades computadas**, también podemos tener un **setter**, que es el mismo concepto, pero en lugar de obtener información, para establecer información.

Si incluímos un **getter** y un **setter** a una propiedad en una clase, podremos modificarladirectamente:

```
class Animal {
  constructor(n) {
    this.nombre = n;
  }

  get nombre() {
    return "Sr. " + this._nombre;
  }

  set nombre(n) {
    this._nombre = n.trim();
  }

  hablar() {
    return "Cuak";
  }
  quienSoy() {
    return "Hola, soy " + this.nombre;
  }
}

// Creación de objetos
```

```
const pato = new Animal("Donald");

pato.nombre; // 'Sr. Donald' pato.nombre = " Lucas "; // ' Lucas ' pato.nombre; // 'Sr. Lucas'
```

Observa que de la misma forma que con los **getters**, podemos realizar tareas sobre los parámetros del **setter** antes de guardarlos en la propiedad interna. Esto nos servirá para hacer modificaciones previas, como por ejemplo, en el ejemplo anterior, realizando un **trim()**para limpiar posibles espacios antes de guardar esa información.

Fuente: lenguajejs.com

Iteración sobre el objeto con for..in

El siguiente bucle for...initera sobre todas las propiedades enumerables que no son símbolos del objeto y registra una cadena de los nombres de propiedad y sus valores.

```
let obj = {a: 1, b: 2, c: 3};

for (let prop in obj) {

console.log(`obj.${prop} = ${obj[prop]}`);

}

// Produce: // "obj.a = 1" // "obj.b = 2" // "obj.c = 3"
```

For…of // For…in

La sentencia **for...of** ejecuta un bloque de código para cada elemento de un objeto iterable, como lo son: String, Array, entre otros.

Sintaxis

```
for (variable of iterable) {statement
}
```

variable

En cada iteración el elemento (propiedad enumerable) correspondiente es asignadoa variable.

iterable

Objeto cuyas propiedades enumerables son iteradas.

Ejemplos Iterando un Array

```
let iterable = [10, 20, 30];

for (let value of iterable) {value += 1; console.log(value);
}
// 11
// 21
// 31
```

Es posible usar const en lugar de **let** si no se va a modificar la variable dentro del bloque.let iterable = [10, 20, 30];

for (const value of iterable) {console.log(value);
}
// 10
// 20
// 30

Iterando un String

let iterable = "boo";

for (let value of iterable) {console.log(value);
}
// "b"
// "o"
// "o"

Diferencia entre for...of y for...in

El bucle **for…in** iterará sobre todas las propiedades de un objeto. La sintaxis de **for...of** esespecífica para las colecciones, y no para todos los objetos.

El siguiente ejemplo muestra las diferencias entre un bucle for...of y un bucle for...in enarrays.

let arr = [3, 5, 7];arr.foo = "hola";

for (let i in arr) {
console.log(i); // logs "0", "1", "2", "foo"
}

```
for (let i of arr) {
console.log(i); // logs "3", "5", "7"
}
```

DOM

Si únicamente utilizamos HTML/CSS, sólo podremos crear páginas «estáticas» (sin demasiada personalización por parte del usuario), pero si añadimos Javascript, podremos crear páginas «dinámicas». Cuando hablamos de páginas dinámicas, nos referimos a que podemos dotar de la potencia y flexibilidad que nos da un lenguaje de programación para crear documentos y páginas mucho más ricas, que brinden una experiencia más completa y con el que se puedan automatizar un gran abanico de tareas y acciones.

¿Qué es el DOM?

Las siglas DOM significan Document Object Model, o lo que es lo mismo, la estructura del documento HTML. Una página HTML está formada por múltiples etiquetas HTML, anidadas una dentro de otra, formando un árbol de etiquetas relacionadas entre sí, que se denomina árbol DOM (o simplemente DOM).

En Javascript, cuando nos referimos al DOM nos referimos a esta estructura, que podemos modificar de forma dinámica desde Javascript, añadiendo nuevas etiquetas, modificando o eliminando otras, cambiando sus atributos HTML, añadiendo clases, cambiando el contenido de texto, etc...

Al estar "amparado" por un lenguaje de programación, todas estas tareas se pueden automatizar, incluso indicando que se realicen cuando el usuario haga acciones determinadas, como por ejemplo: pulsar un botón, mover el ratón, hacer click en una parte del documento, escribir un texto, etc...

El objeto document

En Javascript, la forma de acceder al DOM es a través de un objeto llamado document, que representa el árbol DOM de la página o pestaña del navegador donde nos encontramos. En su interior pueden existir varios tipos de elementos, pero principalmente serán Element o Node:

Element no es más que la representación genérica de una etiqueta: HTMLElement. Node es una unidad más básica, la cuál puede ser Element o un nodo de texto.

Todos los elementos HTML, dependiendo del elemento que sean, tendrán un tipo de dato específico. Algunos ejemplos:

Tipo de dato	Tipo específico	Etiqueta	Descripción
HTMLElement	HTMLDivElement		Capa divisoria invisible (en bloque).
HTMLElement	HTMLSpanElement		Capa divisoria invisible (en línea).
HTMLElement	HTMLImageElement	<img>	Imagen.
HTMLElement	HTMLAudioElement	<audio>	Contenedor de audio.

Obviamente, existen muchos tipos de datos específicos, uno por cada etiqueta HTML.

API nativa de Javascript

En los siguientes capítulos veremos que Javascript nos proporciona un conjunto de herramientas para trabajar de forma nativa con el DOM de la página, entre las que se encuentran:

Capítulo del DOM	Descripción
☐ Buscar etiquetas	Familia de métodos entre los que se encuentran funciones como .getElementById(), .querySelector() o .querySelectorAll(), entre otras.
☐ Crear etiquetas	Una serie de métodos y consejos para crear elementos en la página y trabajar con ellos de forma dinámica.
☐ Insertar etiquetas	Las mejores formas de añadir elementos al DOM, ya sea utilizando propiedades como .innerHTML o método como .appendChild(), .insertAdjacentHTML(), entre otros.
☐ Gestión de clases CSS	Consejos para la utilización de la API .classList de Javascript que nos permite manipular clases CSS desde JS, de modo que podamos añadir, modificar, eliminar clases de CSS de un elemento de una forma práctica y cómoda.
☐ Navegar entre elementos	Utilización de una serie de métodos y propiedades que nos permiten «navegar» a través de la jerarquía del DOM, ciñéndonos a la estructura del documento y la posición de los elementos en la misma.

Librerías de terceros

En muchos casos, el rendimiento no es lo suficientemente importante como para justificar trabajar a bajo nivel, por lo que se prefiere utilizar algunas librerías de terceros que nos facilitan el trabajo a costa de reducir minimamente el rendimiento, pero permitiéndonos programar más rápidamente.

Si es tu caso, puedes utilizar alguna de las siguientes librerías para abstraerte del DOM:

Librería	Descripción	GitHub
RE:DOM	Librería para crear interfaces de usuario, basada en DOM.	@redom/redom
Voyeur.js	Pequeña librería para manipular el DOM	@adriancooney/voyeur.js
HtmlJs	Motor de renderización de HTML y data binding (MVVM)	@nhanfu/htmljs
DOMtastic	Libraría moderna y modular para DOM/Events	@webpro/DOMtastic
Umbrella JS	Librería para manipular el DOM y eventos	@franciscop/umbrella
SuperDOM	Manipulando DOM como si estuvieras en 2018	@szaranger/superdom

Muchas veces, también se eligen frameworks de Javascript para trabajar, que en cierta forma también te abstraen de tener que gestionar el DOM a bajo nivel, y lo cambian por realizar otras tareas o estrategias relacionadas con el framework escogido.

Si nos encontramos en nuestro código Javascript y queremos hacer modificaciones en un elemento de la página HTML, lo primero que debemos hacer es buscar dicho elemento. Para ello, se suele intentar identificar el elemento a través de alguno de sus atributos más utilizados, generalmente el id o la clase.

Métodos tradicionales

Existen varios métodos, los más clásicos y tradicionales para realizar búsquedas de elementos en el documento. Observa que si lo que buscas es un elemento específico, lo mejor sería utilizar getElementById(), en caso contrario, si utilizamos uno de los 3 siguientes métodos, nos devolverá un array donde tendremos que elegir el elemento en cuestión posteriormente:

Métodos de búsqueda	Descripción
Element .getElementById(id)	Busca el elemento HTML con el id id. Si no, devuelve .
Array .getElementsByClassName(class)	Busca elementos con la clase class. Si no, devuelve [].
Array .getElementsByName(name)	Busca elementos con atributo name name. Si no, devuelve [].
Array .getElementsByTagName(tag)	Busca elementos tag. Si no encuentra ninguno, devuelve [].

Estos son los 4 métodos tradicionales de Javascript para manipular el DOM. Se denominan tradicionales porque son los que existen en Javascript desde versiones más antiguas. Dichos métodos te permiten buscar elementos en la página dependiendo de los atributos id, class, name o de la propia etiqueta, respectivamente.

getElementById()

El primer método, .getElementById(id) busca un elemento HTML con el id especificado en id por parámetro. En principio, un documento HTML bien construído no debería tener más de un elemento con el mismo id, por lo tanto, este método devolverá siempre un solo elemento:

```
const page = document.getElementById("page"); // <div id="page"></div>
```

Recuerda que en el caso de no encontrar el elemento indicado, devolverá .

getElementsByClassName()

Por otro lado, el método .getElementsByClassName(class) permite buscar los elementos con la clase especificada en class. Es importante darse cuenta del matiz de que el metodo tiene getElements en plural, y esto es porque al devolver clases (al contrario que los id) se pueden repetir, y por lo tanto, devolvernos varios elementos, no sólo uno.

```
const items = document.getElementsByClassName("item"); // [div, div, div]


console.log(items[0]); // Primer item encontrado: <div class="item"></div>

console.log(items.length); // 3
```

Estos métodos devuelven siempre un Array con todos los elementos encontrados que encajen con el criterio. En el caso de no encontrar ninguno, devolverán un Array vacío: [].

Exactamente igual funcionan los métodos getElementsByName(name) y getElementsByTagName(tag), salvo que se encargan de buscar elementos HTML por su atributo name o por su propia etiqueta de elemento HTML, respectivamente:

// Obtiene todos los elementos con atributo name="nickname" const

nicknames = document.getElementsByName("nickname");

// Obtiene todos los elementos de la página const divs =

document.getElementsByTagName("div");

OJO: Aunque en esta documentación se hace referencia a Array , realmente los métodos de búsqueda generalmente devuelven un tipo de dato HTMLCollection o NodeList, que aunque actúan de forma muy similar a un Array , no son arrays, y por lo tanto pueden carecer de algunos métodos, como por ejemplo .forEach().

Recuerda que el primer método tiene getElement en singular y el resto getElements en plural. Ten en cuenta ese detalle para no olvidarte que uno devuelve un sólo elemento y el resto una lista de ellos.

Métodos modernos

Aunque podemos utilizar los métodos tradicionales que acabamos de ver, actualmente tenemos a nuestra disposición dos nuevos métodos de búsqueda de elementos que son mucho más cómodos y prácticos si conocemos y dominamos los selectores CSS https://lenguajecss.com/css/selectores/selectores-basicos/. Es el caso de los métodos .querySelector() y .querySelectorAll():

Método de búsqueda	Descripción
Element .querySelector(sel)	Busca el primer elemento que coincide con el selector CSS sel. Si no, .
Array .querySelectorAll(sel)	Busca todos los elementos que coinciden con el selector CSS sel. Si no, [].

Con estos dos métodos podemos realizar todo lo que hacíamos con los métodos tradicionales mencionados anteriormente e incluso muchas más cosas (en menos código), ya que son muy flexibles y potentes gracias a los selectores CSS.

querySelector()

El primero, .querySelector(selector) devuelve el primer elemento que encuentra que encaja con el selector CSS suministrado en selector. Al igual que su «equivalente»

.getElementById(), devuelve un solo elemento y en caso de no coincidir con ninguno, devuelve null :

const page = document.querySelector("#page"); // const info

= document.querySelector(".main .info"); //

Lo interesante de este método, es que al permitir suministrarle un selector CSS básico o incluso un selector CSS avanzado, se vuelve un sistema mucho más potente.

El primer ejemplo es equivalente a utilizar un .getElementById(), sólo que en la versión de .querySelector() indicamos por parámetro un selector , y en el primero le pasamos un simple String . Observa que estamos indicando un # porque se trata de un id.

En el segundo ejemplo, estamos recuperando el primer elemento con clase info que se encuentre dentro de otro elemento con clase main. Eso podría realizarse con los métodos tradicionales, pero sería menos directo ya que tendríamos que realizar varias llamadas, con .querySelector() se hace directamente con sólo una.

querySelectorAll()

Por otro lado, el método .querySelectorAll() realiza una búsqueda de elementos como lo hace el anterior, sólo que como podremos intuir por ese All(), devuelve un Array con todos los elementos que coinciden con el Selector CSS:

```
// Obtiene todos los elementos con clase "info" const

infos = document.querySelectorAll(".info");



// Obtiene todos los elementos con atributo name="nickname"

const nicknames = document.querySelectorAll('[name="nickname"]');



// Obtiene todos los elementos <div> de la página HTML const

divs = document.querySelectorAll("div");
```

En este caso, recuerda que .querySelectorAll() siempre nos devolverá un Array de elementos. Depende de los elementos que encuentre mediante el Selector, nos devolverá un Array de 0 elementos o de 1, 2 o más elementos.

Al realizar una búsqueda de elementos y guardarlos en una variable, podemos realizar la búsqueda posteriormente sobre esa variable en lugar de hacerla sobre document.
Esto permite realizar búsquedas acotadas por zonas, en lugar de realizarlo siempre sobre document, que buscará en todo el documento HTML.

Sobre todo si te encuentras en fase de aprendizaje, lo normal suele ser crear código HTML desde un fichero HTML. Sin embargo, y sobre todo con el auge de las páginas SPA (Single Page Application*) y los frameworks Javascript, esto ha cambiado bastante y es bastante frecuente crear código HTML desde Javascript de forma dinámica.

Esto tiene sus ventajas y sus desventajas. Un fichero .html siempre será más sencillo, más «estático» y más directo, ya que lo primero que analiza un navegador web es un fichero de marcado HTML. Por otro lado, un fichero .js es más complejo y menos directo, pero mucho más potente, «dinámico» y flexible, con menos limitaciones.

En este artículo vamos a ver cómo podemos crear elementos HTML desde Javascript y aprovecharnos de la potencia de Javascript para hacer cosas que desde HTML, sin ayuda de Javascript, no podríamos realizar o costaría mucho más.

Elementos HTML

Crear elementos HTML

Existen una serie de métodos para crear de forma eficiente diferentes elementos HTML o nodos, y que nos pueden convertir en una tarea muy sencilla el crear estructuras dinámicas, mediante bucles o estructuras definidas:

Métodos	Descripción
Element .createElement(tag, options)	Crea y devuelve el elemento HTML definido por el String tag.
Nodo .createComment(text)	Crea y devuelve un nodo de comentarios HTML <!-- text --.
Node .createTextNode(text)	Crea y devuelve un nodo HTML con el texto text.
Node .cloneNode(deep)	Clona el nodo HTML y devuelve una copia. deep es false por defecto.
Boolean .isConnected	Indica si el nodo HTML está insertado en el documento HTML.

Para empezar, nos centraremos principalmente en la primera, que es la que utilizamos para crear elementos HTML en el DOM.

El método createElement()

Mediante el método .createElement() podemos crear un elemento HTML en memoria (¡no estará insertado aún en nuestro documento HTML!). Con dicho elemento almacenado en una variable, podremos modificar sus características o contenido, para posteriormente insertarlo en una posición determinada del DOM o documento HTML.

Vamos a centrarnos en el proceso de creación del elemento, y en el próximo capítulo veremos el apartado de insertarlo en el DOM. El funcionamiento de .createElement() es muy sencillo: se trata de pasarle el nombre de la etiqueta tag a utilizar.

```
const div = document.createElement("div"); // Creamos un <div></div>


const span = document.createElement("span"); // Creamos un <span></span> const

img = document.createElement("img"); // Creamos un <img>
```

De la misma forma, podemos crear comentarios HTML con createComment() o nodos de texto sin etiqueta HTML con createTextNode(), pasándole a ambos un String con el

texto en cuestión. En ambos, se devuelve un Nodo que podremos utilizar luego para insertar en el documento HTML:

```
const comment = document.createComment("Comentario"); // <!--Comentario--> const

text = document.createTextNode("Hola"); // Nodo de texto: 'hola'
```

El método createElement() tiene un parámetro opcional denominado options. Si se indica, será un objeto con una propiedad is para definir un elemento personalizado en una modalidad menos utilizada. Se verá más adelante en el apartado de Web Components.

Ten presente que en los ejemplos que hemos visto estamos creando los elementos en una constante, pero de momento no están añadiéndose al documento HTML, por lo que no aparecerían visualmente. Más adelante veremos cómo añadirlos.

El método cloneNode()

Hay que tener mucho cuidado al crear y duplicar elementos HTML. Un error muy común es asignar un elemento que tenemos en otra variable, pensando que estamos creando una copia (cuando no es así):

```
const div = document.createElement("div");

div.textContent = "Elemento 1";


const div2 = div; // NO se está haciendo una copia div2.textContent =

"Elemento 2";


div.textContent; // 'Elemento 2'
```

Con esto, quizás pueda parecer que estamos duplicando un elemento para crearlo a imagen y semejanza del original. Sin embargo, lo que se hace es una referencia al elemento original, de modo que si se modifica el div2, también se modifica el elemento original. Para evitar esto, lo ideal es utilizar el método cloneNode():

```
const div = document.createElement("div");

div.textContent = "Elemento 1";
```

```
const div2 = div.cloneNode(); // Ahora SÍ estamos clonando

div2.textContent = "Elemento 2";


div.textContent; // 'Elemento 1'
```

El método cloneNode(deep) acepta un parámetro Boolean deep opcional, a false por defecto, para indicar el tipo de clonación que se realizará:

Si es true, clonará también sus hijos, conocido como una clonación profunda (Deep Clone).

Si es false, no clonará sus hijos, conocido como una clonación superficial (Shallow Clone).

La propiedad isConnected

Por último, la propiedad isConnected nos indica si el nodo en cuestión está conectado al DOM, es decir, si está insertado en el documento HTML:

Si es true, significa que el elemento está conectado al DOM.

Si es false, significa que el elemento no está conectado al DOM.

Hasta ahora, hemos creado elementos que no lo están (permanecen sólo en memoria). En despues veremos como insertarlos en el documento HTML para que aparezca visualmente en la página.

Atributos HTML de un elemento

Hasta ahora, hemos visto como crear elementos HTML con Javascript, pero no hemos visto como modificar los atributos HTML de dichas etiquetas creadas. En general, una vez tenemos un elemento sobre el que vamos a crear algunos atributos, lo más sencillo es asignarle valores como propiedades de objetos:

```
const div = document.createElement("div"); // <div></div>

div.id = "page"; // <div id="page"></div>

div.className = "data"; // <div id="page" class="data"></div>


div.style = "color: red"; // <div id="page" class="data" style="color: red"></div>
```

Sin embargo, en algunos casos esto se puede complicar (como se ve en uno de los casos del ejemplo anterior). Por ejemplo, la palabra class (para crear clases) o la palabra for (para bucles) son palabras reservadas de Javascript y no se podrían utilizar para crear atributos. Por ejemplo, si queremos establecer una clase, se debe utilizar la propiedad className.

Es posible asignar a la propiedad className varias clases en un String separadas por espacio. De esta forma se asignarán múltiples clases. Aún así, recomendamos utilizar

la propiedad classList que explicaremos más adelante en el capítulo manipulación de clases CSS.

Aunque la forma anterior es la más rápida, tenemos algunos métodos para utilizar en un elemento HTML y añadir, modificar o eliminar sus atributos:

Métodos	Descripción
hasAttributes()	Indica si el elemento tiene atributos HTML.
hasAttribute(attr)	Indica si el elemento tiene el atributo HTML attr.
getAttributeNames()	Devuelve un Array con los atributos del elemento.
getAttribute(attr)	Devuelve el valor del atributo attr del elemento o Null si no existe.
removeAttribute(attr)	Elimina el atributo attr del elemento.
setAttribute(attr, value)	Añade o cambia el atributo attr al valor value.
getAttributeNode(attr)	Idem a getAttribute() pero devuelve el atributo como nodo.
removeAttributeNode(attr)	Idem a removeAttribute() pero devuelve el atributo como nodo.
setAttributeNode(attr, value)	Idem a setAttribute() pero devuelve el atributo como nodo.

Estos métodos son bastante autoexplicativos y fáciles de entender, aún así, vamos a ver unos ejemplos de uso donde podemos ver cómo funcionan:

// Obtenemos const

div = document.querySelector("#page");

div.hasAttribute("data-number"); // true (data-number existe)

div.hasAttributes(); // true (tiene 3 atributos)

div.getAttributeNames(); // ["id", "data-number", "class"]

div.getAttribute("id"); // "page"

div.removeAttribute("id"); //

div.setAttribute("id", "page"); // (Vuelve a añadirlo)

Los tres últimos métodos mencionados: getAttributeNode(), removeAttributeNode() y setAttributeNode() son versiones idénticas a sus homónimos, sólo que devuelven el Nodo afectado, útil si queremos guardarlo en una variable y seguir trabajando con él.

Recuerda que hasta ahora hemos visto como crear elementos y cambiar sus atributos, pero no los hemos insertado en el DOM o documento HTML, por lo que no los veremos visualmente en la página. En el siguiente capítulo abordaremos ese tema.

Hemos visto como crear elementos en el DOM, pero dichos elementos se creaban en memoria y los almacenábamos en una variable o constante. No se conectaban al DOM o documento HTML de forma automática, sino que debemos hacerlo manualmente, que es justo lo que veremos en este artículo: como insertar elementos en el DOM, así como eliminarlos.

En este artículo vamos a centrarnos en tres categorías:

Reemplazar contenido de elementos en el DOM

Insertar elementos en el DOM

Eliminar elementos del DOM

Reemplazar contenido

Comenzaremos por la familia de propiedades siguientes, que enmarcamos dentro de la categoría de reemplazar contenido de elementos HTML. Se trata de una vía rápida con la cual podemos añadir (o más bien, reemplazar) el contenido de una etiqueta HTML.

Las propiedades son las siguientes:

Propiedades	Descripción
.nodeName	Devuelve el nombre del nodo (etiqueta si es un elemento HTML). Sólo lectura.
.textContent	Devuelve el contenido de texto del elemento. Se puede asignar para modificar.
.innerHTML	Devuelve el contenido HTML del elemento. Se puede usar asignar para modificar.
.outerHTML	Idem a .innerHTML pero incluyendo el HTML del propio elemento HTML.
.innerText	Versión no estándar de .textContent de Internet Explorer con diferencias.

	Evitar.
.outerTex t	Versión no estándar de .textContent/.outerHTML de Internet Explorer. Evitar.

La propiedad nodeName nos devuelve el nombre del todo, que en elementos HTML es interesante puesto que nos devuelve el nombre de la etiqueta en mayúsculas. Se trata de una propiedad de sólo lectura, por lo cuál no podemos modificarla, sólo acceder a ella.

La propiedad textContent

La propiedad .textContent nos devuelve el contenido de texto de un elemento HTML. Es útil para obtener (o modificar) sólo el texto dentro de un elemento, obviando el etiquetado HTML:

const div = document.querySelector("div"); //

div.textContent = "Hola a todos"; // Hola a todos div.textContent; //

"Hola a todos"

Observa que también podemos utilizarlo para reemplazar el contenido de texto, asignándolo como si fuera una variable o constante. En el caso de que el elemento tenga anidadas varias etiquetas HTML una dentro de otra, la propiedad .textContent se quedará sólo con el contenido textual completo, como se puede ver en el siguiente ejemplo:

// Obtenemos Hola amigos const

div = document.querySelector(".info");

div.textContent; // "Hola amigos"

La propiedad innerHTML

Por otro lado, la propiedad .innerHTML nos permite hacer lo mismo, pero interpretando el código HTML indicado y renderizando sus elementos:

const div = document.querySelector(".info"); //

div.innerHTML = "Importante"; // Interpreta el HTML

div.innerHTML; // "Importante"

div.textContent; // "Importante"

div.textContent = "Importante"; // No interpreta el HTML

Observa que la diferencia principal entre .innerHTML y .textContent es que el primero renderiza e interpreta el marcado HTML, mientras que el segundo lo inserta como contenido de texto literalmente.

Ten en cuenta que la propiedad .innerHTML comprueba y parsea el marcado HTML escrito (corrigiendo si hay errores) antes de realizar la asignación. Por ejemplo, si en el ejemplo anterior nos olvidamos de escribir el cierre de la etiqueta,

.innerHTML automáticamente lo cerrará. Esto puede provocar algunas incongruencias si el código es incorrecto o una disminución de rendimiento en textos muy grandes que hay que preprocesar.

Por otro lado, la propiedad .outerHTML es muy similar a .innerHTML. Mientras que esta última devuelve el código HTML del interior de un elemento HTML, .outerHTML devuelve también el código HTML del propio elemento en cuestión. Esto puede ser muy útil para reemplazar un elemento HTML combinándolo con .innerHTML:

```
const data = document.querySelector(".data");

data.innerHTML = "<h1>Tema 1</h1>";



data.textContent; // "Tema 1" data.innerHTML; //

"<h1>Tema 1</h1>"

data.outerHTML; // "<div class="data"><h1>Tema 1</h1></div>"
```

En este ejemplo se pueden observar las diferencias entre las propiedades .textContent (contenido de texto), .innerHTML (contenido HTML) y .outerHTML (contenido y contenedor HTML).

Las propiedades .innerText y .outerText son propiedades no estándar de Internet Explorer. Se recomienda sólo utilizarlas con motivos de fallbacks o para dar soporte a versiones antiguas de Internet Explorer. En su lugar debería utilizarse .textContent.

Insertar elementos

A pesar de que los métodos anteriores son suficientes para crear elementos y estructuras HTML complejas, sólo son aconsejables para pequeños fragmentos de código o texto, ya que en estructuras muy complejas (con muchos elementos HTML) la legibilidad del código sería menor y además, el rendimiento podría resentirse.

Hemos aprendido a crear elementos HTML y sus atributos https://lenguajejs.com/javascript/dom/crear-elementos-dom/, pero aún no hemos visto como añadirlos al documento HTML actual (conectarlos al DOM), operación que se puede realizar de diferentes formas mediante los siguientes métodos disponibles:

Métodos	Descripción
.appendChild(node)	Añade como hijo el nodo node. Devuelve el nodo insertado.
.insertAdjacentElement(pos, elem)	Inserta el elemento elem en la posición pos. Si falla, .
.insertAdjacentHTML(pos, str)	Inserta el código HTML str en la posición pos.
.insertAdjacentText(pos, text)	Inserta el texto text en la posición pos.
.insertBefore(new, node)	Inserta el nodo new antes de node y como hijo del nodo actual.

De ellos, probablemente el más extendido es .appendChild(), no obstante, la familia de métodos .insertAdjacent*() también tiene buen soporte en navegadores y puede usarse de forma segura en la actualidad.

El método appendChild()

Uno de los métodos más comunes para añadir un elemento HTML creado con Javascript es appendChild(). Como su propio nombre indica, este método realiza un

«append», es decir, inserta el elemento como un hijo al final de todos los elementos hijos que existan.

Es importante tener clara esta particularidad, porque aunque es lo más común, no siempre querremos insertar el elemento en esa posición:

```
const img = document.createElement("img"); img.src =

"https://lenguajejs.com/assets/logo.svg"; img.alt = "Logo

Javascript";



document.body.appendChild(img);
```

En este ejemplo podemos ver como creamos un elemento <img> que aún no está conectado al DOM. Posteriormente, añadimos los atributos src y alt, obligatorios en una etiqueta de imagen. Por último, conectamos al DOM el elemento, utilizando el método .appendChild() sobre document.body que no es más que una referencia a la etiqueta <body> del documento HTML.

Veamos otro ejemplo:

```
const div = document.createElement("div"); div.textContent =

"Esto es un div insertado con JS.";
```

```
const app = document.createElement("div"); // <div></div>

app.id = "app"; // <div id="app"></div>

app.appendChild(div); // <div id="app"><div>Esto es un div insertado con JS</div></div>
```

En este ejemplo, estamos creando dos elementos, e insertando uno dentro de otro. Sin embargo, a diferencia del anterior, el elemento app no está conectado aún al DOM, sino que lo tenemos aislado en esa variable, sin insertar en el documento. Esto ocurre porque app lo acabamos de crear, y en el ejemplo anterior usabamos document.body que es una referencia a un elemento que ya existe en el documento.

Los métodos insertAdjacent*()

Los métodos de la familia insertAdjacent son bastante más versátiles que

.appendChild(), ya que permiten muchas más posibilidades. Tenemos tres versiones diferentes:

.insertAdjacentElement() donde insertamos un objeto Element

.insertAdjacentHTML() donde insertamos código HTML directamente (similar a innerHTML)

.insertAdjacentText() donde no insertamos elementos HTML, sino un Nodo con texto

En las tres versiones, debemos indicar por parámetro un String pos como primer parámetro para indicar en que posición vamos a insertar el contenido. Hay 4 opciones posibles:

beforebegin: El elemento se inserta antes de la etiqueta HTML de apertura. afterbegin: El elemento se inserta dentro de la etiqueta HTML, antes de su primer hijo.

beforeend: El elemento se inserta dentro de la etiqueta HTML, después de su último hijo. Es el equivalente a usar el método .appendChild().

afterend: El elemento se inserta después de la etiqueta HTML de cierre.

Veamos algunos ejemplo aplicando cada uno de ellos con el método .insertAdjacentElement():

```
const div = document.createElement("div"); // <div></div>

div.textContent = "Ejemplo"; // <div>Ejemplo</div>


const app = document.querySelector("#app"); // <div id="app">App</div>


app.insertAdjacentElement("beforebegin", div);

// Opción 1: <div>Ejemplo</div> <div id="app">App</div>


app.insertAdjacentElement("afterbegin", div);
```

// Opción 2: Ejemplo App

app.insertAdjacentElement("beforeend", div);

// Opción 3: App Ejemplo

app.insertAdjacentElement("afterend", div);

// Opción 4: App Ejemplo

Ten en cuenta que en el ejemplo muestro varias opciones alternativas, no lo que ocurriría tras ejecutar las cuatro opciones una detrás de otra.

Por otro lado, notar que tenemos tres versiones en esta familia de métodos, una que actua sobre elementos HTML (la que hemos visto), pero otras dos que actuan sobre código HTML y sobre nodos de texto. Veamos un ejemplo de cada una:

app.insertAdjacentElement("beforebegin", div);

// Opción 1: Ejemplo App

app.insertAdjacentHTML("beforebegin", ' Hola ');

// Opción 2: Hola App

app.insertAdjacentText("beforebegin", "Hola a todos");

// Opción 3: Hola a todos App

El método insertBefore()

Por último, el método insertBefore(newnode, node) es un método más específico y menos utilizado en el que se puede especificar exactamente el lugar a insertar un nodo. El parámetro newnode es el nodo a insertar, mientras que node puede ser:

Null; insertando newnode después del último nodo hijo. Equivalente a .appendChild(). Nodo

o Elemento ; insertando newnode antes de dicho node de referencia.

Eliminar elementos

Al igual que podemos insertar o reemplazar elementos, también podemos eliminarlos. Ten en cuenta que al «eliminar» un nodo o elemento HTML, lo que hacemos realmente no es borrarlo, sino desconectarlo del DOM o documento HTML, de modo que no están conectados, pero siguen existiendo.

El método remove()

Probablemente, la forma más sencilla de eliminar nodos o elementos HTML es utilizando el método .remove() sobre el nodo o etiqueta a eliminar:

```
const div = document.querySelector(".deleteme");



div.isConnected; // true

div.remove(); div.isConnected;

// false
```

En este caso, lo que hemos hecho es buscar el elemento HTML en el documento HTML y desconectarlo de su elemento padre, de forma que dicho elemento pasa a no pertenecer al documento HTML.

Sin embargo, existen algunos métodos más para eliminar o reemplazar elementos:

Métodos	Descripción
.remove()	Elimina el propio nodo de su elemento padre.
.removeChild(node)	Elimina y devuelve el nodo hijo node.
.replaceChild(new, old)	Reemplaza el nodo hijo old por new. Devuelve old.

El método .remove() se encarga de desconectarse del DOM a sí mismo, mientras que el segundo método, .removeChild(), desconecta el nodo o elemento HTML

proporcionado. Por último, con el método .replaceChild() se nos permite cambiar un nodo por otro.

El método removeChild()

En algunos casos, nos puede interesar eliminar un nodo hijo de un elemento. Para esas situaciones, podemos utilizar el método .removeChild(node) donde node es el nodo hijo que queremos eliminar:

```
const div = document.querySelector(".item:nth-child(2)"); // <div class="item">2</div>



document.body.removeChild(div); // Desconecta el segundo .item
```

El método replaceChild()

De la misma forma, el método replaceChild(new, old) nos permite cambiar un nodo hijo old por un nuevo nodo hijo new. En ambos casos, el método nos devuelve el nodo reemplazado:

```
const div = document.querySelector(".item:nth-child(2)");
```

```
const newnode = document.createElement("div");

newnode.textContent = "DOS";


document.body.replaceChild(newnode, div);
```

Clases CSS

Manipular clases CSS

En CSS es muy común utilizar múltiples clases CSS para asignar estilos relacionados dependiendo de lo que queramos. Para ello, basta hacer cosas como la que veremos a continuación:

```
<div class="element shine dark-theme"></div>
```

La clase element sería la clase general que representa el elemento, y que tiene estilos fijos.

La clase shine podría tener una animación CSS para aplicar un efecto de brillo.

La clase dark-theme podría tener los estilos de un elemento en un tema oscuro.

Todo esto se utiliza sin problema de forma estática, pero cuando comenzamos a programar en Javascript, buscamos una forma dinámica, práctica y cómoda de hacerlo desde Javascript, y es de lo que tratará este artículo.

La propiedad className

Javascript tiene a nuestra disposición una propiedad .className en todos los elementos HTML. Dicha propiedad contiene el valor del atributo HTML class, y puede tanto leerse como reemplazarse:

Propiedad	Descripción
.className	Acceso directo al valor del atributo HTML class. También se puede asignar.
.classList	Objeto especial para manejar clases CSS. Contiene métodos y propiedades de ayuda.

La propiedad .className viene a ser la modalidad directa y rápida de utilizar el getter .getAttribute("class") y el setter .setAttribute("class", v). Veamos un ejemplo utilizando estas propiedades y métodos y su equivalencia:

```
const div = document.querySelector(".element");


// Obtener clases CSS

div.className; // "element shine dark-theme" div.getAttribute("class"); //
"element shine dark-theme"
```

// Modificar clases CSS

div.className = "elemento brillo tema-oscuro"; div.setAttribute("class",

"elemento brillo tema-oscuro");

Trabajar con .className tiene una limitación cuando trabajamos con múltiples clases CSS, y es que puedes querer realizar una manipulación sólo en una clase CSS concreta, dejando las demás intactas. En ese caso, modificar clases CSS mediante una asignación .className se vuelve poco práctico. Probablemente, la forma más interesante de manipular clases desde Javascript es mediante el objeto .classList.

El objeto classList

Para trabajar más cómodamente, existe un sistema muy interesante para trabajar con clases: el objeto classList. Se trata de un objeto especial (lista de clases) que contiene una serie de ayudantes que permiten trabajar con las clases de forma más intuitiva y lógica.

Si accedemos a .classList, nos devolverá un Array (lista) de clases CSS de dicho elemento. Pero además, incorpora una serie de métodos ayudantes que nos harán muy sencillo trabajar con clases CSS:

Método	Descripción
.classList	Devuelve la lista de clases del elemento HTML.
.classList.item(n)	Devuelve la clase número n del elemento HTML.

.classList.add(c1, c2, ...)	Añade las clases c1, c2... al elemento HTML.
.classList.remove(c1, c2, ...)	Elimina las clases c1, c2... del elemento HTML.
.classList.contains(clase)	Indica si la clase existe en el elemento HTML.
.classList.toggle(clase)	Si la clase no existe, la añade. Si no, la elimina.
.classList.toggle(clase, expr)	Si expr es true, añade clase. Si no, la elimina.
.classList.replace(old, new)	Reemplaza la clase old por la clase new.

OJO: Recuerda que el objeto .classList aunque parece que devuelve un Array no es un array, sino un elemento que actúa como un array, por lo que puede carecer de algunos métodos o propiedades concretos. Si quieres convertirlo a un array real, utiliza Array.from().

Veamos un ejemplo de uso de cada método de ayuda. Supongamos que tenemos el siguiente elemento HTML en nuestro documento. Vamos a acceder a el y a utilizar el objeto .classList con dicho elemento:

```
<div id="page" class="info data dark" data-number="5"></div>
```

Observa que dicho elemento HTML tiene:

Un atributo id

Tres clases CSS: info, data y dark Un

metadato HTML data-number **Añadir y**

eliminar clases CSS

Los métodos classList.add() y classList.remove() permiten indicar una o múltiples clases CSS a añadir o eliminar. Observa el siguiente código donde se ilustra un ejemplo:

```
const div = document.querySelector("#page");



div.classList; // ["info", "data", "dark"]



div.classList.add("uno", "dos"); // No devuelve nada.

div.classList; // ["info", "data", "dark", "uno", "dos"]



div.classList.remove("uno", "dos"); // No devuelve nada.

div.classList; // ["info", "data", "dark"]
```

En el caso de que se añada una clase CSS que ya existía previamente, o que se elimine una clase CSS que no existía, simplemente no ocurrirá nada.

Conmutar o alternar clases CSS

Un ayudante muy interesante es el del método classList.toggle(), que lo que hace es añadir o eliminar la clase CSS dependiendo de si ya existía previamente. Es decir, añade la clase si no existía previamente o elimina la clase si existía previamente:

```
const div = document.querySelector("#page");

div.classList; // ["info", "data", "dark"]


div.classList.toggle("info"); // Como "info" existe, lo elimina. Devuelve "false"

div.classList; // ["data", "dark"]


div.classList.toggle("info"); // Como "info" no existe, lo añade. Devuelve "true"

div.classList; // ["info", "data", "dark"]
```

Observa que .toggle() devuelve un que será true o false dependiendo de si, tras la operación, la clase sigue existiendo o no. Ten en cuenta que en .toggle(), al contrario que .add() o .remove(), sólo se puede indicar una clase CSS por parámetro.

Otros métodos de clases CSS

Por otro lado, tenemos otros métodos menos utilizados, pero también muy interesantes:

El método .classList.item(n) nos devuelve la clase CSS ubicada en la posición n.

El método .classList.contains(name) nos devuelve si la clase CSS name existe o no. El método .classList.replace(old, current) cambia la clase old por la clase current.

Veamos un ejemplo:

```
const div = document.querySelector("#page");


div.classList; // ["info", "data", "dark"]


div.classList.item(1); // 'data'


div.classList.contains("info"); // Devuelve `true` (existe la clase) div.classList.replace("dark",

"light"); // Devuelve `true` (se hizo el cambio)
```

Con todos estos métodos de ayuda, nos resultará mucho más sencillo manipular clases CSS desde Javascript en nuestro código.

Eventos

En la programación tradicional, las aplicaciones se ejecutan secuencialmente de principio afin para producir sus resultados. Sin embargo, en la actualidad el modelo predominante es el de la programación basada en eventos. Los scripts y programas esperan sin realizar ninguna tarea hasta que se produzca un evento. Una vez producido, ejecutan alguna tarea asociada a la aparición de ese evento y cuando concluye, el script o programa vuelve al estado de espera.

JavaScript permite realizar scripts con ambos métodos de programación: secuencial y basada en eventos. Los eventos de JavaScript permiten la interacción entre las aplicacionesJavaScript y los usuarios. Cada vez que se pulsa un botón, se produce un evento. Cada vezque se pulsa una tecla, también se produce un evento. No obstante, para que se produzca un evento no es obligatorio que intervenga el usuario, ya que, por ejemplo, cada vez que se carga una página, también se produce un evento.

TIPOS DE EVENTOS

Cada elemento HTML tiene definida su propia lista de posibles eventos que se le pueden asignar. Un mismo tipo de evento (por ejemplo, pinchar el botón izquierdo del ratón) puedeestar definido para varios elementos HTML y un mismo elemento HTML puede tener asociados diferentes eventos.

El nombre de los eventos se construye mediante el prefijo on, seguido del nombre en inglésde la acción asociada al evento. Así, el evento de pinchar un elemento con el ratón se denomina onclick y el evento asociado a la acción de mover el ratón se denomina onmousemove.

La siguiente tabla resume los eventos más importantes definidos por JavaScript:

Evento	Descripción	Elementos para los que está definido
onblur	Un elemento pierde el foco	<button>, <input>, <label>, <select>, <textarea>, <body>

onchange	Un elemento hasido modificado	<input>, <select>, <textarea>
onclick	Pulsar y soltar elratón	Todos los elementos
ondblclick	Pulsar dos veces seguidas con el ratón	Todos los elementos
onfocus	Un elemento obtiene el foco	<button>, <input>, <label>, <select>, <textarea>, <body>
onkeydown	Pulsar una tecla yno soltarla	Elementos de formulario y <body>
onkeypress	Pulsar una tecla	Elementos de formulario y <body>
onkeyup	Soltar una tecla pulsada	Elementos de formulario y <body>
onload	Página cargada completamente	<body>

onmousedown	Pulsar un botóndel ratón y no soltarlo	Todos los elementos
onmousemove	Mover el ratón	Todos los elementos
onmouseout	El ratón "sale" del elemento	Todos los elementos
onmouseover	El ratón "entra" enel elemento	Todos los elementos
onmouseup	Soltar el botón del ratón	Todos los elementos
onreset	Inicializar elformulario	<form>
onresize	Modificar el tamaño de laventana	<body>
onselect	Seleccionar untexto	<input>, <textarea>
onsubmit	Enviar el formulario	<form>

onunload	Se abandona la página, por ejemplo al cerrarel navegador	<body>

Los eventos más utilizados en las aplicaciones web tradicionales son onload para esperar aque se cargue la página por completo, los eventos onclick, onmouseover, onmouseout paracontrolar el ratón y onsubmit para controlar el envío de los formularios.

Algunos eventos de la tabla anterior (onclick, onkeydown, onkeypress, onreset, onsubmit) permiten evitar la "acción por defecto" de ese evento. Más adelante se muestra en detalle este comportamiento, que puede resultar muy útil en algunas técnicas de programación.

Las acciones típicas que realiza un usuario en una página web pueden dar lugar a una sucesión de eventos. Al pulsar por ejemplo sobre un botón de tipo <input type="submit"> sedesencadenan los eventos onmousedown, onclick, onmouseup y onsubmit de forma consecutiva.

MANEJADORES DE EVENTOS

Un evento de JavaScript por sí mismo carece de utilidad. Para que los eventos resulten útiles, se deben asociar funciones o código JavaScript a cada evento. De esta forma, cuando se produce un evento se ejecuta el código indicado, por lo que la aplicación puederesponder ante cualquier evento que se produzca durante su ejecución.

Las funciones o código JavaScript que se definen para cada evento se denominan *manejador de eventos* (*event handlers* en inglés).

HANDLERS Y LISTENERS

En las secciones anteriores se introdujo el concepto de "event handler" o manejador de eventos, que son las funciones que responden a los eventos que se producen. Además, se vieron tres formas de definir los manejadores de eventos para el modelo básico de eventos:

1. Código JavaScript dentro de un atributo del propio elemento HTML
2. Definición del evento en el propio elemento HTML pero el manejador es unafunción externa
3. Manejadores semánticos asignados mediante DOM sin necesidad de modificar el código HTML de la página

Cualquiera de estos tres modelos funciona correctamente en todos los navegadores disponibles en la actualidad. Las diferencias entre navegadores surgen cuando se definemás de un manejador de eventos para un mismo evento de un elemento. La forma de asignar y *"desasignar"* manejadores múltiples depende completamente del navegador utilizado.

MANEJADORES DE EVENTOS DE DOM

La especificación DOM define otros dos métodos similares a los disponibles para InternetExplorer y denominados addEventListener() y removeEventListener() para asociar y desasociar manejadores de eventos.

La principal diferencia entre estos métodos y los anteriores es que en este caso se requieren tres parámetros: el nombre del *"event listener"*, una referencia a la funciónencargada de procesar el evento y el tipo de flujo de eventos al que se aplica.

El primer argumento no es el nombre completo del evento como sucede en el modelo de Internet Explorer, sino que se debe eliminar el prefijo on. En otras palabras, si en InternetExplorer se utilizaba el nombre onclick, ahora se debe utilizar click.

A continuación, se muestran los ejemplos anteriores empleando los métodos definidos porDOM:

```
function muestraMensaje() {
  console.log("Has pulsado el ratón");
}
var elDiv =
document.getElementById("div_principal");elDiv.addEventListener("click",
muestraMensaje);

// Más adelante se decide desasociar la función al eventoelDiv.removeEventListener("click",
muestraMensaje);
```

Asociando múltiples funciones a un único evento:

```
function muestraMensaje() {
  console.log("Has pulsado el ratón");
}
```

```
function muestraOtroMensaje() {
  console.log("Has pulsado el ratón y por eso se muestran estos mensajes");
}

var elDiv =
document.getElementById("div_principal");elDiv.addEventListener("click",
muestraMensaje); elDiv.addEventListener("click", muestraOtroMensaje);
```

Si se asocia una función a un flujo de eventos determinado, esa función sólo se puededesasociar en el mismo tipo de flujo de eventos. Si se considera el siguiente ejemplo:

```
function muestraMensaje() {
  console.log("Has pulsado el ratón");
}

var elDiv =
document.getElementById("div_principal");elDiv.addEventListener("click",
muestraMensaje);

// Más adelante se decide desasociar la función al eventoelDiv.removeEventListener("click",
muestraMensaje);
```

EL OBJETO EVENT

Cuando se produce un evento, no es suficiente con asignarle una función responsable de procesar ese evento. Normalmente, la función que procesa el evento necesita informaciónrelativa al evento producido: la tecla que se ha pulsado, la posición del ratón, el elemento que ha producido el evento, etc.

El objeto event es el mecanismo definido por los navegadores para proporcionar toda esa información. Se trata de un objeto que se crea automáticamente cuando se produce un evento y que se destruye de forma automática cuando se han ejecutado todas las funcionesasignadas al evento.

El estándar DOM especifica que el objeto event es el único parámetro que se debe pasar alas funciones encargadas de procesar los eventos.

```
elDiv.onclick = function(event) {
 ...
}
```

El funcionamiento de los navegadores que siguen los estándares puede parecer "mágico",ya que en la declaración de la función se indica que tiene un parámetro, pero en la aplicación no se pasa ningún parámetro a esa función. En realidad, los navegadores que siguen los estándares crean automáticamente ese parámetro y lo pasan siempre a la función encargada de manejar el evento.

PROPIEDADES Y MÉTODOS

A pesar de que el mecanismo definido por los navegadores para el objeto event es similar,existen numerosas diferencias en cuanto a las propiedades y métodos del objeto.

PROPIEDADES DEFINIDAS POR DOM

La siguiente tabla recoge las propiedades definidas para el objeto event en los navegadoresque siguen los estándares:

Propiedad/ Método	Devuelve	Descripción
altKey	Boolean	Devuelve true si se ha pulsado la tecla ALT yfalse en otro caso
button	Númeroentero	El botón del ratón que ha sido pulsado. Posibles valores: 0 — Ningún botón pulsado 1 — Se ha pulsado el botón izquierdo2– Se ha pulsado el botón derecho3– Se pulsan a la vez el botón izquierdo y el derecho4– Se ha pulsado el botón central5– Se pulsan a la vez el botón izquierdo y el central6– Se pulsan a la vez el botón derecho yel central7` — Se pulsan a la vez los 3 botones
cancelable	Boolean	Indica si el evento se puede cancelar
charCode	Númeroentero	El código unicode del carácter correspondiente ala tecla pulsada
clientX	Númeroentero	Coordenada X de la posición del ratón respectodel

		área visible de la ventana
clientY	Númeroentero	Coordenada Y de la posición del ratón respectodel área visible de la ventana
ctrlKey	Boolean	Devuelve true si se ha pulsado la tecla CTRL yfalse en otro caso
currentTarget	ElemenEl	elemento que es el objetivo del evento
detail	Númeroentero	El número de veces que se han pulsado losbotones del ratón
isChar	BooleanIndica	si la tecla pulsada corresponde a uncarácter
keyCode	Númeroentero	Indica el código numérico de la tecla pulsada

metaKey	Númeroentero	Devuelve true si se ha pulsado la tecla META yfalse en otro caso
pageX	Númeroentero	Coordenada X de la posición del ratón respectode la página
pageY	Númeroentero	Coordenada Y de la posición del ratón respectode la página
preventDefault()	FunciónSe emplea para cancelar la acción predefinida del evento	
relatedTarget	ElemenEl elemento que es el objetivo secundario delevento (relacionado con los eventos de ratón)	
screenX	Númeroentero	Coordenada X de la posición del ratón respectode la pantalla completa

screenY	Númeroentero	Coordenada Y de la posición del ratón respectode la pantalla completa
shiftKey	Boolean	Devuelve true si se ha pulsado la tecla SHIFT yfalse en otro caso
target	ElemenEl elemento que origina el evento	
timeStamp	NúmeroLa fecha y hora en la que se ha producido elevento	
type	Cadena detexto	El nombre del evento

Al contrario de lo que sucede con Internet Explorer, la mayoría de propiedades del objeto event de DOM son de sólo lectura. En concreto, solamente las siguientes propiedades sonde lectura y escritura: altKey, button y keyCode.

La tecla META es una tecla especial que se encuentra en algunos teclados de ordenadoresmuy antiguos. Actualmente, en los ordenadores tipo PC se asimila a la tecla Alt o a la tecla de Windows, mientras que en los ordenadores tipo Mac se asimila a la tecla Command.

Prevenir el comportamiento por defecto

Una de las propiedades más interesantes es la posibilidad de impedir que se complete el comportamiento normal de un evento. En otras palabras, con JavaScript es posible no mostrar ningún carácter cuando se pulsa una tecla, no enviar un formulario después de pulsar el botón de envío, no cargar ninguna página al pulsar un enlace, etc. El método avanzado de impedir que un evento ejecute su acción asociada depende de cada navegador:

```
// Navegadores que siguen los estándaresobjetoEvento.preventDefault();
```

TIPOS DE EVENTOS

La lista completa de eventos que se pueden generar en un navegador se puede dividir encuatro grandes grupos. La especificación de DOM define los siguientes grupos:

- Eventos de ratón: se originan cuando el usuario emplea el ratón para realizaralgunas acciones.
- Eventos de teclado: se originan cuando el usuario pulsa sobre cualquier tecla desu teclado.
- Eventos HTML: se originan cuando se producen cambios en la ventana delnavegador o cuando se producen ciertas interacciones entre el cliente y el servidor.
- Eventos DOM: se originan cuando se produce un cambio en la estructura DOM dela página. También se denominan "eventos de mutación".

EVENTOS DE RATÓN

Los eventos de ratón son, con mucha diferencia, los más empleados en las aplicacionesweb. Los eventos que se incluyen en esta clasificación son los siguientes:

Evento	Descripción
click	Se produce cuando se pulsa el botón izquierdo del ratón. También se produce cuando el foco de la aplicación está situado en un botón y sepulsa la tecla ENTER
dblclick	Se produce cuando se pulsa dos veces el botón izquierdo del ratón
mousedown	Se produce cuando se pulsa cualquier botón del ratón
mouseout	Se produce cuando el puntero del ratón se encuentra en el interior deun elemento y el usuario mueve el puntero a un lugar fuera de ese elemento
mouseover	Se produce cuando el puntero del ratón se encuentra fuera de un elemento y el usuario mueve el puntero hacia un lugar en el interiordel elemento
mouseup	Se produce cuando se suelta cualquier botón del ratón que haya sido pulsado
mousemove	Se produce (de forma continua) cuando el puntero del ratón seencuentra sobre un elemento

Todos los elementos de las páginas soportan los eventos de la tabla anterior.

PROPIEDADES

El objeto event contiene las siguientes propiedades para los eventos de ratón:

- Las coordenadas del ratón (todas las coordenadas diferentes relativas a losdistintos elementos)
- La propiedad type
- La propiedad srcElement (Internet Explorer) o target (DOM)
- Las propiedades shiftKey, ctrlKey, altKey y metaKey (sólo DOM)
- La propiedad button (sólo en los eventos mousedown, mousemove, mouseout, mouseover y mouseup)

Los eventos mouseover y mouseout tienen propiedades adicionales. Internet Explorer define la propiedad fromElement, que hace referencia al elemento desde el que el punterodel ratón se ha movido y toElement que es el elemento al que el puntero del ratón se ha movido. De esta forma, en el evento mouseover, la propiedad toElement es idéntica a srcElement y en el evento mouseout, la propiedad fromElement es idéntica a srcElement.

En los navegadores que soportan el estándar DOM, solamente existe una propiedad denominada relatedTarget. En el evento mouseout, relatedTarget apunta al elemento al quese ha movido el ratón. En el evento mouseover, relatedTarget apunta al elemento desde el que se ha movido el puntero del ratón.

Cuando se pulsa un botón del ratón, la secuencia de eventos que se produce es la siguiente: mousedown, mouseup, click. Por tanto, la secuencia de eventos necesaria para llegar al doble click llega a ser tan compleja como la siguiente: mousedown, mouseup, click,mousedown, mouseup, click, dblclick.

EVENTOS DE TECLADO

Los eventos que se incluyen en esta clasificación son los siguientes:

Evento	Descripción
keydown	Se produce cuando se pulsa cualquier tecla del teclado. También seproduce de forma continua si se mantiene pulsada la tecla
keypress	Se produce cuando se pulsa una tecla correspondiente a un carácter alfanumérico (no se tienen en cuenta teclas como SHIFT, ALT, etc.). También se produce de forma continua si se mantiene pulsada la tecla
keyup	Se produce cuando se suelta cualquier tecla pulsada

PROPIEDADES

El objeto event contiene las siguientes propiedades para los eventos de teclado:

- La propiedad keyCode
- La propiedad charCode (sólo DOM)
- La propiedad srcElement (Internet Explorer) o target (DOM)
- Las propiedades shiftKey, ctrlKey, altKey y metaKey (sólo DOM)

Cuando se pulsa una tecla correspondiente a un carácter alfanumérico, se produce la siguiente secuencia de eventos: keydown, keypress, keyup. Cuando se pulsa otro tipo detecla, se produce la siguiente secuencia de eventos: keydown, keyup. Si se mantiene pulsada la tecla, en el primer caso se repiten de forma continua los eventos keydown y keypress y en el segundo caso, se repite el evento keydown de forma continua.

EVENTOS HTML

Los eventos HTML definidos se recogen en la siguiente tabla:

Evento	Descripción
load	Se prod cuando la página se carga por completo. En el elemento <img> cuando se carga por completo la imagen. En el elemento` cuando se carga el objeto
unload	Se produce en el objeto window cuando la página desaparece por completo (al cerrar la ventana del navegador por ejemplo). En el elemento <object> cuando desaparece el objeto.
abort	Se produce en un elemento <object> cuando el usuario detiene la descarga del elemento antes de que haya terminado
error	Se produce en el objeto window cuando se produce un error de JavaScript. En el elemento <img> cuando la imagen no se ha podido cargar por completo y en el elemento <object> cuando el elemento nose carga correctamente

select	Se produce cuando se seleccionan varios caracteres de un cuadro de texto (<input> y <textarea>)
change	Se produce cuando un cuadro de texto (<input> y <textarea>) pierdee foco y su contenido ha variado. También se produce cuando varía e valor de un elemento <select>
submit	Se produce cuando se pulsa sobre un botón de tipo submit (<input type="submit">)
reset	Se produce cuando se pulsa sobre un botón de tipo reset (<input type="reset">)
resize	Se produce en el objeto window cuando se redimensiona la ventanadel navegador
scroll	Se produce en cualquier elemento que tenga una barra de scroll, cuando el usuario la utiliza. El elemento <body> contiene la barra descroll de la página completa
focus	Se produce en cualquier elemento (incluido el objeto window) cuandoel elemento obtiene el foco
blur	Se produce en cualquier elemento (incluido el objeto window) cuandoel elemento pierde el foco

Uno de los eventos más utilizados es el evento load, ya todas las manipulaciones quese realizan mediante DOM requieren que la página esté cargada por completo y por tanto,el árbol DOM se haya construido completamente.

El elemento <body> define las propiedades scrollLeft y scrollTop que se pueden emplearjunto con el evento scroll.

¿Qué es JSON?

Cuando trabajamos con mucha cantidad de información, se puede volver necesario aislar elcódigo de programación de los datos. De esta forma, podemos guardar información en un fichero independiente, separado del archivo donde tenemos el código de nuestro programa.Así, si necesitamos actualizar o modificar datos, no tenemos que tocar el código de nuestroprograma.

JSON son las siglas de **JavaScript Object Notation**, y no es más que un formato ligero dedatos, con una estructura (*notación*) específica, que es totalmente compatible de forma nativa con JavaScript. Como su propio nombre indica, **JSON** se basa en la sintaxis que tiene JavaScript para crear objetos.

Un archivo **JSON** mínimo debe tener la siguiente sintaxis:

```
{

}
```

Esto simplemente es un objeto vacío. Un archivo JSON, puede contener varios tipos dedatos:

```
{
  "name": "Manz",
  "life": 99,
  "dead": false,
  "props": ["invisibility", "coding", "happymood"],
  "senses": {
    "vision": 50,
    "audition": 75,
    "taste": 40,
    "smell": 50,
    "touch": 80
  }
}
```

Como se puede ver, en **JSON** todos los textos deben estar entrecomillados con «comillasdobles», y solo se pueden utilizar tipos de datos como **string**, **number**, **object**, **array**, **boolean** o **null**. Un valor **null**, simplemente, también sería un **JSON** válido.

OJO: JSON no permite utilizar tipos de datos como **function**, **regexp** o valores **undefined**.Tampoco es válido incluir comenta

Mucho cuidado con las **comillas mal cerradas** o las **comas sobrantes** (*antes de un cierrede llaves, por ejemplo*). Suelen ser motivos de error de sintaxis frecuentemente. Si tienes dudas sobre si la

sintaxis del JSON que has construido es correcta, puedes utilizar JSONLint, una página que te permitirá pegar el código JSON y validarlo para saber si es correcto. También nos puede servir para indentar correctamente el JSON.

¿Cómo utilizar JSON?

Si analizamos bien la sintaxis de un JSON, nos daremos cuenta que es muy similar a algo alo que ya deberíamos estar acostumbrados:

```
const o = {
  name: "Manz",
  life: 99,
};
```

Simplemente añadiendo **const o =** al principio, nos daremos cuenta (*si no era evidente ya*)de que se trata de un **objeto** de Javascript y que no debería ser muy sencillo pasar de JSON a Javascript y viceversa.

En Javascript tenemos una serie de métodos que nos facilitan esa tarea, pudiendo trabajarcon **strings** que contengan JSON y objetos Javascript de forma indiferente:

Método	Descripción
JSON.parse(str)	Convierte el texto **str** (un JSON válido) a un objeto y lo devuelve.
JSON.stringify(obj)	Convierte un objeto Javascript **obj** a su representación JSON y la devuelve.

Convertir JSON a Objeto

La acción de **convertir JSON a objeto** Javascript se le suele denominar **parsear**. Es una acción que analiza una **string** que contiene un **JSON** válido y devuelve un objeto Javascript con dicha información correctamente estructurada. Para ello, utilizaremos el método JSON.parse():

```
const str = '{ "name": "Manz", "life": 99 }';

const obj = JSON.parse(str);

obj.name; // 'Manz' obj.life; // 99
```

Como se puede ver, **obj** es un objeto generado a partir del JSON recogido en la variable **str** y podemos consultar sus propiedades y trabajar con ellas sin problemas.

Convertir Objeto a JSON

La acción inversa, **convertir un objeto JavaScript a JSON** también se puede realizar fácilmente haciendo uso del método **JSON.stringify()**. Este método difícil de pronunciarviene a ser algo así

como «convertir a texto», y lo podemos utilizar para transformar un objeto de JavaScript a **JSON** rápidamente:

```
const obj = {
  name: "Manz",
  life: 99,
  saludar: function () {
    return "Hola!";
  },
};

const str = JSON.stringify(obj);

str; // '{"name":"Manz","life":99}'
```

Observa que, como habíamos dicho, las funciones no están soportadas por **JSON**, por lo que si intentamos convertir un objeto que contiene métodos o funciones, **JSON.stringify()**no fallará, pero simplemente devolverá una **string** omitiendo las propiedades que contengan funciones.

Fuente: lenguajejs.com

localStorage y sessionStorage: ¿qué son?

El objeto Storage (API de almacenamiento web) nos permite almacenar datos de maneralocal 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 lafunción 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 utilizardicho 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 variableslet 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 variableslet 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 primerodebemos 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 ymultimedia
- La información está almacenada en la computadora del cliente y NO es enviada encada 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)

Fuente: lenguajejs.com