FRONTEND I

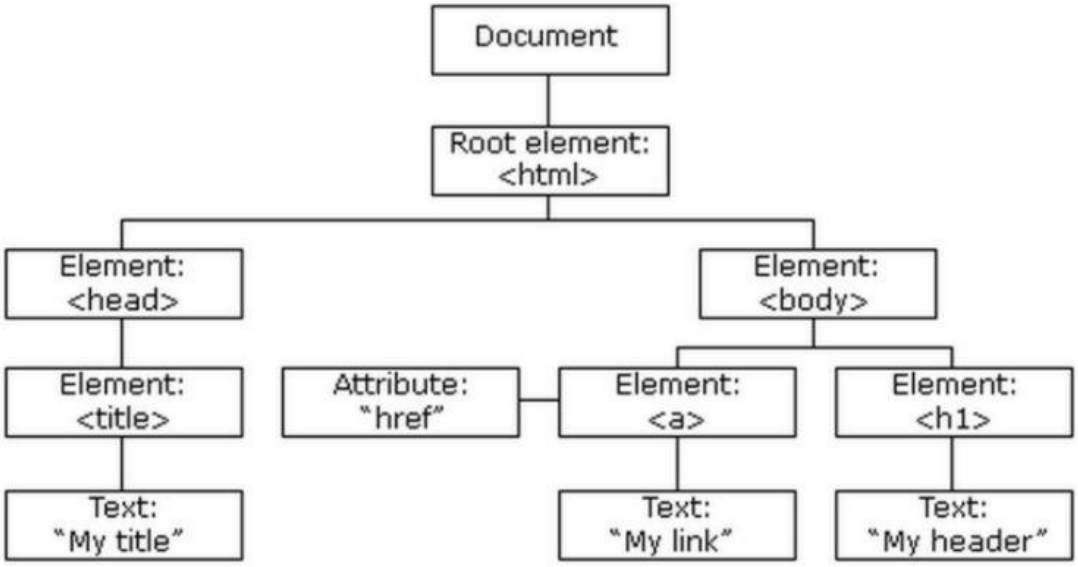
DOM

Es una interfaz de programación para documentos de HTML y XML. Facilita una representación estructurada del documento y define de que manera los programas pueden acceder a el para modificarlo, tanto su estructura como en estilo y contenido.

Da una representación del documento como un grupo de nodos y objetos estructurados que tienen propiedades y métodos. Esencialmente, conecta las paginas web a scripts o lenguajes de programación.

Una página web es un documento, el cual puede exhibirse en la ventana de un navegador o también como código fuente HTML.

El DOM es una representación completamente orientada al objeto de la pagina web y puede ser modificado con un lenguaje de script como JavaScript



El objeto **window** representa una ventana que contiene un documento **DOM**. La propiedad *document* apunta al documento DOM cargado en esa ventana.

En los navegadores el **nodo raíz** **o** **document** define el principio y final de la pagina web. Sin embargo, este no es el objeto mas alto de la jerarquía, ya que la pagina esta incluida en el navegador.

El objeto **document** depende del objeto **window** el cual es el objeto mas alto del navegador y del que dependen los demás.

El nodo raíz del **DOM** (objeto Document) es una propiedad y por tanto hijo del objeto **window**.

Nótese que hablamos aquí de objetos y no de nodos. Esto es así porque una vez que subimos un nivel encima del document ya no estamos hablando del DOM, sino de un objeto window que engloba al DOM como propiedad de este.

Dentro del **DOM** cada componente o nodo del esquema se comporta como un objeto ya que puede tener propiedades y métodos.

Casi todos los navegadores web pueden ejecutar **JavaScript**, lo que lo convierte en uno de los lenguajes de programación mas populares del mundo. Tiene una barrera de entrada baja: todo lo que necesita para programar en **JavaScript** es un editor de texto y un navegador web.

JavaScript es un lenguaje de scripting de alto nivel que se interpreta y compila en tiempo de ejecución. Esto significa que requiere un motor que es responsable de interpretar un programa y ejecutarlo.

Los motores de **JavaScript** mas comunes de encuentran en navegadores como **Firefox, Chrome o Safari**, aunque **JavaScript** puede ejecutarse sin un navegador utilizando un motor **como Google V8**.

La fiebre por entender JavaScript fue mucho más allá de lo imaginado y encontró en **Node.js** la pieza que faltaba para encajar en ciertos entornos como aplicaciones de escritorio o la gestión de redes.

Todo ello permite que JavaScript deje de ser un lenguaje exclusivo de la web para ir mucho mas allá, adentrándose incluso en el desarrollo de Internet of Things (IoT) y la Robótica.

Scripts en HTML

Para vincular un documento HTML con un documento JavaScript (que lleva toda lo lógica) se agrega en el HTML la línea:

```
<script src = "archivos.js"></script>
```

Cuando un document HTML hace referencia a una URL externa como este caso, el browser lo incluye en la página.

Es importante recordar colocar el tag **afuera** del body y cerrarlo.

¿Cómo manipular el DOM?

El primer paso para poder manipular el DOM es adquirir cierta destreza en el manejo de los selectores ya que siempre los selectores serán el primer paso para realizar operaciones de lectura o modificación del DOM

- **.getElementById()**
  Permite la selección de un elemento por su id:

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

- **.getElementsByName()**
  Permite la selección de varios elementos por su atributo name:

```
1   // <form name="miForm"></form>
2   document.getElementsByName("miForm");
```

- **.getElementsByTagName()**
  Permite la selección de varios elementos por su etiqueta.

```
// <input>
document.getElementsByTagName("input");
```

- **.getElementsByClassName()**
  Permite la selección de varios elementos por su clase.

```
// <div class="rojo"></div>
document.getElementsByClassName("rojo");
```

Existen muchas posibilidades para realizar selecciones dentro de un documento html que van mucho más allá de la clase, id, etiqueta o propiedades ya que el soporte para ello es muy bueno en todos los navegadores.

JavaScript no se queda atrás y se pueden usar **querySelector** y **querySelectorAll** que además gozan de gran soporte.

.querySelector()

Devuelve el primer elemento que coincida con el selector

```
<div id="miDiv">
    <span id="miId5" class="miClase" title="cinco"></span>
    <span id="miId4" class="miClase" title="cuatro"></span>
    <span id="miId3" class="miClase" title="tres"></span>
    <span id="miId2" class="miClase" title="dos"></span>
    <span id="miId1" class="miClase" title="uno"></span>
</div>
```

```
document.getElementById('miId1').title // uno
document.querySelector('#miDiv .miClase').title // cinco
document.querySelector('#miDiv #miId1.miClase').title // uno
document.querySelector('#miDiv .inventado').title // ERROR -> undefined
document.querySelector('#miDiv .miClase[title^=u]').title // uno
```

.querySelectorAll()

Devuelve todos los elementos que coincidan con el selector en un pseudo-array.

```
document.querySelectorAll('p') // los párrafos
document.querySelectorAll('div, img') // divs e imágenes
document.querySelectorAll('a > img') // imágenes contenidas en enlaces
```

Por lo general se debe optar por el selector que haga el trabajo con mayor claridad.

Para agregar texto a una etiqueta específica del documento HTML se utiliza. innerHTML

```
<script>
document.getElementById("p1").innerHTML="New text!";
</script>
```

Para cambiar el valor de un atributo:

```
<script>
document.getElementById("image").src="landscape.jpg";
</script>
```

Para cambiar el estilo CSS de una etiqueta:

```
<script>
document.getElementById("p2").style.color="blue";
</script>
```

Para cambiar el nombre de la clase en CSS:

```
document.getElementById('objeto').className="NombredeEstilo";
```

Eventos

Los eventos notifican a nuestro código sobre lo que sucede en nuestra pagina web. Es decir, lo que ha ejecutado el usuario.

Uno de los mayores desafíos a la hora de introducir el dinamismo en una web es la gestión de los eventos, ya que estos son asíncronos por naturaleza, por lo que no puedes saber de antemano por ejemplo cuantos segundos tardara un usuario en clickear un botón. Por ello, se ha sumado una función de propagación que hace que el dominio de los eventos sea una tarea que requiere **atención**. Básicamente podemos seleccionar un elemento HTML y suscribir uno d ellos posibles eventos de los que dispone:

- Eventos del ratón (mouse)
- Eventos del teclado
- Etc.

Una vez definido donde escuchamos y que esperamos, solo queda definir que haremos cuando esto ocurra.

Lógicamente al tratarse de asincronía debemos definir una función ya ssea en línea o reutilizable para gestionaro todo.

Si necesitamos información adicional sobre el evento sucedido, el propio sistema mandara un objeto con detalles claves sobre el evento, como argumento a la hora de ejecutar nuestro callback.

Existen dos formas básicas de añadir eventos a nuestra aplicación. Una es por medio de HTML con atributos como **onclick** y la otra desde el JavaScript, haciendo uso de metodos como **addEventListener().**

Debido a la necesidad de separar HTML, CSS y JavaScript en documentos separados al trabajar con proyectos complejos es conveniente utilizar assEventListener().

- Uso de onclick

```
<body onclick="cambiarFondo()">
  <h1>Eventos</h1>
  <script>
    function cambiarFondo() {
      let color = "red";
      document.body.style.backgroundColor = color;
      console.log("Nuevo color:", color);
    }
  </script>
</body>
```

- Uso de addEventListener()

```
<button>Click me</button>
<p>No handler here.</p>
```

```
let button = document.querySelector("button");
button.addEventListener("click", () => {
  console.log("Button clicked.");
});
```

<u>Eventos del teclado</u>

Cuando se pulsa una tecla del teclado, el navegador lanza un evento **keydown**.

Cuando se suelta, se produce un evento **keyup**.

```
<p>This page turns violet when you hold the V key.</p>
```

```
window.addEventListener("keydown", (event) => {
  if (event.key == "v") {
    document.body.style.background = "violet";
  }
});
window.addEventListener("keyup", (event) => {
  if (event.key == "v") {
    document.body.style.background = "";
  }
});
```

A pesar de su nombre, **keydown** se dispara no solo cuando la tecla se presiona físicamente hacia abajo sino también cuando se mantiene pulsada, donde el evento se dispara de nuevo cada vez que la tecla se repita. A veces hay que tener cuidado con esto.

Las teclas modificadoras como shift, control y alt general eventos de teclado igual que las teclas normales, pero al buscar la combinación de teclas también se puede averiguar si estas teclas se mantienen pulsadas mirando las propiedades **shiftKey, ctrlKey y altKey** de los eventos de teclado y mouse:

```
<p>Press Control-Space to continue.</p>
```

```
window.addEventListener("keydown", (event) => {
  if (event.key == " " && event.ctrlKey) {
    console.log("Continuing!");
  }
});
```

Eventos de mouse

Al pulsar un botón del ratón se disparan varios eventos. Los eventos **mousedown** y **mouseup** son similares a keydown y keyup y se disparan cuando el botón es presionado y liberado. Estos eventos ocurren en los nodos del DOM que están inmediatamente debajo del puntero del ratón cuando se produce el evento.

Después del evento **mouseup** se dispara un evento **click** en el nodo más especifico que contenga tanto la pulsación como la liberación del botón.

Si se producen dos clics juntos, también se producirá un evento dbclick que se dispara después del segundo evento de clic.

Cada vez que el puntero del ratón se mueve, se dispara un evento **maousemove.** Este evento puede utilizarse para seguir la posición del ratón. Una situación común en la que esto es útil es cuando se implementa alguna forma de funcionalidad de arrastre del ratón.

Cada vez que se desplaza un elemento, se dispara un evento **scroll** sobre él. Esto tiene varios usos, como saber lo que el usuario esta viendo en ese momento. El siguiente ejemplo dibuja una barra de progreso sobre el documento y actualiza que se va llenando a medida que se desplaza hacia abajo:

```
<div id="progress"></div>
```

```
document.body.appendChild(document.createTextNode(
"supercalifragilisticexpialidocious ".repeat(1000)));
let bar = document.querySelector("#progress");
window.addEventListener("scroll", () => {
let max = document.body.scrollHeight - innerHeight;
bar.style.width = `${(pageYOffset / max) * 100}%`;
});
```

Eventos por interacción con un input

Se utiliza como primer parámetro de addEventListener() la cadena “input”, la cual esta constantemente esperando las entradas en los inputs del form:

```
form.addEventListener("input", (event)=>{
    const value = event.target.value;

    if((event.target === nameClient && value.length<6)||(event.target === dniClient
&& value.length!==8)){
        event.target.classList.add("is-invalid");
    }
    else if(event.target === nameClient || event.target === dniClient){
        event.target.classList.remove("is-invalid");
    }


    if(!nameClient.value || !dniClient.value || form.querySelector(".is-invalid")){
        button.disabled = true;
    }
    else{
        button.disabled = false;
    }
});
```

Con **event.target.value** toma los valores ingresados en el input que se esta manipulando del lado del usuario en ese momento. Con **event.target** tomo la etiqueta del input que se esta manipulando en ese momento.

Esto permite verificar si los datos ingresados por el usuario son correctos.

Removiendo eventos

Un nodo puede tener un atributo onclick, por lo que solo puede resgistrar un manejador por nodo de esa manera. El método addEventListener le permite añadir cualquier numero de manejadores, por lo que es seguro añadir manejadores incluso si ya hay otro manejador en el elemento.

El método **removeEventListener**, llamado con argumentos similares a addEventListener, elimina un controlador.

```
<button>Act-once button</button>
```

```
let button = document.querySelector("button");

function once() {
  console.log("Done.");
  button.removeEventListener("click", once);
}
button.addEventListener("click", once);
```

Propagación de eventos

Para la mayoría de los tipos de eventos, los manejadores registrados en nodos con hijos también recibirán los eventos que ocurran en los hijos.

```
<div id="parent">
    <div id="children">
        Click
    </div>
</div>
```

En cualquier momento, un controlador de eventos puede llamar al método stopPropagation del objeto de evento para evitar que los manejadores que están mas adelante reciban el evento.

La mayoría de los objetos de evento tienen una propiedad de destino que se refiere al nodo donde se originaron.

Puedes usar esta propiedad para asegurarte de que no estás manejando accidentalmente que se propaga desde un nodo que no quieres manejar.

También es posible utilizar la propiedad target para lanzar una red amplia para un tipo específico de evento. Por ejemplo, si tienes un nodo que contiene una larga lista de botones puede ser más conveniente registrar un único controlador de clic en el nodo exterior y hacer que utilice la propiedad target para averiguar si se ha pulsado un botón, en lugar de registrar manejadores individuales en todos los botones.

```
<button>A</button>
<button>B</button>
<button>C</button>
```

```
document.body.addEventListener("click", (event) => {
  if (event.target.nodeName == "BUTTON") {
    console.log("Clicked", event.target.textContent);
  }
});
```

Acciones por defecto

Muchos eventos tienen una acción por defecto asociada a ellos. Si hace clic en un enlace se le llevara al objetico del enlace. Si pulsa la flecha hace abajo, el navegador desplazara la pagina hacia abajo. Si haces clic con el botón derecho, obtendrás un menú contextual. Y así más.

Para la mayoría de los tipos de eventos, los manejadores de eventos de JavaScript son llamados antes de que el comportamiento por defecto tenga lugar. Si el manejador no quiere este comportamiento normal, porque se ha encargado de manejar el evento, puede llamar al método **preventDefault** en el objeto del evento. Esto se puede utilizar para implementar tus propios atajos de teclado o menú contextual.

```
<a href="https://developer.mozilla.org/">MDN</a>
```

```
let link = document.querySelector("a");
link.addEventListener("click", (event) => {
  console.log("Nope.");
  event.preventDefault();
});
```

AppendChild

Se utiliza para agregar un nodo como hijo de otro nodo en el DOM. Específicamente, se utiliza para agregar un nodo al final de a lista de hijos de un nodo padre dado.

```
var divElement = document.getElementById("miDiv"); // Obtener el elemento div
var pElement = document.createElement("p"); // Crear un nuevo elemento p
var textNode = document.createTextNode("Este es un nuevo párrafo"); // Crear un
nodo de texto
pElement.appendChild(textNode); // Agregar el nodo de texto como hijo del
elemento p
divElement.appendChild(pElement); // Agregar el elemento p como hijo del
elemento div
```

Formularios

Los **formularios** son un método muy común de interacción con una página web.

Los formularios se componen de un elemento <form> (etiqueta HMTL) que contiene controles de formulario como campos de entrada, menús de selección y botones. Estos campos de entrada pueden rellenarse con información que se procesa una vez que el formulario ha sido enviado.

Tradicionalmente, cuando se enviaba un **formulario**, se enviaba a un **servidor** en el que la información se procesaba mediante un sistema de gestión. La información se procesaba utilizando un lenguaje de fondo como PHP o Ruby.

Es posible, y cada vez más común, procesar la información en un **formulario** en el **front-end** antes de que se envíe al servidor usando JavaScript.

Un **formulario** puede ser enviado manualmente por el usuario empleando un botón o elemento de entrada con un atributo type de submit, o incluso un elemento input con un atributo type de imagen:

```
<button type='submit'>Submit</button>
<input type='submit' value='Submit'>
<input type='image' src='button.png'>
```

El método **form.reset()** devolverá a todos los controles del formulario sus valores iniciales especificados en el HTML. Un botón con un atributo **type of reset** también puede ser utilizado para hacer esto sin la necesidad de scripts adicionales:

```
<button type='reset'>Reset</button>
```

Tip: Los “reset buttons” no suelen brindar buena usabilidad ya que pueden ser accionados sin querer y borrar toda la data ingresada en el formulario, por lo cual se recomienda ser cuidadosos con su uso.

Eventos en formularios

El **evento focus** se utiliza cuando se enfoca un elemento. Por ejemplo, el caso de un <input> esto sucede cuando el cursor se sitúa en el interior del elemento, ya sea clickeando, tocando o navegando a través del teclado.

Podemos ver el siguiente ejemplo:

```
const input = form.elements.searchInput;

 input.addEventListener('focus', () => alert('focused'),
↪ false);
```

Cuando abramos el formulario en el browser y coloquemos el cursor en el input veremos el cuadro con la alerta.

Para el envio de formularios se utiliza el **evento submit**. Normalmente esto enviara el contenido del formulario al servidor para procesar, pero podemos usar JavaScript para interceptar el formulario antes de que se envíe añadiendo un receptor de eventos de envío.

```
const form = document.forms['search'];
form.addEventListener ('submit', search, false);

function search() {
alert(' Form Submitted');
}
```

Cargará la página especificada en el atributo action de form

Luego de enviar el formulario y aceptar la alerta, el navegador intentara cargar una página inexistente (la URL debe terminar en algo similar a …/search?searchInput=hello). Esto se debe a que cuando el evento se disparó, nuestra función search() fue invocada, mostrando el dialogo de alerta. Luego, el formulario se envió a la URL proporcionada en el atributo action para su procesamiento, pero en este caso la URL no es una URL real, por lo que no va a ninguna parte.

Podemos evitar que el formulario se envíe a esa URL utilizando el método *preventDefault()*.

```
function search(event) {
alert('Form Submitted');
event.preventDefault();
}
```

Los objetos de elementos de entrada de texto (inputs de type text) tienen una propiedad **value** que puede utilizarse para recuperar el texto del campo. Podemos usar esto para informar de lo que el usuario ha buscado en un campo de búsqueda, por ejemplo:

```
function search(event) {
alert(`You Searched for: ${input.value}`);
event.preventDefault();
}
```

<u><select> en formularios</u>

Consideraciones:

- El atributo selected puede utilizarse para establecer el valor inicial en el HTML.
- El atributo name se utiliza para acceder al elemento select en JavaScript como una propiedad del objeto form. Si solo se ha seleccionado un elemento, esto devolverá una referencia a esa selección, en caso contrario, se devolverá una colección que contiende cada selección.

- Cada objeto de elección tiene una propiedad value que es igual al atributo value de la etiqueta <option> que fue seleccionada.

```
<select name="tipoCliente">
            <option value="min">Minorista</option>
            <option value="may">Mayorista</option>
        </select>
```

```
const form = document.querySelector("form");
const tipo = document.createElement("p");

const formData = new FormData(form);
const data = Object.fromEntries(formData);
tipo.innerHTML="Tipo de cliente: "+data.tipoCliente; //tendrá valor min o may
```

Validaciones en formularios

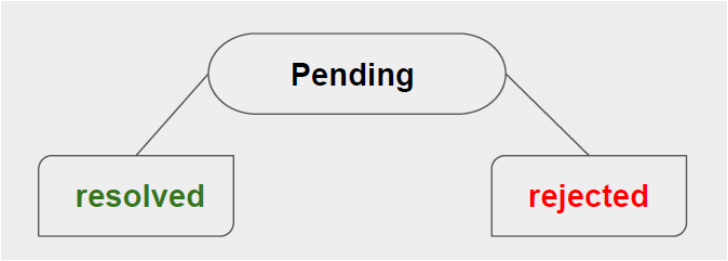
Podemos realizar dos tipos de validaciones básicas para los formularios y poder enviarlos con la información completa.

- OnSubmit: es la más básica pero que quizás no ofrece la mejor experiencia al usuario. Una vez que se ingresaron los datos, al tratar de enviar el formulario, en ese momento se le informará si hay algún error o falta un campo por completar, lo cual no es recomendable, pero sin embargo suele usarse aun y nos servirá para poder practicar las validaciones básicas.

- OnTime: da una respuesta instantánea en lugar de esperar a que se envíe el formulario. Esto se puede conseguir añadiendo el evento listener directamente al campo de entrada que se dispara cuando el usuario presione una tecla (utilizando el evento keyup). La retroalimentación puede entonces ser insertada dentro del elemento label del campo de entrada, junto con una clase de error para una respuesta más directa.

Asincronía en JavaScript

La programación asíncrona en JavaScript utiliza promesas en vez de callbacks, debido a que una promesa es no bloqueante y eventualmente devolverá un resultado o un error a la persona que llama a través de la promesa.



- Si una función asíncrona no ha completado su tarea, la promesa que ha devuelto estará en estado pendiente. Cuando la función asíncrona complete con éxito, pondrá la promesa en el estado resuelto. En este momento, la promesa genera o emite el resultado pasado a través de ella – nos referimos a esto como promesa resulta.
- Si una función asíncrona termina con un error, entonces pone la promesa en el estado rechazado. En este momento, la promesa genera o emite el error pasado a través de ella - nos referimos a esto como promesa rechazada.

Una promesa puede ser creada en uno de estos 3 estados, sin embargo, resolved y rejected son estados finales.

Funciones async

La función async y await se introdujo para mantener la estructura del código idéntica entre el código sincrónico y el asíncrono. Esto no afecta a la forma en que escribimos las funciones asíncronas, pero cambia en gran medida la forma en que las usamos.

Hay 2 reglas para utilizar esta función:

- Para poder utilizar la función asincrónica como si fuera una función síncrona, colocar la palabra **async** delante de la función asíncrona que devuelve la promesa.
- Para llamar a la función asíncrona como su fuera una función síncrona, coloque la palabra clave **await** justo delante de la llamada. La palabra clave await puede utilizarse solo dentro de las funciones marcadas como asíncronas.

Fetch

La API Fetch proporciona una interfaz JavaScript para acceder y manipular partes del canal HTTP, tales como peticiones y respuestas. También posee un método global fetch() que proporciona una forma fácil y lógica de obtener recursos de forma asíncrona por la red.

```
async function getProducts(){
    const result = await fetch('https://fakestoreapi.com/products');
    const data = await result.json();
    return data;
}
```

Object Location

Este objeto, el cual puede utilizarse sin el profeijo window, nos sirve tanto para obtener la url o parte de ella, de la web donde nos encontramos para redireccionarnos hacia otra pagina.

- Obtener la url de la pagina

```
window.location.href
```

- Obtener el nombre del hosting

```
window.location.hostname
```

- Obtener el protocolo

```
window.location.protocol
```

- Redireccionando con location

```
window.location=”dirección donde queremos ir”
```

Local Storage

Es una característica de los navegadores web que permite a las aplicaciones web almacenar datos de manera persistente en el dispositivo del usuario. Estos datos se guardan en el navegador y permanecen allí incluso después de que se cierre la ventana del navegador o se apague el dispositivo. Esto permite que las aplicaciones web guarden información como preferencias del usuario, configuraciones, datos de sesión, entre otros, y accedan a ellos rápidamente en futuras visitas al sitio web.

- **Guardar en el local storage:**
  Solo se almacenan strings. Por lo tanto, si queremos guardar un ovejo debemos pasarlo antes por JSON.stringify(elObjeto) y almacenarlo:

  ```
  localStorage.setItem(“nombre”, JSON.stringify(elObjeto));
  ```

- **Obtener del local storage:**
  Para obtenerlo Podemos acceder con

  ```
  localStorage.getItem(“nombre”,propiedad);
  ```

  o si es un objeto

  ```
  JSON.parse(localStorage.getItem(“nombre”));
  ```

- **Eliminar elemento:**

  ```
  localStorage.removeItem(“nombre”);
  ```

Session Storage

La propiedad sessionStorage es similar a localStorage, la única diferencia es que la información almacenada en localStorage no posee tiempo de expiración, por el contrario la información almacenada en sessionStorage es *eliminada al finalizar la sesión de la página*.

La sesión de la pagina perdura mientras el navegador se encuentra abierto y se mantiene por sobre la recargas y reaperturas de la página. Abrir una pagina en una nueva pestaña o ventana iniciara una nueva sesión, lo que difiere en la forma en que trabajan las cookies de sesión.

- Guardar en la sessionStorage

```
sessionStorage.setItem('key', 'value');
```

- Obtener de la sessionStorage

```
Let data = sessionStorage.getItem('key');
```