Fecha 27/04/2024
Subgrupo N. 1 y 2
Nota: [ ]

PROGRAMACION WEB - JAVASCRIPT PARTE II

Abstract
JavaScript es un lenguaje de programación o scripting que posibilita la creación de funcionalidades dinámicas en páginas web. Cuando una página va más allá de mostrar contenido estático y ofrece actualizaciones en tiempo real, mapas interactivos, animaciones, reproducción de vídeo y más, es muy probable que JavaScript esté siendo utilizado.

1. Recibir funciones por parámetro

Significa escribir funciones que reciban otras funciones como parámetro. Revisar el siguiente ejemplo:

```
function porCadaUno(arr, fn) {
    for (const el of arr) {
        fn(el)
    }
}
```

Supongamos que quiero recorrer un array y hacer algo con cada uno de sus elementos.

Esta función recibe un array por primer parámetro y una función por el segundo. Recorre el array y, por cada elemento, hace un llamado a la función mencionada enviando dicho elemento por parámetro.

```
const numeros= [1, 2, 3, 4]

porCadaUno(numeros, console.log)
// 1
// 2
// 3
// 4
```

Enviando console.log por parámetro, se ejecuta esa función con cada elemento del array. Podemos enviar funciones diferentes en distintos llamados y ejecutar distintas acciones sobre los elementos del array, todo con una misma función.

```
let total = 0

function acumular(num) {
    total += num
}

porCadaUno(numeros, acumular)
console.log(total) // 10
```

2. Arrow Function

Es usual definir la función directamente sobre el parámetro como una función anónima, aprovechando la sintaxis de arrow function. Esto permite definir acciones más dinámicas.

```
const duplicado = []

porCadaUno(numeros, (el)=> {
    duplicado.push(el * 2)
})

console.log(duplicado) // [2, 4, 6, 8]
```

Para recordar: Javascript incorpora nativamente varias funciones de orden superior. Existen métodos para operar sobre arrays que trabajan con esta lógica. Los siguientes, funcionan siempre iterando sobre el array correspondiente. Reciben una función por parámetro, la cual recibe a la vez el elemento del array que se está iterando.

EJERCICIO EN CLASE

Revisar e implementar TODOS LOS EJEMPLOS DE AQUI EN ADELANTE

```
const porCadaUno=(arr, fn)=>{
    for (const el of arr) {
        fn(el);
    }
}

const precios=[120,300,140, 400]

porCadaUno(precios,
(valor)=>{console.log(valor);});

porCadaUno(precios, (valor)=>{
    const resultado=valor*10;
    console.log(resultado);
});
```

3. Métodos de búsqueda y transformación

Cada uno de estos métodos están pensados para solucionar problemas recurrentes con los arrays.

Métodos de búsqueda y transformación

- **ForEach:** El método forEach() es similar al ejemplo porCadaUno anterior. Itera sobre el array y por cada elemento ejecuta la función que enviemos por parámetro, la cual recibe a su vez el elemento del array que se está recorriendo:

```
const numeros = [1, 2, 3, 4, 5, 6]

numeros.forEach( (num)=> {
    console.log(num)
} )
```

- **Find:** El método find() recibe una función de comparación por parámetro. Captura el elemento que se está recorriendo y retorna true o false según la comparación ejecutada. El método retorna el primer elemento que cumpla con esa condición:

  Nótese que el find() retorna el primer elemento del array que cumpla con la condición enviada, de ahí que podemos almacenarlo en una variable o usarlo de referencia para otro proceso. Si no hay ninguna coincidencia en el array, el método find retorna undefined.

```
const cursos = [
    {nombre: 'Javascript', precio: 15000},
    {nombre: 'ReactJS', precio: 22000},
]

const resultado = cursos.find((el) => el.nombre === "ReactJS")
const resultado2 = cursos.find((el) => el.nombre === "DW")

console.log(resultado) // {nombre: 'ReactJS', precio: 22000}
console.log(resultado2) // undefined
```

- **Filter:** El método filter() recibe, al igual que find(), una función comparadora por parámetro, y retorna un nuevo array con todos los elementos que cumplan esa condición. Si no hay coincidencias, retornará un array vacío.

```
const cursos = [
    {nombre: 'Javascript', precio: 15000},
    {nombre: 'ReactJS', precio: 22000},
    {nombre: 'AngularJS', precio: 22000},
    {nombre: 'Desarrollo Web', precio: 16000},
]

const resultado = cursos.filter((el) => el.nombre.includes('JS'))
const resultado2 = cursos.filter((el) => el.precio < 14000)

console.log(resultado)
/* [
    {nombre: 'ReactJS', precio: 22000},
    {nombre: 'Angular', precio: 22000}
] */

console.log(resultado2) // []
```

- **Some:** El método some() funciona igual que el find() recibiendo una función de búsqueda. En vez de retornar el elemento encontrado, some() retorna true o false según el resultado de la iteración de búsqueda.

```
console.log( cursos.some((el) => el.nombre == "Desarrollo
Web"))
// true
console.log( cursos.some((el) => el.nombre == "VueJS"))
// false
```

- **Map:** El método map() crea un nuevo array con todos los elementos del original transformados según las operaciones de la función enviada por parámetro. Tiene la misma cantidad de elementos pero los almacenados son el return de la función:

```
const cursos = [
    {nombre: 'Javascript', precio: 15000},
    {nombre: 'ReactJS', precio: 22000},
    {nombre: 'AngularJS', precio: 22000},
    {nombre: 'Desarrollo Web', precio: 16000},
]

const nombres = cursos.map((el) => el.nombre)
console.log(nombres)
// [ 'Javascript', 'ReactJS', 'AngularJS', 'Desarrollo
Web' ]
```

  En el ejemplo, la función retorna la propiedad nombre de cada elemento y eso es lo que se almacena en el nuevo array nombres. Map() se utiliza mucho para transformación de arrays.

Si quisiera aumentar el precio de todos los cursos en este ejemplo, puedo mapear y retornar una copia de los elementos con el precio modificado:

```
const actualizado = cursos.map((el) => {
    return {
        nombre: el.nombre,
        precio: el.precio * 1.25
    }
})

console.log(actualizado)
/* [
     { nombre: 'Javascript', precio: 18750 },
     { nombre: 'ReactJS', precio: 27500 },
     { nombre: 'AngularJS', precio: 27500 },
     { nombre: 'Desarrollo Web', precio: 20000 }
] */
```

- **Reduce:** El método reduce() nos permite obtener un único valor tras iterar sobre el array. Funciona como un método que resume el array a un único valor de retorno.

  Ejemplos de aplicación: i) Cuando queremos acumular la suma de alguna propiedad de los elementos, ii) O cuando deseamos obtener algún resultado general sobre todo el array.

  A diferencia de los métodos anteriores, el método reduce recibe dos parámetros.

  1. El primero es la función que ordena qué queremos resumir del array. Recibe un parámetro que funciona como acumulador, y el elemento del array que iteramos.
  2. El segundo es el valor inicial del acumulador.

  En este ejemplo, en el acumulador sumamos cada elemento del array y al terminar la iteración nos devuelve ese resultado. El segundo parámetro del reduce, que aquí se ve como 0, es el valor inicial del acumulador.

  Con este caso podría, pensando por ejemplo en un simulacro de compra, sumar el precio de todos los productos elegidos:

```
const numeros = [1, 2, 3, 4, 5, 6]
const total = numeros.reduce((acumulador, elemento) => acumulador + elemento, 0)

console.log(total) // 21
```

- **Sort:** El método sort() nos permite reordenar un array según un criterio que definamos. Recibe una función de comparación por parámetro que, a la vez, recibe dos elementos del array. La función retorna un valor numérico (1, -1, 0) que indica qué elemento se posiciona antes o después. Este método es destructivo, es decir, modifica el array sobre el cual se llama.

  Para ordenar números, basta con restar uno al otro, y el orden indica si será ordenado de forma ascendente o descendente:

```
const numeros = [ 40, 1, 5, 200 ];
numeros.sort((a, b) => a - b);  // [ 1, 5, 40, 200 ]
numeros.sort((a, b) => b - a);  // [ 200, 40, 5, 1 ]
```

Para ordenar un array por algún string, debemos definir una función comparadora que retorne un valor numérico de referencia, según queramos el orden ascendente o descendente:

```
const items = [
    { name: 'Pikachu', price: 21 },
    { name: 'Charmander', price: 37 },
    { name: 'Pidgey', price: 45 },
    { name: 'Squirtle', price: 60 }
  ]
items.sort((a, b) => {
    if (a.name > b.name) {
        return 1;
    }
    if (a.name < b.name) {
        return -1;
    }
    // a es igual a b
    return 0;
})
```

Veamos a continuación un ejemplo aplicado, donde se utilizan los métodos presentados en una solución más aproximada al escenario una tienda virtual.

```
const productos = [{ id: 1,  producto: "Arroz", precio: 125 },
                   { id: 2,  producto: "Fideo", precio: 70 },
                   { id: 3,  producto: "Pan"  , precio: 50},
                   { id: 4,  producto: "Flan" , precio: 100}]

const buscado = productos.find(producto => producto.id === 3)
console.log(buscado) //{id: 3, producto: "Pan", precio: 50}

const existe = productos.some(producto => producto.nombre === "Harina")
console.log(existe ) // false

const baratos = productos.filter(producto => producto.precio < 100)
console.log(baratos)
// [{id: 2,producto:"Fideo",precio:70},{id:3,producto:"Pan",precio: 50}]

const listaNombres = productos.map(producto => producto.nombre)
console.log(listaNombres);
//["Arroz", "Fideo", "Pan", "Flan"]
```

4. Funciones de Orden Superior 2

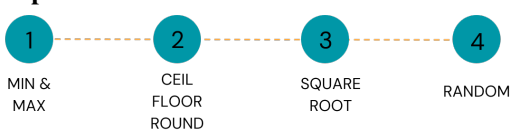
- **Objeto Math:** Javascript provee el objeto Math que funciona como un contenedor de herramientas y métodos para realizar operaciones matemáticas.

  El objeto Math contiene una serie de métodos que nos permiten realizar algunas operaciones matemáticas más complejas.

  Veremos a continuación algunas de las funciones que se desprenden de este objeto, aunque el repertorio completo lo pueden ver en su documentación aquí

```
console.log( Math.E ) // 2.718281828459045
console.log( Math.PI ) // 3.141592653589793
```

Propiedades



Los métodos de Math.min() y Math.max() reciben una serie de argumentos numéricos y devuelven aquel de valor máximo o mínimo, según corresponda:

```
console.log( Math.max(55, 13, 0, -25, 93, 4) ) // 93
console.log( Math.min(55, 13, 0, -25, 93, 4) ) // -25
```

También se pueden referenciar los valores de infinito positivo o negativo a través de la variable global Infinity, de tipo number:

```
console.log( Math.max(55, Infinity, 0, -25, 93, 4) ) // Infinity
console.log( Math.min(55, 13, 0, -Infinity, 93, 4) ) // -Infinity
```

- **Random:** El método Math.random() genera un número pseudo-aleatorio entre 0 y 1, siendo el 0 límite inclusivo y el 1 exclusivo.

```
console.log( Math.random() ) // 0.6609867980868442
console.log( Math.random() ) // 0.09291446900104305
console.log( Math.random() ) // 0.6597817047013095
```

Para generar números aleatorios dentro de un rango deseado, distinto de 0-1, podemos multiplicar su resultado por el rango esperado. A la vez podemos sumar el límite inferior si lo necesitamos:

```
// números entre 0 y 10
console.log( Math.random() * 10 )
// números entre 0 y 50
console.log( Math.random() * 50)
// números entre 20 y 50
console.log( Math.random() * 30 + 20 )
```

En el último ejemplo quiero generar números entre 20 y 50. Por eso, el rango de números es de 30 a partir del número 20 (límite inferior adicionado). Pero todos los números siguen conteniendo una larga serie de decimales.

Esto se suele combinar con las funciones de redondeo para obtener números enteros aleatoriamente, que suelen ser de uso más común.

Al usar Math.round, esta función retornará números aleatorios en el rango de 0-100 inclusive. Si usara Math.ceil los números irían de 1 a 100, ya que siempre redondeará hacia arriba; y si usa Math.floor el rango sería de 0 a 99.

- **Date:** Instanciar un objeto Date nos genera la fecha y tiempo actual:

```
console.log( new Date() )
```

La convención con la que trabaja Javascript para construir fechas cuenta los meses a partir del 0 (0 = enero, 11 = diciembre) y los días a partir del 1:

```
console.log(new Date(2020, 1, 15))
// Sat Feb 15 2020 00:00:00 GMT-0300 (hora estándar de Argentina)

const casiNavidad = new Date(2021, 11, 25, 23, 59, 59)
console.log(casiNavidad)
// Sat Dec 25 2021 23:59:59 GMT-0300 (hora estándar de Argentina)
```

El constructor de la clase Date nos permite crear objetos date con fechas diferentes. Puede recibir parámetros en el orden año, mes, día, hora, minutos, segundos, milisegundos (todos tipo number).

```
const casiNavidad = new Date("December 25, 2021 23:59:59")
console.log(casiNavidad)
// Sat Dec 25 2021 23:59:59 GMT-0300 (hora estándar de Argentina)
```

Valor singular: Instanciado un objeto Date, podemos aplicar diferentes métodos para devolver determinados valores de una fecha:

`getMonth()`: Nos retornará el number que representa el mes (0 y 11)

`get FullYear()`: Nos devolverá el number que representa el año creado

`getDay()`: Nos retornará el number que representa el día creado (1=lunes, 7=domingo)

Los resultados de las diferencias entre fechas se generan en milisegundos.

Si quisiera calcular la diferencia de días entre dos fechas habría que generar cálculos adicionales sobre esta diferencia en milisegundos.

Por suerte, existen librerías que solucionan estos problemas de forma eficiente y rápida, pero las trabajaremos en clases posteriores.

5. ¿Qué es el Modelo de objetos del documento (DOM) y cómo funciona?

El Modelo de Objetos del Documento (DOM) es una estructura de objetos generada por el navegador, la cual representa la página HTML actual.

Con JavaScript la empleamos para acceder y modificar de forma dinámica elementos de la interfaz.

Es decir que, por ejemplo, desde JavaScript podemos modificar el texto contenido de una etiqueta ¡h1¿.

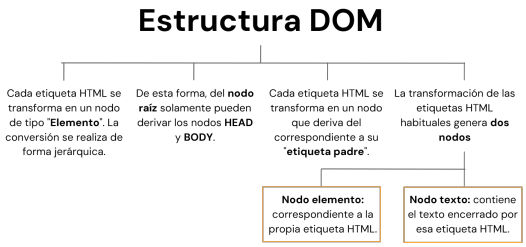
La estructura de un documento HTML son las etiquetas.

En el Modelo de Objetos del Documento (DOM), cada etiqueta HTML es un objeto, al que podemos llamar nodo.

Las etiquetas anidadas son llamadas “nodos hijos” de la etiqueta “nodo padre” que las contiene.

Todos estos objetos son accesibles empleando JavaScript mediante el objeto global document.

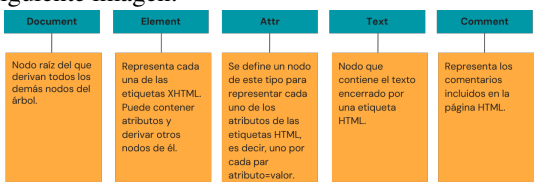
Por ejemplo, document.body es el nodo que representa la etiqueta `body`.



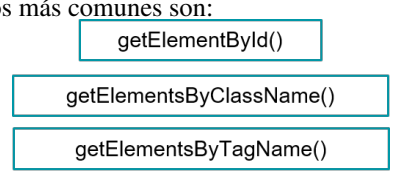
Para editar el DOM desde el navegador, mediante los navegadores modernos se puede, solamente hay que acceder a las Herramientas para desarrolladores en inspeccionar y

en la pestaña ”Elements”. Para revisar un video que amplíe el conocimiento, por favor haga clic aquí ya que si bien la estructura DOM está simplificada, es un medio muy útil para verificar y probar actualizaciones en la estructura.

Además, la especificación completa de DOM define 12 tipos de nodos, en donde, los más utilizados se presentan en la siguiente imagen:



Por otra parte, Existen distintos métodos para acceder a los elementos del DOM empleando en la clase Document. Los más comunes son:



- **getElementById:** El método getElementById() sirve para acceder a un elemento de la estructura HTML, utilizando su atributo ID como identificación.

```
//CODIGO HTML DE REFERENCIA
<div id = "app">
    <p id = "parrafo1" >Hola Mundo</p>
</div>

//CODIGO JS
let div     = document.getElementById("app");
let parrafo = document.getElementById("parrafo1");
console.log(div.innerHTML);
console.log(parrafo.innerHTML);
```

- **getElementsByClassName:** El método getElementsByClassName() sirve para acceder a un conjunto de elementos de la estructura HTML, utilizando su atributo class como identificación. Se retornará un Array de elementos con todas las coincidencias:

```
//CODIGO HTML DE REFERENCIA
 <ul>
     <li class="paises">AR</li>
     <li class="paises">CL</li>
         <li class="paises">UY</li>
 </ul>
//CODIGO JS
let paises = document.getElementsByClassName("paises");
console.log(paises[0].innerHTML);
console.log(paises[1].innerHTML);
console.log(paises[2].innerHTML);
```

- **getElementsByTagName:** El método getElementsByTagName() sirve para acceder a un conjunto de elementos de la estructura HTML, utilizando su nombre de etiqueta como identificación. Esta opción es la menos específica de todas, ya que es muy probable que las etiquetas se repitan en el código HTML.

```
//CODIGO HTML DE REFERENCIA
<div>
    <div>CONTENEDOR 2</div>
    <div>CONTENEDOR 3</div>
</div>
//CODIGO JS
let contenedores = document.getElementsByTagName("div");
console.log(contenedores[0].innerHTML);
console.log(contenedores[1].innerHTML);
console.log(contenedores[2].innerHTML);
```

MODIFICAR NODOS

- **innerText:** La propiedad innerText de un nodo nos permite modificar su nodo de texto. Es decir, acceder y/o modificar el contenido textual de algún elemento del DOM.

```
//CODIGO HTML DE REFERENCIA
<h1 id=”titulo”>Hola Mundo!</h1>
//CODIGO                                                    JS
let titulo = document.getElementById("titulo")
console.log( titulo.innerText ) // “Hola Mundo!”
// cambio el contenido del elemento
titulo.innerText = “Hola Coder!”
console.log( titulo.innerText ) // “Hola Coder!”
```

- **innerHTML:** Permite definir el código html interno del elemento seleccionado. El navegador lo interpreta como código HTML y no como contenido de texto, permitiendo desde un string crear una nueva estructura de etiquetas y contenido.

  Al pasar un string con formato de etiquetas html y contenido a través de la propiedad innerHTML, el navegador genera nuevos nodos con su contenido dentro del elemento seleccionado.

```
//CODIGO HTML DE REFERENCIA
<div id=”contenedor”></div>
//CODIGO                                                    JS
let container = document.getElementById("contenedor")
// cambio el código HTML interno
container.innerHTML = “<h2>Hola mundo!</h2><p>Lorem ipsum</p>”

//Resultado en el DOM
<div id=”contenedor”>
    <h2>Hola mundo!</h2>
    <p>Lorem ipsum</p>
</div>
```

- **className:** A través de la propiedad className de algún nodo seleccionado podemos acceder al atributo class del mismo y definir cuáles van a ser sus clases:

```
//CODIGO HTML DE REFERENCIA
<div id=”contenedor”></div>
//CODIGO JS
let container =
document.getElementById("contenedor")
// cambio el código HTML interno
container.innerHTML = “<h2>Hola mundo!</h2>”
// cambio el atributo class
container.className = “container row”
//Resultado en el DOM
<div id=”contenedor” class=“container row”>
    <h2>Hola mundo!</h2>
</div>
```

AGREGAR O QUITAR NODOS

- **createElement:** Para crear elementos se utiliza la función document.createElement(), y se debe indicar el nombre de etiqueta HTML que representará ese elemento. Luego debe agregarse como hijo el nodo creado con append(), al body o a otro nodo del documento actual.

```
// Crear nodo de tipo Elemento, etiqueta p
let parrafo = document.createElement("p");
// Insertar HTML interno
parrafo.innerHTML = "<h2>¡Hola Coder!</h2>";
// Añadir el nodo Element como hijo de body
document.body.append(parrafo);
```

- **remove:** Se pueden eliminar nodos existentes y nuevos. El método remove() permite eliminar un nodo seleccionado del DOM:

```
let parrafo = document.getElementById("parrafo1");
//Elminando el propio elemento
parrafo.remove();

let paises = document.getElementsByClassName("paises");
//Eliminando el primer elemento de clase paises
paises[0].remove()
```

OBTENER DATOS DE INPUTS

- **value:** Para obtener o modificar datos de un formulario HTML desde JS, podemos hacerlo mediante el DOM. Accediendo a la propiedad value de cada input seleccionado:

```
//CODIGO HTML DE REFERENCIA
  <input id = "nombre" type="text">
  <input id = "edad"   type="number">

//CODIGO JS
document.getElementById("nombre").value = "HOMERO";
document.getElementById("edad").value   = 39;
```

6. PLANTILLAS LITERALES

En versiones anteriores a ES6, solía emplearse la concatenación para incluir valores de las variables en una cadena de caracteres (string). Esta forma puede ser poco legible ante un gran número de referencias. En JS ES6 que solventa esta situación son los template strings.

```
let producto = { id: 1,  nombre: "Arroz", precio: 125 };
let concatenado = "ID : " + producto.id +" - Producto: " + producto.nombre + "$
"+producto.precio;
let plantilla   = `ID: ${producto.id} - Producto ${producto.nombre} $
${producto.precio}`;
//El valor es idéntico pero la construcción de la plantilla es más sencilla
console.log(concatenado);
console.log(plantilla);
```

PLANTILLAS LITERALES E innerHTML : La plantillas son un medio para incluir variables en la estructura HTML de nodos nuevos o existentes , modificando el innerHTML.

```
let producto   = { id: 1,  nombre: "Arroz", precio: 125 };
let contenedor = document.createElement("div");
//Definimos el innerHTML del elemento con una plantilla de texto
contenedor.innerHTML = `<h3> ID: ${producto.id}</h3>
                        <p>  Producto: ${producto.nombre}</p>
                        <b> $ ${producto.precio}</b>`;
//Agregamos el contenedor creado al body
document.body.appendChild(contenedor);
```

7. EVENTOS EN JS

Los eventos son la manera que tenemos en Javascript de controlar las acciones de los usuarios, y definir un comportamiento de la página o aplicación cuando se produzcan. Con Javascript es posible definir qué sucede cuando se produce un evento, por ejemplo, cuando se realiza un clic en cierto elemento o se inserta un texto en un determinado campo.

JavaScript permite asignar una función a cada uno de los eventos. Reciben el nombre de event handlers o manejadores de eventos. Así, ante cada evento, JavaScript asigna y ejecuta la función asociada al mismo.

Hay que entender que los eventos suceden constantemente en el navegador. JavaScript lo que nos permite hacer es escuchar eventos sobre elementos seleccionados.

Cuando escuchamos el evento que esperamos, se ejecuta la función que definimos en respuesta.

A esta escucha se la denomina event listener.

¿Cómo definir eventos en JS?

OPCION1: El método addEventListener() permite definir qué evento escuchar sobre cualquier elemento seleccionado. El primer parámetro corresponde al nombre del evento y el segundo a la función de respuesta.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Mi primer App</title>
  </head>
  <body>
    <h2>Programación Web</h2>
    <button id="btnPrincipal">CLICK</button>
    <script>
      let boton = document.getElementById("btnPrincipal")
      boton.addEventListener("click", respuestaClick)
      function respuestaClick(){
        console.log("Respuesta evento");
      }
    </script>
  </body>
</html>
```

OPCION2: Emplear una propiedad del nodo para definir la respuesta al evento. Las propiedades se identifican con el nombre del evento y el prefijo on. También es posible emplear funciones anónimas para definir los manejadores de eventos.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Mi primer App</title>
  </head>
  <body>
    <h2>Programación Web</h2>
    <button id="btnPrincipal">CLICK</button>
    <script>
      let boton = document.getElementById("btnPrincipal")
      boton.onclick = () =>{console.log("Respuesta 2")}
    </script>
  </body>
</html>
```

OPCION3: SINTAXIS

Determinar el evento especificando el manejador de evento en el atributo de una etiqueta HTML. La denominación del atributo es idéntica al de la propiedad de la opción 2 (prefijo on). La función puede declararse entre la comillas o bien tomarse una referencia existen en el script.

```
<input type="button" value="CLICK2" onclick="alert('Respuesta 3');"
                                    />
```

Las opciones 1 y 2 son las recomendadas.

Si bien se pueden presentar casos de aplicación específicos (por ejemplo, en la opción 1 el nombre del evento puede venir de una variable al usar la propiedad, y esto no puede hacerse en la 2), se identifican como formas de definición de eventos equivalentes.

La opción 3, aunque es de fácil implementación, no es

recomendada para proyectos en producción.

No se considera buena práctica declarar funciones y código JavaScript dentro del HTML.

EVENTOS DEL MOUSE

- mousedown/mouseup: Se oprime/suelta el botón del ratón sobre un elemento.
- mouseover/mouseout: El puntero del mouse se mueve sobre/sale del elemento.
- mousemove: El movimiento del mouse sobre el elemento activa el evento.
- click: Se activa después de mousedown o mouseup sobre un elemento válido.

```
//CODIGO HTML DE REFERENCIA
<button id="btnMain">CLICK</button>
//CODIGO JS
let boton = document.getElementById("btnMain")
boton.onclick = () => {console.log("Click")}
boton.onmousemove = () => {console.log("Move")}
```

EVENTOS DEL TECLADO

Se producen por la interacción del usuario con el teclado.

Entre ellos se destacarán los que se encuentran a continuación.

- keydown: Cuando se presiona una tecla.
- keyup: Cuando se suelta una tecla.

```
//CODIGO HTML DE REFERENCIA
<input id = "nombre" type="text">
<input id = "edad"   type="number">
//CODIGO JS
let input1 = document.getElementById("nombre")
let input2 = document.getElementById("edad")
input1.onkeyup = () => {console.log("keyUp")}
input2.onkeydown = () =>
{console.log("keyDown")}
```

- change: El evento change se activa cuando se detecta un cambio en el valor del elemento. Por ejemplo, mientras se escribe en un input de tipo texto no hay evento change, pero cuando se pasa a otra sección de la aplicación entonces sí ocurre.

```
//CODIGO HTML DE REFERENCIA
<input id = "nombre" type="text">
<input id = "edad"   type="number">
//CODIGO JS
let input1  =
document.getElementById("nombre");
let input2  = document.getElementById("edad");
input1.onchange = () =>
{console.log("valor1")};
input2.onchange = () =>
{console.log("valor2")};
```

- input: Si queremos ejecutar una función cada vez que se tipea sobre el campo, conviene trabajar directamente con el evento input.

```
//CODIGO HTML DE REFERENCIA
<input id = "nombre" type="text">

//CODIGO JS
let input1  = document.getElementById("nombre")
input1.addEventListener('input', () => {
     console.log(input1.value)
})
```

- submit: El evento submit se activa cuando el formulario es enviado. Normalmente se utiliza para validar el formulario antes de ser enviado al servidor o bien para abortar el envío y procesarlo con JavaScript.

```
//CODIGO HTML DE REFERENCIA
 <form id="formulario">
      <input type="text">
      <input type="number">
      <input type="submit" value="Enviar">
 </form>
//CODIGO JS
let miFormulario      =
document.getElementById("formulario");
miFormulario.addEventListener("submit", validarFormulario);

function validarFormulario(e){
    e.preventDefault();
    console.log("Formulario Enviado");
}
```

OTROS EVENTOS

Existen otros eventos que podemos utilizar.

Algunos son eventos estándar definidos en las especificaciones oficiales, mientras que otros son eventos usados internamente por navegadores específicos.

La forma de declararlos es similar a lo abordado en esta clase, lo que necesitamos aprender es bajo qué condición se disparan los eventos que buscamos implementar.

Para conocer más eventos se recomienda verificar la referencia de eventos en la documentación aquí.

8. EJERCICIO

PARTE 1 Realizar un pequeño cuestionario dinámico, en éste lo que se hará es pedir al usuario que configure un formulario, en donde, el usuario responderá con un número de preguntas. A continuación, según el número que éste elija, saldrá un pequeño aviso que diga: Tipo de pregunta, en donde el usuario podrá seleccionar: - Pregunta de texto, - Pregunta de Verdadero/Falso, -Opción Múltiple. Dependiendo el tipo de pregunta que elija, se irá configurando el cuestionario, en donde se dará un cuadro de texto cuando quiera una respuesta textual, radio buttons o checkbox, y un Botón que diga ”Aceptar”. Una vez que el usuario final presione ese botón, se mostrarán en un cuadro de texto las respuestas del cuestionario previamente configurado.

PARTE 2 Revisar detenidamente todos y cada uno de los eventos de aquí