JAVASCRIPT FRONTEND AVANZADO
CLASE 1 : JAVASCRIPT

INTRODUCCIÓN

Qué es y para qué nos sirve Javascript?

JavaScript (**JS**) es un lenguaje ligero e interpretado, orientado a objetos con **funciones de primera clase**, más conocido como el lenguaje de script para páginas web, pero también usado en muchos entornos sin navegador, tales como **node.js o Apache CouchDB.** Es un lenguaje script multi-paradigma, **basado en prototipos**, dinámico, soporta estilos de programación funcional, orientada a objetos e imperativa. [1]

Front-end vs Back-end

Front-end y back-end son términos que se refieren a la separación de intereses entre una capa de presentación y una capa de acceso a datos, respectivamente. Pueden traducirse al español el primero como interfaz, frontal final o frontal y el segundo como motor, dorsal finalo zaga, aunque es común dejar estos por separado.
En diseño de software el *front-end* es la parte del software que interactúa con los usuarios y el *back-end* es la parte que procesa la entrada desde el *front-end*. La separación del sistema en *front-ends* y *back-ends* es un tipo de abstracción que ayuda a mantener las diferentes partes del sistema separadas. La idea general es que el front-end sea el responsable de recolectar los datos de entrada del usuario, que pueden ser de muchas y variadas formas, y los transforma ajustándolos a las especificaciones que demanda el back-end para poder procesarlos, devolviendo generalmente una respuesta que el front-end recibe y expone al usuario de una forma entendible para este. La conexión del front-end y el back-end es un tipo de interfaz. [2]

ECMAScript

ECMAScript es una especificación de lenguaje de programación publicada por ECMA International. El desarrollo empezó en 1996 y estuvo basado en el popular lenguaje JavaScript propuesto como estándar por Netscape Communications Corporation.

ECMAScript define un lenguaje de tipos dinámicos ligeramente inspirado en Java y otros lenguajes del estilo de C. Soporta algunas características de la programación orientada a objetos mediante objetos basados en prototipos y pseudoclases.

Desde el lanzamiento en junio de 1997 del estándar **ECMAScript** 1, han existido las versiones 2, 3 y 5, que es la más usada actualmente (la 4 se abandonó). En junio de 2015 se cerró y publicó la versión ECMAScript 6. [3]

WEB APIs

WebAPI es un término usado para referirse al conjunto de APIs compatibles y de acceso a los dispositivos que permite a las Web apps y contenido acceder al hardware del dispositivo (como el estado de la batería o la vibración de hardware), al igual que acceso a información almacenada en el dispositivo (como el calendario o la lista de contactos). Agregando estas APIs, esperamos expandir lo que la Web puede hacer hoy y solo plataformas propietarias fueron capaces de hacer en el pasado. [4]
Javascript posee una cantidad bastante amplia de WebAPIs (las cuales pueden ser revisadas a través del siguiente link https://developer.mozilla.org/en-US/docs/Web/API) y entre todas componen por completo el lenguaje como se ejecuta del lado del navegador del cliente.

TIPOS DE DATOS

El último estándar ECMAScript define siete tipos de datos : [5]

- **Primitivos**
  - **Boolean** : true y false.
  - **null :** Una palabra clave especial que denota un valor null. Como JavaScript es case-sensitive, null no es lo mismo que null, NULL, o cualquier otra variante.
  - **Undefined** : Una propiedad de alto nivel cuyo valor no es definido.
  - **Number** : 42 o 3.14159.
  - **String** : "Hola" , 'hola', `hola`
  - **Symbol** (nuevo en ECMAScript 6).
- **Object**

- **Object :** { indice : valor } Son matrices ordenadas de manera asociativa de un valor con un índice.
- **Array :** [valor] Son matrices ordenadas de manera secuencial numérica ascendente.
- **Function :** Son objetos con la habilidad de poder ser ejecutados.
- etc...

CONSTRUCTORES

Para construir variables lo tradicional es que usemos el constructor 'var'. A partir de ECMAScript 6 aparecieron dos nuevas formas de declarar una variable : let y const

VAR

Las variables inicializadas con el constructor 'var' cumplen con los siguientes tres puntos :

- Admiten redeclaración : Es decir que puedo volver a declarar una variable que ya había sido inicializada:

```
var a = true
var a = false
```

- Admiten redefinición : Es decir que puedo cambiarle su valor en cualquier momento del programa :

```
var a = true
a = false
```

- Admite alcance global : Es decir que salvo por los bloques funcionales, una variable declarada siempre es global, por lo tanto es una propiedad del objeto global window.

```
if (true) {
   var a = true
}
console.log(a)//true
```

LET

Las variables inicializadas con el constructor ‘vlet’ cumplen con los siguientes tres puntos :

- No admiten redeclaración : Es decir que no puedo volver a declarar una variable que ya había sido inicializada:

```
let a = true
let a = false //Error!!!
```

- Admiten redefinición : Es decir que puedo cambiarle su valor en cualquier momento del programa :

```
let a = true
a = false
```

- No admiten alcance global : Es decir las variables declaradas pertenecen únicamente al bloque donde se crearon. Por otro lado, si las mismas fueran globales, lo serán pero no como propiedad de window.

```
if (true) {
   let a = true
}
console.log(a)//Error!!!
```

CONST

Las variables inicializadas con el constructor ‘const’ cumplen con los siguientes tres puntos :

- No admiten redeclaración : Es decir que no puedo volver a declarar una variable que ya había sido inicializada:

```
const a = true
const a = false //Error!!!
```

- No admiten redefinición : Es decir que no puedo cambiarle su valor

```
var a = true
a = false //Error!!!
```

- No admiten alcance global :Es decir las variables declaradas pertenecen únicamente al bloque donde se crearon.

```
if (true) {
   const a = true
}
console.log(a)//Error!!!
```

- Admiten redefinición de componentes internos

```
const obj = {
   x : 1,
   y : 2
}

obj.x = 10 // {x:10,y:2}
```

NOTACIÓN DE PUNTO vs CORCHETE

En Javascript contamos con dos tipos de notación para poder ingresar a los datos internos de una matriz cualquiera. A diferencia de otros lenguajes en donde se suelen tener notación de sintaxis específicas para cada tipo de dato, acá ambas son para todos los tipos de

objetos con la diferencia en que una nos permite acceder a más cosas que la otra.Observemos el ejemplo desde el siguiente objeto :

```
const obj = {x:1,y:2,0:true}
```

Notación de corchete

- **Índice Number :** Puedo ingresar a un elemento usando obj[0]

```
obj[0]//true
```

- **Indice String :** Puedo ingresar a un elemento usando obj[“x”]

```
obj["x"]//1
```

- **Índice variable :** Puedo ingresar a un elemento usando una variable

```
const indice = "x"
obj[indice]//1
```

Notación de punto

- **Índice Number :** No puedo ingresar a un elemento usando un numero obj.0

```
obj.0 //Error : Unexpected Number
```

- **Índice String** : Puedo ingresar a un elemento usando obj.x

```
obj.x // 1
```

- **Índice variable :** No puedo ingresar a un elemento usando una variable como índice.

```
const indice = "x"
obj.indice//undefined
```

ITERACIONES

Además de contar con las tradicionales iteraciones de cualquier construcción de lenguaje como for, while, do...while , en Javascript contamos con otras iteraciones especializadas en determinadas entidades :

```
Array.forEach(Function callback(Any element?,Number index?))
```

El método forEach se encuentra disponible en el prototipo de todos los arrays y algunos otros elementos que no son de la clase arrays per se pero aún así implementan esta interfaz. Toma un parámetro y es una función callback sincrónica que se ejecuta por cada elemento que exista dentro del array. El callback a su vez recibe por iteración al elemento por el cual está iterando y su índice dentro del array que lo contiene.

```
for(let i in obj){}
```

El bucle for...in es ideal para recorrer una matriz asociativa, osea un objeto literal cualquiera.

FUNCIONES FLECHA

La **expresión de función flecha** tiene una sintaxis más corta que una expresión de función convencional y no vincula sus propios this, arguments, super, o new.target. Las funciones flecha siempre son anónimas. Estas funciones son funciones no relacionadas con métodos y no pueden ser usadas como constructores. [10]

```
var foo = function(){
    console.log("Foo!")
}

//Puede también expresarse como :
var foo = () => { console.log ("Foo!") }
```

Las funciones flecha tienen muchas variantes para poderlas escribir, las cuales vamos a ir viendo a lo largo del curso [11]. Por el momento vamos a nombrar una muy útil :

```
var foo = function(unParametro){
    console.log("Foo!")
}

//Si la función flecha solo recibe un parámetro puede
expresarse como :
var foo = unParametro => { console.log ("Foo!") }
```

BOM y DOM

El **BOM** ó **Browser Object Model** es la forma en la que Javascript representa al navegador en formato JSON [6]. Es un objeto como cualquier otro con la particularidad de que es global, es decir que puede ser accedido en cualquier parte del programa usando su nombre **window**. Es tan global, que si quisiéramos usar alguna de las propiedades o métodos que tiene adentro, ni siquiera necesitamos mencionarlo para usarlas. Por ejemplo cada vez que usamos nuestra función log :

```
console.log("Soy Global!")

//Nunca decimos

window.console.log("Yo tambien soy global!")
```

De hecho, es tan global que todas las variables que escribimos sueltas en nuestros archivos van a ir a parar, a menos que les digamos lo contrario, a window! :

```
var a = true
console.log(a)//true
console.log(window.a)//true
```

Dentro de este objeto tenemos muchas propiedades que vamos a ir viendo a lo largo del curso pero para ir mencionando algunas que podríamos llegar a usar para escribir front-end podríamos mencionar cosas como :

```
window.innerHeight // Alto del documento abierto
window.innerWidth // Ancho del documento abierto
//Los inner*  no toman en cuenta la consola de
desarrollo, barras de scroll, barra de navegación, etc.
window.outerHeight // Alto de todo el navegador
window.outerWidth // Ancho de todo el navegador
//Los outer* toman en cuenta absolutamente todo en el
navegador
window.location.href // Variable que controla la
direccion de la barra de navegacion
window.console // Objeto interfaz de la API Console. En
el tenemos la popular función log
window.alert // Para notificaciones de alerta
window.confirm // Para ventanas de confirmación
window.prompt // Para ventanas de ingreso de texto
window.history // Para revisar el estado de historial de
navegación del cliente
window.document // Para acceder a información relativa
del documento abierto
```

El **DOM** ó **Document Object Model** es la forma en la que Javascript representa HTML en formato JSON [6]. Este no es un objeto global, pero si pertenece a uno (window). El Web API del DOM nos permite manipular nuestro HTML de la manera que queramos ya que dentro de su interfaz expuesta en la propiedad window.document tenemos una gran cantidad de métodos para llevar a cabo esta tarea :

Crear un nodo [7]

```
document.createElement(tagName[, options]) => Element Object{}
```

La función document.createElement() crea un elemento HTML especificado por su `tagName`, o un `HTMLUnknownElement` si su `tagName` no se reconoce. Por lo tanto podríamos obtener una nueva etiqueta escribiendo algo como lo siguiente :

```
let h1 = document.createElement("h1")
```

Seleccionar un nodo existente

```
document.getElementById(id);
document.getElementsByClassName(names);
Element.getElementsByTagName(name);
Element.querySelector(selectors[8]);
Element.querySelectorAll(selectors);
```

Los métodos de getElementById y getElementsByTagName están desde el comienzo del lenguaje prácticamente por lo que no necesitamos un fallback para usarlos en navegadores muy viejos como IE6+. Todos los demás se agregaron a partir de IE8+ [9].

Tener en cuenta que si optamos por usar los métodos nuevos, tenemos las ventaja de poder usar selectores de nodos mucho más avanzados que solo por ID o por nombre de etiqueta HTML.

Editar un nodo

Cada nodo, desde document en adelante, va a ser una referencia de alguna etiqueta de HTML y por consiguiente, los atributos de HTML se ven reflejados como las propiedades de cada nodo cada vez que seleccionamos o creamos uno nuevo; además existe otro lugar en donde se almacenan los atributos de HTML con su valor inicial y es la propiedad .attributes de cada nodo. La mayoría de los atributos de HTML se inicializan en cada nodo con el mismo nombre aunque algunos tienen un nombre distinto por cuestiones de nombres reservados. Entonces si quisiera modificar el atributo ID de HTML de algún nodo podría :

```
let h1 = document.createElement("h1")
h1.id = "titulo"
console.log(h1) // <h1 id="titulo"></h1>
```

Hay propiedades que nos presentan alguna complejidad extra a veces . Podríamos mencionar la de el atributo CLASS y DATA-* . Si uno quisiera manipular las clases de una etiqueta de HTML tendría que usar la propiedad **.className** la cual guarda un string con el o las clases registradas que tenga esa etiqueta hasta ese momento. Por lo tanto, agregar o remover clases re volvería una tarea demasiado tediosa para realizar a mano. Por lo que Javascript nos brinda facilidades como por ejemplo :

```
h1.classList.add("clase1")
h1.classList.remove("clase1")
h1.classList.toggle("clase1")
```

De esta forma tenemos una manera más conveniente de manipular clases. De esta misma forma también podemos observar el caso de los atributos data :

```
<h1 data-type="title">Titulo</h1>
```

```
let h1 = document.createElement("h1")
console.log(h1.dataset)
h1.dataset.type = "nuevoTipo"
```

Para poder manipular este tipo de atributos también tenemos una interfaz JSON más funcional.

Agregar / Remover Nodos del DOM

Cada nodo por si solo tiene la habilidad de incorporar y remover nodos de su propio interior, es decir que todos van a tener un método que corresponde a cada operación :

```
element.appendChild(Child);
```

Este método nos permite agregar un nodo cualquiera dentro de otro que hayamos seleccionado previamente. Por ejemplo, si quisiéramos agregar una nueva etiqueta que creamos a una que ya teníamos escrita en HTML podríamos hacer lo siguiente :

En el HTML tendría una etiqueta con las siguientes características :

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

Y luego en mi script :

```
const cont = document.getElementById("cont")
const p = document.createElement("p")
p.innerText = "Lorem Ipsum"
cont.appendChild(p)
```

Si tuviera que agregar muchos elementos al mismo tiempo solo puedo hacerlos uno por uno, no admite ingreso de múltiples nodos en simultáneo.

Por otro lado si quisiera remover un nodo existente podría realizar algo como esto en el HTML :

```
<div id="cont">
   <p>lorem ipsum</p>
</div>
```

Y algo como esto en el script :

```
const p = document.querySelector("p")
const parent = p.parentNode
parent.removeChild(p)
```

1. https://developer.mozilla.org/es/docs/Web/JavaScript
2. https://es.wikipedia.org/wiki/Front-end_y_back-end
3. https://es.wikipedia.org/wiki/ECMAScript
4. https://developer.mozilla.org/es/docs/WebAPI
5. https://developer.mozilla.org/es/docs/Web/JavaScript/Data_structures
6. JSON : Javascript Object Notation
7. Nodo : Es una etiqueta de HTML pero referenciada desde el punto de vista de Javascript
8. https://drafts.csswg.org/selectors/
9. https://caniuse.com/#search=getelements
10. https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Funciones/Arrow_functions