RESUMEN GENERAL MODULO II – JAVASCRIPT

Escritura JS:
Utilizamos por el momento Javascript “Vanilla”, es decir, escribimos código JS plano, sin ningún Framework o herramienta extra.

Inserción de código:
Para ejecutar el código JS, lo embebemos dentro del HTML entre etiquetas <script> </script>, o bien utilizamos la etiqueta <script type=”module” src=”archivo.js”> para insertar un archivo JS externo.

Importación / exportación de módulos:
Para organizar mejor nuestro código en diferentes archivos, podemos exportar diferentes elementos desde un archivo con export, e importarlos en otro con import, ej:
En archivo1.js: export const var01 = 23;
En archivo2.js: import {var01} from ‘./archivo2.js’;

Constantes y variables:
Definimos constantes con la palabra reservada const, y variables con let, no utilizamos más var:
const K = 23;
let var01 = 23;

Ambito de variables (scope):
Una variable definida dentro de un bloque específico de código (como una función por ejemplo), es local y solo accesible dentro de ese ámbito; otra definida en el flujo principal de código, es global. Siempre que podamos, evitaremos variables globales.

Tipos de datos básicos:

```
// String
let cadena1 = ‘Carlos’;

// Number
let nro1 = 34;

// Boolean
let activo = true;

// Undefined
let var01;
let var01 = null;
```

Operadores:

// Asignación (=)
let numero = 24;

// Aritméticos (+, -, *, /, %)
let suma = numero1 + numero2;
% es operador de módulo, retorna el resto de la división entera
4 % 2 (es cero), 4 % 3 (es uno).

// Unitarios (++, --, !)
let valor = 23;
valor++; // valor pasa a 24
valor--; // valor pasa a 22

let activo = true;
activo = !activo; // activo pasa a false

// Relacionales
== igual a
=== igual a (incluyendo tipo de dato)
!= distinto a
> mayor a
< menor a
>= mayor o igual a
<= menor o igual a

// Lógicos
&& compuerta Y lógica (una condición Y la otra)
|| compuerta O lógica (una condición O la otra)
! compuerta negación (NOT)

// Spread
… (3 puntos), utilizado en arrays, para copiar y otras operaciones, ver docs.

Cadenas de caracteres:

A mayúsculas: cadena.toUpperCase();
A minúsculas: cadena.toLowerCase();
Longitud: cadena.length;
Una letra: cadena.charAt(1); -> obtiene segunda legra
Concatenación actualizada con Template strings:
let cr1 = “mi”;
let template = `Este es ${cr1} template string`; // notar comillas invertidas
Convertir string a entero: parseInt(cadena);
Convertir string a coma flotante: parseFloat(cadena);

Números:

Entero: let entero = 23;
Coma flotante: let decinal = 23.331;

Redondeo a entero más cercano: Math.round(decimal);
Redondeo de decimales: let redondeo = decimal.toFixed(2); // 23.33
Elevar a la potencia: Math.pow(numero, potencia);
Raíz cuadrada: Math.sqrt(numero);
Encontrar el máximo: Math.max(numeros_separados_por_comas);
Encontrar el mínimo: Math.min(numeros_separados_por_comas);
Generar número al azar entre 0 y 100: Math.floor(Math.random() * 100);

Condiciones:

```
if (condicion) {
	// hacer si es verdadera
} else {
	// hacer si es falsa
}
```

Condición compuesta AND (una y la otra): if (condicion1 && condicion2)
Condición compuesta OR (una o la otra): if (condicion1 || condicion2)
Condición negada: if (!condicion)
Notación con operador ternario: condicion ? // hacer si es verdadera : // hacer si es falsa;
Condición de igualdad común: if (elemento == valor)
Condición de igualdad de valor y tipo de dato: if (elemento === valor)

Switch case: puede emplearse como sintaxis alternativa a una secuencia de ifs.

```
let indicador = 1;

switch (indicador) {
	case 0:
		break;
	case 1:
		// Se ejecuta en este caso
		break;
	default:
		// se ejecuta si no es ninguno de los anteriores

}
```

Ciclos:

```
// For
for (let var01 = 0; var01 < 10; var01++) {
	// Estas instrucciones se ejecutan 10 veces (0 a 9)
}

// While
while (condicion) {
	// Estas instrucciones se ejecutan mientras condición sea verdadera
};
```

```
// Do while
do {
        // Estas instrucciones se ejecutan mientras condición sea verdadera,
        // do{} garantiza que sí o sí se ejecutarán al menos una vez, ya que
        // la condición se evalúa al final
} while (condicion);
```

Arrays (corchetes):

const / let lecturas = [23, 22, 21, 22, 23, 24];

Un array puede contener cualquier tipo de objeto, y mezclar distintos tipos.

// Recorrer un array:
// Con for: for (let i = 0; i < array.legth; i++) { console.log(array[i]); }
// Alternativa: for (let item of lecturas) { console.log(item); }
// Con forEach: array.forEach((item) => { console.log(item); })
(forEach no retorna un nuevo array)
// Con map: const array2 = array.map((item, index) => { item[index] = item[index] * 3; })
(map retorna un nuevo array, array2 en este caso tendrá los elementos de array multiplicados por 3. array queda sin modificar).

// Acceder a un elemento: array[indice]; // indice comienza desde 0
// Agregar un elemento al final del array: array.push(elemento);
// Agregar un elemento al principio del array: array.unshift(elemento);
// Quitar el último elemento del array: array.pop();
// Quitar el primer elemento del array: array.unshift();
// Retornar parte de un array: array.slice(desde_indice, hasta_indice);
// Reemplazar elementos: array.splice();
// Encontrar índice de elemento: array.indexOf(elemento);
// Encontrar máximo: Math.max(…array);
// Encontrar mínimo: Math.min(…array);
(recordar que … es el operador spread).
// Ordenar: array.sort();

Objetos (llaves):

```
const objeto = {
        id: 1,
        nombre: ‘Carlos’,
        saldo: 1000
}
```

Un objeto se compone de pares nombre: valor, separados por comas, podemos por supuesto mezclar distintos tipos de valores, y también realizar arrays de objetos:

```
const array_objetos = [
        { id: 1, nombre: ‘Carlos’, saldo: 1000 },
        { id: 2, nombre: ‘Carolina’, saldo: 2000 },
]
```

```
// Convertir objeto a cadena json: JSON.stringify(objeto);
// Convertir cadena JSON a objeto JS manejable: JSON.parse(cadena_json);

// Recorrer array de objetos:
array_objetos.map((item) => { console.log(item.nombre) });

// Acceso a propiedades de objeto, por notación de puntos:
objeto.nombre, objeto.saldo, etc

// Notación alternativa:
objeto['nombre'], objeto['saldo'], etc.
```

Funciones (function):

```
function nombre_funcion (argumentos) {
        // Estas instrucciones se ejecutarán cada vez que "llamemos" a la función
        // utilizando su nombre en algún lugar del código)
        return retorno;
}
```

Una función es un bloque de código encapsulado. Cada vez que lo necesitemos, podremos "invocarlo" utilizando su nombre.

Los argumentos son parámetros que puede recibir la función, separados por comas. Estos parámetros se transforman en variables internas de la función:

```
function triplicar (valor) {
        console.log(valor * 3);
}
```

return permite retornar un valor desde la función, ej:

```
function triplicar (valor) {
        return valor * 3;
}
```

const calculo = triplicar(2); // en este caso le pasamos un argumento con valor 2 a triplicar, y calculo recibe el valor 6, retornado por la función.

Función anónima: sintaxis alternativa que se utiliza para asignar su resultado a un elemento o para ser pasada como argumento a otra función, ej:

```
const saludar = function () { console.log('Hola') };
```

Arrow function: notación más compacta para declarar funciones, en la mayoría de los casos equivalente al uso de function tradicional (ver docs por detalles específicos).

```
const saludar = () => { console.log('Hola') };
```

Callback: función anónima pasada como argumento, que es ejecutada por el sistema de forma automática al sucederse un determinado evento.

```
setTimeout(() => {
        console.log('Pasaron 3 segundos');
}, 3000);
```

En este caso utilizando la notación arrow, pasamos como primer argumento a timeout una función, y como segundo el valor 3000. Al cumplirse el timeout, se dispara el evento que ejecuta el contenido de la función, mostrando el mensaje 'Pasaron 3 segundos'.

Clases (Class):

Una clase es sintéticamente un molde, que define un paquete de características relacionadas a un objeto. Estas características pueden ser tanto propiedades (valores) como métodos (acciones).

Luego de definir una clase, podemos crear (instanciar) tantos objetos de ese tipo como necesitemos:

```
Class Persona {
        constructor (apellido, nombre, saldo) {
                this.apellido = apellido;
                this.nombre = nombre;
                this.saldo = saldo;
        }

        mostrarNombreCompleto () {
                console.log(`${this.apellido}, ${this.nombre}`);
        }
}
```

Los métodos no son más que funciones declaradas dentro de la clase. El método llamado constructor, se ejecuta de forma automática cada vez que creamos una nueva instancia de la clase, es decir, cada vez que utilizamos el molde para generar un nuevo objeto de ese tipo. La instancia se crea con new:

```
const persona1 = new Persona('Perren', 'Carlos', 1000);
const persona2 = new Persona('Ferrero', 'Carolina', 2000);
etc
```

En este caso el constructor espera 3 parámetros (apellido, nombre y saldo) y los inicializa internamente según los valores pasados.

La palabra reservada **this** hace referencia a la instancia, es decir, utiliza el paquete de valores que corresponda. Por ejemplo, cuando se utilice en los métodos de la clase this.saldo, para el caso de persona1 será 1000, y para persona2, 2000.

BOM (Browser Object Model) / DOM (Document Object Model):

Es una estructura jerárquica de objetos disponible en todo navegador. Desde JS podemos manejarla para modificar dinámicamente el contenido mostrado al usuario. Leer detalle en https://www.w3schools.com/js/js_window.asp.

Enlazar un elemento HTML con una variable JS:
En el archivo HTML: Contenido inicial
En el archivo JS:
const parrafo1 = document.getElementById('parrafo1');

A partir de ahora, podemos controlar el comportamiento del elemento <p> desde JS, por ejemplo cambiar su contenido:
parrafo1.innerHTML = 'Nuevo contenido';

(Ver docs por otros métodos para relacionar elementos del DOM, como querySelector).

localStorage / sessionStorage:

Un mecanismo muy sencillo para almacenar datos del lado del navegador. Ambos utilizan la misma sintaxis, sessionStorage borra su contenido al cerrar la ventana del navegador, localStorage no, permanece hasta que sea borrado manualmente o desde código.

// Guardar en localStorage
localStorage.setItem('nombre_para_guardarlo', contenido);
contenido es un string, si tomamos un objeto de JS, lo pasamos como JSON.stringify(contenido).

// Recuperar de localStorage
localStorage.getItem('nombre_con_el_que_se_guardo');

Atención!: este NO ES un mecanismo seguro para almacenar datos sensibles. Si bien lo utilizamos por ejemplo para un login, lo hacemos a título ilustrativo ya que aún no operamos con bases de datos. Tener presente este punto para futuros desarrollos.

Eventos (events):

Es un concepto esencial en la programación actual. Un evento es un suceso que se dará en determinado momento, pero no sabemos exactamente cuándo. En función de esto, debemos activar una "escucha" (listener) sobre este evento, para estar atentos a cuando suceda y ejecutar lo que corresponda.

// Escuchar evento utilizando atributos embebidos de HTML:
<button onclick="controlarIngreso()">Ingresar</button>

El atributo onclick nos permite activar la escucha del evento click (en este caso sobre un botón, pero puede ser cualquier elemento del DOM). Cuando el usuario haga click sobre el botón, se ejecutará la función controlarIngreso(), por supuesto, deberá haber sido definida previamente en JS.

// Activar escucha de evento desde JS:
// En el archivo HTML: <button id="btn_ingresar">Ingresar</button>
// En el archivo JS:
const btn_ingresar = document.getElementById('btn_ingresar');

btn_ingresar.addEventListener('click', function () { console.log('Se pulsó el botón'); })

El primer parámetro pasado a addEventListener es el tipo de evento (click) y el segundo, una función anónima con el paquete de instrucciones a ejecutar cuando se suceda el evento.

Detalle eventos DOM: https://www.w3schools.com/jsref/dom_obj_event.asp.

Síncrono vs asíncrono:

La programación actual utiliza grandes cantidades de operaciones a nivel de archivos y tráfico de red. Estas operaciones son por naturaleza más lentas que solicitudes manejadas completamente en memoria, y por lo tanto van agregando demoras en los procesos, que deben manejarse de manera inteligente.

Síncrono: hace referencia a procesos que se ejecutan en orden, debe terminar el anterior para que pueda iniciar el siguiente. Si un “eslabón” en la cadena se demora, esa demora es acarreada; si directamente no responde, la secuencia general queda bloqueada hasta que se agote el tiempo de espera (timeout).

Asíncrono: hace referencia a procesos que pueden iniciarse en simultáneo. El sistema habilita un listener por cada uno, que queda al aguardo de la respuesta, mientras libera rápidamente los recursos para que el flujo principal pueda seguir su ejecución. A medida que las respuestas van llegando, los listeners ejecutan el paquete de instrucciones que hemos indicado para cada una.

Este método es por supuesto más eficiente, permitiendo ahorro de tiempo y mejorando la experiencia del usuario al momento de visitar nuestro sitio.

Javascript permite desde hace muchos años el manejo asíncrono de peticiones a través del objeto XMLHttpRequest y los callbacks. Más acá en el tiempo, incorporó el mecanismo de Promesas y el Async / Await. Las distintas opciones conviven.

Promesas:

Una promesa es esencialmente un **compromiso a futuro**. Cuando solicitamos por ejemplo un contenido remoto utilizando el mecanismo de promesas, el sistema se compromete a entregarnos el resultado ni bien lo reciba (porque sabemos que tardará, aunque sea unos pocos milisegundos), es decir, queda atento a esa conexión y libera el flujo principal para que nuestro código pueda continuar con otras tareas.

// Generar promesa: utilizamos new Promise(). Recibe una función anónima con dos argumentos (resolve y reject). Con resolve podemos retornar lo que necesitemos si la promesa se cumple, y con reject lo que sea si no se cumple.

Uso de promesas / fetch:

Si bien hay casos en los cuales nos resultará útil generar promesas por nosotros mismos, en muchos simplemente utilizaremos otros elementos ya diseñados para trabajar con el mecanismo de promesas, como **fetch**. **fetch** nos permite recuperar un contenido remoto:

```
const recuperar_info = () => {
        fetch(url_remota)
        .then((datos) => { return datos.json(); })
        .then((datos) => { console.log(datos); })
        .catch((error) => { console.log(error.message); })
}
```

Siempre que veamos then() y catch(), estaremos utilizando promesas. then() recibe como argumento una función que se ejecuta cuando la promesa se cumple; catch() una que se ejecuta si no se cumple. Vemos en este caso 2 then() seguidos, es habitual en fetch() ya que el primero recupera los datos recibidos y los convierte a un objeto json (lo cual genera a su vez una nueva promesa, por eso el segundo then()), y en el segundo es donde realmente tenemos los datos disponibles y listos para usar como objeto JS.

Opcionalmente podemos colocar al final un tercer bloque .finally(), lo que pongamos dentro se ejecutará sin importar si la promesa se cumple o no.

async / await:

No es más que una sintaxis alternativa a then() y catch(), es decir, seguimos utilizando el mismo mecanismo de promesas, solo que escribimos una sintaxis más limpia, y por lo general más compacta que utilizando la alternativa tradicional. El mismo ejemplo anterior se podría escribir como:

```
const recuperar_info = async () => {
        let datos = await fetch(url_remota);
        datos = await datos.json();
        console.log(datos);
}
```

Vemos que aplicamos un modificador **async** a la función, para indicarle al sistema que se trata de un proceso asíncrono, y delante de cada instrucción en la que debamos esperar por el resultado, agregamos **await**.