# Introducción

Desarrollo de Sistemas Web - Back End

Conociendo el entorno

Versión 1.3 F. Act 21/08/2024

# Node.JS

Node.js es un entorno de tiempo de ejecución de JavaScript construido sobre el motor V8 de Google Chrome. A diferencia de las librerías y frameworks que mencioné anteriormente, Node.js no se utiliza directamente para construir interfaces de usuario en el navegador, sino que se utiliza para ejecutar código JavaScript en el lado del servidor.

Node.js permite a los desarrolladores crear aplicaciones de servidor altamente escalables y de alto rendimiento. Una de las características clave de Node.js es su modelo de operaciones asincrónicas y sin bloqueo, lo que significa que puede manejar muchas conexiones simultáneas sin bloquear el flujo de ejecución. Esto es especialmente útil para aplicaciones en tiempo real, como aplicaciones de chat y juegos en línea.

Además de ser utilizado para construir servidores, Node.js también es ampliamente utilizado en el desarrollo de herramientas y utilidades de línea de comandos, así como en el ecosistema de desarrollo web moderno.

Se pueden utilizar módulos npm (Node Package Manager) para acceder a una amplia variedad de paquetes y librerías de terceros que facilitan muchas tareas diferentes, desde la manipulación de archivos hasta la comunicación con bases de datos.

# VARIABLES

En JavaScript es un lenguaje de **tipado dinámico**, lo que significa que el tipo de una variable puede cambiar en tiempo de ejecución. Además.

- **boolean**: Representa un valor de verdadero (`true`) o falso (`false`). Ocupa muy poco espacio en la memoria.
- **number**: Representa tanto enteros como números de punto flotante. JavaScript utiliza un único tipo para todos los números, sin distinción entre enteros y decimales.
- **string**: Representa una secuencia de caracteres. El tamaño de una cadena de texto depende de la longitud de la misma, es decir, del número de caracteres que contiene.
- **array**: En JavaScript, los arrays son objetos especiales que pueden almacenar una colección de elementos. No hay un tamaño fijo para los arrays; su tamaño se ajusta dinámicamente según el número de elementos que contienen.
- **object**: Los objetos en JavaScript son estructuras de datos que pueden contener propiedades y métodos. El tamaño de un objeto varía según la cantidad y tipo de propiedades y métodos que posee. Los objetos tampoco tienen un tamaño fijo.
- **null y undefined**: Ambos son valores especiales que indican la ausencia de un valor. `null` se utiliza para representar intencionalmente la falta de un valor, mientras que `undefined` indica que una variable ha sido declarada pero no ha sido asignada. Aunque no tienen un tamaño específico asociado, ocupan un pequeño espacio en la memoria para almacenar sus referencias.

# Let y Const

**Redefinición del Operador let**

El operador let fue introducido en ES6 (ECMAScript 2015) y proporciona una forma más robusta de declarar variables en comparación con var.

Una de las principales ventajas de let es su capacidad para ofrecer un **"alcance de bloque"** (block scope). Esto significa que las variables declaradas con let están restringidas al bloque en el que se declaran, a diferencia de var, que tiene un alcance de función.

```
let i = 60;
(function Ejemplo() {
    for (let i = 0; i < 5; i++) {
        console.log(i); // Salida: 0, 1, 2, 3, 4
    }
})();
console.log('Después del loop', i); // Salida: 60
```

```
function Ejemplo() {
    const x = 10;
    console.log(x); // output 10
    x = 20; //throws type error
    console.log(x);
}
```

Const es igual que let, con una pequeña gran diferencia: no podemos re-asignar su valor.

# Funciones

Una **función** es un bloque de código diseñado para realizar una tarea específica. Las funciones permiten agrupar una serie de instrucciones bajo un nombre, que luego puede ser llamado para ejecutar ese bloque de código en cualquier momento. Esto facilita la reutilización del código y la organización de tareas complejas en partes más manejables.

## Definición de una Función

Una función en JavaScript se define utilizando la palabra clave `function`, seguida por un nombre, una lista de parámetros entre paréntesis y un bloque de código encerrado entre llaves `{}`.

```
function saludo(nombre){
    console.log ('Hola ' + nombre);
}
saludo('Emir');
```

# Ventajas de usar funciones

- **Reutilización de Código**: Puedes llamar a una función múltiples veces, evitando la duplicación de código.
- **Modularidad**: Facilita la división del código en partes manejables.
- **Abstracción**: Oculta los detalles de implementación y muestra solo la interfaz necesaria para utilizar la función.
- **Mantenimiento**: Hacer cambios en una función afecta a todas las llamadas a esa función, simplificando el mantenimiento.

# Funciones Flecha

En lugar de la palabra clave `function`, usa una flecha (`=>`) compuesta de los caracteres igual y mayor

```
const potencia = (base, exponente) => {
    let resultado = 1;
    for (let cuenta = 0; cuenta < exponente; cuenta++) {
      resultado *= base;
    }
    return resultado;
  };
```

# Recursividad

Es una técnica de programación en la que una función se llama a sí misma para resolver un problema. Esta técnica se utiliza para descomponer problemas complejos en subproblemas más simples y es especialmente útil en la resolución de problemas que tienen una estructura repetitiva o jerárquica.

## Conceptos Clave de la Recursividad

1. **Caso Base**: Es la condición que detiene la recursión. Sin un caso base, la función se llamaría a sí misma indefinidamente, provocando un desbordamiento de pila.
2. **Caso Recursivo**: Es la parte de la función que llama a sí misma con un conjunto de parámetros diferentes, acercándose eventualmente al caso base.

# Consideraciones al usar recursividad

1. **Profundidad de la Recursión**: Las llamadas recursivas pueden llevar a un desbordamiento de pila si la profundidad de la recursión es demasiado alta. Es importante asegurarse de que la recursión no sea demasiado profunda.
2. **Eficiencia**: Algunos problemas recursivos, pueden ser ineficientes debido a la repetición de cálculos.
3. **Comprensión**: Las funciones recursivas pueden ser más difíciles de entender y depurar que las soluciones iterativas. Es crucial tener una buena comprensión del problema y de cómo la recursión divide el problema en partes más simples.

# Flujo de control

El flujo de control en programación se refiere a la secuencia en la que se ejecutan las instrucciones de un programa. Es fundamental para entender cómo se toma una decisión, se repiten operaciones y se maneja la ejecución de diferentes partes del código.

### Estructuras de Decisión

Estas estructuras permiten que el programa tome decisiones basadas en condiciones específicas.

**`if` y `else`**

Permiten ejecutar diferentes bloques de código basados en una condición.

```
let edad = 18;

if (edad >= 18) {
    console.log("Eres mayor de edad.");
} else {
    console.log("Eres menor de edad.");
}
```

# Aspectos clave de flujo de control

**Decisiones:** Utiliza estructuras condicionales (como `if`, `else`, `switch`) para tomar decisiones y ejecutar diferentes bloques de código según ciertas condiciones.

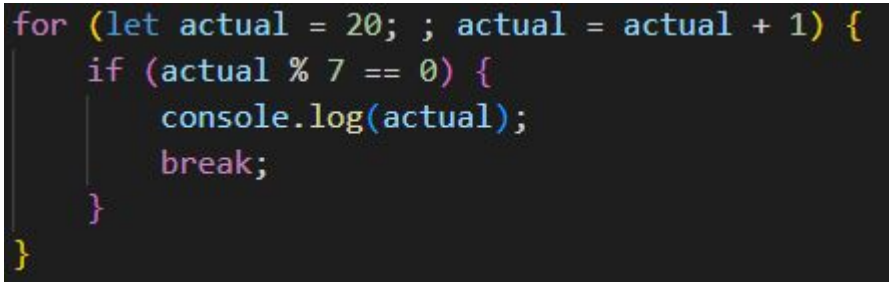
**Repeticiones:** Emplea estructuras de bucles (como `for`, `while`, `do...while`) para ejecutar repetidamente un bloque de código mientras se cumpla una condición.

**Saltos y Salidas:** Usa comandos como `break` para salir prematuramente de bucles, y `continue` para saltar a la siguiente iteración de un bucle.

**Funciones:** Permite la modularización del código mediante la definición y llamada de funciones, facilitando la reutilización y organización del programa.

# Rompiendo un flujo de control

Hacer que la condición del ciclo produzca false no es la única forma en que el ciclo puede terminar. Hay una declaración especial llamada `break` (“romper”) que tiene el efecto de inmediatamente saltar fuera del ciclo circundante. Este programa ilustra la declaración `break`. Encuentra el primer número que es a la vez mayor o igual a 20 y divisible por 7.



```
for (let actual = 20; ; actual = actual + 1) {
    if (actual % 7 == 0) {
        console.log(actual);
        break;
    }
}
```

# Arrays

En JavaScript, un **array** (o arreglo) es una estructura de datos que permite almacenar una colección de elementos en una sola variable. Los elementos en un array pueden ser de cualquier tipo de datos, incluyendo números, cadenas, objetos y otros arrays. Los arrays en JavaScript son objetos especiales que se pueden utilizar para almacenar listas de valores, y ofrecen numerosos métodos y propiedades para manipular esos valores.

```
// Creación de arrays
let colores = new Array('rojo', 'verde', 'azul'); // Usando el constructor Array
let frutas = ['manzana', 'banana', 'cereza']; // Usando la notación de corchetes
console.log(frutas[1]); // Imprime: banana
let numeros = [1, 2, 3];
numeros.push(4); // Añade 4 al final del array. El array ahora es [1, 2, 3, 4]

// Uso del método pop para eliminar el último elemento
let ultimoElemento = numeros.pop(); // Elimina y devuelve el último elemento del array.
console.log(ultimoElemento); // Imprime: 4

// Verificación del estado final del array
console.log(numeros); // Imprime: [1, 2, 3]
```

# Arrays de objetos

Array llamado `tasks` que contiene varios objetos. Cada objeto representa una tarea con dos propiedades:

- `name`: El nombre de la tarea.
- `duration`: La duración de la tarea en minutos.

## Notación de punto

El nombre del objeto (tasks) actúa como el espacio de nombre (namespace); al cual se debe ingresar primero para acceder a cualquier elemento encapsulado dentro del objeto.

```
let tasks = [
  {
    name: 'Tarea de matemática',
    duration: 120
  },
  {
    name: 'Limpiar cocina',
    duration: 60
  },
  {
    name: 'Visitar abuela',
    duration: 240
  }
];
```

# Números Aleatorios

```
let numeroAleatorio=Math.floor(Math.random() * (max - min)) + min;
```

`Math.random()` genera un número decimal entre 0 y 1.

`Math.random() * (max - min)` escala el número aleatorio al rango deseado.

`Math.floor()` redondea hacia abajo para obtener un número entero.

Sumamos `min` para desplazar el rango al intervalo deseado.

# Itinerar Array

JavaScript permite recorrer arrays y colecciones de objetos usando un tipo especial de bucle: el bucle `for - in` (que a su vez tiene ciertas similitudes con el for – each)

```
let arr = [1, 2, 3, 4, 5, 6];
for (let i in arr) {
 console.log(arr[i]);
}
```

# Transformando con map

El método `map` en JavaScript es una herramienta poderosa para transformar elementos de un array. Este método aplica una función a cada elemento del array original y crea un nuevo array con los resultados. El array original no se modifica; en su lugar, `map` devuelve un nuevo array con los valores transformados.

**`elemento`**: El elemento actual del array que se está procesando.
**`indice`** (opcional): El índice del elemento actual en el array.
**`array`** (opcional): El array sobre el que se está llamando `map`.

```
let nuevoArray = array.map(function(elemento, indice, array) {
    // Retorna el nuevo valor para cada elemento
});
```

# Ejemplo de MAP

Supongamos que tienes un array de números y quieres obtener un nuevo array con números al cuadrado

```
let numeros = [1, 2, 3, 4, 5];

let cuadrados = numeros.map(numero => numero ** 2);

console.log(cuadrados); // Imprime: [1, 4, 9, 16, 25]
```

# Filter

El método `filter()` se utiliza para crear un nuevo array con todos los elementos que cumplen una condición específica. Este método es muy útil para filtrar elementos de un array según un criterio.

```
array.filter(function(element, index, array) {
    // condición a evaluar
});
```

**Parámetros:**

- **`element`**: El elemento actual que se está procesando en el array.
- **`index`** (opcional): El índice del elemento actual en el array.
- **`array`** (opcional): El array sobre el que se llamó el método `filter()`

# Ejemplo y cómo Funciona filter()

**Recorre cada elemento del array.**
**Aplica la función de callback** a cada elemento. Esta función debe devolver `true` o `false`.
**Crea un nuevo array** que contiene solo los elementos para los cuales la función de callback devolvió `true`.

Supongamos que tienes un array de números y quieres filtrar solo los números mayores que 10:

```
const numbers = [5, 12, 8, 130, 44];

const filteredNumbers = numbers.filter(number => number > 10);

console.log(filteredNumbers); // Imprime: [12, 130, 44]
```

# Ejemplos con objetos

Supongamos que tienes un array de objetos que representan personas, y quieres filtrar solo aquellas que son mayores de 18 años:

```
const people = [
    { name: 'Ana', age: 22 },
    { name: 'Luis', age: 17 },
    { name: 'Carla', age: 19 },
    { name: 'Javier', age: 16 }
];

const adults = people.filter(person => person.age >= 18);

console.log(adults);
// Imprime: [ { name: 'Ana', age: 22 }, { name: 'Carla', age: 19 } ]
```

# Diferencias entre filter() y map()

Las funciones de **filtro** (`filter()`) y **mapeo** (`map()`) en JavaScript son métodos de los arrays que sirven para transformar y extraer información de los datos, pero tienen propósitos y comportamientos distintos.

`filter()`:

- **Objetivo**: Seleccionar y devolver solo aquellos elementos que cumplen con una condición específica.
- **Resultado**: Un nuevo array que contiene solo los elementos que pasan el test de la función de callback.

`map()`:

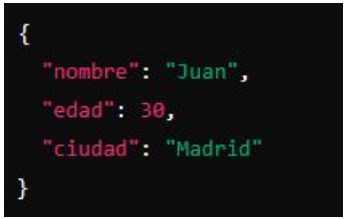
- **Objetivo**: Transformar cada elemento del array original según una función de transformación y devolver un nuevo array con los elementos transformados.
- **Resultado**: Un nuevo array con la misma longitud que el original, pero con cada elemento modificado según la función de callback.

# JSON

**JSON** (JavaScript Object Notation) es un formato ligero de intercambio de datos que es fácil de leer y escribir para los humanos y fácil de analizar y generar para las máquinas. Es ampliamente utilizado en aplicaciones web para enviar datos entre el cliente y el servidor.

## Características Clave de JSON:

1. **Formato de Datos:**
   - **Texto Plano:** JSON es un formato basado en texto que representa estructuras de datos en un formato de pares clave-valor.
   - **Estructura:** Los datos en JSON se organizan en **objetos** y **arrays**.
2. **Sintaxis de JSON:**
   - **Objetos:** Se representan con llaves `{}` y contienen pares clave-valor.
   - **Arrays:** Se representan con corchetes `[]` y contienen una lista de valores.

# Ejemplo

Representa la información de un usuario.

**A tener en cuenta**

- JSON es sólo un formato de datos -contiene sólo propiedades, no métodos-.
- JSON requiere usar comillas dobles para las cadenas y los nombres de propiedades. Las comillas simples no son válidas.
- Los comentarios no están permitidos en JSON.

## Manejo de JSON en JavaScript:

- **Parsear JSON:** Convertir una cadena JSON en un objeto JavaScript

```
const jsonString = '{"nombre": "Luis", "edad": 25}';
const obj = JSON.parse(jsonString);
console.log(obj.nombre); // Luis
```

- **Convertir a JSON:** Convertir un objeto JavaScript en una cadena JSON.

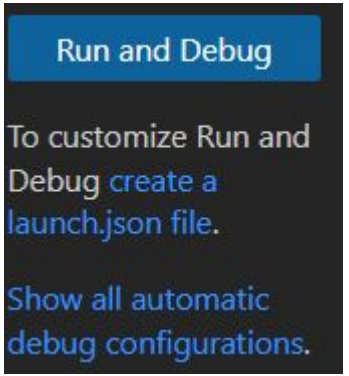
```
const obj = { nombre: "Luis", edad: 25 };
const jsonString = JSON.stringify(obj);
console.log(jsonString); // '{"nombre":"Luis","edad":25}'
```

# Para tener en cuenta JSON

El error "No debugger available, can not send 'variables'" en Visual Studio Code, es probable que esté relacionado con la configuración del depurador o con un problema en el entorno de desarrollo.

### 1. Verificar la Configuración del Depurador

1. **Abrir la Configuración de Depuración:**
   - Ve a la barra lateral izquierda y selecciona el ícono de "Ejecutar" o presiona `Ctrl+Shift+D` (Cmd+Shift+D en macOS).
   - Asegúrate de que tu archivo de configuración de depuración (`launch.json`) esté correctamente configurado.
2. **Revisar `launch.json`:**
   - Si no tenes un archivo `launch.json`, Visual Studio Code debería generarlo automáticamente cuando configuras una sesión de depuración.



# Función callback

Una **función callback** es una función que se pasa como argumento a otra función y se ejecuta dentro de esa función. Las funciones callback se utilizan comúnmente para manejar operaciones asincrónicas, eventos y para personalizar el comportamiento de funciones. Permiten la ejecución de código en respuesta a eventos o como parte de una operación que se completará en el futuro.