# JavaScript Teoría

Herramientas: Visual Studio Code - Code Editing. Redefined

Plugins: Live Server, Bracket Pair Colorizer, JavaScript (ES6) code snippets, vscodes icons

Para probar código en la web: https://stackblitz.com/edit/typescript-vxnz8z?file=index.ts

## Variables

**Variables:**

En JavaScript, una variable es un nombre que se utiliza para almacenar valores. Estos valores pueden ser números, cadenas de texto, objetos, arreglos u otros tipos de datos. Las variables permiten que puedas manipular y almacenar información de manera dinámica en tu programa. Para declarar una variable en JavaScript, puedes usar la palabra clave `var`, `let`, o `const` (a partir de ECMAScript 6). Por ejemplo:

```
var edad = 25;    // Declara una variable llamada "edad" y le asigna el valor 25
let nombre = "Juan"; // Declara una variable llamada "nombre" y le asigna el valor "Juan"
const pi = 3.1416;  // Declara una constante llamada "pi" y le asigna el valor 3.1416
```

**Constantes:**

Una constante es similar a una variable en términos de almacenamiento de valores, pero una vez que se le asigna un valor, no puede cambiar a lo largo de la ejecución del programa. Esto es útil cuando necesitas asegurarte de que un valor no cambie accidentalmente. La declaración de constantes se hace utilizando la palabra clave `const`, y el valor debe asignarse al momento de declararla:

```
const gravedad = 9.8; // Declara una constante llamada "gravedad" y le asigna el valor 9.8
```

Es importante entender que `var`, `let` y `const` tienen diferencias en cuanto al alcance (scope) y la mutabilidad:

- `**var**`: Tiene un alcance de función o global y puede ser redeclarada y reasignada.

- `**let**`: Tiene un alcance de bloque (bloque `{}`) y puede ser reasignada, pero no redeclarada en el mismo ámbito.

- `**const**`: Tiene un alcance de bloque y no puede ser reasignada ni redeclarada después de su declaración inicial. Sin embargo, para objetos y arreglos, el contenido interno puede ser modificado.

Aquí tienes un ejemplo práctico de cómo podrías utilizar variables y constantes en JavaScript:

```
let saldo = 1000;
const tasaInteres = 0.05;
saldo = saldo - (saldo * tasaInteres);
console.log("Saldo después de aplicar la tasa de interés:", saldo);
```

# ¿Qué son los string en JS?

En JavaScript, los "strings" (cadenas de caracteres) son secuencias de caracteres que representan texto. Un string puede contener letras, números, símbolos y espacios en blanco. Los strings son utilizados para almacenar y manipular datos de tipo texto en programas de JavaScript.

Los strings se pueden crear utilizando comillas simples (' '), comillas dobles (" "), o comillas invertidas (` `) en JavaScript. Aquí hay ejemplos de cómo se crean strings:

```
let str1 = 'Hola, esto es un string con comillas simples.';
let str2 = String ("Hola, esto es un string con comillas dobles.");
let str3 = new String (`Hola, esto es un string con comillas
invertidas.`);
```

# Métodos String

JavaScript proporciona una variedad de métodos que puedes utilizar para manipular y trabajar con strings. Aquí hay algunos de los métodos más comunes para strings en JavaScript:

1. `length`: Devuelve la longitud (cantidad de caracteres) de un string.

```
let mensaje = 'Hola, mundo!';
console.log(mensaje.length); // Salida: 13
```

2. `charAt(index)`: Devuelve el carácter en la posición especificada por `index`.

```
let mensaje = 'Hola';
console.log(mensaje.charAt(0)); // Salida: H
```

3. `concat(string1, string2, ...)`: Combina dos o más strings y devuelve uno nuevo.

```
let saludo = 'Hola';
let nombre = 'Juan';
let mensaje = saludo.concat(', ', nombre);
```

```
console.log(mensaje); // Salida: Hola, Juan
```

4. `toUpperCase()`: Convierte el string a mayúsculas.

```
let mensaje = 'Hola, mundo!';
console.log(mensaje.toUpperCase()); // Salida: HOLA, MUNDO!
```

5. `toLowerCase()`: Convierte el string a minúsculas.

```
let mensaje = 'HOLA, MUNDO!';
console.log(mensaje.toLowerCase()); // Salida: hola, mundo!
```

6. `slice(start, end)`: Extrae una porción del string desde la posición `start` hasta `end` (sin incluir `end`).

```
let mensaje = 'Hola, mundo!';
console.log(mensaje.slice(0, 4)); // Salida: Hola
```

7. `substring(start, end)`: Similar a `slice`, pero permite valores negativos.

```
let mensaje = 'Hola, mundo!';
console.log(mensaje.substring(6, 11)); // Salida: mundo
```

8. `indexOf(substring, fromIndex)`: Devuelve la primera posición de la subcadena encontrada, o -1 si no se encuentra. Puede empezar la búsqueda desde la posición `fromIndex`.

```
let mensaje = 'Hola, mundo!';
console.log(mensaje.indexOf('mundo')); // Salida: 6
```

9. `lastIndexOf(substring, fromIndex)`: Similar a `indexOf`, pero comienza la búsqueda desde el final del string.

```
let mensaje = 'Hola, mundo, hola!';
console.log(mensaje.lastIndexOf('hola')); // Salida: 13
```

10. `replace(searchValue, replaceValue)`: Reemplaza `searchValue` por `replaceValue` en el string.

```
let mensaje = 'Hola, mundo!';
console.log(mensaje.replace('mundo', 'amigo')); // Salida: Hola, amigo!
```

Estos son solo algunos de los métodos más comunes para manipular strings en JavaScript. Hay muchos otros métodos disponibles, así que te recomiendo consultar la documentación oficial de JavaScript para obtener más información:
https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/String

# Orden de las operaciones en JavaScript

En JavaScript, al igual que en la mayoría de los lenguajes de programación, las operaciones se realizan en un orden específico siguiendo reglas establecidas. Este orden de operaciones se conoce como "precedencia de operadores". A continuación, se presenta un resumen del orden de operaciones en JavaScript, de mayor a menor precedencia:

1. Paréntesis: Las expresiones dentro de paréntesis se evalúan primero.

2. Exponente (**): Las operaciones de potenciación se realizan después de las expresiones entre paréntesis.

3. Multiplicación (*), División (/), Módulo (%): Estas operaciones se realizan de izquierda a derecha.

4. Suma (+), Resta (-): Estas operaciones también se realizan de izquierda a derecha.

5. Concatenación de cadenas: Cuando se utilizan operadores + con cadenas, se concatenan en lugar de sumarse.

6. Operadores de comparación: Estos incluyen ==, ===, !=, !==, >, >=, <, <=, que se evalúan de izquierda a derecha.

7. Operadores lógicos: Estos incluyen && (AND lógico) y || (OR lógico). && tiene mayor precedencia que ||.

8. Asignación (=, +=, -=, *=, /=, %=, **=): Las asignaciones se realizan después de las operaciones aritméticas y lógicas.

Es importante mencionar que el uso de paréntesis puede alterar el orden de operaciones y permitirte controlar explícitamente el flujo de cálculos en tu código.

Aquí hay un ejemplo para ilustrar el orden de operaciones:

```
var resultado = (10 + 5) * 2 - 3 / 2;
// 10 + 5 = 15
// 15 * 2 = 30
// 3 / 2 = 1.5
// 30 - 1.5 = 28.5
```

En este ejemplo, las operaciones dentro de los paréntesis se realizan primero, seguidas de las multiplicaciones y divisiones, y finalmente la resta.

*Nota: Siempre es recomendable usar paréntesis de manera explícita para evitar confusiones y asegurarse de que las operaciones se realicen en el orden deseado.*

# Objetos Literales

En JavaScript, los "objetos literales" se refieren a una forma de crear y definir objetos de manera directa utilizando una sintaxis concisa y legible. Los objetos literales son una parte fundamental de la programación en JavaScript, ya que permiten organizar y agrupar datos relacionados en una estructura única y flexible.

Un objeto literal en JavaScript se define utilizando llaves `{}` y puede contener pares clave-valor separados por comas. Cada clave es una cadena (o un identificador válido) que actúa como nombre de propiedad, y su valor asociado puede ser cualquier tipo de dato válido en JavaScript, como números, cadenas, booleanos, funciones, arreglos u otros objetos.

Aquí hay un ejemplo sencillo de un objeto literal en JavaScript:

```
const persona = {
    nombre: "Juan",
    edad: 30,
    casado: false,
    saludar: function() {
      console.log("¡Hola! Mi nombre es " + this.nombre);
    }
  };
```

En este ejemplo, `persona` es un objeto literal que tiene tres propiedades: `nombre`, `edad` y `casado`. También tiene un método llamado `saludar` que muestra un mensaje en la consola utilizando el valor de la propiedad `nombre`. Puedes acceder a las propiedades y métodos de un objeto literal utilizando la notación de punto (`objeto.propiedad`) o la notación de corchetes (`objeto['propiedad']`).

Los objetos literales son una forma poderosa de organizar y manipular datos en JavaScript, y se utilizan ampliamente en el desarrollo web y en la programación en general. Puedes crear objetos anidados, actualizar sus propiedades y métodos, e incluso utilizarlos como estructuras de datos más complejas en tus aplicaciones.

Anidacion de Objetos:

```
const persona = {
    nombre: "Juan",
    edad: 30,
    direccion: {
        calle: "123 Calle Principal",
        ciudad: "Ciudad Ejemplo",
        pais: "País Imaginario"
    }
};
console.log(persona.nombre);  // Salida: Juan
console.log(persona.direccion.calle);  // Salida: 123 Calle Principal
```

También puedes anidar aún más objetos dentro de otros objetos, creando una estructura de datos compleja:

```
var empresa = {
    nombre: "Mi Empresa",
    empleados: [
      { nombre: "Ana", edad: 25 },
      { nombre: "Pedro", edad: 28 },
      { nombre: "María", edad: 22 }
    ]
  };
```

En JavaScript, puedes agregar y eliminar propiedades de objetos literales de varias formas. Aquí te muestro cómo hacerlo:

**### Agregar propiedades:**

Puedes agregar propiedades a un objeto literal asignando un valor a una nueva clave utilizando la notación de punto o la notación de corchetes. Aquí tienes ejemplos de ambas formas:

```
const persona = {
    nombre: "Juan",
    edad: 30
};

// Agregar propiedad usando notación de punto
persona.direccion = "123 Calle Principal";

// Agregar propiedad usando notación de corchetes
persona["telefono"] = "555-1234";

console.log(persona);
```

En este ejemplo, hemos agregado las propiedades `direccion` y `telefono` al objeto `persona`.

### Eliminar propiedades:

Puedes eliminar propiedades de un objeto utilizando el operador `delete`. Aquí tienes un ejemplo:

```
const persona = {
    nombre: "Juan",
    edad: 30,
    direccion: "123 Calle Principal"
};

delete persona.direccion;
```

```
console.log(persona);
```

En este ejemplo, hemos eliminado la propiedad `direccion` del objeto `persona`.

Es importante tener en cuenta que si intentas acceder a una propiedad que no existe en un objeto, obtendrás el valor `undefined`. Por lo tanto, al eliminar una propiedad, ya no podrás acceder a ella en el futuro.

```
const persona = {
    nombre: "Juan",
    edad: 30,
    direccion: "123 Calle Principal"
};

console.log(persona.pais);  // Salida: undefined
```

# “use strict”

La declaración `"use strict";` en JavaScript se utiliza para activar el modo estricto en un bloque de código o en todo el ámbito de un archivo JavaScript. Este modo estricto impone una serie de restricciones y mejoras en el comportamiento del lenguaje, lo que ayuda a evitar errores comunes y a escribir un código más robusto y seguro.

En relación con los objetos literales, el modo estricto puede ayudar a prevenir ciertos comportamientos inesperados y a mejorar la calidad de tu código. Algunas de las razones para usar `"use strict";` con objetos literales incluyen:

1. **Evitar declaraciones implícitas de variables:** En el modo estricto, no puedes crear variables sin declararlas explícitamente utilizando `var`, `let` o `const`. Esto ayuda a prevenir errores al intentar asignar valores a propiedades de objetos que no se han declarado previamente.

# Object.freeze y Object.seal

`Object.freeze` y `Object.seal` son dos métodos en JavaScript que se utilizan para limitar la capacidad de modificar objetos y sus propiedades. Ambos métodos se utilizan para establecer ciertos niveles de inmutabilidad y control sobre los objetos, pero tienen diferencias clave en cuanto a su comportamiento. Aquí están las principales diferencias entre `Object.freeze` y `Object.seal`:

Object.freeze:

1. **Inmutabilidad completa:** `Object.freeze` hace que un objeto sea completamente inmutable, lo que significa que no se pueden agregar, eliminar ni modificar propiedades existentes del objeto.

2. **Propiedades no configurables:** Además de hacer que el objeto sea inmutable, `Object.freeze` también establece todas las propiedades existentes del objeto como no configurables, lo que significa que no se pueden cambiar sus atributos de configuración (como `writable`, `configurable` y `enumerable`).

3. **Recursivo:** Si el objeto contiene propiedades que son objetos, `Object.freeze` se aplicará de manera recursiva a esos objetos internos, lo que resulta en una inmutabilidad completa en toda la jerarquía.

Object.seal:

1. **Inmutabilidad parcial:** `Object.seal` permite modificar las propiedades existentes del objeto, pero no permite agregar ni eliminar nuevas propiedades. Esto hace que el objeto sea parcialmente inmutable.

2. **Propiedades no configurables:** Al igual que `Object.freeze`, `Object.seal` establece todas las propiedades existentes como no configurables, lo que impide cambiar sus atributos de configuración.

3. **Recursivo:** Similar a `Object.freeze`, `Object.seal` también se aplica de manera recursiva a los objetos internos en caso de que haya propiedades que sean objetos.

## Ejemplo con Object.freeze:

```
var persona = {
    nombre: "Juan",
    edad: 30
};

Object.freeze(persona);

// Intentamos modificar una propiedad existente (esto no surtirá efecto)
persona.nombre = "Carlos";

// Intentamos agregar una nueva propiedad (esto no surtirá efecto)
persona.direccion = "123 Calle Principal";

console.log(persona); // Salida: { nombre: "Juan", edad: 30 }

// Si el objeto contiene objetos internos, también serán inmutables
persona.detalles = {
    altura: 180,
    peso: 75
};
Object.freeze(persona.detalles);

persona.detalles.altura = 185; // Esto no surtirá efecto
console.log(persona.detalles); // Salida: { altura: 180, peso: 75 }
```

# Ejemplo con Object.seal:

```
var persona = {
    nombre: "Juan",
    edad: 30
};

Object.seal(persona);

// Modificamos una propiedad existente
persona.nombre = "Carlos";

// Intentamos agregar una nueva propiedad (esto no surtirá efecto)
persona.direccion = "123 Calle Principal";

console.log(persona); // Salida: { nombre: "Carlos", edad: 30 }

// Si el objeto contiene objetos internos, también serán sellados
persona.detalles = {
    altura: 180,
    peso: 75
};
Object.seal(persona.detalles);

persona.detalles.altura = 185; // Esto se reflejará en el objeto sellado
console.log(persona.detalles); // Salida: { altura: 185, peso: 75 }
```

# Objeto Constructor

Un "objeto constructor" en JavaScript se refiere a una función que se utiliza para crear y construir nuevos objetos. Los objetos constructores se utilizan en conjunción con la palabra clave `new` para instanciar y crear objetos basados en un "prototipo" definido por la función constructora. Los objetos construidos de esta manera heredan las propiedades y métodos del prototipo.

Aquí tienes un ejemplo básico de cómo crear un objeto constructor en JavaScript:

```
// Definición del objeto constructor
function Persona(nombre, edad) {
    this.nombre = nombre;
    this.edad = edad;
    this.saludar = function () {
        console.log("Hola, mi nombre es " + this.nombre + " y tengo " +
this.edad + " años.");
    };
}

// Crear una instancia de Persona utilizando el objeto constructor
var persona1 = new Persona("Juan", 30);
var persona2 = new Persona("María", 25);

persona1.saludar(); // Salida: Hola, mi nombre es Juan y tengo 30 años.
persona2.saludar(); // Salida: Hola, mi nombre es María y tengo 25 años.
```

En este ejemplo, `Persona` es un objeto constructor que acepta dos parámetros (`nombre` y `edad`). Cuando utilizas `new Persona(...)`, se crea una nueva instancia del objeto `Persona` y se asignan los valores proporcionados a las propiedades `nombre` y `edad` de esa instancia.

Los objetos constructores también pueden tener propiedades y métodos que son compartidos por todas las instancias creadas a partir de ellos. Sin embargo, en el ejemplo anterior, cada instancia tiene su propia copia del método `saludar`, lo que podría ser ineficiente si se crean muchas instancias.

Para evitar este problema, puedes definir los métodos en el "prototipo" del constructor en lugar de dentro de la función constructora misma:

```
function Persona(nombre, edad) {
    this.nombre = nombre;
    this.edad = edad;
}

Persona.prototype.saludar = function () {
    console.log("Hola, mi nombre es " + this.nombre + " y tengo " +
this.edad + " años.");
};
```

De esta manera, el método `saludar` se compartirá entre todas las instancias de `Persona`, lo que puede ser más eficiente en términos de uso de memoria.

# Prototype

```
function Persona(nombre, edad) {
    this.nombre = nombre;
    this.edad = edad;
}

Persona.prototype.saludar = function () {
    console.log("Hola, mi nombre es " + this.nombre + " y tengo " +
this.edad + " años.");
};
```

Cuando defines una función constructora, como en tu ejemplo `function Persona(nombre, edad) { ... }`, estás creando una plantilla para crear nuevos objetos con una estructura y propiedades específicas. Cada vez que utilizas la palabra clave `new` con esta función constructora, se crea una nueva instancia de objeto basada en esa plantilla.

Ahora, en el ejemplo, tienes una función constructora `Persona` que acepta dos parámetros: `nombre` y `edad`. Dentro de esta función, asignas los valores de `nombre` y `edad` a las propiedades `nombre` y `edad` del objeto que se está creando.

Hasta aquí, todo bien. Pero aquí es donde entra en juego el concepto de prototipos. Cada función en JavaScript tiene una propiedad llamada `prototype`, que es un objeto. Cuando añades propiedades o métodos a esta propiedad `prototype`, estás agregando propiedades y métodos compartidos que todas las instancias creadas a partir de esa función constructora pueden acceder y usar.

En tu caso, estás agregando un método llamado `saludar` al prototipo de la función constructora `Persona`

```
Persona.prototype.saludar = function () {
    console.log("Hola, mi nombre es " + this.nombre + " y tengo " +
this.edad + " años.");
};
```

Esto significa que todas las instancias creadas con `Persona` tendrán acceso a este método `saludar`. Cuando llamas a `persona1.saludar()`, el motor de JavaScript busca en el objeto `persona1` para ver si tiene una propiedad `saludar`. Como no la encuentra directamente en `persona1`, busca en el prototipo de la función constructora `Persona`, donde encuentra el método `saludar` y lo ejecuta.

Esto permite que todas las instancias de `Persona` **compartan** el mismo método `saludar` sin tener que duplicar el código en cada objeto individual.