**Que son las variables var y las variables let (de un ejemplo)**

**var**:

* Antes de la llegada de **let** en ECMAScript 6 (también conocido como ES6), **var** era la forma más común de declarar variables en JavaScript.
* Las variables declaradas con **var** tienen un alcance de función o global, lo que significa que pueden ser accesibles dentro de la función en la que se declararon o en todo el ámbito global del script, dependiendo de dónde se declaren.
* Las variables **var** pueden ser re-declaradas y actualizadas dentro de su ámbito.

Texto

Descripción generada automáticamente

**let**:

* Introducido en ECMAScript 6 (ES6), **let** tiene un alcance de bloque, lo que significa que solo están disponibles dentro del bloque en el que se declaran (por ejemplo, dentro de un bucle **for**, una declaración **if**, un bloque de función, etc.).
* Las variables **let** no pueden ser re-declaradas dentro del mismo ámbito y no permiten la duplicación de nombres de variable dentro de un mismo bloque.
* A diferencia de **var**, las variables **let** no se elevan al principio del ámbito, lo que significa que no puedes acceder a ellas antes de su declaración.

Texto

Descripción generada automáticamente

**Que son las constantes (conts) (de un ejemplo)**

las constantes (const) en JavaScript son variables cuyo valor no puede cambiar después de su declaración inicial. Son útiles para almacenar valores que no deben modificarse durante la ejecución del programa, como constantes matemáticas, valores de configuración, o referencias a objetos inmutables.

// Ejemplo de constante

const PI = 3.14159;

console.log(PI); // Imprime: 3.14159

// Intentar cambiar el valor de una constante generará un error

// PI = 3; // Esto generaría un error

// Las constantes deben asignarse un valor al momento de la declaración

// const MY\_CONSTANT; // Esto generaría un error, ya que no se ha asignado un valor a la constante

// Las constantes pueden ser de cualquier tipo de dato

const SALUDO = "¡Hola, mundo!";

console.log(SALUDO); // Imprime: ¡Hola, mundo!

// Las constantes pueden ser objetos o arreglos, y sus propiedades pueden modificarse

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

console.log(persona.nombre); // Imprime: Juan

persona.edad = 31; // Esto es válido, ya que no estamos cambiando la referencia de la constante, solo sus propiedades

console.log(persona.edad); // Imprime: 31

// Sin embargo, no podemos asignar una nueva referencia a la constante

// persona = { nombre: "María", edad: 25 }; // Esto generaría un error

**Que son las cadenas de texto (Strings) (de un ejemplo)**

Las cadenas de texto, también conocidas como strings en inglés, son una secuencia de caracteres encerrados entre comillas simples (''), comillas dobles ("") o comillas invertidas (``). En JavaScript, las cadenas de texto son utilizadas para representar datos de texto.

**/ Ejemplo de cadenas de texto**

let mensaje1 = '¡Hola, mundo!';

let mensaje2 = "Esta es una cadena de texto.";

let mensaje3 = `Las comillas invertidas también pueden usarse para crear cadenas de texto.`;

console.log(mensaje1); // Imprime: ¡Hola, mundo!

console.log(mensaje2); // Imprime: Esta es una cadena de texto.

console.log(mensaje3); // Imprime: Las comillas invertidas también pueden usarse para crear cadenas de texto.

**Que son las plantillas de cadenas texto (Template Strings) (de un ejemplo)**

Las plantillas de cadenas de texto, también conocidas como template strings en inglés, son una característica introducida en ECMAScript 6 (ES6) que permite crear cadenas de texto de una manera más legible y flexible, permitiendo la interpolación de variables y expresiones dentro de la cadena utilizando la sintaxis de comillas invertidas (``).

**// Ejemplo de plantillas de cadenas de texto**

let nombre = 'Juan';

let edad = 30;

// Utilizando plantillas de cadenas de texto

let mensaje = `Hola, mi nombre es ${nombre} y tengo ${edad} años.`;

console.log(mensaje); // Imprime: Hola, mi nombre es Juan y tengo 30 años.

**Que son los números (Numbres) (de un ejemplo)**

Los números, también conocidos como números en inglés, son un tipo de datos en JavaScript que representan valores numéricos, ya sean enteros o de punto flotante (números decimales). En JavaScript, los números pueden ser escritos con o sin decimales.

**// Ejemplo de números**

let entero = 42; // Un número entero

let decimal = 3.14; // Un número decimal (de punto flotante)

console.log(entero); // Imprime: 42

console.log(decimal); // Imprime: 3.14

// Se pueden realizar operaciones matemáticas con números

let suma = entero + decimal;

console.log(suma); // Imprime: 45.14

// Los números también pueden ser negativos

let negativo = -7;

console.log(negativo); // Imprime: -7

// Los números en JavaScript pueden representar valores muy grandes o pequeños

let muyGrande = 1.5e10; // 1.5 \* 10^10

let muyPequenio = 2.5e-4; // 2.5 \* 10^(-4)

console.log(muyGrande); // Imprime: 15000000000

console.log(muyPequenio); // Imprime: 0.00025

**Que son los Booleans (de un ejemplo)**

Los booleanos, también conocidos como valores booleanos, son un tipo de datos en JavaScript que representan un valor de verdad, es decir, pueden ser true (verdadero) o false (falso). Estos valores son fundamentales para el control de flujo y la toma de decisiones en programación.

**// Ejemplo de booleanos**

let esMayorDeEdad = true; // Variable booleana que indica si una persona es mayor de edad

let esDiaSoleado = false; // Variable booleana que indica si el día está soleado

console.log(esMayorDeEdad); // Imprime: true

console.log(esDiaSoleado); // Imprime: false

// Los booleanos se utilizan comúnmente en estructuras condicionales

if (esMayorDeEdad) {

console.log("La persona es mayor de edad");

} else {

console.log("La persona es menor de edad");

}

if (esDiaSoleado) {

console.log("Hace sol hoy");

} else {

console.log("No hace sol hoy");

}

**Que son los undefined, null y Nan (de un ejemplo)**

En JavaScript, undefined, null y NaN son valores especiales que representan diferentes conceptos:

1. **undefined:** Este valor se asigna automáticamente a las variables que han sido declaradas pero no inicializadas con un valor. También puede ser el valor de retorno de una función si no se especifica explícitamente lo que debe devolver.

**let variableSinValor;**

console.log(variableSinValor); // Imprime: undefined

function obtenerValor() {

// No se especifica un valor de retorno

}

console.log(obtenerValor()); // Imprime: undefined

1. **null**: Este valor se utiliza para indicar la ausencia intencional de cualquier valor o referencia de objeto. Es diferente de **undefined**, que generalmente indica que una variable no está definida o que una función no tiene un valor de retorno.

let objetoNulo = null;

console.log(objetoNulo); // Imprime: null

1. **NaN** (Not a Number): Este valor se utiliza para representar un valor que no es un número válido en JavaScript. Puede ocurrir, por ejemplo, al intentar realizar operaciones matemáticas inválidas, como dividir una cadena por un número.

let resultado = "texto" / 2;

console.log(resultado); // Imprime: NaN

**Que son las funciones y que tipos de funciones existente en JavaScript (de ejemplos)**

las funciones son bloques de código reutilizable que pueden ser llamados o invocados para realizar una tarea específica. Las funciones pueden aceptar parámetros como entrada y pueden devolver un valor como salida. Son una parte fundamental del lenguaje y se utilizan para organizar y reutilizar el código de manera eficiente.

Existen varios tipos de funciones en JavaScript, incluyendo:

1. **Funciones Declarativas**: También conocidas como funciones nombradas o funciones con nombre, se declaran utilizando la palabra clave **function**. Pueden ser llamadas antes de que se definan en el código, debido al hoisting.

// Ejemplo de función declarativa

function saludar(nombre) {

return "¡Hola, " + nombre + "!";

}

console.log(saludar("Juan")); // Imprime: ¡Hola, Juan!

1. **Expresiones de Funciones**: Son funciones asignadas a una variable o propiedad de objeto. Pueden ser anónimas o nombradas.

// Ejemplo de expresión de función anónima

let sumar = function(a, b) {

return a + b;

};

console.log(sumar(3, 5)); // Imprime: 8

// Ejemplo de expresión de función nombrada

let multiplicar = function multiplicar(a, b) {

return a \* b;

};

console.log(multiplicar(2, 4)); // Imprime: 8

1. **Funciones Flecha (Arrow Functions)**: Son una forma más concisa de escribir funciones en JavaScript introducidas en ES6. Tienen una sintaxis más corta y conservan el contexto **this** del entorno en el que se definen.

// Ejemplo de función flecha

let cuadrado = (x) => {

return x \* x;

};

console.log(cuadrado(3)); // Imprime: 9

1. **Funciones Constructoras**: Son funciones utilizadas junto con el operador **new** para crear objetos. Se utilizan como plantillas para crear múltiples objetos del mismo tipo.

// Ejemplo de función constructora

function Persona(nombre, edad) {

this.nombre = nombre;

this.edad = edad;

}

let persona1 = new Persona("Juan", 30);

let persona2 = new Persona("María", 25);

console.log(persona1); // Imprime: Persona { nombre: 'Juan', edad: 30 }

console.log(persona2); // Imprime: Persona { nombre: 'María', edad: 25 }

1. **Funciones anónimas**: Las funciones anónimas son útiles cuando solo necesitas definir una función en un lugar específico del código y no la vas a reutilizar en otros lugares.

// Ejemplo de función anónima

let saludar = function(nombre) {

return "¡Hola, " + nombre + "!";

};

console.log(saludar("Juan")); // Imprime: ¡Hola, Juan!

**Que son los arreglos (Arrays) (de ejemplos)**

los arreglos (arrays en inglés) son estructuras de datos que se utilizan para almacenar una colección ordenada de elementos. Los elementos de un arreglo pueden ser de cualquier tipo de datos, como números, strings, booleanos, objetos u otros arreglos. Cada elemento en un arreglo tiene una posición numérica conocida como índice, que comienza desde 0 para el primer elemento y se incrementa secuencialmente.

**// Ejemplo de creación de un arreglo con números**

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

// Ejemplo de creación de un arreglo con strings

let colores = ['rojo', 'verde', 'azul'];

// Ejemplo de creación de un arreglo mixto

let mixto = [1, 'dos', true, [6, 7, 8]];

// También se puede crear un arreglo vacío y luego añadir elementos

let vacio = [];

vacio.push(10);

vacio.push(20);

**// Accediendo a elementos por su índice**

console.log(numeros[0]); // Imprime: 1

console.log(colores[1]); // Imprime: verde

// Accediendo a elementos de un arreglo dentro de otro arreglo

console.log(mixto[3][0]); // Imprime: 6

**// Modificando un elemento del arreglo**

numeros[2] = 30;

// Añadiendo un nuevo elemento al final del arreglo

colores.push('amarillo');

// Eliminando el último elemento del arreglo

mixto.pop();

console.log(numeros.length); // Imprime: 5

console.log(colores.length); // Imprime: 4

**Que son los objetos en JavaScript (de un ejemplo)**

los objetos son estructuras de datos que permiten almacenar datos y funciones relacionadas en un solo lugar. Los objetos pueden representar entidades del mundo real, como personas, productos o cualquier otro concepto, y se componen de pares de clave-valor, donde cada clave es una cadena única que actúa como identificador del valor correspondiente.

// Ejemplo de objeto en JavaScript

let persona = {

nombre: 'Juan',

edad: 30,

ocupacion: 'Desarrollador',

saludar: function() {

console.log('¡Hola! Soy ' + this.nombre + ', tengo ' + this.edad + ' años y soy ' + this.ocupacion + '.');

}

};

// Accediendo a propiedades del objeto

console.log(persona.nombre); // Imprime: Juan

console.log(persona.edad); // Imprime: 30

// Llamando a un método del objeto

persona.saludar(); // Imprime: ¡Hola! Soy Juan, tengo 30 años y soy Desarrollador.

**Tipos de operadores (de un ejemplo de cada uno)**

hay varios tipos de operadores que se utilizan para realizar diferentes operaciones, como operaciones aritméticas, comparaciones, asignaciones y lógicas. Aquí tienes ejemplos de cada uno de los tipos de operadores:

1. **Operadores Aritméticos:**
   * Se utilizan para realizar operaciones matemáticas.

// Ejemplo de operadores aritméticos

let x = 5;

let y = 2;

console.log(x + y); // Suma: Imprime 7

console.log(x - y); // Resta: Imprime 3

console.log(x \* y); // Multiplicación: Imprime 10

console.log(x / y); // División: Imprime 2.5

console.log(x % y); // Módulo: Imprime 1 (resto de la división)

console.log(++x); // Incremento: Imprime 6

console.log(--y); // Decremento: Imprime 1

1. **Operadores de Comparación**:

* Se utilizan para comparar dos valores y devolver un resultado booleano.

// Ejemplo de operadores de comparación

let a = 5;

let b = 10;

console.log(a == b); // Igual a: Imprime false

console.log(a != b); // No igual a: Imprime true

console.log(a > b); // Mayor que: Imprime false

console.log(a < b); // Menor que: Imprime true

console.log(a >= b); // Mayor o igual que: Imprime false

console.log(a <= b); // Menor o igual que: Imprime true

1. **Operadores de Asignación**:

* Se utilizan para asignar valores a variables.

// Ejemplo de operadores de asignación

let c = 10;

let d = 5;

c += d; // Equivalente a c = c + d;

console.log(c); // Imprime 15

d \*= 2; // Equivalente a d = d \* 2;

console.log(d); // Imprime 10

1. **Operadores Lógicos**:

* Se utilizan para combinar expresiones condicionales y devolver un resultado booleano.

// Ejemplo de operadores lógicos

let llueve = true;

let haceFrio = false;

console.log(llueve && haceFrio); // AND lógico: Imprime false

console.log(llueve || haceFrio); // OR lógico: Imprime true

console.log(!llueve); // NOT lógico: Imprime false

1. **Operadores de Concatenación de Cadenas**:

* Se utilizan para concatenar cadenas de texto.

/ Ejemplo de operador de concatenación de cadenas

let nombre = "Juan";

let apellido = "Pérez";

let nombreCompleto = nombre + " " + apellido;

console.log(nombreCompleto); // Imprime "Juan Pérez"

**Condicionales (de un ejemplo)**

Los condicionales en JavaScript se utilizan para ejecutar diferentes bloques de código según una condición específica. El condicional más común es la instrucción **if**, que ejecuta un bloque de código si una condición dada es verdadera. También se pueden utilizar instrucciones **else if** y **else** para especificar bloques de código alternativos que se ejecutan cuando la condición **if** es falsa.

// Ejemplo de condicional if

let edad = 20;

if (edad >= 18) {

console.log("Eres mayor de edad.");

} else {

console.log("Eres menor de edad."

/ Ejemplo de condicional else if

let puntaje = 85;

if (puntaje >= 90) {

console.log("Aprobaste con una A.");

} else if (puntaje >= 80) {

console.log("Aprobaste con una B.");

} else if (puntaje >= 70) {

console.log("Aprobaste con una C.");

} else {

console.log("Has reprobado.");

}

**Ciclos (Loops) (de un ejemplo de cada uno)**

hay varios tipos de ciclos o bucles que se utilizan para repetir la ejecución de un bloque de código hasta que se cumple una condición específica. Los tres tipos principales de ciclos son: for, while y do...while.

**Ciclo for**:

* Se utiliza para iterar sobre una secuencia de valores.

// Ejemplo de ciclo for

for (let i = 0; i < 5; i++) {

console.log("Iteración " + (i + 1));

}

**Ciclo while**:

* Se utiliza para ejecutar un bloque de código mientras una condición específica sea verdadera.

// Ejemplo de ciclo while

let contador = 0;

while (contador < 5) {

console.log("Iteración " + (contador + 1));

contador++;

}

**Ciclo do...while**:

* Similar al ciclo **while**, pero garantiza que el bloque de código se ejecute al menos una vez antes de verificar la condición.

// Ejemplo de ciclo do...while

let x = 0;

do {

console.log("Iteración " + (x + 1));

x++;

} while (x < 5);

**Manejo de errores (de un ejemplo)**

// Función que divide dos números y maneja cualquier error que pueda ocurrir

function dividir(a, b) {

try {

if (b === 0) {

throw new Error("División por cero no permitida");

}

return a / b;

} catch (error) {

// Captura y maneja el error

console.error("Ha ocurrido un error:", error.message);

return null; // Devuelve null para indicar que hubo un error

}

}

// Ejemplo de uso de la función dividir

console.log(dividir(10, 2)); // Imprime: 5

console.log(dividir(8, 0)); // Imprime: Ha ocurrido un error: División por cero no permitida

console.log(dividir(15, 3)); // Imprime: 5

**Break & Continue (de un ejemplo de cada uno)**

las palabras clave break y continue se utilizan dentro de bucles para controlar el flujo de ejecución

**Ejemplo de break**:

* La palabra clave **break** se utiliza para salir de un bucle antes de que se complete su iteración normal.

// Ejemplo de break

for (let i = 0; i < 5; i++) {

console.log("Iteración " + (i + 1));

if (i === 2) {

break; // Salir del bucle cuando i sea igual a 2

}

}

**Ejemplo de continue**:

* La palabra clave **continue** se utiliza para saltar una iteración del bucle y continuar con la siguiente iteración

// Ejemplo de continue

for (let i = 0; i < 5; i++) {

if (i === 2) {

continue; // Saltar la iteración cuando i sea igual a 2

}

console.log("Iteración " + (i + 1));

}

**Que es la destructuración (de un ejemplo)**

La destructuración es una característica de JavaScript que permite descomponer o extraer valores de objetos y arreglos en variables individuales de una manera más concisa y legible. Esto simplifica la asignación de valores a variables, especialmente cuando trabajas con objetos o arreglos complejos.

// Ejemplo de destructuración de objetos

let persona = {

nombre: 'Juan',

edad: 30,

ocupacion: 'Desarrollador'

};

// Extrayendo valores del objeto persona en variables individuales

let { nombre, edad, ocupacion } = persona;

console.log(nombre); // Imprime: Juan

console.log(edad); // Imprime: 30

console.log(ocupacion); // Imprime: Desarrollador

En este ejemplo, la destructuración se utiliza para extraer las propiedades **nombre**, **edad** y **ocupacion** del objeto **persona** y asignarlas a variables del mismo nombre. Esto elimina la necesidad de acceder a las propiedades del objeto utilizando la notación de puntos (**persona.nombre**, **persona.edad**, etc.).

// Ejemplo de destructuración de arreglos

let numeros = [1, 2, 3];

// Extrayendo valores del arreglo numeros en variables individuales

let [num1, num2, num3] = numeros;

console.log(num1); // Imprime: 1

console.log(num2); // Imprime: 2

console.log(num3); // Imprime: 3

En este ejemplo, la destructuración se utiliza para extraer los valores individuales del arreglo **numeros** y asignarlos a variables **num1**, **num2** y **num3**.

**Que son los objetos literales (de un ejemplo)**

Los objetos literales, también conocidos como objetos de notación literal, son una forma de definir y crear objetos en JavaScript utilizando una sintaxis simple y concisa. Los objetos literales se definen entre llaves {} y consisten en pares de clave-valor separados por comas.

// Ejemplo de objeto literal

let persona = {

nombre: 'Juan',

edad: 30,

ocupacion: 'Desarrollador',

saludar: function() {

console.log('¡Hola! Soy ' + this.nombre + ', tengo ' + this.edad + ' años y soy ' + this.ocupacion + '.');

}

};

// Accediendo a propiedades y métodos del objeto literal

console.log(persona.nombre); // Imprime: Juan

console.log(persona.edad); // Imprime: 30

console.log(persona.ocupacion); // Imprime: Desarrollador

persona.saludar(); // Imprime: ¡Hola! Soy Juan, tengo 30 años y soy Desarrollador.

En este ejemplo, el objeto literal **persona** tiene cuatro propiedades: **nombre**, **edad**, **ocupacion** y **saludar**, donde **nombre**, **edad** y **ocupacion** son pares de clave-valor que representan información sobre una persona, y **saludar** es un método que imprime un mensaje de saludo utilizando las propiedades del objeto.

**Que son los parámetros REST y Operador Spread (de un ejemplo de cada uno)**

Los parámetros REST y el operador Spread son características de JavaScript que se utilizan para trabajar con un número variable de argumentos en funciones y para clonar objetos y arreglos de una manera más conveniente, respectivamente.

**Parámetros REST**:

* Los parámetros REST permiten a una función aceptar un número variable de argumentos como un arreglo.

// Ejemplo de parámetros REST

function sumar(...numeros) {

let total = 0;

for (let num of numeros) {

total += num;

}

return total;

}

console.log(sumar(1, 2, 3, 4, 5)); // Imprime: 15

console.log(sumar(10, 20, 30)); // Imprime: 60

En este ejemplo, la función **sumar** acepta un número variable de argumentos utilizando el parámetro REST **...numeros**, que recopila todos los argumentos pasados a la función en un arreglo llamado **numeros**. La función luego suma todos los números dentro del arreglo **numeros** y devuelve el resultado.

1. **Operador Spread**:
   * El operador Spread se utiliza para expandir los elementos de un arreglo o los pares clave-valor de un objeto en un lugar donde se esperan múltiples elementos.

// Ejemplo de operador Spread con arreglos

let arreglo1 = [1, 2, 3];

let arreglo2 = [...arreglo1, 4, 5, 6];

console.log(arreglo2); // Imprime: [1, 2, 3, 4, 5, 6]

// Ejemplo de operador Spread con objetos

let objeto1 = { a: 1, b: 2 };

let objeto2 = { ...objeto1, c: 3, d: 4 };

console.log(objeto2); // Imprime: { a: 1, b: 2, c: 3, d: 4 }

**Que son las arrow functions (de un ejemplo)**

Las arrow functions, también conocidas como funciones flecha, son una característica introducida en ECMAScript 6 (también conocido como ES6) de JavaScript. Proporcionan una sintaxis más concisa para definir funciones en comparación con las funciones tradicionales, y tienen un comportamiento especial con respecto al alcance (this). Aquí tienes un ejemplo de una arrow function:

// Ejemplo de arrow function

const sumar = (a, b) => {

return a + b;

};

console.log(sumar(3, 5)); // Imprime: 8

En este ejemplo, **sumar** es una arrow function que toma dos parámetros **a** y **b**, y devuelve la suma de ambos. La sintaxis de la arrow function es **(parametros) => { cuerpo de la función }**, donde los parámetros están entre paréntesis y el cuerpo de la función está entre llaves. En este caso, el cuerpo de la función es una expresión que devuelve la suma de **a** y **b**.

**Que son los prototipos (de un ejemplo)**

cada objeto tiene un enlace interno a otro objeto llamado "prototipo" (prototype en inglés). Los prototipos permiten compartir propiedades y métodos entre múltiples objetos de manera eficiente, lo que es una parte clave del sistema de herencia en JavaScript. Cuando se accede a una propiedad o método en un objeto y este no está definido en el propio objeto, JavaScript busca en su prototipo y continúa buscando en la cadena de prototipos hasta encontrar la propiedad o método deseado.

// Definición de un objeto persona

function Persona(nombre, edad) {

this.nombre = nombre;

this.edad = edad;

}

// Agregando un método al prototipo de Persona

Persona.prototype.saludar = function() {

console.log('¡Hola! Mi nombre es ' + this.nombre + ' y tengo ' + this.edad + ' años.');

};

// Creando una instancia de Persona

let juan = new Persona('Juan', 30);

// Llamando al método saludar

juan.saludar(); // Imprime: ¡Hola! Mi nombre es Juan y tengo 30 años.

En este ejemplo, primero se define una función constructora llamada **Persona**, que tiene dos propiedades: **nombre** y **edad**. Luego, se agrega un método llamado **saludar** al prototipo de **Persona** utilizando **Persona.prototype**. Este método estará disponible para todas las instancias de **Persona**, pero no se duplicará en cada instancia.

**Que es la herencia prototípica (de un ejemplo)**

La herencia prototípica es un modelo de herencia en JavaScript que se basa en el concepto de prototipos. En este modelo, cada objeto tiene un prototipo que sirve como plantilla para crear el objeto y proporciona propiedades y métodos compartidos entre los objetos. Cuando se accede a una propiedad o método en un objeto y este no está definido en el propio objeto, JavaScript busca en su cadena de prototipos hasta encontrar la propiedad o método deseado.

// Definición del prototipo Persona

function Persona(nombre, edad) {

this.nombre = nombre;

this.edad = edad;

}

// Agregando un método al prototipo de Persona

Persona.prototype.saludar = function() {

console.log('¡Hola! Mi nombre es ' + this.nombre + ' y tengo ' + this.edad + ' años.');

};

// Definición del prototipo Estudiante que hereda de Persona

function Estudiante(nombre, edad, grado) {

Persona.call(this, nombre, edad); // Llamada al constructor de Persona

this.grado = grado;

}

// Herencia prototípica: Estudiante hereda de Persona

Estudiante.prototype = Object.create(Persona.prototype);

Estudiante.prototype.constructor = Estudiante;

// Agregando un método adicional al prototipo de Estudiante

Estudiante.prototype.estudiar = function() {

console.log(this.nombre + ' está estudiando en el grado ' + this.grado + '.');

};

// Creando una instancia de Estudiante

let estudiante1 = new Estudiante('María', 25, 'Décimo');

// Llamando a métodos de Estudiante y Persona

estudiante1.saludar(); // Método heredado de Persona: ¡Hola! Mi nombre es María y tengo 25 años.

estudiante1.estudiar(); // Método propio de Estudiante: María está estudiando en el grado Décimo.

En este ejemplo, primero se define el prototipo **Persona** con un constructor que acepta **nombre** y **edad**, y se agrega un método **saludar** al prototipo **Persona**. Luego se define el prototipo **Estudiante** que hereda de **Persona**, utilizando **Object.create(Persona.prototype)** para establecer la cadena de prototipos.

**Que son los métodos estáticos, getters y setters (de un ejemplo de cada uno)**

En JavaScript, los métodos estáticos, getters y setters son técnicas que se utilizan para manipular y acceder a propiedades de clases de una manera controlada y conveniente. Aquí tienes un ejemplo de cada uno:

1. **Métodos estáticos**:
   * Los métodos estáticos son métodos asociados a la clase en sí misma, en lugar de a las instancias individuales de la clase. Se invocan en la clase en lugar de en las instancias y generalmente se utilizan para realizar operaciones que no dependen de las instancias de la clase.

// Ejemplo de método estático

class Calculadora {

static sumar(a, b) {

return a + b;

}

}

console.log(Calculadora.sumar(3, 5)); // Imprime: 8

**Getters**:

* Los getters son métodos que se utilizan para obtener el valor de una propiedad de un objeto. Permiten acceder a una propiedad de forma similar a la lectura de una variable, pero permiten realizar operaciones adicionales antes de devolver el valor.

// Ejemplo de getter

class Persona {

constructor(nombre) {

this.\_nombre = nombre;

}

get nombre() {

return this.\_nombre.toUpperCase();

}

}

let persona = new Persona('Juan');

console.log(persona.nombre); // Imprime: JUAN

**Setters**:

* Los setters son métodos que se utilizan para asignar un valor a una propiedad de un objeto. Permiten realizar validaciones u operaciones adicionales antes de asignar el valor.

// Ejemplo de setter

class Rectangulo {

constructor(ancho, altura) {

this.\_ancho = ancho;

this.\_altura = altura;

}

set ancho(valor) {

if (valor > 0) {

this.\_ancho = valor;

} else {

console.error('El ancho debe ser mayor que cero.');

}

}

}

let rectangulo = new Rectangulo(10, 20);

rectangulo.ancho = 15; // Asigna un nuevo valor al ancho

console.log(rectangulo); // Imprime: Rectangulo { \_ancho: 15, \_altura: 20 }

rectangulo.ancho = -5; // Intenta asignar un valor negativo al ancho, lo que genera un error

**Que es el objeto console (de un ejemplo)**

El objeto console es una característica integrada en los navegadores web y en el entorno de ejecución de JavaScript que proporciona métodos para interactuar con la consola del navegador y mostrar mensajes de depuración, advertencias, errores, etc. Es muy útil para desarrolladores durante el proceso de depuración y prueba de aplicaciones web.

// Ejemplo de uso del objeto console

let nombre = "Juan";

let edad = 30;

let ciudad = "Barcelona";

console.log("Bienvenido a nuestra aplicación.");

console.log("Nombre:", nombre);

console.log("Edad:", edad);

console.warn("¡Atención! Esta es una advertencia.");

console.error("Se ha producido un error en la aplicación.");

console.info("Información adicional sobre el proceso.");

console.table({ Nombre: nombre, Edad: edad, Ciudad: ciudad });

En este ejemplo, se utilizan varios métodos del objeto **console**:

* **console.log()**: Muestra mensajes informativos en la consola.
* **console.warn()**: Muestra mensajes de advertencia en la consola.
* **console.error()**: Muestra mensajes de error en la consola.
* **console.info()**: Muestra mensajes de información en la consola.
* **console.table()**: Muestra datos en forma de tabla en la consola.

**Que es el objeto date (de un ejemplo)**

El objeto Date en JavaScript se utiliza para trabajar con fechas y horas. Proporciona métodos para obtener y establecer diferentes componentes de la fecha y la hora, como el año, el mes, el día, la hora, el minuto, el segundo, etc.

// Ejemplo de uso del objeto Date

let fechaActual = new Date();

// Obtener la fecha y hora actual

console.log("Fecha y hora actual:", fechaActual);

// Obtener el año actual

let anio = fechaActual.getFullYear();

console.log("Año actual:", anio);

// Obtener el mes actual (los meses se indexan desde 0)

let mes = fechaActual.getMonth() + 1;

console.log("Mes actual:", mes);

// Obtener el día del mes

let dia = fechaActual.getDate();

console.log("Día del mes:", dia);

// Obtener el día de la semana (0 para Domingo, 1 para Lunes, ..., 6 para Sábado)

let diaSemana = fechaActual.getDay();

console.log("Día de la semana:", diaSemana);

// Obtener la hora actual

let hora = fechaActual.getHours();

console.log("Hora actual:", hora);

// Obtener el minuto actual

let minuto = fechaActual.getMinutes();

console.log("Minuto actual:", minuto);

// Obtener el segundo actual

let segundo = fechaActual.getSeconds();

console.log("Segundo actual:", segundo);

En este ejemplo, primero se crea un objeto **Date** llamado **fechaActual**, que representa la fecha y hora actuales. Luego, se utilizan varios métodos del objeto **Date** para obtener diferentes componentes de la fecha y la hora, como el año, el mes, el día, la hora, el minuto y el segundo.

**Que es el objeto Math (de un ejemplo)**

El objeto Math en JavaScript proporciona un conjunto de funciones y constantes matemáticas predefinidas que se pueden utilizar para realizar operaciones matemáticas comunes. Estas funciones incluyen funciones trigonométricas, funciones exponenciales, funciones de redondeo, funciones de generación de números aleatorios, entre otras.

// Ejemplo de uso del objeto Math

// Obtener el valor absoluto de un número

let numero1 = -10;

let valorAbsoluto = Math.abs(numero1);

console.log("Valor absoluto de", numero1, ":", valorAbsoluto);

// Redondear un número hacia abajo al entero más cercano

let numero2 = 3.7;

let redondeoAbajo = Math.floor(numero2);

console.log("Redondeo hacia abajo de", numero2, ":", redondeoAbajo);

// Redondear un número hacia arriba al entero más cercano

let numero3 = 3.2;

let redondeoArriba = Math.ceil(numero3);

console.log("Redondeo hacia arriba de", numero3, ":", redondeoArriba);

// Obtener el máximo de dos números

let maximo = Math.max(5, 10);

console.log("Máximo entre 5 y 10:", maximo);

// Obtener el mínimo de dos números

let minimo = Math.min(5, 10);

console.log("Mínimo entre 5 y 10:", minimo);

// Generar un número aleatorio entre 0 (inclusive) y 1 (exclusivo)

let aleatorio = Math.random();

console.log("Número aleatorio:", aleatorio);

En este ejemplo, se utilizan varios métodos del objeto **Math**:

* **Math.abs()**: Devuelve el valor absoluto de un número.
* **Math.floor()**: Redondea un número hacia abajo al entero más cercano.
* **Math.ceil()**: Redondea un número hacia arriba al entero más cercano.
* **Math.max()**: Devuelve el máximo de los argumentos dados.
* **Math.min()**: Devuelve el mínimo de los argumentos dados.
* **Math.random()**: Devuelve un número aleatorio entre 0 (inclusive) y 1 (exclusivo).

**Que es el operador de cortocircuito en javascript (de un ejemplo)**

El operador de cortocircuito en JavaScript es una técnica que se utiliza en expresiones lógicas para mejorar la eficiencia y evitar la evaluación innecesaria de operandos. Cuando se encuentra un operador de cortocircuito, la evaluación se detiene tan pronto como se determina el resultado final de la expresión, sin necesidad de evaluar los operandos restantes.

Los dos operadores de cortocircuito en JavaScript son **&&** (AND lógico) y **||** (OR lógico).

**Operador && (AND lógico)**:

* Si el primer operando es falso, la expresión se evalúa como falsa y se detiene la evaluación. Si el primer operando es verdadero, la expresión se evalúa utilizando el segundo operando.

// Ejemplo de operador de cortocircuito &&

let resultado = false && hacerAlgo(); // Como el primer operando es falso, no se evalúa la función hacerAlgo()

console.log(resultado); // Imprime: false

function hacerAlgo() {

console.log("Haciendo algo...");

return true;

}

En este ejemplo, como el primer operando de **&&** es falso, la función **hacerAlgo()** nunca se llama y no se imprime "Haciendo algo...". Esto muestra cómo el operador de cortocircuito **&&** evita la evaluación innecesaria cuando el primer operando ya determina el resultado final.

**Operador || (OR lógico)**:

* Si el primer operando es verdadero, la expresión se evalúa como verdadera y se detiene la evaluación. Si el primer operando es falso, la expresión se evalúa utilizando el segundo operando.

// Ejemplo de operador de cortocircuito ||

let resultado = true || hacerAlgo(); // Como el primer operando es verdadero, no se evalúa la función hacerAlgo()

console.log(resultado); // Imprime: true

function hacerAlgo() {

console.log("Haciendo algo...");

return false;

}

En este ejemplo, como el primer operando de **||** es verdadero, la función **hacerAlgo()** nunca se llama y no se imprime "Haciendo algo...". Esto demuestra cómo el operador de cortocircuito **||** evita la evaluación innecesaria cuando el primer operando ya determina el resultado final.

**Que es el alert, confirm y pormpot (de un ejemplo de cada una)**

**alert():**

* La función alert() muestra un cuadro de diálogo con un mensaje y un botón de "Aceptar". Es útil para mostrar mensajes de alerta o información importante al usuario.

// Ejemplo de alert

alert("¡Hola! Este es un mensaje de alerta.");

**confirm()**:

* La función **confirm()** muestra un cuadro de diálogo con un mensaje, un botón de "Aceptar" y un botón de "Cancelar". Devuelve **true** si el usuario hace clic en "Aceptar" y **false** si hace clic en "Cancelar".

// Ejemplo de confirm

let respuesta = confirm("¿Estás seguro de que quieres continuar?");

console.log(respuesta); // true si se hace clic en "Aceptar", false si se hace clic en "Cancelar"

**prompt()**:

* La función **prompt()** muestra un cuadro de diálogo con un mensaje, un campo de entrada de texto y botones de "Aceptar" y "Cancelar". Devuelve el valor ingresado por el usuario como una cadena de texto, o **null** si se hace clic en "Cancelar".

// Ejemplo de prompt

let nombre = prompt("Por favor, ingresa tu nombre:");

console.log("Hola, " + nombre + "! Bienvenido.");

**Que son las expresiones regulares (de un ejemplo)**

Las expresiones regulares, también conocidas como regex, son patrones de búsqueda y manipulación de texto utilizados en programación. Permiten realizar búsquedas avanzadas, validaciones y manipulaciones de cadenas de texto de manera muy flexible y poderosa. Las expresiones regulares se componen de caracteres literales y metacaracteres que representan clases de caracteres, repeticiones, posiciones, etc.

// Ejemplo de expresión regular para buscar números de teléfono

let texto = "Mi número de teléfono es 123-456-7890. Llámame.";

let patron = /\d{3}-\d{3}-\d{4}/; // Expresión regular para buscar números de teléfono en formato ###-###-####

let resultado = patron.exec(texto);

if (resultado) {

console.log("Número de teléfono encontrado:", resultado[0]);

} else {

console.log("Número de teléfono no encontrado.");

}

En este ejemplo, la expresión regular **/d{3}-\d{3}-\d{4}/** busca un patrón de tres dígitos seguido de un guion, tres dígitos más seguido de otro guion, y cuatro dígitos más. Cuando se ejecuta esta expresión regular en el texto, encuentra el número de teléfono "123-456-7890".

**Que son las funciones anónimas autoejecutables (de un ejemplo)**

// Ejemplo de función anónima autoejecutable

(function() {

let mensaje = "¡Hola desde la función anónima autoejecutable!";

console.log(mensaje);

})();

En este ejemplo, se define una función anónima entre paréntesis **(function() { ... })** y se agrega un par de paréntesis adicionales **( ... )** al final de la declaración de la función para ejecutarla inmediatamente. Dentro de la función, se imprime un mensaje en la consola.

**Que son los módulos (Import y export) (de un ejemplo)**

Los módulos en JavaScript son una forma de organizar y reutilizar código al dividirlo en archivos separados. Los módulos permiten definir variables, funciones, clases y otros recursos en un archivo y luego exportarlos para que estén disponibles para su uso en otros archivos. Además, los módulos pueden importar recursos de otros archivos para utilizarlos en su propio código. Aquí tienes un ejemplo de cómo se utilizan los módulos con las declaraciones import y export:

Supongamos que tenemos dos archivos: **modulo1.js** y **modulo2.js**.

En **modulo1.js**, definimos una función y la exportamos para que esté disponible para otros módulos:

// modulo1.js

export function saludar(nombre) {

console.log(`¡Hola, ${nombre}!`);

}

En **modulo2.js**, importamos la función **saludar** del módulo **modulo1.js** y la utilizamos:

// modulo2.js

import { saludar } from './modulo1.js';

saludar('Juan');

En este ejemplo, **modulo2.js** importa la función **saludar** del módulo **modulo1.js** utilizando la declaración **import { saludar } from './modulo1.js';**. Luego, invoca la función **saludar('Juan');** para imprimir "¡Hola, Juan!" en la consola.