Material descargable

egg

Teoría **Javascript II**

¡Hola! 👋 Te damos la bienvenida a Javascript II

En este módulo exploraremos aspectos avanzados y fundamentales de JavaScript que te permitirán elevar tus habilidades de programación a un nivel superior.

Nos sumergiremos en el fascinante mundo de la Programación Orientada a Objetos (POO). Aprenderás cómo diseñar y crear objetos, clases y métodos, permitiéndote escribir código más modular, reutilizable y estructurado.

Además, exploraremos el Document Object Model (DOM), una representación en memoria de la estructura de una página web que te permitirá manipular y modificar su contenido de manera dinámica. Con el DOM, podrás hacer que tus sitios web cobren vida y respondan a las interacciones del usuario.

Y eso no es todo, abordaremos el tema de las promesas, una poderosa herramienta para trabajar con operaciones asíncronas y evitar el anidamiento excesivo de funciones. Aprenderás cómo manejar eficazmente las operaciones que llevan tiempo, como las peticiones a servidores y el acceso a bases de datos.

¡Prepárate para desarrollar habilidades prácticas y conocimientos sólidos en JavaScript! Al finalizar este curso, serás capaz de construir aplicaciones web más interactivas, eficientes y profesionales, lo que te abrirá las puertas a nuevas oportunidades y desafíos emocionantes en el mundo del desarrollo web.

¡Que comience el viaje! 🚀

Programación Orientada a Objetos (POO)

Definición y usos

La programación orientada a objetos (POO) es un paradigma de programación que se basa en la creación y manipulación de objetos para resolver problemas. En JavaScript, aunque es un lenguaje principalmente orientado a objetos, también es un lenguaje multi-paradigma, lo que significa que permite implementar tanto programación orientada a objetos como otros enfoques.

En POO, un objeto es una entidad que combina **datos (propiedades)** y **comportamientos (métodos)** relacionados. Los objetos son instancias de clases, que son como plantillas o modelos que definen la estructura y el comportamiento de los objetos.

Vamos a ver los conceptos principales:

¿Qué son las clases?

En JavaScript, una **clase es una plantilla que define la estructura y el comportamiento de los objetos**. Se declaran usando la palabra clave class.

Por ejemplo:

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

  saludar() {
    console.log(`Hola, mi nombre es ${this.nombre} y tengo ${this.edad}
años.`);
  }
}
```

En el ejemplo vemos que la clase tiene un **constructor, que es una particularidad que tienen las clases.** Es obligatorio ponerlo y nos va a indicar las propiedades del objeto.

¿Qué son los objetos?

Los objetos son instancias de clases y se crean utilizando la palabra clave **'new'.**

Por ejemplo:

```
const persona1 = new Persona('Juan', 30);
const persona2 = new Persona('María', 25);

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

¿Qué son las propiedades o atributos?

Son variables que pertenecen a un objeto y almacenan datos relacionados con ese objeto. En el ejemplo anterior**, 'nombre'** y **'edad'** son propiedades de la clase **'Persona'.**

¿Qué son los métodos?

Son funciones que pertenecen a un objeto y definen su comportamiento. En el ejemplo anterior, **´saludar()´** es un método de la clase **'Persona'**. Es decir, las cosas que puede hacer nuestro objeto.

Estos son los conceptos esenciales de la programación orientada a objetos en JavaScript. Con estos elementos, puedes crear estructuras de datos complejas, organizar tu código de manera más eficiente y facilitar la reutilización de código en tus proyectos.

¿Cómo hago para crear una clase?

1. **Definir la clase**:

Para comenzar, usamos la palabra clave class seguida del nombre de la clase. Por convención, los nombres de clase empiezan con una letra mayúscula.

```
class MiClase {
  // Contenido de la clase aquí
}
```

2. **Constructor**:

El constructor es un método especial que se ejecuta cuando se crea una instancia de la clase. Puedes usarlo para inicializar propiedades y realizar configuraciones iniciales.

```
class MiClase {
  constructor(parametro1, parametro2) {
    this.propiedad1 = parametro1;
    this.propiedad2 = parametro2;
  }
}
```

3. **Métodos**:

Puedes agregar métodos a la clase usando la sintaxis de función dentro de la definición de la clase.

```
class MiClase {
  constructor(parametro1, parametro2) {
    this.propiedad1 = parametro1;
    this.propiedad2 = parametro2;
  }

  // Método de la clase
  miMetodo() {
    // Contenido del método aquí
  }
}
```

4. **Crear instancias de la clase**:

Para crear una instancia de la clase, utilizamos la palabra clave new seguida del nombre de la clase y los parámetros del constructor, si los hay.

```
const instancia = new MiClase(valor1, valor2);
```

Entonces, finalmente la clase terminaría construida de la siguiente manera:

```
// Definir la clase "Persona"
class Persona {
  constructor(nombre, edad) {
    this.nombre = nombre;
    this.edad = edad;
  }

  saludar() {
    console.log(`Hola, mi nombre es ${this.nombre} y tengo ${this.edad}
años.`);
  }
}

// Crear una instancia de la clase "Persona"
const persona1 = new Persona("Juan", 30);

// Llamar al método "saludar" de la instancia
persona1.saludar(); // Salida: "Hola, mi nombre es Juan y tengo 30 años."
```

Características de la programación orientada a objetos

Vamos a ver los cuatro pilares principales de la POO: la **abstracción**, la **encapsulación**, la **herencia** y el **polimorfismo**.

¿Qué es la abstracción?

En términos sencillos, la abstracción consiste en identificar las características esenciales e ignorar los detalles irrelevantes o complejos de un objeto para modelarlo en el programa de manera más simple y comprensible. Es decir, llevar al objeto a su mínima expresión comprensible.

La abstracción se logra definiendo una clase que representa una entidad con características y comportamientos específicos. Esta clase actúa como una plantilla o modelo que define las propiedades y métodos relevantes para la entidad, ocultando los detalles internos de implementación que no son necesarios para su uso.

Un ejemplo simple de abstracción podría ser una clase **'Coche'**:

```
class Coche {
  constructor(marca, modelo, año) {
    this.marca = marca;
    this.modelo = modelo;
    this.año = año;
  }

  acelerar() {
    console.log('El coche está acelerando.');
  }

  frenar() {
    console.log('El coche está frenando.');
  }

  obtenerInformacion() {
    return `Marca: ${this.marca}, Modelo: ${this.modelo}, Año:
${this.año}`;
  }
}
```

En este ejemplo, la **clase Coche** representa la **abstracción de un coche** con sus **propiedades (marca, modelo, año)** y sus **métodos (acelerar(), frenar(),**

obtenerInformacion()). Estos métodos representan el comportamiento del coche sin entrar en los detalles internos de cómo se realiza la aceleración o el frenado. Esto permite que otros desarrolladores utilicen esta clase para interactuar con coches en su programa sin necesidad de preocuparse por los detalles de implementación.

La abstracción en POO nos permite manejar la complejidad al dividir el problema en entidades más pequeñas y comprensibles, lo que facilita la creación de software más modular, mantenible y fácil de entender. Al trabajar con objetos abstractos, podemos concentrarnos en lo que hacen y cómo interactuar con ellos, sin necesidad de entender todos los detalles internos de su funcionamiento, lo que da como resultado un código más limpio y eficiente.

¿Qué es el encapsulamiento?

La encapsulación es el concepto de agrupar datos y métodos relacionados en un objeto y ocultar los detalles internos de su funcionamiento. En JavaScript, esto se logra mediante el uso de **métodos públicos y propiedades privadas utilizando la convención del guion bajo (_)**. Aunque es importante destacar que JavaScript no impone un nivel estricto de privacidad, sino que depende de las convenciones de nombramiento para indicar la intención.

¿Qué es la modularidad?

Se refiere a la capacidad de dividir un programa en módulos o unidades más pequeñas y bien definidas, donde cada módulo representa una funcionalidad específica o un conjunto de funcionalidades relacionadas. Cada módulo se implementa como una clase o un conjunto de clases que interactúan entre sí para lograr un objetivo particular.

El principio de modularidad busca simplificar el diseño y la estructura del código, ya que permite que cada parte del programa sea independiente y cohesiva, lo que facilita su comprensión, mantenimiento y reutilización. Además, los módulos pueden ser desarrollados por diferentes equipos o programadores, lo que facilita la colaboración en proyectos más grandes.

Un ejemplo sencillo que ilustra la modularidad en POO puede ser un programa que simula una biblioteca. Podemos dividir el programa en módulos como **Libro**, **Usuario**, **Prestamo**, etc. Cada módulo se implementaría como una clase con sus propios métodos y propiedades.

```
// Módulo Libro
class Libro {
  constructor(titulo, autor, genero) {
    this.titulo = titulo;
    this.autor = autor;
    this.genero = genero;
  }

  // Otros métodos relacionados con los libros...
}

// Módulo Usuario
class Usuario {
  constructor(nombre, edad, tipo) {
    this.nombre = nombre;
    this.edad = edad;
    this.tipo = tipo;
  }

  // Otros métodos relacionados con los usuarios...
}

// Módulo Prestamo
class Prestamo {
  constructor(libro, usuario, fechaInicio, fechaFin) {
    this.libro = libro;
    this.usuario = usuario;
    this.fechaInicio = fechaInicio;
    this.fechaFin = fechaFin;
  }

  // Otros métodos relacionados con los préstamos...
}
```

Cada uno de estos módulos puede ser desarrollado y probado por separado, lo que facilita la tarea de desarrollo. Además, al utilizar interfaces bien definidas para comunicarse entre módulos, se logra un bajo acoplamiento, lo que significa que los cambios en un módulo no afectarán a otros módulos, siempre que la interfaz se mantenga inalterada.

La modularidad es una práctica importante en la programación orientada a objetos y es clave para construir sistemas complejos y extensibles de manera organizada y mantenible.

¿Qué es el polimorfismo?

Se refiere a la capacidad de una clase para tomar diferentes formas o comportarse de diferentes maneras a través de una interfaz común.

En otras palabras, el polimorfismo permite que objetos de diferentes clases que heredan de una misma clase base puedan ser tratados como objetos de la clase base cuando se interactúa con ellos. Esto permite que el código sea más flexible y reusable, ya que un mismo método puede ser invocado en diferentes objetos y producir resultados específicos según el tipo de objeto que lo esté implementando.

¿Qué es la herencia?

La herencia permite que una clase (llamada subclase) herede propiedades y métodos de otra clase (llamada clase base o superclase). Esto permite **reutilizar código y crear una jerarquía de clases**. En JavaScript, se utiliza la palabra clave ´**extends**´ para establecer la herencia.

Por ejemplo:

```
class Empleado extends Persona {
  constructor(nombre, edad, cargo) {
    super(nombre, edad); // Llama al constructor de la clase base
(Persona).
```

```
    this.cargo = cargo;
  }

  presentarse() {
    console.log(`Hola, soy ${this.nombre}, tengo ${this.edad} años y soy
${this.cargo}.`);
  }
}

const empleado1 = new Empleado('Pedro', 35, 'Desarrollador');
empleado1.saludar(); // Output: Hola, mi nombre es Pedro y tengo 35 años.
empleado1.presentarse(); // Output: Hola, soy Pedro, tengo 35 años y soy
Desarrollador.
```

En el ejemplo se ve cómo se reutiliza la clase Persona ya creada, pero se le agrega el parámetro de cargo, por eso se crea la clase Empleado heredando los parámetros anteriores de la clase persona.

super se utiliza en la herencia de clases en JavaScript para acceder a los métodos y propiedades de la superclase desde la subclase, permitiendo una extensión y personalización de la funcionalidad heredada.

Veamos otro ejemplo:

```
// Definición de la superclase Animal
class Animal {
  constructor(nombre) {
    this.nombre = nombre;
  }

  hacerSonido() {
    return "Haciendo sonidos genéricos";
  }
}

// Definición de la subclase Perro que extiende Animal
class Perro extends Animal {
  constructor(nombre, raza) {
    super(nombre); // Llamamos al constructor de la superclase usando
```

```
super()
    this.raza = raza;
  }

  hacerSonido() {
    return "Guau, guau!";
  }

  moverCola() {
    return "Moviendo la cola felizmente";
  }
}

// Creación de una instancia de la subclase Perro
const miPerro = new Perro("Firulais", "Labrador");

// Llamando a los métodos de la subclase y superclase
console.log(miPerro.nombre); // Salida: "Firulais"
console.log(miPerro.raza); // Salida: "Labrador"
console.log(miPerro.hacerSonido()); // Salida: "Guau, guau!"
console.log(miPerro.moverCola()); // Salida: "Moviendo la cola felizmente"
```

En este ejemplo, hemos creado una **superclase Animal** con un constructor que toma un **parámetro nombre** y un **método hacerSonido()**. Luego, definimos una **subclase Perro** que **extiende Animal**. En el **constructor** de Perro, usamos **super(nombre)** para llamar al constructor de la superclase Animal y pasarle el nombre del perro.

Además, hemos **redefinido** el método **hacerSonido**() en la subclase Perro, lo que significa que el método **hacerSonido**() de la superclase Animal se ha reemplazado en la subclase. Por lo tanto, cuando llamamos a **miPerro.hacerSonido()**, se ejecuta el método de la subclase Perro.

El método **moverCola()** es específico de la subclase Perro y no existe en la superclase Animal. Al crear una instancia de Perro y llamar al método moverCola(), podemos ver que solo está disponible en la subclase.

Otras características de la programación orientada a objetos

¿Qué es un método estático?

Es un tipo especial de método que está asociado directamente con la clase en lugar de con instancias individuales de la clase. Esto significa que un método estático se puede llamar directamente desde la clase misma, sin necesidad de crear un objeto o instancia de la clase.

Las principales características de los métodos estáticos son las siguientes:

No requieren una instancia: A diferencia de los métodos de instancia que operan en objetos creados a partir de la clase, los métodos estáticos no necesitan una instancia para ser invocados.

No tienen acceso a las propiedades de instancia: Al no estar asociados con una instancia particular, los métodos estáticos no pueden acceder a las propiedades de instancia de la clase.

No pueden utilizar "this": Debido a que los métodos estáticos no están vinculados a una instancia, no pueden utilizar la referencia "this" dentro de su implementación.

Pueden acceder a métodos estáticos y propiedades estáticas: Los métodos estáticos pueden acceder a otros métodos estáticos y propiedades estáticas de la clase.

Para definir un método estático en una clase, se utiliza la palabra clave static antes del nombre del método.

Por ejemplo:

```
class MathUtils {
  static sumar(a, b) {
    return a + b;
  }

  static restar(a, b) {
    return a - b;
  }
```

```
}

// Llamando a los métodos estáticos directamente desde la clase
console.log(MathUtils.sumar(5, 3)); // Output: 8
console.log(MathUtils.restar(10, 4)); // Output: 6
```

En este ejemplo, **sumar()** y **restar()** son métodos estáticos de la clase **MathUtils**, por lo que podemos invocarlos directamente desde la clase sin necesidad de crear una instancia de **MathUtils**. Estos métodos no necesitan acceder a ninguna propiedad de instancia o estado interno, ya que sus resultados solo dependen de los argumentos que se les pasan.

¿Qué son los métodos accesores (getters, setters)?

Son métodos que permiten acceder y modificar los atributos (propiedades) de un objeto de una manera controlada. Estos métodos ofrecen una forma de encapsulación, lo que significa que ocultan los detalles internos de la implementación de las propiedades y proporcionan una interfaz para interactuar con ellas.

1. **Getter (método de acceso):** Un getter es un método que se utiliza para obtener el valor de una propiedad privada de un objeto. Se define utilizando la palabra clave get seguida del nombre de la propiedad que se quiere obtener. Los getters no toman argumentos y se invocan simplemente llamando al nombre de la propiedad como si fuera un atributo.

Por ejemplo, supongamos que tenemos una **clase Persona** con una **propiedad privada _edad** y un **getter** para acceder a esa propiedad:

```
class Persona {
  constructor(nombre, edad) {
    this.nombre = nombre;
    this._edad = edad;
  }

  get edad() {
```

```
    return this._edad;
  }
}

const persona = new Persona('Juan', 30);
console.log(persona.edad); // Output: 30
```

2. **Setter (método de modificación):** Un setter es un método que se utiliza para modificar el valor de una propiedad privada de un objeto. Se define utilizando la palabra clave set seguida del nombre de la propiedad que se quiere modificar. Los setters toman un argumento que representa el nuevo valor de la propiedad y se invocan como si estuvieras asignando un valor a una propiedad.

Continuando con el ejemplo anterior, agreguemos un setter para modificar la edad de la persona:

```
class Persona {
  constructor(nombre, edad) {
    this.nombre = nombre;
    this._edad = edad;
  }

  get edad() {
    return this._edad;
  }

  set edad(nuevaEdad) {
    if (nuevaEdad > 0) {
      this._edad = nuevaEdad;
    }
  }
}

const persona = new Persona('Juan', 30);
console.log(persona.edad); // Output: 30

persona.edad = 35;
```

```
console.log(persona.edad); // Output: 35

persona.edad = -5; // No se permiten valores negativos
console.log(persona.edad); // Output: 35
```

Los métodos accesores son una forma de garantizar que el acceso y la modificación de los atributos de un objeto se realicen de acuerdo con ciertas reglas o validaciones, proporcionando un mayor control sobre la integridad de los datos y el comportamiento de los objetos. Además, permiten mantener la compatibilidad con código anterior, ya que los métodos accesores se pueden agregar o modificar sin afectar directamente el código que utiliza los atributos.

Ejemplos

✏️ Ejemplo celular

```
// Objeto Celular
const celular = {
  marca: "Samsung",
  modelo: "Galaxy S21",
  color: "Negro",
  almacenamiento: "128GB",
  sistemaOperativo: "Android",
  camaraMP: 64,
  encendido: false,

  // Método para sacar fotos
  sacarFoto: function() {
    if (!this.encendido) {
      console.log("El celular está apagado. Por favor, enciéndelo para
sacar fotos.");
    } else {
      console.log(`Foto tomada con la cámara de ${this.camaraMP} MP en el
${this.modelo}. ¡Sonríe! 📸`);
    }
  },

  // Método para encender el celular
  encender: function() {
    this.encendido = true;
```

```
    console.log("¡El celular está encendido!");
  },

  // Método para apagar el celular
  apagar: function() {
    this.encendido = false;
    console.log("El celular se ha apagado.");
  }
};

// Prueba del objeto Celular
console.log("Información del celular:");
console.log(`Marca: ${celular.marca}`);
console.log(`Modelo: ${celular.modelo}`);
console.log(`Color: ${celular.color}`);
console.log(`Almacenamiento: ${celular.almacenamiento}`);
console.log(`Sistema Operativo: ${celular.sistemaOperativo}`);

// Encendemos el celular
celular.encender();

// Sacamos una foto
celular.sacarFoto();

// Apagamos el celular
celular.apagar();
```

En este ejemplo, el objeto "celular" tiene propiedades como marca, modelo, color, almacenamiento, sistema operativo, y una propiedad "camaraMP" para representar los megapíxeles de la cámara. También cuenta con métodos para sacar fotos, encender el celular y apagarlo.

Al ejecutar este código, verás cómo el objeto "celular" es capaz de sacar una foto solo si está encendido. Si intentas sacar una foto sin encender el celular, mostrará un mensaje indicando que el celular está apagado

✏️ Ejemplo cuenta bancaria

```
// Objeto Cuenta Bancaria
const cuentaBancaria = {
  titular: "Juan Pérez",
  saldo: 1000,

  // Método para depositar dinero en la cuenta
  depositar: function(monto) {
    if (monto <= 0) {
      console.log("El monto a depositar debe ser mayor que cero.");
    } else {
      this.saldo += monto;
      console.log(`Se ha depositado $${monto} en la cuenta. Saldo actual:
$${this.saldo}`);
    }
  },

  // Método para retirar dinero de la cuenta
  retirar: function(monto) {
    if (monto <= 0) {
      console.log("El monto a retirar debe ser mayor que cero.");
    } else if (monto > this.saldo) {
      console.log("Saldo insuficiente. No es posible realizar el retiro.");
    } else {
      this.saldo -= monto;
      console.log(`Se ha retirado $${monto} de la cuenta. Saldo actual:
$${this.saldo}`);
    }
  },

  // Método para consultar el saldo de la cuenta
  consultarSaldo: function() {
    console.log(`Saldo actual: $${this.saldo}`);
  }
};

// Prueba del objeto Cuenta Bancaria
console.log(`Titular de la cuenta: ${cuentaBancaria.titular}`);

cuentaBancaria.consultarSaldo(); // Saldo actual: $1000
```

```
cuentaBancaria.depositar(500); // Se ha depositado $500 en la cuenta. Saldo
actual: $1500

cuentaBancaria.retirar(200); // Se ha retirado $200 de la cuenta. Saldo
actual: $1300

cuentaBancaria.consultarSaldo(); // Saldo actual: $1300
```

En este ejemplo, el objeto "cuentaBancaria" representa una cuenta bancaria con un titular y un saldo. Los métodos "depositar", "retirar" y "consultarSaldo" permiten realizar operaciones bancarias. El saldo se va actualizando de acuerdo a las operaciones realizadas.

Este ejemplo muestra cómo se pueden usar objetos en JavaScript para representar cuentas bancarias y llevar a cabo operaciones sobre ellas. Puedes experimentar con diferentes montos y operaciones para probar el funcionamiento del objeto

---

Métodos de cadenas de caracteres

Cadenas de caracteres

En JavaScript, los métodos de cadenas son funciones predefinidas que se utilizan para manipular y operar con cadenas de texto. Las cadenas son una secuencia de caracteres y, debido a que son muy comunes en la programación, JavaScript proporciona una variedad de métodos para trabajar con ellas.

Algunos de los métodos de cadenas más utilizados son los siguientes:

- **length**: Devuelve la longitud de la cadena (el número de caracteres).
- **charAt(index)**: Devuelve el carácter en la posición especificada por el índice. El primer carácter tiene un índice de 0.
- **indexOf(substring)**: Devuelve el índice de la primera ocurrencia de la subcadena especificada. Si no se encuentra la subcadena, devuelve -1.
- **lastIndexOf(substring)**: Devuelve el índice de la última ocurrencia del subcadena especificada. Si no se encuentra la subcadena, devuelve -1.

- **substring(startIndex, endIndex):** Devuelve una subcadena que comienza en el índice startIndex y se extiende hasta, pero sin incluir, el índice endIndex.
- **slice**(**startIndex**, **endIndex**): Similar a substring, pero también permite índices negativos, que cuentan desde el final de la cadena.
- **toUpperCase**(): Devuelve la cadena en mayúsculas.
- **toLowerCase**(): Devuelve la cadena en minúsculas.
- **trim**(): Devuelve la cadena sin espacios en blanco al principio y al final.
- **split**(**separator**): Divide la cadena en un array de subcadenas utilizando el separator como criterio.
- **replace**(**searchValue**, **replaceValue**): Reemplaza la primera ocurrencia de searchValue con replaceValue.
- **startsWith():** para verificar si una cadena comienza con un determinado prefijo.
- **includes():** para verificar si una cadena contiene una subcadena específica
- **reverse()**: El método reverse() invierte el orden de los elementos de un array, es decir, el primer elemento pasa a ser el último y el último pasa a ser el primero.
- **join()**: El método join() une todos los elementos de un array en una cadena, utilizando un separador especificado. Por defecto, el separador es una coma ,.

💡**Tip:** *Existen muchos métodos de cadenas (String Methods). Te dejamos a continuación un link de W3Schools para que puedas ver todos los que existen*
👉 **String Methods**

Ejemplos

Veamos ejemplos de cómo se pueden utilizar los métodos de cadenas de caracteres:

✏️trim()

```
const textoConEspacios = "    Hola, cómo estás?    ";
```

```
const textoSinEspacios = textoConEspacios.trim();

console.log(textoSinEspacios); // Output: "Hola, cómo estás?"
```

Aquí **trim()** elimina los espacios en blanco al principio y al final de la cadena " Hola, cómo estás? ", y el resultado es "Hola, cómo estás?".

```
function validarNombre(nombre) {
  const nombreValido = nombre.trim();
  return nombreValido !== "" && nombreValido.length >= 3;
}

console.log(validarNombre("  John  ")); // Output: true
console.log(validarNombre("  A  "));    // Output: false
```

Aquí **trim()** se utiliza para asegurarse de que no haya espacios innecesarios alrededor del nombre antes de validar su longitud. Así se garantiza que el nombre tenga al menos 3 caracteres y no esté compuesto solo por espacios en blanco.

✏️startsWith()

```
const texto = "Hola, ¿cómo estás?";
const prefijo = "Hola";

if (texto.startsWith(prefijo)) {
  console.log("El texto comienza con el prefijo.");
} else {
  console.log("El texto NO comienza con el prefijo.");
}
```

En este ejemplo, tenemos una cadena texto que contiene "Hola, ¿cómo estás?" y queremos verificar si comienza con el prefijo "Hola". Utilizamos el método **startsWith()** para hacer la comprobación. En este caso, el resultado sería "El texto comienza con el prefijo.", ya que la cadena texto efectivamente comienza con el prefijo "Hola". Si cambias el valor de prefijo a otra cadena, el resultado cambiará en consecuencia.

✏️includes()

```
const texto = "La programación es divertida";
const subcadena = "divertida";

if (texto.includes(subcadena)) {
  console.log("El texto contiene la subcadena.");
} else {
  console.log("El texto NO contiene la subcadena.");
}
```

En este ejemplo, tenemos una cadena texto que dice "La programación es divertida" y queremos verificar si contiene la subcadena "divertida". Utilizamos el método **includes()** para hacer la comprobación. En este caso, el resultado sería "El texto contiene la subcadena.", ya que la cadena texto efectivamente contiene la palabra "divertida". Si cambias el valor de subcadena a otra cadena, el resultado cambiará en consecuencia.

✏️split()

```
const texto = "Hola, cómo estás, hoy?";
const arraySubcadenas = texto.split(", ");

console.log(arraySubcadenas);
```

Output:

```
[ 'Hola', 'cómo estás', 'hoy?' ]
```

En este ejemplo, tenemos una cadena **texto** que dice "Hola, cómo estás, hoy?" y queremos dividirla en un array de subcadenas utilizando la coma seguida de un espacio como delimitador. Utilizamos el método **split(", ")**, donde **", "** es el delimitador. El método **split()** busca todas las ocurrencias del delimitador en la cadena y divide la cadena en subcadenas en esos puntos.

El resultado, almacenado en **arraySubcadenas**, será un array que contiene las subcadenas resultantes después de dividir la cadena original por el delimitador. En este caso, el array resultante sería **['Hola', 'cómo estás', 'hoy?'].**

Puedes usar diferentes delimitadores para dividir la cadena en función de tus necesidades, como espacios, puntos, comas, etc. Además, puedes obtener el número de elementos en el array resultante utilizando **arraySubcadenas.length.**

✏️indexOf()

```
const texto = "La programación es divertida";
const subcadena = "divertida";

const posicion = texto.indexOf(subcadena);

if (posicion !== -1) {
  console.log(`La subcadena "${subcadena}" se encuentra en la posición
${posicion}.`);
} else {
  console.log(`La subcadena "${subcadena}" no se encuentra en el texto.`);
}
```

En este ejemplo, tenemos una cadena texto que dice "La programación es divertida" y queremos obtener la posición donde se encuentra la subcadena "divertida". Utilizamos el método indexOf() para realizar esta búsqueda. Si la subcadena se encuentra en la cadena, indexOf() devolverá la posición de la primera ocurrencia de la subcadena. Si la subcadena no se encuentra en la cadena, indexOf() devolverá -1.

En este caso, el resultado sería "La subcadena "divertida" se encuentra en la posición 21.", ya que la subcadena se encuentra a partir del índice 21 en la cadena texto. Si cambias el valor de subcadena a otra cadena, el resultado cambiará en consecuencia, indicando la posición de la nueva subcadena en el texto o si no se encuentra.

✏️reverse() & join()

```
function reverseString(str) {
  // Convierte la cadena en un array de caracteres utilizando split('')
  const arrayCaracteres = str.split('');
```

```
  // Invierte el orden de los caracteres utilizando reverse()
  const arrayReverso = arrayCaracteres.reverse();

  // Une los caracteres invertidos en una nueva cadena utilizando join('')
  const cadenaReversa = arrayReverso.join('');

  return cadenaReversa;
}

const original = "Hola, ¿cómo estás?";
const invertida = reverseString(original);

console.log("Original:", original);
console.log("Invertida:", invertida);

//Output
//Original: Hola, ¿cómo estás?
//Invertida: ?sátse moc¿ ,aloH
```

En este ejemplo, la función **reverseString()** recibe una cadena str, la convierte en un array de caracteres con **split(' ')**, luego invierte el orden de los caracteres en el array con **reverse()**, y finalmente une los caracteres invertidos en una nueva cadena utilizando **join(' ')**. Como resultado, obtenemos la cadena original invertida.

Expresiones Regulares (RegExp)

RegExp (Expresiones Regulares) en JavaScript es una herramienta poderosa que te permite buscar, validar y manipular patrones específicos en cadenas de texto. Una expresión regular es una secuencia de caracteres que define un patrón de búsqueda y se utiliza con métodos de cadenas como **match(), test(), replace(), split()**, entre otros.

En JavaScript, puedes crear una expresión regular utilizando el **constructor RegExp** o utilizando una **sintaxis literal entre barras /**.

1. Sintaxis con el **constructor RegExp**:

```
const regex = new RegExp('patrón');
```

2. Sintaxis literal entre barras /:

```
const regex = /patrón/;
```

Donde **'patrón'** es el patrón que deseas buscar en la cadena. Los patrones pueden incluir letras, números, caracteres especiales y metacaracteres que tienen significados especiales dentro de las expresiones regulares.

Algunos metacaracteres comunes son:

- .: Representa cualquier carácter excepto una nueva línea.
- *: Coincide con el elemento anterior 0 o más veces.
- +: Coincide con el elemento anterior 1 o más veces.
- ?: Coincide con el elemento anterior 0 o 1 vez (hace que el elemento sea opcional).
- \: Se utiliza para escapar metacaracteres para que sean tratados como caracteres literales.
- []: Define una clase de caracteres, coincidirá con cualquier carácter dentro de los corchetes.
- (): Define un grupo de captura, permite agrupar partes de la expresión regular.
- |: Se utiliza para realizar una coincidencia "o" (OR) entre dos patrones.

Algunos métodos de cadenas que aceptan expresiones regulares son:

- match(): Devuelve un array con las coincidencias encontradas.
- test(): Devuelve true si se encuentra al menos una coincidencia, de lo contrario, false.
- replace(): Reemplaza las coincidencias encontradas con otro texto.
- search(): Devuelve la posición de la primera coincidencia encontrada.

✏️ Veamos un ejemplo

Buscar un teléfono

```
const texto = "Hola, mi número de teléfono es 123-456-7890";
const patronTelefono = /\d{3}-\d{3}-\d{4}/;

const telefonoEncontrado = texto.match(patronTelefono);
console.log(telefonoEncontrado); // Output: ["123-456-7890"]
```

En este ejemplo, la expresión regular **/\d{3}-\d{3}-\d{4}/** busca un patrón de número de teléfono en el texto y encuentra "123-456-7890".

Las expresiones regulares son una herramienta extremadamente útil y flexible para trabajar con cadenas de texto y realizar operaciones complejas de búsqueda y manipulación. Sin embargo, pueden ser complejas de entender al principio, pero a medida que te familiarices con ellas, te permitirán realizar tareas avanzadas con mayor facilidad.

Reemplazar palabras con expresiones regulares

```
function reemplazarPalabraConRegex(cadena, palabraABuscar,
palabraReemplazo) {
  // Utilizamos la sintaxis con barras para crear la expresión regular con
la palabra a buscar y los flags 'gi' (global e insensitive)
  const regex = new RegExp(palabraABuscar, 'gi');

  // Utilizamos el método replace() con la expresión regular para realizar
el reemplazo
  const nuevaCadena = cadena.replace(regex, palabraReemplazo);

  return nuevaCadena;
}

const original = "La casa es bonita, la casa es grande.";
const nuevaCadena = reemplazarPalabraConRegex(original, "casa", "hogar");

console.log("Original:", original);
console.log("Reemplazada:", nuevaCadena);
```

```
//Original: La casa es bonita, la casa es grande.
//Reemplazada: La hogar es bonita, la hogar es grande.
```

En este ejercicio, definimos una función reemplazarPalabraConRegex() que acepta una cadena, la palabra a buscar y la palabra de reemplazo. Luego, utilizamos la sintaxis con barras para crear una expresión regular con los flags 'gi' (global e insensitive). La expresión regular se utiliza con el método replace() para realizar el reemplazo de todas las ocurrencias de la palabra a buscar con la palabra de reemplazo.

💡 **Expresiones Regulares es un tema muy abarcativo, por lo que les dejamos el siguiente recurso para que puedan seguir aprendiendo sobre este tema**
👉 **Expresiones Regulares**

ARRAYS

¿Qué es un array?

Un array es una estructura de datos que te permite almacenar y organizar múltiples elementos en una sola variable. Los arrays son una de las estructuras de datos más fundamentales y utilizadas en la programación y son útiles para trabajar con conjuntos de datos.

Para crear un array en JavaScript, puedes usar la siguiente sintaxis:

```
// Array vacío
let miArray = [];

// Array con elementos
let miArrayConElementos = [1, 2, 3, 4, 5];
```

Los arrays en JavaScript son dinámicos, lo que significa que pueden cambiar de tamaño y puedes agregar o eliminar elementos en cualquier momento. También pueden contener diferentes tipos de datos en sus elementos, como números, cadenas, objetos y otras arrays.

Acceder a elementos del array

Puedes acceder a los elementos de un array utilizando su índice. El índice de un array comienza desde 0 para el primer elemento y aumenta en 1 para cada elemento subsiguiente. Para acceder a un elemento específico, simplemente proporciona el índice entre corchetes:

```
let miArray = [10, 20, 30, 40, 50];

console.log(miArray[0]); // Muestra 10
console.log(miArray[3]); // Muestra 40
```

Modificar elementos del array

Para modificar un elemento en el array, simplemente accede a su posición y asigna un nuevo valor:

```
let miArray = [10, 20, 30, 40, 50];

miArray[2] = 35; // Cambia el tercer elemento a 35

console.log(miArray); // Muestra [10, 20, 35, 40, 50]
```

Longitud del array

Puedes obtener la longitud de un array utilizando la propiedad length:

```
let miArray = [10, 20, 30, 40, 50];

console.log(miArray.length); // Muestra 5
```

Métodos comunes de arrays

Así como y trabajos con métodos para los strings (cadenas de caracteres), JavaScript proporciona una variedad de métodos integrados que facilitan la manipulación de arrays.

Algunos de los métodos más comunes son:

Métodos transformadores: llamamos métodos transformadores a los que generan un cambio en el array que estamos afectando.

- **push**(): Agrega uno o más elementos al final del array.
- **pop**(): Elimina el último elemento del array.
- **shift**(): Elimina el primer elemento del array.
- **unshift**(): Agrega uno o más elementos al inicio del array.
- **reverse**(): Se utiliza para revertir el orden de los elementos de un array
- **splice**(): Permite agregar, eliminar o reemplazar elementos en cualquier posición del array.
- **sort**(): Se utiliza para ordenar los elementos de un array. Por defecto, el método sort() ordena los elementos en función de sus representaciones de cadena Unicode (UTF-16). Esto significa que, para los elementos que son cadenas, se realizará una ordenación alfabética. Para los elementos que son números, también se realizará una ordenación de menor a mayor.

Métodos accesores:

- **slice**(): Devuelve una copia de una porción del array original, especificada por los índices de inicio y fin.
- **join**(): Se utiliza para crear una nueva cadena concatenando todos los elementos de un array. Permite especificar un separador opcional que se utilizará entre los elementos mientras se los une en la cadena resultante. Si no se proporciona ningún separador, los elementos se concatenarán sin ningún carácter intermedio.
- **indexOf**(): Busca un elemento y devuelve su índice en el array.
- **includes**(): Verifica si un elemento está presente en el array y devuelve true o false.

Métodos de repetición/iteración:

- **filter**(): Crea un nuevo array con todos los elementos que cumplan con la condición dada.
- **forEach**(): Se utiliza para iterar sobre los elementos de un array y ejecutar una función de devolución de llamada (callback) para cada elemento

Estos son solo algunos de los muchos métodos disponibles para trabajar con arrays en JavaScript.

Ejemplos

✏️ push()

```
// Definimos un array vacío
let miArray = [];

// Agregamos elementos al array usando push()
miArray.push(10);
miArray.push(20);
miArray.push(30);

console.log(miArray); // Muestra: [10, 20, 30]
```

En este ejemplo, primero creamos un array vacío llamado **miArray**. Luego, utilizamos el método **push()** para agregar tres elementos **(10, 20 y 30)** al final del array. Después de ejecutar las operaciones, el array contiene **[10, 20, 30]**.

Puedes usar **push()** para agregar uno o más elementos a un array. Si deseas agregar más de un elemento en una sola llamada, simplemente agrégales como argumentos separados por comas:

```
let miArray = [1, 2, 3];

miArray.push(4, 5, 6);

console.log(miArray); // Muestra: [1, 2, 3, 4, 5, 6]
```

✏️ pop()

```
let miArray = [10, 20, 30, 40, 50];

// Utilizamos pop() para eliminar el último elemento
let ultimoElemento = miArray.pop();

console.log(ultimoElemento); // Muestra 50
console.log(miArray); // Muestra [10, 20, 30, 40]
```

En este ejemplo, el método pop() elimina el último elemento del array miArray, que es 50, y lo asigna a la variable ultimoElemento. Después de la operación, el array miArray ya no contiene 50.

✏️ sort()

```
// Array de ejemplo
const numeros = [4, 2, 8, 1, 5];

// Ordenar el array de manera ascendente (de menor a mayor)
const numerosAscendente = numeros.slice().sort((a, b) => a - b);

// Ordenar el array de manera descendente (de mayor a menor)
const numerosDescendente = numeros.slice().sort((a, b) => b - a);

// Mostrar resultados
console.log("Array original:", numeros);
console.log("Array ordenado ascendente:", numerosAscendente);
console.log("Array ordenado descendente:", numerosDescendente);
```

Resultado en la consola:

```
Array original: [4, 2, 8, 1, 5]
Array ordenado ascendente: [1, 2, 4, 5, 8]
Array ordenado descendente: [8, 5, 4, 2, 1]
```

En este ejemplo, usamos el método **sort()** en dos ocasiones. Primero, creamos una copia del array original mediante **slice()** para evitar modificar el array original. Luego, pasamos una función de comparación como argumento al método **sort()**.

Esta función de comparación toma dos elementos del array y devuelve un número negativo si el primer elemento debe aparecer antes que el segundo, cero si ambos elementos son iguales en orden y un número positivo si el segundo elemento debe aparecer antes que el primero.

Cuando restamos **b - a**, **ordena de forma descendente** porque si b es mayor que a, el resultado de la resta será positivo, lo que coloca b antes de a. Por otro lado, **si restamos a - b, el orden es ascendente** porque si a es mayor que b, el resultado de la resta será positivo, lo que coloca a antes de b.

✏️ splice()

Supongamos que tenemos un array que contiene los nombres de algunos colores:

```
let colores = ["rojo", "verde", "azul", "amarillo", "naranja"];
```

Ahora, vamos a realizar algunas operaciones utilizando el método splice():

1. **Agregar un nuevo color al inicio del array:**

```
colores.splice(0, 0, "violeta");

console.log(colores); // Muestra ["violeta", "rojo", "verde", "azul",
"amarillo", "naranja"]
```

En este ejemplo, **splice(0, 0, "violeta")** inserta el color "violeta" en la posición 0 (es decir, al inicio) del array colores, sin eliminar ningún elemento.

2. **Reemplazar un color en una posición específica:**

```
colores.splice(2, 1, "blanco");

console.log(colores); // Muestra ["violeta", "rojo", "blanco", "amarillo",
"naranja"]
```

En este caso, splice(2, 1, "blanco") reemplaza un elemento en la posición 2 (que es "azul") con el color "blanco".

3. **Eliminar elementos desde una posición específica:**

```
colores.splice(3, 2);

console.log(colores); // Muestra ["violeta", "rojo", "blanco"]
```

Aquí, splice(3, 2) elimina dos elementos a partir de la posición 3, eliminando "amarillo" y "naranja" del array.

Recuerda que el método **splice()** modifica el array original y puede realizar operaciones de inserción, eliminación o reemplazo, dependiendo de los argumentos que le proporciones. Su sintaxis es **splice(start, deleteCount, item1, item2, ...)**:

- **start**: Índice donde comenzar a realizar las operaciones.
- **deleteCount**: Número de elementos a eliminar a partir de start.
- **item1**, **item2**, ...: Elementos que se agregarán en lugar de los eliminados (opcional).

💡 ***Aclaración:*** *El método **splice()** no se utiliza para agregar elementos al final de un array. En su lugar, se utiliza para insertar, eliminar o reemplazar elementos en una posición específica dentro del array. Para agregar un elemento al final de un array, puedes utilizar el método **push()**.*

Ejemplos iteración

✏️ filter()

La sintaxis básica de **filter()** es la siguiente:

```
const nuevoArreglo = arregloOriginal.filter(funcionCallback(elemento,
index, arreglo) {
  // Condición de filtrado
  // Retornar "true" si se desea incluir el elemento en el nuevo arreglo, o
"false" si se desea excluirlo.
});
```

Los parámetros de la función de callback son opcionales:

- **elemento**: El valor del elemento actual del arreglo que se está procesando.

- **index** (opcional): El índice del elemento actual en el arreglo.
- **arreglo** (opcional): El arreglo original al que se está aplicando el método filter().

La función de callback debe retornar un valor booleano (true o false) que indique si el elemento actual cumple o no con la condición de filtrado. Si el valor retornado es true, el elemento se incluirá en el nuevo arreglo; si es false, el elemento se excluye.

Veamos un ejemplo práctico:

```
const numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

// Filtrar solo los números pares
const numerosPares = numeros.filter(numero => numero % 2 === 0);

console.log(numerosPares); // Salida: [2, 4, 6, 8, 10]
```

En este ejemplo, utilizamos filter() para obtener un nuevo arreglo llamado numerosPares, que solo contiene los números pares del arreglo original numeros. La función de callback (numero => numero % 2 === 0) verifica si el número es divisible por 2 (es par), y si es así, retorna true, lo que resulta en la inclusión del número en el nuevo arreglo. Si el número es impar, la función retorna false, y el número se excluye del nuevo arreglo.

filter() es una herramienta útil cuando necesitas seleccionar ciertos elementos de un arreglo que cumplan con ciertas condiciones, sin modificar el arreglo original. Recuerda que **filter**() no altera el arreglo original, sino que crea un nuevo arreglo con los elementos que pasan el filtro.

✏️ forEach()

La sintaxis del método forEach() es la siguiente:

```
array.forEach(callback(elemento, indice, array))
```

Donde array es el array sobre el cual se va a iterar, y callback es una función que se ejecutará una vez por cada elemento del array. La función callback puede aceptar hasta tres argumentos:

- elemento: El valor del elemento actual del array en cada iteración.
- indice (opcional): El índice del elemento actual dentro del array.
- array (opcional): El array sobre el cual se está iterando.

Es importante destacar que **forEach()** no devuelve nada, es decir, devuelve undefined. Se utiliza principalmente para efectuar operaciones en cada elemento del array, como realizar cálculos, modificar valores o mostrar información.

Ejemplo:

```
// Array de ejemplo
const numeros = [1, 2, 3, 4, 5];

// Usar forEach() para mostrar cada número en la consola
numeros.forEach((numero, indice) => {
  console.log(`Elemento ${indice}: ${numero}`);
});
```

Resultado en la consola:

```
Elemento 0: 1
Elemento 1: 2
Elemento 2: 3
Elemento 3: 4
Elemento 4: 5
```

Una ventaja de forEach() es que facilita el acceso a cada elemento del array y su índice, lo que puede ser útil en diversas situaciones.

Objeto Math

Definición y usos

El objeto "Math" en JavaScript es una utilidad incorporada que proporciona propiedades y métodos para realizar operaciones matemáticas. No es necesario crear una instancia de este objeto, ya que está disponible globalmente en el lenguaje.

El objeto "Math" incluye una variedad de constantes y funciones matemáticas útiles que puedes utilizar en tus programas JavaScript. Algunas de las operaciones más comunes que puedes realizar con el objeto "Math" son:

- Operaciones matemáticas básicas:
    - Math.abs(x): Devuelve el valor absoluto de x (el valor sin signo).
    - Math.ceil(x): Redondea un número hacia arriba al número entero más cercano.
    - Math.floor(x): Redondea un número hacia abajo al número entero más cercano.
    - Math.round(x): Redondea un número al número entero más cercano.
    - Math.max(x1, x2, ..., xn): Devuelve el valor más grande de una lista de números.
    - Math.min(x1, x2, ..., xn): Devuelve el valor más pequeño de una lista de números.
    - Math.random(): Devuelve un número pseudoaleatorio entre 0 (inclusive) y 1 (exclusivo).

Los superiores son los métodos más utilizados para

- Funciones trigonométricas:
    - Math.sin(x): Devuelve el seno de x (en radianes).
    - Math.cos(x): Devuelve el coseno de x (en radianes).
    - Math.tan(x): Devuelve la tangente de x (en radianes).
    - Math.atan(x): Devuelve el arco tangente de x (en radianes).
- Funciones exponenciales y logarítmicas:

    - Math.exp(x): Devuelve el valor de e (la base del logaritmo natural) elevado a la potencia x.
    - Math.log(x): Devuelve el logaritmo natural de x.
    - Math.pow(base, exponente): Devuelve la base elevada al exponente especificado.
- Constantes:
    - Math.PI: El valor de π (pi), aproximadamente 3.141592653589793.
    - Math.E: El número de Euler, aproximadamente 2.718281828459045.

Estos son solo algunos ejemplos de las operaciones que puedes realizar utilizando el objeto "Math" en JavaScript. Es una herramienta muy útil para realizar cálculos matemáticos en tus aplicaciones web o programas.

Ejemplos

✏️ métodos Math.xxx()

Te dejamos los siguientes ejemplos para que puedas ver cómo se utiliza el objeto Math.

```
// Calcular la raíz cuadrada
numero = Math.sqrt(25) //5

//Calcular la raíz cúbica
numero = Math.cbrt(27) //3

// Devolver el número más grande
numero = Math.max(25,3,67,89,70) //89

// Devolver el número más chico
numero = Math.min(25,3,67,89,70) //3

// Numero aletorio entre 0 y 1 no incluye el cero ni el 1
numero = Math.random() //0.68345635345

// Numero aletorio entre 0 y 100 redondeado con metodo round
// redondea al enteero más cerecano
```

```
numero = Math.round(Math.random()*100) //33

// Redondeado para abajo
numero = Math.floor(4.99) //4

// Elimina los decimales
numero = Math.trunc(4.99) //4
```

Este viaje no termina aquí, te esperamos en la siguiente parte teórica 🚀