Módulo 1: Introducción a JavaScript (2 Horas)

1.1. ¿Qué es JavaScript?

JavaScript es un lenguaje de programación ampliamente utilizado en el desarrollo web. Es un componente esencial para la creación de sitios web dinámicos e interactivos. A lo largo de esta introducción, aprenderemos los conceptos fundamentales de JavaScript, cómo se integra en las páginas web y cómo se utiliza para hacer que las páginas web sean más interactivas y atractivas.

JavaScript es un lenguaje de programación que permite a los desarrolladores web agregar funcionalidad interactiva a las páginas web. A diferencia de HTML (HyperText Markup Language) y CSS (Cascading Style Sheets), que se utilizan para definir la estructura y el estilo de una página web, respectivamente, JavaScript se utiliza para programar la lógica y la interacción en una página.

¿Por qué aprender JavaScript?

JavaScript es esencial para el desarrollo web moderno, ya que permite a los desarrolladores:

Crear páginas web dinámicas: JavaScript permite cambiar el contenido de una página en respuesta a acciones del usuario o eventos específicos, como hacer clic en un botón o llenar un formulario.

Validar formularios: Puedes utilizar JavaScript para verificar y validar la información ingresada por los usuarios en formularios web.

Interactuar con el usuario: Con JavaScript, puedes crear ventanas emergentes, alertas y solicitar la confirmación del usuario, lo que mejora la experiencia del usuario.

Realizar solicitudes a servidores: Puedes utilizar JavaScript para hacer solicitudes a servidores web y cargar o enviar datos sin necesidad de recargar la página.

Crear juegos y aplicaciones web: JavaScript es la base de muchas aplicaciones web interactivas y juegos en línea.

¿Dónde se ejecuta JavaScript?

JavaScript se ejecuta en el navegador web del usuario. Los navegadores modernos, como Google Chrome, Mozilla Firefox, Safari y Microsoft Edge, son compatibles con JavaScript. Esto significa que cualquier dispositivo con un navegador web puede ejecutar JavaScript, lo que lo convierte en una herramienta poderosa para crear aplicaciones web multiplataforma.

Sintaxis Básica

JavaScript utiliza una sintaxis similar a otros lenguajes de programación. Aquí hay un ejemplo simple de JavaScript que muestra un mensaje en una ventana emergente cuando se hace clic en un botón en una página web:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Ejemplo de JavaScript</title>
    <script>
     // Definir una función que se ejecutará cuando se haga clic en el botón.
    function mostrarMensaje() {
      alert("¡Hola, Mundo!");
    }
    </script>
  </head>
  <body>
    <button onclick="mostrarMensaje()">Haz clic aquí</button>
  </body>
</html>
```

1.2. Breve historia de Javascript:

A mediados de la década de 1990, la web estaba en constante evolución. La mayoría de las páginas web eran estáticas, compuestas principalmente de texto e imágenes. Sin embargo, un ingeniero de software de Netscape Communications Corporation llamado Brendan Eich se enfrentó a un desafío innovador.

En 1995, Netscape, uno de los navegadores web líderes de la época, se dio cuenta de la necesidad de incorporar un lenguaje de programación en su navegador para hacer que las páginas fueran más interactivas. Brendan Eich fue el encargado de crear un lenguaje que pudiera ejecutarse en el navegador del usuario y proporcionar funcionalidades dinámicas.

En solo diez días, Brendan Eich desarrolló lo que inicialmente se llamó "Mocha" y luego "LiveScript". Sin embargo, Netscape se asoció con Sun Microsystems y, debido al auge de Java en ese momento, decidieron cambiar el nombre a "JavaScript". A pesar del nombre, JavaScript y Java no están relacionados en términos de lenguaje o tecnología.

En diciembre de 1995, JavaScript se presentó con el lanzamiento de Netscape Navigator 2.0, lo que marcó el inicio de una revolución en el desarrollo web. El lenguaje permitía a los desarrolladores web agregar interactividad a las páginas y crear funciones como validación de formularios y manipulación del DOM (Document Object Model).

El éxito de JavaScript llevó a una competencia entre navegadores para adoptar y mejorar el lenguaje, lo que finalmente resultó en la estandarización del lenguaje a través de ECMA International. Esto dio lugar a la versión estándar llamada ECMAScript.

Con el tiempo, JavaScript se convirtió en una parte esencial del desarrollo web, y su versatilidad permitió la creación de aplicaciones web avanzadas, juegos en línea y más. Hoy en día, JavaScript se ejecuta en todos los navegadores modernos y ha dado lugar a una amplia variedad de bibliotecas y frameworks, como jQuery, Angular, React y Vue.js, que facilitan la creación de aplicaciones web más complejas.

JavaScript ha recorrido un largo camino desde su humilde comienzo y sigue siendo una herramienta esencial en el mundo del desarrollo web, impulsando la creación de aplicaciones web interactivas y ricas en contenido.

1.3. Configuración del entorno de desarrollo

Desarrollo en el Navegador

El desarrollo en el navegador web implica la creación, depuración y prueba de código JavaScript directamente en el navegador.

Los navegadores web modernos, como Google Chrome, Mozilla Firefox o Microsoft Edge, vienen con herramientas de desarrollo incorporadas. **<F12>** es el atajo de teclado comúnmente utilizado para acceder a las herramientas de desarrollo.

Los desarrolladores web pueden abrir la Consola del Navegador para ver mensajes de registro, errores y resultados de sus scripts. También pueden inspeccionar y modificar el DOM en tiempo real para depurar problemas en la interfaz de usuario.

Editor de Código:

Para escribir código Javascript debemos utilizar un editor de código. Algunas opciones populares incluyen *Visual Studio Code*, *Sublime Text*, *Atom* y *Notepad++.*

Entorno web:

Existen herramientas que se ejecutan en entornos web que nos permiten escribir código, ejecutarlo y depurarlo. Son especialmente útiles para probar pequeños programas o algoritmos.

Algunas de esas herramientas son:

JSFiddle: JSFiddle es una herramienta en línea que te permite escribir, probar y compartir código JavaScript, HTML y CSS en un entorno de navegador. Es útil para prototipar rápidamente y colaborar en proyectos pequeños.

CodePen: Similar a JSFiddle, CodePen es una plataforma en línea que te permite escribir y probar código JavaScript, HTML y CSS. Ofrece un entorno de desarrollo en vivo y una comunidad activa para compartir proyectos y obtener retroalimentación.

JS Bin: JS Bin es otra herramienta en línea que te permite escribir y ejecutar código JavaScript en un entorno de navegador. Proporciona opciones para ver la salida en tiempo real y compartir proyectos con otros.

StackBlitz: StackBlitz es un entorno de desarrollo en línea para aplicaciones Angular, React y Vue.js. Permite escribir, probar y compartir aplicaciones web de una manera fácil y colaborativa.

1.4. Hola Mundo en JavaScript

Introducción al "Hola Mundo":

"Hola Mundo" es un programa simple que muestra un mensaje en la pantalla, y es una tradición en la programación para empezar con él.

Escribiendo el Código "Hola Mundo" en consola:

Abrimos las herramientas de desarrollo <F12>. Abrimos la pestaña **console** y escribimos:

```
console.log("¡Hola Mundo!");
```

En la siguiente línea aparecerá el mensaje “Hola Mundo!”

Escribiendo el código “Hola Mundo!” en un editor de texto.

Abrimos VSCode y creamos un nuevo fichero llamado test.js. Escribimos:

```
console.log("¡Hola Mundo!");
```

Aparecerá en el terminal integrado del editor el mensaje “Hola Mundo”.

Escribiendo el código “Hola Mundo!” en JSFiddle.

JSFiddle es una plataforma en línea que te permite escribir, ejecutar y compartir código HTML, CSS y JavaScript de manera rápida y sencilla. A continuación, te proporcionaré un pequeño tutorial sobre cómo utilizar JSFiddle para ejecutar código JavaScript:

Acceder a JSFiddle

Abre tu navegador web y ve a https://jsfiddle.net/. No es necesario registrarse para usar JSFiddle, pero si te registras, puedes guardar tus fiddles y acceder a ellos en cualquier momento.

Interfaz de JSFiddle

Una vez en la página de inicio de JSFiddle, verás una interfaz con cuatro paneles:

HTML: Aquí puedes escribir tu código HTML.

CSS: Este panel es para el código CSS.

JavaScript: Aquí es donde escribirás tu código JavaScript.

Resultado: El resultado de tu código se mostrará en este panel.

Escribir tu código

Puedes comenzar a escribir tu código HTML, CSS y JavaScript en los paneles respectivos. Puedes usar las bibliotecas externas, como jQuery, si es necesario, y configurar las opciones del fiddle.

Ejecutar tu código

Para ejecutar tu código, simplemente haz clic en el botón "Run" en la esquina superior izquierda de la pantalla. JSFiddle procesará tu código y mostrará el resultado en el panel "Resultado".

Depuración y consola

Puedes depurar tu código JavaScript usando la consola. Abre la consola haciendo clic en la pestaña "Consola" en la parte inferior del panel de JavaScript. Aquí puedes ver mensajes de error o depuración.

Compartir tu fiddle

Si deseas compartir tu código con otros, puedes hacerlo fácilmente. Haz clic en el botón "Compartir" en la parte superior del panel. JSFiddle generará una URL única que puedes compartir con otros.

Guardar tu fiddle:

Si deseas guardar tu trabajo para futuras referencias o modificaciones, puedes registrarte en JSFiddle y hacer clic en "Save" para guardar tu fiddle. También puedes establecer opciones de privacidad (público, privado o sin listar) para tu fiddle.

Forking (bifurcación) de fiddles:

Puedes bifurcar (hacer una copia) de fiddles existentes haciendo clic en el botón "Fork" en la parte superior del fiddle de otra persona. Esto te permite trabajar en una versión modificada del código original.

Versiones (revisions):

Puedes acceder a las diferentes versiones de un fiddle haciendo clic en "Revisions" y ver el historial de cambios.

Opciones adicionales:

JSFiddle ofrece muchas opciones adicionales, como la selección de versiones de bibliotecas, el ajuste del diseño de la página y más. Explora estas opciones para personalizar tu experiencia.

JSFiddle es una herramienta muy útil para probar y compartir código JavaScript de manera rápida y sencilla. Puedes usarla para experimentar con código, depurar problemas y colaborar con otros desarrolladores en proyectos.

Módulo 2: Variables y Tipos de Datos (2 Horas)

2.1. Variables y declaraciones

¿Qué son las Variables en JavaScript?

Explicación de que las variables son como cajas etiquetadas donde se pueden almacenar datos. Estos datos pueden ser números, texto, objetos, etc.

Mencionar que las variables son esenciales para mantener y manipular información en un programa.

Declaración de Variables:

En JavaScript, se pueden declarar variables utilizando tres palabras clave: var, let y const.

var se utilizaba en versiones antiguas de JavaScript, pero con las versiones más recientes, let y const se consideran las mejores prácticas. Se debe enfatizar el uso de let y const en lugar de var.

Ejemplo de Declaración de Variables:

Proporcionar un ejemplo concreto de cómo declarar una variable utilizando let y const:

```
// Declaración de variables con 'let' (mutable)
let cantidadProductos = 10;
// Declaración de variables con 'const' (inmutable)
const pi = 3.14159;
```

Explicar que las variables declaradas con let pueden cambiar su valor, mientras que las declaradas con const son inmutables y no pueden cambiar una vez que se les asigna un valor.

Nombres de Variables:

Explicar las reglas y convenciones para nombrar variables:

Los nombres de variables son sensibles a mayúsculas y minúsculas.

Deben comenzar con una letra, guion bajo o signo de dólar.

Pueden contener letras, números, guiones bajos o signos de dólar.

Se recomienda utilizar nombres descriptivos y significativos para las variables.

Asignación de Valores:

Cómo se asignan valores a las variables después de declararlas.

Ejemplos de asignación de valores a variables:

```
let nombre = "Juan";
const edad = 25;
```

2.2. Tipos de datos en Javascript

JavaScript es un lenguaje "dinámico", lo que significa que no es necesario declarar explícitamente el tipo de dato al crear una variable. JavaScript determina automáticamente el tipo de dato basándose en el valor que se le asigna. Los principales tipos de datos en JavaScript son:

Números (Number):

Los números pueden ser enteros o decimales.

Ejemplo: let edad = 30; let precio = 19.99;

Cadenas (String):

Las cadenas son secuencias de caracteres.

Ejemplo: let nombre = "Juan";

Booleanos (Boolean):

Los booleanos representan un valor verdadero (true) o falso (false).

Ejemplo: let esMayorDeEdad = true;

Arreglos (Array):

Los arreglos son colecciones ordenadas de datos.

Ejemplo: let colores = ["rojo", "verde", "azul"];

Objetos (Object):

Los objetos son colecciones de propiedades y valores.

Ejemplo:

```
let persona = {
  nombre: "Ana",
  edad: 25,
  ciudad: "Madrid"
};
```

Indefinido (Undefined):

Cuando una variable se declara pero no se le asigna un valor, su tipo es undefined.

Ejemplo:

```
let direccion;
```

Nulo (Null):

El tipo null representa la ausencia intencional de cualquier valor u objeto.

Ejemplo:

```
let resultado = null;
```

2.3 Conversión de Tipos de Datos:

JavaScript permite convertir entre diferentes tipos de datos. Por ejemplo, puedes convertir un número en una cadena o viceversa. Esto es útil al trabajar con datos y realizar cálculos.

Ejemplos de Conversión:

Conversión de número a cadena:

```
let numero = 42;
let cadena = String(numero); // "42"
```

Conversión de cadena a número:

```
let texto = "123";
```

```
let numero = Number(texto); // 123
```

Módulo 3: Operadores y Expresiones (2 Horas)

Los operadores son herramientas fundamentales en la programación que permiten realizar cálculos y tomar decisiones en el código. JavaScript incluye varios tipos de operadores:

3.1. Operadores aritméticos

Introducción a los Operadores Aritméticos:

Explicación de que los operadores aritméticos son símbolos que se utilizan para realizar operaciones matemáticas en JavaScript.

Operadores Aritméticos Básicos:

Suma +.

Resta -.

Multiplicación *.

División /.

Módulo % (que devuelve el residuo de una división).

Suma:

1. Suma de números:

```
var numero1 = 5;
var numero2 = 3;
var resultado = numero1 + numero2;
console.log(resultado); // Muestra 8
```

En este ejemplo, el operador + se utiliza para sumar dos números, numero1 y numero2, y el resultado se almacena en la variable resultado.

2. Concatenación de cadenas:

```
var nombre = "Juan";
var apellido = "Pérez";
var nombreCompleto = nombre + " " + apellido;
console.log(nombreCompleto); // Muestra "Juan Pérez"
```

El operador + se utiliza para concatenar dos cadenas, nombre y apellido, con un espacio en blanco en el medio.

3 Conversión de tipos de datos:

```
var numero = 42;
var cadena = "3";
var suma = numero + Number(cadena);
console.log(suma); // Muestra 45
```

En este caso, se utiliza Number(cadena) para convertir la cadena "3" en un número antes de sumarlo al número 42.

4. Combinación de cadenas y números:

```
var edad = 30;
var mensaje = "Tengo " + edad + " años.";
console.log(mensaje); // Muestra "Tengo 30 años."
```

Aquí, el operador + se usa para combinar una cadena con una variable numérica, creando una cadena que describe la edad.

5. Concatenación de variables y texto:

```
var producto = "zapatos";
var precio = 50;
var descripcion = "Los " + producto + " cuestan $" + precio + ".";
console.log(descripcion); // Muestra "Los zapatos cuestan $50."
```

El operador + se utiliza para combinar variables con texto estático y crear una descripción.

Resta:

1. Resta de números:

```
var numero1 = 10;
var numero2 = 4;
var resultado = numero1 - numero2;
console.log(resultado); // Muestra 6
```

En este ejemplo, el operador - se utiliza para restar numero2 de numero1, y el resultado se almacena en la variable resultado.

2. Resta con variables:

```
var precioTotal = 100;
var descuento = 20;
var precioConDescuento = precioTotal - descuento;
console.log("Precio con descuento: $" + precioConDescuento); // Muestra
"Precio con descuento: $80"
```

El operador - se usa para calcular un precio con un descuento aplicado.

3. Uso en una expresión más compleja:

```
var a = 15;
var b = 7;
var c = 3;
var resultado = a - (b * c);
console.log(resultado); // Muestra 6
```

En este caso, el operador - se utiliza en una expresión más compleja para calcular un resultado basado en valores de varias variables.

4. Cambio de signo:

```
var numero = 8;
var numeroNegativo = -numero;
console.log(numeroNegativo); // Muestra -8
```

El operador - se utiliza para cambiar el signo de un número, convirtiendo un número positivo en negativo o viceversa.

Multiplicacion:

1. Multiplicación de números:

```
var numero1 = 5;
var numero2 = 3;
var resultado = numero1 * numero2;
console.log(resultado); // Muestra 15
```

En este ejemplo, el operador * se utiliza para multiplicar numero1 y numero2, y el resultado se almacena en la variable resultado.

2. Cálculo de área:

```
var largo = 6;
var ancho = 4;
var area = largo * ancho;
console.log("El área del rectángulo es " + area + " unidades cuadradas.");
```

Aquí, el operador * se usa para calcular el área de un rectángulo multiplicando su largo por su ancho.

3. Cálculo de precio total:

```
var precioUnitario = 25;
var cantidad = 3;
var precioTotal = precioUnitario * cantidad;
console.log("El precio total es $" + precioTotal);
```

El operador * se utiliza para calcular el precio total de un producto multiplicando el precio unitario por la cantidad.

4. Potenciación:

```
var base = 2;
var exponente = 3;
var resultado = Math.pow(base, exponente);
console.log(resultado); // Muestra 8
```

Para calcular una potencia, puedes usar el operador * para elevar una base a un exponente.

5. Uso en una expresión más compleja:

```
var a = 4;
var b = 2;
var c = 3;
var resultado = a * (b + c);
console.log(resultado); // Muestra 20
```

El operador * se utiliza en una expresión más compleja para calcular un resultado basado en valores de varias variables.

División:

1. División de números:

```
var dividendo = 10;
var divisor = 2;
var resultado = dividendo / divisor;
console.log(resultado); // Muestra 5
```

En este ejemplo, el operador / se utiliza para dividir el dividendo por el divisor, y el resultado se almacena en la variable resultado.

2. Cálculo de promedio:

```
var suma = 30;
var cantidad = 6;
var promedio = suma / cantidad;
console.log("El promedio es " + promedio);
```

Aquí, el operador / se usa para calcular el promedio dividiendo la suma total por la cantidad de elementos.

3. Cálculo de proporción:

```
var total = 200;
var porcentaje = 25;
var parte = (porcentaje / 100) * total;
console.log("El " + porcentaje + "% de " + total + " es igual a " + parte);
```

En este caso, el operador / se utiliza para calcular una parte proporcional de un valor total en función de un porcentaje.

4. Uso en una expresión más compleja:

```
var a = 15;
var b = 3;
var c = 2;
var resultado = (a * b) / c;
console.log(resultado); // Muestra 22.5
```

El operador / se utiliza en una expresión más compleja para calcular un resultado basado en valores de varias variables.

5. Redondeo de números decimales:

```
var dividendo = 7;
var divisor = 3;
var cociente = dividendo / divisor;
var redondeado = Math.round(cociente);
console.log("Resultado: " + cociente + " (redondeado: " + redondeado + ")");
```

El operador / se usa para dividir números, y luego utilizamos Math.round() para redondear el resultado a un número entero.

Módulo:

1. Resto de una división:

```
var dividendo = 10;
var divisor = 3;
var resto = dividendo % divisor;
console.log("El resto de " + dividendo + " dividido por " + divisor + " es " + resto);
```

En este ejemplo, el operador % se utiliza para calcular el resto de una división. En este caso, el resto es 1.

2. Detección de números pares e impares:

```
var numero = 7;
if (numero % 2 === 0) {
    console.log(numero + " es un número par.");
} else {
    console.log(numero + " es un número impar.");
}
```

El operador % se usa para determinar si un número es par o impar. Si el resto de la división por 2 es 0, el número es par; de lo contrario, es impar.

3. Ciclo repetido:

```
for (var i = 1; i <= 10; i++) {
    if (i % 2 === 0) {
        console.log(i + " es un número par.");
    } else {
        console.log(i + " es un número impar.");
    }
}
```

En este caso, el operador % se usa en un bucle for para iterar a través de los números del 1 al 10 y determinar si son pares o impares.

4. Paginación en una tabla:

```
var totalItems = 25;
var itemsPorPagina = 10;
var paginas = Math.ceil(totalItems / itemsPorPagina);
console.log("Se necesitan " + paginas + " páginas para mostrar " + totalItems + " elementos en lotes de " + itemsPorPagina + ".");
```

El operador % se utiliza para calcular el número de páginas necesarias para mostrar una cantidad total de elementos en lotes de un tamaño específico.

5. Generación de secuencias cíclicas:

```
var secuencia = [0, 1, 2, 3, 4, 5];
var indice = 7;
var elemento = secuencia[indice % secuencia.length];
console.log("El elemento en el índice " + indice + " es " + elemento);
```

El operador % se utiliza para crear secuencias cíclicas accediendo a elementos de un arreglo en bucle.

Precedencia de los Operadores:

La precedencia en JavaScript es una regla que determina el orden en que se evalúan y ejecutan las operaciones en una expresión. Cada operador tiene un nivel de precedencia, lo que significa que algunos operadores se ejecutan antes que otros. Esto es esencial para comprender cómo se evalúan las expresiones y cómo se toman decisiones en el código.

Operadores Básicos:

Comencemos con los operadores básicos y su precedencia:

Operadores de Asignación (=): Tienen la menor precedencia y se utilizan para asignar un valor a una variable. Por ejemplo: let x = 5;.

Operadores Aritméticos (+, -, , /, %): Estos operadores realizan operaciones matemáticas y se evalúan en el siguiente orden: multiplicación (), división (/), módulo (%), suma (+) y resta (-).

Operadores de Comparación (==, ===, !=, !==, <, >, <=, >=): Se utilizan para comparar dos valores y devuelven un valor booleano (verdadero o falso).

Operadores Lógicos (&&, ||): Estos operadores se utilizan para combinar expresiones lógicas y se evalúan en el siguiente orden: && (AND) antes que || (OR).

Paréntesis: Los paréntesis pueden utilizarse para anular la precedencia y forzar un orden específico de evaluación. Las expresiones dentro de paréntesis se evalúan primero.

Por ejemplo:

Supongamos que tenemos la siguiente expresión:

```
let resultado = 5 + 3 * 2;
```

La precedencia determina que primero se multiplicará 3 por 2 (6), y luego se sumará 5, lo que resultará en un resultado de 11. Si quisieras que la suma se hiciera antes, deberías usar paréntesis:

```
let resultado = (5 + 3) * 2; // Esto dará un resultado de 16.
```

Operadores de Incremento y Decremento:

Los operadores de incremento y decremento son herramientas poderosas en JavaScript que permiten aumentar o disminuir el valor de una variable en una unidad. Estos operadores son especialmente útiles en bucles, al mantener el control del índice o en situaciones en las que necesitas realizar un seguimiento de cambios en una variable.

Operador de Incremento (++)

El operador de incremento ++ aumenta el valor de una variable en uno. Puedes usarlo tanto en forma de post-incremento como de pre-incremento:

Post-incremento: variable++ aumenta el valor de variable después de utilizarlo en una expresión. Por ejemplo:

```
let x = 5;
let y = x++; // Aquí, y será igual a 5, y luego x se incrementa a 6.
```

Pre-incremento: ++variable aumenta el valor de variable antes de utilizarlo en una expresión. Por ejemplo:

```
let a = 3;
let b = ++a; // Aquí, b será igual a 4, y a ya se ha incrementado a 4.
```

Operador de Decremento (--)

El operador de decremento -- disminuye el valor de una variable en uno, y al igual que con el operador de incremento, puedes usarlo en forma de post-decremento o pre-decremento:

Post-decremento: variable-- disminuye el valor de variable después de utilizarlo en una expresión.

Pre-decremento: --variable disminuye el valor de variable antes de utilizarlo en una expresión.

Usos Comunes:

Los operadores de incremento y decremento se utilizan en diversas situaciones:

Bucles: Pueden ser útiles para controlar los índices en bucles for, while o do-while.

Contadores: Se utilizan para llevar un registro de eventos o elementos procesados.

Toggle de Estado: A menudo se emplean para cambiar el estado de una variable booleana entre verdadero y falso.

Precauciones:

Aunque los operadores de incremento y decremento son poderosos, es importante usarlos con precaución. La ubicación en la que los empleas (como pre-incremento o post-incremento) puede afectar el resultado. Además, el abuso de estos operadores puede llevar a código confuso y difícil de entender.

3.2 Operadores de comparación

Igualdad (==)

Este operador se utiliza para comparar si dos valores son iguales en términos de su valor, pero no necesariamente en términos de su tipo de dato. Por ejemplo:

```
5 == "5" // Verdadero, porque ambos valores son iguales en términos de valor
```

Desigualdad (!=)

Este operador se utiliza para verificar si dos valores no son iguales en términos de valor. Por ejemplo:

```
5 != 10 // Verdadero, porque 5 no es igual a 10
```

Igualdad Estricta (===)

A diferencia de ==, este operador verifica si dos valores son iguales tanto en términos de valor como en términos de tipo de dato. Por ejemplo:

```
5 === "5" // Falso, porque los tipos de dato son diferentes
```

Desigualdad Estricta (!==)

Este operador verifica si dos valores no son iguales en términos de valor o tipo de dato. Por ejemplo:

```
5 !== "5" // Verdadero, porque los tipos de dato son diferentes
```

Mayor que (>)

Se utiliza para verificar si un valor es mayor que otro. Por ejemplo:

```
10 > 5 // Verdadero, porque 10 es mayor que 5
```

Menor que (<)

Este operador verifica si un valor es menor que otro. Por ejemplo:

```
5 < 10 // Verdadero, porque 5 es menor que 10
```

Mayor o Igual que (>=)

Este operador verifica si un valor es mayor o igual que otro. Por ejemplo:

```
10 >= 10 // Verdadero, porque 10 es igual a 10
```

Menor o Igual que (<=)

Se utiliza para verificar si un valor es menor o igual que otro. Por ejemplo:

```
5 <= 10 // Verdadero, porque 5 es menor que 10
```

Estos operadores son esenciales para construir condiciones en tus programas. Puedes usarlos en estructuras de control como if, else if, y else para tomar decisiones basadas en comparaciones. Por ejemplo:

```
let edad = 20;

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

En este caso, el operador de comparación >= se utiliza para determinar si la edad es mayor o igual a 18, y con base en esa comparación, se muestra un mensaje específico en la consola.

3.3 Operadores lógicos

AND Lógico (&&):

Este operador se utiliza para combinar dos expresiones y devuelve true si ambas son verdaderas. Si al menos una de las expresiones es falsa, el resultado es false. Por ejemplo:

```
let tieneLicencia = true
let edad = 21
if (edad >= 18 && tieneLicencia) {
  console.log("Puede conducir");
} else {
  console.log("No puede conducir");
}
```

En este ejemplo, el operador && se utiliza para verificar si una persona tiene al menos 18 años y si posee una licencia de conducir. Ambas condiciones deben ser verdaderas para que pueda conducir.

OR Lógico (||)

Este operador se utiliza para combinar dos expresiones y devuelve true si al menos una de ellas es verdadera. Si ambas son falsas, el resultado es false. Por ejemplo:

```
esEstudiante = true
esEmpleado = false
if (esEstudiante || esEmpleado) {
  console.log("Tiene descuento");
} else {
  console.log("No tiene descuento");
}
```

En este caso, el operador || se utiliza para verificar si una persona es estudiante o empleado. Si cualquiera de las dos condiciones es verdadera, obtendrá un descuento.

NOT Lógico (!)

Este operador se utiliza para negar una expresión. Si una expresión es verdadera, ! la convierte en falsa y viceversa. Por ejemplo:

```
estaLloviendo = true
if (!estaLloviendo) {
  console.log("Puedes salir a jugar");
} else {
  console.log("Mejor quédate en casa");
}
```

El operador ! se utiliza para verificar si no está lloviendo, lo que significa que es un día adecuado para salir a jugar.

Estos operadores lógicos son fundamentales para construir condiciones más complejas en tus programas y para tomar decisiones basadas en múltiples condiciones. Puedes combinar operadores lógicos y operadores de comparación para crear condiciones sofisticadas y tomar decisiones precisas en tu código.

3.4. Operadores de asignación

1. Operador de Asignación (=): El operador de asignación básico se utiliza para asignar un valor a una variable. Por ejemplo:

```
let numero = 10;
```

En este caso, se asigna el valor 10 a la variable numero.

Operadores de Asignación con Operación (+=, -=)

Estos operadores realizan una operación aritmética y luego asignan el resultado a la variable. Por ejemplo:

```
let contador = 5;
contador += 3; // Esto es equivalente a contador = contador + 3;
```

El valor de contador se incrementa en 3 unidades.

Operadores de Asignación con Multiplicación y División (*=, /=)

Estos operadores multiplican o dividen el valor actual de la variable por otro valor y luego asignan el resultado a la variable. Por ejemplo:

```
let total = 50;
total *= 2; // Esto es equivalente a total = total * 2;
```

El valor de total se multiplica por 2.

Operadores de Asignación con Módulo (%=)

Este operador calcula el módulo del valor actual de la variable y otro valor y luego asigna el resultado a la variable. Por ejemplo:

```
let numero = 15;
numero %= 4; // Esto es equivalente a numero = numero % 4;
```

El valor de numero se establece en el residuo de la división de 15 por 4, que es 3.

Módulo 4: Control de Flujo (2 Horas)

4.1 Estructuras de control: if, else if, else

if:

La expresión if se utiliza para ejecutar un bloque de código si una condición es verdadera. Por ejemplo:

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

En este caso, si la variable edad es mayor o igual a 18, se ejecutará la instrucción dentro del bloque {}.

if...else

La expresión if...else te permite ejecutar un bloque de código si la condición es verdadera y otro bloque si es falsa. Por ejemplo:

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

else if:

La expresión else if se utiliza para evaluar múltiples condiciones secuencialmente. Por ejemplo:

```
if (puntaje >= 90) {
  console.log("Tienes una calificación A");
} else if (puntaje >= 80) {
  console.log("Tienes una calificación B");
} else if (puntaje >= 70) {
  console.log("Tienes una calificación C");
} else {
  console.log("Tienes una calificación D");
}
```

4.2 Operadores ternarios

Los operadores ternarios en JavaScript son una forma concisa de escribir expresiones condicionales que te permiten tomar decisiones y asignar valores a una variable en una sola línea de código. Estos operadores se llaman "ternarios" porque tienen tres operandos: una condición seguida de dos expresiones separadas por un símbolo de interrogación ? y un símbolo de dos puntos :. Aquí tienes una explicación detallada de cómo funcionan:

Estructura básica de un operador ternario:

```
condicion ? expresionSiVerdadero : expresionSiFalso
```

La condicion es una expresión que se evalúa como verdadera o falsa.

Si la condicion es verdadera, se ejecuta expresionSiVerdadero.

Si la condicion es falsa, se ejecuta expresionSiFalso.

Ejemplo de un operador ternario:

Supongamos que quieres asignar un mensaje a una variable saludo basado en si la hora actual es antes o después de las 12 PM:

```
const hora = new Date().getHours();
const saludo = hora < 12 ? "Buenos días" : "Buenas tardes";
```

En este ejemplo, el operador ternario evalúa si hora es menor que 12. Si es cierto, asigna el valor "Buenos días" a saludo; de lo contrario, asigna "Buenas tardes".

Ventajas de los operadores ternarios:

Concisión: Los operadores ternarios te permiten escribir decisiones en una sola línea de código, lo que puede hacer que tu código sea más legible y conciso.

Facilitan el enfoque en una variable: Los operadores ternarios son útiles cuando deseas asignar un valor a una variable basado en una condición y te permite enfocarte en una única variable en lugar de utilizar bloques if...else.

Limitaciones de los operadores ternarios:

Sencillez en la lógica: Los operadores ternarios son útiles para condiciones simples, pero cuando las condiciones son más complejas, los bloques if...else pueden ser más legibles.

Uso apropiado de los operadores ternarios:

Los operadores ternarios son ideales cuando deseas tomar decisiones simples y asignar valores en una sola línea. Sin embargo, para condiciones más complejas o flujos de control más extensos, es recomendable utilizar estructuras if...else más descriptivas y fáciles de leer.

4.3 Uso de switch

La instrucción switch en JavaScript es una estructura de control que se utiliza para tomar decisiones basadas en el valor de una expresión. Es especialmente útil cuando tienes que evaluar una expresión y comparar su valor con varios casos posibles. Aquí tienes una explicación detallada de cómo funciona la instrucción switch:

Estructura básica de la instrucción switch:

```
switch (expresion) {
  case valor1:
    // Código a ejecutar si expresion es igual a valor1
    break;
  case valor2:
    // Código a ejecutar si expresion es igual a valor2
    break;
  // Puedes tener más casos aquí
  default:
    // Código a ejecutar si ninguno de los casos anteriores coincide
}
```

expresion es el valor que se va a evaluar.

valor1, valor2, etc., son los valores que se compararán con la expresion.

Cada case representa un valor que se compara con la expresion.

El bloque de código correspondiente a un case se ejecuta si la expresion es igual a ese valor.

La palabra clave break se utiliza para salir del switch después de ejecutar el bloque de código correspondiente a un case.

El default es opcional y se ejecuta si la expresion no coincide con ninguno de los case anteriores.

Ejemplo de la instrucción switch:

Supongamos que deseas mostrar el día de la semana en función de un número (1 para lunes, 2 para martes, etc.):

```
let dia = 3;
let nombreDia;

switch (dia) {
  case 1:
    nombreDia = "Lunes";
    break;
  case 2:
    nombreDia = "Martes";
    break;
  case 3:
    nombreDia = "Miércoles";
    break;
  case 4:
    nombreDia = "Jueves";
    break;
  case 5:
    nombreDia = "Viernes";
    break;
  case 6:
    nombreDia = "Sábado";
    break;
  case 7:
    nombreDia = "Domingo";
    break;
  default:
    nombreDia = "Día no válido";
}

console.log("Hoy es " + nombreDia);
```

En este ejemplo, la variable dia se compara con diferentes casos y se asigna un valor a nombreDia según el valor de dia.

Ventajas de la instrucción switch:

Legibilidad: La instrucción switch es especialmente útil cuando tienes muchas condiciones para evaluar. Puede hacer que el código sea más legible que una serie de declaraciones if...else if.

Eficiencia: En algunas implementaciones de JavaScript, switch puede ser más eficiente que una serie de declaraciones if...else if para casos múltiples.

Limitaciones de la instrucción switch:

Comparaciones de igualdad: La instrucción switch realiza comparaciones de igualdad estrictas (con el operador ===), lo que significa que no permite comparaciones más complejas o evaluaciones de rangos.

Sintaxis rígida: La sintaxis de switch es rígida y no permite expresiones más complicadas en las comparaciones.

Uso apropiado de la instrucción switch:

La instrucción switch es una excelente opción cuando necesitas tomar decisiones basadas en el valor de una expresión y tienes una serie de casos claros y específicos que comparar. Sin embargo, para condiciones más complejas o casos no tan definidos, las declaraciones if...else if pueden ser más adecuadas.

4.4 Bucles: for, while, do-while

El bucle for en JavaScript es una estructura de control que te permite ejecutar un bloque de código repetidamente hasta que se cumpla una condición especificada. Es ampliamente utilizado para recorrer listas de elementos, realizar cálculos iterativos y ejecutar tareas repetitivas. Aquí tienes una explicación detallada de cómo funciona el bucle for:

Estructura básica de un bucle for:

```
for (inicialización; condición; actualización) {
  // Bloque de código a ejecutar en cada iteración
}
```

La inicialización se ejecuta una vez al principio del bucle y se utiliza para inicializar una variable de control.

La condición se evalúa antes de cada iteración. Si es verdadera, el bucle continúa; si es falsa, el bucle se detiene.

La actualización se ejecuta después de cada iteración y se utiliza para modificar la variable de control.

El bloque de código se ejecuta en cada iteración mientras la condición sea verdadera.

Ejemplo de un bucle for:

Supongamos que deseas imprimir los números del 1 al 5:

```
for (let i = 1; i <= 5; i++) {
  console.log(i);
}
```

En este ejemplo, el bucle comienza con i inicializada en 1. En cada iteración, se verifica si i es menor o igual a 5; si es cierto, se imprime i, y luego i se incrementa en 1 con i++. El bucle se ejecuta hasta que i sea mayor que 5.

Ventajas del bucle for:

Control preciso: El bucle for proporciona un control preciso sobre el número de iteraciones y el estado de la variable de control, lo que lo hace ideal para recorrer listas y realizar cálculos iterativos.

Legibilidad: La estructura del bucle for es muy clara y fácil de entender, lo que lo hace útil en situaciones en las que necesitas una iteración controlada.

Limitaciones del bucle for:

Requisitos de inicialización, condición y actualización: Debes proporcionar una inicialización, una condición y una actualización en la declaración del bucle, lo que puede no ser necesario en algunas situaciones más simples.

Uso apropiado del bucle for:

El bucle for es ideal cuando sabes cuántas veces deseas que se ejecute un bloque de código o cuando necesitas recorrer una colección de elementos, como un array. También es útil cuando deseas realizar cálculos iterativos.

Estructura básica de la instrucción while:

La instrucción while en JavaScript es una estructura de control que te permite ejecutar un bloque de código repetidamente mientras se cumpla una condición específica. A diferencia del bucle for, el bucle while no tiene un número fijo de iteraciones; en cambio, depende de que la condición se evalúe como verdadera o falsa. Aquí tienes una explicación detallada de cómo funciona la instrucción while:

```
while (condición) {
  // Bloque de código a ejecutar mientras la condición sea verdadera
}
```

La condición es una expresión que se evalúa antes de cada iteración. Si la condición es verdadera, el bucle continúa; si es falsa, el bucle se detiene.

El bloque de código se ejecuta en cada iteración mientras la condición sea verdadera.

Ejemplo de la instrucción while:

Supongamos que deseas imprimir números del 1 al 5 usando un bucle while:

```
let i = 1;

while (i <= 5) {
  console.log(i);
  i++;
}
```

En este ejemplo, el bucle while comienza con i inicializada en 1. En cada iteración, se verifica si i es menor o igual a 5; si es cierto, se imprime i, y luego i se incrementa en 1 con i++. El bucle se ejecuta hasta que i sea mayor que 5.

Ventajas de la instrucción while:

Flexibilidad: El bucle while es flexible y puede utilizarse en situaciones donde no conoces de antemano cuántas iteraciones se necesitan. El bucle continúa hasta que la condición sea falsa.

Legibilidad: La estructura del bucle while es clara y fácil de entender, lo que lo hace útil en situaciones en las que necesitas una iteración controlada por una condición específica.

Limitaciones de la instrucción while:

Potencial de bucles infinitos: Si no se actualiza correctamente la variable de control o si la condición nunca se vuelve falsa, el bucle puede ejecutarse infinitamente, lo que se conoce como un bucle infinito.

Uso apropiado de la instrucción while:

El bucle while es útil cuando deseas repetir un bloque de código mientras se cumpla una condición específica. Puede ser especialmente útil en situaciones en las que no conoces el número exacto de iteraciones requeridas de antemano.

La instrucción do...while

Es otra estructura de control que te permite ejecutar un bloque de código repetidamente mientras se cumple una condición. A diferencia del bucle while, la instrucción do...while garantiza que el bloque de código se ejecute al menos una vez, independientemente de si la condición es verdadera o falsa en la primera evaluación. Aquí tienes una explicación detallada de cómo funciona la instrucción do...while:

Estructura básica de la instrucción do...while:

```
do {
  // Bloque de código a ejecutar al menos una vez
} while (condición);
```

El bloque de código se ejecuta primero, antes de que se evalúe la condición.

La condición es una expresión que se evalúa después de que se ha ejecutado el bloque de código. Si la condición es verdadera, el bucle continúa; si es falsa, el bucle se detiene.

Ejemplo de la instrucción do...while:

Supongamos que deseas imprimir números del 1 al 5 usando un bucle do...while:

```
let i = 1;

do {
  console.log(i);
  i++;
} while (i <= 5);
```

En este ejemplo, el bucle do...while comienza con i inicializada en 1. El bloque de código se ejecuta al menos una vez, independientemente de la condición. Luego, se verifica si i es menor o igual a 5; si es cierto, el bucle continúa. El bucle se ejecuta hasta que i sea mayor que 5.

Ventajas de la instrucción do...while:

Ejecución garantizada: La instrucción do...while garantiza que el bloque de código se ejecute al menos una vez, lo que puede ser útil en situaciones en las que se necesita realizar una acción inicial antes de verificar una condición.

Flexibilidad: Al igual que con while, do...while es flexible y puede utilizarse en situaciones donde no conoces de antemano cuántas iteraciones se necesitan.

Limitaciones de la instrucción do...while:

Sintaxis adicional: La estructura de la instrucción do...while requiere una sintaxis adicional en comparación con while, lo que puede hacer que el código sea ligeramente más largo.

Uso apropiado de la instrucción do...while:

La instrucción do...while es útil cuando deseas repetir un bloque de código mientras se cumple una condición, pero necesitas que el bloque se ejecute al

menos una vez, independientemente de si la condición es verdadera o falsa en la primera evaluación.

4.5 Rompiendo y continuando bucles

En JavaScript, a veces necesitas más control sobre bucles, especialmente para evitar que un bucle se ejecute indefinidamente o para omitir una iteración en particular. Puedes lograr esto utilizando las sentencias break y continue. Aquí tienes una explicación detallada de cómo funcionan y cuándo utilizarlas:

La sentencia break:

La sentencia break se utiliza para salir inmediatamente de un bucle cuando se cumple una cierta condición. Esto detiene la ejecución del bucle por completo. La sintaxis es la siguiente:

```
for (let i = 1; i <= 10; i++) {
  if (i === 5) {
    break; // Sal del bucle cuando i sea igual a 5
  }
  console.log(i);
}
```

En este ejemplo, cuando i es igual a 5, la sentencia break se ejecuta y el bucle se detiene, evitando que se impriman números mayores a 5.

Cuándo usar break:

- Para salir de un bucle cuando se cumple una condición específica.
- Para evitar bucles infinitos o innecesarios.

La sentencia continue:

La sentencia continue se utiliza para omitir la iteración actual y continuar con la siguiente iteración del bucle. La sintaxis es la siguiente:

```
for (let i = 1; i <= 5; i++) {
  if (i === 3) {
    continue; // Omite la iteración cuando i sea igual a 3
  }
  console.log(i);
}
```

En este ejemplo, cuando i es igual a 3, la sentencia continue se ejecuta y el bucle salta la iteración actual, evitando que se imprima el número 3.

Cuándo usar continue:

- Para saltar una iteración del bucle cuando se cumple una condición específica.
- Para evitar la ejecución de ciertas partes del código en una iteración específica.

Importancia de break y continue:

Estas sentencias son fundamentales en la programación porque te brindan un mayor control sobre la ejecución de bucles. Puedes utilizarlas para manejar casos especiales y controlar el flujo de tu programa de manera más precisa.

Módulo 5: Funciones en JavaScript (3 Horas)

5.1. Declaración y llamada de funciones

En JavaScript, las funciones son una parte fundamental de la programación. Te permiten agrupar un conjunto de instrucciones que pueden ser ejecutadas en cualquier momento. Las funciones hacen que el código sea más organizado, reutilizable y fácil de mantener. Aquí te mostramos cómo declarar y llamar funciones en JavaScript:

Declaración de Funciones:

Para declarar una función en JavaScript, puedes usar la siguiente sintaxis:

```
function nombreDeLaFuncion(parametro1, parametro2, ...) {
  // Cuerpo de la función: instrucciones a ejecutar
}
```

nombreDeLaFuncion: Este es el nombre que eliges para tu función. Debe seguir las reglas de nomenclatura de JavaScript y ser descriptivo de lo que hace la función.

parametro1, parametro2, ...: Estos son los parámetros de la función. Son valores que la función espera recibir cuando se llama. Puedes tener cero o más parámetros.

Cuerpo de la función: Aquí colocas el código que se ejecutará cuando la función sea llamada. Puedes realizar cálculos, interactuar con variables y realizar cualquier tarea necesaria.

Ejemplo de declaración de función:

```
function saludar(nombre) {
  console.log("Hola, " + nombre + "!");
}
```

Llamada de Funciones:

Para ejecutar una función, simplemente la llamas por su nombre y, si tiene parámetros, proporcionas los valores necesarios. La llamada a una función se realiza de la siguiente manera:

nombreDeLaFuncion(argumento1, argumento2, ...);

Ejemplo de llamada de función:

```
saludar("Juan");
```

Cuando llamas a la función saludar("Juan"), se ejecuta el código dentro de la función saludar, y se muestra "Hola, Juan!" en la consola.

Valor de Retorno:

Las funciones en JavaScript pueden devolver un valor usando la palabra clave return. Esto permite que una función calcule un resultado y lo pase de vuelta a quien la llamó. Por ejemplo:

```
function sumar(a, b) {
  return a + b;
}
var resultado = sumar(3, 5);
console.log(resultado); // Esto mostrará 8 en la consola.
```

En este ejemplo, la función sumar toma dos argumentos, los suma y devuelve el resultado, que luego se almacena en la variable resultado.

5.2. Parámetros y argumentos

Los parámetros y argumentos son conceptos fundamentales en JavaScript y en la mayoría de los lenguajes de programación. Te ayudan a pasar información a una función y trabajar con datos de manera dinámica. Vamos a explorar en profundidad estos conceptos.

Parámetros de una Función:

Los parámetros de una función son variables locales que actúan como marcadores de posición para los valores que se pasan cuando se llama a la función. Estos parámetros se definen en la declaración de la función y son utilizados en el cuerpo de la función para realizar operaciones.

Aquí tienes un ejemplo de declaración de una función con parámetros:

```
function suma(a, b) {
  return a + b;
}
```

En este caso, la función suma tiene dos parámetros: a y b. Estos parámetros se utilizan para sumar dos valores y devolver el resultado.

Argumentos de una Función:

Los argumentos de una función son los valores reales que se pasan cuando se llama a la función. Los argumentos deben coincidir en número y orden con los parámetros de la función.

Ejemplo de llamada a la función con argumentos:

```
var resultado = suma(5, 3);
```

En este caso, 5 y 3 son los argumentos que se pasan a la función suma. Cuando se llama a la función, a se toma como 5 y b se toma como 3. La función realiza la suma y devuelve 8, que se almacena en la variable resultado.

Número de Argumentos:

En JavaScript, las funciones no tienen restricciones en cuanto al número de argumentos que pueden recibir. Puedes definir una función con parámetros, pero también puedes llamarla con menos argumentos de los que se esperan. Aquí hay un ejemplo:

```
function saludar(nombre, saludo) {
  return saludo + " " + nombre;
}
var mensaje = saludar("Ana");
console.log(mensaje); // Esto mostrará "undefined Ana" en la consola.
```

En este caso, la función saludar espera dos argumentos (nombre y saludo), pero solo se proporciona uno. Como resultado, saludo se interpreta como undefined.

Valores por Defecto:

A partir de ECMAScript 6 (ES6), puedes asignar valores predeterminados a los parámetros de una función. Esto significa que si no se proporciona un valor al llamar a la función, se utilizará el valor predeterminado.

```
function saludar(nombre, saludo = "Hola") {
  return saludo + " " + nombre;
}
var mensaje = saludar("Ana");
console.log(mensaje); // Esto mostrará "Hola Ana" en la consola.
```

En este ejemplo, si no se proporciona un segundo argumento al llamar a la función saludar, se utilizará el valor predeterminado "Hola".

Rest Parameters y Spread Operator:

ES6 también introdujo características avanzadas para trabajar con parámetros en funciones. El operador ... se utiliza para crear "rest parameters", que permiten manejar un número variable de argumentos como un array. Además, el "spread operator" (...) se usa para descomponer un array en argumentos individuales. Estas características son especialmente útiles cuando se trabaja con un número desconocido de argumentos.

Ejemplo de Rest Parameters:

```
function suma(...numeros) {
  return numeros.reduce((total, num) => total + num, 0);
}
var resultado = suma(1, 2, 3, 4, 5);
console.log(resultado); // Esto mostrará 15 en la consola.
```

Ejemplo de Spread Operator:

```
var numeros = [1, 2, 3, 4, 5];
var resultado = suma(...numeros);
console.log(resultado); // Esto mostrará 15 en la consola.
```

Los rest parameters y el spread operator son herramientas poderosas para trabajar con funciones que requieren flexibilidad en la cantidad de argumentos que pueden manejar.

5.3. Retorno de valores

El retorno de valores en funciones es un concepto fundamental en JavaScript y en la programación en general. Permite que una función procese datos y devuelva un resultado que puede ser utilizado en otras partes del programa. A continuación, se proporciona un resumen extenso de cómo funciona el retorno de valores en JavaScript y su importancia.

Devolviendo un Valor desde una Función:

Cuando defines una función en JavaScript, puedes especificar que esta función devuelva un valor utilizando la palabra clave return. La sintaxis general de una función con retorno de valor es la siguiente:

```
function nombreDeLaFuncion(parametro1, parametro2, ...) {
  // Código de la función
  return valorDeRetorno;
}
```

nombreDeLaFuncion: El nombre de la función que defines.

parametro1, parametro2, ...: Los parámetros que acepta la función (opcional).

valorDeRetorno: El valor que la función devolverá al ser llamada.

Importancia del Retorno de Valores:

El retorno de valores en funciones es esencial por varias razones:

Reutilización de Código: Permite encapsular una lógica específica en una función y reutilizar esa función en diferentes partes de tu programa sin necesidad de volver a escribir el mismo código.

Modularidad: Facilita la división de un programa en módulos o funciones independientes que realizan tareas específicas. Esto hace que el código sea más legible y mantenible.

Comunicación de Resultados: Las funciones pueden calcular y devolver resultados que luego se utilizan en otras partes del programa. Esto es especialmente útil cuando deseas realizar operaciones complejas y utilizar el resultado en varios lugares.

Abstracción: Puedes utilizar funciones para abstractos detalles de implementación y centrarte en lo que hace la función en lugar de cómo lo hace.

Ejemplo de Retorno de Valores:

Supongamos que tienes una función que calcula el área de un círculo:

```
function calcularAreaDelCirculo(radio) {
  var area = Math.PI * radio * radio;
  return area;
}
```

Puedes llamar esta función con un valor de radio y obtener el área del círculo como resultado:

```
var radio = 5;
var area = calcularAreaDelCirculo(radio);
console.log("El área del círculo es: " + area);
```

La función calcularAreaDelCirculo toma el radio como argumento, realiza el cálculo y devuelve el área. Luego, puedes usar ese valor en otras partes de tu programa.

Valores de Retorno Múltiples:

En JavaScript, una función puede devolver un solo valor, pero ese valor puede ser un objeto, un array o cualquier otro tipo de dato complejo. Esto permite que una función devuelva múltiples valores como parte de una estructura de datos.

Por ejemplo, una función podría devolver un objeto con múltiples propiedades:

```
function obtenerInformacionDelEstudiante(nombre, edad) {
  var informacion = {
    nombre: nombre,
    edad: edad,
    grado: "Décimo",
    promedio: 8.5
  };
  return informacion;
}
```

Manejo de Valores de Retorno:

Cuando una función devuelve un valor, puedes almacenarlo en una variable, utilizarlo en expresiones, pasarlo como argumento a otras funciones o realizar operaciones adicionales con él.

```
var resultado = obtenerInformacionDelEstudiante("Ana", 16);
console.log(resultado.nombre); // Muestra "Ana"
console.log(resultado.edad); // Muestra 16
```

Valor de Retorno Opcional:

Es importante destacar que no todas las funciones en JavaScript necesitan devolver un valor. Si no se utiliza la palabra clave return, la función devolverá undefined por defecto. Esto es especialmente común en funciones que se utilizan para realizar acciones y no para calcular resultados.

```
function saludar(nombre) {
  console.log("¡Hola, " + nombre + "!");
}
var saludo = saludar("Juan");
console.log(saludo); // Muestra "¡Hola, Juan!" y luego "undefined"
```

5.4. Ámbito de funciones

El ámbito de funciones se refiere al contexto en el que una variable es visible y accesible en tu programa. En JavaScript, existen dos tipos principales de ámbitos: el ámbito global y el ámbito de función.

Ámbito Global:

Las variables declaradas fuera de cualquier función se consideran globales y pueden ser accedidas desde cualquier lugar del programa. Estas variables son visibles tanto dentro como fuera de funciones.

```
var variableGlobal = 10;
function miFuncion() {
  console.log(variableGlobal); // Puedes acceder a la variable global
}
```

Ámbito de Función:

Las variables declaradas dentro de una función son locales a esa función, lo que significa que solo son visibles y accesibles desde dentro de esa función.

```
function miFuncion() {
  var variableLocal = 5;
  console.log(variableLocal); // Puedes acceder a la variable local
}
console.log(variableLocal); // Esto generará un error, ya que variableLocal no está definida en este ámbito.
```

Ámbito Global vs. Ámbito de Función:

El ámbito de función se utiliza para crear variables que solo tienen sentido y valor dentro de una función en particular. Esto ayuda a prevenir conflictos de nombres y a mantener el código organizado.

Por otro lado, las variables globales son visibles desde cualquier parte del programa, pero esto puede llevar a problemas si no se gestionan adecuadamente, ya que pueden ser modificadas desde cualquier función, lo que puede llevar a errores difíciles de depurar.

Variable Hoisting:

Un concepto importante relacionado con el ámbito de funciones es el "hoisting" (elevación). En JavaScript, todas las declaraciones de variables (tanto globales como locales) se "elevan" al comienzo del ámbito en el que están declaradas. Esto significa que, aunque declares una variable más adelante en una función, JavaScript la tratará como si la hubieras declarado al comienzo de esa función.

```
function miFuncion() {
   console.log(variableHoisted); // Esto no generará un error, pero mostrará "undefined".
  var variableHoisted = 10;
}
```

El "hoisting" puede conducir a comportamientos inesperados si no se comprende bien, por lo que es importante declarar las variables al principio de la función para evitar confusiones.

Aunque las variables globales son accesibles en cualquier lugar, es recomendable minimizar su uso y, en su lugar, utilizar variables locales en las funciones siempre que sea posible. Las variables globales pueden ser modificadas desde múltiples lugares del código, lo que puede hacer que sea difícil rastrear errores.

5.5. Funciones anónimas y expresiones de función

Las funciones anónimas y las expresiones de función son conceptos fundamentales en JavaScript que permiten una flexibilidad considerable al escribir código. Aquí se explora en profundidad su significado y uso:

Funciones Anónimas:

Una función anónima es una función que no tiene un nombre identificativo asociado. Se declaran sin un identificador y a menudo se utilizan en contextos en los que no se necesita un nombre específico. Las funciones anónimas son especialmente útiles cuando deseas crear funciones ad hoc o pasárselas como argumentos a otras funciones. Aquí hay un ejemplo de una función anónima:

```
var suma = function(a, b) {
  return a + b;
};
```

En este ejemplo, hemos declarado una función anónima que toma dos parámetros a y b y devuelve la suma de ambos. La función se asigna a la variable suma, lo que permite llamarla posteriormente como suma(3, 4).

Expresiones de Función:

Las expresiones de función son una forma de definir funciones en JavaScript. A diferencia de las declaraciones de función, que son izadas (elevadas) al principio del ámbito en el que están definidas, las expresiones de función son tratadas como cualquier otra expresión y solo están disponibles después de su definición. Aquí tienes un ejemplo de una expresión de función:

```
var multiplicar = function(a, b) {
  return a * b;
};
```

En este caso, hemos asignado una función anónima a la variable multiplicar. La función se crea y se asigna a la variable en el momento en que se ejecuta la expresión de función.

Beneficios de las Funciones Anónimas y Expresiones de Función:

Funciones de Primera Clase: En JavaScript, las funciones son ciudadanos de primera clase, lo que significa que pueden asignarse a variables, pasarse como argumentos a otras funciones y devolverse como valores de otras funciones. Las funciones anónimas y las expresiones de función permiten aprovechar esta característica poderosa.

Manejo de Callbacks: Son útiles al trabajar con callbacks, que son funciones que se pasan como argumentos a otras funciones. Por ejemplo, al usar funciones como map(), filter(), y forEach() en matrices.

Encapsulación: Puedes utilizar funciones anónimas para encapsular lógica y evitar la contaminación del ámbito global. Esto es especialmente útil en aplicaciones más grandes.

Ejemplos de Uso:

1. Funciones como Argumentos:

```
var numeros = [1, 2, 3, 4, 5];
var doble = numeros.map(function(numero) {
  return numero * 2;
});
```

En este ejemplo, hemos pasado una función anónima como argumento a la función map(), que se ejecutará para cada elemento de la matriz.

2. Autocall o Funciones Autoejecutables:

```
(function() {
  var mensaje = "Hola desde una función autoejecutable";
  console.log(mensaje);
})();
```

Esto crea una función anónima y la ejecuta inmediatamente sin necesidad de llamarla por separado. Es útil para encapsular código y variables dentro de un ámbito local.

Conclusión:

Las funciones anónimas y las expresiones de función son herramientas poderosas en JavaScript que brindan flexibilidad y modularidad al código. Son ampliamente utilizadas en programación moderna, especialmente en situaciones donde se requieren funciones como argumentos o cuando se necesita encapsular lógica en un ámbito local. Comprender cómo utilizar estas construcciones de funciones es esencial para sacar el máximo provecho de JavaScript y escribir código más limpio y eficiente.

Módulo 6: Objetos en JavaScript (3 Horas)

6.1. Introducción a objetos

¿Qué es un Objeto en JavaScript?

Un objeto en JavaScript es una entidad que agrupa datos (propiedades) y funciones (métodos) relacionados. Las propiedades son pares clave-valor que representan datos o valores, y los métodos son funciones que realizan acciones o cálculos. Los objetos permiten organizar y estructurar el código de manera eficiente y representar conceptos del mundo real en el código.

Creación de Objetos:

Existen varias formas de crear objetos en JavaScript. La más común es utilizando la notación de objeto literal:

```
var persona = {
  nombre: "Juan",
  edad: 30,
  saludar: function() {
    console.log("¡Hola, soy " + this.nombre + "!");
  }
};
```

En este ejemplo, hemos creado un objeto persona con tres propiedades (nombre, edad) y un método (saludar). Las propiedades son pares clave-valor, y el método es una función dentro del objeto.

Acceso a Propiedades y Métodos:

Puedes acceder a las propiedades y métodos de un objeto utilizando la notación de punto:

```
console.log(persona.nombre); // Acceder a una propiedad
persona.saludar(); // Llamar a un método
```

Agregar Propiedades y Métodos:

Puedes agregar propiedades y métodos a un objeto existente en cualquier momento:

```
persona.profesion = "Desarrollador";
persona.cumplirAnios = function() {
  this.edad++;
};
```

Eliminar Propiedades y Métodos:

También puedes eliminar propiedades y métodos de un objeto:

```
delete persona.profesion; // Eliminar una propiedad
delete persona.cumplirAnios; // Eliminar un método
```

Uso de "this" en Métodos:

Dentro de los métodos de un objeto, puedes usar la palabra clave this para referirte al propio objeto. Esto es útil para acceder a las propiedades del objeto desde sus propios métodos:

```
var coche = {
  marca: "Toyota",
  modelo: "Corolla",
  describir: function() {
      console.log("Este es un coche " + this.marca + " modelo " +
this.modelo);
  }
};
coche.describir(); // Muestra "Este es un coche Toyota modelo Corolla"
```

6.2. Propiedades y métodos de objetos

Las propiedades y métodos son componentes esenciales de los objetos en JavaScript. Las propiedades representan datos, mientras que los métodos son funciones que realizan acciones.

Propiedades de Objetos:

Las propiedades son variables que almacenan datos en un objeto. Cada propiedad tiene un nombre (clave) y un valor asociado. Las propiedades pueden ser de cualquier tipo de dato, como números, cadenas, objetos o incluso funciones. Aquí hay un ejemplo de un objeto con propiedades:

```
var persona = {
  nombre: "Juan",
```

```
  edad: 30,
  ciudad: "Madrid"
};
```

En este caso, el objeto persona tiene tres propiedades: nombre, edad y ciudad.

Acceso a Propiedades:

Puedes acceder a las propiedades de un objeto utilizando la notación de punto o la notación de corchetes:

```
console.log(persona.nombre); // Acceso utilizando la notación de punto
console.log(persona["edad"]); // Acceso utilizando la notación de corchetes
```

La notación de corchetes es especialmente útil cuando el nombre de la propiedad está contenido en una variable o cuando la propiedad tiene un nombre que no es una identificación válida.

```
var propiedad = "ciudad";
console.log(persona[propiedad]); // Acceso utilizando la notación de corchetes con una variable
```

Métodos de Objetos:

Los métodos son funciones que se almacenan como propiedades en un objeto. Estas funciones pueden realizar acciones específicas relacionadas con el objeto. Por ejemplo, un objeto coche podría tener un método arrancar que inicie el motor. Aquí hay un ejemplo de un objeto con un método:

```
var coche = {
  marca: "Toyota",
  modelo: "Corolla",
  arrancar: function() {
    console.log("El motor del " + this.marca + " " + this.modelo + " ha arrancado.");
  }
};
```

En este caso, el objeto coche tiene una propiedad llamada arrancar, que es una función (método). Cuando se llama a coche.arrancar(), se ejecuta la función y se muestra un mensaje.

El Uso de "this" en Métodos:

Dentro de un método de un objeto, puedes utilizar la palabra clave this para referirte al objeto mismo. Esto es útil para acceder a las propiedades del objeto desde sus métodos. Por ejemplo:

```
var persona = {
  nombre: "Ana",
  edad: 25,
  presentarse: function() {
     console.log("Hola, soy " + this.nombre + " y tengo " + this.edad +
" años.");
  }
};
persona.presentarse(); // Muestra "Hola, soy Ana y tengo 25 años."
```

Propiedades y Métodos Predefinidos:

En JavaScript, muchos objetos predefinidos tienen propiedades y métodos incorporados. Por ejemplo, los objetos Math y Date tienen propiedades y métodos que permiten realizar operaciones matemáticas y trabajar con fechas y horas. Puedes acceder a estas propiedades y métodos de la siguiente manera:

```
var valor = Math.PI; // Acceder a una propiedad de Math
var fecha = new Date(); // Crear un objeto Date
var dia = fecha.getDay(); // Llamar a un método de Date
```

Iteración de Propiedades:

Puedes iterar a través de las propiedades de un objeto utilizando un bucle for...in. Esto es especialmente útil cuando no conoces de antemano las propiedades de un objeto.

```
for (var propiedad in persona) {
  console.log(propiedad + ": " + persona[propiedad]);
}
```

6.3. Creación de objetos

La creación de objetos es un concepto fundamental en JavaScript que te permite representar datos y comportamientos relacionados en un solo

paquete. Hay varias formas de crear objetos en JavaScript, y aquí te presento algunas de las más comunes junto con ejemplos claros.

Notación de Objeto Literal:

La forma más sencilla de crear un objeto es utilizando la notación de objeto literal. En esta notación, defines las propiedades y métodos del objeto dentro de llaves {}.

Ejemplo:

```
var persona = {
  nombre: "Juan",
  edad: 30,
  saludar: function() {
    console.log("¡Hola, soy " + this.nombre + "!");
  }
};
```

En este ejemplo, hemos creado un objeto persona con tres propiedades: nombre, edad y un método saludar. Puedes acceder a las propiedades y métodos de este objeto utilizando la notación de punto, por ejemplo, persona.nombre o persona.saludar().

Constructor de Objetos:

Puedes utilizar funciones constructoras para crear objetos. Una función constructora es una función que actúa como un "molde" para crear nuevos objetos. Usando la palabra clave new, puedes crear múltiplas instancias de un objeto con la misma estructura.

Ejemplo:

```
function Persona(nombre, edad) {
  this.nombre = nombre;
  this.edad = edad;
  this.saludar = function() {
    console.log("¡Hola, soy " + this.nombre + "!");
  };
}

var juan = new Persona("Juan", 30);
var ana = new Persona("Ana", 25);
```

```
juan.saludar(); // Muestra "¡Hola, soy Juan!"
ana.saludar();  // Muestra "¡Hola, soy Ana!"
```

En este ejemplo, hemos definido una función constructora Persona que toma dos parámetros (nombre y edad) y crea objetos Persona con esas propiedades y un método saludar.

Object.create:

Object.create es un método que te permite crear un objeto nuevo con un prototipo específico. Esto es útil cuando deseas heredar propiedades y métodos de otro objeto.

Ejemplo:

```
var personaPrototype = {
  saludar: function() {
    console.log("¡Hola, soy " + this.nombre + "!");
  }
};

var juan = Object.create(personaPrototype);
juan.nombre = "Juan";
juan.edad = 30;
juan.saludar(); // Muestra "¡Hola, soy Juan!"
```

En este ejemplo, hemos creado un objeto juan utilizando Object.create y le hemos asignado un prototipo personaPrototype que define el método saludar.

Clases (ES6+):

Con la introducción de clases en ECMAScript 6 (ES6), puedes crear objetos de una manera más orientada a objetos. Las clases son una forma más estructurada de definir objetos y sus propiedades.

Ejemplo:

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

```
  saludar() {
    console.log("¡Hola, soy " + this.nombre + "!");
  }
}
var juan = new Persona("Juan", 30);
juan.saludar(); // Muestra "¡Hola, soy Juan!"
```

En este ejemplo, hemos definido una clase Persona con un constructor y un método saludar que se utilizan para crear objetos Persona.

6.4. Prototipos y herencia

El concepto de prototipos y herencia es un aspecto fundamental de la programación en JavaScript. En esta guía detallada, exploraremos en profundidad estos temas para que los principiantes comprendan cómo funcionan en JavaScript.

Entendiendo los Prototipos:

En JavaScript, cada objeto tiene un "prototipo" asociado que actúa como un modelo para ese objeto. Cuando intentas acceder a una propiedad o método en un objeto, JavaScript busca primero en el objeto mismo y, si no lo encuentra, sigue buscando en su prototipo.

Ejemplo 1 - Prototipo por defecto:

```
var persona = { nombre: "Juan" };
console.log(persona.nombre); // Muestra "Juan"
console.log(persona.edad);   // Muestra "undefined" (no se encuentra en el prototipo)
```

Prototipos y Herencia:

La herencia en JavaScript se basa en el sistema de prototipos. Cuando un objeto necesita acceder a una propiedad o método que no se encuentra en sí mismo, JavaScript lo buscará en su prototipo, y este proceso continúa hasta llegar al prototipo por defecto, que es el objeto Object.

Ejemplo 2 - Herencia:

```
var persona = { nombre: "Juan" };
var empleado = { salario: 50000 };
```

```
empleado.__proto__ = persona; // Establece el prototipo del objeto "empleado" como "persona"
console.log(empleado.nombre); // Muestra "Juan" (heredado de "persona")
```

En este ejemplo, hemos creado un objeto persona y un objeto empleado. Hemos establecido el prototipo de empleado como persona. Como resultado, empleado hereda la propiedad nombre de persona.

Herencia y el Constructor:

Puedes definir una función constructora que actúa como una plantilla para crear objetos y establecer su prototipo. Luego, puedes crear objetos a partir de esa función constructora utilizando la palabra clave new.

Ejemplo 3 - Función Constructora y Herencia:

```
function Persona(nombre) {
  this.nombre = nombre;
}
var juan = new Persona("Juan");
console.log(juan.nombre); // Muestra "Juan"
function Empleado(nombre, salario) {
  Persona.call(this, nombre);
  this.salario = salario;
}
Empleado.prototype = Object.create(Persona.prototype);
Empleado.prototype.constructor = Empleado;
var empleado1 = new Empleado("Carlos", 60000);
console.log(empleado1.nombre);   // Muestra "Carlos"
console.log(empleado1.salario);  // Muestra 60000
```

En este ejemplo, hemos definido una función constructora Persona y una función constructora Empleado. Hemos establecido el prototipo de Empleado como Persona, lo que permite que los objetos Empleado hereden propiedades y métodos de Persona.

Herencia y las Clases (ES6+):

Con la introducción de clases en ES6, puedes definir objetos y su herencia de una manera más estructurada y legible.

Ejemplo 4 - Clases y Herencia (ES6+):

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

class Empleado extends Persona {
  constructor(nombre, salario) {
    super(nombre);
    this.salario = salario;
  }
}

const empleado2 = new Empleado("Ana", 70000);
console.log(empleado2.nombre);   // Muestra "Ana"
console.log(empleado2.salario);  // Muestra 70000
```

Los prototipos y la herencia son conceptos esenciales en JavaScript que permiten la reutilización de código y la creación de relaciones entre objetos. Comprender cómo funcionan los prototipos y cómo establecer relaciones de herencia es crucial para construir aplicaciones más complejas y eficientes en JavaScript. A medida que avances en tu aprendizaje, te encontrarás utilizando estos conceptos con frecuencia para diseñar y desarrollar aplicaciones web.

Módulo 7: Manipulación del DOM (3 Horas)

7.1. Introducción al DOM

El Document Object Model (DOM) es un concepto fundamental en el desarrollo web con JavaScript. Es la representación en forma de árbol de la estructura de una página web y proporciona una forma de interactuar y manipular elementos HTML y sus propiedades. En este tutorial, te introduciré en el mundo del DOM de una manera amigable para principiantes.

¿Qué es el DOM?

El DOM es un modelo de programación que representa la estructura de un documento HTML como un árbol jerárquico de objetos. Cada elemento HTML, como encabezados, párrafos, imágenes, formularios, etc., se convierte en un objeto dentro de este árbol. El DOM permite a los programadores acceder y manipular estos objetos para interactuar con la página web.

Estructura del DOM:

El DOM comienza con un nodo llamado "documento" que representa todo el documento HTML. Desde ese nodo, se ramifican los nodos para representar la estructura del documento. Aquí tienes un ejemplo sencillo de la estructura del DOM:

- Document (representa la página web)
    - html
    - head
        - title
    - body
        - h1
        - p
        - img
        - form
            - input
            - button

Acceso a Elementos en el DOM:

Puedes acceder a elementos del DOM utilizando JavaScript. Aquí hay algunas de las formas más comunes:

getElementById:

Esta función te permite seleccionar un elemento por su atributo id.

```
var elemento = document.getElementById("miElemento");
```

getElementsByClassName:

Esta función selecciona todos los elementos que tienen una clase específica.

```
var elementos = document.getElementsByClassName("miClase");
```

getElementsByTagName:

Selecciona todos los elementos que tienen una etiqueta HTML específica.

```
var elementos = document.getElementsByTagName("p");
```

querySelector:

Utiliza selectores CSS para seleccionar un elemento.

```
var elemento = document.querySelector("#miElemento");
```

7.2 Manipulación del DOM:

Una vez que has seleccionado un elemento, puedes realizar varias operaciones con él. Aquí hay algunos ejemplos:

Cambiar el contenido de un elemento:

```
var elemento = document.getElementById("miElemento");
elemento.innerHTML = "Nuevo contenido";
```

Cambiar el atributo de un elemento:

```
var imagen = document.getElementById("miImagen");
imagen.src = "nueva-imagen.jpg";
```

Añadir o eliminar clases:

```
var elemento = document.getElementById("miElemento");
elemento.classList.add("nueva-clase");
elemento.classList.remove("clase-antigua");
```

Crear nuevos elementos:

```
var nuevoElemento = document.createElement("div");
nuevoElemento.innerHTML = "Nuevo elemento";
document.body.appendChild(nuevoElemento);
```

Eliminar elementos:

```
var elemento = document.getElementById("elementoAEliminar");
elemento.parentNode.removeChild(elemento);
```

Escuchar eventos:

Puedes asignar funciones a eventos para que se ejecuten cuando ocurra una acción, como hacer clic en un botón.

```
var boton = document.getElementById("miBoton");
boton.addEventListener("click", function() {
  alert("¡Hiciste clic en el botón!");
});
```

El DOM es esencial en la programación web con JavaScript. Permite la interacción y manipulación de elementos HTML, lo que es crucial para la creación de sitios web dinámicos e interactivos. A medida que avances en tu aprendizaje de JavaScript, te darás cuenta de que el DOM es una herramienta poderosa que se utiliza con frecuencia en el desarrollo web.

7.3. Navegación por el DOM

La navegación en el Document Object Model (DOM) es una parte esencial de la manipulación y el acceso a elementos HTML en una página web utilizando JavaScript. En este tutorial, exploraremos las técnicas de navegación en el DOM de manera exhaustiva.

Navegación Vertical:

La navegación vertical se refiere a la relación jerárquica entre elementos en el DOM, lo que implica buscar elementos padres y sus hijos.

Acceso al Padre:

Para acceder al elemento padre de un nodo en el DOM, puedes utilizar la propiedad parentNode:

```
var hijo = document.getElementById("hijo");
var padre = hijo.parentNode;
```

Esto te dará acceso al nodo padre del elemento con el id "hijo".

Acceso a los Hijos:

Puedes acceder a los nodos hijos de un elemento utilizando las propiedades childNodes, firstChild y lastChild. Ten en cuenta que childNodes incluirá todos los nodos, incluidos los nodos de texto y espacios en blanco.

```
var padre = document.getElementById("padre");
var hijos = padre.childNodes;
var primerHijo = padre.firstChild;
var ultimoHijo = padre.lastChild;
```

Puedes iterar a través de la colección hijos y verificar si un nodo es un elemento utilizando la propiedad nodeType.

Navegación Horizontal:

La navegación horizontal implica acceder a elementos en el mismo nivel jerárquico, como los hermanos de un elemento.

Acceso al Siguiente Hermano:

Para acceder al siguiente hermano de un elemento, puedes utilizar la propiedad nextElementSibling:

```
var elemento = document.getElementById("hermano");
var siguiente = elemento.nextElementSibling;
```

Esto te permite acceder al siguiente elemento en el mismo nivel jerárquico que el elemento seleccionado.

Acceso al Hermano Anterior:

De manera similar, puedes acceder al hermano anterior de un elemento utilizando la propiedad previousElementSibling:

```
var elemento = document.getElementById("hermano");
var anterior = elemento.previousElementSibling;
```

Esto te permite acceder al elemento anterior en el mismo nivel jerárquico.

Navegación Basada en Selectores:

Si deseas navegar en el DOM en función de ciertos criterios o selectores CSS, puedes utilizar el método querySelector o querySelectorAll.

querySelector:

```
var primerParrafo = document.querySelector("p");
```

Esto selecciona el primer párrafo en la página. Puedes utilizar cualquier selector CSS válido como argumento para querySelector.

querySelectorAll:

```
var todosLosParrafos = document.querySelectorAll("p");
```

Esto selecciona todos los párrafos en la página y los almacena en una colección que puedes recorrer.

Navegación por Elementos Relacionados:

Algunas propiedades y métodos permiten navegar por elementos relacionados de manera más específica:

parentElement: Similar a parentNode, pero solo accede a elementos padre (ignora nodos de texto y otros tipos de nodos).

children: Devuelve una colección de elementos hijos.

nextSibling y **previousSibling**: Permiten acceder a cualquier nodo hermano, ya sea un elemento o un nodo de texto.

nextElementSibling y **previousElementSibling**: Acceden solo a elementos hermanos, ignorando los nodos de texto.

Método/Propiedad	Descripción
`parentNode`	Accede al elemento padre del nodo.
`childNodes`	Accede a todos los nodos hijos, incluidos los nodos de texto y espacios en blanco.
`firstChild`	Accede al primer nodo hijo.
`lastChild`	Accede al último nodo hijo.
`nextElementSibling`	Accede al siguiente elemento hermano.
`previousElementSibling`	Accede al elemento hermano anterior.
`querySelector`	Selecciona el primer elemento que coincide con un selector CSS válido.
`querySelectorAll`	Selecciona todos los elementos que coinciden con un selector CSS válido y los almacena en una colección.
`parentElement`	Similar a `parentNode`, pero accede solo a elementos padre (ignora nodos de texto y otros tipos de nodos).
`children`	Devuelve una colección de elementos hijos.
`nextSibling` y `previousSibling`	Permiten acceder a cualquier nodo hermano, ya sea un elemento o un nodo de texto.
`nextElementSibling` y `previousElementSibling`	Acceden solo a elementos hermanos, ignorando los nodos de texto.

7.4. Eventos en JavaScript

Los eventos en JavaScript son acciones o sucesos que ocurren en una página web, como hacer clic en un botón, mover el mouse, presionar una tecla, cargar una página, etc. Los eventos permiten que las páginas web sean interactivas y respondan a las acciones del usuario. En este tutorial, exploraremos los fundamentos de los eventos en JavaScript y cómo utilizarlos en tu desarrollo web.

¿Qué es un Evento?

Un evento es una acción que ocurre en la página web y que JavaScript puede detectar y responder. Algunos ejemplos comunes de eventos son:

click: Ocurre cuando un elemento se hace clic.

mouseover: Ocurre cuando el puntero del mouse se mueve sobre un elemento.

keydown: Ocurre cuando una tecla se presiona.

load: Ocurre cuando una página o recurso se ha cargado por completo.

Escuchando Eventos:

Para responder a eventos, primero debes "escuchar" o "observar" los elementos HTML para detectar cuándo ocurren. Puedes hacerlo asignando "event listeners" (escuchas de eventos) a los elementos HTML que deseas controlar.

```
var boton = document.getElementById("miBoton");
boton.addEventListener("click", function() {
  alert("¡Hiciste clic en el botón!");
});
```

En este ejemplo, estamos utilizando addEventListener para escuchar el evento click en el botón con el id "miBoton". Cuando se hace clic en el botón, se ejecutará la función proporcionada como segundo argumento.

Manejo de Eventos:

Cuando un evento se dispara, se ejecuta una función llamada "manejador de eventos" que contiene el código que deseas ejecutar en respuesta al evento. Esta función puede acceder a información sobre el evento, como la posición del mouse, el elemento objetivo, las teclas presionadas, etc.

```
var boton = document.getElementById("miBoton");

boton.addEventListener("click", function(event) {

  alert("PáginaX: " + event.clientX + " PáginaY: " + event.clientY);

});
```

En este caso, el manejador de eventos muestra las coordenadas del puntero del mouse cuando se hace clic en el botón.

Propagación de Eventos:

Los eventos pueden propagarse o "burbujear" a través de la jerarquía de elementos en el DOM. Esto significa que un evento desencadenado en un elemento se propaga hacia arriba en la jerarquía de elementos hasta llegar al elemento raíz (generalmente document). Puedes detener la propagación de un evento utilizando event.stopPropagation().

```
document.getElementById("padre").addEventListener("click", function() {
  alert("Evento en el elemento padre.");
});
document.getElementById("hijo").addEventListener("click",
function(event) {
  alert("Evento en el elemento hijo.");
  event.stopPropagation(); // Detiene la propagación
});
```

En este ejemplo, al hacer clic en el elemento "hijo", se mostrará el mensaje "Evento en el elemento hijo," y el evento no se propagará al elemento "padre."

Eliminación de Event Listeners:

Puedes eliminar un "event listener" cuando ya no necesites escuchar un evento en un elemento específico utilizando removeEventListener.

```
var boton = document.getElementById("miBoton");
var manejador = function() {
  alert("¡Hiciste clic en el botón!");
};
boton.addEventListener("click", manejador);
// Luego, para eliminar el event listener
boton.removeEventListener("click", manejador);
```

Esto evita que la función manejador se ejecute cuando se hace clic en el botón.

7.5. Validación de formularios

Supoongamos el siguiente fichero HTML,que llamaremos **index.html**

```
<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" type="text/css" href="style.css">
```

```
</head>
<body>
    <form id="miFormulario" onsubmit="return validarFormulario()">
        <label for="nombre">Nombre:</label>
        <input type="text" id="nombre" name="nombre" required>
        <span class="error" id="nombreError">Ingresa un nombre
válido.</span>

        <label for="email">Correo Electrónico:</label>
        <input type="email" id="email" name="email" required>
        <span class="error" id="emailError">Ingresa un correo
válido.</span>

        <input type="submit" value="Enviar">
    </form>

    <script src="script.js"></script>
</body>
</html>
```

A continuación escribiremos el código de la función javascript ValidarFormulario() en un fichero llamado **script.js**

```
function validarFormulario() {
    var nombre = document.getElementById("nombre").value;
    var email = document.getElementById("email").value;
    var nombreError = document.getElementById("nombreError");
    var emailError = document.getElementById("emailError");
     var nombreValido = /^[a-zA-Z ]+$/.test(nombre); // Valida que el
nombre contenga solo letras y espacios
       var emailValido = /^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(email); //
Valida el formato de correo electrónico
    // Restablecer los errores

    nombreError.classList.remove("active");
    emailError.classList.remove("active");

    if (!nombreValido) {
        nombreError.classList.add("active");
    }

    if (!emailValido) {
        emailError.classList.add("active");
    }
```

```
    // Evita que el formulario se envíe si hay errores
    if (!nombreValido || !emailValido) {
        return false;
    }

    // Enviar el formulario si todo está bien
    return true;
}
```

Incluimos un fichero de estilos, **style.css**

```
.error {
    display: none;
    color: red;
}

.error.active {
    display: block;
}
```

En este ejemplo:

Creamos un formulario HTML simple con campos de nombre y correo electrónico, así como mensajes de error ocultos (con la clase "error").

En style.css, definimos el estilo para los mensajes de error.

En script.js, escribimos una función validarFormulario que se ejecutará cuando el usuario intente enviar el formulario. Esta función valida el nombre y el correo electrónico y muestra u oculta mensajes de error según corresponda.

Si hay errores, la función evita que el formulario se envíe y muestra los mensajes de error. Si no hay errores, el formulario se envía.

Módulo 8: Introducción a la Programación Asíncrona (2 Horas)

8.1. Introducción a la asincronía

La programación asíncrona es un concepto fundamental en JavaScript y es esencial para comprender cómo funcionan las aplicaciones web modernas. En este tutorial, exploraremos en profundidad la programación asíncrona, su importancia y cómo se implementa en JavaScript.

¿Qué es la Programación Asíncrona?

La programación asíncrona se refiere a la ejecución de tareas en segundo plano o en paralelo con el flujo principal de un programa. En lugar de esperar a que una tarea se complete antes de continuar con la siguiente, las tareas asíncronas se inician y se ejecutan en segundo plano, lo que permite que el programa sea más eficiente y receptivo.

La Necesidad de la Programación Asíncrona en JavaScript

En el contexto de aplicaciones web, la programación asíncrona es crucial debido a las operaciones que pueden llevar tiempo, como cargar recursos externos (imágenes, archivos, datos), realizar solicitudes a servidores web y manejar eventos del usuario. Si JavaScript bloqueara la ejecución del programa para esperar a que estas operaciones se completen, la experiencia del usuario sería lenta e insatisfactoria.

Cómo se Implementa la Programación Asíncrona en JavaScript

JavaScript utiliza varios mecanismos para implementar la programación asíncrona:

Callbacks: Los callbacks son funciones que se pasan como argumentos a otras funciones y se ejecutan después de que se complete una tarea asíncrona. Por ejemplo, al realizar una solicitud AJAX, puedes proporcionar una función de callback que se ejecutará cuando los datos se reciban.

```
function cargarDatos(url, callback) {
  // Simular una solicitud AJAX
  setTimeout(function() {
    var datos = "Datos obtenidos de " + url;
    callback(datos);
  }, 1000);
}
cargarDatos("https://ejemplo.com/datos", function(datos) {
  console.log(datos);
});
```

Promesas: Las promesas son objetos que representan un valor que puede estar disponible ahora, en el futuro o nunca. Permiten una forma más estructurada de manejar tareas asíncronas y gestionar tanto el éxito como el error.

```
function cargarDatos(url) {
  return new Promise(function(resolve, reject) {
    // Simular una solicitud AJAX
    setTimeout(function() {
      var datos = "Datos obtenidos de " + url;
      resolve(datos); // Éxito
      // reject("Error"); // Error
    }, 1000);
  });
}

cargarDatos("https://ejemplo.com/datos")
  .then(function(datos) {
    console.log(datos);
  })
  .catch(function(error) {
    console.error(error);
  });
```

Async/Await: Esta es una característica más reciente de JavaScript que simplifica la escritura de código asíncrono. Permite escribir código asíncrono de manera más similar a un estilo síncrono, lo que lo hace más legible y fácil de mantener.

```
async function obtenerDatos(url) {
  try {
    let response = await fetch(url);
    let datos = await response.text();
    console.log(datos);
  } catch (error) {
    console.error(error);
  }
}
obtenerDatos("https://ejemplo.com/datos");
```

Conclusión:

La programación asíncrona es esencial en JavaScript para crear aplicaciones web interactivas y eficientes. Al comprender cómo funcionan los callbacks, las promesas y el async/await, puedes gestionar tareas asíncronas de manera efectiva y ofrecer una experiencia de usuario más fluida. La programación asíncrona es un concepto fundamental para desarrolladores web y es un paso importante en el aprendizaje de JavaScript.

8.2. Callbacks

Los callbacks y las funciones asincrónicas son conceptos esenciales en JavaScript que te permiten trabajar con código que se ejecuta de forma no bloqueante y receptiva. En este apartado, exploraremos en profundidad cómo funcionan los callbacks.

Los callbacks son funciones que se pasan como argumentos a otras funciones y se ejecutan después de que se complete una tarea asíncrona o un evento. Son una parte fundamental de la programación asíncrona en JavaScript. Aquí hay un ejemplo básico de un callback:

```
function tareaAsincrona(callback) {
  setTimeout(function() {
    console.log("Tarea asincrónica completada");
    callback();
  }, 1000);
}

function miCallback() {
  console.log("Callback ejecutado");
}

tareaAsincrona(miCallback);
```

En este ejemplo, tareaAsincrona es una función asincrónica que toma un callback como argumento. Después de completar la tarea asincrónica (simulada con setTimeout), ejecuta el callback miCallback.

Problema de Callback Hell:

Un problema común con los callbacks es el "Callback Hell" o el "Infierno de Callbacks", que ocurre cuando tienes múltiples tareas asincrónicas anidadas. Esto puede hacer que el código sea difícil de leer y mantener. Para evitar este problema, se introdujeron las promesas y las funciones asincrónicas.

8.3. Promesas en JavaScript

Las promesas son un concepto fundamental en JavaScript que se utiliza para manejar operaciones asíncronas y eventos en el lenguaje. Proporcionan una forma más estructurada y flexible de trabajar con código asíncrono en comparación con los callbacks. En este tutorial, exploraremos en profundidad cómo funcionan las promesas en JavaScript.

¿Qué son las Promesas?

Una promesa es un objeto que representa un valor que puede estar disponible ahora, en el futuro o nunca. En esencia, una promesa es una garantía de que una tarea asíncrona se completará, y se puede usar para manejar tanto el éxito como el error de dicha tarea.

Estados de una Promesa:

Las promesas en JavaScript pueden estar en uno de los siguientes tres estados:

Pending (Pendiente): Cuando se crea una promesa, se encuentra en estado pendiente. Esto significa que la tarea aún no se ha completado.

Fulfilled (Cumplida): Una promesa está en estado cumplida cuando la tarea se completa con éxito. En este estado, la promesa contiene el valor resultante.

Rejected (Rechazada): Una promesa está en estado rechazada cuando la tarea no se completa con éxito y ocurre un error. En este estado, la promesa contiene información sobre el error.

Creación de Promesas:

Puedes crear una promesa utilizando el constructor Promise. La promesa recibe una función ejecutora con dos argumentos: resolve y reject. El resolve se llama cuando la tarea se completa exitosamente, y el reject se llama cuando ocurre un error.

```
const miPromesa = new Promise((resolve, reject) => {
  // Simular una tarea asíncrona
  setTimeout(() => {
    const exito = true; // Cambia a false para simular un error
    if (exito) {
      resolve("¡Tarea completada con éxito!");
    } else {
      reject("Ocurrió un error.");
    }
  }, 1000);
});
```

Manejo de Promesas

Para manejar el resultado de una promesa, puedes encadenar llamadas a los métodos then y catch. then se utiliza para manejar el caso de éxito, mientras que catch se utiliza para manejar errores.

```
miPromesa
  .then((resultado) => {
    console.log(resultado);
  })
  .catch((error) => {
    console.error(error);
  });
```

También puedes encadenar múltiples then para manejar resultados en una secuencia. Esto es especialmente útil cuando necesitas realizar varias operaciones asíncronas de manera ordenada.

```
miPromesa
  .then((resultado) => {
    console.log(resultado);
    return "Nueva tarea asíncrona";
  })
  .then((nuevoResultado) => {
    console.log(nuevoResultado);
  })
  .catch((error) => {
    console.error(error);
  });
```

Promise.all y Promise.race

Las promesas también se pueden combinar y controlar utilizando Promise.all y Promise.race.

Promise.all toma un array de promesas y devuelve una nueva promesa que se cumple cuando todas las promesas del array se han cumplido o se rechaza cuando al menos una promesa se rechaza.

```
const promesa1 = fetch("https://api1.com");
const promesa2 = fetch("https://api2.com");

Promise.all([promesa1, promesa2])
  .then((resultados) => {
    // Ambas promesas se han cumplido
    console.log(resultados[0]);
    console.log(resultados[1]);
  })
  .catch((error) => {
    console.error("Al menos una promesa se ha rechazado.");
  });
```

Promise.race toma un array de promesas y devuelve una nueva promesa que se cumple o se rechaza tan pronto como una de las promesas del array se cumple o se rechaza.

```
const promesa1 = fetch("https://api1.com");
const promesa2 = fetch("https://api2.com");

Promise.race([promesa1, promesa2])
  .then((resultado) => {
    // La primera promesa que se complete gana
    console.log(resultado);
  })
  .catch((error) => {
    console.error("La promesa más rápida se ha rechazado.");
  });
```

8.4 Async / Await

La introducción de async y await en JavaScript ha revolucionado la forma en que manejamos el código asíncrono. Estas características hacen que el código asincrónico sea más legible, mantenible y parecido al código síncrono. En este tutorial, profundizaremos en async y await y cómo se utilizan para simplificar la asincronía en JavaScript.

Entendiendo Async/Await:

async se usa para declarar una función asincrónica. Una función marcada como async siempre devuelve una promesa, incluso si no hay una palabra clave return.

await se utiliza dentro de una función asincrónica para esperar a que una promesa se resuelva. Esto evita que el código continúe hasta que la promesa esté resuelta.

Sintaxis Básica:

```
async function miFuncionAsincronica() {
  // Usando "await" para esperar a que una promesa se resuelva
  const resultado = await promesaAsincronica();
  // El código aquí se ejecuta después de que la promesa se resuelve
  return resultado;
}

miFuncionAsincronica().then((valor) => {
  console.log(valor);
});
```

Manejo de Errores:

El manejo de errores con async/await se realiza con try...catch, lo que hace que el código sea más legible y similar al código síncrono.

```
async function manejarErrores() {
  try {
    const resultado = await promesaAsincronica();
    return resultado;
  } catch (error) {
    console.error("Se ha producido un error:", error);
  }
}

manejarErrores();
```

Múltiples Promesas en Paralelo:

Una ventaja significativa de async/await es la facilidad para manejar múltiples promesas en paralelo. Puedes usar Promise.all con await para esperar que varias promesas se resuelvan antes de continuar.

```
async function tareasEnParalelo() {
  const resultado1 = await promesa1();
  const resultado2 = await promesa2();
  return [resultado1, resultado2];
}

tareasEnParalelo().then((resultados) => {
  console.log(resultados);
});
```

Async/Await vs. Promesas:

async/await es una mejora significativa en comparación con las promesas, ya que hace que el código sea más fácil de leer y entender, especialmente cuando trabajas con múltiples promesas en paralelo. Sin embargo, debes recordar que las funciones que utilizan await deben ser declaradas como async, mientras que las promesas pueden utilizarse en cualquier función.

Módulo 9: API Fetch Peticiones HTTP (2 Horas)

La Fetch API es una interfaz de JavaScript que proporciona una forma más moderna y flexible de realizar solicitudes HTTP (por ejemplo, para obtener datos de una API o enviar datos a un servidor) en comparación con las técnicas más antiguas, como XMLHttpRequest. La Fetch API se basa en promesas, lo que la hace especialmente útil para manejar operaciones asíncronas en JavaScript. A continuación, te proporcionaré una explicación detallada de cómo usar la Fetch API.

Realizando una Solicitud GET:

El uso más común de la Fetch API es realizar una solicitud GET para recuperar datos de una URL. Aquí tienes un ejemplo:

```
fetch('https://api.example.com/data')
  .then(response => {
    if (!response.ok) {
      throw new Error('Solicitud fallida');
    }
    return response.json();
  })
  .then(data => {
        console.log(data);  //  Aquí  puedes  trabajar  con  los  datos
recuperados.
  })
  .catch(error => {
    console.error(error);
  });
```

Usamos la función fetch() y le pasamos la URL del recurso que queremos obtener.

fetch() devuelve una promesa que resuelve la respuesta (un objeto Response) a la solicitud.

En el primer then(), verificamos si la respuesta fue exitosa (response.ok). Si no, lanzamos un error.

Luego, llamamos a response.json() para analizar la respuesta como JSON, ya que fetch devuelve datos en formato bruto. Esto devuelve una nueva promesa.

En el segundo then(), trabajamos con los datos analizados.

Si ocurre un error en cualquier punto, capturamos la excepción en el catch().

Realizando Solicitudes POST:

La Fetch API también es útil para realizar solicitudes POST al enviar datos a un servidor. Aquí tienes un ejemplo:

```
fetch('https://api.example.com/submit', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({ key: 'value' })
})
  .then(response => {
    if (!response.ok) {
      throw new Error('Solicitud fallida');
    }
    return response.json();
  })
  .then(data => {
    console.log(data);
  })
  .catch(error => {
    console.error(error);
  });
```

En este ejemplo:

- Establecemos el método de solicitud en 'POST'.
- Especificamos el tipo de contenido que estamos enviando en el encabezado (en este caso, JSON).
- Convertimos los datos que deseamos enviar en una cadena JSON utilizando JSON.stringify() y los proporcionamos como el cuerpo de la solicitud.

Controlando los Encabezados:

Puedes controlar los encabezados de la solicitud, agregar autenticación, encabezados personalizados, etc., para adaptarla a tus necesidades específicas.

Uso de Async/Await:

La Fetch API se integra perfectamente con async/await, lo que facilita la escritura de código asíncrono más legible y limpio:

```
async function fetchData() {
  try {
    const response = await fetch('https://api.example.com/data');
    if (!response.ok) {
      throw new Error('Solicitud fallida');
    }
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error(error);
  }
}
```

Cors y Políticas de Seguridad:

Es importante tener en cuenta las políticas de seguridad del mismo origen (Same Origin Policy) y las reglas de CORS (Cross-Origin Resource Sharing) al realizar solicitudes a dominios diferentes al de origen. Esto puede requerir configuración en el servidor o el uso de proxies.

Acceso a un servidor remoto (Ejemplo real)

En este caso, utilizaremos la API pública "JSONPlaceholder" que es una API de prueba comúnmente utilizada para propósitos educativos. La URL de base de JSONPlaceholder es "https://jsonplaceholder.typicode.com".

Supongamos que deseamos obtener una lista de usuarios de JSONPlaceholder:

```
// Realizando una solicitud GET a la API JSONPlaceholder para obtener
la lista de usuarios
fetch('https://jsonplaceholder.typicode.com/users')
  .then(response => {
    if (!response.ok) {
      throw new Error('Solicitud fallida');
    }
    return response.json(); // Analiza la respuesta como JSON
  })
  .then(users => {
```

```
    console.log('Lista de usuarios:');
    console.log(users);
  })
  .catch(error => {
    console.error(error);
  });
```

En este ejemplo:

- Utilizamos fetch() para realizar una solicitud GET a la URL "https://jsonplaceholder.typicode.com/users".
- Verificamos si la respuesta es exitosa con response.ok. Si no lo es, lanzamos un error.
- Luego, utilizamos response.json() para analizar la respuesta como JSON. Esto devuelve una promesa que resuelve en los datos JSON.
- En el segundo then(), trabajamos con la lista de usuarios obtenida.
- Si ocurre un error en cualquier punto del proceso, capturamos la excepción en el catch().

El resultado de esta solicitud será una lista de usuarios en formato JSON. Puedes utilizar estos datos en tu aplicación web como desees.

Este es un ejemplo sencillo, pero ilustra cómo puedes utilizar la Fetch API para interactuar con servidores remotos y obtener datos en tiempo real. La Fetch API es extremadamente versátil y se utiliza comúnmente para acceder a recursos en línea, como API REST, y manipular datos en aplicaciones web.

En resumen, la Fetch API en JavaScript proporciona una forma moderna y poderosa de realizar solicitudes HTTP, tanto GET como POST, y manejar respuestas en formato JSON. Su integración con promesas y async/await hace que trabajar con operaciones asíncronas sea más sencillo y legible. Es una herramienta esencial para interactuar con servidores y servicios web en el desarrollo web moderno.

Módulo 10: Proyecto Final (1 Hora)

10.1. Desarrollo de una pequeña aplicación web

En este proyecto final para el curso de introducción a JavaScript, vamos a crear un juego de adivinanza de números. Los jugadores intentarán adivinar un número aleatorio generado por la computadora dentro de un rango específico. Vamos a construir este juego paso a paso, y te proporcionaré la solución completa al final.

Pasos para Crear el Juego:

Paso 1: Configuración HTML

Comenzaremos creando la estructura HTML básica para el juego. Necesitarás un campo de entrada para que los jugadores ingresen sus adivinanzas, un botón para enviar la adivinanza y un área para mostrar el resultado.

```
<!DOCTYPE html>
<html>
<head>
    <title>Juego de Adivinanza de Números</title>
</head>
<body>
    <h1>Juego de Adivinanza de Números</h1>
    <p>Ingresa un número entre 1 y 100:</p>
    <input type="text" id="adivinanzaInput">
    <button id="adivinarBtn">Adivinar</button>
    <p id="resultado"></p>
    <script src="juego.js"></script>
</body>
</html>
```

Paso 2: Configuración JavaScript

Luego, crearemos un archivo JavaScript (juego.js) para manejar la lógica del juego.

```
// Generar un número aleatorio entre 1 y 100
const numeroAleatorio = Math.floor(Math.random() * 100) + 1;

// Obtener elementos del DOM
const adivinanzaInput = document.getElementById('adivinanzaInput');
```

```
const adivinarBtn = document.getElementById('adivinarBtn');
const resultado = document.getElementById('resultado');

let intentos = 0;
let intentosMaximos = 10;

// Función para comprobar la adivinanza
function comprobarAdivinanza() {
    const adivinanza = parseInt(adivinanzaInput.value);

    if (isNaN(adivinanza)) {
        resultado.textContent = 'Ingresa un número válido.';
    } else {
        intentos++;
        if (adivinanza === numeroAleatorio) {
            resultado.textContent = `¡Felicidades! Adivinaste el número
${numeroAleatorio} en ${intentos} intentos.`;
            adivinanzaInput.disabled = true;
            adivinarBtn.disabled = true;
        } else if (adivinanza > numeroAleatorio) {
                resultado.textContent = 'Demasiado alto. Intenta de
nuevo.';
        } else {
                resultado.textContent = 'Demasiado bajo. Intenta de
nuevo.';
        }

        if (intentos === intentosMaximos) {
                        resultado.textContent = `¡Has agotado tus
${intentosMaximos} intentos! El número era ${numeroAleatorio}.`;
            adivinanzaInput.disabled = true;
            adivinarBtn.disabled = true;
        }
    }
}

// Agregar un evento al botón
adivinarBtn.addEventListener('click', comprobarAdivinanza);
```

Pasos 3: Lógica del Juego

1. Generamos un número aleatorio entre 1 y 100 utilizando Math.random().
2. Obtenemos elementos del DOM para interactuar con ellos.

3. Definimos una función comprobarAdivinanza que se ejecutará cuando se haga clic en el botón.
4. En la función comprobarAdivinanza, comparamos la adivinanza del jugador con el número aleatorio. Mostramos mensajes según si la adivinanza es correcta, demasiado alta o demasiado baja.
5. Limitamos el número de intentos a un máximo de 10. Si el jugador agota todos los intentos, mostramos el número correcto.

10.2. Aplicación de los conceptos aprendidos sobre promesas y acceso a servidores remotos

En este proyecto, crearemos una aplicación de pronóstico del clima que utiliza promesas y realiza solicitudes a un servicio de pronóstico del clima en línea para obtener datos reales. Vamos a utilizar la API pública de OpenWeatherMap para obtener información sobre el clima actual en una ubicación específica. Aquí tienes los pasos para completar este proyecto:

Pasos para Crear la Aplicación de Clima:

Paso 1: Configuración del Proyecto

Crea una estructura de directorios para tu proyecto y asegúrate de tener un archivo HTML, un archivo JavaScript (app.js) y un archivo CSS (opcional) para el estilo.

proyecto-clima/

├── index.html

├── app.js

├── style.css (opcional)

Paso 2: Configuración de OpenWeatherMap API

Regístrate en el servicio gratuito de OpenWeatherMap para obtener una clave de API. Esta clave es necesaria para hacer solicitudes al servicio de pronóstico del clima. OpenWeatherMap

Paso 3: Estructura HTML

Crea la estructura HTML básica para tu aplicación de clima. Incluye un campo de entrada para ingresar la ubicación y un botón para obtener el pronóstico del clima. Además, agrega un área donde mostrarás los resultados del pronóstico del clima.

```
<!DOCTYPE html>
<html>
<head>
    <title>Aplicación de Clima</title>
</head>
<body>
    <h1>Aplicación de Clima</h1>
       <input type="text" id="ubicacionInput" placeholder="Ingresa la
ubicación">
    <button id="obtenerClimaBtn">Obtener Clima</button>
    <div id="resultado"></div>
    <script src="app.js"></script>
</body>
</html>
```

Paso 4: Configuración de JavaScript

En tu archivo app.js, configura la lógica de JavaScript para hacer solicitudes a la API de OpenWeatherMap y mostrar los resultados en la página.

```
const apiKey = 'TU_CLAVE_DE_API'; // Reemplaza con tu clave de
OpenWeatherMap
const baseUrl = 'https://api.openweathermap.org/data/2.5/weather';

const ubicacionInput = document.getElementById('ubicacionInput');
const obtenerClimaBtn = document.getElementById('obtenerClimaBtn');
const resultado = document.getElementById('resultado');

obtenerClimaBtn.addEventListener('click', () => {
  const ubicacion = ubicacionInput.value;
  obtenerClima(ubicacion);
});

async function obtenerClima(ubicacion) {
  try {
                              const       response       =       await
fetch(`${baseUrl}?q=${ubicacion}&appid=${apiKey}`);
    if (!response.ok) {
      throw new Error('No se pudo obtener el clima.');
```

```
    }

    const data = await response.json();
    mostrarClima(data);
  } catch (error) {
    resultado.textContent = `Error: ${error.message}`;
  }
}

function mostrarClima(data) {
  const nombre = data.name;
  const temperatura = data.main.temp;
  const descripcion = data.weather[0].description;

    resultado.textContent = `Clima en ${nombre}: ${temperatura}°C,
${descripcion}`;
}
```

Paso 5: Estilo (Opcional)

Si lo deseas, puedes agregar estilos CSS para mejorar la apariencia de tu aplicación.

Resumen:

Este proyecto utiliza promesas y solicitudes a un servidor remoto público (OpenWeatherMap) para obtener información sobre el clima en una ubicación específica. Los usuarios pueden ingresar una ubicación y obtener el pronóstico del clima actual. Puedes personalizar y expandir este proyecto añadiendo más características, como un pronóstico extendido o un diseño más atractivo. Es una excelente manera de aprender cómo funcionan las promesas y cómo interactuar con servicios web en tiempo real.

Módulo 11: Recursos Adicionales y Próximos Pasos

11.1. Recursos de aprendizaje

Libros:

"Eloquent JavaScript" de Marijn Haverbeke: Un libro gratuito en línea que cubre JavaScript desde los fundamentos hasta temas avanzados. Eloquent JavaScript

"You Don't Know JS" de Kyle Simpson: Una serie de libros que explora aspectos más profundos de JavaScript. You Don't Know JS

"JavaScript: The Definitive Guide" de David Flanagan: Un libro completo y detallado sobre JavaScript.

Cursos en línea:

freeCodeCamp: Ofrece una serie de cursos interactivos de JavaScript y proyectos prácticos.

Codecademy: Proporciona un curso interactivo de JavaScript para principiantes.

edX - Introduction to JavaScript: Un curso gratuito de edX que introduce los conceptos básicos de JavaScript.

Videos en YouTube:

The Net Ninja: Ofrece una serie de tutoriales de JavaScript y otros temas de desarrollo web.

Traversy Media: Tiene una amplia variedad de tutoriales sobre JavaScript y desarrollo web.

Páginas Web y Documentación:

MDN Web Docs: La documentación oficial de JavaScript en MDN proporciona recursos exhaustivos y ejemplos.

JavaScript.info: Un sitio web que ofrece una introducción completa y actualizada a JavaScript.

Ejercicios y Plataformas de Desarrollo:

Codewars: Plataforma para practicar resolviendo desafíos de programación en JavaScript.

HackerRank: Ofrece desafíos de programación y tutoriales de 10 días en JavaScript.

Proyectos Personales:

La práctica es fundamental para aprender. Intenta construir tus propios proyectos, desde pequeñas aplicaciones hasta juegos simples. Puedes utilizar GitHub para alojar tus proyectos y colaborar con otros desarrolladores.

Comunidad y Foros:

Participa en la comunidad de desarrollo web y JavaScript en sitios como Stack Overflow para obtener ayuda y aprender de otros.

Conferencias y Podcasts:

Explora conferencias como JSConf y podcasts como JavaScript Jabber para mantenerte al tanto de las tendencias y novedades en JavaScript.

11.2. Próximos pasos en el aprendizaje de JavaScript

Una vez que hayas completado un curso de iniciación a JavaScript, es esencial continuar con tu aprendizaje y desarrollar tus habilidades como programador. Aquí te proporcionaré un tutorial con los pasos que debes seguir para avanzar en tu camino como desarrollador de JavaScript.

Paso 1: Profundiza en los Fundamentos

Antes de aventurarte en temas avanzados, asegúrate de comprender los conceptos fundamentales de JavaScript. Repasa los siguientes temas:

Variables y tipos de datos.

Estructuras de control (if, else, while, for, etc.).

Funciones y cómo declararlas.

Objetos y arreglos.

Eventos y manipulación del DOM.

Paso 2: Practica con Proyectos Personales

La práctica es la clave para mejorar tus habilidades. Comienza a construir tus propios proyectos. Inicia con aplicaciones simples y avanza gradualmente hacia proyectos más complejos. Algunas ideas de proyectos incluyen:

Una lista de tareas.

Una calculadora.

Un juego de adivinanza.

Una aplicación de pronóstico del clima (como se mencionó en proyectos anteriores).

Paso 3: Profundiza en Conceptos Intermedios

A medida que te sientas más cómodo con los conceptos básicos, puedes profundizar en temas intermedios:

Promesas y async/await para manejar la asincronía.

Trabajo con APIs y solicitudes HTTP.

Manipulación avanzada del DOM.

Almacenamiento local (localStorage y sessionStorage).

Patrones de diseño de JavaScript.

Paso 4: Explore Frameworks y Bibliotecas

Aprende a utilizar frameworks y bibliotecas populares de JavaScript. Algunos ejemplos incluyen:

React: Para el desarrollo de interfaces de usuario (UI).

Angular: Otro framework para la creación de aplicaciones web.

Vue.js: Una biblioteca progresiva para construir interfaces de usuario.

Selecciona uno de estos y profundiza en él, ya que te permitirán crear aplicaciones web más complejas y modernas.

Paso 5: Aprende Control de Versiones

Es importante aprender a utilizar un sistema de control de versiones como Git. Esto te permitirá colaborar con otros desarrolladores y mantener un registro de las versiones de tu código.

Paso 6: Practica Regularmente

La consistencia es clave para convertirte en un desarrollador experimentado. Dedica tiempo regularmente a programar y abordar nuevos proyectos o desafíos.

Paso 7: Colabora con la Comunidad

Unete a la comunidad de desarrolladores. Participa en foros, blogs y grupos de desarrollo. Aprende de otros y comparte tu conocimiento.

Paso 8: Mantente Actualizado

JavaScript y el desarrollo web evolucionan constantemente. Mantente actualizado con las últimas tendencias, estándares y tecnologías. Suscríbete a blogs, sigue conferencias y lee libros relacionados con JavaScript.

Paso 9: Aporta a Proyectos de Código Abierto

Contribuir a proyectos de código abierto es una excelente manera de aprender y colaborar con otros desarrolladores. Encuentra proyectos que te interesen y comienza a contribuir.

Paso 10: Nunca Dejes de Aprender

El desarrollo web es un campo en constante evolución. Nunca dejes de aprender y experimentar con nuevas tecnologías. El aprendizaje continuo es fundamental para tener éxito en la industria.

Recuerda que el desarrollo de habilidades lleva tiempo y paciencia. Sigue estos pasos y estarás bien encaminado para convertirte en un desarrollador de JavaScript habilidoso y versátil. ¡Buena suerte en tu viaje de aprendizaje!

11.3. Desarrollo web moderno y marcos de trabajo (frameworks)

El desarrollo web moderno ha experimentado un rápido avance en los últimos años, impulsado por tecnologías y prácticas que permiten la creación de aplicaciones web altamente interactivas, eficientes y atractivas. JavaScript desempeña un papel central en esta evolución, y el uso de frameworks y bibliotecas de JavaScript ha sido fundamental para simplificar el desarrollo y mejorar la eficiencia.

Aquí hay una descripción general de algunos aspectos del desarrollo web moderno y los frameworks más populares utilizados con JavaScript:

Componentes Clave del Desarrollo Web Moderno:

HTML5 y CSS3: Estándares web actualizados que permiten la creación de interfaces de usuario más ricas y flexibles.

JavaScript: El lenguaje de programación fundamental para el desarrollo web.

AJAX: Tecnología que permite la comunicación asincrónica con el servidor, lo que brinda una experiencia de usuario más fluida.

APIs Web: Las APIs proporcionan funcionalidades específicas, como geolocalización, notificaciones push y acceso a la cámara.

Responsive Web Design: Diseño adaptable que garantiza una experiencia óptima en dispositivos de diferentes tamaños.

Frameworks y Bibliotecas de JavaScript:

React: Desarrollado por Facebook, React es una biblioteca de JavaScript para la creación de interfaces de usuario. Utiliza un enfoque basado en componentes y es ampliamente utilizado para construir aplicaciones de una sola página (SPA).

Angular: Desarrollado por Google, Angular es un framework completo de JavaScript para el desarrollo web. Proporciona herramientas para la creación de aplicaciones web complejas y se enfoca en la estructura y la modularidad del código.

Vue.js: Un framework de JavaScript progresivo y fácil de integrar en proyectos existentes. Vue.js es conocido por su enfoque en la capa de vista y su simplicidad.

Ember.js: Otro framework de JavaScript que sigue la convención sobre la configuración y es conocido por su eficiencia en la creación de aplicaciones web grandes.

Svelte: Un enfoque novedoso que compila el código escrito en Svelte en JavaScript altamente eficiente durante la compilación, lo que lo hace rápido y fácil de usar.

Gestión de Estado:

Los frameworks modernos a menudo ofrecen soluciones para la gestión del estado de la aplicación. Algunos ejemplos incluyen:

Redux: Una biblioteca de gestión de estado para aplicaciones JavaScript, especialmente aquellas basadas en React.

Vuex: Una solución de gestión de estado específica de Vue.js.

NgRx: Un conjunto de bibliotecas para Angular que facilita la gestión del estado.

Herramientas y Entorno de Desarrollo:

Webpack: Un empaquetador de módulos que permite la construcción eficiente de aplicaciones y la administración de dependencias.

Babel: Un transpilador que permite escribir código JavaScript moderno y compilarlo para versiones anteriores de JavaScript que funcionen en todos los navegadores.

ESLint y Prettier: Herramientas de linting y formateo de código para mantener un código limpio y consistente.

Servicios en la Nube y API REST:

El desarrollo web moderno a menudo se integra con servicios en la nube y se comunica con API REST para acceder a datos y funcionalidades externas.

Pruebas y Automatización:

El desarrollo moderno enfatiza las pruebas automatizadas, utilizando herramientas como Jest, Mocha y Jasmine. También se enfoca en la automatización de tareas de desarrollo y despliegue.

En resumen, el desarrollo web moderno se caracteriza por la creación de aplicaciones web interactivas y eficientes utilizando JavaScript como lenguaje principal. Los frameworks y bibliotecas de JavaScript son herramientas esenciales para simplificar el proceso de desarrollo y permitir la creación de aplicaciones web complejas. La elección de un framework o biblioteca dependerá de los requisitos y las preferencias del proyecto.