Javascript

Juan Carlos Trejo

1

Módulo 1

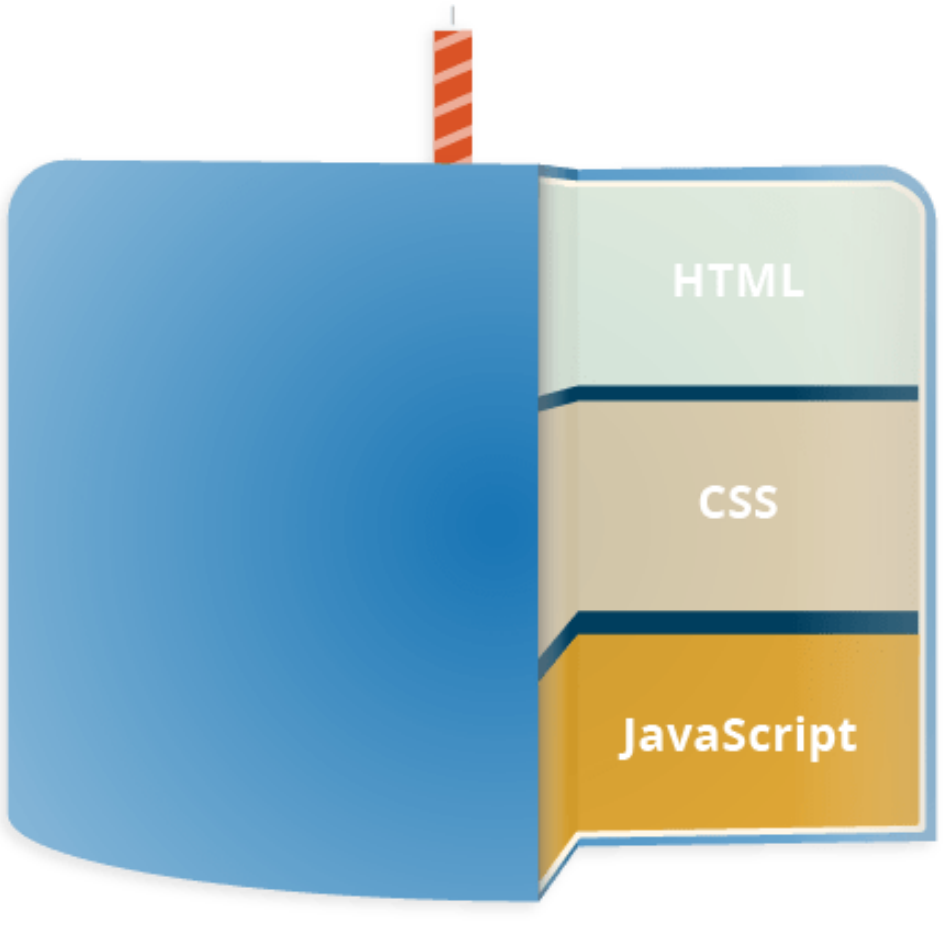
Introducción

Introducción

¿Qué es un Javascript?

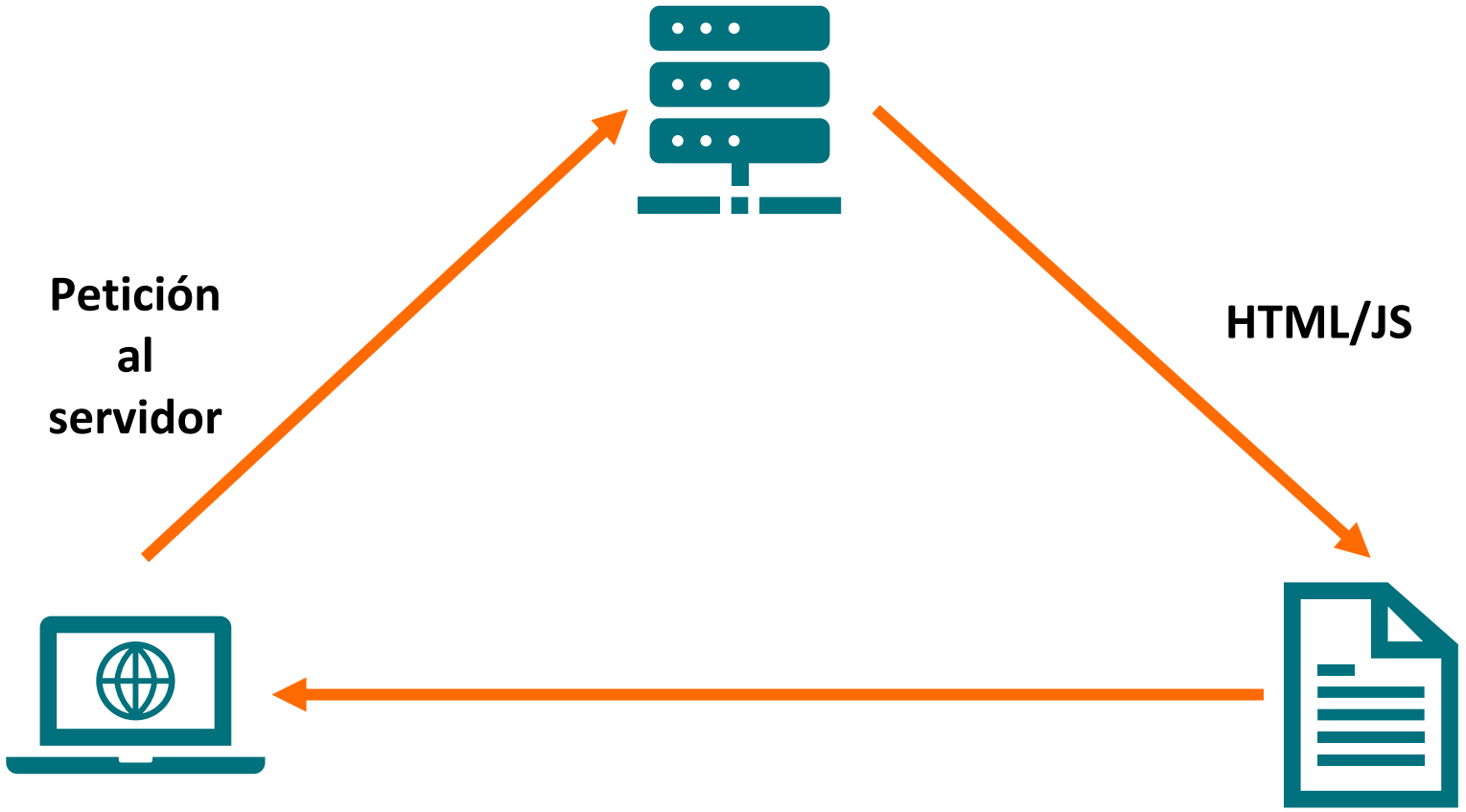
JavaScript es un lenguaje de programación o de secuencias de comandos que te permite implementar funciones complejas en páginas web.

Introducción



Componentes de una pagina WEB

Introducción



Introducción

¿Qué puede hacer Javascript?

- Agregar nuevo HTML a la página, cambiar el contenido existente y modificar estilos.
- Reaccionar a las acciones del usuario, ejecutarse con los clics del ratón, movimientos del puntero y al oprimir teclas.
- Enviar solicitudes de red a servidores remotos, descargar y cargar archivos.
- Obtener y configurar cookies, hacer preguntas al visitante y mostrar mensajes.
- Recordar datos en el lado del cliente con el almacenamiento local (“local storage”).

Consola de desarrollador

El código es propenso a errores. Para ver los errores y obtener mucha otra información útil sobre los scripts, se han incorporado “herramientas de desarrollo” en los navegadores. En la mayoría de los exploradores se puede activar con la tecla F12.

Introducción

Manuales

- La especificación EMAC

ECMAScript® 2021 Language Specification (ecma-international.org)

- MDN Mozilla Refrence

JavaScript reference - JavaScript | MDN (mozilla.org)

2

Módulo 2

Variables y objetos

Variables

Hay ocho tipos de datos básicos en JavaScript. Podemos almacenar un valor de cualquier tipo dentro de una variable. Por ejemplo, una variable puede contener en un momento un string y luego almacenar un número.

```
1  // no hay error
2  let message = "hola";
3  message = 123456;
```

Variables

Number	Para números de cualquier tipo: enteros o de punto flotante, los enteros están limitados por $\pm(2^{53}-1)$.
Bigint	Para números enteros de longitud arbitraria.
String	Para cadenas. Una cadena puede tener cero o más caracteres, no hay un tipo especial para un único carácter.
boolean	Para verdadero y falso: true/false.
Null	Para valores desconocidos – un tipo independiente que tiene un solo valor nulo: null
Undefined	Para valores no asignados – un tipo independiente que tiene un único valor indefinido: undefined
Symbol	Para identificadores únicos.
Object	Para estructuras de datos complejas.

Variables

Una variable es un “almacén con un nombre” para guardar datos. Existen 3 formas de declarar una variable:

1. let
2. const
3. var

Variables
let

Formas de declarar una variable con let

```
let message;
message = 'Hola!';
alert(message);
```

```
let user = 'John';
let age = 25;
let message = 'Hola';
```

```
let user = 'John', age = 25, message = 'Hola';
```

```
let user = 'John',
  age = 25,
  message = 'Hola';
```

Variables

const

Una variable de tipo const es inmutable, por lo cual solo se puede guardar una vez y sólo una vez un valor. Tales constantes se nombran utilizando letras mayúsculas y guiones bajos.

```
const COLOR_RED = "#F00";
const COLOR_GREEN = "#0F0";
const COLOR_BLUE = "#00F";
const COLOR_ORANGE = "#FF7F00";
```

```
const COLOR_RED = "#F00";
COLOR_RED = "#0F0";
```

Variables
var

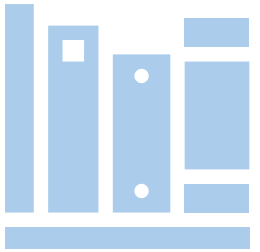
Las variables declaradas con var pueden tener a la función como entorno de visibilidad, o bien ser globales. Su visibilidad atraviesa los bloques.

```
if (true) {
  var test = true;
}

alert(test)
```

```
if (true) {
  let test = true;
}

alert(test);
```

Variables
objetos

Los objetos son usados para almacenar colecciones de varios datos y entidades más complejas asociados con un nombre clave. En JavaScript, los objetos penetran casi todos los aspectos del lenguaje.



Variables
objetos

Los objetos son usados para almacenar colecciones de varios datos y entidades más complejas asociados con un nombre clave. En JavaScript, los objetos penetran casi todos los aspectos del lenguaje.

```
// Sintaxis de "constructor de objetos"
let usuario = new Object();
// Sintaxis de "objeto literal"
let usuario = {};
```

Variables
objetos

Se puede declarar un objeto y agregar propiedades. Una propiedad tiene una clave (también conocida como “nombre” o “identificador”) antes de los dos puntos ":" y un valor a la derecha.

```
// Objeto user
let user = {
    // En la clave "name" se almacena el valor "John"
    name: "John",
    // En la clave "age" se almacena el valor 30
    age: 30
};
```

Variables
objetos

Se puede declarar un objeto y agregar propiedades. Una propiedad tiene una clave (también conocida como “nombre” o “identificador”) antes de los dos puntos ":" y un valor a la derecha.

```
// Objeto user
let user = {
    // En la clave "name" se almacena el valor "John"
    name: "John",
    // En la clave "age" se almacena el valor 30
    age: 30
};
```

```
// Obteniendo los valores
alert( user.name ); // John
alert( user.age ); // 30
```

Variables
objetos

1. Agregar una propiedad

```
user.isAdmin = false;
alert(user);
```

2. Eliminar una propiedad

```
delete user.name;
alert(user);
```

Variables
objetos

3. Acceder a una propiedad

```
// Objeto user
let user = {
    // En la clave "name" se almacena el valor "John"
    name: "John",
    // En la clave "age" se almacena el valor 30
    age: 30
};


console.log(user.age);
console.log(user['name']);
```

3

Módulo 3

Funciones

Funciones

Las funciones son los principales “bloques de construcción” del programa. Permiten que el código se llame muchas veces sin repetición.

```
function name(parameter1, parameter2, ... parameterN) {
    // body
}
```

```
function showMessage() {
    alert( '¡Hola a todos!' );
}

showMessage();
showMessage();
```

Funciones

Si una variable con el mismo nombre se declara dentro de la función, le *hace sombra* a la externa.

```
let usuario = 'John';

function showMessage() {
  let usuario = "Bob"; // declara variable local

  let mensaje = 'Hello, ' + usuario;
  // ¿Cual es la salida?
  alert(mensaje);
}

// la función crea y utiliza su propia variable local usuario
showMessage();

// ¿Cual es la salida?
alert( usuario );
```

Funciones

Se pueden pasar N numero de parámetros a una función.

```
// parámetros: from, text
function showMessage(from, text) {
    alert(from + ': ' + text);
}
// Llamar la funcion
showMessage('Ann', '¡Hola!');
```

Funciones

Si una función es llamada, pero no se le proporciona un argumento, su valor correspondiente se convierte en undefined. Podemos especificar un valor llamado “predeterminado” o “por defecto” (es el valor que se usa si el argumento fue omitido) en la declaración de función usando =

```
function showMessage(from, text = "sin texto") {
    alert( from + ": " + text );
}

showMessage("Ann");
```

Funciones

En JavaScript, una función no es una estructura del lenguaje, sino un tipo de valor especial. Existe otra sintaxis para crear una función que se llama una Expresión de Función.

```
let saludar = function() {
    alert( "Hola" );
};

typeof saludar;
```

Funciones

```
function saludar() {    // a) Crear la funcion
    alert( "Hola" );
}

let copia = saludar;    // b) Hacer una copia

copia(); // Hola        // c) Ejecutar la funcion copia
saludar(); // Hola      // d) Ejecutar la funcion original
```

Funciones callback

Una función de callback es una función que se pasa a otra función como un argumento, que luego se invoca dentro de la función externa para completar algún tipo de rutina o acción, permitiendo así que el código en la función callback se ejecute en un momento específico.

```
function saludar(nombre) {
    alert('Hola ' + nombre);
}

function procesarEntradaUsuario(callback) {
    var nombre = prompt('Por favor ingresa tu nombre.');
    callback(nombre);
}

procesarEntradaUsuario(saludar);
```

Funciones flecha

Una función de flecha (también conocida como "arrow function" en inglés) es una característica introducida en ECMAScript 6 (ES6) de JavaScript. Proporciona una sintaxis más concisa y simplificada para crear funciones en comparación con las funciones tradicionales.

```
(parametro1, parametro2, ...) => {
    // Cuerpo de la función
    // Puede contener una o varias instrucciones
    return resultado;
}
```

Funciones flecha

Una función de flecha (también conocida como "arrow function" en inglés) es una característica introducida en ECMAScript 6 (ES6) de JavaScript. Proporciona una sintaxis más concisa y simplificada para crear funciones en comparación con las funciones tradicionales.

```
// Funcion 1
// Recibir parametros. Funcion normal
const sumar1 = function(num1, num2) {
    return num1 + num2;
}

// Funcion 2 Recibir parametros
// Recibir parametros. Funcion flecha
const sumar2 = (num1, num2) => {
    return num1 + num2;
}

// Funcion 3 Recibir parametros
// Recibir parametros. Funcion flecha
const sumar3 = (num1, num2) => num1 + num2;
```

Funciones flecha

Una función de flecha (también conocida como "arrow function" en inglés) es una característica introducida en ECMAScript 6 (ES6) de JavaScript. Proporciona una sintaxis más concisa y simplificada para crear funciones en comparación con las funciones tradicionales.

```
// Funcion 4 Un solo parametro
const imprimir = nombre => console.log(nombre);

// Funcion 5 Sin parametros
const imprimirDocument = () => { console.log(document) }

// Funcion 6 Retornar objeto
const retornarObjeto = (id, nombre) => ({ id: id, nombre: nombre})
```

4

Módulo 4

Números

Números

JavaScript proporciona una amplia variedad de funciones y métodos para el manejo de números.

1. Declaración de variables numéricas.

```
var numero = 10;
let otroNumero = 20;
const PI = 3.14159;
```

Números

JavaScript proporciona una amplia variedad de funciones y métodos para el manejo de números.

2. Operadores aritméticos

```
var suma = 5 + 3;     // Suma
var resta = 10 - 4;    // Resta
var multiplicacion = 6 * 2;  // Multiplicación
var division = 8 / 4;    // División
var modulo = 10 % 3;   // Módulo (resto de la división)
```

Números

2. Funciones matemáticas

JavaScript proporciona funciones matemáticas incorporadas para realizar operaciones más complejas.

```
// Devuelve 5
var raizCuadrada = Math.sqrt(25);
 // Devuelve 8 (2 elevado a la potencia de 3)
var potencia = Math.pow(2, 3);
// Genera un número aleatorio entre 0 y 1
var numeroAleatorio = Math.random();
```

Números

4. Conversores de tipo

JavaScript permite convertir entre diferentes tipos de datos, incluidos los números.

Función	Descripción
parseInt()	Convierte una cadena en un número entero.
parseFloat()	Convierte una cadena en un número de punto flotante.

```
var numeroTexto = "10";
// Convierte "10" en 10
var numeroEntero = parseInt(numeroTexto);
```

5. Comparaciones numéricas

Puedes comparar números utilizando operadores de comparación como <, >, <=, >=, ===, y !==. Estos operadores devolverán un valor booleano (true o false) según la comparación.

```
var x = 5;
var y = 10;
// Devuelve false
var esMayor = x > y;
// Devuelve false
var sonIguales = x === y;
```

Números

Números

El objeto Number

En JavaScript, la clase Number es un objeto incorporado que proporciona funcionalidades y propiedades relacionadas con números.

Números

El objeto Number

Funciones estáticas

Método	Descripción
Number.parseFloat()	Convierte una cadena en un número de punto flotante.
Number.parseInt()	Convierte una cadena en un número entero.
Number.isFinite()	Determina si un valor es un número finito.
Number.isNaN()	Determina si un valor es "Not-a-Number" (no es un número).
Number.isInteger()	Determina si un valor es un número entero.
Number.isSafeInteger()	Determina si un valor es un número entero seguro (puede ser representado de forma segura en JavaScript).

Números

El objeto Number

Funciones de instancia

Método	Descripción
toExponential()	Devuelve una cadena que representa el objeto en notación exponencial.
toFixed()	Devuelve una cadena que representa el objeto redondeado a un número específico de decimales.
toPrecision()	Devuelve una cadena que representa el objeto con una precisión especificada.
toString()	Devuelve una cadena que representa el objeto.
valueOf()	Devuelve el valor primitivo del objeto.

Números

Método isInteger

Determina si el valor proporcionado es un número entero.

```
let entero = 42;
let esEntero = Number.isInteger(entero);
// Resultado: true
console.log(esEntero);
```

Números

En JavaScript, el valor NaN representa "Not a Number" (No es un número). Es un valor especial que se utiliza para indicar que una operación aritmética ha producido un resultado que no es un número válido

```
console.log(0 / 0)
console.log(Math.sqrt(-1))
console.log(NaN + 5)
```

Números

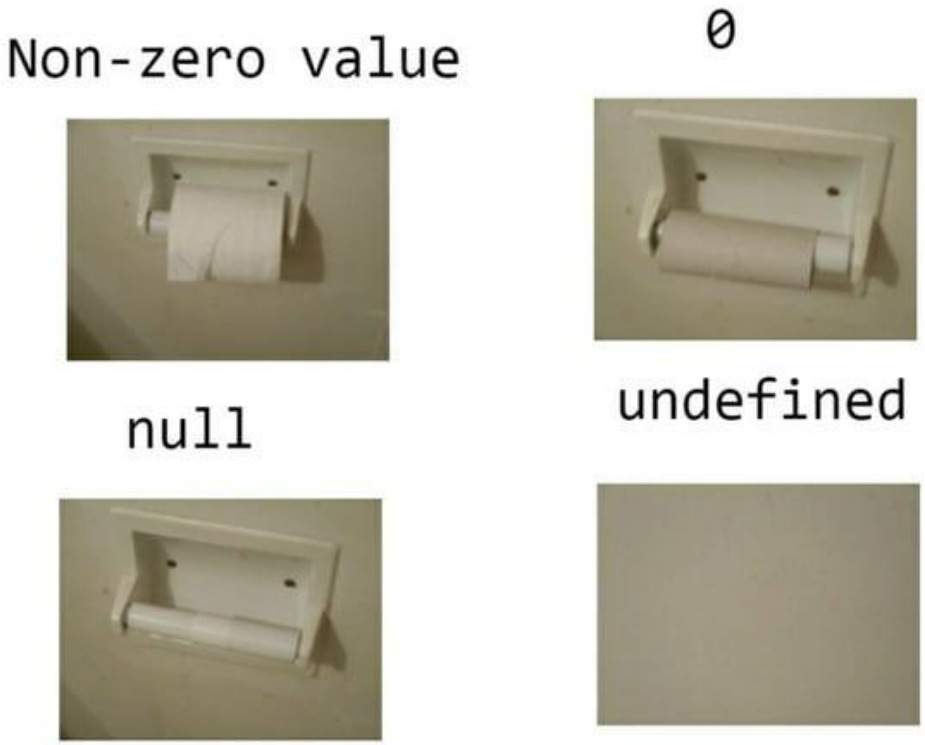
Método isNaN

Determina si el valor proporcionado no es un número (NaN).

```
let noEsNumero = "Hola";
let esNaN = isNaN(noEsNumero);
// Resultado: true
console.log(esNaN);
```

Números

- **Non-zero (Distinto de cero):** Este término se refiere a cualquier valor numérico que no sea igual a cero.
- **0:** Es simplemente el valor numérico cero.
- **null:** Es un valor especial en JavaScript que indica la ausencia de un objeto o valor.
- **undefined:** Es un valor que tiene una variable cuando no se le ha asignado un valor.



Números

Un entero seguro en JavaScript es aquel que se encuentra en el rango -2^53 hasta 2^53, inclusive.

```
var enteroSeguro = Number.MAX_SAFE_INTEGER;
var noEsEnteroSeguro = Number.MAX_SAFE_INTEGER + 1;

console.log(Number.isSafeInteger(enteroSeguro));  // true
console.log(Number.isSafeInteger(noEsEnteroSeguro));  // false
```

Números

Método isSafeInteger

Determina si el valor proporcionado es un número entero seguro en términos de representación precisa

```
// Es el máximo entero seguro en JavaScript
let enteroSeguro = 9007199254740991;
let esEnteroSeguro = Number.isSafeInteger(enteroSeguro);
// Resultado: true
console.log(esEnteroSeguro);
```

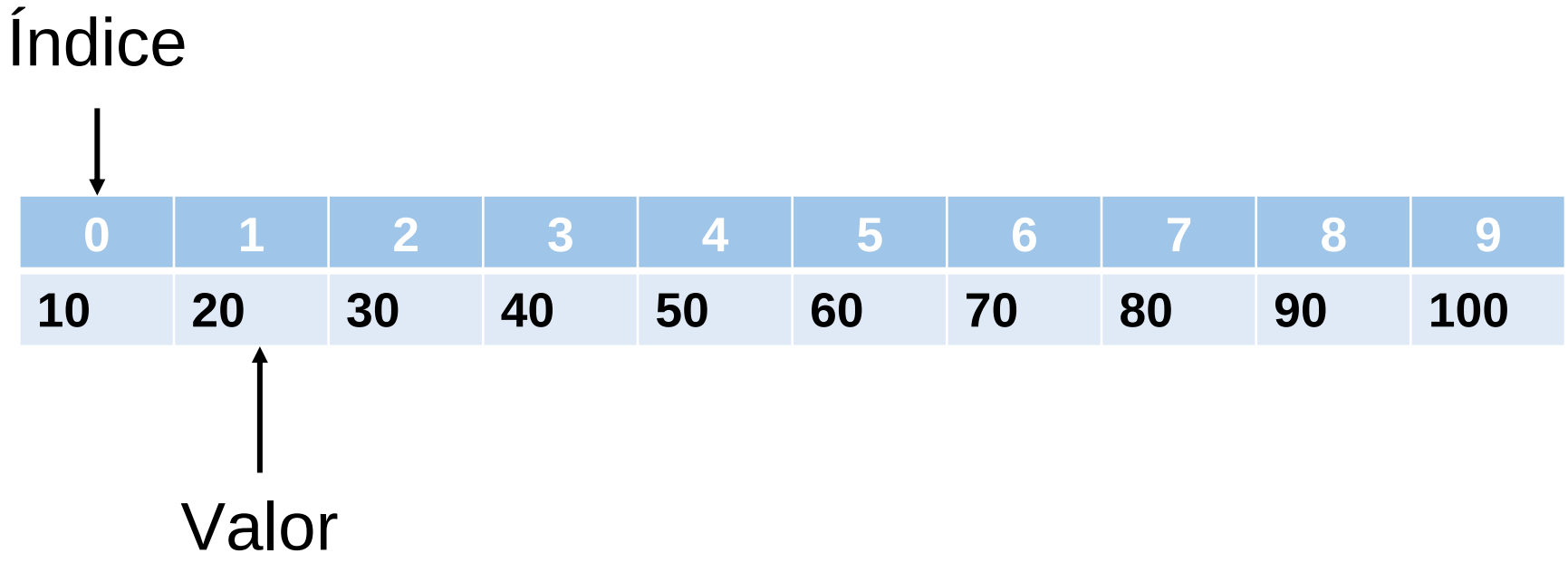
5

Módulo 5

Arreglos

Arreglos

Es una estructura o colección para insertar datos de forma ordenanda.



Arreglos

Es una estructura o colección para insertar datos de forma ordenanda.

```
let arreglo = new Array();
let arreglo = [];
```

```
let frutas = ["Apple", "Orange", "Plum"];

alert( frutas[0] ); // Apple
alert( frutas[1] ); // Orange
alert( frutas[2] ); // Plum
```

Arreglos

Podemos acceder a los métodos de manipulación de arreglos a través de:

```
// Metodos estaticos
Array.method()
// Metodos de instancia
Array.prototype.method()
```

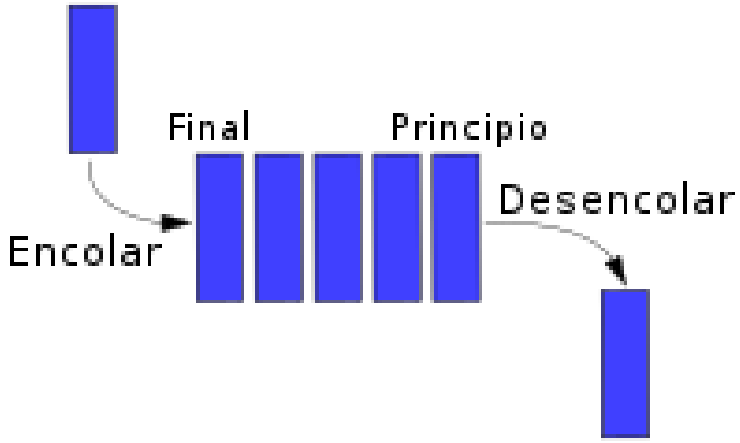
Arreglos

Principales métodos

Método	Descripción
push()	Añade uno o más elementos al final del arreglo.
pop()	Elimina el último elemento del arreglo.
shift()	Elimina el primer elemento del arreglo.
unshift()	Añade uno o más elementos al principio del arreglo.
splice()	Permite agregar o eliminar elementos de cualquier posición en el arreglo.
filter()	Retorna una porción del arreglo en base a una condición.
concat()	Combina dos o más arreglos.
forEach()	Itera sobre cada elemento del arreglo y ejecuta una función proporcionada.
map()	Crea un nuevo arreglo con los resultados de aplicar una función a cada elemento del arreglo.
find()	Retorna el primer elemento que cumple con una condición dada en la función proporcionada.
findIndex()	Retorna el índice del primer elemento que cumple con una condición dada en la función proporcionada.
keys()	Retorna un nuevo objeto Iterador que contiene las claves de índice del arreglo.
values()	Retorna un nuevo objeto Iterador que contiene los valores del arreglo.
fill()	Rellena todos los elementos de un arreglo con un valor estático.
from()	Crea un nuevo arreglo a partir de un objeto iterable o de un arreglo similar a un arreglo.

Arreglos

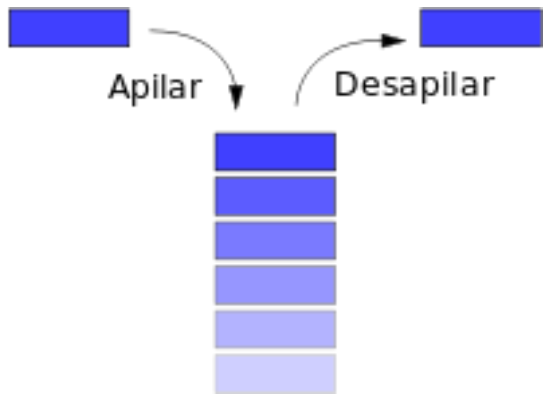
Push/Shift




```
// Creamos un arreglo inicial
let miArreglo = [1, 2, 3, 4, 5];

// Utilizamos el método 'shift' para
// eliminar el primer elemento del arreglo
let primerElemento = miArreglo.shift();

// Utilizamos el método 'push' para
// agregar elementos al final del arreglo
miArreglo.push(6, 7);

console.log("Arreglo final:", miArreglo);
```

Arreglos
Push/Pop

```
// Creamos un arreglo inicial
let miArreglo = ["manzana", "banana", "cereza"];

// Utilizamos el método 'push' para
// agregar un elemento al final del arreglo
miArreglo.push("durazno");

// Utilizamos el método 'pop' para
// eliminar el último elemento del arreglo
let ultimoElemento = miArreglo.pop();

// Salida final
console.log("Arreglo final:", miArreglo);
```



Arreglos

Spread operator

En JavaScript, el operador de propagación (spread operator) es un operador que se introdujo en ECMAScript 6 (también conocido como ES6) y se utiliza para descomponer elementos en varias ubicaciones. Este operador se representa con tres puntos consecutivos (...).

```
const array1 = [1, 2, 3];
const array2 = [4, 5, 6];
// Resultado: [1, 2, 3, 4, 5, 6]
const combinedArray = [...array1, ...array2];
```

```
const obj1 = { a: 1, b: 2 };
const obj2 = { c: 3, d: 4 };
// Resultado: { a: 1, b: 2, c: 3, d: 4 }
const combinedObject = { ...obj1, ...obj2 };
```

6

Módulo 6

Cadenas

Cadenas

El objeto **String** se utiliza para representar y manipular una secuencia de caracteres.

Las cadenas son útiles para almacenar datos que se pueden representar en forma de texto. Algunas de las operaciones más utilizadas en cadenas son verificar su length, para construirlas y concatenarlas usando operadores de cadena + y +=

Cadenas

Las cadenas se pueden crear como primitivas, a partir de cadena literales o como objetos, usando el constructor

```
// Cadenas primitivas
const string1 = "Una cadena primitiva";
const string2 = 'Tambien una cadena primitiva';
const string3 = `Otra cadena primitiva más`;

// Un objeto de tipo String
const string4 = new String("Un objeto String");
```

Las strings primitivas y los objetos string se pueden usar indistintamente en la mayoría de las situaciones.

Cadenas

Acceder a los caracteres de una cadena

Hay dos formas de acceder a un carácter individual en una cadena.

La primera es con el método *charAt()*

```
// devuelve "a"
return "cat".charAt(1);
```

La otra forma es tratar a la cadena como un objeto similar a un arreglo, donde los caracteres individuales corresponden a un índice

```
// devuelve "a"
return "cat"[1];
```

Cadenas

Comparar cadenas

```
let a = "a";
let b = "b";

if (a < b) {
  // true
  console.log(a + " es menor que " + b);
} else if (a > b) {
  console.log(a + " es mayor que " + b);
} else {
  console.log(a + " y " + b + " son iguales.");
}
```

Cadenas

Comparar cadenas a vacío

En JavaScript, es importante comprobar si una cadena está vacía o nula antes de realizar cualquier operación.

Una cadena vacía es una cadena que no tiene caracteres, mientras que una cadena nula es una cadena que no tiene ningún valor asignado.

Cadenas

Comparar cadenas a vacío

```
let cadena = "";

if (cadena.length === 0) {
  console.log("La cadena esta vacia");
} else {
  console.log("La cadena no esta vacia");
}
```

En este ejemplo, se usa la propiedad length de la cadena para comprobar su longitud, si esta es 0, la cadena es vacía.

Cadenas

Comparar cadenas a vacío

```
let cadena = "      ";

if (cadena.trim().length === 0) {
  console.log("La cadena esta vacía");
} else {
  console.log("La cadena no esta vacía");
}
```

En este ejemplo, se usa el método trim para borrar los espacios y la propiedad length para comprobar la longitud dela cadena, si esta es 0 la cadena es vacía.

Cadenas

Comparar cadenas a vacío

```
let  cadena = "texto";

if (!cadena) {
    console.log("La cadena está vacía.");
} else {
    console.log("La cadena no está vacía.");
}
```

En este ejemplo, JavaScript trata una cadena vacía como false cuando se convierte a un valor booleano.

Cadenas

Tipo primitivo vs Objeto

JavaScript distingue entre objetos String y valores de primitivas string

Las cadenas literales (denotadas por comillas simples o dobles en un contexto que no es de constructor (es decir, llamado sin usar la palabra clave new) son cadenas primitivas.

```
let s_prim = "cadena";
let s_obj = new String(s_prim);

// Registra "string"
console.log(typeof s_prim);
// Registra "object"
console.log(typeof s_obj);
```

Cadenas

Tipo primitivo vs Objeto

Las primitivas de String y los objetos String también dan diferente resultado cuando se usa eval(). Las primitivas pasadas a eval se tratan como código fuente; Los objetos String se tratan como todos los demás objetos, devuelven el objeto.

```
// crea una string primitiva
let s1 = "2 + 2";
// crea un objeto String
let s2 = new String("2 + 2");
// devuelve el número 4
console.log(eval(s1));
// devuelve la cadena "2 + 2"
console.log(eval(s2));
```

Cadenas

String templates

Las plantillas de cadena son una característica introducida en ECMAScript 6 que proporciona una forma más conveniente y legible de crear cadenas en JavaScript.

En lugar de usar comillas simples (') o dobles (") para delimitar las cadenas, las plantillas de cadena utilizan comillas invertidas (`) para crear cadenas utilizando ${}.

```
const nombre = "Juan";
const edad = 30;
const frase = `Hola, me llamo ${nombre} y tengo ${edad} años.`;
// Resultado: "Hola, me llamo Juan y tengo 30 años."
console.log(frase);
```

7

Módulo 7

Setenencias control

Sentencias de Control

Sentencia IF – ELSE

La sentencia IF evalúa la condición en los paréntesis, y si el resultado es verdadero (true), ejecuta un bloque de código.

```
if (condición) {
    // Bloque de código que se ejecuta
    // si la condición es verdadera
} else {
    // Bloque de código que se ejecuta
    // si la condición es falsa
}
```

```
let edad = 20;

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

Sentencias de Control

Conversión booleana

La sentencia if (…) evalúa la expresión dentro de sus paréntesis y convierte el resultado en booleano.

Recordemos las reglas de conversión del capítulo Conversiones de Tipos:

- El número 0, un string vacío "", null, undefined, y NaN, se convierten en false. Por esto son llamados valores “falsos”.
- El resto de los valores se convierten en true, entonces los llamaremos valores “verdaderos”.

```
// 0 es falso
if (0) {

}

// 1 es verdadero
if (1) {

}
```

Sentencias de Control

Operación ternaria

El operador ternario nos permite ejecutar esto en una forma más corta y simple.

El operador está representado por el signo de cierre de interrogación “?”. A veces es llamado “ternario” porque el operador tiene tres operandos, es el único operador de JavaScript que tiene esa cantidad.

La Sintaxis es:

```
let result =
    condition
        ? value1
        : value2;
```

Sentencias de Control

Operación ternaria

Se evalúa la condición, si es verdadera entonces devuelve “Eres mayor de edad” , de lo contrario “Eres menor de edad”.

```
let edad = 20;

let mensaje =
    (edad >= 18)
        ? "Eres mayor de edad"
        : "Eres menor de edad";

console.log(mensaje);
```

Sentencias de Control

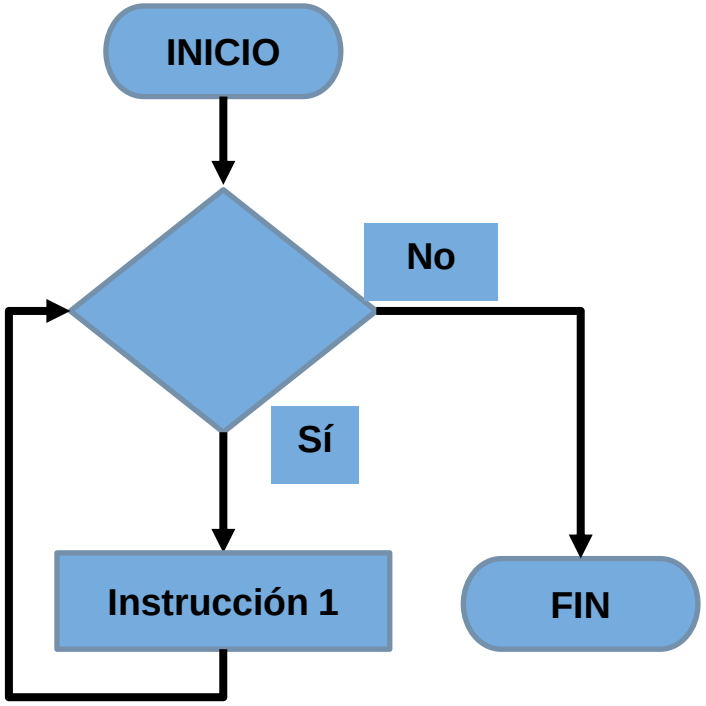
Ejercicio

Convertir la siguiente estructura if – else a un operador ternario.

```
if (edad < 3) {
    mensaje = '¡Hola, bebé!';
  } else if (edad < 18) {
    mensaje = '¡Hola!';
  } else if (edad < 100) {
    mensaje = '¡Felicidades!';
  } else {
    mensaje = '¡Qué edad tan inusual!';
  }

alert(mensaje)
```

Sentencias de Control

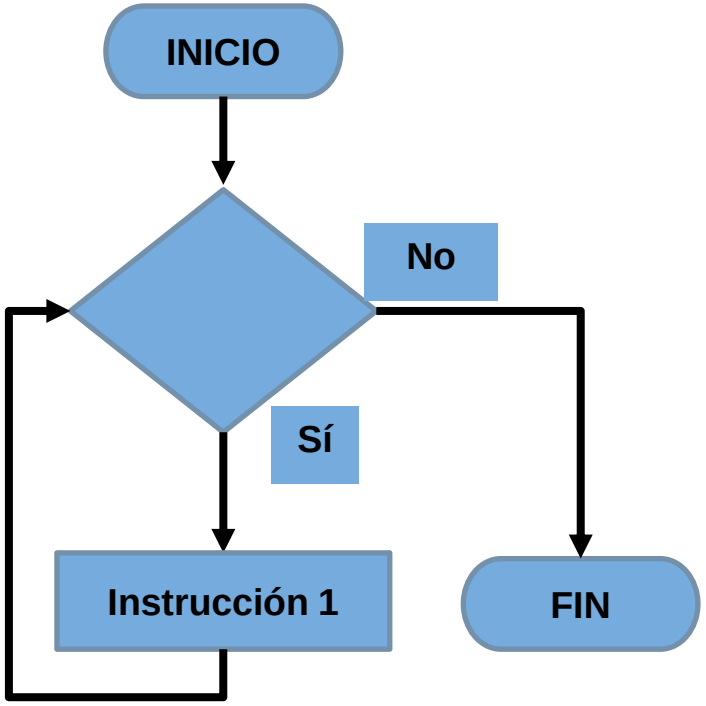
While




```
var contador = 0;

while (contador < 5) {
  console.log("El contador es: " + contador);
  contador++;
}
```

Sentencias de Control

For




```for (var contador = 0; contador < 5; contador++) {
    console.log("El contador es: " + contador);
}
```

Sentencias de Control

For.. of

El bucle for...of es útil cuando solo necesitas acceder a los valores de los elementos de un arreglo, ya que te permite iterar directamente sobre ellos.

```
var arreglo = ["a", "b", "c", "d"];

console.log("Iterando con for...of:");
for (var elemento of arreglo) {
  console.log(elemento);
}
```

Sentencias de Control

For.. in

for...in está diseñado para iterar sobre las propiedades enumerables de un objeto, y aunque también puede usarse para iterar sobre los índices de un arreglo, puede no ser tan eficiente o predecible como for...of,

```
var arreglo = ["a", "b", "c", "d"];

console.log("Iterando con for...of:");
for (var elemento in arreglo) {
  console.log(elemento);
}
```

Sentencias de Control

forEach

El método forEach itera sobre cada elemento del arreglo y ejecuta la función proporcionada en cada iteración. En cada iteración, el valor del elemento actual se asigna automáticamente al parámetro de la función (en este caso, fruta), lo que nos permite realizar alguna operación con él.

```
// Creamos un arreglo de frutas
let frutas = ["manzana", "banana", "cereza"];

// Utilizamos el método 'forEach' para iterar
// sobre cada elemento del arreglo
frutas.forEach(function(fruta) {
  console.log(fruta);
});
```

Arreglos

Map

El método map crea un nuevo arreglo que contiene los valores resultantes de aplicar la operación a cada elemento del arreglo original.

```
// Creamos un arreglo de números
let numeros = [1, 2, 3, 4, 5];

// Utilizamos el método 'map' para realizar
// una operación en cada elemento del arreglo
let duplicados = numeros.map(function(numero) {
  return numero * 2;
});
```

Arreglos
Filter

El método filter en JavaScript se utiliza para crear un nuevo arreglo con todos los elementos que cumplan cierta condición especificada en una función de filtrado.

```
// Creamos un arreglo de números
let numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

// Utilizamos el método 'filter' para obtener
// los números pares del arreglo
let numerosPares = numeros.filter(function(numero) {
  return numero % 2 === 0;
});

console.log("Números pares:", numerosPares);
```

Arreglos

find/findIndex

Retorna el primer elemento que cumple con una condición dada en la función proporcionada.

```
// Definir un arreglo de objetos
let personas = [
  { nombre: 'Juan', edad: 25 },
  { nombre: 'María', edad: 30 },
  { nombre: 'Ana', edad: 22 },
  { nombre: 'Carlos', edad: 28 }
];

// Utilizar la función find para encontrar la primera
// persona que cumple con cierta condición
let personaEncontrada = personas.find(function(persona) {
  return persona.edad > 25;
});

// Imprimir el resultado
console.log(personaEncontrada);
```

Arreglos
from

Array.from() toma el objeto iterable (en este caso, una cadena de texto) y crea un nuevo arreglo donde cada elemento del arreglo es un carácter de la cadena original.

```
// Crear un objeto iterable
let iterable = 'Hola';

// Usar Array.from() para crear un nuevo arreglo
let nuevoArreglo = Array.from(iterable);

// Resultado: ['H', 'o', 'l', 'a']
console.log(nuevoArreglo);
```

Arreglos
from

Se puede usar la función para transformar cada elemento durante la creación del nuevo arreglo.

```
// Crear un arreglo de números
let numeros = [1, 2, 3, 4];

// Usar Array.from() con una función de mapeo
// para duplicar cada número
let nuevoArreglo = Array.from(numeros, num => num * 2);

// Resultado: [2, 4, 6, 8]
console.log(nuevoArreglo);
```

8

Módulo 8

Destructuración

Destructuring

La sintaxis de asignación de desestructuración es una expresión de JavaScript que hace posible descomprimir valores de matrices o propiedades de objetos en distintas variables.

```
// Objeto user
let user = {
    // En la clave "name" se almacena el valor "John"
    name: "John",
    // En la clave "age" se almacena el valor 30
    age: 30
};

const {nombre, edad} = producto;
```

Destructuring

La sintaxis de asignación de desestructuración es una expresión de JavaScript que hace posible descomprimir valores de matrices o propiedades de objetos en distintas variables.

```
let [variable1, variable2, ..., variableN] = arreglo;
```

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

let [a, b, c] = numeros;

console.log(a); // Resultado: 1
console.log(b); // Resultado: 2
console.log(c); // Resultado: 3
```

Destructuring

También podemos asignar propiedades a variables con cualquier nombre.

```
const myObject = {
    one: 'a',
    two: 'b',
    three: 'c'
};

// ES6 destructuring example
const { one: first, two: second, three: third } = myObject;

// Ahora se puede usar las variables first, second y third
console.log(first);  // 'a'
console.log(second); // 'b'
console.log(third);  // 'c'
```

Destructuring

También se puede hacer referencia a objetos anidados más complejos.

```
const meta = {
    title: 'Destructuracion',
    authors: [{
            firstname: 'Lot',
            lastname: 'Alvarez'
        }
    ],
    publisher: {
        name: 'SitePoint',
        url: 'http://www.pagina.com/'
    }
};

const {
    title: doc,
    authors: [{ firstname: name }],
    publisher: { url: web }
} = meta;
```

Destructuring

¿Qué produce el siguiente código?

```
var a = 1, b = 2;

// Destructuracion de intercambio
[a, b] = [b, a];

console.log(a)
console.log(b)
```

Destructuring

La función **printInfo** toma un solo parámetro que es un objeto. En lugar de pasar el objeto directamente como un argumento, se está utilizando la destructuring en la propia declaración de la función para extraer las propiedades específicas del objeto **(name, age, city)** y luego imprimirlas.

```
// Definición de una función que toma un objeto como parámetro
function printInfo({ name, age, city }) {
    console.log(`Nombre: ${name}`);
    console.log(`Edad: ${age}`);
    console.log(`Ciudad: ${city}`);
}

// Objeto que será pasado como argumento a la función
const person = {
    name: 'Juan',
    age: 25,
    city: 'Mexico'
};

// Llamada a la función con destructuración del objeto
printInfo(person);
```

Destructuring

```
// Definición de una función que devuelve un
// objeto con múltiples valores
function getPersonInfo() {
    const name = 'Ana';
    const age = 30;
    const city = 'Barcelona';

    // Devolver un objeto con destructuración
    return { name, age, city };
}

// Llamada a la función y destructuración de
// los valores devueltos
const { name, age, city } = getPersonInfo();

// Imprimir los valores obtenidos
console.log(`Nombre: ${name}`);
console.log(`Edad: ${age}`);
console.log(`Ciudad: ${city}`);
```

La función **getPersonInfo** devuelve un objeto con las propiedades name, age y city. Luego, al llamar a la función y asignar el resultado a una variable utilizando destructuración, se puede acceder fácilmente a esos valores individualmente.

Destructuring

La función **forEach** se utiliza para iterar sobre cada objeto en el array libros. En el cuerpo de la función de iteración, la destructuración se utiliza para extraer las propiedades titulo, autor e isbn de cada objeto, y luego se imprimen esos valores.

```
// Array de objetos que representan libros
const libros = [
    { titulo: 'JavaScript: The Good Parts', autor: 'Douglas Crockford', isbn: '978-0596517748' },
    { titulo: 'Clean Code', autor: 'Robert C. Martin', isbn: '978-0132350884' },
    { titulo: 'The Pragmatic Programmer', autor: 'Andrew Hunt', isbn: '978-0201616224' }
  ];

  // Iterar sobre el array de libros utilizando destructuración
  libros.forEach(({ titulo, autor, isbn }) => {
    console.log('--- Libro ---');
    console.log(`Título: ${titulo}`);
    console.log(`Autor: ${autor}`);
    console.log(`ISBN: ${isbn}`);
    console.log('\n');
  });
```

9

Módulo 9

Programación Orientada
a Objetos

Programación Orientada a Objetos

El comportamiento del programa es llevado a cabo por objetos, entidades que representan elementos del problema a resolver y tienen atributos y comportamiento.

Clase

Modelo donde se establecen las características comunes de un grupo de objetos.

Instancia

Es un ejemplar o un objeto que se ha creado a partir de la clase o molde

OBJETO →

Atributos de Clase

Los atributos, también llamados datos o variables miembro son porciones de información que un objeto posee o conoce de sí mismo.

```
class Carro {
    color;
    modelo;
    linea;
}
```

Métodos de clase o instancia

Un método es una abstracción de una operación que puede hacer o realizarse con un objeto. Una clase puede declarar cualquier número de métodos que lleven a cabo operaciones de lo más variado con los objetos. Los métodos de instancia operan sobre las variables de instancia de los objetos pero también tienen acceso a las variables de clase.

```
class Carro {
    color;

    cambiarColor() {

    }
}
```

Métodos set y get

Los métodos get y set, son métodos que se asocian a un atributo de la clase y que se usan en las clases para mostrar (get) o modificar (set) el valor de un atributo. El nombre del método siempre será get o set y a continuación el nombre del atributo.

```
class Carro {
    // Atributo color
    _color;

    // Getter para obtener el color
    get color() {
        return this._color;
    }

    // Setter para establecer el color
    set color(nuevoColor) {
        this._color = nuevoColor;
    }
}
```

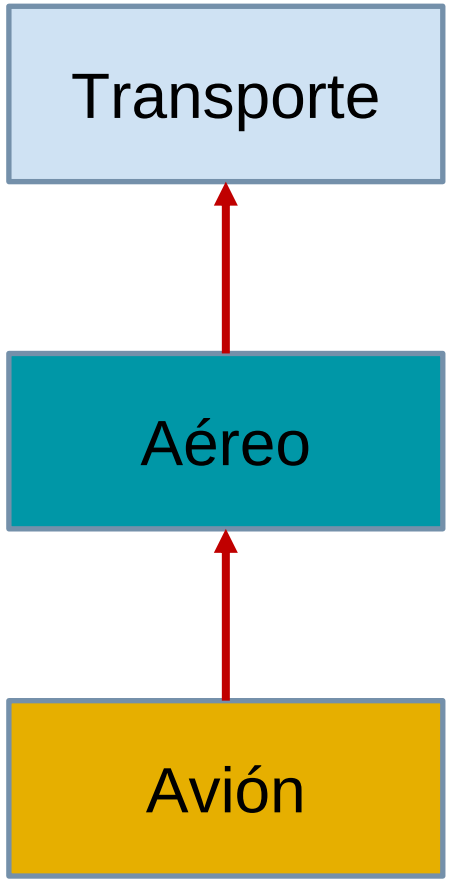
Obtener el estado de un objeto

En muchas ocasiones necesitamos obtener el estado actual de un objeto, para lo cual podemos crear un método personalizado o podemos sobre escribir el método toString()

```
class Carro {
    // Atributo color
    color;

    // Método toString para obtener la
    // representación en cadena del objeto
    toString() {
        return `Color: ${this.color}`;
    }
}
```

Herencia

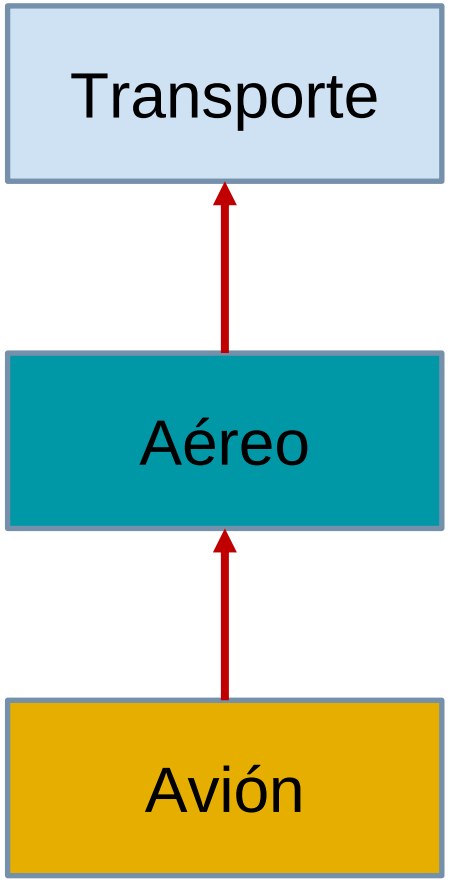
Es el mecanismo por el cual una clase permite heredar las características (atributos y métodos) de otra clase.



Herencia

Superclase

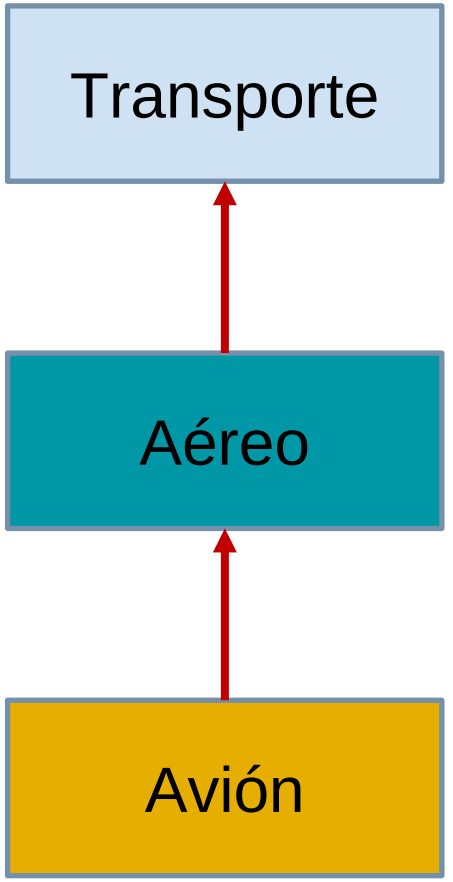
La clase cuyas características se heredan se conoce como superclase (o una clase base o una clase principal).



Herencia

Subclase

La clase que hereda la otra clase se conoce como subclase (o una clase derivada, clase extendida o clase hija). La subclase puede agregar sus propios campos y métodos además de los campos y métodos de la superclase.

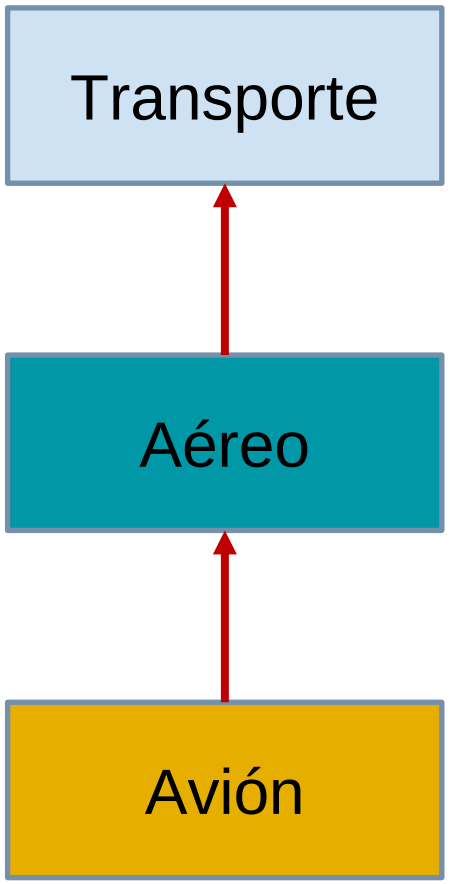


Herencia

```
class Transporte {

}
```

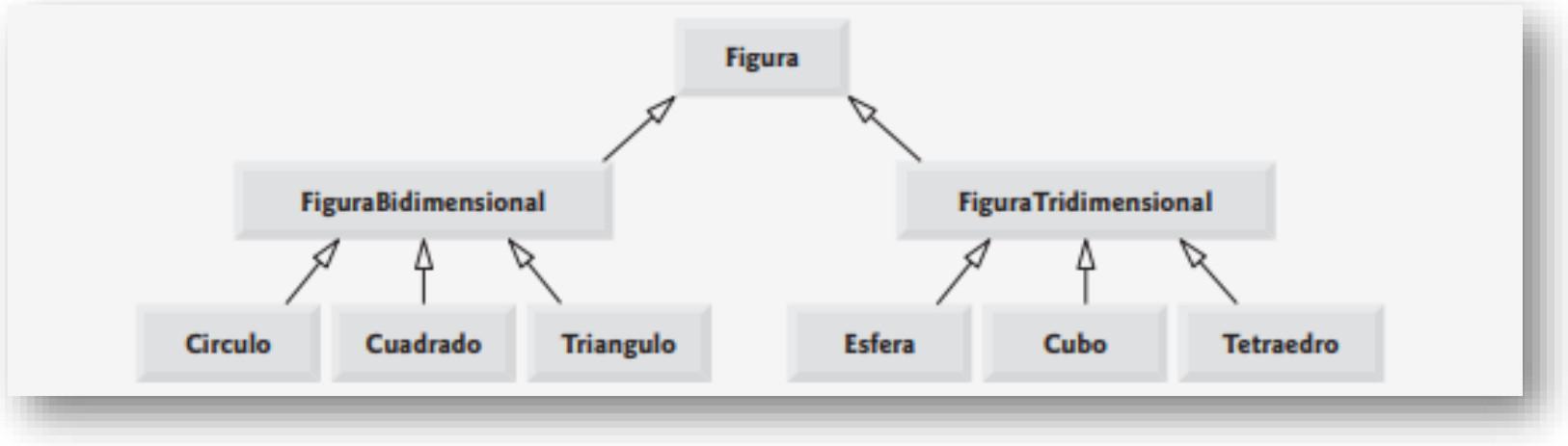
```
class Aereo extends Transporte {

}
```

```
class Avion extends Aereo {

}
```



Ejercicio 1

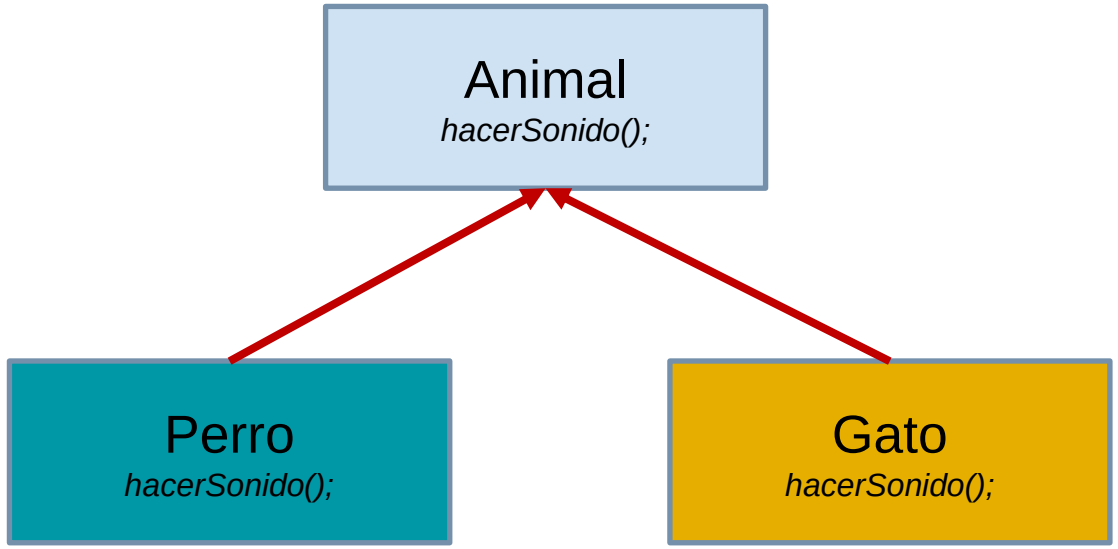
Implementar el siguiente diagrama de clases con Herencia



Polimorfismo

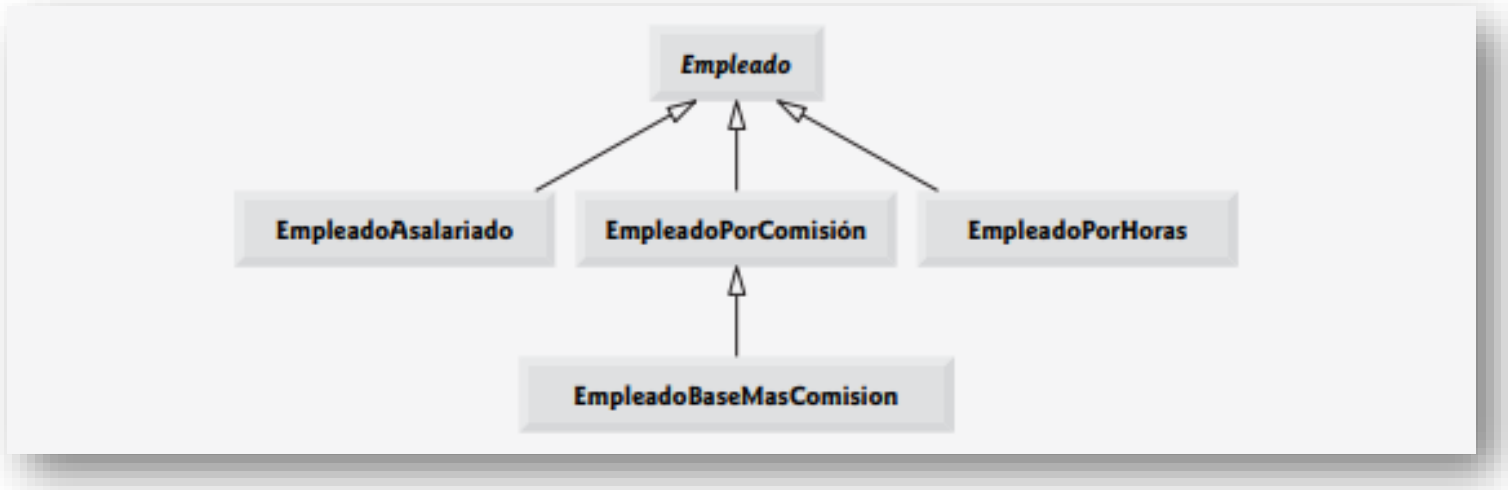
En programación orientada a objetos, polimorfismo es la capacidad que tienen los objetos de una clase en ofrecer respuesta distinta e independiente en función de los parámetros.

Es una característica de la programación orientada a objetos que permite llamar a métodos con igual nombre pero que pertenecen a clases distintas.



Ejercicio 1

Implementar el siguiente diagrama de clases



Cada clase debe contener:

1. Atributos
2. Métodos setter y getter
3. Constructor(es)

Ejercicio 1

Cada clase tiene las siguientes especificaciones:

	ingresos	toString
Empleado	abstract	*primerNombre apellidoPaterno* número de seguro social: *NSS*
Empleado-Asalariado	salarioSemanal	empleado asalariado: *primerNombre apellidoPaterno* número de seguro social: *NSS* salario semanal: *salarioSemanal*

EmpleadoPor-Horas	if horas <= 40
	  sueldo * horas
	else if horas > 40
	  40 * sueldo +
	  (horas - 40) *
	  sueldo * 1.5
		empleado por horas: *primerNombre apellidoPaterno*
		número de seguro social: *NSS*
		sueldo por horas: *sueldo*; horas trabajadas: *horas*

EmpleadoPor-Comisión	tarifaComisión * ventasBrutas	empleado por comisión: *primerNombre apellidoPaterno* número de seguro social: *NSS* ventas brutas: *ventasBrutas*; tarifa de comisión: *tarifaComisión*
Empleado-BaseMas-Comision	(tarifaComision * ventasBrutas) + salarioBase	empleado por comisión con salario base: *primerNombre apellidoPaterno* número de seguro social: *NSS* ventas brutas: *ventasBrutas*; tarifa de comisión: *tarifaComision*; salario base: *salarioBase*

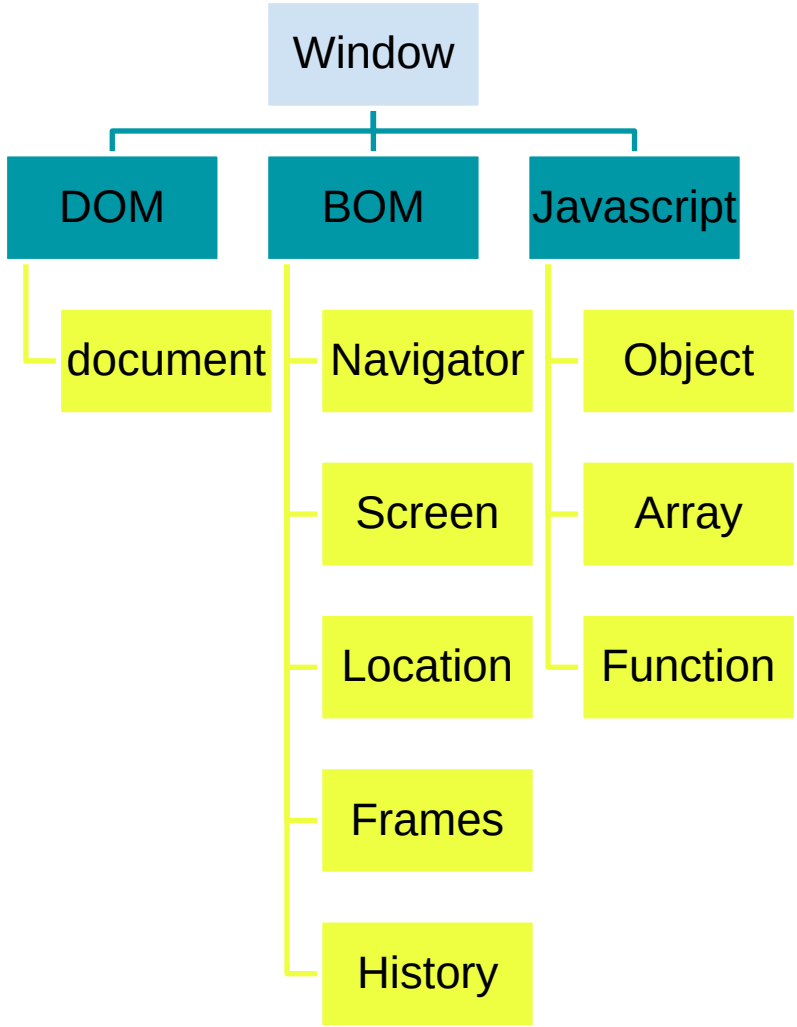
10

Módulo 10

DOM

DOM

Cuando se ejecuta el explorador, carga por detrás un entorno que da soporte a todas las operaciones mismas del explorador.



BOM (Modelo de Objetos del Navegador)

El Modelo de Objetos del Navegador (Browser Object Model, BOM) son objetos adicionales proporcionados por el navegador (entorno host) para trabajar con todo excepto el documento.

Por ejemplo:

- El objeto navigator proporciona información sobre el navegador y el sistema operativo. Hay muchas propiedades.
- El objeto location nos permite leer la URL actual y puede redirigir el navegador a una nueva.

DOM (Modelo de Objetos del Documento)

- La estructura de un documento HTML son las etiquetas.
- Según el *Modelo de Objetos del Documento* (DOM), cada etiqueta HTML es un objeto. Las etiquetas anidadas son llamadas “hijas” de la etiqueta que las contiene. El texto dentro de una etiqueta también es un objeto.
- Todos estos objetos son accesibles empleando JavaScript, y podemos usarlos para modificar la página.

```
<!DOCTYPE HTML>
<html>
    <head>
        <title>
            Titulo del documento
        </title>
    </head>

    <body>
        Cuerpo del documento
    </body>
</html>
```

Recorriendo el DOM

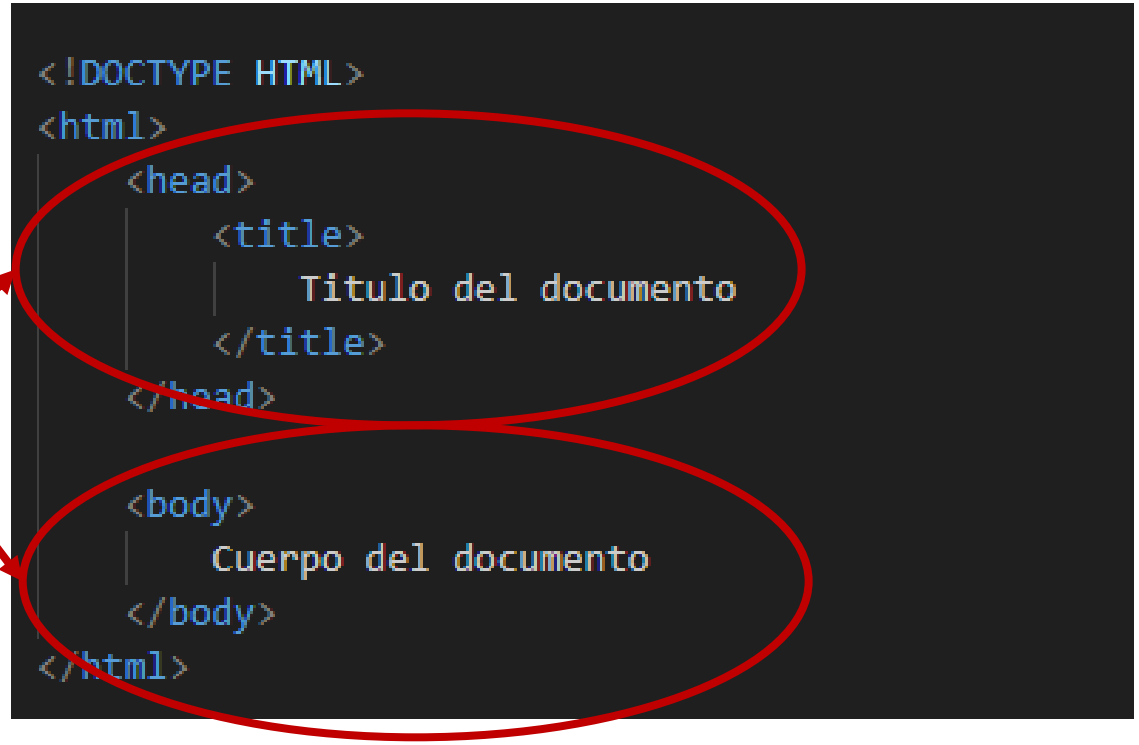
- El DOM nos permite hacer cualquier cosa con sus elementos y contenidos, pero lo primero que tenemos que hacer es llegar al objeto correspondiente del DOM.

- Todas las operaciones en el DOM comienzan con el objeto **document**. Este es el principal “punto de entrada” al DOM.

```
<!DOCTYPE HTML>
<html>
    <head>
        <title>
            Titulo del documento
        </title>
    </head>

    <body>
        Cuerpo del documento
    </body>
</html>
```

Recorriendo el DOM

Childnodes



Un childNode son hijos directos, es decir sus descendientes inmediatos. Por ejemplo, <head> y <body> son hijos del elemento <html>.

Recorriendo el DOM

```html
<!DOCTYPE HTML>
<html>
    <head>
        <title>
            Titulo del documento
        </title>
    </head>

    <body>
        Cuerpo del documento
    </body>
</html>
```

```javascript
for (let i = 0; i < document.body.childNodes.length; i++) {
    console.log( document.body.childNodes[i] );
}
```

DOM

Selectores

Los "Query Selectors" o "Selectores de consulta" en JavaScript son métodos que te permiten seleccionar elementos HTML en el documento utilizando una sintaxis similar a la de los selectores de CSS (hojas de estilo en cascada).

getElementById()	Selecciona un elemento por su atributo "id".
getElementsByClassName()	Selecciona elementos por su atributo "class".
getElementsByTagName()	Selecciona elementos por su etiqueta.
querySelector()	Selecciona el primer elemento que coincida con el selector especificado.
querySelectorAll()	Selecciona todos los elementos que coincidan con el selector especificado y devuelve una lista (NodeList) de elementos.

Eventos

Un evento es una señal de que algo ocurrió. Todos los nodos del DOM generan dichas señales (pero los eventos no están limitados sólo al DOM).

Eventos del mouse	
Click	Cuando el mouse hace click sobre un elemento (los dispositivos touch lo generan con un toque).
Contextmenu	Cuando el mouse hace click derecho sobre un elemento.
Mouseove	Cuando el cursor del mouse ingresa/abandona un elemento.
Mousedown	Cuando el botón del mouse es presionado/soltado sobre un elemento.
Mousemove	Cuando el mouse se mueve.

Eventos

Un evento es una señal de que algo ocurrió. Todos los nodos del DOM generan dichas señales (pero los eventos no están limitados sólo al DOM).

Eventos del teclado	
Keydown/Keyup	Cuando se presiona/suelta una tecla.

Eventos del formulario	
submit	Cuando el visitante envía un <form>.
focus	Cuando el visitante se centra sobre un elemento, por ejemplo un <input>.

Registrar un evento

```
<input
    type="button"
    value="Haz click"
    onclick="evento()">
<script>
    const evento = function() {
        alert("Hola");
    }
</script>
```

A través del atributo de la etiqueta se registra el tipo de evento deseado y se le asocia una función ya existente.

```
<input
    type="button"
    value="Haz click"
    id="miBoton">

<script>
    const evento = document.getElementById("miBoton")

    evento.onclick = () => {
        alert("Hola Mundo");
    };
</script>
```

A través del atributo id de la etiqueta se obtiene la referencia y se le asocia el evento deseado y una función.

Ejercicios

1. Acceder a todas las etiquetas de enlaces (a).
2. Acceder a todas las etiquetas a través de su clase.
3. Acceder a todas las etiquetas de entrada (input).
4. Agregar un nuevo enlace (a)
5. Agregar una nueva caja de texto (input)
6. Cambiar el titulo de la pagina.
7. Agregar un valor a la caja de texto para el nombre.
8. Eliminar un elemento del documento.
9. Cambiar el color del fondo de la pagina.
10. Agregar una función para el botón del formulario para que recoja los valores del usuario y los muestre en consola.

11

Módulo 11

Soicitudes de red

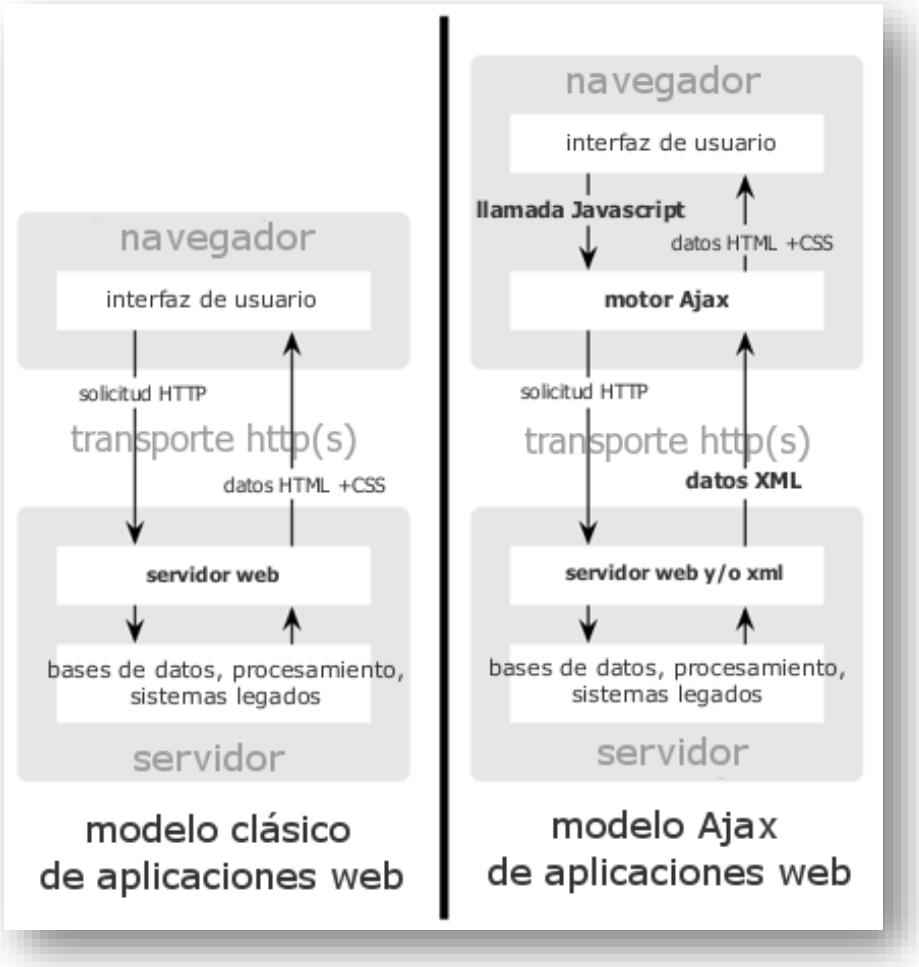
Destructuring

JavaScript puede enviar peticiones de red al servidor y cargar nueva información siempre que se necesite. Por ejemplo, se puede utilizar una petición de red para:

- Crear una orden.
- Cargar información de usuario.
- Recibir las últimas actualizaciones desde un servidor.

Solicitudes de red
AJAX

AJAX permite que una página web que ya ha sido cargada solicite nueva información al servidor.



Solicitudes de red

AJAX

AJAX permite que una página web que ya ha sido cargada solicite nueva información al servidor.

```
const xhr = new XMLHttpRequest();
const url = 'https://api.example.com/data';

xhr.open('GET', url, true);

xhr.onreadystatechange = function() {
  if (xhr.status === 200) {
    const response = JSON.parse(xhr.responseText);
    // Hacer algo con la respuesta recibida
  } else if (xhr.status !== 200) {
    // Manejar errores de la solicitud
  }
};

xhr.send();
```

Solicitudes de red
Promise

Una Promise (promesa) en JavaScript es un objeto que representa la eventual finalización (éxito o fracaso) de una operación asíncrona y la entrega de su resultado.

```
function fetchData() {
    return new Promise((resolve, reject) => {
      // Simulando una operación asincrónica
      setTimeout(() => {
        const data = 'Datos obtenidos de forma exitosa';
        if (data) {
            // La promesa se cumple
            resolve(data);
        } else {
            // La promesa se rechaza
            reject('Error al obtener los datos');
        }
      }, 2000); // Espera de 2 segundos
    });
}
```

Solicitudes de red
Promise

Estados de una promise:

1. Pendiente (pending): Estado inicial. Se crea una promesa y la operación está en progreso.

2. Cumplida (fulfilled): La promesa se resuelve correctamente y se obtiene el resultado esperado.

3. Rechazada (rejected): La promesa falla y se obtiene un motivo de error.

Solicitudes de red
Promise

Las promesas tienen dos partes principales: el productor (también conocido como "executor") y el consumidor. El productor es la función que realiza la operación asíncrona y eventualmente resuelve o rechaza la promesa.

```
// Crear una promesa
let miPromesa = new Promise((resolve, reject) => {
    // Simular una operación asíncrona
    setTimeout(() => {
        // Cambiar a false para simular un fallo
        let exito = true;

        if (exito) {
            resolve("¡Operación completada con éxito!");
        } else {
            reject("¡La operación ha fallado!");
        }

    }, 2000); // Simular un retardo de 2 segundos
});
```

El consumidor utiliza los métodos then y catch para manejar la resolución o el rechazo de la promesa.

```
// Consumir la promesa
miPromesa
  .then((resultado) => {
    // Se ejecuta si la promesa se resuelve
    console.log(resultado);
  })
  .catch((error) => {
    // Se ejecuta si la promesa es rechazada
    console.error(error);
  });
```

Solicitudes de red
Promise

Solicitudes de red
Promise

El método fetch() no es soportado por navegadores antiguos pero es perfectamente soportado por los navegadores actuales y modernos.

```
let promise = fetch(url, [options])
```

- **url:** representa la dirección URL a la que deseamos acceder.
- **options:** representa los parámetros opcionales, como puede ser un método o los encabezados de nuestra petición, etc.

Solicitudes de red

Promise

```
fetch('https://api.example.com/data')
  .then(response => {
    if (!response.ok) {
      throw new Error('Error en la solicitud');
    }
    return response.json();
})
  .then(data => {
    // Hacer algo con los datos obtenidos
})
  .catch(error => {
    // Manejar el error de la promesa
});
```

Solicitudes de red
Comunicación asíncrona

La llamada telefónica es comunicación sincrónica, porque el mensaje se recibe al mismo tiempo que se manda.

Un mensaje de texto es comunicación asincrónica, pues es una acción que se ejecuta en un momento, pero solo finaliza cuando la otra parte lee el texto.

Asíncrona en javascript

Lo que hacemos en este lenguaje de programación es programar acciones que se ejecutarán en caso de que otra acción suceda. Sin embargo, no podemos controlar cuándo, ni quiera si sucederá la acción.

¿Para qué sirve?

La programación orientada a eventos es aquella que prepara la ejecución de código en función de los eventos que pueden ocurrir. Es decir, preparamos el código por si un evento sucede.

Async/Await

Async y await en JavaScript son dos palabras clave que nos permiten transformar un código asíncrono para que parezca ser síncrono.

```
async function fetchData() {
    try {
      const response = await fetch(
        'https://api.example.com/data');

      if (!response.ok) {
        throw new Error('Error en la solicitud');
      }
      const data = await response.json();
      // Hacer algo con los datos obtenidos
      return data;
    } catch (error) {
      // Manejar el error
      console.error(error);
      throw error;
    }
}
```

Async/Await

- La palabra clave async se utiliza en una función para envolver el contenido de la función en una promesa.
- La palabra clave await en JavaScript nos permite definir una sección de la función a la cual el resto del código debe esperar.

```
async function fetchData() {
    try {
      const response = await fetch(
        'https://api.example.com/data');

      if (!response.ok) {
        throw new Error('Error en la solicitud');
      }
      const data = await response.json();
      // Hacer algo con los datos obtenidos
      return data;
    } catch (error) {
      // Manejar el error
      console.error(error);
      throw error;
    }
}
```

Solicitudes de red
Comunicación asíncrona

Async/Await

Mandando a llamar la función fetchData con async/await

```
// Llamada a la función fetchData utilizando
//then y catch
fetchData()
.then(data => {
  // Manejar los datos obtenidos
})
.catch(error => {
  // Manejar el error
});
```