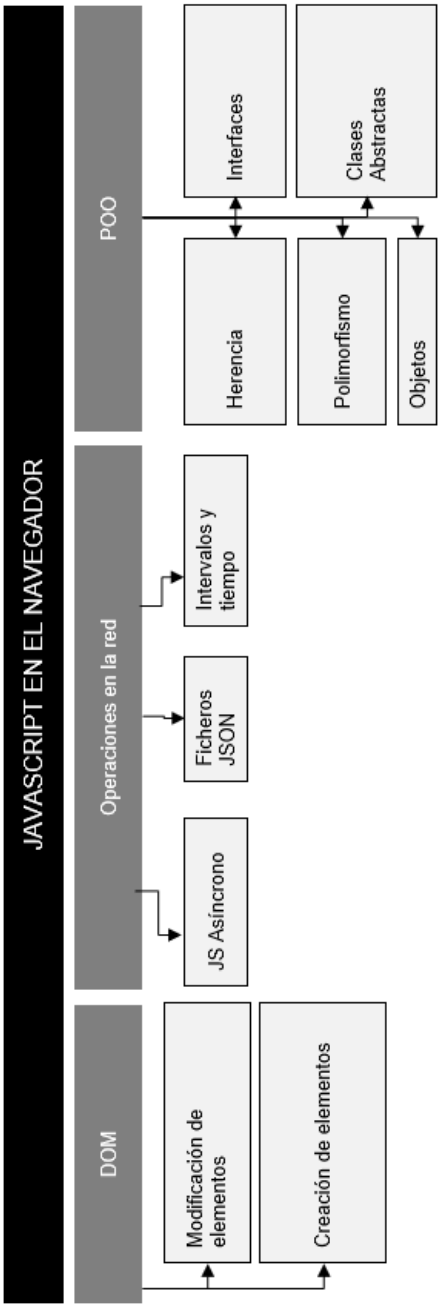
Desarrollo Web en Entorno Cliente

# JavaScript en el navegador

# Índice

# Esquema


JAVASCRIPT EN EL NAVEGADOR
DOM
Modificación de elementos
Creación de elementos
Operaciones en la red
JS Asíncrono
Ficheros JSON
Intervalos y tiempo
POO
Herencia
Polimorfismo
Objetos
Interfaces
Clases Abstractas

# Material de estudio

## 2.1. Introducción y objetivos

Un documento HTML no deja de ser un conjunto de caracteres que por sí mismos no podrían mostrar ningún sitio web. Es el navegador al interpretar dicho documento quien realmente muestra un sitio. Para ello crea el DOM y una vez que el navegador lo ha creado, es posible alterarlo mediante JavaScript, alterando así, la visualización de la página.

Durante el desarrollo de este tema se deben conseguir los siguientes objetivos:

- Conocer los fundamentos principales de JavaScript.
- Aprender a codificar correctamente, sabiendo qué uso podemos dar a las variables y estructuras de control.
- Aprender qué es el DOM y como se puede manejar.
- Conocer como implementar la POO en JS.
- Manejar correctamente las funciones asíncronas.

## 2.2. Modelo de Objetos del Documento (DOM)

El DOM es una representación en memoria del documento HTML que muestra un navegador. Cada uno de los elementos HTML que conforman el documento, se convierte en objetos en memoria con sus propiedades y métodos a los que es posible acceder mediante una API. Todo cambio que se realice en el DOM, cambiará la visualización del sitio web en el navegador.

```
<html>
    <body>
      <h1 id="saludo">Hola Mundo</h1>
    </body>
  </html>

  const elementoSaludo = document.getElementById('saludo');
  elementoSaludo.innerHTML = "Adiós mundo";
  // Tras ejecutar esto, en el navegador se visualizará el título h1
mostrando: Adiós mundo
```

En el anterior ejemplo, JavaScript está modificando la visualización de la página utilizando para ello el DOM y su API. Al ejecutar JavaScript en un navegador está disponible la variable global document la cual hace referencia al documento. Al documento, por ejemplo, se le puede pedir mediante el **método** getElementById(), que busque un elemento con un id determinado.

El elemento con id *saludo* se guarda en la constante elementoSaludo. Gracias a la propiedad **innerHTML** de un elemento HTML para escribir dentro de él un nuevo contenido: _Adiós mundo_. Una vez realizado este cambio, el navegador mostrará en pantalla la visualización del nuevo DOM, es decir, mostrará un título h1 con el contenido *Adiós mundo* (reemplazando al anterior texto).

**Eventos**

Los eventos son sucesos que le ocurren a elementos HTML, JavaScript se puede suscribirse a dichos eventos y actuar en consecuencia. Por ejemplo, que el usuario pulse sobre un botón dispara el evento onClick y JavaScript al suscribirse a ese evento puede actuar en consecuencia mostrando una alerta.

Un evento por tanto es una manera de comunicación: un elemento dispara un evento y hay **manejadores** que lo tratan. Un manejador es una función que se ejecuta en el momento en el que se lanza un evento al que está suscrito.

La manera recomendada de utilizar eventos en JavaScript es a través del método addEventListener:

```
<button id="button"></button>
const button = document.getElementById('button');
button.addEventListener('click', function(event) {
  console.log("Botón pulsado!");
});
```

En el anterior ejemplo, el método addEventListener está escuchando el evento click. El elemento button al ser un elemento HTML podrá disparar este evento en el momento en el que el usuario hace click. Cuando esto ocurra, se ejecutará el manejador asociado al evento click que es la función que muestra por consola el mensaje «Botón pulsado!».

https://developer.mozilla.org/es/docs/Web/Events esta es la enorme lista de eventos DOM que pueden disparar elementos. Estos son algunos de los más utilizados:

- Eventos del ratón: mousedown, mouseenter, mouseleave, mousemove, mouseout, mouseover, mouseup, dblclick, click.
- Eventos táctiles: touchstart, touchmove, touchend, touchcancel.
- Eventos del teclado: keydown, keypress, keyup.
- Eventos de formulario: focus, change, submit.
- Eventos de la ventana: scroll, resize, load.

**Document ready**

La creación del DOM no es un proceso inmediato: el navegador para mostrar una página tiene que leer el documento HTML y convertirlo a un objeto en memoria. Antes de tener el DOM listo, el navegador empezará a ejecutar JavaScript, es por ello por lo que **el código JavaScript que vaya a interactuar con el DOM tiene que esperar**

**a que el DOM se haya generado**, de otra manera no podrá interactuar con él porque literalmente, no hay nada con lo que interactuar.

Para ello lo más sencillo es añadir un manejador al evento DOMContentLoaded que dispara el documento cuando el DOM ya está listo:

```
document.addEventListener('DOMContentLoaded', function(event) {
  // Código JavaScript que manipula el DOM
})
```

**Selectores y métodos de acceso**

Un documento HTML no es más que un árbol de nodos: Hay un nodo padre document del que cuelgan el resto de los elementos HTML, estos elementos HTML son **nodos** del árbol y si un elemento HTML tiene texto dentro, es un nodo de tipo texto.

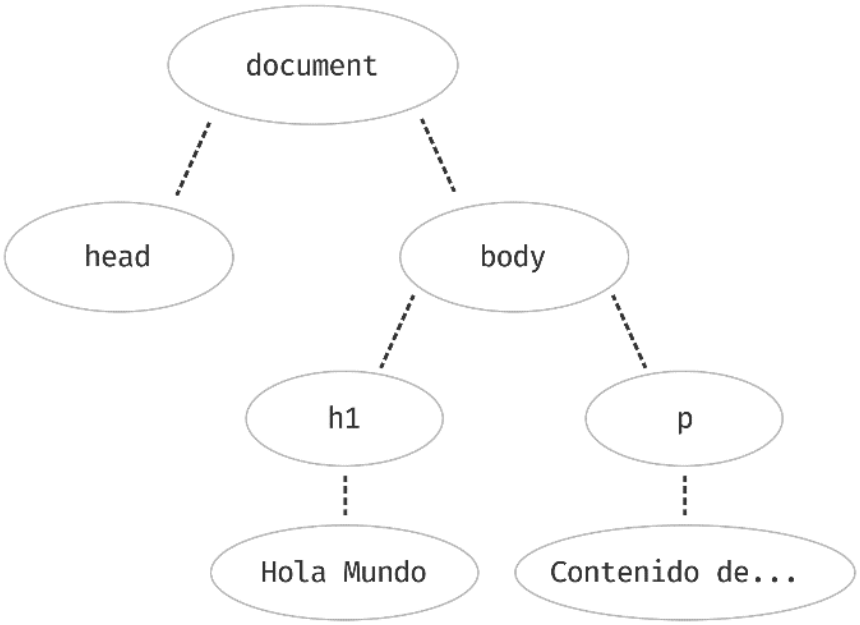



Figura 1. Árbol de nodos DOM. Fuente: contenidos MFSD.

```
<html>
```

```
    <head>
    </head>
    <body>
      <h1>Hola Mundo</h1>
      <p>Contenido de un párrafo</p>
    </body>
  </html>
```

Conocer esta estructura de árbol es importante para recorrer el DOM en busca de elementos: cada nodo tendrá únicamente un padre (excepto document) y un padre podrá tener uno o varios hijos. Tanto desde document como desde cualquier nodo se puede recorrer el DOM. Los métodos que proporciona la API para seleccionar elementos son los siguientes:

- getElementById(id): Devuelve el elemento con un id especifico.
- getElementsByName(name): Devuelve los elementos con un valor del atributo name específico.
- getElementsByTagName(tagname): Devuelve los elementos con un nombre de tag específico.
- getElementsByClassName(classname): Devuelve los elementos con un nombre de clase específico.
- querySelector(selector): Devuelve un único elemento que corresponda con el selector. El formato del selector es el mismo que el de un selector CSS.
- querySelectorAll(selector): Devuelve un array con los elementos que correspondan con con el selector. El formato del selector es el mismo que el de un selector CSS.

```
<p class="parrafo" id="parrafo1">
    <span class="estilado">
      Nueva
    </span>
    Camiseta blanca XL
    <span class="precio">
```

```
      9.3€
    </span>
  </p>
  <p class="parrafo" id="parrafo2">
    <span class="estilado">
      Envío gratis
    </span>
    Camiseta negra XL
    <span class="precio">
      9.3€
    </span>
  </p>

  document.getElementsByClassName('precio'); // Devuelve los dos precios
  document.getElementById('parrafo1'); // Devuelve el primer párrafo
  document.querySelector('.estilado'); // Devuelve el primer elemento span
con contenido "Nueva"
  document.querySelectorAll('.estilado'); // Devuelve los dos elementos con
etiqueta span con contenido "Nueva" y "Envío gratis"
  document.getElementsById('parrafo3'); // Devuelve null
```

# 2.3. Manipulación del DOM

Accediendo al DOM se puede manipular su contenido, el navegador de primeras mostrará lo que le llegue desde el documento HTML (primera carga del DOM), pero mediante JavaScript se puede modificar ese DOM creado y la visualización en el navegador reflejará esos cambios.

**Manipulando el documento**

- createElement(etiqueta): Crea un elemento HTML, como atributo se puede seleccionar la etiqueta que tendrá ese elemento.
- appendChild(node): Permite añadir un hijo a un elemento.
- removeChild(child): Elimina el nodo hijo que se indica con child.

**Manipulando elementos**

- Manipular clases:
  - classList.add('clase'): Añade la clase que se pasa por parámetro.
  - classList.remove('clase'): Quita la clase que se pasa por parámetro.
  - classList.toggle('clase'): Si tiene la clase que se le pasa por parámetro la quita y si no la tiene, la pone.

- Manipular contenido:
  - innerHTML. Mediante esta propiedad se puede añadir contenido HTML a cualquier elemento.
  - innerText. Utilizando esta propiedad se puede añadir texto plano a un elemento.

- Manipular cualquier atributo:
  - getAttribute(nombre): Método que obtiene el valor del atributo cuyo nombre se pasa por argumento.
  - setAttribute(nombre): Método que escribe el valor del atributo para el nombre que se pasa por argumento.

En el siguiente extracto de código se realizan operaciones habituales de manipulación del DOM como crear elementos, añadir una clase y atributo:

```
// Crea un párrafo
const paragraph = document.createElement('p');

// Se le añade contenido
paragraph.innerText = 'soy un párrafo';

// Se le añade una clase
paragraph.classList.add('tipo1');
```

```
// Se añade al body
document.appendChild(paragraph);

// Crea una imagen
const image = document.createElement('img');

// Se añade el atributo src con la ruta de la imagen
image.setAttribute('src', 'image.jpg');

// Se añade al body
document.appendChild(image);
```

El anterior código JavaScript, genera el siguiente resultado en el DOM:

```
<body>
  <p class="tipo1">soy un párrafo</p>
  <img src="image.jpg">
</body>
```

**Intervalos y timeout**

setTimeout() y setInterval() son dos métodos incorporados de JavaScript que permiten ejecutar algo pasado un determinado tiempo.

setTimeout() ejecutará la función que se le pase por argumento pasado un tiempo determinado, mientras que setInterval() ejecutará la función cada un determinado tiempo de manera infinita. clearTimeout() y clearInterval() permiten borrar estos temporizadores.

```
// Recibe dos argumentos: la función a ejecutar y el tiempo (en milisegundos) que va a esperar a ejecutarla
const saludar = function() {
  console.log("Hola");
}

setTimeout(saludar, 1000);
```

```
// Pasado 1 segundo mostrará por consola: Hola

setInterval(function() {
  console.log("hola");
}, 1000);
// Cada segundo mostrará por consola: Hola

// Tanto setTimeout y setInterval al crearse devuelven un id
// Ese id es necesario para poder "limpiar" el intervalo
const intervalId = setInterval(function() {
  console.log("hola");
}, 1000);

setTimeout(function() {
  clearInterval(intervalId);
}, 3000);

// Segundo 1: hola
// Segundo 2: hola
// Segundo 3: no se ve nada por consola porque se ha limpiado el intervalo
```

## 2.4. JavaScript orientado a objetos

Para definir una clase en JavaScript se utiliza la palabra reservada class seguida del nombre de la clase. Los nombres de clase como los de variables son case-sensitive pero con convención, se empiezan con mayúscula.

```
class Coche {
}
```

Dentro de las llaves {} irá todo el código de la clase y de esa manera queda encapsulado: todo lo que tiene que ver con la naturaleza de la clase, está ahí dentro.

Las clases tienen un método especial llamado constructor que se ejecuta automáticamente al instanciar una clase. Para instanciar una clase se utiliza la palabra reservada new seguida del nombre de la clase y de unos paréntesis ().

```
class Coche {
  constructor() {
    console.log("Se ha creado un coche");
  }
}

// Se instancia la clase coche y se deja el objeto en la variable coche
const coche = new Coche();
// Mostrará por consola:
// Se ha creado un coche
```

**Propiedades**

La palabra reservada this utilizada dentro de una clase se refiere a la instancia de la misma clase, es decir, esa parte de memoria donde se guarda toda la información sobre la instancia.

this en JavaScript es utilizado para guardar propiedades de una clase. Una propiedad de una clase se asemeja a una variable: un valor accesible mediante un nombre de propiedad que se guarda en la instancia. Por ejemplo, para una clase Coche una propiedad sería el color.

Es habitual definir parte de estas propiedades en el constructor, pero se pueden modificar o acceder a ellas en cualquier momento, tanto dentro como fuera de la clase utilizando la misma notación que para acceder a claves de objetos (con . o con [clave]).

```
class Coche {
  constructor(color, marca) {
    this.color = color;
    this.marca = marca;
```

```
  }
}

const seatRojo = new Coche('rojo', 'seat');
console.log(seatRojo.color); // rojo

// Sobreescribiendo propiedad
seatRojo.color = 'verde';
console.log(seatRojo.color); // verde

// Añadiendo nueva propiedad
console.log(seatRojo.nPuertas); // undefined
seatRojo.nPuertas = 4;
console.log(seatRojo.nPuertas); // 4
```

**Métodos**

Los métodos de una clase se asemejan a funciones, es decir, bloques de código que ejecutan un determinado número de sentencias. Al igual que el concepto de clase, un método también se asemeja al modelo mental de qué cosas puede hacer una clase.

```
class Coche {
  constructor() {
    this.encendido = false;
  }

  arrancar() {
    this.encendido = true;
  }

  detener() {
    this.encendido = false;
  }
}
```

En el ejemplo anterior, se han añadido los métodos arrancar y detener, gracias a ellos se puede manejar la propiedad encendida. De esta manera la clase *Coche* es más sencilla de utilizar y recrea un poco más lo que un coche hace en la vida real.

Los métodos pueden ser tan complicados como se necesite, no dejan de ser funciones dentro de la clase y que pueden acceder al espacio en memoria que ocupa la instancia a través de la palabra this.

Un ejemplo un poco más completo:

```
class Coche {
    constructor(marca, color, bastidor) {
      this.marca = marca;
      this.color = color;
      this.bastidor = bastidor;
      this.encendido = false;
      this.velocidad = 0;
    }

    arrancar() {
      if (!this.encendido) {
        this.encendido = true;
      }
    }

    apagar() {
      if (this.encendido) {
        this.encendido = false;
      }
    }

    acelerar(velocidad) {
      this.velocidad += velocidad;
    }

    frenar(velocidad) {
      this.velocidad -= velocidad;
```

```
    if (this.velocidad < 0) {
      // No puede ir a velocidad negativa
      this.velodad = 0;
    }
  }
  itv() {
    if (this.velocidad > 0) {
      console.log("detenga el coche");
    }else if (this.encendido) {
      console.log("apague el coche");
    } else {
      console.log('Marca: ', this.marca);
      console.log('Color: ', this.color);
      console.log('Bastidor: ', this.bastidor);
    }
  }
}
```

**Encapsulación**

La encapsulación es un mecanismo de la programación orientada a objetos. En una clase puede haber métodos y/o propiedades de una instancia que el programador no quiere que puedan utilizarse directamente, haciendo así restringido su acceso.

```
class Coche {
  constructor(bastidor) {
    this.bastidor = bastidor;
  }
}

const coche = new Coche('123');
coche.bastidor = 456;
console.log(coche.bastidor); // 456
```

En el ejemplo anterior, el número de bastidor del coche que se define en el constructor se puede cambiar de una manera muy sencilla. El número de bastidor en el mundo real es algo relacionado intrínsecamente con el coche y no se puede

cambiar. Este comportamiento se puede codificar con una propiedad privada, para ello, simplemente se define dentro del cuerpo de la clase el nombre de la propiedad con un # como prefijo:

```
class Coche {
  #bastidor;
  constructor(bastidor) {
    this.#bastidor = bastidor;
  }

  leerBastidor() {
    return this.#bastidor;
  }
}

const coche = new Coche('123');
console.log(coche.bastidor); // undefined
console.log(coche.leerBastidor()); // 123
```

Un método también puede ser privado, simplemente hay que añadirle un # como prefijo:

```
class Coche {
  constructor() {
    #procesosInternos();
  }

  #procesosInternos() {
    ...
  }
}
```

En el anterior ejemplo, hay procesos internos que ocurren al crearse un coche que solo se deben ejecutar en el constructor, por ello, se crean dentro de un método privado para que no puedan llamarse desde fuera.

Relacionado con la encapsulación, JavaScript dispone de getters y setters que es una manera de hacer propiedades virtuales, es decir, desde fuera parece una propiedad, pero dentro de la clase no lo es:

```
class Persona {
  #nombre
  #apellidos

  constructor(nombre, apellidos) {
    this.#nombre = nombre;
    this.#apellidos = apellidos;
  }

  get nombreCompleto() {
    return this.#nombre + ' ' + this.#apellidos;
  }

  set nombreCompleto(input) {
    const arrayInput = input.split(' '); // Devuelve un array con cada elemento que esté separado por el carácter ' ' (espacio)
    this.#nombre = arrayInput[0];
    this.#apellidos = arrayInput[1];
  }
}

const persona = new Persona('Alonso', 'Paz');

console.log(persona.nombreCompleto); // Alonso Paz
persona.nombreCompleto = "Martin Sola";
console.log(persona.nombreCompleto); // Martin Sola
```

**Herencia**

Otro concepto relacionado con la programación orientada a objetos es la herencia, un mecanismo mediante el cual una clase hija hereda código de una clase padre.

Siguiendo con la codificación del modelo mental del mundo real y pensando en una buena estructura de código y reutilización de este, se puede pensar en clases que de

alguna manera sean parecidas y se pueda encontrar un padre de estas. Poniendo un ejemplo en la vida real: un sedán, una furgoneta y un descapotable con conceptos diferentes, pero todos heredan del concepto coche: Todos tienen 4 ruedas, un volante o necesitan un conductor.

Es por lo tanto que se puede codificar una clase padre que contenga código que las clases hijas puedan utilizar y de esta manera las clases hijas también tienen algo de la clase padre, es decir, son en parte también padre.

```
class Coche {
  constructor(bastidor) {
    this.bastidor = bastidor;
    this.encendido = false;
  }

  arrancar() {
    this.encendido = true;
  }

  apagar() {
    this.encendido = false;
  }
}

class Sedan extends Coche {
  constructor(bastidor) {
    super(bastidor);
  }
}

class Descapotable extends Coche {
  constructor(bastidor) {
    super(bastidor);
    this.capotaAbierta = false;
  }

  capotar() {
```

```
    this.capotaAbierta = false;
  }

  descapotar() {
    this.capotaAbierta = true;
  }
}

const sedan = new Sedan('123');
console.log(sedan.bastidor); // 123
sedan.arrancar();
sedan.apagar();
console.log(sedan.encendido); // false

const descapotable = new Descapotable('456');
descapotable.descapotar();
console.log(descapotable.capotaAbierta); // true
```

La codificación de clases hijas en JavaScript se hace añadiendo la palabra reservada extend seguida del nombre de la clase a la que extiende en la definición de la clase.

Además, es necesario llamar al constructor del padre a través del constructor del hijo, de no ser así, el padre no se construye y la clase no tendrá contexto de ejecución (no tendrá this). Para llamar al padre a través del hijo se utiliza la palabra super y para construir al padre, simplemente ejecutarlo: super ().

```
class Coche {
  constructor(marca, color, bastidor) {
    this.marca = marca;
    this.color = color;
    this.bastidor = bastidor;
    this.encendido = false;
    this.velocidad = 0;
  }

  arrancar() {
    if (!this.encendido) {
```

```
      this.encendido = true;
    }
  }

  apagar() {
    if (this.encendido) {
      this.encendido = false;
    }
  }

  acelerar(velocidad) {
    this.velocidad += velocidad;
  }

  frenar(velocidad) {
    this.velocidad -= velocidad;
    if (this.velocidad < 0) {
      // No puede ir a velocidad negativa
      this.velodad = 0;
    }
  }

  itv() {
    if (this.velocidad > 0) {
      console.log("detenga el coche");
    }else if (this.encendido) {
      console.log("apague el coche");
    } else {
      console.log('Marca: ', this.marca);
      console.log('Color: ', this.color);
      console.log('Bastidor: ', this.bastidor);
    }
  }
}
```

# 2.5. JavaScript asíncrono

El término asíncrono se refiere a que ocurre más de una cosa al mismo tiempo. Pensando en términos de programación: la existencia de varios hilos/flujos de ejecución ocurriendo al mismo tiempo.

JavaScript es un lenguaje de programación pensado para trabajar en asíncrono de una manera sencilla.

**Promesas**

Una promesa lanza código de **manera asíncrona**, es decir **sale del flujo de ejecución para ejecutarse en otro flujo**. Al igual que el flujo principal, una promesa puede funcionar bien o puede tener algún error en la ejecución.

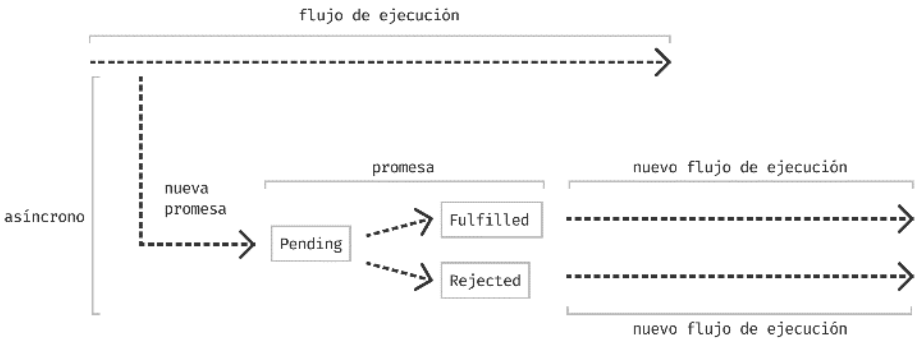



Figura 2. Esquema de una promesa. Fuente: contenidos MFSD.

Una promesa empieza en el estado pending, es decir el proceso todavía no se ha completado y hay que esperar para que se complete. Si dicho proceso se completa correctamente la promesa pasará al estado fulfilled pero si ocurre algún error que hace que el proceso no se haya podido completar, la promesa acabará con un estado rejected.

Por ejemplo, una petición a un servidor puede ser codificada como una promesa, donde al lanzar la petición la promesa se queda en pending, si los datos llegan correctamente la promesa pasará al estado fulfilled pero si hay algún error en el

proceso, como que se ha cortado la conexión o que el servidor no responde la promesa acabará como rejected.

```
  const promesa = new Promise(function(resolve, rejected) {
    // proceso

    // si el proceso va bien
    resolve()

    // si el proceso va mal
    rejected()
});
```

En JavaScript una promesa se codifica utilizando la clase Promise. En el constructor de dicha clase se añade como argumento una función que recibe dos parámetros: resolve y rejected. Ambos parámetros son funciones que se deben ejecutar para que la función acabe en fulfilled o en rejected.

La promesa entonces al crearse mediante su constructor empezará a ejecutar en otro flujo de ejecución a la vez que el flujo de ejecución principal (esto se conoce como ejecución asíncrona) el código que haya dentro de la función. Si en ese código va todo bien y se realiza la tarea correctamente, la promesa para comunicarse con el exterior cambia su estado a fulfilled utilizando para ello la función resolve(), en caso contrario, basta con ejecutar rejected() para comunicar que ha habido un error y la promesa pasará al estado rejected.

**Then/catch/finally**

Las promesas son más útiles si tras ellas se sigue ejecutando otro flujo, que actúe en consecuencia. Para ello, las promesas incorporan los métodos then(), catch() y finally(). Estos métodos ejecutan la función que le llega por argumento:

- then() si la promesa ha funcionado bien.
- catch() si la promesa no ha funcionado correctamente.
- finally() se ejecutará de todos modos, haya ido bien o mal la promesa.

```
  const promesa = new Promise(function(resolve, rejected) {
    resolve("la promesa ha ido bien");
  });

  promesa.then(function(result){
    console.log(result);
  });

  // Mostrará por consola:
  // la promesa ha ido bien

  // estos métodos devuelven la promesa en sí de nuevo, por lo que pueden
concatenarse
  promesa.then(function(result){
    console.log(result);
  }).finally(function(result){
    console.log("la promesa se ejecutó")
  });

  // Mostrará por consola
  // la promesa ha ido bien
  // la promesa se ejecutó
```

**async/await**

Utilizar la palabra reservada async convierte una función en una función asíncrona: saldrá del flujo de ejecución y se ejecutará en otro flujo.

```
  async function funcionAsincrona() {
  }

  funcionAsincrona(); // Promise
```

De esta manera, la ejecución de la función por sí misma se convierte en una promesa. Tanto es así que devuelve una promesa que dependiendo de las necesidades puede ser tratada o no (es decir, esperar si se ha ejecutado correctamente o no). Si se quiere

tratar la función se devuelve directamente la resolución o error de una promesa y de esta manera se puede enganchar a un .then() o .catch():

```
async function funcionAsincrona() {
  console.log('Función asíncrona');
  return Promise.resolve("ok");
}

funcionAsincrona().then(function(result){
  console.log(result);
})

// Mostrará por consola:
// Functión asíncrona
// ok
```

**Peticiones con fetch**

Uno de los casos de uso de funciones asíncronas por excelencia en una aplicación web es la consulta de datos a un servidor: el código del flujo principal ejecuta una consulta a un servidor, la petición se ejecuta en otro flujo y puede pasar que funcione todo bien y la promesa se resuelva correctamente o que, por ejemplo, el servidor no responsa por lo que la promesa falle.

Para realizar peticiones HTTP JavaScript ofrece el método fetch que realiza la petición y devuelve simplemente una promesa para tratarla en el código.

```
fetch(recurso).then(function(response) {
  // La petición se ha realizado
  // La respuesta está disponible en response
});
```

Esa respuesta que da fetch no es simplemente el contenido, es una instancia de la clase Response de JavaScript, la cual provee propiedades y métodos para tratar la respuesta, algunas de ellas son:

- Propiedades:
    - status: el código de estado de la petición HTTP
    - ok: Booleano cuyo valor es true si el código de estado de la petición es 2XX.
- Métodos:
    - text(): Devuelve una promesa que una vez resuelta devuelve la respuesta en formato texto.
    - json(): Devuelve una promesa que una vez resuelta devuelve la respuesta en formato json.

```
// Código típico de petición fetch a servidor que devuelve JSON
fetch('https://jsonplaceholder.typicode.com/posts')
  .then(function(response) {
    return response.json()
  })
  .then(function(posts) {
    // En el parámetro post está un array de posts
    // listo para usar
  });

  // Utilizando arrow functions
  fetch('https://jsonplaceholder.typicode.com/posts')
    .then(response => response.json())
    .then(posts => {
      // En el parámetro post está un array de posts
      // listo para usar
    });
```

El servicio online JSONPlaceholder (https://jsonplaceholder.typicode.com/) ofrece una API REST a la que realizar peticiones HTTP con el fin de aprender o probar desarrollos. Es muy similar a como sería un servidor real y devuelve información suficiente para poder probar todo tipo de llamadas.

# 2.6. Formato de intercambio de datos JSON

JSON es el formato por excelencia en JavaScript para realizar intercambios de datos como en una petición HTTP. Su nombre viene de JavaScript Object Notation y se parece muchísimo a como se escriben y se representan en pantalla objetos en JavaScript. JSON es casi un objeto pasado a String.

```
const persona = {
  nombre: 'Mercer',
  apellido: 'Roussel',
  direccion: 'Cruce Casa de Postas, 33'
};

console.log(typeof persona); // object

const personaJSON = '{"nombre":"Mercer","apellido":"Roussel","direccion":"Cruce Casa de Postas, 33"}';
console.log(typeof personaJSON); // string
```

De esta manera, se pueden pasar datos estructurados mediante un string que es más fácil de enviar y recibir. Un flujo típico de petición podría ser el siguiente:

- El cliente realiza una petición.
- El servidor convierte el objeto en memoria a un string que contiene un JSON.
- El servidor envía un string.
- El cliente recibe un string.
- El cliente convierte ese string en un objeto con el que ya puede trabajar.

Para trabajar con JSON, JavaScript provee los siguientes métodos:

- JSON.parse(cadenadetexto): Convierte una cadena de texto en un objeto.
- JSON.stringify(objeto): Convierte un objeto en una cadena de texto utilizando el formato JSON.

## 2.7. Referencias bibliográficas

Azaustre, C. (2021). *Aprendiendo JavaScript: Desde cero hasta ECMAScript 6+.* Independently published.

Gómez, M. R. (2021). *Curso de desarrollo Web. HTML, CSS y JavaScript.* ANAYA MULTIMEDIA.

# A fondo

**Programación orientada a objetos**

Angel Alvarez, M. (2021). *Qué es la programación orientada a objetos* [documentación en línea]. https://desarrolloweb.com/articulos/499.php

JavaScript no soporta todas las ideas que la Programación Orientada a Objetos, si se desea profundizar en el tema, es recomendable estudiarla desde un punto de vista conceptual en lugar para posteriormente aplicarla a un lenguaje determinado.

**Códigos de respuesta HTTP**

Silva, C. (2023). Códigos de estado HTTP: una guía sin tecnicismos [artículo en línea]. https://es.semrush.com/blog/codigos-de-estado-http/

Los códigos de respuesta del protocolo HTTP proveen de mucha información cuando se realiza una llamada. Conocer su naturaleza y al menos sus rangos es algo que todo desarrollador web debe conocer.

**fetch**

MDN Contributors. (2022). *Uso de Fetch* [documentación en línea]. https://developer.mozilla.org/es/docs/Web/API/Fetch_API/Using_Fetch

El método fetch tiene muchas más opciones que las que se han visto en esta documentación, es una pieza fundamental en cualquier proyecto de hoy en día y conviene echar un vistazo a su documentación para ser consciente de lo que es capaz.

# Entrenamientos

**Entrenamiento 1**

- Escribe un programa en JavaScript para crear el juego de “Piedra, papel o tijera”, el programa debe de seguir la siguiente estructura.
  - Explicarle al jugador cómo se juega.
  - Generar la jugada aleatoria de cada jugador (usar una función).
  - Decidir quién ha ganado.
- Desarrollo paso a paso:
  - Explicarle al jugador cómo se juega.
  - Generar la jugada aleatoria del ordenador.
  - Pedir al jugador su jugada mediante una letra (P para piedra, L para papel, T para tijeras):
  - Decidir quién ha ganado (comparación de las jugadas).
  - Mostrar la jugada y el ganador.
- Solución: https://github.com/Anuar-UNIR/DWEC_2024_2025/tree/main/Tema%202/Entrenamiento%203

**Entrenamiento 2**

- Desarrollar un programa que cree un array con 100 números reales aleatorios entre 0.0 y 10.0, utilizando Math.random(), nos calcule la media, mediana, la suma, el máximo, el mínimo y el valor más repetido.
- Desarrollo paso a paso
  - Generar un array de dimensión 100
  - Rellenar con valores aleatorios usando Math.random()
  - Calcular media, mediana, la suma, el máximo, el mínimo y el valor más repetido (se pueden usar funciones independientes o funciones que calculen más de una cosa a la vez)
  - Mostar en la consola la información.

- Solución: https://github.com/Anuar-UNIR/DWEC_2024_2025/tree/main/Tema%202/Entrenamiento%205

**Entrenamiento 3**

- Escribe un programa en JavaScript, HTML y css para crear un párrafo y un botón mediante el uso del DOM en Javascript
- Desarrollo paso a paso:
    - Suscríbete al evento DOMContentLoaded.
    - Crea un elemento párrafo (<p>, modifica su contenido y añádelo un estilo.
    - Crea un elemento botón (<button>) modifica su contenido y añádelo un estilo.
    - Suscríbete al evento click del botón y muestra una alert por pantalla.
    - Añade los dos elementos al document.
- Solución: https://github.com/Anuar-UNIR/DWEC_2024_2025/tree/main/Tema%202/Entrenamiento%207

**Entrenamiento 4**

- Crea un programa un programa en JavaScript, HTML y css que genere un formulario con los campos, nombre, edad, email, teléfono y un checkbox para aceptar los términos.
- Desarrollo paso a paso:
    - Generar una etiqueta form e insertar los elementos input.
    - Definir un estilo básico.
    - Suscríbete al evento DOMContentLoaded y al submit del formulario.
    - Realizar las validaciones que se creen necesarias.
- Solución: https://github.com/Anuar-UNIR/DWEC_2024_2025/tree/main/Tema%202/Entrenamiento%208

**Entrenamiento 5**

- Crea un programa un programa en JavaScript, HTML y css que se conecta a una API REST en línea y muestra los datos en un HTML. Para este ejemplo, utilizaremos la API de prueba pública JSONPlaceholder que proporciona datos simulados. Vamos a obtener una lista de usuarios y mostrarla en una página web.
- Desarrollo paso a paso
    - El HTML tiene que tener una estructura básica con un contenedor div con el id user-list, donde se mostrarán los usuarios obtenidos de la API.
    - Tenemos que utilizar fetch para realizar una solicitud HTTP GET a la API pública de JSONPlaceholder.
    - Crea una función fetchUsers() utiliza async/await para manejar las llamadas asíncronas y procesar los datos una vez que son devueltos.
    - Si los datos se obtienen correctamente, se llama a displayUsers(), que se encarga de crear dinámicamente los elementos HTML con la información de cada usuario y agregarlos al DOM.
- Solución: https://github.com/Anuar-UNIR/DWEC_2024_2025/tree/main/Tema%202/Entrenamiento%209