Máster en Full Stack Developer

Desarrollo Web Avanzado con Vanilla JS

Índice

Tema 11. Conceptos básicos de JavaScript

11.1. Introducción

Un programa no deja de ser una colección de instrucciones que ejecuta un dispositivo como un ordenador o un teléfono móvil una detrás de otra. Por muy complicado que sea un programa, al final se reduce a operaciones simples como sumar números, escribir un valor en memoria o recuperarlo.

Un **lenguaje de programación** es ese idioma en el que se escriben dichas instrucciones: el programador se comunica con el dispositivo utilizando un lenguaje, en este caso de programación. JavaScript es un lenguaje de programación orientado a web, antiguamente se ejecutaba únicamente en navegadores, pero poco a poco ha conseguido ser full stack, es decir, se puede utilizar JavaScript como lenguaje de programación en todas las capas del desarrollo web.

Añadir JavaScript a un documento HTML

Normalmente si un navegador encuentra código JavaScript lo ejecutará nada más encontrarlo (con determinadas técnicas avanzadas este comportamiento se puede alterar). Para que un navegador encuentre código JavaScript este debe estar añadido a un documento HTML.

Las maneras de añadir código JavaScript son muy similares a las maneras de añadir código CSS a un documento HTML:

Código online

Se puede añadir código JavaScript directamente a atributos de elementos HTML, esta manera de añadir código JavaScript no es nada recomendable:

```
<!-- el navegador mostrará un aviso al pulsar sobre el enlace -->
<a href="#" onclick="alert('Hola Mundo')">Pulsar</a>
```

Dentro de una etiqueta script

HTML dispone de la etiqueta script que tiene la posibilidad de uso de contener JavaScript dentro de ella. En cuanto el navegador encuentre esa etiqueta, si encuentra código dentro de ella empezará a ejecutarlo.

Esta etiqueta puede ser añadida dentro de la etiqueta head o dentro de la etiqueta body. Si se añade dentro del head seguro que el código JavaScript es ejecutado antes de que se empiece a mostrar la página en el navegador (lo cual puede ser interesante o no dependiendo de las necesidades de ese código JavaScript).

```
<body>
  <h1>...</h1>
  <script>
    alert("hola");
  </script>
</body>
```

Enlazado desde un fichero .js

Enlazar el código JavaScript desde un fichero externo es la forma recomendada de añadir código JavaScript a un documento HTML. De esta manera se separa el código JavaScript del contenido del documento HTML y puede ser reutilizado en diferentes páginas.

Un archivo JavaScript (al igual que uno HTML o CSS) es simplemente un fichero de texto plano con extensión .js que contiene código JavaScript. Para enlazar un archivo JavaScript se utiliza la etiqueta script:

```
// fichero index.js
alert("Hola Mundo");
```

```
<body>
  <h1>...</h1>
  <script src="index.js"></script>
</body>
```

En el vídeo *¿Cómo añadir código JavaScript a un documento HTML?* se hace un repaso y se enseña de manera práctica estas tres formas.

11.2. Sentencias

Cada una de las instrucciones que forman un programa se llama *sentencia*. Un programa será entonces una lista de sentencias.

Cada sentencia en JavaScript debe ser terminada con punto y coma y por mejorar la legibilidad del código (salvo excepciones) debe escribirse una sentencia por línea.

```
  const a; // Sentencia de declaración de variable a
  const b; // Sentencia de declaraciónd de variable b
  a = 3; // Sentencia de asignación del valor 3 en la variable a
  b = 2; // Sentencia de asignación del valor 2 en la variable b
  alert(a + b); // Sentencia de función alert que mostrará una alerta con el contenido '5'
```

11.3. Funciones de salida

JavaScript por sí mismo puede mostrar datos cuando se ejecuta en un navegador de las siguientes maneras:

console.log

Una de las maneras más utilizadas para mostrar datos directamente desde JavaScript con propósito de desarrollo. Los navegadores en sus herramientas de desarrollo incluyen la consola JavaScript mediante la cual se pueden mostrar datos desde código a través de la función console.log.

```
<body>
  <script>
    /* En la consola del navegador aparecerá el valor 5 */
    console.log(2 + 3);
  </script>
</body>
```

innerHTML

Gracias a esta función que tiene todos los elementos HTML se puede escribir dentro de ellos a través de JavaScript.

```
<body>
    <h1>...</h1>
    <p id="salida"></p>
    <script>
      /*
      Este código selecciona el elemento HTML con id salida
      y dentro le escribe el resultado de 2 + 3
      */
      document.getElementById("salida").innerHTML = 2 + 3;
    </script>
  </body>
```

document.write()

Únicamente recomendado para desarrollo, con esta función JavaScript se puede escribir directamente en el documento, lo que se escriba a través de esta función aparecerá después del contenido que tenga body.

```
  <body>
    <h1>...</h1>
    <script>
      document.write(2 + 3);
    </script>
  </body>
```

window.alert()

Esta función dispara una ventana emergente a través del navegador. Habitualmente los navegadores muestran una alerta mostrando la información y con un botón para que el usuario pueda cerrar dicha alerta.

```
<body>
  <h1>...</h1>
  <script>
    window.alert(2 + 3);
  </script>
</body>
```

Se puede utilizar directamente alert() en lugar de window.alert() ya que en los navegadores el objeto window corresponde con el ámbito global.

11.4. Comentarios

Como cualquier lenguaje de programación, JavaScript incluye la posibilidad de añadir comentarios a su código.

Los comentarios son caracteres que no se ejecutarán y son útiles para documentar el código, explicar funcionalidades a futuros desarrolladores u ocultar al intérprete de código partes del programa mientras se está desarrollando. El código debe ser los suficientemente claro para que se pueda explicar por sí mismo, si esto no es posible se debe recurrir a los comentarios.

JavaScript tiene dos maneras de escribir comentarios: comentarios de **una línea** que empiezan con // o comentarios de **bloque** que empiezan con /* y acaban con */. Siempre que sea posible conviene utilizar los comentarios de línea, a no ser que realmente se esté comentando un bloque grande de código y/o texto que se utilizarán los comentarios de bloque para ello.

```
const b; // Declaración de variable b
a = 3; // Asignación del valor 3 en la variable a

/*
  Este código muestra la suma de dos números
```

```
    que se han declarado antes en una alerta del navegador.
*/
const a = 3;
const b = 2;
alert(a + b);
```

11.5. Tipos de datos

El tipo de dato es un concepto muy común en programación, como su propio nombre indica, son las diferentes clases de datos que JavaScript puede manejar. Cada tipo de dato tiene sus características y operaciones permitidas, por ejemplo, se pueden multiplicar dos datos que sean números, sin embargo, no se pueden multiplicar cadenas de texto.

JavaScript maneja los siguientes tipos de datos:

- Cadenas de texto.
- Números.
- Booleano.
- Array.
- Objetos.
- Indefinido (undefined).

```
const string = "Hola mundo!";
const number = 3;
const boolean = true;
const array = [1, 2, 3];
const object = {
  name: "Gerardo",
  lasname: "Fernández"
};
let indefinido;
```

Gracias a la palabra reservada typeof se puede preguntar a JavaScript por el tipo de dato de una variable.

```
typeof "Hola Mundo!"; // Devuelve 'string'
typeof 3;             // Devuelve 'number`
```

El tipo de datos especial undefined significa que esa variable todavía no alberga nada o lo que albergaba se ha borrado pero la variable sigue definida:

```
let a;
typeof a; // undefined
a = 3;
typeof a; // number
a = undefined;
typeof a // undefined
```

Tipado débil

Comprobar el tipo de variable puede ser útil en determinados casos ya que **JavaScript es un lenguaje de programación de tipado débil**, esto quiere decir que para declarar una variable no hace falta indicar el tipo de dato que va a albergar y en la vida de esa variable el tipo de dato puede cambiar:

```
// Simplemente se declara la variable a, no hace falta indicar qué tipo de dato va a albergar, en otros lenguajes de programación si hace falta
let a;
typeof a; // undefined
a = 3;
typeof a; // number
a = "3";
typeof a; // string
```

Este tipado débil puede acarrear problemas y comportamientos inesperados ya que JavaScript al poder cambiar el tipo de una variable sin consultar con el programador, puede dar lugar a resultados que no son intuitivos:

```
const a = 3;
const b = 2;
console.log(a + b); // 5
typeof a + b; // number

const c = "3";
console.log(c + b); // 32
typeof c + b; // string
/* En este caso JavaScript ha hecho lo posible por sumar un número y una cadena de texto, como eso no es posible lo que ha hecho ha sido transformar en cadena de texto el número y concatenar las 2 cadenas de texto, dando resultado a otra cadena de texto */

console.log(
+ true + true) // 2
/* JavaScript no puede sumar booleanos, ha decidido convertirlos en números de tal manera que
= 0 y true = 1, por lo que 0 + 1 + 1 = 2 */
```

Conocer estos comportamientos de JavaScript es necesario para entender el lenguaje, pero no es recomendable utilizarlos a propósito ya hacen que el código sea difícil de entender y de peor mantenimiento. Una buena práctica es el chequeo del tipo de variable cuando se hacen operaciones que puedan provocar estos comportamientos.

11.6. Variables

Una variable permite almacenar valores para poder ser leídos en otro momento. Una variable en JavaScript se declara con la palabra reservada: let y el identificador de la variable.

```
let a; // Declaración de variable
a = 3; // Asignación de valor en variable
console.log(a); // la consola mostrará: 3
```

```
a = 2; // Reasignación de valor en variable
console.log(a); // la consola mostrará: 2
```

El identificador debe ser único, es decir, no puede haber dos variables con el mismo identificador/nombre. Conviene utilizar identificadores cortos, pero a la vez descriptivos para hacer la labor de desarrollo más sencilla. Al elegir un identificador, además se debe saber que:

- Se puede utilizar cualquier combinación de letras, números y el guion bajo.
- El primer carácter de un identificador no puede ser un número.
- JavaScript es case-sensitive, es decir, distingue entre minúsculas y mayúsculas por lo que la variable nombre y Nombre son dos variables distintas al tener distinto identificador.
- No se pueden utilizar palabras reservadas del lenguaje como identificador de variable, es decir, aquellas palabras que el lenguaje utiliza para sí mismo como this, let, const, if, etc.

Se puede declarar una variable y asignarle un valor en la misma sentencia:

```
let a = 3;
```

Si el valor de la variable no va a cambiar a lo largo de la ejecución, es conveniente utilizar la palabra reservada const para declararla.

```
const pi = 3.14;
pi = 33; // Esta sentencia dará un error ya que no puede reasignarse una variable declarada utilizando const
```

11.7. Operadores

Los operadores permiten hacer diferentes operaciones con variables.

Operadores aritméticos

Estos operadores permiten realizar operaciones aritméticas básicas, algunos de ellos son:

- Suma: +
- Resta: -
- Multiplicación: *
- División: /
- Resto: %: Devuelve el resto de una división entera.

```
const a = 3;
const b = 2;
let result;

result = a + b;
console.log(result); // 5
result = result - a;
console.log(result); // 2
result = result * b;
console.log(result); // 4
result = result / b;
console.log(result); // 2
```

Operadores de comparación

Estos operadores permiten comparar dos expresiones. JavaScript hará lo posible por compararlas y devolver un valor booleano: true o false.

- == igual.
- === igual estricto (compara tipo de la variable).
- != distinto.
- !== distinto estricto (compara tipo de la variable).
- > mayor que.
- < menor que.
- >= mayor o igual.
- <= menor o igual.

```
const a = 3;
const b = 100;

console.log(a == b); //

console.log(a != b); // true
console.log(a > b); //

console.log(a < b); // true

const c = 3;
const d = "3"; // Tipo cadena de texto, no número

console.log(a == c); // true
console.log(a === c); // true
console.log(a == d); // true
console.log(a === d); //
```

Operadores lógicos

Estos operadores permiten realizar operaciones lógicas, JavaScript hará lo posible por comparar los valores y realizar la operación lógica entre ellos.

- Operador lógico and: &&.
- Operador lógico or: ||.
- Operador lógico not: !.

```
  const a = true;
  const b =
;
  const c = true;

  console.log(a && b); //

  console.log(a || b); // true
  console.log(!c); // true
```

Operadores de asignación

Con un operador de asignación se asigna el valor de la derecha en el operador de la izquierda. Por ejemplo: x = 3 se asigna el valor 3 a x.

Algunos operadores de asignación:

- =: Asignación simple: x = 3.
- +=: Asignación de suma: x += 3 es lo mismo que: x = x + 3.
- -=: Asignación de resta: x -=3 es lo mismo que: x = x - 3.

Operadores de cadena

El operador + si es utilizado con una cadena de texto no realizará una suma (ya que no es posible sumar cadenas de texto), si no que realizará una concatenación:

```
  const hello = "Hola";
  const world = "Mundo";
  const greeting = hello + " " + world + "!";
  console.log(greeting); // "Hola Mundo!"

  const anumber = 3;
  const helloNanumber = hello + anumber;
  console.log(helloNanumber); // "Hola3
```

Tema 12. Estructuras de control de flujo

12.1. Introducción

El flujo del programa es la línea que sigue el dispositivo ejecutando código. Si no ocurre nada que lo altere, el flujo empezará en la primera sentencia e irá ejecutando una a una cada sentencia de arriba a abajo hasta llegar al final.

Sin embargo, es complicado entender un programa sin que el flujo pueda variar. Por ejemplo, en una web de la previsión meteorológica habrá condiciones en el flujo: si hace buen tiempo ocurre una cosa y si hace mal tiempo ocurre otra cosa. De esta manera el flujo tiene caminos diferentes que recorre hasta llegar al final de la ejecución.

12.2. Estructuras de control condicional: if/if-else

La estructura de control condicional if es la estructura más sencilla, con ella mediante una condición booleana el flujo puede tomar un camino específico para esa condición.

```
  // Suponiendo la variable temperatura como temperatura actual
  let mensaje = temperatura + "º";

  // Si la temperatura es menor que 0, se añaden unos caracteres haciendo
de copo de nieve
  if (temperatura < 0) {
    mensaje = "***" + mensaje + "***";
```

```
}
console.log(mensaje);
```

La condición booleana aparece entre paréntesis (). Esta condición será evaluada por JavaScript y si el resultado es true, el flujo entrará a ejecutar las sentencias que hay dentro de las llaves {}, en caso contrario, saltará dichas sentencias.

Se puede complicar un poco más añadiendo otro camino para el flujo mediante la estructura if-else: Si se cumple la condición el flujo seguirá un camino y si no, se irá a otro camino:

```
// Suponiendo la variable edad declarada con un número que representa la edad de una persona
if (edad >= 18) {
  console.log("mayor de edad");
} else {
  console.log("menor de edad");
}
```

Operador ternario

El operador ternario recrea la estructura de if-else pero de una manera muy simple y para casos en los que solo hay una sentencia para cada condición. Es simplemente una sintaxis diferente para esta estructura de control en el que se prescinde del if, else y de las llaves y se separa la condición de la primera sentencia con «?» y la primera sentencia de la segunda con «:».

```
// Todos los bloques siguientes imprimen por pantalla lo mismo
// Utilizando estructura if-else
if (edad >= 18) {
  console.log("mayor de edad");
} else {
  console.log("menor de edad");
}
```

```
// Utilizando operador ternario
(edad >=18) ? console.log("mayor de edad") : console.log("menor de edad");

// Utilizando el operador ternario en una asignación
const mensaje = (edad >=18) ? "mayor de edad" : "menor de edad";
console.log(mensaje);
// Utilizando el operador ternario directamente como parámetro de función
console.log((edad >=18) ? "mayor de edad" : "menor de edad");
```

12.3. Estructura de control condicional: switch

La estructura switch resuelve un flujo condicional cuando este tiene muchas opciones. Podría resolverse con un if-else muy grande, pero la estructura switch es más fácil de leer y mantener.

```
// Convierte una variable donde está el día de la semana en número en la palabra en castellano para ese día
let dia;
switch (diaNumero) {
  case 0:
    dia = "Lunes";
    break;
  case 1:
    dia = "Martes";
    break;
  case 2:
    dia = "Miércoles";
    break;
  case 3:
    dia = "Jueves";
    break;
  default:
    dia = "Error";
}
console.log(dia);
```

La sentencia break es parte de la estructura switch, es necesario añadirla ya que sin ella se evaluaría las sentencias del siguiente case, aunque la condición no fuese verdadera.

Por otro lado, default se puede utilizar como caso especial, el flujo evaluará las sentencias que estén dentro de él cuando ningún caso anterior haya sido seleccionado.

12.4. Estructuras de control iterativas: for/for in/for of

For

Estas estructuras es la forma más sencilla de manejar un bucle. Un bucle es una estructura de control que ejecuta una y otra vez un mismo conjunto de sentencias.

La estructura for funciona utilizando un índice (un número entero que va cambiando de valor) como control, de tal manera que se establece:

- Un punto de inicio.
- La manera en la que ese índice va a cambiar.
- Un punto de final.

A cada vuelta se puede consultar el índice. Utilizando la sintaxis del for se indica esas condiciones para el bucle, dentro de los paréntesis () y separado por «;» se indica:

- El índice que se va a utilizar.
- La condición para la que el bucle seguirá repitiéndose.
- Qué transformación se hace del índice a cada iteración (vuelta del bucle).

```
// En el siguiente bucle el índice empieza en el 0
// Seguirá repitiéndose mientras el índice sea menor que 10
// A cada vuelta, se sumará 1 al índice (i++ es lo mismo que i = i + 1)

//For para rellenar lista
for (i = 0; i < 10; i++) {
  console.log(i);
}

// Esto mostrará por consola:
// 0123456789
```

for in

Funciona de una manera similar al bucle for, sin embargo, en lugar de utilizar un índice, recorre un array o un objeto y deja disponible un índice correspondiente a cada uno de los elementos del array u objeto:

```
const arr = ["a", "b", "5"];
for (indice in arr) {
  console.log(indice); // Cada uno de los índices / posiciones del array
  console.log(arr[indice]); // Cada uno de los elementos que ocupan posiciones en el array
}
// Se mostrará por consola
// 0 a 1 b 2 c
```

for of

Esta estructura itera un objeto iterable como un array y deja en cada vuelta del bucle disponible para una de las partes que componen ese objeto iterable:

```
let array = ["a", "b", "c"];

for (let value of array) {
  value = value + "33";
```

```
    console.log(value);
  }
  // a33 b33 c33

  // Si el valor no se va a modificar dentro del bucle, se puede declarar
la variable con const
  let array = ["a", "b", "c"];

  for (const value of array) {
    console.log(value);
  }
  // a b c
```

12.5. Estructuras de control iterativas: while/do while

La estructura while como su nombre indica realiza un bucle mientras una condición se cumpla, en el momento que esa condición se evalúe como falsa, el bucle acabará.

```
  let i = 0;
  while (i < 4) {
    console.log(i);
    i++;
  }

  // Se mostrará por consola
  // 0 1 2 3 4


  let i = 10;
  while (i < 4) {
    console.log(i);
    i++;
  }
```

```
  // No se mostrará nada por consola porque en la primera vuelta del bucle
se evalúa ( 10 < 4) a
 por lo que no entra en el bucle
```

Similar al while, la estructura do-while que es similar con una salvedad: la primera vuelta se ejecutará siempre:

```
let i = 1;
const n = 5;

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

// Se mostrará por consola
// 1 2 3 4 5
```

12.6. Interrumpir, continuar o abandonar en bucle

La sentencia brake sirve para terminar bucles, hacer que el flujo no respete el bucle y salte como si el bucle hubiese acabado.

La utilización de brake debe hacerse en muy determinados casos, siempre es más recomendable diseñar correctamente un bucle antes de tener que recurrir a esta sentencia, a excepción de la sentencia switch donde la utilización de brake es requerida.

Por ejemplo, el siguiente bucle está diseñado para ir del 1 al 5, sin embargo, en su ejecución cuando el valor del índice es 3, la sentencia break hace que el flujo salte, evitando el resto del bucle.

```
// Bucle con un índice del 1 al 5
for (let i = 1; i <= 5; i++) {
  if (i === 3) {
    break;
  }
  console.log(i);
}

// Se mostrará por consola
// 1 2
```

12.7. Gestión de excepciones try-catch

Una excepción es un error que se produce (por la razón que sea) cuando un programa se está ejecutando, y salvo que se gestione esa excepción, la ejecución del programa no puede seguir.

Para manejar excepciones y que, aunque ocurra una el programa pueda seguir ejecutándose, se utiliza la estructura try/catch.

```
try {
  console.log("pre error");
  console.log(a);
  console.log("post error");
}

catch(error) {
  console.log('Error: ' + error);
}

// Mostará por consola
// pre error
// Error: ReferenceError: a is not defined
```

En el anterior ejemplo el flujo de ejecución entra en el try, cualquier excepción que ocurra dentro de ese bloque try será manejado por el bloque catch, por donde continuará la ejecución (saltando el resto de las sentencias que pudiese haber en el bloque try).

Como complemento al try-catch existe finally, gracias a finally se pueden agrupar sentencias de manera que se **ejecutarán tanto ocurra una excepción como si no**.

```
// Se produce excepción
try {
  console.log("pre error");
  console.log(a);
  console.log("post error");
}
catch(error) {
  console.log('Error: ' + error);
}
finally {
  console.log("finally");
}

// pre error
// Error: ReferenceError: a is not defined
// finally
// No se produce excepción
try {
  console.log("pre error");
  console.log("post error");
}
catch(error) {
  console.log('Error: ' + error);
}
finally {
  console.log("finally");
}
// pre error
// post error
// finally
```

Tema 13. Funciones

13.1. Introducción

Una función es un bloque de código que realiza una determinada tarea. A base de esos pequeños bloques de código es como se construyen los programas (dividen un problema grande en otros más pequeños).

Por ejemplo, una función puede revisar si una cuenta bancaria es correcta (es un cálculo matemático simple) y es muy posible que esté dentro del código de la página web de un banco.

```
function nombreDeLaFuncion () {
  // Contenido de la función
}
```

Las funciones se declaran en JavaScript utilizando la palabra reservada function seguida del nombre que se desea dar a la función y entre llaves {} ese trozo de código que debe ejecutar la función.

Una vez declarada la función, se puede llamar a su ejecución utilizando el nombre de la función seguido por ().

```
console.log("pre function");
function saludaPorConsola() {
  console.log("Hola, soy JavaScript");
}
console.log("post function");

console.log("pre llamada");
saludaPorConsola();
console.log("post llamada");
```

```
// El anterior código escribirá en la consola:
// pre function
// post function
// pre llamada
// Hola, soy JavaScript
// post llamada
```

En el ejemplo anterior, el flujo de ejecución pasará por la función sin ejecutarla, pero posteriormente al llamarla por su nombre, el flujo entra dentro de la función y ejecuta ese código que está envuelto en las llaves, cuando el código de la función acaba, el flujo vuelve a donde la función se había llamado.

13.2. Parámetros de una función

Con el fin de hacer esos trozos de código más útiles y reutilizables, las funciones pueden recibir parámetros.

Los parámetros son valores que se reciben en la función y actúan como variables predeclaradas solo en el ámbito (entre llave y llave) de la función.

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

saludar("Félix");
// Mostrará por consola: Hola Félix

saludar("Fran");
// Mostrará por consola: Hola Fran
```

En el ejemplo anterior, se ha definido la función saludar que acepta el parámetro nombre de manera que puede haber muchos tipos de saludos, tantos como

parámetros nombre reciba la función. De esta manera podemos saludar a diferentes nombres pasando como argumento de la función una cadena de texto en el momento de llamar a la función: saludar ("Félix").

Las funciones pueden tener varios parámetros, cada uno de esos parámetros se separa con «,».

```
function sumar(a, b) {
  console.log(a + b);
}

sumar(2,3);
// Mostrará por consola 5
```

Los parámetros de una función pueden tener un valor por defecto, de esta manera ese parámetro se convierte en opcional, ya que no es necesario pasarlo al llamar la función (hay un valor por defecto en tal caso).

```
function saludar(nombre = "Desconocido") {
  console.log("Hola " + nombre);
}

saludar(); // Hola Desconocido
saludar("Víctor") // Hola Víctor
```

13.3. Retorno de una función

Ese conjunto de sentencias que es una función puede además devolver un valor, de esta manera es muy útil para abstraer cálculos y simplemente llamando a una función, se obtiene un resultado.

```
function sumar(a, b) {
  return a + b;
```

```
}

const x = sumar(2,3);
console.log("La suma es: " + x);
// Mostrará por consola
// La suma es: 5
```

En el ejemplo anterior la función sumar devuelve la suma de los dos números que entran como parámetro. Ese resultado se guarda en la constante `x` que posteriormente se utiliza.

Conviene saber que:

- Una función no ejecuta más código tras encontrar la palabra return.
- Si una función no tiene la palabra return, devolverá siempre undefined.

13.4. Ámbito de variables

El ámbito de una variable define las partes del código en la que una variable está disponible para utilizarse.

JavaScript tiene dos ámbitos de variable:

- Ámbito de variables global.
- Ámbito de variables local.

Ámbito global

Las variables declaradas fuera de una función se consideran variables de ámbito global. Al declararse fuera de una función estas variables están disponibles para utilizarse en cualquier parte del programa.

```
const nombre = "Gerardo";

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

saludar();
// Mostrará por consola
// Hola Gerardo
```

Es conveniente controlar las variables de ámbito global y utilizarlas realmente cuando se necesita que una variable esté disponible en todo el código. Las variables también se pueden modificar en cualquier parte del código y eso puede llevar a comportamientos indeseados:

```
function saludar() {
  nombre = "Juanjo";
}
let nombre = "Víctor";

saludar();

console.log("Hola " + nombre);
// Mostrará por consola
// Hola Juanjo
```

Ámbito local

Si una variable se declara dentro de una función, esa variable estará únicamente disponible dentro de esa función. Cada función tiene su ámbito propio llamado ámbito local.

```
const nombre = "Víctor";

function saludar() {
  const prefijo = "Hola ";
```

```
  return prefijo + nombre;
}

console.log(saludar());
// Hola Víctor

console.log(prefijo);
// ReferenceError: prefijo is not defined
```

En el ejemplo anterior:

- La variable nombre es de ámbito global y por lo tanto está disponible para usar dentro de todo el código, incluido dentro de la función saludar.
- La variable prefijo es de ámbito local ya que se ha declarado dentro de la función saludar y por lo tanto solo estará disponible dentro de dicha función. Al intentar utilizarla fuera de la función JavaScript no sabe de ella y da error.

Con la declaración de variables con let y const se puede hablar incluso de **ámbito de bloque**. Toda variable declarada de esta manera estará disponible entre las llaves {} en la que se encuentre:

```
function saludar(nombre) {
  let prefijo = "Hola ";

  if (nombre === "Félix") {
    let sufijo = ", cuanto tiempo!";
    console.log(prefijo + nombre + sufijo);
  } else {
    console.log(prefijo + nombre);
    console.log(sufijo);
  }
}

saludar("Félix");
// Hola Félix, cuanto tiempo!
```

```
saludar("Gerardo");
// Hola Gerardo
// ReferenceError: sufijo is not defined
```

En el vídeo *Contexto de ejecución en JavaScript* se explica de manera práctica los contextos de ejecución / ámbitos de variable.

 Accede al vídeo

Reasignación de variables

Se puede dar el caso de que se use el mismo identificador para una variable global que para una local, JavaScript está preparado para ello y utilizará primero las variables de ámbito local sobre las de ámbito global.

```
const plato = "Paella";

function cocinar() {
  const plato = "Bistec";
  console.log("local", plato);
}

cocinar(); // local Bistec
console.log("global", plato); // global Paella
```

13.5. Funciones anónimas

Las funciones se pueden definir como expresión, es decir, se pueden guardar en una variable. Al guardar una función en una variable **ya no es necesario asignarle nombre** por lo que se convierte en una función anónima. Para llamar a una función anónima

guardada en una variable simplemente se debe escribir el nombre de esa variable seguido de ().

```
// La constante saludar guarda una función anónima
const saludar = function() {
  console.log("hola!");
}

// Llamada a la función en la variable saludar
saludar(); // hola!
```

13.6. Arrow functions

Esta nueva sintaxis para funciones se introdujo en la revisión ES6 de JavaScript, la cual hace más sencillo escribir funciones.

```
// La función sumar funciona de la misma manera en los siguientes ejemplos
const sumar = function(a, b) {
  return a + b;
}

// Equivalencia en arrow function
const sumar = (a, b) => {
  return a + b
}

// Como únicamente tiene una sentencia return, se puede quitar junto con las llaves
const sumar = (a, b) => a + b
```

13.7. Rest parameters

Uno de los problemas que se presenta en proyectos complejos es la cantidad de parámetros que finalmente puede tener una función. Utilizando un parámetro rest una función puede recibir un número indeterminado de parámetros.

Simplemente, en el último parámetro de una función se debe añadir como prefijo …, esto hará que el resto de los parámetros pasados a la función se agrupen en un array.

```
function as(a, b, ...otros) {
  console.log('a', a);
  console.log('b', b);
  console.log('otros', otros);
}

as(1,2,3,4,5,6,7,8,9);
// 'a' 1
// 'b' 2
// 'otros' [3,4,5,6,7,8,9]
```

Tema 14. Manejo de datos

14.1. Introducción

Gracias a determinado tipo de datos que incluye JavaScript como los arrays y objetos se hace muy sencillo para el programador manejar gran cantidad de datos.

14.2. Arrays

Array es una colección de elementos que se guardan juntos con el fin de hacer operaciones como recorridos, mutaciones u ordenación.

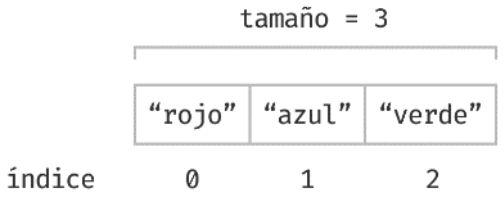

Figura 1. Esquema de un array. Fuente: elaboración propia.

La manera más sencilla de crear un array es utilizar los caracteres [] y asignar dicho array a una variable.

```
const arrayVacio = [];
const colores = ["rojo","verde","azul"];

// Otra manera de crear un array
const arrayVacioOldSchool = new Array();
const numerosOldSchool = new Array(1, 2, 3);
```

Una vez declarado un array, se puede acceder a sus elementos utilizando un índice. Para ello, basta con escribir el nombre del array y el índice entre dos []. Los arrays en JavaScript empiezan en el índice 0 y tendrán tantos índices como elementos - 1:

```
// Hay 3 colores en el array ocupando los índices 0, 1 y 2
const colores = ["rojo", "azul", "verde"];
console.log(colores[0]); // rojo
console.log(colores[1]); // azul
console.log(colores[2]); // verde
console.log(colores[3]); // undefined
```

Una vez creado un array es posible añadir o quitar elementos con los métodos propios del array para tal caso:

- push() Añade un elemento al final del array.
- unshift() Añade un elemento al principio del array.
- pop() Quita el último elemento de un array.
- shift() Quita el primer elemento de un array.

```
const array = [];
array.push("rojo");
array.push("azul"); // push Añade un elemento al final de array
console.log(array); // rojo, azul
array.unshift("verde");
console.log(array) // verde, rojo, azul
array.pop();
console.log(array) // verde, rojo
array.shift();
console.log(array) // rojo
```

Además, es posible reemplazar un elemento que ya pertenezca al array sabiendo su índice, simplemente se debe realizar una asignación:

```
const colores = ["rojo", "azul", "verde"];
colores[1] = "violeta";
```

```
console.log(colores) // rojo, violeta, verde
```

Arrays multidimensionales

Un array multidimensional es un array de arrays, es decir, un array cuyos elementos son otros arrays. Los arrays multidimensionales se tratan de la misma manera que cualquier array, tanto el array principal como los arrays que lo componen, simplemente cambia el modelo mental del programador al utilizarlos.

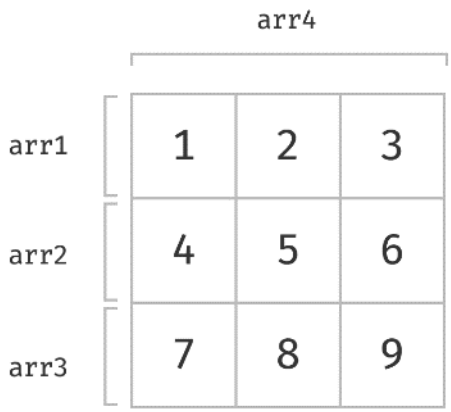

Figura 2. Esquema de un array multidimensional. Fuente: elaboración propia.

```
const arr1 = [1, 2, 3];
  const arr2 = [4, 5, 6];
  const arr3 = [7, 8, 9];
  const arr4 = [arr1, arr2, arr3]; // Array multidimensional
```

Los arrays multidimensionales se recorren como cualquier otro array, simplemente se debe entender que cada elemento de este tipo de array es otro array y por lo tanto también se puede recorrer. No hay límite de dimensiones, simplemente es un límite de memoria.

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

// Recorrido del primer nivel del array: iterador i
for (i = 0; i < multiarray.length; i++) {
  // Para cada nivel, se recorre el array de ese nivel: iterador j
  for (j = 0; j < multiarray[i].length; j++) {
    // Se utilizan los dos iteradores para acceder a cada elemento
    console.log(multiarray[i][j]);
  }
}
// Mostrará por consola
// 1 2 3 4 5 6 7 8 9
```

14.3. Métodos y propiedades de arrays

length

Gracias a la propiedad length se puede consultar el tamaño de un array. Como los arrays empiezan en el índice 0, se puede decir que el último índice disponible de un array será su tamaño - 1.

```
let array = [];
console.log(array.length); // 0
array = [1, 2, 3, 4, 33];
console.log(array.length); // 5
console.log(array[array.length - 1]); // 33
```

foreach

Gracias al método forEach incluido en los arrays de JavaScript, se puede iterar un array de una manera muy sencilla, al igual que se hace con un bucle for pero sin tener que mantener el índice.

forEach es un método que necesita como argumento una función. Dicha función recibe como parámetro cada uno de los elementos del array en cada una de las vueltas, es decir, esa función se ejecuta tantas veces como elementos tenga el array y en cada ejecución su primer parámetro es el elemento actual.

```
const colores = ["rojo", "verde", "azul"];
colores.forEach(function(value) {
  // En value está el elemento del array correspondiente
  // En la primera ejecución de esta función value = rojo
  // Segunda ejecución: value = verde
  // Tercera ejecución: value = azul
  console.log(value);
});

// Mostrará por consola
// rojo verde azul
```

El método forEach recorrerá desde el primer hasta el último elemento del array ejecutando la función para cada elemento, de manera que el programador no debe preocuparse por el índice. Ese índice se puede consultar ya que le llega a la función como el segundo parámetro (es opcional utilizarlo).

```
const colores = ["rojo", "verde", "azul"];
colores.forEach(function(value, i) {
  console.log(value);
  console.log(i);
});

// Mostrará por consola
// rojo 0 verde 1 azul 2
```

map

El método map mapea un array: para cada elemento lo transforma con una función y devuelve el resultado en otro array.

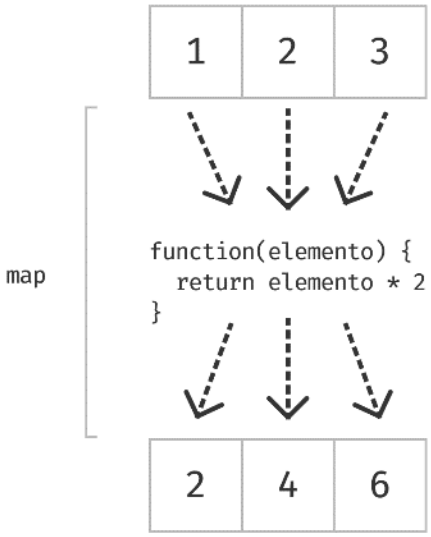

Figura 3. Esquema del método map() transformando un array. Fuente: elaboración propia.

Este método se utiliza asignando el resultado, ya que **no transforma el array original**. Recibe como argumento una función que se utilizará para transformar cada elemento del array. Dicha función recibe como parámetro cada elemento y mediante una sentencia return devuelve el elemento transformado.

```
const numeros = [1, 2, 3, 4, 33];

const numerosPor2 = numeros.map(function(elemento) {
  return elemento * 2;
})

console.log(numerosPor2); // 2, 4, 6, 8, 66
```

find

El método find permite buscar elementos dentro de un array. Este método revisará cada uno de los elementos y parará de buscar una vez haya encontrado aquel elemento que está buscando, esto quiere decir que o bien no encuentra nada o bien encuentra un elemento, pero no encontrará más de un elemento. Como otros métodos de este tipo, no modifica el array, si no que devuelve el resultado.

Este método necesita una función como argumento que utilizará por cada uno de los elementos del array. Esa función tiene como parámetro cada uno de los elementos del array y se puede comprobar por cada elemento si cumple la condición de búsqueda: si la función devuelve true, dará por encontrado el elemento, sin embargo, si la función devuelve seguirá buscando.

```
const numeros = [1, 2, 3, 4, 33];

// Búsqueda con fin y función tradicional
const numeroMayorQueCuatro = numeros.find(function (elemento) {
  return elemento > 4;
});

console.log(numerMayorQueCuatro); // 33

// Utilizando arrow function
const numeroMayorQueCuatro = numeros.find(elemento => elemento > 4);
console.log(numerMayorQueCuatro); // 33

// Dejando la función en una constante
const mayorQueCuatro = elemento => elemento > 4;
const numeroMayorQueCuatro = numeros.find(mayorQueCuatro);
console.log(numerMayorQueCuatro); // 33
```

filter

El método filter es similar a find. Su diferencia es que puede encontrar todos los elementos que cumplan una condición: esa condición es la que decida la función que se le pasa como argumento.

```
const numeros = [1, 2, 3, 4, 16, 33, 1502];

const pares = numeros.filter(function(numero) {
  return numero % 2 === 0;
});
```

```
console.log(pares); // 2, 4, 16, 1502
```

sort

El método sort ordena un array siguiendo la condición que marque la función que se le pasa como argumento. La función de ordenación recibe dos parámetros que son utilizados para hacer precisamente la comparación como si fuesen dos elementos del array.

```
const numeros = [33, 45, 1, 100, 2, 4];

const ordenados = numeros.sort(function(a, b) {
  return a - b;
});

console.log(ordenados); // 1, 2, 4, 33, 45, 100
```

Una función de comparación function(a,b) devuelve:

- < 0. Si el valor devuelto por la función de comparación es menor que 0, entonces el elemento b es mayor que a.
- Si el valor devuelto por la función de comparación es 0, entonces a y b deben tener la misma posición.
- 0. Si el valor devuelto por la función de comparación es mayor que 1, entonces el elemento.

reduce

El método reduce como su nombre indica, reduce un array a una variable, es decir, de una forma similar al map, pasa por todos los elementos de un array y ejecuta una función.

La función recibe por parámetro el elemento actual del array y un acumulador, ese acumulador es lo que al final se reduce del array, es decir, para cada elemento del array transforma el acumulador que ya previamente han transformado los elementos anteriores y que transformarán los elementos siguientes. Para que el acumulador pase al siguiente elemento, la función debe devolver la transformación que le ha hecho al acumulador.

```
const numeros = [33, 45, 1, 100, 2, 4];

const suma = numeros.reduce(function(acumulador, numero) {
  return acumulador + numero;
});

console.log(suma); // 185
```

En el vídeo *Métodos asociados a arrays en JavaScript* se explican todos estos métodos partiendo del mismo array y escribiendo funciones que transforman, ordenan o filtran.

 Accede al vídeo

14.4. Objetos

Los objetos son un tipo de dato para almacenar múltiples colecciones de datos mediante clave/valor.

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

Cada una de las propiedades del objeto: claves, funciona como una variable declarada dentro del objeto, de manera que la clave es el nombre de la variable y el valor es el contenido de la variable que puede ser de cualquier tipo: un string, booleano, un array, incluso otro objeto.

Para acceder a las propiedades de un objeto se puede utilizar la misma sintaxis que para acceder a un array o mediante el carácter «:».

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

console.log(persona['nombre']); // Mercer
console.log(persona.apellido); // Roussel

// Si la propiedad no existe, se devuelve undefined
console.log(persona['xyz']); // undefined
```

14.5. Métodos de objetos

En JavaScript una variable puede contener una función, por lo que una propiedad de un objeto puede contener una función:

```
const persona = {
  nombre: 'Mercer',
  apellido: 'Roussel',
  saludar: function() { console.log('Soy Mercer') }
}

persona.saludar(); // Soy Mercer
```

De esta manera se añaden nuevas funcionalidades al objeto para hacerlo un poco más eficaz al utilizarse. Esas funciones predefinidas pueden incluso acceder al propio objeto en sí, utilizando para ello la palabra reservada this. this es una palabra reservada en JavaScript que se refiere al contexto de ejecución en el que se realiza una sentencia, en el caso de funciones dentro de objetos, this hace referencia al mismo objeto ya que es el contexto en el que se ejecuta dicha sentencia.

```
const persona = {
  nombre: 'Mercer',
  apellido: 'Roussel',
  saludar: function() { console.log('Soy Mercer') }
}

persona.saludar(); // Soy Mercer
```

14.6. Recorrer un objeto

Aunque un objeto está diseñado para algerbar información de todo tipo, en determinados casos puede ser interesante recorrerlo (por ejemplo, para buscar información).

Es interesante destacar que JavaScript no asegura el orden de las claves de un objeto, porque un objeto en realidad está pensado para albergar información y recuperarla con una clave y no para almacenar información de manera ordenada (para eso están los arrays), por lo tanto, el orden en el que aparecen las claves en una iteración puede no ser el mismo orden en el que se definieron en el objeto.

```
// Bucle for ... in
const numeros = { a: 1, b: 2, c: 3 };

for (const clave in numeros) {
  console.log(`${clave}: ${numeros[clave]}`);
}
```

```
// Muestra por consola:
// a: 1
// b: 2
// c: 3

// Object.keys
// Devuelve un array con las claves del objeto de manera que se puede iterar
const numeros = { a: 1, b: 2, c: 3 };
Object.keys(numeros).forEach(function(clave){
  console.log(`${clave}: ${numeros[clave]}`);
});

// Muestra por consola:
// a: 1
// b: 2
// c: 3
```

14.7. Recorriendo un array de datos consumidos desde API

Un programador JavaScript es muy habitual que necesite hacer llamadas a una API. Una API es un servicio que expone un servidor mediante al cual se pueden realizar peticiones con el fin de recuperar o almacenar datos, realizar acciones, etc.

Una llamada a API muy habitual se realiza para recoger datos, por ejemplo, un listado de usuarios. Habitualmente esos listados se entregan como un array de objetos: cada objeto es una entidad y el array los contiene.

```
const listadoUsuarios = [
  {
    nombre: "Gabriel Pacheco",
    edad: 34,
    signoZodiaco: "Virgo"
```

```
  },
  {
    nombre: "Lola Rocha Tercero",
    edad: 30,
    signoZodiaco: "Capricornio"
  },
  ...
];
```

Gracias a los métodos de los que disponen los arrays en JavaScript, se pueden realizar búsquedas o transformaciones habituales de dichos listados:

```
// Obtener solo nombres
const soloNombres = [];
listadoUsuarios.forEach(function(usuario){
  soloNombres.push(usuario.nombre);
});

// Obtener solo nombres (manera más sencilla)
const soloNombres2 = listadoUsuarios.map(function(usuario) {
  return usuario.nombre;
});

// Edad total (suma de todas las edades)
// Con arrow function
const edadTotal = listadoUsuarios.reduce((acc, usuario) => acc + usuario.edad, 0);

// El segundo parámetro que recibe reduce es el valor inicial del acumulador en este caso, es 0

// Obtener solo los usuarios con signo Virgo
const soloVirgo = listadoUsuarios.filter(function(usuario) {
  return usuario.signoZodiaco === 'Virgo';
});
```

Tema 15. Manejo del DOM

15.1. Introducción

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.

15.2. 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).

15.3. 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
})
```

15.4. 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.

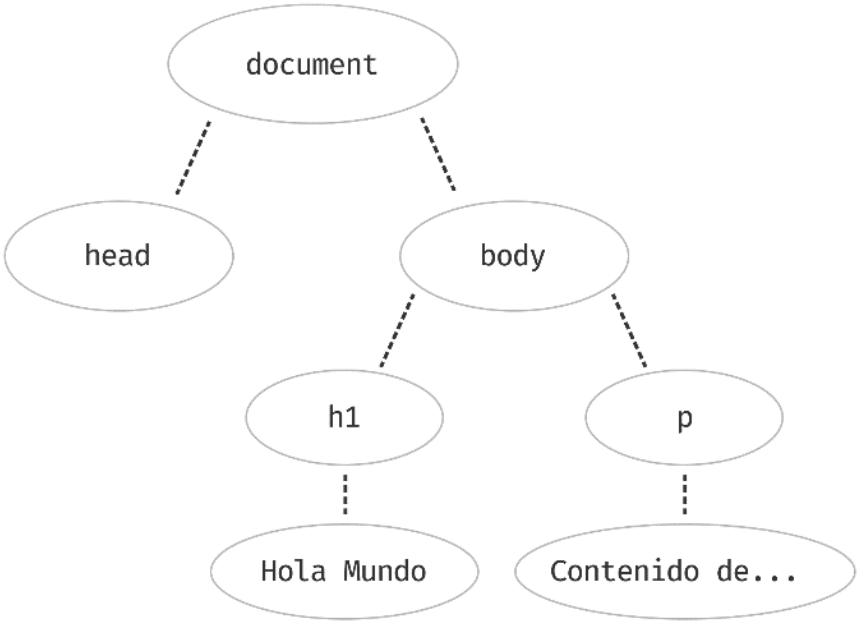

Figura 4. Árbol de nodos DOM. Fuente: elaboración propia.

```
<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
```

15.5. 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>
```

15.6. 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
```

Tema 16. JavaScript orientado a objetos

16.1. Introducción

La programación orientada a objetos es una manera de programar, organizando el código en bloques denominados clases, y mediante la creación de objetos (instancias de dichas clases) que comunicándose entre sí crean los programas.

Las clases de la programación orientada a objetos son cercanas a cómo se expresarían los conceptos en el mundo real, pero, por así decirlo, codificados para que sea un programa quién los cree, modifique o destruya. Esta manera de abstraer bloques de código cercanos al mundo real hace que las aplicaciones sean más fáciles de entender, mantener y de hacer el código más reutilizable.

16.2. ¿Qué es una clase? ¿Y un objeto?

Una clase es un concepto con unas propiedades definidas y unos métodos que implementan sus funcionalidades. Por ejemplo, pensando en el mundo real, un coche podría codificarse como una clase, Un coche tiene:

- **Propiedades**: Como el color, marca, tiene o no airbag, abs, matrícula, etc.
- **Métodos**: El coche puede arrancar, detenerse, acelerar, frenar, etc.

Al pensar en esta clase abstracta de coche, se puede pensar en todos los coches del mundo: todos los coches tienen esas propiedades y esos métodos. La **idea de coche**

es la clase y la representación real de esa idea es un objeto: un coche verde, matrícula XXXX de la marca Seat. Un objeto es una **instancia** de una clase.

Los objetos hablan entre sí

De nada sirve un objeto por sí mismo, al igual que en la vida real, en los programas, los objetos tienen que hablar entre sí para solventar problemas.

Siguiendo el ejemplo del mundo real, un coche pertenece a una persona, se guarda en un garaje o circula por una autopista. Persona, garaje o autopista también se pueden codificar como clases, de manera que tienen, albergan o circulan por ellas instancias de la clase *coche*. De esta manera se crean los programas, se codifica el modelo mental de la realidad para que la programación sea más cercana a ella y más fácil de entender por los desarrolladores.

16.3. Definición de una clase en JavaScript

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);
    }
  }
}
```

16.4. Conceptos avanzados de POO en JavaScript

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);
    }
  }
}
```

Tema 17. JavaScript asíncrono

17.1. Introducción

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.

17.2. 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.

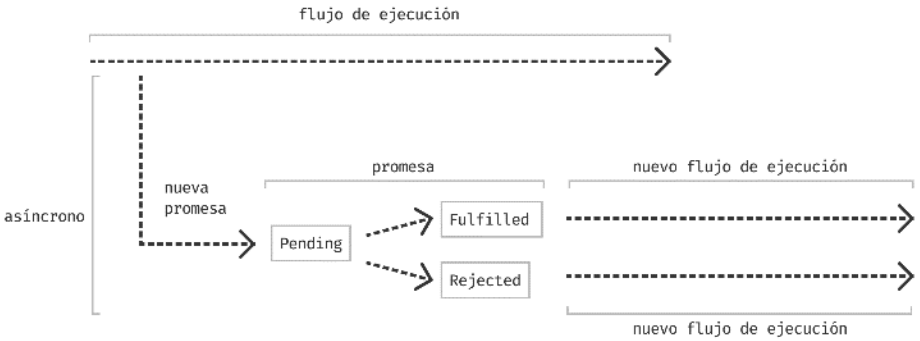

Figura 5. Esquema de una promesa. Fuente: elaboración propia.

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.

17.3. 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
```

17.4. 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:
// Función asíncrona
// ok
```

17.5. 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.

En el vídeo *Consultando una API con fetch* se explica este método a través de la consulta de una API real.

 Accede al vídeo

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.

A fondo

Primeros pasos con JavaScript

MDN Contributors. (2022). *Primeros pasos con JavaScript* [documentación en línea]. https://developer.mozilla.org/es/docs/Learn/JavaScript/First_steps

Introducción a JavaScript de la mano de MDN Web Docs, conviene tener a mano este tipo de documentaciones ya que suelen estar actualizadas, pero no se debe tomar como documentación de estudio.

Rendimiento de los diferentes tipos de bucle de JavaScript

Dulanga, C. (2020). *Measuring Performance of Differente JavaScript Loop Types* [artículo en línea]. https://blog.bitsrc.io/measuring-performance-of-different-javascript-loop-types-c0e9b

Una de las labores del programador no es solamente que su programa funcione, si no que este funcione rápido. Los bucles pueden ser una fuente de bajada de rendimiento si no se utilizan correctamente, en este artículo se miden cuanto tardan los tipos de bucle de JavaScript.

Tipo de dato Set

MDN Contributors, J. (2022). *Set* [documentación en línea]. https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Set

Set es un tipo de dato que trae JavaScript similar a un array pero con métodos interesantes para manejar los datos.

Tipo de dato Map

MDN Contributors, J. (2022). *Map* [documentación en línea]. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map

Map es un tipo de dato en JavaScript parecido a un objeto, pero con métodos que hacen más sencilla su utilización.

Guardar datos en el navegador

Kantor, I. (2021). *Cookies, document.cookies* [documentación en línea]. https://es.javascript.info/cookie

Kantor, I. (2021). *LocalStorage, sessionStorage* [documentación en línea]. https://es.javascript.info/localstorage

Mediante el uso de cookies o localStorage, el programador puede guardar información en el navegador del usuario, con el fin de tenerla disponible cuando el usuario entre otra vez en el sitio web, realizar cálculos en el lado del cliente u otros usos.

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.

Propiedades y métodos estáticos

Kantor, i. (2021). *Propiedades y métodos estáticos* [documentación en línea]. https://es.javascript.info/static-properties-methods

Las clases en JavaScript soportan un tipo de propiedades y métodos útiles en determinadas situaciones llamados *estáticos* para los que no hace falta instanciar la clase para acceder a ellos.

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.

Códigos de respuesta HTTP

Ofiwe, M. (2021). *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.

Test

1. ¿Cómo se puede saber el tipo de una variable?

A. Todas las variables tienen una propiedad *.type.*

B. JavaScript como tiene tipado débil no tiene tipos de variable.

C. Con typeof.

D. No se puede saber el tipo de una variable.

2. ¿En qué caso se da un bucle infinito?

A. Un while donde la condición es siempre true.

B. Al no utilizar la sentencia brake dentro de un bucle.

C. En un bucle for … in mal construido.

D. Al no utilizar la sentencia brake en cada condición de un switch

3. Si una función anónima está asignada a una variable llamada saludar que está declarada en un ámbito global. ¿Cómo se puede utilizar dentro de una función? (suponiendo que en ningún sitio se declare otra variable con el identificador saludar).

A. this.saludar().

B. saludar().

C. global.saludar().

D. console.log("saludar").

4. El array *numeros = [10, 20, 33]* ¿qué elemento tendrá en el índice 3 (*numeros[3]*)

A. 20.

B. 33.

C: null.

D: undefined.

5. ¿Cuál de las siguientes sentencias devuelve 3? Suponiendo *numeros = [1,2,3];*

A. numeros.length().

B. .numeros.length.

C. .numeros.map((elemento) => elemento + 1).

D. números.lenght.

6. ¿Qué devuelve document.querySelectorAll('.noexiste') suponiendo que en el documento no hay ningún elemento con la clase .noexiste?

A. Array vacío.

B. undefined.

C. null.

D. .noexiste is not defined.

7. ¿Qué se debe tener en cuenta antes de manipular el DOM a través de JavaScript?

A. Debemos asegurarnos de que el navegador ha recibido el documento HTML.

B. Debemos asegurarnos de que el navegador ha ejecutado todo el JavaScript de la página utilizando para ello el evento onload en todos los scripts de la página

C. Debemos asegurarnos de que el documento ha lanzado el evento DOMContentLoaded.

D. Debemos asegurarnos de tener el carné de manipulador de documentos.

8. ¿Cuál de las siguientes afirmaciones es falsa?

A. Gracias a la herencia de la programación orientada a objetos código del padre puede ser reutilizado por hijos.

B. En el constructor de una clase que hereda de una clase padre es recomendable añadir una llamada al constructor del padre.

C. En una clase hija, todas sus propiedades son privadas.

D. Es recomendable que el nombre de una clase empiece con mayúsculas.

9. ¿Qué devuelve fetch() si se le introduce por parámetro una url válida?

A. Un array.

B. undefined.

C. Una promesa.

D Un string con formato JSON.

10. ¿Qué devolverá JSON.stringify() pasándole como parámetro una cadena de texto que contenga: {name: “David”}:

A. Un objeto.

B. Undefined.

C. Un JSON.

D. Un string.