Checkpoint nº 7: teoría

1. ¿Qué diferencia a JavaScript de cualquier otro lenguaje de programación?

JavaScript comparte muchas características con otros lenguajes de programación, pero tiene varias diferencias clave que lo distinguen, especialmente debido a su historia y su rol principal en el desarrollo web.

1. **El Navegador como Entorno Principal (Históricamente)**
    - **Diferencia:** JavaScript fue diseñado *específicamente* para ejecutarse en navegadores web y manipular el Document Object Model (DOM). Ningún otro lenguaje tiene esta posición nativa y universal dentro de los navegadores. Aunque ahora existen tecnologías como WebAssembly (WASM) que permiten ejecutar otros lenguajes, JavaScript sigue siendo el lenguaje fundamental del front-end web.
    - **Comparación:** Lenguajes como Python, Java, C#, Ruby, etc., se ejecutan principalmente en servidores o como aplicaciones de escritorio/móviles, no directamente en el navegador del usuario de forma nativa (sin plugins o tecnologías adicionales como WASM).
2. **Lenguaje Interpretado y "Just-in-Time Compiled"**
    - **Diferencia:** tradicionalmente, JavaScript ha sido un lenguaje interpretado, lo que significa que el código se ejecuta línea por línea por un motor (como el V8 de Google) sin una fase de compilación previa a un ejecutable binario. Sin embargo, los motores modernos de JavaScript utilizan técnicas de compilación "Just-in-Time" (JIT) para optimizar el rendimiento, compilando partes del código caliente a código máquina durante la ejecución.
    - **Comparación:** a pesar de esto, la percepción y la forma en que se suele trabajar con se alinea más con los lenguajes interpretados en comparación con lenguajes puramente compilados como C++ o Java (que compila a bytecode).
3. **Modelo de Concurrencia: Single-Threaded con Bucle de Eventos (Event Loop)**
    - **Diferencia:** El núcleo de JavaScript (en la mayoría de sus entornos, como navegadores y Node.js) opera en un único hilo principal. Maneja la concurrencia (múltiples tareas a la vez) no mediante múltiples hilos (como Java o C++), sino a través de un *bucle de eventos* y operaciones de entrada/salida (I/O) no bloqueantes. Las tareas que toman tiempo (como peticiones de red o timers) se delegan y, cuando terminan, colocan una función (callback, Promise, async/await) en una cola para ser ejecutada por el hilo principal cuando esté libre.
    - **Comparación:** Muchos otros lenguajes populares utilizan multihilo (multi-threading) para la concurrencia, lo que presenta diferentes desafíos (condiciones de carrera, bloqueos, etc.) y requiere mecanismos de sincronización explícitos. El modelo de simplifica algunas cosas pero requiere una forma diferente de pensar sobre el flujo del programa (programación asíncrona).
4. **Tipado Dinámico y Débil (Aunque con Evoluciones)**
    - **Diferencia:** JavaScript es un lenguaje de tipado dinámico, lo que significa que los tipos de las variables se comprueban durante la ejecución, no en tiempo de compilación. Además, a menudo se considera de tipado débil porque realiza conversiones de tipo implícitas de forma muy flexible (a veces de maneras inesperadas, ej. '5' == 5 es true).
    - **Comparación:** Lenguajes como Java, C#, C++, Rust o Go son de tipado estático (los tipos se comprueban en compilación, detectando errores antes). Python y Ruby también son de tipado

dinámico, pero a menudo se consideran de tipado fuerte porque no realizan tantas conversiones implícitas como JavaScript. La popularidad de TypeScript (un superconjunto de que añade tipos estáticos) muestra el deseo de la comunidad de mitigar algunos de los inconvenientes del tipado dinámico en proyectos grandes.

5. **Herencia Basada en Prototipos (Aunque con Sintaxis de Clases)**
    - **Diferencia:**Fundamentalmente, JavaScript utiliza un modelo de herencia basado en prototipos. Los objetos heredan propiedades y métodos directamente de otros objetos (sus prototipos). Aunque ES6 introdujo la sintaxis de class, es en gran medida "azúcar sintáctico" sobre el sistema de prototipos subyacente.
    - **Comparación:** La mayoría de los lenguajes orientados a objetos populares (Java, C++, C#, Python) utilizan herencia basada en clases, donde los objetos son instancias de clases que definen su estructura y comportamiento, y la herencia ocurre entre clases.
6. **Naturaleza Asíncrona Fundamental**
    - **Diferencia:** Debido a su entorno original (navegador, esperando eventos del usuario o respuestas de red) y su modelo de concurrencia de un solo hilo, la programación asíncrona (con callbacks, Promises, async/await) no es solo una característica, sino una parte *fundamental* y omnipresente del desarrollo en JavaScript moderno.
    - **Comparación:** Otros lenguajes también soportan operaciones asíncronas, pero a menudo no es tan central en el flujo de trabajo diario como lo es en JavaScript, especialmente en el desarrollo front-end y en Node.js.
7. **Versatilidad y Ecosistema (Full-Stack y Más Allá)**
    - **Diferencia:** A pesar de empezar en el navegador, se ha expandido enormemente. Con Node.js, se usa ampliamente en el backend. Con frameworks como React Native o NativeScript, se usa para desarrollo móvil. Con Electron, para aplicaciones de escritorio. Esta capacidad de usar (casi) el mismo lenguaje en toda la pila (front-end, back-end, móvil, escritorio) es bastante distintiva. Su gestor de paquetes, npm, es uno de los repositorios de software más grandes del mundo.
    - **Comparación:** Si bien otros lenguajes pueden usarse en múltiples dominios, pocos tienen la misma extensión y un ecosistema tan masivo que abarque desde el cliente hasta el servidor y más allá con tanta fluidez como JavaScript.

8. **Funciones como Ciudadanos de Primera Clase**
    - **Diferencia:** las funciones son objetos de primera clase. Esto significa que pueden ser asignadas a variables, pasadas como argumentos a otras funciones y retornadas por otras funciones. Esta característica facilita la programación funcional y patrones como los callbacks y los cierres (closures).
    - **Comparación**: muchos lenguajes soportan funciones, pero no todos las tratan con el mismo nivel de flexibilidad que JavaScript.

En resumen, lo que principalmente diferencia a JavaScript es su **rol intrínseco en la web como lenguaje del lado del cliente**, su **naturaleza interpretada con optimizaciones JIT**, su **tipado dinámico y débil**, su **modelo de concurrencia basado en eventos y asincronía** y su **modelo de herencia prototipal**. Si bien otros lenguajes pueden compartir algunas de estas características, la combinación de todas ellas y su fuerte acoplamiento con el entorno del navegador son lo que hacen a JavaScript único.

2. ¿Cuáles son todos los tipos de datos JS?

Hay 8 tipos de datos principales, agrupados en dos categorías: **primitivos** y **no primitivos** (Objetos).

Datos Primitivos
Los tipos primitivos representan valores únicos e inmutables. Cuando asignas una variable con un valor primitivo a otra variable, se copia el **valor.**

1. **String**. Se utiliza para representar texto. Se puede definir usando comillas simples ('...'), comillas dobles ("...") o backticks (`...`) para las plantillas literales (template literals).
    - Ejemplo con comillas simples

```
let nombre = 'Alice';
console.log(nombre);          // Salida: Alice
console.log(typeof nombre);   // Salida: string
```

    - Ejemplo con comillas dobles

```
let saludo = "Hola, Mundo!";
console.log(saludo);          // Salida: Hola, Mundo!
console.log(typeof saludo);   // Salida: string
```

    - Ejemplo con backticks (template literal):

```
let pais = 'España';
let mensaje = `Estoy en ${pais}.`;
console.log(mensaje);          // Salida: Estoy en España.
console.log(typeof mensaje);   // Salida: string
```

2. **Number**. Representa tanto números enteros como números de punto flotante (decimales). También incluye los valores especiales Infinity, -Infinity y NaN (Not a Number).
    - Ejemplo de entero

```
let edad = 30;
console.log(edad);          // Salida: 30
console.log(typeof edad);   // Salida: number
```

    - Ejemplo de punto flotante

```
let precio = 19.99;
console.log(precio);          // Salida: 19.99
console.log(typeof precio);   // Salida: number
```

    - Ejemplo de Infinity

```
let infinitoPositivo = 1 / 0;
console.log(infinitoPositivo);          // Salida: Infinity
console.log(typeof infinitoPositivo); // Salida: number
```

- Ejemplo de NaN

```
let resultadoErroneo = "hola" * 5;
console.log(resultadoErroneo);          // Salida: NaN
console.log(typeof resultadoErroneo);   // Salida: number
```

3. **Boolean**. Representa un valor lógico que puede ser true (verdadero) o false (falso). Se utiliza comúnmente en estructuras de control como condicionales y bucles.
   - Ejemplo

```
let estaActivo = true;
let esMayorDeEdad = false;
console.log(estaActivo);          // Salida: true
console.log(typeof estaActivo);   // Salida: boolean
console.log(esMayorDeEdad);       // Salida: false
console.log(typeof esMayorDeEdad);// Salida: boolean
```

4. **Null.** Este tipo tiene un único valor: null. Se utiliza para indicar la ausencia intencionada de cualquier valor o la ausencia de un objeto.
   - Ejemplo

```
let datoNulo = null;
console.log(datoNulo);          // Salida: null
console.log(typeof datoNulo);   // Salida: object (esto un error histórico)
```

5. **Undefined.** Representa una variable que ha sido declarada pero a la que aún no se le ha asignado un valor.
   - Ejemplo

```
let variableSinValor;
console.log(variableSinValor);          // Salida: undefined
console.log(typeof variableSinValor);   // Salida: undefined
```

6. **Symbol**. (Introducido en ECMAScript 6) Representa un valor primitivo único e inmutable. Cada valor Symbol creado es único, incluso si tienen la misma descripción. Son útiles para crear identificadores de propiedades de objeto que no colisionen.
   - Ejemplo

```
const id1 = Symbol('identificador');
const id2 = Symbol('identificador');
console.log(id1 === id2);          // Salida: false (son únicos)
console.log(typeof id1);           // Salida: symbol
```

7. **BigInt**. (Introducido en ECMAScript 11) Permite trabajar con números enteros de tamaño arbitrario, superando el límite de precisión de Number para enteros grandes. Se crea añadiendo n al final de un número entero o utilizando la función BigInt().
    - Ejemplo

```
const numeroGrande = 9007199254740991n;                    // Literal BigInt
const otroNumeroGrande = BigInt("9007199254740991"); // Usando la
función BigInt()
console.log(numeroGrande);                    // Salida: 9007199254740991n
console.log(typeof numeroGrande);             // Salida: bigint
// Puedes realizar operaciones con BigInt
const sumaGrande = numeroGrande + 1n;
console.log(sumaGrande); // Salida: 9007199254740992n
```

Dato No Primitivo

A diferencia de los primitivos, los objetos son **mutables** y se pasan por **referencia**.

8. **Object**. Es el tipo fundamental para estructuras de datos más complejas. Representa una colección de propiedades, donde cada propiedad tiene un nombre (una clave, que es una cadena o un Symbol) y un valor (que puede ser cualquier tipo de dato, incluidos otros objetos). Arrays, funciones, fechas, expresiones regulares, Mapas, Sets, etc., son todos tipos de objetos.
    - Ejemplo de Objeto Literal

```
let persona = {
  nombre: 'Carlos',
  edad: 25,
  ciudad: 'Madrid'
};
console.log(persona);          // Salida: { nombre: 'Carlos', edad: 25, ciudad:
'Madrid' }
console.log(typeof persona);          // Salida: object
```

    - Ejemplo de Array (que es un tipo de Objeto)

```
let numeros = [1, 2, 3, 4, 5];
console.log(numeros);                    // Salida: [ 1, 2, 3, 4, 5 ]
console.log(typeof numeros);             // Salida: object
console.log(Array.isArray(numeros)); // Salida: true (forma más fiable de
comprobar si es un array)
```

    - Ejemplo de Función (que también es un tipo de Objeto)

```
function saludar(nombre) {
  return "Hola, " + nombre;
}
console.log(saludar('Ana'));              // Salida: Hola, Ana
```

```
console.log(typeof saludar);          // Salida: function (Nota: typeof para funcione devuelve "function", aunque son objetos)
```

Comprender estos tipos de datos es fundamental para trabajar eficazmente con JavaScript. La naturaleza dinámica y, en el caso de los objetos, mutable, son aspectos clave a tener en cuenta al escribir código.

Comprobación con typeof

La función **typeof** nos permitie comprobar el tipo de dato de un elemento.

```
console.log(typeof undefined);        // "undefined"
console.log(typeof null);             // "object" (error histórico)
console.log(typeof true);             // "boolean"
console.log(typeof 42);               // "number"
console.log(typeof "texto");          // "string"
console.log(typeof Symbol());         // "symbol"
console.log(typeof 10n);              // "bigint"
console.log(typeof {});               // "object"
console.log(typeof []);               // "object"
console.log(typeof function(){});     // "function" (aunque es un objeto)
```

Casos Especiales:

- NaN (Not a Number):
  Es de tipo number, pero representa un valor inválido.
  - Ejemplo:

    ```
    console.log(typeof NaN); // "number"
    ```

- **Array vs Objeto**:
  Los arrays son objetos, pero se diferencian por su estructura.
  - Ejemplo:

    ```
    console.log(Array.isArray([])); // true
    console.log(Array.isArray({})); // false
    ```

Importante recordar

- Los primitivos se pasan **por valor** (copias independientes).
- Los objetos se pasan **por referencia** (modificar uno afecta a todos sus "copias").
- JavaScript realiza **coerción de tipos** automática (ej: "5" + 2 = "52").

3. ¿Cuáles son las tres funciones de String en JS?

Aquí tienes un **listado de los métodos más comunes** (no solo 3 funciones), **organizado por categorías** de los métodos de String en JavaScript, con ejemplos prácticos para cada uno.

1. Métodos de Búsqueda y Verificación

Método	Descripción	Ejemplo
.includes(substr)	Verifica si contiene una subcadena (case-sensitive).	"JavaScript".includes("Script") → true
.startsWith(substr)	Comprueba si empieza con un texto.	"Hola".startsWith("Ho") → true
.endsWith(substr)	Comprueba si termina con un texto.	"Mundo".endsWith("do") → true
.indexOf(substr)	Devuelve el índice de la primera coincidencia.	"Java".indexOf("a") → 1
.lastIndexOf(substr)	Índice de la última coincidencia.	"Banana".lastIndexOf("a") → 5
.match(regex)	Busca coincidencias con una expresión regular.	"123abc".match(/\d+/) → ["123"]
.search(regex)	Retorna el índice de la primera coincidencia con regex.	"abc123".search(/\d/) → 3

2. Métodos de Transformación

Método	Descripción	Ejemplo
.toUpperCase()	Convierte a mayúsculas.	"hola".toUpperCase() → "HOLA"
.toLowerCase()	Convierte a minúsculas.	"HOLA".toLowerCase() → "hola"
.trim()	Elimina espacios al inicio/final.	" texto ".trim() → "texto"
.trimStart() / .trimEnd()	Elimina espacios solo al inicio o final.	" hola".trimStart() → "hola"
.replace(old, new)	Reemplaza un fragmento.	"Hola".replace("a", "o") → "Holo"
.replaceAll(old, new)	Reemplaza todas las ocurrencias.	"a-a-a".replaceAll("a", "b") → "b-b-b"
.normalize()	Normaliza caracteres Unicode (ej: á → a´).	"á".normalize("NFD") → "a´"

3. Métodos de Extracción y Segmentación

Método	Descripción	Ejemplo
.slice(start, end)	Extrae una parte (índices negativos permitidos).	"JavaScript".slice(0, 4) → "Java"
.substring(start, end)	Similar a slice, pero sin índices negativos.	"JS".substring(0, 1) → "J"
.substr(start, length)	**(Obsoleto)** Extrae desde un índice con longitud.	"JS".substr(1, 1) → "S"
.split(separador)	Divide el string en un array.	"a,b,c".split(",") → ["a", "b", "c"]
.charAt(index)	Retorna el carácter en una posición.	"JS".charAt(1) → "S"
.charCodeAt(index)	Retorna el código Unicode del carácter.	"A".charCodeAt(0) → 65

4. Métodos de Validación y Comparación

Método	Descripción	Ejemplo
.localeCompare(str)	Compara dos strings (orden alfabético).	"a".localeCompare("b") → -1
.toString()	Convierte a string (útil para objetos).	(123).toString() → "123"
.valueOf()	Retorna el valor primitivo del string.	new String("JS").valueOf() → "JS"

5. Métodos de Relleno y Repetición (ES6+)

Método	Descripción	Ejemplo
.padStart(length, fill)	Rellena al inicio hasta cierta longitud.	"5".padStart(3, "0") → "005"
.padEnd(length, fill)	Rellena al final hasta cierta longitud.	"JS".padEnd(4, "!") → "JS!!"
.repeat(n)	Repite el string n veces.	"na".repeat(3) → "nanana"

6. Métodos de Iteración

Método	Descripción	Ejemplo
.concat(str1, str2)	Combina strings (equivalente a +).	"Hola".concat(" ", "Mundo") → "Hola Mundo"
[Symbol.iterator]()	Permite iterar carácter por carácter.	for (let c of "JS") { console.log(c) }

7. Métodos de Plantillas (ES6+)

Método	Descripción	Ejemplo
Template Strings	Strings con interpolación y multilínea.	`Hola ${nombre}`

5. ¿Qué es un condicional?

En JavaScript, los **condicionales** son estructuras que permiten controlar el flujo del programa basado en condiciones. Veamos qué tipos de condicionales existen.

1. if

La declaración **if** es la más básica. Ejecuta un bloque de código si la condición especificada es verdadera (true).

Sintaxis

```
if (condicion) {
  // Código a ejecutar si la condicion es verdadera
}
```

Ejemplo

```
let edad = 18;

if (edad >= 18) {

  console.log("Eres mayor de edad");

}
```

2. if - else

Evalúa una condición y ejecuta bloques de código según sea true o false.

Sintaxis

```
if (condicion) {
  // Código a ejecutar si la condicion es verdadera
} else {
  // Código a ejecutar si la condicion es falsa
}
```

Ejemplo

```
let edad = 18;

if (edad >= 18) {

  console.log("Eres mayor de edad");

} else {

  console.log("Eres menor de edad");

}
```

3. if – else – if – else

Este tipo se usa para múltiples condiciones.

Sintaxis

```
if (condicion1) {

  // Código a ejecutar si condicion1 es verdadera
```

```
} else if (condicion2) {
  // Código a ejecutar si condicion2 es verdadera
} else {
  // Código a ejecutar si ninguna de las condiciones anteriores es verdadera
}
```

Ejemplo

```
let hora = 14;
if (hora < 12) {
  console.log("Buenos días");
} else if (hora < 19) {
  console.log("Buenas tardes"); // Este se ejecuta
} else {
  console.log("Buenas noches");
}
```

4. **switch**

La declaración switch se utiliza para realizar diferentes acciones basadas en diferentes condiciones de un mismo valor o expresión. Es especialmente útil cuando tienes múltiples posibles valores para una variable y quieres ejecutar código diferente para cada uno.

Sintaxis

```
switch (expresion) {
  case valor1:
    // Código a ejecutar si expresion coincide con valor1
    break;
  case valor2:
    // Código a ejecutar si expresion coincide con valor2
    break;
  // ... más casos
  default:
    // Código a ejecutar si expresion no coincide con ningún caso
}
```

Ejemplo

```
let dia = "Lunes";
```

```
switch (dia) {
  case "Lunes":
    console.log("Inicio de semana");
    break;
  case "Viernes":
    console.log("¡Fin de semana cerca!");
    break;
  default:
    console.log("Día normal");
}
```

Importante: el break evita que se ejecuten los casos siguientes.

5. **Evaluación de "truthy" y "falsy"**

JavaScript evalúa condiciones de forma flexible. Los valores **falsy** son: false, 0, "", null, undefined, NaN.

El resto son **truthy**.

```
let nombre = "";
if (nombre) {
  console.log("Hola " + nombre);
} else {
  console.log("Nombre no definido"); // Se ejecuta esto
}
```

6. **Operadores lógicos en condicionales**

Se pueden combinar condiciones con && (AND), || (OR), ! (NOT).

```
let usuario = { loggedIn: true, rol: "admin" };
if (usuario.loggedIn && usuario.rol === "admin") {
  console.log("Acceso total permitido");
}
```

Ejemplo integrado

```
let clima = "soleado";
let hora = 15;
```

```
if (clima === "soleado" && hora < 18) {
  console.log("Puedes salir al parque");
} else if (clima === "lluvioso") {
  console.log("Mejor quédate en casa");
} else {
  console.log("Plan alternativo");
}
// Resultado: "Puedes salir al parque"
```

Diferencia clave entre if y switch

If - else	switch
Evalúa rangos o condiciones complejas.	Compara **igualdad estricta** (===) en un valor.
Más flexible.	Más legible para múltiples casos fijos.

Consejos

1. Usa switch cuando tengas más de 3 casos para una misma variable.
2. El operador ternario es genial para asignaciones simples.
3. Siempre usa {} en if para evitar errores (incluso en una línea).

6. ¿Qué es un operador ternario?

El **operador ternario** (también conocido como operador condicional ternario) en JavaScript es un atajo para una declaración if...else simple. Es el **único operador en JavaScript que toma tres operandos**.

Sintaxis

```
condicion ? expresionSiVerdadero : expresionSiFalso;
```

Componentes

1. **'condicion'** -> Una expresión que se evalúa como true o false. Va al principio.
2. **'?'** -> Separa la condición de las dos posibles expresiones de resultado.
3. **'expresionSiVerdadero'** -> El valor o la expresión que se ejecuta o devuelve si la condición es **true**. Va inmediatamente después del **'?'**.
4. **':'** -> Separa la expresión verdadera de la expresión falsa.
5. **'expresionSiFalso'** -> El valor o la expresión que se ejecuta o devuelve si la condición es **false**. Va después del **':'**.

Funcionamiento

El operador evalúa la condicion.

- Si la condicion es **true**, el operador devuelve el valor de **expresionSiVerdadero.**
- Si la condicion es **false**, el operador devuelve el valor de **expresionSiFalso**.

Ejemplo para comparar con un if...else:

- Usando **if...else**

```
let edad = 16;
let mensaje;
if (edad >= 18) {
  mensaje = "Puedes votar.";
} else {
  mensaje = "Aún no puedes votar.";
}
console.log(mensaje); // Salida: Aún no puedes votar.
```

- Usando el **operador ternario**

```
let edad = 16;
let mensaje = (edad >= 18) ? "Puedes votar." : "Aún no puedes votar.";
console.log(mensaje); // Salida: Aún no puedes votar.
```

Casos de uso

Es ideal para:

- Asignar un valor a una variable basado en una condición simple, como en el ejemplo anterior.
- Devolver un valor de una función basado en una condición simple.
- Hacer tu código más conciso cuando la lógica condicional es muy sencilla.

¿Cuándo no usarlo?

Para lógicas más complejas con múltiples else if o código extenso dentro de cada rama, es mejor usar if...else if...else o switch para mantener la legibilidad del código. Anidar operadores ternarios puede hacer que el código sea difícil de leer y entender.

En resumen, el operador ternario es una herramienta concisa para decisiones simples de "si esto es verdadero, haz A, de lo contrario, haz B".

7. ¿Cuál es la diferencia entre una declaración de función y una expresión de función?

La diferencia principal entre una declaración de función y una expresión de función en JavaScript radica en su **sintaxis** y en cómo se manejan durante la **fase de elevación** (**hoisting**) del código.

Ambas formas se utilizan para crear funciones, pero tienen comportamientos distintos.

1. **Declaración de Función (Function Declaration)**

- **Sintaxis**. Comienza con la palabra clave function, seguida del nombre de la función, paréntesis para los parámetros y llaves para el cuerpo de la función.

```
function nombreFuncion(parametros) {
  // Código
}
```

- **Hoisting**. Las declaraciones de función se elevan completamente al principio de su ámbito (scope) antes de que se ejecute el código. Esto significa que puedes llamar a una declaración de función antes de que aparezca en el código.
- **Nombre**. Siempre tienen un nombre.

Ejemplo

```
// ¡Podemos llamar a esta función ANTES de su definición en el código!
saludar("Mundo");                    // Salida: Hola, Mundo!
function saludar(nombre) {
  console.log("Hola, " + nombre + "!");
}
// También podemos llamarla después
saludar("JavaScript");               // Salida: Hola, JavaScript!
```

Explicación

JavaScript "lee" primero todas las declaraciones de función en un ámbito y las pone a disposición en la parte superior. Por eso, en el ejemplo, podemos llamar a *saludar()* antes de la línea donde se escribe la declaración function *saludar(...).*

2. **Expresión de Función (Function Expression)**

- **Sintaxis.** La función se crea como parte de una expresión. A menudo, esto significa asignar una función (que puede ser **anónima** o **nombrada**) a una variable.

Función anónima

```
let* nombreFuncion = function(parametros) {
  // Código
};
```

Función nombrada

```
let* variable = function nombreFuncion(parametros) {
  // Código
};
```

*Podemos emplear *let*, *const* o *var*

- **Hoisting.** Solo la variable a la que se asigna la función se eleva (declarada), pero la asignación del valor (la función en sí) no ocurre hasta que la línea de código donde se define la expresión es ejecutada.
- **Nombre.** Pueden ser anónimas (sin nombre, lo más común) o nombradas. Si tienen un nombre (expresión de función nombrada), ese nombre solo está disponible dentro de la propia función (para recursión, por ejemplo) y no fuera.

Ejemplo de Expresión de Función (Anónima)

```
// despedir("Usuarios");        // ReferenceError: Cannot access 'despedir' before initialization
// o si usas var: TypeError: despedir is not a function
let despedir = function(nombre) {     // La función se define en esta línea
  console.log("Adiós, " + nombre + ".");
};
// Ahora sí podemos llamarla
despedir("Usuarios");                 // Salida: Adiós, Usuarios.
```

Explicación

Con una expresión de función, la función no está completamente disponible hasta que el intérprete llega a la línea donde se define y asigna a la variable (despedir en este caso). Intentar usar la variable despedir antes de esa línea resultará en un error porque la variable existe (gracias al hoisting si se usa var, let o const), pero su valor aún no es la función esperada.

Ejemplo de Expresión de Función (Nombrada)

```
let calcularFactorial = function factorial(n) {
  if (n <= 1) {
    return 1;
  } else {
    // Podemos usar el nombre 'factorial' internamente
    return n * factorial(n - 1);
  }
};
console.log(calcularFactorial(5));      // Salida: 120
```

```
// No podemos usar el nombre 'factorial' fuera de la función
// console.log(factorial(3));          // ReferenceError: factorial is not defined
```

Explicación

El nombre *factorial* solo es accesible dentro del cuerpo de la función *calcularFactorial*. Fuera, solo se puede acceder a la función a través de la variable *calcularFactorial*. Esto es útil para recursión o para facilitar la depuración (el nombre aparece en los rastreos de pila).

Diferencias Clave en Resumen

Característica	Declaración de Función (function nombre() { ... })	Expresión de Función (let variable = function() { ... };)
Sintaxis	function palabra clave seguida de nombre	Como parte de una expresión (usualmente asignada a variable)
Hoisting	Se eleva completamente (cuerpo y definición)	Solo se eleva la variable, la definición no hasta ejecución
Llamar antes def.	Sí	No (resulta en error)
Nombre	Obligatorio	Opcional (puede ser anónima o nombrada)
Ámbito del Nombre	Disponible en el ámbito donde se declara	Si es nombrada, el nombre solo disponible dentro de la función

En general, las declaraciones de función son útiles para definir funciones de utilidad generales que puedes necesitar llamar desde cualquier parte de tu código dentro de su ámbito. Las expresiones de función son más versátiles y se usan a menudo para:

- Callbacks (funciones que se pasan como argumentos a otras funciones).
- Funciones inmediatamente invocadas (IIFEs).
- Métodos de objetos.
- Cuando necesitas que la función no esté disponible antes de cierto punto en la ejecución.

Entender el **hoisting** es clave para comprender la diferencia en el comportamiento de estas dos formas de definir funciones.

8. ¿Qué es la palabra clave "this" en JS?

La palabra clave **this** en JavaScript es uno de los conceptos que más confusión genera, ya que su valor **no es fijo**, sino que depende completamente del **contexto en el que una función es llamada**. Es decir, this se refiere al objeto que está "ejecutando" el código actual. Se puede entender que **this** es como un pronombre que cambia de significado dependiendo de quién está hablando o en qué situación se usa. Las reglas principales que determinan el valor de this son las siguientes

1. Vinculación Global (Default Binding)

Cuando una función es llamada de forma "suelta", sin un objeto asociado explícitamente, **this** generalmente se refiere al **objeto global**. En un navegador, el objeto global es **window**. En **Node.js**, es global.

- **En modo no estricto:** this es el objeto global (window o global).
- **En modo estricto ('use strict';):** this es undefined.

Ejemplos

```
// En modo no estricto (por defecto en scripts normales)
function mostrarThisNoEstricto() {
  console.log(this);
}
mostrarThisNoEstricto();       // En un navegador: [object Window], en Node.js: [object global]
// En modo estricto
function mostrarThisEstricto() {
  'use strict';
  console.log(this);
}
mostrarThisEstricto();            // Salida: undefined
```

2. Vinculación Implícita (Implicit Binding)

Cuando una función es llamada como un **método de un objeto** (usando la notación de punto . o corchetes []), this se refiere al **objeto que posee el método**.

```
const persona = {
  nombre: "Juan",
  saludar: function() {
    console.log("Hola, soy " + this.nombre);   // 'this' se refiere al objeto 'persona'
  }
};
```

```
persona.saludar(); // Salida: Hola, soy Juan
```

En este caso, saludar es un método del objeto persona. Cuando llamamos persona.saludar(), **this** dentro de saludar se vincula a persona.

3. **Vinculación Explícita (Explicit Binding)**

Puedes **forzar** el valor de **this** usando los métodos call(), apply(), o bind() que tienen las funciones.

- **call(thisArg, arg1, arg2, ...):** Llama a la función inmediatamente, estableciendo this al primer argumento (thisArg) y pasando los argumentos adicionales individualmente.
- **apply(thisArg, [argsArray]):** Similar a call(), pero los argumentos se pasan como un array.
- **bind(thisArg, arg1, ...): No** llama a la función inmediatamente. En su lugar, devuelve una **nueva función** donde this está permanentemente vinculado al thisArg que le pasaste.

Ejemplo

```
const otraPersona = {
  nombre: "María"
};
function presentar() {
  console.log("Mi nombre es " + this.nombre);
}
presentar.call(otraPersona);          // Salida: Mi nombre es María ('this' es otraPersona)
presentar.apply(otraPersona);         // Salida: Mi nombre es María ('this' es otraPersona)
const funcionPresentarMaria = presentar.bind(otraPersona); // Crea una nueva función
funcionPresentarMaria();              // Salida: Mi nombre es María (Llamamos la nueva función después, 'this' sigue siendo otraPersona)
```

4. **Vinculación new (New Binding)**

Cuando una función es invocada usando el operador new (como en los constructores), se siguen estos pasos:

- Se crea un **nuevo objeto** vacío.
- this dentro de la función constructora se vincula a **ese nuevo objeto**.
- El nuevo objeto es devuelto implícitamente (a menos que la función retorne explícitamente otro objeto).

Ejemplo

```
function Coche(marca, modelo) {
  // 'this' se refiere al nuevo objeto creado por 'new'
```

```
  this.marca = marca;
  this.modelo = modelo;
  this.mostrarInfo = function() {
    console.log("Coche: " + this.marca + " " + this.modelo);
  };
}
const miCoche = new Coche("Seat", "León");
// 'this' dentro de Coche se vinculó a 'miCoche'
console.log(miCoche.marca); // Salida: Seat
miCoche.mostrarInfo();     // Salida: Coche: Seat León ('this' dentro de mostrarInfo es 'miCoche')
```

5. **Funciones Flecha (Arrow Functions)**

Las funciones flecha (=>) **no tienen su propia vinculación de this**. En su lugar, **heredan el valor de this del ámbito (scope) léxico en el que fueron definidas**. Ignoran las reglas de vinculación implícita, explícita y new.

Esto las hace muy útiles en callbacks o métodos donde quieres que this se refiera al objeto circundante y no cambie dinámicamente.

Ejemplo

```
const contador = {
  valor: 0,
  iniciar: function() {
    // En esta función regular, 'this' se refiere al objeto 'contador'
    console.log("this en iniciar:", this); // Salida: this en iniciar: { valor: 0, iniciar: [Function: iniciar], aumentar: [Function: aumentar] }
    // Usando una función tradicional aquí, 'this' dentro del setTimeout sería global (o undefined en strict mode)
    /*
    setTimeout(function() {
      console.log("this dentro de setTimeout (func tradicional):", this); // En navegador: [object Window], en Node.js: [object Timeout] (puede variar)
      console.log("Valor:", this.valor); // Esto daría NaN o undefined.valor -> error
    }, 1000);
    */
```

```
    // Usando una función flecha, 'this' hereda el 'this' de 'iniciar' (que es 'contador')
    setTimeout(() => {
      console.log("this dentro de setTimeout (arrow func):", this); // Salida: this dentro de setTimeout (arrow func): { valor: 0, iniciar: [Function: iniciar], aumentar: [Function: aumentar] }
      this.aumentar(); // Ahora 'this' dentro de aumentar se refiere a 'contador'
      console.log("Valor actual:", this.valor); // Salida: Valor actual: 1
    }, 1000);
  },
  aumentar: function() {
    this.valor++;
  }
};
contador.iniciar();
```

En el ejemplo anterior, la función flecha dentro de setTimeout no crea su propio this. Hereda el this de la función iniciar que la contiene, que a su vez está vinculada al objeto contador (debido a la vinculación implícita contador.iniciar()).

En resumen

El valor de this en JavaScript se determina por **cómo se llama la función**, no dónde se define (excepto en el caso de las funciones flecha que son léxicas). Conocer estas reglas es esencial para entender cómo acceder a las propiedades correctas o al contexto adecuado dentro de tus funciones.