

# Deconstruyendo la Magia: Un Viaje al Interior del Motor de JavaScript

Comprendiendo los mecanismos fundamentales que todo desarrollador profesional debe dominar.

# La Superficie: El Código que Vemos y Escribimos

Todos comenzamos aquí. Escribimos variables, funciones y manipulamos el DOM. Este código es la capa visible de JavaScript.

- <> let, const, var para almacenar valores.
- ☰ Tipos de datos como Number, String, Object.
- 👤 Funciones para crear bloques de código reutilizables.
- 📅 Interactuamos con el HTML a través del DOM y los Eventos.

**Pero**, ¿qué sucede realmente cuando el motor de **JavaScript lee este código?** ¿Cómo pasa de ser texto en un archivo a una aplicación funcional? El viaje hacia el interior comienza ahora.

```
...  
// Declarar una variable  
const greeting = "Hello, World!";  
  
// Crear una función  
function sayHello(name) {  
    console.log(` ${greeting} My name is  
    ${name}.`);  
}  
  
// Llamar a la función  
sayHello("Developer");
```

# El Primer Enigma: ¿Por qué `undefined` y no un error?

## El Código

```
console.log(miVariable); // Output: undefined  
  
var miVariable = "Hola Mundo";
```

## La Pregunta



El código se lee de arriba abajo, ¿verdad? Si la variable aún no ha sido declarada, ¿por qué no obtenemos un `ReferenceError`?

La respuesta no está en el orden en que escribimos, sino en cómo el motor **prepara** nuestro código antes de ejecutarlo.

# Capa 1: Cómo el Motor Lee Nuestro Código (El Contexto de Ejecución)

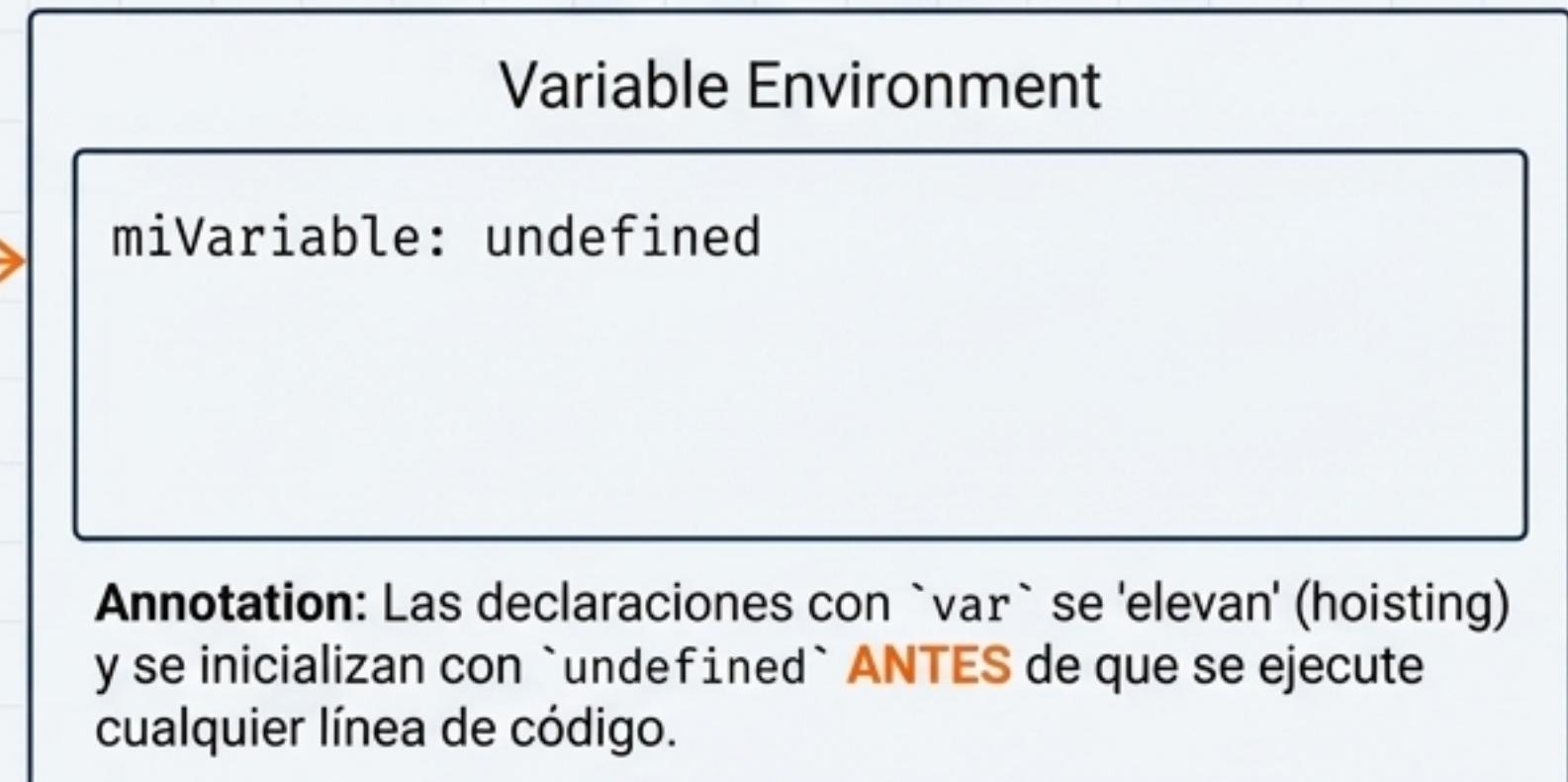
El motor de JS no ejecuta el código línea por línea al instante. Pasa por dos fases para cada bloque de código (global o de función):

1. **Fase de Creación:** El motor escanea en busca de declaraciones. Aquí ocurre el *Hoisting*.
2. **Fase de Ejecución:** El motor ejecuta el código y asigna valores.

## El Código

```
console.log(miVariable); // Output: undefined  
  
var miVariable = "Hola Mundo";
```

## Dentro del Motor - Fase de Creación



# Una Complicación: La Zona Muerta Temporal (TDZ)

## El Código

```
// console.log(miVariable); // undefined  
// ya lo vimos  
// var miVariable = "Hola";  
  
console.log(miConstante); // ReferenceError: Cannot access  
'miConstante' before initialization  
  
const miConstante = "Mundo";
```

## La Nueva Pregunta

Si el *hoisting* existe, ¿por qué `let` y `const` se comportan de manera diferente y lanzan un error? ¿Acaso no se 'elevan'?

La respuesta es sí, se elevan, pero entran en un estado especial.

### Script Scope

Temporal Dead Zone

miConstante: <uninitialized> 

# Capa 2: La Memoria del Código (Closures y Ámbito Léxico)

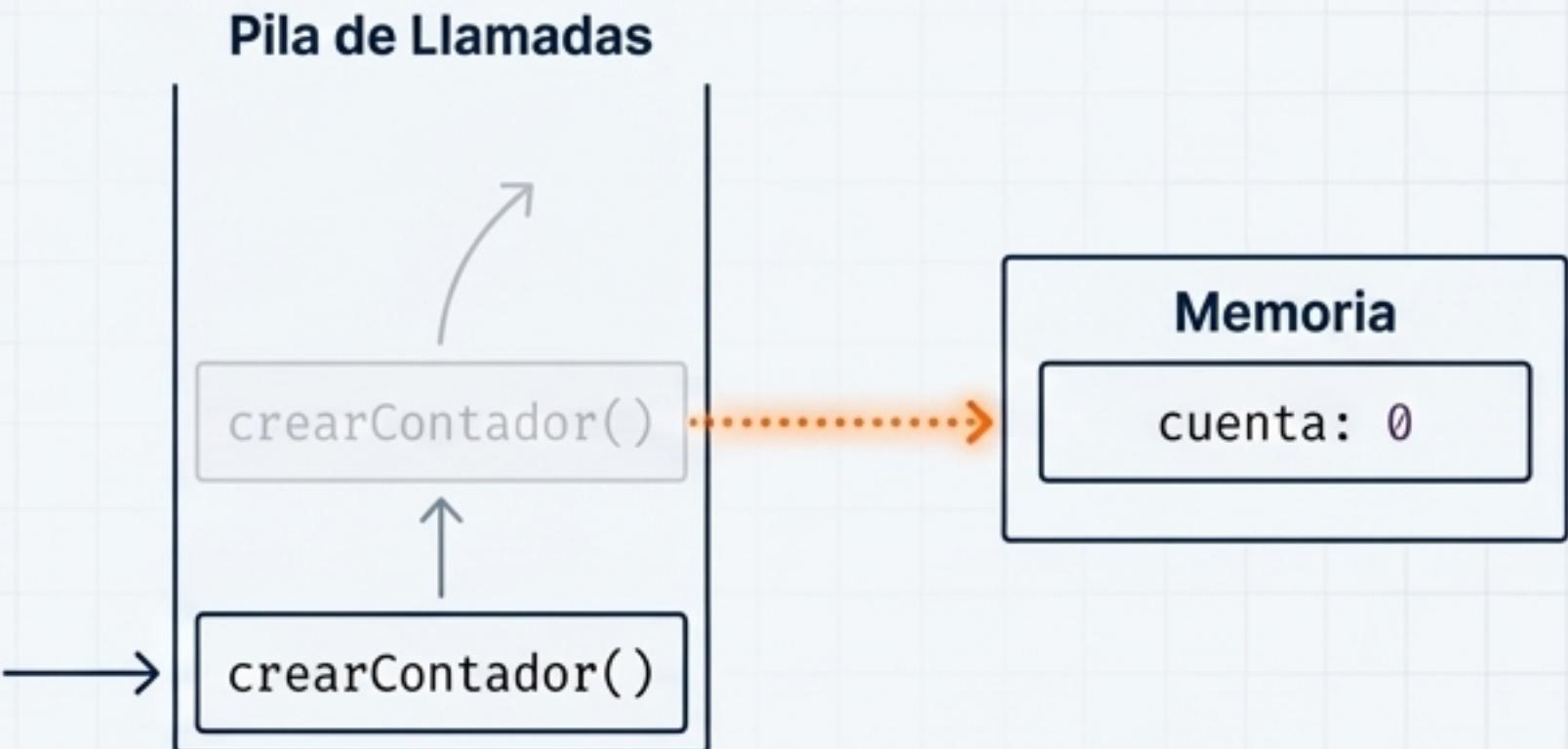
## El Código

```
function crearContador() {  
  let cuenta = 0; // Esta variable "vive" aquí  
  
  return function() {  
    cuenta++;  
    console.log(cuenta);  
  };  
  
const miContador = crearContador();  
  
miContador(); // Imprime 1  
miContador(); // Imprime 2
```

## El Enigma de la Persistencia

La función `crearContador` se ejecuta y termina, su contexto de ejecución desaparece de la Pila de Llamadas.

Entonces, ¿cómo es que la función interna (ahora `miContador`) todavía tiene acceso y puede modificar la variable `cuenta`?



# El Mecanismo: El Entorno Léxico

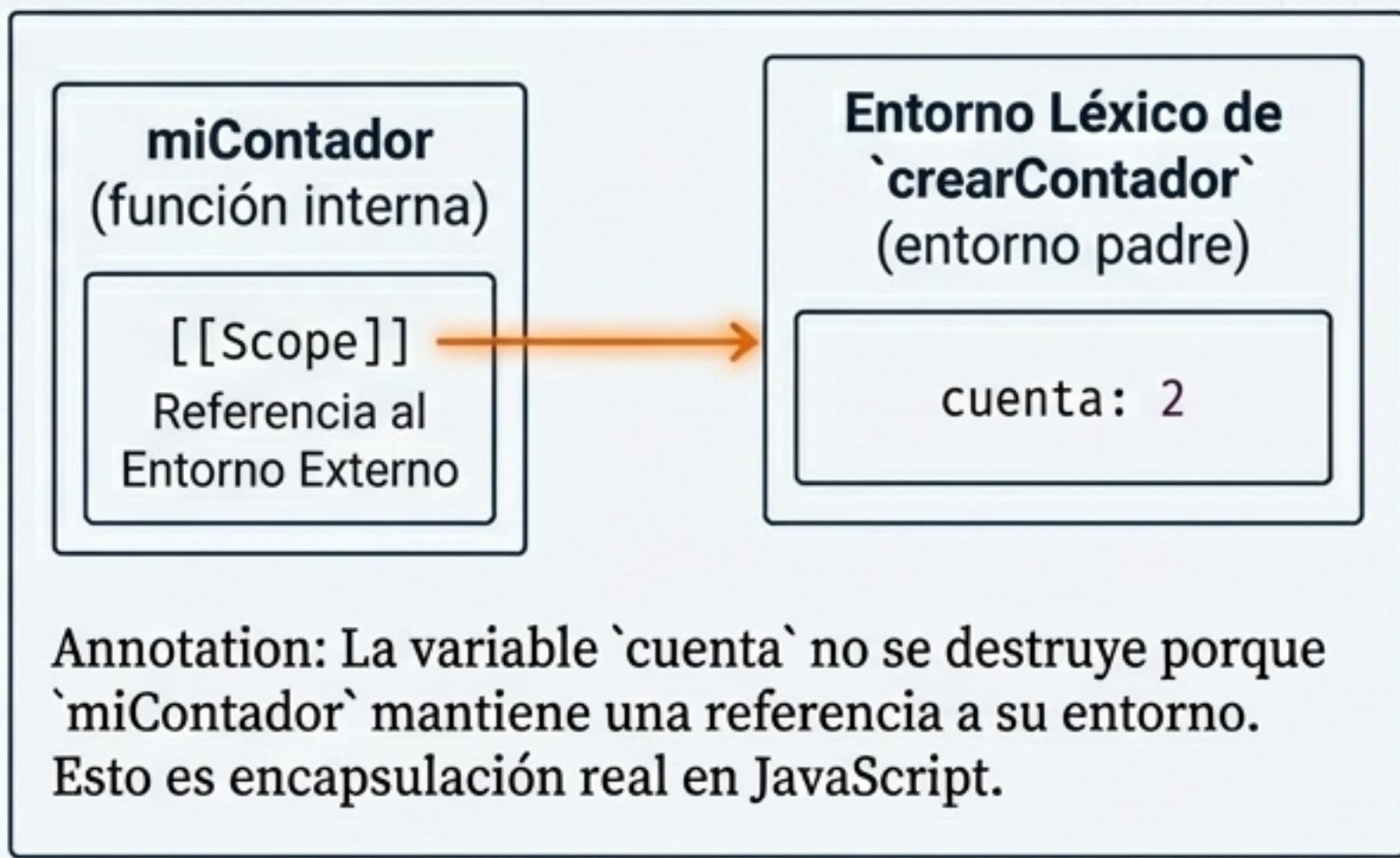
No es magia. Es el **Ámbito Léxico**. El ámbito de una función se define donde se *declara*, no donde se *ejecuta*.

- \* Cuando se crea una función, esta mantiene un enlace a su **entorno léxico** padre.
- \* Este enlace es lo que llamamos un **Closure**. La función ‘recuerda’ las variables que la rodeaban en el momento de su creación.

## El Código

```
function crearContador() {  
  let cuenta = 0;  
  
  return function() {  
    /* ... */  
  };  
}  
  
const miContador = crearContador();  
  
miContador(); // 1  
miContador(); // 2
```

## Dentro del Motor



# Capa 3: El Contexto Cambiante (`this`)

El valor de `this` no lo define dónde se crea la función, sino **cómo se la invoca** (el *call-site*). Su comportamiento cambia según 4 reglas de precedencia.

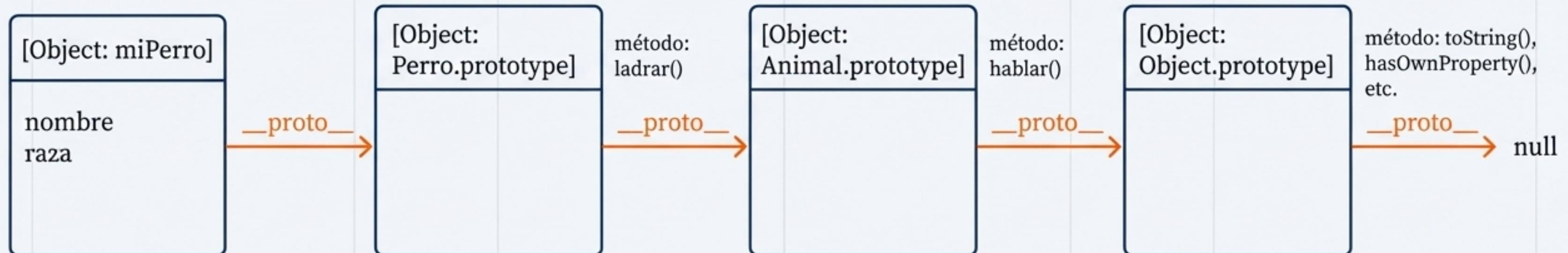
Regla de Binding	Descripción	Ejemplo
<b>1. `new` Binding</b>	Si la función se usa como constructor con `new`.	<code>const p = new Persona();</code> (`this` es la nueva instancia `p`)
<b>2. Explícito</b>	Forzamos el valor de `this` con `call`, `apply`, o `bind`.	<code>fn.call(otroObjeto);</code> (`this` es `otroObjeto`)
<b>3. Implícito</b>	Si la función se llama como método de un objeto.	<code>obj.saludar();</code> (`this` es `obj`)
<b>4. Por defecto</b>	Llamada a función sin contexto.	<code>funcionSuelta();</code> (`this` es `window` o `undefined` en modo estricto)

**Special Case:** Las **Arrow Functions** rompen estas reglas. Estas no tienen su propio `this`; lo heredan de su entorno léxico, haciéndolas predecibles.

# La Verdadera Herencia: La Cadena de Prototipos

La palabra clave `class` es "azúcar sintáctico". JavaScript no tiene clases tradicionales; tiene un sistema de herencia basado en prototipos.

- Cada objeto tiene un enlace interno (`\_\_proto\_\_`) a otro objeto, su **prototipo**.
- Cuando buscas una propiedad, JS la busca en el objeto. Si no la encuentra, sube por la **Cadena de Prototipos** hasta que la encuentra o llega a `null`.



# El Corazón del Motor: El Enigma de la Asincronía

JavaScript es de **un solo hilo (single-threaded)**. Solo puede hacer una cosa a la vez. Entonces, ¿cómo es posible este resultado?

## El Código

```
console.log('1'); .....  
setTimeout(() => console.log('2'), 0); .....  
Promise.resolve().then(() => console.log('3'));  
console.log('4'); .....
```

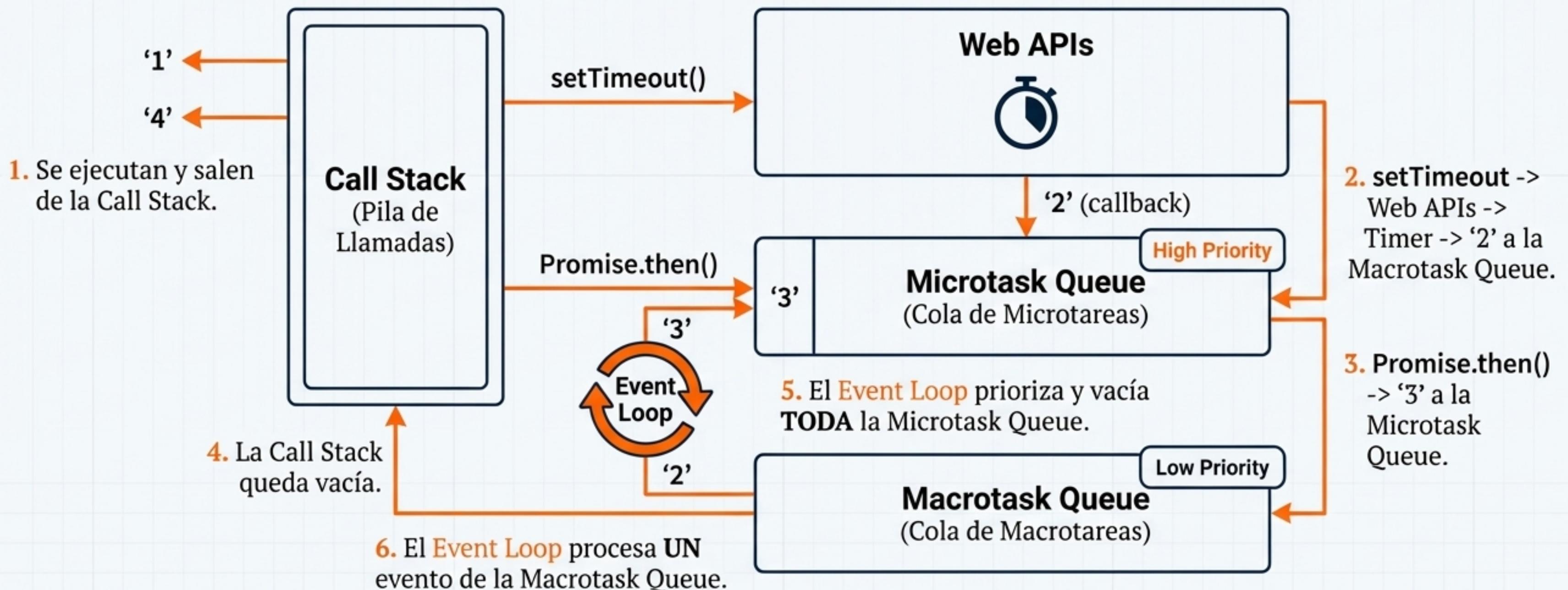
## La Consola

```
// Output en la consola:  
1  
4  
3  
2
```

La respuesta está en un modelo de concurrencia brillante que funciona fuera de la Pila de Llamadas: **el Bucle de Eventos (Event Loop)**.

# El Bucle de Eventos, Revelado

El entorno de ejecución de JS es más que solo la Pila de Llamadas.



# Comportamientos 'Extraños' que Ahora Tienen Sentido

Otros fenómenos de JavaScript dejan de ser 'raros' cuando comprendes sus reglas fundamentales.

## Example 1: Coerción de Tipos

```
"5" - 3; // → 2 (El operador '-' prefiere números)
"5" + 3; // → "53" (El operador '+' prefiere strings)
[] + []; // → "" (string vacío)
![] // → false (Un objeto es "truthy", negarlo es falso)
```

## Example 2: Paso por Valor vs. Referencia

```
let numero = 5; // Primitivo
let objeto = { prop: "original" }; // Objeto

function modificar(num, obj) {
  num = 10;
  obj.prop = "modificado";
}

modificar(numero, objeto);
// numero sigue siendo 5
// objeto.prop es "modificado"
```

No es aleatorio; sigue reglas predecibles de conversión de tipos.

Los primitivos se pasan por **valor** (una copia). Los objetos se pasan por "**referencia**" (técnicamente, la referencia se pasa por **valor**).

# Conclusión: De la Magia a la Maestría

## The Journey We Took:

- Vimos el código en la **superficie**.
- Descendimos al **Contexto de Ejecución** para entender el *hoisting*.
- Exploramos cómo los **Closures** le dan memoria al código a través del **Ámbito Léxico**.
- Decodificamos las reglas de `this` y la herencia real con **Prototipos**.
- Llegamos al **corazón del motor**: el **Bucle de Eventos** que orquesta la asincronía.

## The Core Takeaway:

El JavaScript que parece "raro" o "mágico" es el resultado de un sistema lógico y predecible.

Entender estas reglas fundamentales es la diferencia entre escribir código que funciona y ser un desarrollador que entiende por qué funciona.

