

EL CÓDICE DEL DESARROLLADOR

Libro de consulta rápida sobre el lenguaje **JavaScript**.

VOLUMEN III

CÉSAR DELGADO

© 2026 César Delgado Ruiz.

El Códice del Desarrollador.

Libro de consulta rápida sobre el lenguaje **JavaScript**.

Reservados todos los derechos. Queda prohibida la reproducción total o parcial de esta obra por cualquier medio, incluido el electrónico, sin la autorización escrita del autor.

JavaScript es una marca comercial de Oracle Corporation (antes de Sun Microsystems); el lenguaje está estandarizado por ECMA International (ECMAScript). Las marcas comerciales y nombres citados en esta obra pertenecen a sus respectivos propietarios. Este Libro es una recopilación original de referencia y no está afiliado a Oracle Corporation ni a ECMA International.

Índice general

Prólogo.....	5
Cómo usar este libro.....	7
Capítulo 1 — Tipos de datos.....	13
Capítulo 2 — Variables y constantes.....	27
Capítulo 3 — Control de flujo y bucles.....	33
Capítulo 4 — Operadores y características adicionales.....	41
Capítulo 5 — Objetos.....	49
Capítulo 6 — Arrays.....	71
Capítulo 7 — Strings.....	95
Capítulo 8 — Colecciones y utilidades de la librería estándar.....	113
Capítulo 9 — Funciones completas.....	151
Capítulo 10 — Asincronía: callbacks, promesas y async/await.....	161
Capítulo 11 — Funciones y utilidades integradas.....	173
Capítulo 12 — Clases, prototipos y campos privados.....	197
Capítulo 13 — Structs y enums.....	207

Capítulo 14 — Node.js en línea de comandos.....	211
Capítulo 15 — Módulos.....	215
Capítulo 16 — Manejo de archivos y E/S.....	221
Capítulo 17 — DOM.....	231
Capítulo 18 — Eventos y navegador.....	265
Capítulo 19 — Manejo de errores.....	335
Capítulo 20 — Node.js: módulos del núcleo.....	343
Capítulo 21 — Seguridad en aplicaciones web.....	365
Capítulo 22 — Rendimiento y memoria.....	371
Capítulo 23 — Accesibilidad.....	379
Capítulo 24 — Calidad y herramientas.....	383
Capítulo 25 — Guía de decisiones.....	389
Capítulo 26 — Recetas «Quiero...».....	395
Capítulo 27 — Gramática léxica.....	403
Capítulo 28 — Execution Contexts, Lexical Environment y This Binding	409
Capítulo 29 — Conversiones abstractas completas.....	415
Capítulo 30 — Precedencia completa de operadores.....	421
Capítulo 31 — Errores completos.....	425

Capítulo 32 — Iterator helpers y Built-in Iterator object.....	431
Capítulo 33 — Explicit Resource Management (using / await using) ..	441
Capítulo 34 — Expresiones regulares avanzadas.....	449
Capítulo 35 — JSON avanzado (rawJSON).....	453
Capítulo 36 — Metaprogramación: Well-known Symbols y Reflect....	457
Capítulo 37 — Intl completo.....	471
Capítulo 38 — Temporal API.....	485
Capítulo 39 — Changelog resumido por edición ECMAScript.....	489
Capítulo 40 — SharedArrayBuffer avanzado y Atomics.waitAsync...	493
Capítulo 41 — Upsert (getOrInsert, getOrInsertComputed).....	497
Apéndice A — Compatibilidad por entorno.....	499
Anexo A — Derechos sobre el contenido.....	501
Glosario.....	503
Índice analítico.....	509
Referencias.....	515

Prólogo

Este vademécum reúne en un único volumen la referencia esencial de JavaScript: sus tipos de datos, estructuras de datos, funciones, modelo de objetos y entornos de ejecución. Está pensado como un libro de consulta para quien ya programa o está aprendiendo y necesita encontrar rápido un método, una sintaxis o una regla de comportamiento, con ejemplos ejecutables.

El libro se organiza en capítulos con numeración continua. Los primeros capítulos cubren los fundamentos del lenguaje: tipos, variables y control de flujo. Después se abordan las estructuras de datos, las funciones y la asincronía, la programación orientada a objetos y, por último, los entornos de ejecución. Cada capítulo abre con un breve párrafo introductorio antes de entrar en tablas y fragmentos de código. Al final del volumen encontrará el apéndice de referencia, un glosario de términos, un índice analítico y las referencias oficiales del lenguaje.

Para consultar una sintaxis concreta, utilice el **índice analítico** de la parte final; para leer el lenguaje de forma ordenada, siga la secuencia de capítulos. El código de ejemplo se muestra sin traducir, como es propio del lenguaje, mientras que las explicaciones están redactadas en español con una terminología consistente a lo largo de todo el volumen.

Cómo usar este libro

- Cada **capítulo** se divide en **secciones** numeradas (1.1, 2.3, ...) y combina tablas, descripciones y ejemplos de código.
- Los **capítulos** 1–4 cubren los fundamentos del lenguaje; los capítulos 5–8, las estructuras de datos; los capítulos 9–13, las funciones, la asincronía y la POO; los capítulos 14–16, los entornos de ejecución y módulos; los capítulos 17–19, la plataforma web; el capítulo 20, Node.js en profundidad; los capítulos 21–24, la práctica profesional; los capítulos 25–26, consulta rápida.
- Los **capítulos** 27–41 cubren gramática léxica, semántica del lenguaje, conversiones abstractas, precedencia de operadores, errores (early errors y error objects), iterator helpers, explicit resource management (ES2027), expresiones regulares avanzadas, JSON rawJSON, well-known Symbols, Reflect, Intl, Temporal API (ES2027), changelog ECMAScript, SharedArrayBuffer/Atomics y Upsert (ES2026).
- Al final del volumen, el **apéndice** A reúne las tablas de referencia (compatibilidad por entorno).
- El **glosario** define los términos técnicos empleados en el libro.
- El **índice analítico** permite localizar cada método, operador o concepto por su nombre.
- Las **referencias** citan la especificación oficial (ECMA-262) y las fuentes de documentación.

Cómo se documentan los métodos y APIs

Cada método, función o API se documenta con esta estructura:

Elemento	Descripción
Qué hace	Definición concisa del propósito
Sintaxis	Forma de invocarlo con parámetros
Parámetros	Nombre, tipo y descripción de cada argumento
Valor de retorno	Qué devuelve y de qué tipo
Ejemplo	Caso de uso práctico con código
Excepciones	Errores que puede lanzar (si aplica)
Notas	Comportamiento especial, mutabilidad, alternativas

```
// Qué hace: Convierte un string a número entero
// Parámetros: string (cadena a analizar), radix (base
numérica, 2-36)
// Retorna: number (entero) o NaN si no es válido
parseInt("42px", 10); // 42
```

Parte I — Fundamentos del lenguaje

- 2. Capítulo 1 — Tipos de datos
- 3. Capítulo 2 — Variables y constantes
- 4. Capítulo 3 — Control de flujo y bucles
- 5. Capítulo 4 — Operadores y características adicionales

Parte II — Estructuras de datos

- 6. Capítulo 5 — Objetos
- 7. Capítulo 6 — Arrays
- 8. Capítulo 7 — Strings
- 9. Capítulo 8 — Colecciones y utilidades de la librería estándar

Parte III — Funciones y contexto

- 10. Capítulo 9 — Funciones completas

- 11. Capítulo 10 — Asincronía: callbacks, promesas y async/await
- 12. Capítulo 11 — Funciones y utilidades integradas
- 13. Capítulo 12 — Clases, prototipos y campos privados
- 14. Capítulo 13 — Structs y enums

Parte IV — Entorno de ejecución y módulos

- 15. Capítulo 14 — Node.js en línea de comandos
- 16. Capítulo 15 — Módulos
- 17. Capítulo 16 — Manejo de archivos y E/S

Parte V — Plataforma web

- 18. Capítulo 17 — DOM
- 19. Capítulo 18 — Eventos y navegador
- 20. Capítulo 19 — Manejo de errores

Parte VI — Node.js en profundidad

- 21. Capítulo 20 — Node.js: módulos del núcleo

Parte VII — Práctica profesional

- 22. Capítulo 21 — Seguridad en aplicaciones web
- 23. Capítulo 22 — Rendimiento y memoria
- 24. Capítulo 23 — Accesibilidad
- 25. Capítulo 24 — Calidad y herramientas

Parte VIII — Consulta rápida

- 26. Capítulo 25 — Guía de decisiones
- 27. Capítulo 26 — Recetas «Quiero...»

Parte IX — Gramática y semántica del lenguaje

- 28. Capítulo 27 — Gramática léxica
- 29. Capítulo 28 — Execution Contexts, Lexical Environment y This Binding
- 30. Capítulo 29 — Conversiones abstractas completas
- 31. Capítulo 30 — Precedencia completa de operadores
- 32. Capítulo 31 — Errores completos

Parte X — Colecciones, iteradores y asincronía

- 33. Capítulo 32 — Iterator helpers y Built-in Iterator object
- 34. Capítulo 33 — Explicit Resource Management (using / await using)

Parte XI — Módulos, regex y metaprogramación

- 35. Capítulo 34 — Expresiones regulares avanzadas
- 36. Capítulo 35 — JSON avanzado (rawJSON)
- 37. Capítulo 36 — Metaprogramación: Well-known Symbols y Reflect

Parte XII — Internacionalización, fechas y funcionalidades emergentes

- 38. Capítulo 37 — Intl completo
- 39. Capítulo 38 — Temporal API
- 40. Capítulo 39 — Changelog resumido por edición ECMAScript
- 41. Capítulo 40 — SharedArrayBuffer avanzado y Atomics.waitAsync
- 42. Capítulo 41 — Upsert (getOrInsert, getOrInsertComputed)

Apéndices y referencia

- 43. Apéndice A — Compatibilidad por entorno
- 44. Anexo A — Derechos sobre el contenido
- 45. Glosario
- 46. Índice analítico
- 47. Referencias

Capítulo 1

Tipos de datos

Parte I — Fundamentos del lenguaje · Nivel: Básico

JavaScript es un lenguaje de **tipado dinámico**: las variables no declaran su tipo y este puede cambiar en tiempo de ejecución. Los tipos se clasifican en **primitivos** (inmutables, almacenados por valor) y **no primitivos** (mutables, almacenados por referencia). Este capítulo describe cada tipo, el operador `typeof` para comprobarlos, las reglas de conversión entre ellos y los conceptos de NaN, valores falsy y comparación de objetos.

- **APIs:** `typeof`, `Number()`, `String()`, `Boolean()`, `parseInt()`, `parseFloat()`, `isNaN()`, `Number.isNaN()`
- **Prerrequisitos:** Ninguno — es el primer capítulo del libro

Trampas frecuentes: `> typeof null` devuelve "object" (bug histórico): para comprobar null usa `> valor === null`. Y `isNaN("hola")` devuelve true porque convierte el string antes de comparar: para detectar el valor NaN usa `Number.isNaN(valor)`.

1.1 Primitivos

Los tipos primitivos son los valores básicos del lenguaje, almacenados directamente y tratados por valor. Son **inmutables**: no pueden modificarse una vez creados. No tienen métodos propios ni propiedades: al acceder a uno (p. ej. `"abc".length` o `(3).toFixed()`) el motor lo envuelve temporalmente en su objeto contenedor (`String`, `Number`, ...), mecanismo llamado *auto-boxing*.

Tipo	Ejemplo	Descripción
string	<code>"hello"</code> , <code>'hello'</code> , <code>`hello`</code>	Cadena de texto UTF-16; puede usar comillas simples, dobles o template literals
number	<code>42</code> , <code>3.14</code> , <code>0xff</code> , <code>1e3</code>	Float de 64 bits IEEE 754 (doble precisión); incluye enteros, decimales, hexadecimales, <code>Infinity</code> , <code>-Infinity</code> y <code>NaN</code>
bigint	<code>12345678901234567890n</code> , <code>BigInt("123")</code>	Entero de precisión arbitraria; se declara con <code>n</code> o con <code>BigInt()</code> ; no se puede mezclar con <code>number</code> en operaciones
boolean	<code>true</code> , <code>false</code>	Valor lógico; solo tiene dos valores posibles
undefined	<code>let x;</code>	Valor por defecto de una variable declarada pero no inicializada; también se obtiene en parámetros sin valor, objetos con propiedades ausentes y funciones sin <code>return</code>
null	<code>let x = null;</code>	Ausencia de valor intencional; solo se obtiene asignando <code>null</code> explícitamente
symbol	<code>Symbol("id")</code>	Valor único e inmutable; se usa como clave de propiedad para evitar colisiones; puede tener descripción opcional

💡 **Nota:** Los primitivos se almacenan en la pila (*stack*) y se comparan por valor. Los objetos se almacenan en el montón (*heap*) y se comparan por referencia.

1.2 No primitivos

Los tipos no primitivos se guardan por referencia y agrupan otros valores. A diferencia de los primitivos, son **mutables** y se comparan por referencia (no por valor).

Tipo	Ejemplo	Descripción
object	<code>{ key: "value" }</code>	Colección de pares clave-valor; las claves son strings o symbols
array	<code>[1, 2, 3]</code>	Lista ordenada de elementos (objeto Array)
function	<code>function f() {}</code>	Objeto invocable; puede tener propiedades y métodos
Map	<code>new Map([["a", 1]])</code>	Colección de pares clave-valor donde las claves pueden ser cualquier tipo
Set	<code>new Set([1, 2, 3])</code>	Colección de valores únicos
WeakMap	<code>new WeakMap()</code>	Mapa con claves que son objetos (referencias débiles); permite garbage collection
WeakSet	<code>new WeakSet()</code>	Conjunto con elementos que son objetos (referencias débiles)
RegExp	<code>/patrón/gi</code>	Expresión regular para coincidencia de patrones en texto
Error	<code>new Error("msg")</code>	Representa un error; tiene <code>message</code> , <code>name</code> y <code>stack</code>
Date	<code>new Date()</code>	Representa fecha y hora
Promise	<code>new Promise(...)</code>	Representa una operación asíncrona completada o fallida

1.3 El operador `typeof`

`typeof` operador

Qué hace: Devuelve el tipo de un valor como string. Es la forma habitual —y segura, porque nunca lanza excepción— de obtener el tipo de un valor primitivo, con la única salvedad de `null`, que se reporta como `"object"`; para comprobar `null` usa valor `=== null`.

Sintaxis: `typeof valor`

Parámetros:

- **Parámetro:** valor
- **Tipo:** any
- **Descripción:** Cualquier valor de JavaScript

Retorna: string — Uno de los siguientes valores:

Resultado	Tipo verificado
"number"	Números (incluye NaN, Infinity, -Infinity)
"string"	Cadenas de texto
"boolean"	Booleanos
"undefined"	Undefined
"object"	Objetos, arrays, null, Date, RegExp
"function"	Funciones y clases
"symbol"	Symbols
"bigint"	BigInts

Ejemplo:

```
typeof 42;           // "number"
typeof "hola";       // "string"
typeof true;         // "boolean"
typeof undefined;    // "undefined"
typeof null;         // "object" (bug histórico)
typeof Symbol();     // "symbol"
typeof 10n;          // "bigint"
typeof {};           // "object"
typeof [];           // "object"
typeof function(){} // "function"

// Para arrays, usar Array.isArray()
Array.isArray([]);   // true
Array.isArray({});   // false
```

⚠ **Advertencia:** `typeof null` devuelve "object" por un bug histórico de la primera implementación de JavaScript. Para verificar si un valor es `null`, use `valor === null`.

1.4 Conversión de tipos

La conversión puede ser explícita, mediante los constructores `Number`, `String` y `Boolean` o las funciones `parseInt` y `parseFloat`, o implícita, mediante la coerción que aplica el lenguaje en algunas operaciones.

Number(value)

Qué hace: Convierte un valor a tipo `number`.

Sintaxis: `Number(value)`

Parámetros:

- **Parámetro:** `value`
- **Tipo:** `any`
- **Descripción:** Valor a convertir a número

Retorna: `number` — El valor numérico convertido, o `NaN` si la conversión no es posible.

Ejemplo:

```
Number("42");      // 42
Number("");         // 0
Number(true);       // 1
Number(false);      // 0
Number(null);       // 0
Number(undefined);  // NaN
Number("hola");     // NaN
Number("3.14");     // 3.14
Number("0xff");     // 255
```

Notas:

- "" (string vacío) → 0
- true → 1, false → 0
- null → 0
- undefined → NaN
- Cadenas con caracteres no numéricos → NaN

String(value)

Qué hace: Convierte un valor a tipo string.

Sintaxis: String(value)

Parámetros:

- **Parámetro:** value
- **Tipo:** any
- **Descripción:** Valor a convertir

Retorna: string — Representación en texto del valor.

Ejemplo:

```
String(42);           // "42"
String(true);         // "true"
String(false);        // "false"
String(null);         // "null"
String(undefined);    // "undefined"
String([1,2,3]);      // "1,2,3"
```

Boolean(value)

Qué hace: Convierte un valor a tipo boolean.

Sintaxis: Boolean(value)

Parámetros:

- **Parámetro:** value
- **Tipo:** any
- **Descripción:** Valor a evaluar

Retorna: boolean — true si el valor es truthy, false si es falsy.

Ejemplo:

```
Boolean(1);           // true
Boolean(0);           // false
Boolean("hola");      // true
Boolean("");          // false
Boolean(null);        // false
Boolean(undefined);  // false
Boolean(NaN);         // false
Boolean([]);          // true
```

Notas:

- Valores falsy: false, 0, -0, 0n, "", null, undefined, NaN, document.all
- Todos los demás valores son truthy (incluye objetos, arrays, funciones)

parseInt(string, radix?)

Qué hace: Analiza un string y devuelve un entero.

Sintaxis: parseInt(string, radix?)

Parámetros:

- **Parámetro:** string
- **Tipo:** string

- **Descripción:** Cadena a analizar
- **Parámetro:** radix
- **Tipo:** number (*opcional*)
- **Descripción:** Base numérica (2-36). Si se omite se interpreta 10, salvo que la cadena empiece por 0x/0X (16). Los prefijos 0o y 0b **no** se reconocen: la lectura se detiene en o/b y el resultado es 0 (parseInt("0b101") → 0)

Retorna: number — Entero parseado o NaN si no es válido.

Ejemplo:

```
parseInt("42px");      // 42 (detiene en 'p')
parseInt("0xff", 16);  // 255
parseInt("1010", 2);   // 10 (binario a decimal)
parseInt("abc");       // NaN
parseInt("");          // NaN
parseInt("42.5");      // 42 (trunca decimales)
```

Notas:

- Detiene el análisis en el primer carácter no numérico
- Si el string comienza con 0x o 0X, interpreta como hexadecimal
- El parámetro radix es recomendable siempre para evitar ambigüedades

parseFloat(string)

Qué hace: Analiza un string y devuelve un número decimal. Ignora espacios en blanco iniciales y soporta notación exponencial (e/E).

Sintaxis: parseFloat(string)

Parámetros:

- **Parámetro:** string
- **Tipo:** string
- **Descripción:** Cadena a analizar

Retorna: number — Decimal parseado o NaN si no es válido.

Ejemplo:

```
parseFloat("3.14m");    // 3.14
parseFloat("42");        // 42
parseFloat("0xff");      // 0 (solo analiza hasta 'x')
parseFloat("abc");       // NaN
parseFloat("");          // NaN
```

Notas:

- Solo analiza la parte numérica decimal
- A diferencia de parseInt, no interpreta hexadecimales
- Trunca en el primer carácter no numérico después del decimal

isNaN(value)

Qué hace: Comprueba si un valor es NaN, convirtiéndolo a número primero.

Sintaxis: `isNaN(value)`

Parámetros:

- **Parámetro:** value
- **Tipo:** any
- **Descripción:** Valor a evaluar

Retorna: boolean — true si el valor convertido a número es NaN.

Ejemplo:

```
isNaN(NaN);      // true
isNaN("hola");   // true (convierte "hola" a NaN)
isNaN(undefined); // true (convierte undefined a NaN)
isNaN(123);      // false
isNaN("123");    // false (convierte a 123)
isNaN(true);     // false (convierte a 1)
```

 **Advertencia:** Convierte el valor a número antes de evaluar, lo que puede dar falsos positivos con strings no numéricos.

Number.isNaN(value)

Qué hace: Comprueba si un valor es exactamente NaN (sin conversión previa).

Sintaxis: `Number.isNaN(value)`

Parámetros:

- **Parámetro:** value
- **Tipo:** any
- **Descripción:** Valor a evaluar

Retorna: boolean — true solo si el valor es NaN.

Ejemplo:

```
Number.isNaN(NaN);      // true
Number.isNaN("hola");   // false (no convierte)
Number.isNaN(undefined); // false (no convierte)
Number.isNaN(123);      // false
Number.isNaN("123");    // false
Number.isNaN(true);     // false
```

💡 **Buenas prácticas:** Usa `Number.isNaN()` en lugar de `isNaN()` para evitar falsos positivos con valores no numéricos.

Coerciones implícitas comunes:

Operación	Resultado	Motivo
"5" + 1	"51"	+ con string fuerza concatenación
"5" - 1	4	- fuerza conversión numérica
"5" * 2	10	* fuerza conversión numérica
"5" / 2	2.5	/ fuerza conversión numérica (como - y *)
"5" > "10"	true	Comparación lexicográfica por código Unicode: "5" (U+0035) > "1" (U+0031)

Operación	Resultado	Motivo
<code>5 > "10"</code>	<code>false</code>	"10" se convierte a 10, luego <code>5 > 10</code> es <code>false</code>
<code>"" == 0</code>	<code>true</code>	String vacío se convierte a 0
<code>null == undefined</code>	<code>true</code>	Son equivalentes en <code>==</code>
<code>null == 0</code>	<code>false</code>	<code>null</code> solo es igual a <code>null</code> y <code>undefined</code> en <code>==</code>
<code>"0" == false</code>	<code>true</code>	Ambos se convierten a 0
<code>NaN == NaN</code>	<code>false</code>	<code>NaN</code> no es igual a sí mismo (tampoco con <code>===</code>)
<code>[] == false</code>	<code>true</code>	Array vacío se convierte a 0
<code>[] == ![]</code>	<code>true</code>	<code>![]</code> es <code>false</code> , y <code>[] == false</code> es <code>true</code>
<code>!0</code>	<code>true</code>	0 es falsy, la negación lo convierte a <code>true</code>
<code>!"texto"</code>	<code>false</code>	Strings no vacíos son truthy

Valores falsy (se evalúan como `false` en contexto booleano):

Valor	Descripción
<code>false</code>	Booleano falso
<code>0</code>	Cero (entero)
<code>-0</code>	Cero negativo
<code>0n</code>	Cero BigInt
<code>""</code>	String vacío
<code>null</code>	Ausencia de valor intencional
<code>undefined</code>	Variable sin inicializar
<code>NaN</code>	Resultado de operación numérica inválida
<code>document.all</code>	Caso histórico no estándar (solo en navegadores) ⚠

💡 **Buenas prácticas:** Para verificar si un valor es truthy/falsy, úsalo directamente en un contexto booleano (`if (valor)`) o con `Boolean(valor)`. Evita usar `!!valor` por legibilidad.

Comparación de objetos:

Los objetos se comparan por referencia, no por valor:

```
// Objetos con el mismo contenido son diferentes
({ a: 1 }) === ({ a: 1 }); // false (diferentes referencias)

[1, 2] === [1, 2];        // false (diferentes referencias)
```

```
// Mismo objeto, misma referencia  
const obj = { a: 1 };  
obj === obj;           // true  
  
// Arrays son objetos  
const arr = [1, 2, 3];  
arr === arr;           // true  
arr == [1, 2, 3];      // false (diferentes referencias)  
  
// Para comparar contenido, usar JSON.stringify() o iterar  
JSON.stringify({ a: 1 }) === JSON.stringify({ a: 1 }); //  
true
```