This page was translated from English by the community. Learn more and join the MDN Web Docs community.

# Tipos de datos y estructuras en JavaScript

Todos los lenguajes de programación tienen estructuras de datos integradas, pero estas a menudo difieren de un lenguaje a otro. Este artículo intenta enumerar las estructuras de datos integradas disponibles en JavaScript y las propiedades que tienen. Estas se pueden utilizar para construir otras estructuras de datos. Siempre que es posible, se hacen comparaciones con otros lenguajes.

## Tipado dinámico

JavaScript es un lenguaje *débilmente tipado* y *dinámico*. Las variables en JavaScript no están asociadas directamente con ningún tipo de valor en particular, y a cualquier variable se le puede asignar (y reasignar) valores de todos los tipos:

JS

```
let foo = 42; // foo ahora es un número
foo = "bar"; // foo ahora es un string
foo = true; // foo ahora es un booleano
```

## Estructuras y tipos de datos

mdn web docs

- Seis **tipos de datos** primitivos, controlados por el `operador typeof`
  - Undefined: `typeof instance === "undefined"`
  - Boolean: `typeof instance === "boolean"`
  - Number: `typeof instance === "number"`
  - String: `typeof instance === "string"`

  - BigInt: `typeof instance === "bigint"`
  - Symbol: `typeof instance === "symbol"`
- Null: `typeof instance === "object"`. Tipo primitivo especial que tiene un uso adicional para su valor: si el objeto no se hereda, se muestra `null`;
- Object: `typeof instance === "object"`. Tipo estructural especial que no es de datos pero para cualquier instancia de objeto construido que también se utiliza como estructuras de datos: new `Object`, new `Array`, new `Map` (en-US), new `Set`, new `WeakMap`, new `WeakSet`, new `Date` y casi todo lo hecho con la palabra clave `new`;
- Function: una estructura sin datos, aunque también responde al operador `typeof`: `typeof instance === "function"`. Esta simplemente es una forma abreviada para funciones, aunque cada constructor de funciones se deriva del constructor `Object`.

Ten en cuenta que el único propósito valioso del uso del operador `typeof` es verificar el tipo de dato. Si deseamos verificar cualquier Tipo Estructural derivado de `Object`, no tiene sentido usar `typeof` para eso, ya que siempre recibiremos "`object`". La forma correcta de comprobar qué tipo de Objeto estamos usando es la palabra clave `instanceof`. Pero incluso en ese caso, puede haber conceptos erróneos.

## Valores primitivos

Todos los tipos, excepto los objetos, definen valores inmutables (es decir, valores que no se pueden cambiar). Por ejemplo (y a diferencia de C), las cadenas son inmutables. Nos referimos a los valores de estos tipos como "*valores primitivos*".

### Tipo `Boolean`

`Boolean` representa una entidad lógica y puede tener dos valores: `true` y `false`. Consulta Boolean y `Boolean` para obtener más detalles.

### Tipo `Null`

El tipo `Null` tiene exactamente un valor: `null`. Consulta `null` y Null para obtener más detalles.

### Tipo `Undefined`

Una variable a la que no se le ha asignado un valor tiene el valor `undefined`. Consulta `undefined` y Undefined para obtener más detalles.

### Tipo `Number`

ECMAScript tiene dos tipos numéricos integrados: **`Number`** y **`BigInt`** (ve más abajo).

El tipo `Number` es un valor en formato binario de 64 bits de doble precisión IEEE 754 (números entre -(253 - 1) y 253 - 1). Además de representar números de punto flotante, el tipo `Number` tiene tres valores simbólicos: `+Infinity`, `-Infinity` y `NaN` ("**N**ot a **N**umber" o No es un número).

Para verificar el valor disponible más grande o el valor más pequeño disponible dentro de `±Infinity`, puedes usar las constantes `Number.MAX_VALUE` o `Number.MIN_VALUE`.

> **Nota: A partir de ECMAScript 2015**, también puedes comprobar si un número está en el rango de números de punto flotante de doble precisión mediante `Number.isSafeInteger()` así como `Number.MAX_SAFE_INTEGER` y `Number.MIN_SAFE_INTEGER` (en-US).Más allá de este rango, los enteros en JavaScript ya no son seguros y serán una aproximación de punto flotante de doble precisión del valor.

El tipo `number` solo tiene un entero con dos representaciones: `0` se representa como `-0` y `+0`. (`0` es un alias de `+0`).

En la praxis, esto casi no tiene impacto. Por ejemplo, `+0 === -0` es `true`. Sin embargo, puedes notar esto cuando divides entre cero:

JS

```
> 42 / +0
Infinity
> 42 / -0
-Infinity
```

Aunque un `number` a menudo representa solo su valor, JavaScript proporciona `operadores binarios (bitwise)` (en-US).

> **Nota:** Aunque los operadores bit a bit se *pueden* usar para representar múltiples valores Booleanos dentro de un solo número usando el enmascaramiento de bits, esto generalmente se considera una mala práctica. JavaScript ofrece otros medios para representar un conjunto de valores booleanos (como un arreglo de valores booleanos o un objeto con valores booleanos asignados a propiedades con nombre). El enmascaramiento de bits también tiende a hacer que el código sea más difícil de leer, comprender y mantener.

Posiblemente sea necesario utilizar estas técnicas en entornos muy restringidos, como cuando se intenta hacer frente a las limitaciones del almacenamiento local, o en casos extremos (como cuando cada bit de la red cuenta). Esta técnica solo se debe considerar cuando sea la última medida que se pueda tomar para optimizar el tamaño.

## Tipo `BigInt`

El tipo `BigInt` (en-US) es un primitivo numérico en JavaScript que puede representar números enteros con precisión arbitraria. Con `BigInt` s, puedes almacenar y operar de forma segura en números enteros grandes incluso más allá del límite seguro de enteros para `Number` s.

Un `BigInt` se crea agregando `n` al final de un número entero o llamando al constructor.

Puedes obtener el valor más seguro que se puede incrementar con `Number` s utilizando la constante `Number.MAX_SAFE_INTEGER` . Con la introducción de `BigInt` s, puedes operar con números más allá de `Number.MAX_SAFE_INTEGER` .

Este ejemplo demuestra, dónde, incrementando el `Number.MAX_SAFE_INTEGER` devuelve el resultado esperado:

JS

```
> const x = 2n ** 53n;
9007199254740992n
> const y = x + 1n;
9007199254740993n
```

Puedes utilizar los operadores `+` , `*` , `-` , `**` y `%` con `BigInt` s, igual que con `Number` s. Un `BigInt` no es estrictamente igual a un `Number` , pero lo es en términos generales.

Un `BigInt` se comporta como un `Number` en los casos en que se convierte a `Boolean` : `if` , `||` , `&&` , `Boolean` , `!` .

Los `BigInt` no se pueden utilizar indistintamente con los `Number` . En su lugar, se lanzará un `TypeError` .

## Tipo `String`

El tipo `String` de JavaScript se utiliza para representar datos textuales. Es un conjunto de "elementos" de valores enteros sin signo de 16 bits. Cada elemento del `String` ocupa una

posición en la cadena. El primer elemento está en el índice `0`, el siguiente en el índice `1`, y así sucesivamente. La longitud de una cadena es el número de elementos que contiene.

A diferencia de algunos lenguajes de programación (tal como C), las cadenas de JavaScript son inmutables. Esto significa que una vez que se crea una cadena, no es posible modificarla.

Sin embargo, todavía es posible crear otra cadena basada en una operación en la cadena original. Por ejemplo:

- Una subcadena de la original seleccionando letras individuales o usando `String.substr()`.
- Una concatenación de dos cadenas usando el operador de concatenación (`+`) o `String.concat()`.

### ¡Ten cuidado de no "convertir a cadenas" tu código!

Puede resultar tentador utilizar cadenas para representar datos complejos. Hacer esto viene con beneficios a corto plazo:

- Es fácil construir cadenas complejas con concatenación.
- Las cadenas son fáciles de depurar (lo que ves impreso siempre es lo que está en la cadena).
- Las cadenas son el denominador común de muchas APIs (campos de entrada — `input` s—, valores de almacenamiento local, respuestas `XMLHttpRequest` cuando se usa `responseText`, etc.) y puede resultar tentador trabajar solo con cadenas.

Con las convenciones, es posible representar cualquier estructura de datos en una cadena. Esto no la convierte en una buena idea. Por ejemplo, con un separador, se podría emular una lista (mientras que un arreglo de JavaScript sería más adecuado). Desafortunadamente, cuando el separador se usa en uno de los elementos de la "lista", la lista se rompe. Se puede elegir un caracter de escape, etc. Todo esto requiere convenciones y crea una innecesaria carga de mantenimiento.

Utiliza cadenas para datos textuales. Cuando quieras representar datos complejos, *procesa* las cadenas y usa la abstracción adecuada.

## Tipo `Symbol`

Un símbolo es un valor primitivo **único** e **inmutable** y se puede utilizar como clave de una propiedad de objeto (ve más abajo). En algunos lenguajes de programación, los símbolos se denominan "átomos".

Para obtener más detalles, consulta Symbol y el contenedor de objetos `Symbol` en JavaScript.

# Objetos

En ciencias de la computación, un objeto es un valor en la memoria al que posiblemente hace referencia un identificador.

## Propiedades

En JavaScript, los objetos se pueden ver como una colección de propiedades. Con la sintaxis de objeto literal (en-US), se inicia un conjunto limitado de propiedades; luego se pueden agregar y eliminar propiedades. Los valores de propiedad pueden ser valores de cualquier tipo, incluidos otros objetos, lo que permite construir estructuras de datos complejas. Las propiedades se identifican mediante valores *clave*. Un valor *clave* es un valor de cadena o un símbolo.

Hay dos tipos de propiedades de objeto que tienen ciertos atributos: la propiedad *data* y la propiedad *accessor*.

> **Nota:** Cada propiedad tiene *atributos correspondientes*. Los atributos, internamente los utiliza el motor JavaScript, por lo que no puedes acceder a ellos directamente. Es por eso que los atributos se enumeran entre corchetes dobles, en lugar de simples.Consulta `Object.defineProperty()` para obtener más información.

## Propiedad `Data`

Asocia una clave con un valor y tiene los siguientes atributos:

#### Atributos de una propiedad `data`

| Atributo | Tipo | Descripción | Valor predeterminado |
|---|---|---|---|
| [[Value]] | Cualquier tipo de JavaScript | El valor recuperado por un captador de acceso `get` a la propiedad. | `undefined` |
| [[Writable]] | `Boolean` | Si es `false`, el [[Value]] de la propiedad no se puede cambiar. | `false` |

| Atributo | Tipo | Descripción | Valor predeterminado |
|---|---|---|---|
| [[Enumerable]] | `Boolean` | Si es `true`, la propiedad se enumerará en bucles `for...in`. Consulta también Enumerabilidad y posesión de propiedades. | `false` |
| [[Configurable]] | `Boolean` | Si es `false`, la propiedad no se puede eliminar, no se puede cambiar a una propiedad de acceso descriptor y los atributos que no sean [[Value]] y [[Writable]] no se pueden cambiar. | `false` |

**Atributos obsoletos (a partir de ECMAScript 3, renombrado en ECMAScript 5).**

| Atributo | Tipo | Descripción |
|---|---|---|
| `Read-only` | `Boolean` | Estado inverso del atributo ES5 [[Writable]]. |
| `DontEnum` | `Boolean` | Estado inverso del atributo ES5 [[Enumerable]]. |
| `DontDelete` | `Boolean` | Estado inverso del atributo ES5 [[Configurable]]. |

## Propiedad `Accessor`

Asocia una clave con una de las dos funciones de acceso (`get` y `set`) para recuperar o almacenar un valor y tiene los siguientes atributos:

| Atributo | Tipo | Descripción | Valor predeterminado |
|---|---|---|---|
| [[Get]] | Objeto `Function` o `undefined` | La función se llama con una lista de argumentos vacía y recupera el valor de la propiedad cada vez que se realiza un acceso al valor. Consulta también `get` (en-US). | `undefined` |
| [[Set]] | Objeto `Function` o `undefined` | La función se llama con un argumento que contiene el valor asignado y se ejecuta siempre que se intenta | `undefined` |

| Atributo | Tipo | Descripción | Valor predeterminado |
|---|---|---|---|
| | | cambiar una propiedad específica. Consulta también `set` (en-US). | |
| [[Enumerable]] | `Boolean` | Si es `true`, la propiedad se enumerará en bucles `for...in`. | `false` |
| [[Configurable]] | `Boolean` | Si es `false`, la propiedad no se puede eliminar y no se puede cambiar a una propiedad de datos. | `false` |

## Objetos y funciones "normales"

Un objeto JavaScript es una asociación entre *claves* y *valores*. Las claves son cadenas (o `Symbol` s), y los *valores* pueden ser cualquier cosa. Esto hace que los objetos se ajusten naturalmente a `hashmaps` .

Las funciones son objetos regulares con la capacidad adicional de ser *invocables*.

## Fechas

Al representar fechas, la mejor opción es utilizar la utilidad `Date` incorporada en JavaScript.

## Colecciones indexadas: arreglos y arreglos tipados

Los arreglos (en-US) son objetos regulares para los que existe una relación particular entre las propiedades de clave entera y la Propiedad `length` .

Además, los arreglos heredan de `Array.prototype` , que les proporciona un puñado de convenientes métodos para manipular arreglos. Por ejemplo, `indexOf` (buscando un valor en el arreglo) o `push` (agrega un elemento al arreglo), y así sucesivamente. Esto hace que el `Array` sea un candidato perfecto para representar listas o conjuntos.

Los Arreglos tipados son nuevos en JavaScript con ECMAScript 2015 y presentan una vista similar a un arreglo de un búfer de datos binarios subyacente. La siguiente tabla ayuda a determinar los tipos de datos equivalentes en C:

| Tipo | Intervalo de valores | Tamaño en bytes | Descripción | Tipo de IDL web | Tipo C equivalente |
|---|---|---|---|---|---|
| `Int8Array` (en-US) | `-128` a `127` | 1 | Dos enteros complementarios de 8 bits con signo | `byte` | `int8_t` |
| `Uint8Array` | `0` a `255` | 1 | Entero de 8-bit sin signo | `octet` | `uint8_t` |
| `Uint8ClampedArray` (en-US) | `0` a `255` | 1 | Entero de 8 bits sin signo (sujeto) | `octet` | `uint8_t` |
| `Int16Array` (en-US) | `-32768` a `32767` | 2 | Dos enteros complementarios de 16 bits con signo | `short` | `int16_t` |
| `Uint16Array` (en-US) | `0` a `65535` | 2 | Entero de 16 bits sin signo | `Short sin signo` | `uint16_t` |
| `Int32Array` (en-US) | `-2147483648` a `2147483647` | 4 | dos enteros complementarios de 32 bits con signo | `long` | `int32_t` |
| `Uint32Array` (en-US) | `0` a `4294967295` | 4 | Enteros de 32 bits sin signo | `long sin signo` | `uint32_t` |
| `Float32Array` (en-US) | `1.2 × 10-38` a `3.4 × 1038` | 4 | Número de coma flotante IEEE de 32 bits (7 dígitos significativos, p. ej., `1.1234567` ) | `float sin restricciones` | `float` |
| `Float64Array` (en-US) | `5.0 × 10-324` a `1.8 × 10308` | 8 | Número de coma flotante IEEE de 64 bits (16 dígitos significativos, p. ej., `1.123...15` ) | `doble sin restricciones` | `double` |

| Tipo | Intervalo de valores | Tamaño en bytes | Descripción | Tipo de IDL web | Tipo C equivalente |
|---|---|---|---|---|---|
| `BigInt64Array` (en-US) | `-263` a `263-1` | 8 | Dos enteros complementarios de 64 bits con signo | `bigint` | `int64_t` (long long con signo) |
| `BigUint64Array` (en-US) | `0` a `264-1` | 8 | Entero de 64 bits sin signo | `bigint` | `uint64_t` (long long sin signo) |

## Colecciones con clave: mapas, conjuntos, `WeakMaps` , `WeakSets`

Estas estructuras de datos, introducidas en ECMAScript Edition 6, toman referencias a objetos como claves. `Set` y `WeakSet` representan un conjunto de objetos, mientras que `Map` (en-US) y `WeakMap` se asocian un valor a un objeto.

La diferencia entre `Map` s y `WeakMap` s es que en el primero, las claves de objeto se pueden enumerar. Esto permite la optimización de la recolección de basura en el último caso.

Se podrían implementar `Map` s y `Set` s en ECMAScript 5 puro. Sin embargo, dado que los objetos no se pueden comparar (en el sentido de `<` "menor que", por ejemplo), el rendimiento de búsqueda sería necesariamente lineal. Las implementaciones nativas de ellos (incluidos los `WeakMap` s) pueden tener un rendimiento de búsqueda que es aproximadamente logarítmico al tiempo constante.

Por lo general, para vincular datos a un nodo DOM, se pueden establecer propiedades directamente en el objeto o usar atributos `data-*` . Esto tiene la desventaja de que los datos están disponibles para cualquier script que se ejecute en el mismo contexto. Los `Map` s y `WeakMap` s facilitan la vinculación *privada* de datos a un objeto.

## Datos estructurados: JSON

JSON (**J**ava **S**cript **O**bject **N**otation) es un formato ligero de intercambio de datos, derivado de JavaScript, pero utilizado por muchos lenguajes de programación. JSON crea estructuras de datos universales.

Consulta JSON y `JSON` para obtener más detalles.

### Más objetos en la biblioteca estándar

JavaScript tiene una biblioteca estándar de objetos integrados.

Échale un vistazo a la referencia para conocer más objetos.

## Determinación de tipos usando el operador `typeof`

El operador `typeof` te puede ayudar a encontrar el tipo de tu variable.

Lee la página de referencia para obtener más detalles y casos extremos.

## Ve también

- Colección de estructura de datos común y algoritmos comunes en JavaScript de Nicholas Zakas.
- Estructuras de datos implementadas en JavaScript

This page was last modified on 15 dic 2023 by MDN contributors.