2. TIPOS DE DATOS

Introducción a los tipos de datos

Tipado dinámico

Hay varios **tipos de datos** diferentes que podemos almacenar en variables.

- el tipo de una variable determina qué **valores** puede almacenar y qué **operaciones** se pueden realizar con ellos

JavaScript es un " **lenguaje tipado dinámicamente** ", lo que significa que, a diferencia de otros lenguajes, no necesita especificar qué tipo de datos contendrá una variable (números, cadenas, matrices, etc.).

Por ejemplo, si declara una variable y le da un valor entre comillas, el navegador trata la variable como una cadena:

```
let miCadena = 'Hola';
```

Incluso si el valor entre comillas son solo dígitos, sigue siendo una cadena, no un número, así que ten cuidado:

```
let miNumero = '500'; // Vaya, esto sigue siendo una cadena
typeof miNumero;
miNumero = 500;       // mucho mejor — ahora esto es un número
typeof miNumero;
```

Intenta ingresar las cuatro líneas anteriores en tu consola una por una y ve cuáles son los resultados.

- notarás que estamos usando un **operador** especial llamado `typeof` : esto devuelve, como un `String`, el tipo de datos de la variable que escribes después.

  `(typeof (typeof 23) === 'string')` es `true`

- la primera vez que se llama, debe devolver `'string'` (= **cadena de texto** = **cadena de caracteres**), ya que en ese momento la variable `miNumero` contiene una cadena, **'500'** .
- echa un vistazo y ve lo que devuelve la segunda vez que lo llamas.

Tipos de datos

El último estándar ECMAScript define 8 tipos de datos:

- 7 tipos de datos que son **primitivos** :
  - `null`
    - una palabra clave especial que denota un valor null.
    - debido a que JavaScript distingue entre mayúsculas y minúsculas, `null` no es lo mismo que `Null`, `NULL` o cualquier otra variante
  - `undefined`
    - una propiedad de nivel superior cuyo valor no está definido.
  - `Boolean`
    - tiene uno de dos valores `true` o `false`
  - `Number`
    - utilizado para todos los valores numéricos (**entero** y **punto flotante**) excepto para números enteros *muy* grandes
    - un número entero o de coma flotante.
    - por ejemplo: `42` o `3.14159` .
  - `BigInt`
    - un entero con **precisión arbitraria**.
    - por ejemplo: `9007199254740992n` .
  - `String`
    - una secuencia de caracteres que representan un valor de texto.
    - por ejemplo: `"Hola"` .
  - `Symbol`
    - un tipo de datos cuyas instancias son únicas e inmutables.
- `Object`
  - todo lo demás se conoce como `Object`
  - los tipos de objetos comunes incluyen:
    - `Function`
    - `Array`
    - `Date`
    - `RegExp`
    - `Error`

Aunque estos tipos de datos son relativamente pocos, te permiten realizar operaciones útiles en tus aplicaciones.

- las **funciones** (que también son objetos) son los otros elementos fundamentales del lenguaje.
- mientras que las funciones son técnicamente un tipo de objeto, puedes pensar en
  - objetos: como contenedores con nombre para **valores,**
  - funciones: como procedimientos (= miniprogramas) que tu programas puede realizar .
    - las funciones no son estructuras de datos especiales en JavaScript: son solo un tipo especial de objeto al que se puede **llamar** (o **invocar** ***call*** / ***invoke***)

Todos los lenguajes de programación tienen **incorporados** (***built-in***):

- **tipos de datos**
- **estructuras de datos**
- y ambos pueden usarse para construir otras estructuras de datos.

Tipado de datos dinámico y tipado de datos débil

Observación
Términos y traducción necesarias para entender este apartado: hablando de tipos de datos:

promote = **promocionar**
coerce = **cohercionar**, **promocionar forzosamente**: normalmente se utiliza un método constructor o función (a diferencia de otros lenguajes como PHP y Java que se puede utilizar un **operador de cast** (molde): `(int)` `(bool)` ...)
conversion = **conversión**, **promoción implícita** (no forzada: lo hace automáticamente el lenguaje siguiendo unas normas internas)

1) Tipado dinámico

JavaScript es un **lenguaje dinámico** con **tipos dinámicos**:

- las **variables** en JavaScript no están directamente asociadas con ningún **tipo** de valor en particular
- cualquier variable puede tener asignados (y reasignados) valores de todos los tipos:

```
let foo = 42;      // foo ahora es un número
foo = "barra";     // foo ahora es una cadena
foo = true;        // foo ahora es un booleano
```

2) Coerción/conversión de tipos

JavaScript también es un **lenguaje débilmente tipado**, lo que significa que permite la **conversión implícita de tipos** cuando una operación involucra **tipos no coincidentes**, en lugar de **arrojar errores** (***throw errors***) de tipo.

```
const foo = 42;                // foo es un numero
const resultado = foo + "1";
// JavaScript promociona implícitamente a foo a una cadena,
//  por lo que se puede concatenar con el otro operando
console.log(resultado);        // "421"
```

Coherciones/conversiones de tipo implícito

- son muy prácticas, pero pueden convertirse en un arma de doble filo si la programadora o programador no tenía la intención de hacer la conversión, o tenían la intención de convertir de otra forma (por ejemplo, cadena a número en lugar de número a cadena).
- para `Symbol`s y `BigInt`s, JavaScript ha rechazado intencionalmente ciertas **conversiones de tipos implícitas** .

Valores primitivos

Todos los tipos excepto `Object` definen valores **inmutables** representados directamente en el nivel más bajo del lenguaje.

- nos referimos a valores de estos tipos como ***valores primitivos*** .
- todos los tipos primitivos pueden ser probados por el operador `typeof`
- `typeof null` devuelve "object", por lo que tenemos que usar `===` `null` para probar si una variable es `null` .
- todos los tipos primitivos, excepto `null` e `undefined`, tienen sus correspondientes tipos de **objeto envoltorio (wrapper object)**, que proporcionan métodos útiles para trabajar con los valores primitivos.
  - por ejemplo, el objeto `Number` proporciona métodos como `toExponential()` .

```
(34).toExponencial(3);
```

- cuando se accede a una propiedad en un valor primitivo, JavaScript **envuelve** automáticamente el valor en el objeto contenedor correspondiente y accede a la propiedad en el objeto en su lugar.
- sin embargo, acceder a una propiedad en `null` o `undefined` arroja una excepción `TypeError`, que requiere la introducción del **operador de encadenamiento opcional** `?.` produciendo entonces `undefined`

```
const objeto = null;
objeto.miDato;
objeto?.miDato;
objeto.noExiste.campo;
objeto.noExiste?.campo;
```

Nombre de Tipo Primitivo	`typeof` devuelto	objeto wrapper (envoltorio)
null	`'object'`	N / A
undefined	`'undefined'`	N / A
Boolean	`'boolean'`	`Boolean`
Number	`'number'`	`Number`
BigInt	`'bigint'`	`Bigint`
String	`'string'`	`String`
Symbol	`'symbol'`	`Symbol`

Tipo `null`

El **tipo Null** está compuesto por exactamente un valor: `null` .

`null` se usa con mucha menos frecuencia que `undefined` en el lenguaje JavaScript.

- el lugar más importante es el final de la **cadena de prototipos**
- posteriormente, los métodos que interactúan con los **prototipos,** como `Object.getPrototypeOf()`, `Object.create()`, etc., aceptan o devuelven **null** en lugar de `undefined` .
- `null` es una **palabra clave,** pero `undefined` es un **identificador** normal que resulta ser una **propiedad global** .
- en la práctica, la diferencia es menor, ya que undefined no debe **redefinirse** ni **sombrearse** (***shadow***).

Tipo `undefined`

El **tipo** `undefined` está compuesto por exactamente un valor: `undefined` .

- conceptualmente :
  `undefined` indica la ausencia de un ***valor,*** mientras que `null` indica la ausencia de un ***objeto***
  (que también podría constituir una excusa para `typeof null === 'object'`).
- es JavaScript normalmente si una variable, función o propiedad de objeto no tiene una valor asignado, devuelve `undefined`:
  - una declaración `return` sin valor (`return;`) implícitamente devuelve `undefined` .
  - acceder a una propiedad de objeto inexistente (`obj.noExisto`) devuelve `undefined` .
  - una declaración de variable sin inicialización (`let x;`) implícitamente inicializa la variable a `undefined` .
  - muchos métodos, como `Array.prototype.find()` y `Map.prototype.get()`, devuelven `undefined` cuando no se encuentra ningún elemento.

Tipo de `Symbol`

Un `Symbol`

- es un valor primitivo **único** e **inmutable**
- puede usarse como la **clave de una propiedad de** `Objeto` (ver más abajo).
- en algunos lenguajes de programación, los `Symbol`s se denominan "átomos".
- el propósito de los `Symbol`s es crear **claves únicas para propiedades de objetos** que garanticen que no entren en conflicto con claves de objetos en otros scripts que estén cargados al mismo tiempo que el nuestro.

Literales

Literales

Los **literales**

- representan **valores** en JavaScript.
- estos son valores fijos, no variables, que *literalmente* proporcionas en tu script.

Booleanos

El **tipo Boolean** representa una entidad lógica y solo tiene dos posibles valores: `true` y `false`.

Los valores booleanos generalmente se usan para **operaciones condicionales,** incluidas

- **operador ternario:** `(condicion boolenana) ? acción1 : acción2`
- **sentencias de decisión**: `if` , `if ...else`
- **sentencias de repetición/iteración** (**bucles**): `while`, `do...while`, `for`
- **parámetros de funciones** y **valores de retorno de funciones**
- **expresiones booleanas**
- etc.

Los valores primitivos booleanos son valores verdadero/falso: pueden tener dos valores, `true` o `false` .

- estos se utilizan generalmente para probar una **condición,** después de lo cual el código se ejecuta según corresponda.
- así que por ejemplo, un caso simple sería:

```
let estoyVivo = true;
```

Otro ejemplo más real sería:

```
let condicion = 6 < 3;
```

Esto es usar el **operador** "**menor que**" (`<`) para probar si 6 es menor que 3.

- como era de esperar, devuelve `false`, ¡porque 6 no es menor que 3!

Valores primitivos booleanos y objetos booleanos

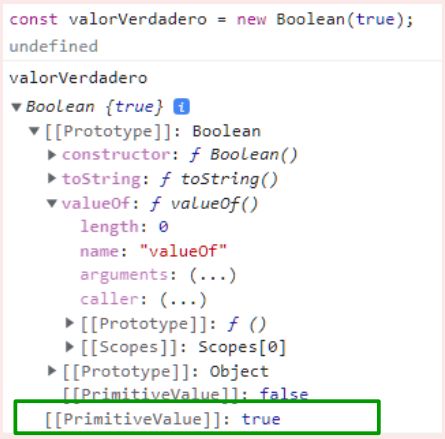
No confundas los conceptos de

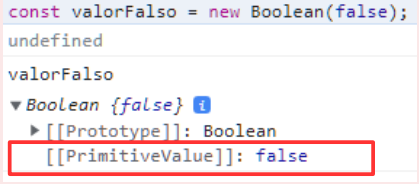
- **valor primitivo booleano:** `true` y `false`
- con los valores **true** y **falso** de un objeto booleano.

Obervación importate:

Es diferente

- valor Boolean **true** que valor primitivo booleano `true`:
  - el valor Boolean **true**: es una propiedad de un objeto `Boolean`, el cual tiene dentro un valor `[[PrimitiveValue]]`: `true`
- valor Boolean **false** que valor primitivo booleano `false`:
  - el valor Boolean **false**: es una propiedad de un objeto `Boolean`





En primer lugar, tenemos que tener claro que:

- `Boolean(true)`: en este caso invocamos el **constructor** `Boolean` y nos devuelve un valor primitivo `true`
- `Boolean(false)`: en este caso invocamos el constructor `Boolean` y nos devuelve un valor primitivo `false`
- `new Boolean(true)`: en este caso creamos un **objeto Boolean**, que devuelve una **referencia a un objeto** que contiene como valor un **true** (lo cual es distinto a `Boolean(true)`)
- `new Boolean(false)`: en este caso creamos un objeto Boolean, que devuelve una **referencia a un objeto** que contiene como valor un **false**
- todo objeto evalúa en una condición a `true` salvo `null`, que evalúa a `false`.

Cualquier objeto, incluido un objeto booleano cuyo valor es **false**, se evalúa como **true** cuando se pasa a una declaración condicional

- por ejemplo, la condición en la siguiente instrucción `if` se evalúa como verdadera :

```
const x = new Boolean(false);
if (x) {
   // este código se ejecutará seguro
}
```

- pero estas no:

```
const x = Boolean(false);
if (x) {
   // este código NO se ejecutará seguro
}

const x = false;
if (x) {
   // este código NO se ejecutará seguro
}
```

No uses el constructor `Boolean()` con `new` para convertir un valor no booleano en un valor booleano: debes usar uno de los 2 siguientes métodos:

- `Boolean` como una función
- un **doble operador lógico NO** (`!!`) en su lugar:

```
const buena = Boolean(expresión);                    // utilizar este
const bueno2 = !!(expresión);                        // o esto
const noTiraEnCondiciones = new Boolean(expresión);  // ¡No uses esto!
```

Si especificas cualquier objeto, incluido un objeto booleano cuyo valor es `false`, como valor inicial de un objeto booleano, el nuevo objeto booleano tiene un valor de **true** si se utiliza como una condición.

```
const miFalso = new Boolean(false);        // valor inicial de falso, miFalso = Boolean {false}
```

```
const miFalso = new Boolean(false);
undefined
miFalso;
▼ Boolean {false} ℹ
  ▶ [[Prototype]]: Boolean
    [[PrimitiveValue]]: false
```

```
const g = Boolean(miFalso);                // valor inicial de verdadero
```

```
const g = Boolean(miFalso);
undefined
typeof g;
'boolean'
g
true
```

```
const miCadena = new String('Hola');       // objeto de cadena
const s = Boolean(miCadena);               // valor inicial de verdadero
```

```
const miCadena = new String('Hola');
const s = Boolean(miCadena);
undefined
s;
true
typeof s;
'boolean'
```

		Tipos primitivos				Objetos	
		`true`	`false`	`Boolean(true)`	`Boolean(false)`	`new Boolean(true)`	`new Boolean(false)`
`==`	`true`	true	false	true	false	true	false
	`false`		true	false	true	false	true
	`Boolean(true)`			true	false	true	false
	`Boolean(false)`				true	false	true
	`new Boolean(true)`					false	false
	`new Boolean(false)`						false
`===`	`true`	true	false	true	false	false	false
	`false`		true	false	true	false	false
	`Boolean(true)`			true	false	false	false
	`Boolean(false)`				true	false	false
	`new Boolean(true)`					false	false
	`new Boolean(false)`						false

valores falsy	`==`		`===`	
	`true`	`false`	`true`	`false`
`""` (cadena vacía)	false	true	false	false
`"no vacía"`	false	false	false	
`0`	false	true	false	false
`+0`	false	true	false	false
`-0`	false	true	false	false
`+0.0`	false	true	false	false
`-0.0`	false	true	false	false
`+Infinity`	false	false	false	false
`-Infinity`	false	false	false	false
`NaN`	false	false	false	false

Number

Puede almacenar números en variables, ya sean números enteros como 30 o **números decimales** como 2.456 (el punto . es la coma en España) (también llamados números **flotantes** o **de punto flotante**).

- no necesita **declarar tipos de variables** en JavaScript, a diferencia de otros lenguajes de programación.
- cuando le das a una variable un valor numérico, no incluyas comillas:

```
let miEdad = 17;
```

En JavaScript:

- los números se implementan en **formato binario de 64 bits de doble precisión IEEE 754**, es decir:
  - exponente:
    - entre: 2^{-1022} o aproximadamente $\pm 10^{-308}$
    - y: 2^{+1023} o aproximadamente $\pm 10^{+308}$,
  - con una **precisión numérica** de unos 53 bits (($2^{-53} \approx 1.11 \times 10^{-16}$) (1bit para signo, 52 bits = mantisa, 11 bits = exponente)
  - ver para ejemplos prácticos:
    - https://babbage.cs.qc.cuny.edu/IEEE-754/
    - https://babbage.cs.qc.cuny.edu/ieee-754.old/64bit.html

- los valores enteros hasta $\pm 2^{53} - 1$ se pueden representar exactamente.

- Los valores fuera de rango se convierten automáticamente a:
  - Valores positivos mayores que `Number.MAX_VALUE` se convierten en `+Infinity` .
  - Valores positivos menores que `Number.MIN_VALUE` se convierten en `+0` .
  - Valores negativos menores que `-Number.MAX_VALUE` se convierten en `-Infinity` .
  - Valores negativos mayores que `-Number.MIN_VALUE` se convierten en `-0` .

- `+Infinity` e `-Infinity` se comportan de manera similar al infinito matemático, pero con algunas pequeñas diferencias

- para valores concretos consultar las propiedades de Number:
  - `Number.MIN_VALUE`
  - `Number.MAX_VALUE`
  - `Number.MIN_SAFE_INTEGER`
  - `Number.MAX_SAFE_INTEGER`

Además de poder representar **números de punto flotante,** el tipo de número tiene tres **valores simbólicos** :

- `+ infinito,`
- `- Infinito,`
- `NaN` (no-un-número).
- +0
- -0

El **tipo** `de Number` solo puede almacenar con seguridad números enteros en el rango:

- desde: $-(2^{53} - 1)$ (`Number.MIN_SAFE_INTEGER`)
- a: $2^{53} - 1$ (`Number.MAX_SAFE_INTEGER`)
  - fuera de este rango, JavaScript ya no puede representar números enteros de manera segura ; en su lugar, estarán representados por una aproximación de punto flotante de doble precisión .
  - puedes verificar si un número está dentro del rango de enteros seguros usando `Number.isSafeInteger()` .

El **tipo Número** tiene un valor (sólo este caso) con múltiples representaciones:

- 0 se representa como -0 y +0 (donde 0 es un alias para +0).
- en la práctica, casi no hay diferencia entre las distintas representaciones; por ejemplo, `+0 === -0` es `true` .
- sin embargo, puedes notar esto cuando divides por cero:

```
console.log(42 / +0); // Infinity
console.log(42 / -0); // -Infinity
```

`NaN` (" **No un Número** ") es un tipo especial de valor numérico que normalmente se encuentra cuando el resultado de una operación aritmética no se puede expresar como un número.

- también es el único valor en JavaScript que no es igual a sí mismo.
- ejemplos: 0/0 , `undefined` / 3, ...

Aunque un número es conceptualmente un "valor matemático" y siempre está implícitamente codificado en coma flotante, JavaScript proporciona **operadores bit a bit : al aplicar operadores bit a bit, el número se convierte primero en un entero de 32 bits** .

Importante:

`Number` opera por defecto con **aritmética de coma flotante de 64 bits**.

Si queremos realizar **operaciones en aritmética entera**, debemos usar la sintaxis siguiente:

- si sabemos que la *operación* va a resultar en números positivos en el rango de 0 a 2^{32} (0 a 4.294.967.296)

```
( (operación)>>>0 )
```

- si operamos con enteros que pueden dar como resultado un número negativo en el rango de -(2^{31}) a (2^{31}-1) (-2.147.483.678 a 2.147.483.677)

```
( (operación)|0 )
```

Cada *operación*

- debe ser una operación aritmética básica (adición, substracción, división, multiplicación, módulo, ...)
- si hay varias operaciones se realizará así:

```
(  ( (operación1)>>>0 ) +  ( (operación2)>>>0 )  + ( (operación3)>>>0 ) )>>>0)

(  ( (operación1)|0 ) +  ( (operación2)|0 )  + ( (operación3)|0 ) )|0)
```

Literales enteros

Literales numéricos en JavaScript:

- dispone de literales enteros en diferentes **bases,** así como **literales de punto flotante** en base 10.
- puedes usar 4 tipos de literales numéricos enteros: decimal, binario, octal y hexadecimal.

Técnicamente los literales numéricos son "sin signo":

- ten en cuenta que la especificación del idioma requiere que los literales numéricos no tengan signo.
- sin embargo, un fragmento de código como `-123.4` están bien, siendo interpretado como un **operador unario** `-` aplicado al literal numérico `123.4`

Los literales enteros y `BigInt` se pueden escribir en **decimal** (base 10), **hexadecimal** (base 16), **octal** (base 8) y **binario** (base 2).

- Un literal **entero *decimal*** es una secuencia de dígitos sin un `0` (cero) inicial.
- Un `0` (cero) inicial en un literal entero, o un `0o inicial` (o `0O`) indica que está en ***formato octal*** .
  - Los literales enteros octales pueden incluir solo los dígitos `0` – `7` .
- Un `0x` (o `0X`) inicial indica un **literal entero *hexadecimal*** .
  - Los enteros hexadecimales pueden incluir dígitos (`0` – `9`) y las letras `a` – `f` y `A` – `F`.
  - (El caso de un carácter no cambia su valor. Por lo tanto: `0xa` = `0xA` = `10` y `0xf` = `0xF` = `15`).
- Un `0b inicial` (o `0B`) indica un **literal entero *binario*** .
  - Los literales enteros binarios solo pueden incluir los dígitos `0` y `1` .
- un sufijo `n` en un literal entero indica que es un **literal** `BigInt`
  - el literal BigInt puede usar cualquiera de las bases anteriores.
  - tenga en cuenta que la sintaxis octal con cero inicial como `0123n` no está permitida, pero `0o123n` está bien.

Algunos ejemplos de literales enteros son:

```
0, 117, 123456789123456789n                 (decimal, base 10)
015, 0001, 0o777777777777n                  (octal, base 8)
0x1123, 0x00111, 0x123456789ABCDEFn         (hexadecimal, "hex" o base 16)
0b11, 0b0011, 0b1110100101010101010n        (binario, base 2)
```

Numeros decimales

```
1234567890
42

// Precaución al usar ceros a la izquierda:
0888 // 888 analizado como decimal, pues tiene el dígito 8 que no pertenece a base octal
0777 // analizado como octal en modo no estricto (511 en decimal)
```

Ten en cuenta que **los literales decimales** pueden comenzar con un cero (`0`) seguido de otro dígito decimal, pero si cada dígito después del `0` inicial es menor que 8, el número se analiza como un **número octal**.

Numeros binarios

La sintaxis de los números binarios

- utiliza un cero inicial seguido de una letra latina "B" en minúsculas o mayúsculas (`0b` o `0B`).
- si los dígitos después de `0b` no son 0 o 1, se genera el siguiente `SyntaxError` : "Faltan dígitos binarios después de 0b".

```
const FLT_SIGNBIT  = 0b10000000000000000000000000000000; // 2147483648
const FLT_EXPONENT = 0b01111111100000000000000000000000; // 2139095040
const FLT_MANTISSA = 0B00000000011111111111111111111111; // 8388607
```

Números octales

La sintaxis estándar para **los números octales** es prefijarlos con `0o` . Por ejemplo:

```
const a = 0o10; // 8
```

También hay una sintaxis heredada para los números octales, al anteponer el número octal con un cero: `0644 === 420` y `"\045" === "%"` . Si los dígitos después del `0` están fuera del rango de 0 a 7, el número se interpretará como un número decimal.

```
const n = 0755; // 493
const m = 0644; // 420
```

El modo estricto prohíbe esta sintaxis octal.

Números hexadecimales

La sintaxis de números hexadecimales utiliza un cero inicial seguido de una letra latina "X" en minúsculas o mayúsculas (`0x` o `0X`).

- si los **dígitos** después de 0x están fuera del rango (0123456789ABCDEF), se genera el siguiente `SyntaxError` : "El identificador comienza inmediatamente después del literal numérico".

```
0xFFFFFFFFFFFFFFFF // 295147905179352830000
0x123456789ABCDEF  // 81985529216486900
0XA                // 10
```

Exponenciación

```
1E3     // 1000
2e6     // 2000000
0.1e2   // 10
```

Literales de punto flotante

Un **literal de punto flotante** puede tener las siguientes partes:

- un **entero decimal sin signo**
- un **punto decimal** (`.`),
- una **fracción** (otro número decimal),
- un **exponente**
  - la parte del exponente es una `e` o una `E` seguida de un número entero, que se puede **firmar** (precedido por `+` o `-`).
- un literal de punto flotante debe tener al menos un dígito y un punto decimal o `e` (o `E`).

Más sucintamente, la sintaxis es:

```
[digitos].[digitos][(E|e)[(+|-)]digitos]
```

Por ejemplo:

```
3.1415926
.123456789
3.1E+12
.1e-23
```

BigInts

Una deficiencia de los valores numéricos es que solo tienen 64 bits.

- en la práctica, debido al uso de la codificación IEEE 754, no pueden representar ningún número entero mayor que `Number.MAX_SAFE_INTEGER` (que es 2^{53} - 1) con precisión.
- para resolver la necesidad de codificar datos binarios e interoperar con otros lenguajes que ofrecen enteros grandes como `i64` (enteros de 64 bits) e `i128` (enteros de 128 bits), JavaScript también ofrece otro tipo de datos para representar ***enteros arbitrariamente grandes*** : `BigInt` .
- elegir entre `BigInt` y el `Number` depende de su caso de uso y el rango de tu entrada a procesar:
  - la precisión de `Number` debería ser capaz de adaptarse a la mayoría de las tareas diarias, y los BigInts son los más adecuados para manejar datos binarios.

`Math` no se pueden usar en valores BigInt

El **tipo** `BigInt` es un **primitivo numérico** en JavaScript que puede representar números enteros con **una magnitud arbitraria** .

- con BigInts, puede almacenar y operar con seguridad en números enteros grandes incluso más allá del límite de número entero seguro (`Number.MAX_SAFE_INTEGER`) para `Number`s.
- un BigInt se crea agregando `n` al final de un número entero o llamando a la función `BigInt()`.

```
const b1 = 123n;
// Puede ser arbitrariamente grande.
const b2 = -1234567890987654321n;
```

BigInts también se puede construir a partir de valores numéricos o valores de cadena mediante el constructor `BigInt` .

```
const b1 = BigInt(123);
// El uso de un string evita la pérdida de precisión, ya que es un número largo
// los literales no representan lo que parecen.
const b2 = BigInt("-1234567890987654321");
```

Conceptualmente, un BigInt es solo una secuencia arbitrariamente larga de bits que codifica un número entero.

- puedes realizar con seguridad cualquier operación aritmética sin perder precisión o **desbordamiento** (***overflow***) o **subdesbordamiento** (***underflow***).

```
const entero = 12 ** 34;        // 4.9222352429520264e+36; solo tiene una precisión limitada
const bigint = 12n ** 34n;      // 49222352429520267040371132431220080064n
```

En comparación con los números, los valores BigInt brindan una mayor **precisión** cuando representan *números enteros grandes:* sin embargo, no pueden representar *números de punto flotante* .

- por ejemplo, la división **redondearía a cero** :

```
const bigintDiv = 5n / 2n;      // 2n, porque no hay 2.5 en BigInt
```

Este ejemplo demuestra dónde al incrementar `Number.MAX_SAFE_INTEGER` devuelve el resultado esperado:

```
// con BigInt
const x = BigInt(Number.MAX_SAFE_INTEGER);          // 9007199254740991n
x + 1n === x + 2n;                                  // falso porque 9007199254740992n y
                                                    //  9007199254740993n son desiguales

// con Number
Number.MAX_SAFE_INTEGER + 1 === Number.MAX_SAFE_INTEGER + 2;    // cierto porque ambos son
                                                                // 9007199254740992
```

Puedes usar la mayoría de los operadores para trabajar con BigInts, incluidos `+, *, -, **` y `%` : el único prohibido es `>>>` .

Un BigInt no es estrictamente igual a un Número con el mismo valor matemático, pero lo es vagamente .

- Los valores de BigInt no son siempre más precisos ni siempre menos precisos que los `Number`, ya que BigInts no puede representar números fraccionarios, pero puede representar números enteros grandes con mayor precisión.
- ningún tipo implica al otro, y no son mutuamente sustituibles.
- se lanza un `TypeError` si **los valores BigInt** se mezclan con números regulares en expresiones aritméticas, o si se **convierten implícitamente** entre sí.

Objeto Number

El objeto incorporado (built-in) Number:

- tiene propiedades para **constantes numéricas,** como **valor máximo, no es un número** e **infinito** .
- no puedes cambiar los valores de estas propiedades y las usa de la siguiente manera:

```
const numberMasGrande    = Number.MAX_VALUE;
const numberMasPeque     = Number.MIN_VALUE;
const numberInfinito     = Number.POSITIVE_INFINITY;
const numberInfinitoNeg  = Number.NEGATIVE_INFINITY;
const noUnNumero         = Number.NaN;
```

Siempre de debe hacer referencia a una propiedad del objeto **Number** como se muestra, y no como una propiedad de un objeto Number que tu crees (por ejemplo: `(new Number(34)).MIN_VALUE`)

La siguiente tabla resume las propiedades del objeto Number .

Propiedad	Descripción
Number.MAX_VALUE	El número positivo más grande representable (`1.7976931348623157e+308`)
Number.MIN_VALUE	El número positivo más pequeño representable (`5e-324`)
Number.NaN	Valor especial "no es un número"
Number.NEGATIVE_INFINITY	Valor infinito negativo especial; devuelto por desbordamiento
Number.POSITIVE_INFINITY	Valor infinito positivo especial; devuelto por desbordamiento
Number.EPSILON	Diferencia entre `1` y el valor más pequeño mayor que `1` que se puede representar como un **Número** (`2.220446049250313e-16`)
Number.MIN_SAFE_INTEGER	Entero seguro mínimo en JavaScript (−(2^53 - 1), o `-9007199254740991`)
Number.MAX_SAFE_INTEGER	Entero seguro máximo en JavaScript (+(2^53 − 1), o `+9007199254740991`)

Método	Descripción
Number.parseFloat()	Analiza un argumento string y devuelve un número de punto flotante. Igual que la **función global** parseFloat() .
Number.parseInt()	Analiza un argumento string y devuelve un entero de la raíz o base especificada. Igual que la función global parseInt() .
Number.isFinite()	Determina si el valor pasado es un número finito.
Number.isInteger()	Determina si el valor pasado es un número entero.
Number.isNaN()	Determina si el valor pasado es **NaN** . Versión más robusta del isNaN() global original .
Number.isSafeInteger()	Determina si el valor proporcionado es un número *entero seguro* .

El **prototipo** de **Number** proporciona métodos para recuperar información de objetos **Number** en varios formatos.

- la siguiente tabla resume los métodos de Number.prototype .

Método	Descripción
toExponential()	Devuelve una cadena que representa el número en notación exponencial.
toFixed()	Devuelve una cadena que representa el número en notación de punto fijo.
toPrecision()	Devuelve una cadena que representa el número con una precisión especificada en notación de punto fijo.

Objeto Math

El objeto incorporado Math tiene propiedades y métodos para funciones y constantes matemáticas.

- a diferencia de muchos otros objetos, nunca se crea o instancia un objeto Math: siempre usa el objeto Math estático.
- por ejemplo, la propiedad PI del objeto Math tiene el valor de pi (3.141…), que usaría en una aplicación como

```
Math.PI
```

De manera similar, las funciones matemáticas estándar son métodos de Math :

- estas incluyen funciones trigonométricas, logarítmicas, exponenciales y otras.
- por ejemplo, si desea utilizar la función trigonométrica seno, escribiría

```
Math.sin(1.56)
```

- ten en cuenta que todos los métodos trigonométricos de **`Math`** toman argumentos en radianes.

La siguiente tabla resume los métodos del objeto **`Math`** .

métodos de **`matemáticas`**

Método	Descripción
`abs()`	Valor absoluto
`sin(), cos(), tan()`	Funciones trigonométricas estándar; con el argumento en radianes.
`asin(), acos(), atan(), atan2()`	funciones trigonométricas inversas; valores devueltos en radianes.
`sinh(), cosh(), tanh()`	funciones hiperbólicas; Argumento en ángulo hiperbólico.
`asinh(), acosh(), atanh()`	Funciones hiperbólicas inversas; valores devueltos en ángulo hiperbólico.
`pow(), exp(), expm1(), log(), log10(), log1p(), log2()`	Funciones exponenciales y logarítmicas.
`floor(), ceil()`	Devuelve el entero mayor/menor menor/mayor que o igual a un argumento.
`min(), max()`	Devuelve el valor mínimo o máximo (respectivamente) de una lista de números separados por comas como argumentos.
`random()`	Devuelve un número aleatorio entre 0 y 1.
`round(), fround(), trunc(),`	Funciones de redondeo y truncamiento.
`sqrt(), cbrt(), hypot()`	Raíz cuadrada, raíz cúbica, Raíz cuadrada de la suma de argumentos cuadrados.
`sign()`	El signo de un número, que indica si el número es positivo, negativo o cero.
`clz32(), imul()`	Número de bits cero iniciales en la representación binaria de 32 bits. El resultado de la multiplicación de 32 bits similar a C de los dos argumentos.