7. OPERADORES

Introducción a operadores y expresiones

Describimos ahora las **expresiones** y los **operadores de JavaScript,** incluidos

- asignación,
- comparación,
- aritméticos,
- bit a bit,
- lógicos,
- string,
- ternario
- y más.

Para empezar:

- una ***expresión*** es una unidad de código válida que se **resuelve en** un **valor** .
- hay dos tipos de expresiones:
  - aquellas que tienen **efectos secundarios** (como la asignación de valores)
  - y los que puramente ***evalúan*** .
- La expresión `x = 7` es un ejemplo del primer tipo.
  - esta expresión usa el **`=` *operador*** para **asignar** el valor siete a la variable `x` .
  - la expresión en sí se evalúa como `7` .
- La expresión `3 + 4` es un ejemplo del segundo tipo.
  - esta expresión usa el operador `+` para sumar `3` y `4` y produce un valor, `7` .
  - sin embargo, si finalmente no forma parte de una construcción más grande (por ejemplo, una **declaración de variable** como `const z = 3 + 4`) su resultado se descartará de inmediato
  - ejecutar una expresión `3 + 4` como por sí soloa suele ser un error del programador porque la evaluación no produce ningún efecto.
- todas las expresiones complejas se unen mediante ***operadores,*** como `=` y `+`.

En esta sección, presentaremos los siguientes operadores:

1. **Operadores de asignación**
2. **Operadores de comparación**
3. **Operadores aritméticos**
4. **Operadores bit a bit**
5. **Operadores logicos**
6. **Operadores `BigInt`**
7. **Operadores de cadena/string**
8. **Operador condicional (ternario)**
9. **Operador de coma**
10. **Operadores unarios**
11. **Operadores relacionales** (`in`, `instanceof`)

Estos operadores unen operandos formados por

- operadores de mayor **precedencia**
- o una de las **expresiones básicas**:
  - **literales**
  - **identificadores**
  - **operador de agrupar `()`**
  - operador `this`
  - operador `super`
  - operador `new`

La ***precedencia*** de los operadores determina el orden en que se aplican al evaluar una expresión. Por ejemplo:

```
const x = 1 + 2 * 3;
const y = 2 * 3 + 1;
```

- a pesar de que `*` y `+` aparecen en diferentes órdenes, ambas expresiones darían como resultado `7` porque `*` **tiene precedencia sobre** `+`, por lo que la expresión unida a `*` siempre se evaluará primero.
- puedes "anular" la precedencia del operador usando **paréntesis** (lo que crea una **expresión agrupada** (que es una de las **expresiónes básicas**).

JavaScript tiene varios tipos de operadores:

- un **operador binario** requiere dos **operandos,** uno antes del operador y otro después del operador:

```
operando1 operador operando2
```

  - por ejemplo, `3 + 4` o `x * y` .
  - esta forma se llama **operador binario *infijo*,** porque el operador se coloca entre dos operandos.
  - todos los operadores binarios en JavaScript son infijos.

- Un **operador unario** requiere un solo operando, ya sea antes o después del operador:

```
operador operando
operando operador
```

  - por ejemplo, `x++` o `++x` .
    - la forma `operador operando` se llama operador unario de ***prefijo***,
    - la forma `operando operador` se llama operador unario ***postfijo*** .
    - `++` y `--` son los únicos operadores de ***postfijo*** en JavaScript
  - todos los demás operadores, como `!`, `typeof`, etc. son prefijos.

- Un **operador ternario** requiere 3 operandos: *condición* `?` *expresión1* `:` *expresión2*

Operadores de Asignación

Un **operador de asignación** asigna un valor a su **operando izquierdo** basado en el valor de su **operando derecho** .

- el operador de asignación simple es igual (`=`), que asigna el valor de su operando derecho a su operando izquierdo.
- es decir, `x = f()` es una expresión de asignación que asigna el valor de `f()` a `x.`
- a veces:
  - al operando de la izquierda se le denomina: l-value (left-value), indicando que sale potencialmente sale modificado después de evaluar la expresión.
  - al operando de la derecha se le denomina: r-value (right-value), indicando que, en principio, se utiliza para producir un valor, pero no suele salir modificado, pero sí puede serlo (por ejemplo ti tiene operadores de post-pre incremento, o invoca una función con efectos secundarios)

También hay **operadores de asignación compuestos** que son abreviaturas de las operaciones enumeradas en la siguiente tabla:

Nombre	Operador abreviado	Significado
Asignación	`x = f()`	`x = f()`
Asignación de adición	`x += f()`	`x = x + f()`
Asignación de resta	`x -= f()`	`x = x - f()`
Asignación de multiplicación	`x *= f()`	`x = x * f()`
Asignación de división	`x /= f()`	`x = x / f()`
Asignación módulo	`x %= f()`	`x = x % f()`
Asignación de exponenciación	`x **= f()`	`x = x ** f()`
Asignación de desplazamiento a la izquierda	`x <<= f()`	`x = x << f()`
Asignación de turno a la derecha	`x >>= f()`	`x = x >> f()`
Asignación de turno a la derecha sin firmar	`x >>>= f()`	`x = x >>> f()`
Asignación AND bit a bit	`x &= f()`	`x = x & f()`
Asignación bit a bit de XOR	`x^=f()`	`x = x ^ f()`
Asignación OR bit a bit	`x |= f()`	`x = x | f()`
Asignación lógica Y	`x &&= f()`	`x && (x = f())`
Asignación OR lógica	`x ||= f()`	`x || (x = f())`
Asignación coalescente nula	`x ?? = f ()`	`X ?? (x = f())`

Asignación a propiedades

Si una expresión **se evalúa como** un objeto, entonces el **lado izquierdo** de una **expresión de asignación** puede hacer asignaciones a las propiedades de esa expresión. Por ejemplo:

```
const obj = {};
```

```
obj.x = 3;
console.log(obj.x);              // Imprime 3.
console.log(obj);                // Imprime { x: 3 }.
const clave = "y";
obj[clave] = 5;
console.log(obj[clave]);         // Prints 5.
console.log(obj);                // Prints { x: 3, y: 5 }.
```

Si una expresión no se evalúa como un objeto, las asignaciones a las propiedades de esa expresión no asignan:

```
const val = 0;
val.x = 3;

console.log(val.x);              // Prints undefined.
console.log(val);                // Prints 0.
```

- en **modo estricto,** el código anterior **arroja una excepción** (**throw**) porque no se pueden asignar propiedades a **las primitivas** .
- se considera un error:
  - asignar valores a **propiedades no modificables**
  - asignar valores a las propiedades de una expresión sin propiedades (`null` o `undefined`).

Desestructurar

Para asignaciones más complejas, la sintaxis de **asignación de desestructuración** es una expresión de JavaScript que permite extraer datos de arrays u objetos mediante una sintaxis que refleja la construcción de arrays y literales objeto.

```
const foo = ['uno', 'dos', 'tres'];

// sin desestructurar
const uno   = foo[0];
const dos   = foo[1];
const tres  = foo[2];

// con asignación de desestructurar
const [uno, dos, tres] = foo;
```

Evaluación y anidamiento

En general, las asignaciones se usan dentro de una **declaración de variable** (es decir, con `const, let` o `var`) o como declaraciones independientes).

```
// Declara una variable x y la inicializa con el resultado de f().
// El resultado de la expresión de asignación x = f() se descarta.
let x = f();           // ojo:: () = operador de invocación de función

x = g();              // Reasigna la variable x al resultado de g().
```

Sin embargo, al igual que otras expresiones, las expresiones de asignación como `x = f() se` evalúan en un **valor de resultado** .

- aunque este valor de resultado generalmente no se usa, luego puede ser usado por otra expresión.
- **encadenar asignaciones** o **anidar asignaciones** en otras expresiones puede resultar en un comportamiento sorprendente.
- por esta razón, algunas guías de estilo de JavaScript desaconsejan el encadenamiento o el anidamiento de asignaciones
- sin embargo, el encadenamiento y el anidamiento de asignaciones pueden ocurrir a veces, por lo que es importante poder comprender cómo funcionan.
- al encadenar o anidar una expresión de asignación, su resultado puede asignarse a otra variable.
- se puede apuntar en un registro, se puede colocar dentro de un literal array o una llamada de función, etc.

```
// ejemplos de asignaciones de encadenamiento
let x;
const y = (x = f()) ;           // O de manera equivalente: const y = x = f() ;
console.log(y);                 // Registra el valor de retorno de la asignación x = f().

console.log(x = f());           // Registra el valor devuelto directamente.

// Una expresión de asignación se puede anidar en cualquier lugar
```

```
// donde generalmente se permiten expresiones,
// como elementos de arrays literales o como argumentos de llamadas a funciones.
console.log( [ 0, x = f(), 0 ] );
console.log( f(0, x = f(), 0) );
```

- el **resultado de la evaluación** coincide con la expresión a la derecha del signo `=` en la columna "Significado" de la tabla anterior.

- eso significa que `x = f()` **se** evalúa en el resultado de `f()`, `x += f()` **se** evalúa en la suma resultante `x + f()`, `x **= f()` se evalúa en la potencia resultante `x ** y`, y así sucesivamente.

- en el caso de **las asignaciones lógicas,** `x &&= f()`, `x ||= f()` y `x ??= f()`, el **valor devuelto** es el de la operación lógica sin reflejar la asignación, por lo que el varor devuelto será `x && f()`, `x || f()`, y `x ?? f()`, respectivamente.

- **al encadenar estas expresiones sin paréntesis u otros operadores de agrupación como los literales array, las expresiones de asignación:**
  - **se agrupan de derecha a izquierda (son asociativas por la derecha)**
  - **pero se evalúan de izquierda a derecha .**

- ten en cuenta que, para todos los operadores de asignación que no sean `=`, los valores resultantes siempre se basan en los valores de los operandos *antes* de la operación.
  - por ejemplo, suponga que se han declarado las siguientes funciones `f` y `g` y las variables `x` e `y` :

```
function f () {
  console.log('F!');
  return 2;
}

function g () {
  console.log('G!');
  return 3;
}

let x, y;
```

Considera estos tres ejemplos:

Ejemplo 1: `y = x = f()`
Ejemplo 2: `y = [f(), x = g()]`
Ejemplo 3: `x[f()] = g()`

Ejemplo de evaluación 1

`y = x = f()`

- es equivalente a `y = (x = f())`, porque el operador de asignación `=` es **asociativo por la derecha**
- sin embargo, **evalúa de izquierda a derecha**:

1. Comienza la evaluación de la expresión de asignación `y = x = f()`.
   1. La `y` en el lado izquierdo de esta tarea se evalúa como una **referencia** a la variable denominada `y` .
   2. La expresión de asignación `x = f()` comienza a evaluar.
      1. La `x` en el **lado izquierdo** de esta tarea se evalúa como una referencia a la variable denominada `x` .
      2. La **llamada de función** `f()` imprime "F!" a la consola y luego se evalúa al número `2` .
      3. Ese `2` resultado de `f()` se asigna a `x` .
   3. La expresión de asignación `x = f()` ahora ha terminado de evaluarse; su resultado es el nuevo valor de `x`, que es `2` .
   4. Ese resultado `2` a su vez también se asigna a `y` .
2. La expresión de asignación `y = x = f()` ahora ha terminado de evaluarse;
   - su resultado es el nuevo valor de `y`, que resulta ser `2` .
   - `x` e `y` están asignados a `2` y la console.ha impreso "F!".

Ejemplo de evaluación 2

`y = [ f(), x = g() ]` también **evalúa de izquierda a derecha** :

1. La expresión de asignación `y = [ f(), x = g() ]` comienza a evaluar.
   1. La `y` en la parte izquierda de esta tarea se evalúa como una referencia a la variable denominada `y` .

2. El literal array `[f(), x = g()]` comienza a evaluarse.
    1. La llamada de función `f()` imprime "F!" en la consola y luego se evalúa al número `2` .
    2. La expresión de asignación `x = g()` comienza a evaluar.
        1. La `x` en el lado izquierdo de esta tarea se evalúa como una referencia a la variable denominada `x`
        2. La llamada de función `g()` imprime "G!" a la consola y luego se evalúa al número `3` .
        3. Ese `3` resultado de `g()` se asigna a `x` .
    3. La expresión de asignación `x = g()` ahora ha terminado de evaluarse;
        - su resultado es el nuevo valor de `x`, que es `3` .
        - ese `3` resultado se convierte en el siguiente elemento en el **literal array** (después del `2` de `f()`).
3. El **literal array** `[ f(), x = g() ]` ahora ha terminado de evaluar; su resultado es un array con dos valores: `[ 2, 3 ]` .
4. Ese array `[ 2, 3 ]` ahora es asignado a `y` .

2. La expresión de asignación `y = [ f(), x = g() ]` ahora ha terminado de evaluarse;
    - su resultado es el nuevo valor de `y`, que resulta ser `[ 2, 3 ]` .
    - `x` ahora está asignado a `3`, `y` ahora está asignado a `[2, 3]`
    - y la consola ha impreso "F!" luego "G!".

Ejemplo de evaluación 3

`x[f()] = g()` también **evalúa de izquierda a derecha** .
- observación: este ejemplo asume que `x` ya está asignado a algún objeto.

1. La expresión de asignación `x[f()] = g()` comienza a evaluar.
    1. El acceso a la propiedad `x[f()]` a la izquierda de esta asignación comienza a evaluarse.
        1. La `x` en este acceso de propiedad se evalúa como una referencia a la variable llamada `x` .
        2. Entonces la llamada a la función `f()` imprime "F!" en la consola y luego se evalúa al número `2` .
    2. El acceso a la propiedad de `x` indicada por `f()`, `x[f()]` , en esta asignación ya ha terminado de evaluarse; su resultado es una referencia de propiedad variable: `x[2]` .
    3. Entonces la llamada a la función `g()` imprime "G!" en la consola y luego se evalúa al número `3` .
    4. Ese `3` ahora está asignado a `x[2]` (Este paso tendrá éxito solo si `x` se asigna a un objeto).

2. La expresión de asignación `x[f()] = g()` ahora ha terminado de evaluarse:
    - su resultado es el nuevo valor de `x[2]`, que resulta ser `3`
    - a `x[2]` se le asigna 3
    - la consola ha impreso "F!" luego "G!".

Recapitulando:

Encadenar asignaciones o **anidar asignaciones** en otras expresiones puede resultar en un comportamiento sorprendente y complicado de anticipar:
- por esta razón, se desaconseja encadenar asignaciones en la misma declaración .
- en particular, colocar una **cadena de variables** en una instrucción `const`, `let` o `var` a menudo *no* funciona.
- solo se declararía la variable más externa/más a la izquierda; otras variables dentro de la cadena de asignación *no son* declaradas por la instrucción `const` / `let` / `var` .
- por ejemplo:

```
const z = y = x = f();
```

    - se agrupa como: `z = ( y = ( x = f() ) )`
        - `x = f()` : declara una variable x, no una constante
        - `y =` : declara una variable y, no una constante
    - por lo tanto, aparentemente esta declaración declara las constantes `x`, `y`, `z`
    - sin embargo, en realidad solo declara la variable `z` .
    - `y` y `x` son referencias no válidas a variables inexistentes (en **modo estricto**) o, peor aún, crearían implícitamente **variables globales** para `x` e `y` en **modo descuidado** .

Operadores de comparación

Un **operador de comparación**

- compara sus operandos y devuelve un **valor lógico** en función de si la comparación es verdadera.
- los **operandos** pueden ser valores numéricos, string, lógicos u objetos .
- las cadenas se comparan según el **orden lexicográfico estándar,** utilizando **valores Unicode** .
- en la mayoría de los casos, si los dos operandos no son del mismo tipo, JavaScript intenta convertirlos a un tipo apropiado para la comparación.
  - este comportamiento generalmente resulta en la comparación numérica de los operandos.
  - las únicas excepciones a la conversión de tipo dentro de las comparaciones implican los operadores `===` y `!==`, que realizan una **comparación de igualdad estricta** y **comparación de desigualdad estricta** .
    - estos operadores no intentan convertir los operandos en tipos compatibles antes de verificar la igualdad.
- La siguiente tabla describe los operadores de comparación en términos de este código de ejemplo:

```
const var1 = 3;
const var2 = 4;
```

Operadores de comparación

Operador	Descripción	Ejemplos que devuelven `true`

Igual (`==`)	Devuelve `true` si los operandos son iguales.	`3 == var1`
		`"3" == var1`
		`3 == '3'`

No igual (`!=`)	Devuelve `true` si los operandos no son iguales.	`var1 != 4` `var2 != "3"`
Igualdad estricta (`===`)	Devuelve `true` si los operandos son iguales y del mismo tipo . Consulte también `Object.is` y la igualdad en JS .	`3 === var1`
Estricto no igual (`!==`)	Devuelve `true` si los operandos son del mismo tipo pero no iguales, o son de diferente tipo.	`var1 !== "3"` `3 !== '3'`
Mayor que (`>`)	Devuelve `true` si el operando izquierdo es mayor que el operando derecho.	`var2 > var1` `"12" > 2`
Mayor que o igual (`>=`)	Devuelve `true` si el operando izquierdo es mayor o igual que el operando derecho.	`var2 >= var1` `var1 >= 3`
Menos que (`<`)	Devuelve `true` si el operando izquierdo es menor que el operando derecho.	`var1 < var2` `"2" < 12`
Menor o igual (`<=`)	Devuelve `true` si el operando izquierdo es menor o igual que el operando derecho.	`var1 <= var2` `var2 <= 5`

Nota: ¡OJO! `=>` no es un operador de comparación sino más bien la notación para **funciones de flecha** .

Operadores aritméticos

Un **operador aritmético** toma valores numéricos (ya sean **literales** o variables) como sus operandos y devuelve un solo valor numérico.

- los operadores aritméticos estándar son **suma** (`+`), **resta** (`-`), **multiplicación** (`*`) y **división** (`/`).
- estos operadores funcionan como lo hacen en la mayoría de los otros lenguajes de programación cuando se usan con números de coma flotante (en particular, tenga en cuenta que la división por cero produce `Infinity`). Por ejemplo:

```
1 / 2;                      // 0.5
1 / 2 === 1,0 / 2,0;        // esto es cierto
```

Además de las operaciones aritméticas estándar (`+, -, *, /`), JavaScript proporciona los operadores aritméticos enumerados en la siguiente tabla:

Operadores aritméticos

Operador	Descripción	Ejemplo
Resto (`%`)	Operador binario. Devuelve el resto entero de dividir los dos operandos.	12 % 5 devuelve 2.
Incremento (`++`)	Operador unario. Agrega uno a su operando. Si se usa como operador de prefijo (`++x`), devuelve el valor de su operando después de agregar uno; si se usa como operador de sufijo (`x++`), devuelve el valor de su operando antes de agregar uno.	Si `x` es 3, entonces `++x` establece `x` en 4 y devuelve 4 mientras que `x++` devuelve 3 y, solo entonces, establece `x` en 4.
Decremento (`--`)	Operador unario. Resta uno de su operando. El valor de retorno es análogo al del operador de incremento.	Si `x` es 3, entonces `--x` establece `x` en 2 y devuelve 2 mientras que `x--` devuelve 3 y, solo entonces, establece `x` en 2.

Operador	Descripción	Ejemplo
Negación unaria (`-`)	Operador unario. Devuelve la negación de su operando.	Si `x` es 3, entonces `-x` devuelve -3. `-false` devuelve `-0`
Más unario (`+`)	Operador unario. Intenta convertir el operando en un número, si aún no lo está.	`+"3"` devuelve `3` . `+true` devuelve `1`
Operador de exponenciación (`**`)	Calcula la `base` a la **potencia del** `exponente`, es decir, `base^exponente`	`2 ** 3` devuelve `8` . `10 ** -1` devuelve `0,1` .

Operadores lógicos a nivel de bit (bit a bit)

Un **operador bit** a bit trata sus operandos como un conjunto de 32 bits (ceros y unos), en lugar de números **decimales, hexadecimales** u **octales** .

- por ejemplo, el número decimal nueve tiene una representación binaria de 1001.
- los operadores bit a bit realizan sus operaciones en dichas representaciones binarias, pero devuelven valores numéricos estándar de JavaScript.
- los números con más de 32 bits pierden sus bits más significativos

La siguiente tabla resume los operadores bit a bit de JavaScript.

Operador	Uso	Descripción
Y bit a bit	`a & b`	Devuelve un uno en cada posición de bit para la que los bits correspondientes de ambos operandos son unos.
O bit a bit	`a | b`	Devuelve un cero en cada posición de bit para la que los bits correspondientes de ambos operandos son ceros.
XOR bit a bit	`a ^ b`	Devuelve un cero en cada posición de bit para la que los bits correspondientes son los mismos. [Devuelve uno en cada posición de bit para la que los bits correspondientes son diferentes.]
bit a bit NO	`~ a`	Invierte los bits de su operando.

Nota: aunque los operadores bit a bit *se* pueden usar para representar varios **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 array 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.
- puede ser necesario utilizar dichas técnicas en entornos muy restringidos, como cuando se trata de hacer frente a las limitaciones del almacenamiento local o en casos extremos (como cuando cada bit de la red cuenta).
- esta técnica solo debe considerarse cuando es la última medida que se puede tomar para optimizar el tamaño.

Conceptualmente, los **operadores lógicos bit a bit** funcionan de la siguiente manera:

- los operandos se convierten en enteros de 32 bits y se expresan mediante una serie de bits (ceros y unos).
- los números con más de 32 bits obtienen sus bits más significativos descartados .
- por ejemplo, el siguiente entero con más de 32 bits se convertirá en un entero de 32 bits:

```
Antes:   1110 0110 1111 1010 0000 0000 0000 0110 0000 0000 0001
Después:              1010 0000 0000 0000 0110 0000 0000 0001
```

- cada bit del primer operando se empareja con el bit correspondiente del segundo operando: primer bit con el primer bit, segundo bit con el segundo bit, y así sucesivamente.
- el operador se aplica a cada par de bits y el resultado se construye bit a bit.

Por ejemplo, la representación binaria de nueve es 1001 y la representación binaria de quince es 1111. Entonces, cuando los operadores bit a bit se aplican a estos valores, los resultados son los siguientes:

Expresión	Resultado	Descripción binaria
`15 & 9`	`9`	`1111 & 1001 = 1001`
`15 | 9`	`15`	`1111 | 1001 = 1111`
`15 ^ 9`	`6`	`1111 ^ 1001 = 0110`
`~ 15`	`-16`	`~ 0000 0000 … 0000 1111 = 1111 1111 … 1111 0000`
`~ 9`	`-10`	`~ 0000 0000 … 0000 1001 = 1111 1111 … 1111 0110`

- tenga en cuenta que los 32 bits se invierten utilizando el operador NOT bit a bit, y que los valores con el bit más significativo (más a la izquierda) establecido en 1 representan números negativos (representación de complemento a dos).

- el resultado de evaluar `~x` es el mismo valor que el de evaluar `-x - 1`.

Operadores de desplazamiento bit a bit

Los operadores de desplazamiento bit a bit toman dos operandos:

- el primero es un entero que se va a desplazar y el segundo especifica el número de posiciones de bit por las que se va a desplazar el primer operando.
- la dirección de la operación de cambio es controlada por el operador utilizado.
- los operadores de desplazamiento convierten/coercionan sus operandos en enteros de 32 bits y devuelven un resultado de tipo `Number` o `BigInt` :
  - específicamente, si el tipo del operando izquierdo es `BigInt`, devuelven `BigInt`;
  - de lo contrario, devuelven `Number` .

Los operadores de desplazamiento se enumeran en la siguiente tabla.

Operadores de desplazamiento bit a bit

Operador	Descripción	Ejemplo
Desplazamiento a la izquierda (`<<`)	Este operador desplaza el primer operando el número especificado de bits hacia la izquierda. Los bits en exceso desplazados hacia la izquierda se descartan. Los bits cero se desplazan desde la derecha.	`9<<2` produce 36, porque 1001 desplazado 2 bits a la izquierda se convierte en 100100, que es 36.
Desplazamiento a la derecha con propagación de signos (`>>`)	Este operador desplaza el primer operando el número especificado de bits a la derecha. Los bits en exceso desplazados hacia la derecha se descartan. Las copias del bit más a la izquierda se desplazan desde la izquierda.	`9>>2` da 2, porque 1001 desplazado 2 bits a la derecha se convierte en 10, que es 2. Del mismo modo, `-9>>2` da -3, porque se conserva el signo.
Desplazamiento a la derecha con relleno con ceros (`>>>`)	Este operador desplaza el primer operando el número especificado de bits a la derecha. Los bits en exceso desplazados hacia la derecha se descartan. Los bits cero se desplazan desde la izquierda.	`19>>>2` produce 4, porque 10011 desplazado 2 bits a la derecha se convierte en 100, que es 4. Para números no negativos, el desplazamiento a la derecha con relleno de ceros y el desplazamiento a la derecha con propagación de signo producen el mismo resultado.

Operadores lógicos

Los operadores lógicos se utilizan normalmente con valores **booleanos** (lógicos); cuando lo son, devuelven un valor booleano.

- sin embargo, el `&&` y `||` los operadores en realidad devuelven el valor de uno de los operandos especificados, por lo que si estos operadores se utilizan con valores no booleanos, pueden devolver un valor no booleano.
- los operadores lógicos se describen en la siguiente tabla.

Operadores logicos

Operador	Uso	Descripción
AND lógico `&&`	`expr1 && expr2`	Devuelve `expr1` si se puede convertir a `false`; de lo contrario, devuelve `expr2`. Por lo tanto, cuando se usa con valores booleanos, `&&` devuelve `true` si ambos operandos son `true`; de lo contrario, devuelve `false` .
O lógico `||`	`expr1 || expr2`	Devuelve `expr1` si se puede convertir a `true` ; de lo contrario, devuelve `expr2.` Por lo tanto, cuando se usa con valores booleanos, `||` devuelve `true` si alguno de los operandos es `true`; si ambos son falsos, devuelve `false` .
NO lógico `!`	`! expr`	Devuelve `false` si su único operando se puede convertir en `true`; de lo contrario, devuelve `true` .

Ejemplos de expresiones que se pueden convertir en `false` son (**valores falsy**):

Valor	Descripción
`false`	La palabra clave false .
`0`	El Número cero (entonces, también 0.0 , etc., y 0x0).
`-0`	El Número menos cero (entonces, también -0.0 , etc., y -0x0).
`0n`	El BigInt cero (entonces, también 0x0n). Tenga en cuenta que no hay un cero negativo de BigInt : la negación de 0n es 0n .
`""` `''`	Valor de cadena vacío.

Valor	Descripción
``	
`null`	null — la ausencia de referencia a objeto.
`undefined`	`undefined` — el valor primitivo.
`NaN`	NaN: no es un número.
`document.all`	Los objetos son falsos si y solo si tienen la ranura interna `[[IsHTMLDDA]]`. Dicha ranura solo existe en `document.all` y no se puede configurar mediante JavaScript.

El siguiente código muestra ejemplos del operador `&&` (AND lógico).

```
const a1 =  true  && true;              // t && t devuelve true
const a2 =  true  && false;             // t && f devuelve false
const a3 =  false && true;              // f && t devuelve false
const a4 =  false && (3 === 4);         // f && f devuelve false
const a5 =  'Cat' && 'Dog';             // t && t devuelve Dog
const a6 =  false && 'Cat';             // f && t devuelve false
const a7 =  'Cat' && false;             // t && f devuelve false
```

El siguiente código muestra ejemplos de `||` (OR lógico) operador.

```
const o1 =  true || true;               // t || t devuelve true
const o2 = false || true;               // f || t devuelve true
const o3 =  true || false;              // t || f devuelve true
const o4 = false || (3 === 4);          // f || f devuelve false
const o5 = 'Cat' || 'Dog';              // t || t devuelve Cat
const o6 = false || 'Cat';              // f || t devuelve Cat
const o7 = 'Cat' || false;              // t || f devuelve Cat
```

El código siguiente muestra ejemplos de `!` (NO lógico) operador.

```
const n1 = !true;                       // !t devuelve false
const n2 = !false;                      // !f devuelve true
const n3 = !'Cat';                      // !t devuelve false
```

Evaluación de cortocircuito

Como las expresiones lógicas se evalúan de izquierda a derecha, se evalúan para una posible **evaluación de "cortocircuito"** (***short-circuit***) utilizando las siguientes reglas:

- `false && cualquier cosa` se evalúa en cortocircuito como `false`.
- `true || cualquier cosa` se evalúa en cortocircuito como verdadera.

Las reglas de la lógica garantizan que estas valoraciones sean siempre correctas.

- ten en cuenta que la parte `cualquier cosa` de las expresiones anteriores no se evalúa, por lo que los efectos secundarios de hacerlo no surten efecto.
- tenga en cuenta que para el segundo caso, en el código moderno puede usar el **operador coalescente nulo** (`??`) que funciona como `||`, pero solo devuelve la segunda expresión, cuando la primera es "nulish ", es decir, `null` o `undefined` .
  - por lo tanto, es la mejor alternativa para proporcionar valores predeterminados, cuando valores como `''` o `0` también son valores válidos para la primera expresión.

Operadores BigInt

La mayoría de los operadores que se pueden usar entre números también se pueden usar entre valores `BigInt` .

```
// Adición de BigInt
const a = 1n + 2n;                          // 3n
// División con BigInts redondeada hacia cero
const b = 1n / 2n;                          // 0n
// Las operaciones bit a bit con BigInts no truncan ningún lado
const c = 40000000000000000n >> 2n;         // 10000000000000000n
```

Una excepción es el **desplazamiento a la derecha sin signo** (`>>>`), que no está definido para valores `BigInt`.

- esto se debe a que `BigInt` no tiene un ancho fijo, por lo que técnicamente no tiene un "bit más alto".

```
const d = 8n >>> 2n; // TypeError: BigInts no tiene desplazamiento a la derecha
                     //   sin firmar, use >> en su lugar
```

BigInts y los números no se reemplazan mutuamente; no puede mezclarlos en los cálculos.

```
constante a = 1n + 2; // TypeError: no se puede mezclar BigInt y otros tipos
```

Esto se debe a que BigInt no es ni un subconjunto ni un superconjunto de números.

- tiene mayor precisión que los `Number` cuando representan números enteros grandes, pero no pueden representar decimales, por lo que la conversión implícita en cualquier lado podría perder precisión.
- usa la **conversión explícita** para señalar si desea que la operación sea una operación numérica o BigInt.

```
const a = Number(1n) + 2;       // 3
const b = 1n + BigInt(2);       // 3n
```

Puedes comparar los BigInt con los Number.

```
const a = 1n > 2;               // false
const b = 3 > 2n;               // true
```

Operadores de cadena

Además de los operadores de comparación, que se pueden usar en valores de cadena, el **operador de concatenación** (+) concatena dos valores de cadena y devuelve otra cadena que es la unión de las dos cadenas de operandos.

Por ejemplo,

```
console.log('mi ' + 'cadena');              //  "mi cadena".
```

El operador de asignación abreviado `+=` también se puede usar para concatenar cadenas. Por ejemplo,

```
let micadena = 'alfa';
micadena += 'beta';                         // "alfabeta"
```

Operador condicional (ternario)

El **operador condicional**

- es el único operador de JavaScript que acepta tres operandos.
- el operador puede tener uno de dos valores en función de una condición.
- la sintaxis es:

```
condición ? val1 : val2
```

- si la **`condición`** es verdadera, el operador tiene el valor de **`val1`**
- de lo contrario, tiene el valor de **`val2`** .
- puedes usar el operador condicional en cualquier lugar donde usaría un operador estándar.

Por ejemplo,

```
const estado = edad >= 18 ? 'adulto' : 'menor';
```

- esta declaración asigna el valor "adulto" a la variable **`estado`** si la **`edad`** es de dieciocho años o más. De lo contrario, asigna el valor "menor" al **`estado`** .

operador de coma

El **operador de coma** (**`,`**) evalúa ambos operandos y devuelve el valor del último operando.

- este operador se usa principalmente dentro de un ciclo **`for`**, para permitir que varias variables se actualicen cada vez que se ejecuta el ciclo.
- se considera de mal estilo utilizarlo en otro lugar, cuando no es necesario.
- a menudo, se pueden y deben usar dos declaraciones separadas en su lugar.

Por ejemplo, si **`a`** es un array bidimensional con 10 elementos en un lado, el siguiente código usa el operador de coma para actualizar dos variables a la vez. El código imprime los valores de los elementos diagonales en el array:

```
const x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
const a = [x, x, x, x, x];

for (sea i = 0, j = 9; i <= j; i++, j--) {
//              ^                ^
   console.log(`a[${i}][${j}]= ${a[i][j]}`);
}
```

Operadores unarios

Una operación unaria es una operación con un solo operando.

Eliminar

El operador de eliminación elimina **`la propiedad de un objeto`** . La sintaxis es:

```
delete objeto.propiedad;
delete objeto[ claveDePropiedad ];
delete objectName[ índice ];
```

- donde **`objeto`** es el nombre de un objeto, *`propiedad`* es una propiedad existente y *`claveDePropiedad`* es una cadena o **`Symbol`** que hace referencia a una propiedad existente.
- si el operador de **`delete`** tiene éxito, elimina la propiedad del objeto.
- intentar acceder a él después producirá **`undefined`** .
- el operador de **`delete`** regresa
  - **`true`** si la operación es posible;
  - **`falso`** si la operación no es posible.

```
delete Math.PI;      // devuelve falso (no se pueden eliminar propiedades no configurables )

const miObj = {h: 4};
delete myObj.h;      // devuelve true (puede eliminar propiedades definidas por el usuario)
```

Eliminación de elementos de matriz

Dado que las arrays son solo objetos, es técnicamente posible **`delete`** elementos de ellas.

- sin embargo, esto se considera una mala práctica, trate de evitarlo.
- cuando elimina una propiedad de un array, la propiedad **`length`** no se ve afectada y otros elementos no se **vuelven a indexar** .
- para lograr ese comportamiento, es mucho mejor simplemente sobrescribir el elemento con el valor **`undefined`** .
- para manipular realmente el array, use los diversos métodos de array, como **`splice`**.

typeof

El operador **`typeof`** se utiliza de cualquiera de las siguientes maneras:

```
typeof operando
```

- el operador `typeof` devuelve una cadena que indica el tipo del operando no evaluado.
- *operando* es la cadena, variable, palabra clave u objeto para el que se devolverá el tipo.
- los paréntesis son opcionales.

Supongamos que define las siguientes variables:

```
const miFuncion = new Function('5 + 2');
const forma = 'redondo';
const tamano = 1;
const foo = ['Manzana', 'Mango', 'Naranja'];
const hoy = new Date();
```

El operador `typeof` devuelve los siguientes resultados para estas variables:

```
typeof miFuncion;     // devuelve "function"
typeof forma;         // devuelve "string"
typeof tamano;        // devuelve "number"
typeof foo;           // devuelve "object"
typeof hoy;           // devuelve "object"
typeof noExisto;      // devuelve "undefined"
```

Para las palabras clave `true` y `null`, el operador `typeof` devuelve los siguientes resultados:

```
typeof true;          // devuelve "boolean"
typeof null;          // devuelve "object"
```

Para un número o cadena, el operador `typeof` devuelve los siguientes resultados:

```
typeof 62;            // devuelve "number"
typeof 'Hola mundo'; // devuelve "string"
```

Para **valores de propiedades,** el operador `typeof` devuelve el tipo de valor que contiene la propiedad:

```
typeof document.lastModified;   // devuelve "string"
typeof ventana.longitud;        // devuelve "number"
typeof Math.LN2;                // devuelve "number"
```

Para **métodos** y **funciones,** el operador `typeof` devuelve resultados de la siguiente manera:

```
typeof desenfoque;              // devuelve "function"
typeof evaluacion;              // devuelve "function"
typeof parseInt;                // devuelve "function"
typeof forma.division;          // devuelve "function"
```

Para objetos predefinidos, el operador `typeof` devuelve resultados de la siguiente manera:

```
typeof Date;                    // devuelve "function"
typeof Function;                // devuelve "function"
typeof Math;                    // devuelve "object"
typeof String;                  // devuelve "function"
```

void

El operador `void` se utiliza de cualquiera de las siguientes maneras:

```
void ( expresión )
void  expresión
```

- el operador `void` especifica una expresión para ser evaluada sin devolver un valor .
- *expresión* es una expresión de JavaScript para evaluar.
- los paréntesis que rodean la expresión son opcionales, pero es un buen estilo usarlos .

Operadores relacionales

Un **operador relacional** compara sus operandos y devuelve un valor booleano en función de si la comparación es verdadera.

in

El operador **in** devuelve **true** si la *propNombreONúmero* está en *nombreDelObjeto* especificado. La sintaxis es:

propNombreONúmero **en** *nombreDelObjeto*

- dónde:
  - *propNameOrNumber* es una expresión de cadena, numérica o de símbolo que representa un **nombre de propiedad** o **un índice de matriz,**
  - *objectName* es el nombre de un objeto.

Los siguientes ejemplos muestran algunos usos del operador **in**:

```
// arrays
const arboles = ['secuoya', 'laurel', 'cedro', 'roble', 'arce'];
0 in arboles ;        // devuelve true
3 in arboles;         // devuelve true
6 in arboles;         // devuelve false
'bahía' in arboles;   // devuelve false (debe especificar el número de índice,
                      // no es el valor en ese índice)
'length' in arboles; // devuelve true (la longitud es una propiedad de Array)

// objetos incorporados
'PI' in Math;         // devuelve true

const miCadena = new String('coral');
'length' in miCadena; // devuelve true

// Objetos personalizados
const miCoche = { marca: 'Honda', modelo: 'Accord', anno: 1998 };
'marca'  in miCoche; // devuelve true
'modelo' in miCoche; // devuelve true
```

instanceof

El operador **instanceof** devuelve **true** si el objeto especificado es del **tipo de objeto** especificado. La sintaxis es:

nombreObjeto **instancia** *tipoObjeto*

- donde *nombreObjeto* es el nombre del objeto para comparar con *tipoObjeto,*
- *tipoObjeto* es un tipo de objeto, como **Date** o **Array** .

Utilice la **instanceof** cuando necesite confirmar el tipo de un objeto **en tiempo de ejecución** .

- por ejemplo, al **capturar excepciones,** puede bifurcarse a diferentes **códigos de manejo de excepciones** según el **tipo de excepción** lanzada.
- por ejemplo, el siguiente código usa una **instanceof** de para determinar si **theDay** es un objeto **Date** .
  - debido a que **theDay** es un objeto **Date, se ejecutan las sentencias en la sentencia if** .

```
const elDia = new Date(1995, 12, 17);
if ( elDia instanceof Date) {
   // sentencias a ejecutar
}
```

Expresiones básicas

Todos los operadores al evaluar una expresión finalmente operan en una o más **expresiones básicas** .

- estas expresiones básicas incluyen
  - **identificadores**
  - **literales**
  - **operador de agrupar** `()`
  - operador `this`
  - operador `super`
  - operador `new`.

this

Usa la palabra clave `this` para referirse al propio objeto desde el que se utiliza `this`.

- usa `this` con la notación de punto o corchete:

```
this[ 'nombreDePropiedad' ]
this.nombreDePropiedad
```

Supongamos que una función llamada `validar` valida la propiedad de `valor` de un objeto, recibidos como parámetros el objeto y los valores `alto` y `bajo`:

```
function validar(obj, bajo, alto) {
   if ((obj.valor < bajo) || (obj.valor > alto)) {
      console.log('¡Valor no válido!');
   }
}
```

Puede llamar a `validar` en el evento `onchange` de cada elemento de `formulario` **manejador de eventos,** usándolo para pasarlo al **elemento de formulario, como en** `el` siguiente ejemplo:

```
<p>Ingrese un número entre 18 y 99:</p>
<input type="text" name="edad" size="3" onchange="validar( this, 18, 99);" />
```

Operador de agrupación

El **operador de agrupación** `( )` controla la **precedencia** de la **evaluación** en las expresiones.

- por ejemplo, puede anular primero la multiplicación y la división, luego la suma y la resta para evaluar primero la suma.

```
const a = 1;
const b = 2;
const c = 3;

// precedencia por defecto
a + b * c               // 7
// evaluado por defecto así
a + (b * c)             // 7

// ahora anulando la precedencia
// suma antes que multiplicacion
(a + b) * c             // 9

// que es equivalente a
a * c + b * c           // 9
```

new

Puede utilizar el operador `new` para crear una **instancia** de un **tipo de objeto definido por el usuario** o de uno de los **tipos de objetos integrados** .

- use `new` de la siguiente manera:

```
const nombre = new tipoObjeto(param1, param2, /* …, */ paramN);
```

super

La palabra clave **super** se utiliza para llamar a funciones en **el padre de un objeto** .

- es útil con las **clases** para llamar al **constructor padre,** por ejemplo.

```
super(argumentos);                          // llama al constructor principal con 0 o más argumentos
super.functionOnParent(argumentos);
```