6. COERCIÓN/CONVERSIÓN DE UN TIPO A OTRO

Introducción a la conversión de tipos

El problema puede aparecer en un contexto en el que se espera un valor primitivo, pero JavaScript no puede inferir claramente cuál sería el tipo deseado:

Contexto	Problema
a) *tipo* `==` *tipo primitivo*	
b) *tipo* `+` *tipo primitivo*	`'37' + 7`

c) tipo `+` *tipo*	`console.log( {} + [] );`
	`+'1.1'`
	`+'' + +'1.1'`

d) tipo `operando-binario` *tipo primitivo*	`'37' - 7`
e) `function` (*tipo primitivo*)	`Date(String)` `Date(Number)`

Conversión de tipos de datos

JavaScript es un **lenguaje de *tipado dinámico*** .

- esto significa que no tienes que especificar el **tipo de datos** de una variable cuando la declara.
- también significa que los tipos de datos se convierten automáticamente según sea necesario durante la ejecución del script: para hacerlo, JavaScript define un puñado de **reglas de coerción**

Entonces, por ejemplo, podrías definir una variable de la siguiente manera:

```
let respuesta = 42;
```

Y más tarde, podrías asignar a la misma variable un valor de string, por ejemplo:

```
respuesta = '¡Gracias por todo el pescado!' ;
```

Debido a que JavaScript es de tipado dinámico, esta asignación no genera un mensaje de error.

Numbers y el operador '+'

En **expresiones** que involucran **valores numéricos** y **valores de string** con el operador +, JavaScript **convierte** los valores numéricos en cadenas.

- por ejemplo, considere las siguientes declaraciones:

```
x = 'La respuesta es ' + 42     // "La respuesta es 42"
y = 42 + 'es la respuesta'      // "42 es la respuesta"
z = '37' + 7                    // "377"
```

Con todos los demás operadores, JavaScript *no* convierte valores numéricos en cadenas .

- por ejemplo:

```
'37' - 7                        // 30
'37' * 7                        // 259
```

Convertir Strings (cadenas de texto) en Numbers

En el caso de que un valor que represente un número esté en la memoria como una cadena/string, existen métodos para la conversión.

- `parseInt()` : solo devuelve números enteros, por lo que no vale para números decimales
- `parseFloat()`

Nota: además, una mejor práctica para `parseInt` es incluir siempre el **parámetro** *radix* (base).
El parámetro radix se usa para especificar qué **sistema numérico** debe usar JavaScript para interpretar el string.

```
parseInt('101', 2)    // Devuelve 5
```

Un método alternativo para recuperar un número de una cadena es con el operador `+` (más unario):

```
'1.1' + '1.1'                   // '1.11.1'
( +'1.1' ) + ( +'1.1' )         // 2.2
                                // Nota: los paréntesis se agregan para mayor claridad,
                                //  no son obligatorios.
```

Coerción hacia tipos primitivos

El proceso de **coerción/conversión de tipos primitivos** va así:

- el proceso "se activa" en un contexto en el que se espera un valor primitivo, pero JavaScript no puede inferir claramente cuál sería el tipo deseado
  - por ejemplo en un contexto donde un string, un `Number` o un `BigInt` son igualmente aceptables.
- Ejemplos:
  - el constructor `Date()`, cuando recibe un argumento que no es una instancia de `Date`:
    - los `String`: representan **cadenas de fecha**
    - los `Number`: representan **marcas de tiempo** .
  - el operador `+`: si un operando es una cadena/string, se realiza la concatenación de cadenas; de lo contrario, se realiza la suma numérica.
  - el operador `==` : si un operando es de un tipo primitivo mientras que el otro es un objeto, el objeto se convierte en un **valor primitivo** pero sin saber de qué tipo en concreto (lo dirá el contexto o las reglas de coerción).

Coerción de objetos hacia un tipo primitivo

Los objetos se convierten en valores de tipos primitivos llamando a uno de los métodos del objeto para ese propósito:

1. `[@@toPrimitive]()` (con `"default"` como pista/sugerencia)
   (NOTA: escrito como `[Symbol.toPrimitive]('parametro')`
   - si el método está definido, debe devolver un valor de un tipo primitivo: devolver un objeto da como resultado un `TypeError`
   - ni `{}` ni `[]` tienen un método `[@@toPrimitive]()` .

2. `valueOf()`
   - tanto `{}` como `[]` heredan `valueOf()` de `Object.prototype.valueOf`, que devuelve al propio objeto sobre el que se invoca.
   - dado que el valor devuelto es un objeto, se ignora.

3. `toString()`

Ten en cuenta que la conversión a un tipo primitivo llama a `valueOf()` antes de `toString()`

- que es similar al comportamiento de la **coerción de números** pero diferente al procedimiento de **coerción de** `String`s
- para `valueOf()` y `toString()`
  - si uno devuelve un objeto, el valor de retorno se ignora y en su lugar se utiliza el valor de retorno del otro
  - si ninguno está presente, o ninguno devuelve un valor de tipo primitivo, **se lanza** (***throw***) un `TypeError` .
    - por ejemplo, en el siguiente código:

```
console.log( {} + [] ); // "[objeto Objeto]"
```

- `({}).toString()` devuelve `"[object Object]"` ,
- `([]).toString()` devuelve `""`
- entonces el resultado es su concatenación: "`[object Object]`" .

El método `[@@toPrimitive]()` siempre tiene prioridad al realizar la conversión a cualquier tipo primitivo:

- la **conversión a tipo primitivo** generalmente se comporta como la **conversión de números**, porque `valueOf()` se llama con prioridad
- sin embargo, los objetos con métodos personalizados `[@@toPrimitive]()` pueden optar por devolver cualquier tipo primitivo.
- `Date` y `Symbol` son los únicos objetos integrados que **sobreescriben** (***override***) el método `[@@toPrimitive]()`:

  - `Date.prototype[@@toPrimitive]()`: trata la sugerencia `"default"` como si fuera un `"string"`
    (NOTA: escrito como `Date.prototype[Symbol.toPrimitive]('string')`

  - `Symbol.prototype[@@toPrimitive]()`: ignora el parámetro de sugerencia y siempre devuelve un `Symbol` .

Coerción entre tipos de números: `Number` y `BigInt`

Hay dos tipos numéricos: `Number` y `BigInt` .

- en ciertos contextos, el lenguaje espera específicamente un `Number` o un `BigInt` :
  - por ejemplo: como `Array.prototype.slice()` : donde el índice debe ser un número
- otras veces, puede valer cualquiera de los 2 tipos numéricos y realizará diferentes operaciones según el tipo de operando.

Coerción numérica :

- es casi lo mismo que la **coerción de números**, excepto que BigInts se devuelve tal cual en lugar de causar un `TypeError`
- La coerción numérica es utilizada por todos los operadores aritméticos, ya que están **sobrecargados** tanto para números como para BigInts.
- la única excepción es el **plus unario**, que siempre hace **coerción numérica** .

Coerción de otros tipos a `Number`

Muchas operaciones integradas que esperan operandos de tipo `Number` primero coercionan/promueven sus operandos a un valor `Number`:

- esto es en gran parte por qué los objetos `Number` se comportan de manera similar a los números primitivos.

El proceso de coerción se puede resumir de la siguiente manera:

- los literales numéricos se devuelven tal cual.
- `undefined` se convierte en `NaN`
- `null` se convierte en 0 .
- `true` se convierte en 1
- `false` se convierte en 0 .
- `String`s
  - se convierten analizándolas como si contuvieran un número literal .
    - un error de análisis da como resultado `NaN` .
    - hay algunas diferencias menores en comparación con un número literal real:
      - los espacios en blanco/terminadores de línea iniciales y finales se ignoran.
      - un dígito 0 inicial no hace que el número se convierta en un **literal octal** (o, en modo estrico, sera rechazado directamente).
      - + y - están permitidos al comienzo de la cadena para indicar su signo.
        - en el código real, "parecen" parte del literal, pero en realidad son **operadores unarios** que no forman parte del número.
        - sin embargo, el signo solo puede aparecer una vez y no debe ir seguido de un espacio en blanco.
      - `Infinity` e `-Infinity` se reconocen como literales.
        - en el código real, son variables globales.
    - los **separadores numéricos** no están permitidos (`_` , `U+005F`)
  - un string vacío `""` o sólo con espacios en blanco `"  "`: se convierte en 0 .
- `BigInt`s lanza un `TypeError` para evitar la **coerción implícita no intencionada** que causa la **pérdida de precisión**.
- los `Symbol`s lanzan un `TypeError` .
- Los objetos se convierten primero en un **primitivo** (=valor de tipo primitivo) llamando a sus métodos:
  - 1º) `[@@toPrimitive]('number')`
  - 2º) `valueOf()`
  - 3º) `toString()`, en ese orden.

  La primitiva resultante se convierte luego en un `Number`.

Hay dos formas de lograr casi el mismo efecto en JavaScript.

- el operador unario `+`: `+x` hace exactamente los pasos de coerción de números explicados anteriormente para convertir `x` .
- la función `Number(x)` : usa el mismo algoritmo para convertir `x` , excepto que `BigInt`s no arroja un `TypeError`, sino que devuelve su valor numérico, con una posible pérdida de precisión.
  - Las funciones estáticas siguintes son similares a `Number()` pero solo convierten `String`s y tienen reglas de análisis ligeramente diferentes.
    - `Number.parseFloat()` : no reconoce el prefijo `0x`

- `Number.parseInt()`: no reconoce el punto decimal

Conversión a números enteros

Algunas operaciones esperan **números enteros:**

- por ejemplo, aquellas que:
  - se utilizan como índices de array/string,
  - se utilizan como componentes de fecha/hora
  - los radix de números (base 2, 8, 16...)
- después de realizar los pasos de conversión de números anteriores, el resultado **se trunca** a un número entero (desechando la **parte fraccionaria**).
  - si el número es `±Infinity`, se devuelve tal cual.
  - si el número es `NaN` o `-0` , se devuelve como 0 .

  Por lo tanto, el resultado siempre es un número entero (que no es `-0`, `NaN` o `±Infinity`).
- en particular, cuando se convierte a números enteros, tanto `undefined` como `null` se convierten en 0 (porque `undefined` se convierte en `NaN`, y `NaN` se convierte en 0).

Conversión de números de ancho fijo

JavaScript tiene algunas **funciones de bajo nivel** (= que operan a nivel de bits o byte)

- se ocupan de la codificación binaria de números enteros, sobre todo **operadores bit a bit** y objetos `TypedArray` .
- los operadores bit a bit siempre convierten los operandos en enteros de 32 bits.
- en estos casos, después de convertir el valor en un número, el número se normaliza al ancho adecuado (32 bits):
  - 1º) truncando la parte fraccionaria
  - 2º) tomando los bits más bajos en la **codificación del complemento a dos** del entero.

```
new Int32Array([1.1, 1.9, -1.1, -1.9]);
// Int32Array(4) [ 1, 1, -1, -1 ]

new Int8Array([257, -257]);
// Int8Array(1) [ 1, -1 ]
//  257 = 0001 0000 0001 = 0000 0001 (mod 2^8) = 1
// -257 = 1110 1111 1111 = 1111 1111 (mod 2^8) = -1 (como un entero en complemento a 2)

new Uint8Array([257, -257]);
// Uint8Array(1) [ 1, 255 ]
// -257 = 1110 1111 1111 = 1111 1111 (mod 2^8) = 255 (tratado como un entero sin signo)
```

Coerción a BigInt

Muchas operaciones integradas que esperan `BigInt`s primero convierten/coercen sus argumentos/operandos a `BigInt`.

La operación se puede resumir de la siguiente manera:

- los `BigInt`s se devuelven tal cual.
- `undefined` y `null`: lanza un `TypeError` .
- `true`: se convierte en `1n`
- `false`: se convierte en `0n` .
- los String se convierten analizándolas como si contuvieran un literal entero.
  - cualquier error de análisis sintáctico da como resultado un `SyntaxError` .
  - la sintaxis es un subconjunto de literales numéricos de string, donde no se permiten puntos decimales ni indicadores de exponente.
- los `Number` arrojan un `TypeError` para evitar la **coerción implícita no intencionada** que causa la **pérdida de precisión**.
- los `Symbol` lanzan un `TypeError` .
- los objetos se convierten primero en un primitivo llamando a sus métodos
  - 1º) `[@@toPrimitive]()` (con `'number'` como sugerencia)
  - 2º) `valueOf()`
  - 3º) `toString()`, en ese orden.

  el primitivo resultante se convierte luego en un `BigInt`.

La mejor manera de lograr casi el mismo efecto en JavaScript es a través de la función **`BigInt`**() :

- **`BigInt(x)`** usa el mismo algoritmo para convertir x , excepto que los **`Number`** NO arrojan un **`TypeError`**, sino que se si son números enteros se convierten a **`BigInt`**. .

Ten en cuenta también que las operaciones integradas que esperan **`BigInt`** a menudo truncan **`BigInt`** a un ancho fijo después de la coerción; por ejemplo:

- **`BigInt.asIntN()`**
- **`BigInt.asUintN()`**
- métodos de **`BigInt64Array`**
- métodos de **`BigUint64Array`**

Otras coerciones/conversiones

Hemos visto que todos los tipos de datos, excepto hacia **`null`** , **`undefined`** y **`Symbol`**, tienen su respectivo proceso de coerción.

Como habrás notado, hay tres caminos distintos a través de los cuales los objetos pueden convertirse en primitivos:

- **coerción hacia tipos primitivos** :

  `[@@toPrimitive]('default')` → **`valueOf()`** → **`toString()`**

- **coerción numérica:** **`Number`** y **`BigInt`**:

  `[@@toPrimitive]('number')` → **`valueOf()`** → **`toString()`**

- **coerción de strings**:

  `[@@toPrimitive]('string')` → **`toString()`** → **`valueOf()`**

En todos los casos:

- **`[@@toPrimitive]()`**: si está definida, debe poder llamarse (ser un **callable**) y devolver un primitivo (valor de tipo primitivo)
- **`valueOf`** o **`toString`**: se ignorarán si no se pueden llamar o devolver un objeto.
- al final del proceso, si tiene éxito, se garantiza que el resultado sea un valor de tipo primitivo.
- el primitivo resultante está luego sujeto a más coerciones dependiendo del contexto.