2.b. Comparaciones de igualdad y semejanza

Operaciónes de comparación de valores

JavaScript proporciona tres **operaciones diferentes de comparación de valores** :

- `===` : **igualdad estricta** (triple igual)
- `==` : **igualdad no estricta** (doble igual)
- `Object.is()`

La operación que se elija depende del tipo de comparación que se desee realizar. Brevemente:

- **doble igual** (`==`)
  - realizará una **conversión de tipo** al comparar dos cosas,
  - manejará `NaN`, `-0` y `+0` especialmente para cumplir con IEEE 754 (entonces **`NaN != NaN`**, y **`-0 == +0`**);

- **triple igual** (`===`)
  - **NO** realizará una conversión **de tipo** al comparar dos cosas,
  - si los tipos difieren, se devuelve **`false`** .
  - hará la misma comparación que doble igual (incluido el manejo especial para `NaN`, `-0` y `+0`) pero sin conversión de tipo ;

- `Object.is()`
  - no hace **conversión** de tipo ni manejo especial para `NaN`, `-0` y `+0`
  - (dándole el mismo comportamiento que `===` excepto en esos valores numéricos especiales).

Corresponden a tres de cuatro **algoritmos de igualdad** en JavaScript:

- **`IsLooselyEqual`** : `==`
- **`IsStrictlyEqual`** : `===`
- **`SameValue`**: `Object.is()`
- **`SameValueZero:`** utilizado por muchas operaciones integradas

Ten en cuenta que

- la distinción entre estos 4 algoritmos sólo tiene que ver con su manejo de los **tipos primitivos** ;
- ninguno de ellos compara si los parámetros son ***conceptualmente similares*** en estructura.
- para cualquier objeto no primitivo `x` e `y` que tenga la misma estructura pero que sean objetos distintos en sí mismos, todas las formas anteriores se evaluarán como **`false`** .

Igualdad estricta usando `===`

La igualdad estricta compara dos valores para la igualdad.

- ninguno de los valores se **convierte implícitamente** en algún otro valor antes de compararse.
  - si los valores tienen tipos diferentes, los valores se consideran desiguales.
  - si los valores son del mismo tipo, se consideran iguales si:
    - no son números, y tienen el mismo **valor**.
    - si son números, si ambos:
      - tienen el mismo valor y no es **`NaN`**
      - si uno es `+0` y el otro es `-0` .

```
const num = 0;
const obj = new String("0");
const str = "0";

console.log(num === num);                 // true
console.log(obj === obj);                 // true
console.log(str === str);                 // true
console.log(null === null);               // true
console.log(undefined === undefined);     // true


console.log(num === obj);                 // false
console.log(num === str);                 // false
console.log(obj === str);                 // false
console.log(null === undefined);          // false
console.log(obj === null);                // false
console.log(obj === undefined);           // false
```

La igualdad estricta es casi siempre la operación de comparación correcta a utilizar.

- para todos los valores, excepto los números, utiliza la semántica obvia: un valor solo es igual a sí mismo.
- para los números, utiliza una **semántica** ligeramente diferente para pasar por alto dos casos extremos diferentes.
  - la primera es que **el punto flotante cero** tiene signo positivo o negativo.
    - esto es útil para representar ciertas soluciones matemáticas, pero como la mayoría de las situaciones no se preocupan por la diferencia entre `+0` y `-0`, la igualdad estricta los trata como el mismo valor.
  - la segunda es que el punto flotante incluye el concepto de un **valor no numérico**, `NaN`, para representar la solución a ciertos problemas matemáticos mal definidos:
    - **infinito negativo** sumado a **infinito positivo**, por ejemplo.
    - la igualdad estricta trata a `NaN` como desigual a cualquier otro valor : incluyéndose a sí mismo.
    - el único caso en el que `(x !== x)` es `true` es cuando `x` es `NaN` .

Además `===`

- la igualdad estricta también es utilizada por
  - **métodos de búsqueda de índices de array** que incluyen
    - `Array.prototype.indexOf()`
    - `Array.prototype.lastIndexOf()`
    - `TypedArray.prototype.index()`
    - `TypedArray.prototype.lastIndexOf()`,
  - y **comparación de tipo de fuente** `mayúscula/minúscula` .
- esto significa que no puedes usar, de forma útil, `indexOf( NaN ):`
  - para encontrar el índice de un valor `NaN` en una matriz
  - o usar `NaN` como un valor `case` en una sentencia `switch`

```
console.log([NaN].indexOf(NaN));       // -1

switch (NaN) {
   case NaN: console.log("Sorpresa"); // No se registra nada
}
```

Igualdad no estricta `==`

Igualdad no estricta:

- es *simétrica* :
  - `A == B` siempre tiene una semántica idéntica a `B == A` para cualquier valor de `A` y `B` (excepto por el orden de las conversiones aplicadas).
- la igualdad no extricta solo la usa el operador `==` .
- el comportamiento para realizar la igualdad no extricta usando `==` es el siguiente:

1. Si los operandos tienen el mismo **tipo,** se comparan de la siguiente manera:
   - Objeto: devuelve `true` solo si ambos operandos hacen referencia a la misma instancia de objeto.
   - Cadena: devuelve `true` solo si ambos operandos tienen los mismos caracteres en el mismo orden.
   - Número:
     - devuelve `true` solo si ambos operandos tienen el mismo **valor** .
     - `+0` y `-0` se tratan como el mismo valor.
     - si alguno de los operandos es `NaN`, devuelve `false` ; entonces `NaN` nunca es igual a `NaN` .
   - Booleano: devuelve `true` solo si los operandos son `true` o `false` .
   - BigInt: devuelve `true` solo si ambos operandos tienen el mismo valor.
   - Symbol: devuelve `true` solo si ambos operandos hacen referencia al mismo **símbolo** .

2. Tenemos que:
   - si uno de los operandos es `null` o `undefined,` el otro también debe ser `null` o `undefined` para devolver `true` .
   - de lo contrario, devuelve `false` .

3. Si uno de los operandos es un objeto y el otro es un tipo primitivo, se convierte el objeto en un primitivo .

4. En este paso, ambos operandos se son primitivos (uno de String, Number, Boolean, Symbol y BigInt). El resto de la conversión se realiza caso por caso.
   - Si son del mismo **tipo,** se comparan usando el paso 1.
   - Si uno de los operandos es un `Symbol` pero el otro no lo es, devuelve `false` .
   - Si uno de los operandos es un booleano pero el otro no lo es, se convierte el booleano en un número :
     - `true` se convierte en 1 y `false` se convierte en 0.
     - luego, se comparan los dos operandos nuevamente.
   - Número y cadena:
     - convertir la cadena en un número: un fallo de conversión da como resultado `NaN`, lo que garantizará que la comparación de igualdad sea `falsa` .
   - Número y BigInt:
     - comparar por su valor numérico.
     - si el número es `±Infinito` o `NaN`, devuelve `false` .
   - Cadena y BigInt:

- convierta la cadena en un BigInt utilizando el mismo algoritmo que el constructor `BigInt()` .
- si la conversión falla, devuelve `false` .

Tradicionalmente, y según ECMAScript,

- todas los **tipos primitivos** y **objetos** **`!=` (no iguales no estricto)** a `undefined` y `null` .
- pero la mayoría de los navegadores permiten que ciertos objetos "especiales" (son pocos) (específicamente, el objeto `document.all` para cualquier página), en algunos contextos, actúen como si *emularan* el valor `undefined` .
  - la igualdad no estricta es uno de esos contextos: `null == A` e `undefined == A` se evalúa como verdadero si, y solo si, A es un objeto que ***emula*** `undefined`
  - en todos los demás casos, un objeto nunca `==` a `undefined` o `null` .

En la mayoría de los casos, se desaconseja el uso de igualdad flexible.

- el resultado de una comparación usando igualdad estricta es más fácil de predecir,
- y puede evaluar más rápidamente debido a la falta de **coerción de tipos** .

El siguiente ejemplo muestra comparaciones de igualdad no estrictas que involucran el número primitivo `0`, el BigInt primitivo `0n`, el String primitivo `'0'` y un objeto cuyo valor `toString()` es `'0'` .

```
const num = 0;
const big = 0n;
const str = "0";
const obj = new String("0");

console.log(num == str); // true
console.log(big == num); // true
console.log(str == big); // true

console.log(num == obj); // true
console.log(big == obj); // true
console.log(str == obj); // true
```

Igualdad del mismo valor (`SameValue`) usando `Object.is()`

Igualdad del mismo valor (`SameValue`)

- determina si dos valores son ***funcionalmente idénticos*** en todos los contextos.
- se usa casi en todas partes en el lenguaje donde se espera un **valor de identidad equivalente** .
  - (Este caso de uso demuestra un ejemplo del principio de sustitución de Liskov).
- un ejemplo de esto es cuando se intenta mutar una propiedad inmutable:

```
// Agrega una propiedad NEGATIVE_ZERO inmutable al constructor Number.
Object.defineProperty(Number, "NEGATIVE_ZERO", {
  value: -0,
  writable: false,
  configurable: false,
  enumerable: false,
});

function intentarMutar(v) {
  Object.defineProperty(Number, "NEGATIVE_ZERO", { value: v });
}
```

- `Object.defineProperty` **generará una excepción** (***throw an exception***) al intentar cambiar una **propiedad inmutable,** pero no hace nada si no se solicita ningún cambio efectivo final : si `v` es `-0`, no se ha solicitado ningún cambio y no se generará ningún error.
- internamente, cuando se redefine una propiedad inmutable, el valor recién especificado se compara con el valor actual utilizando la igualdad del mismo valor.

```
intentarMutar(10);
▶Uncaught TypeError: Cannot redefine property: NEGATIVE_ZERO
    at Function.defineProperty (<anonymous>)
    at intentarMutar (<anonymous>:10:10)
    at <anonymous>:1:1
intentarMutar(-0);
undefined
```

Igualdad del mismo valor cero

Similar a la igualdad del mismo valor, pero +0 y -0 se consideran iguales.

- **la igualdad del mismo valor cero** no se expone como una API de JavaScript, pero se puede implementar con código personalizado:

```
function sameValueZero(x, y) {
  if (typeof x === "number" && typeof y === "number") {
    // x e y son iguales (puede que -0 y 0) o ambos son NaN
    return x === y || (x !== x && y !== y);
  }
  return x === y;
}
```

- mismo-valor-cero
  - solo difiere de **la igualdad estricta** al tratar a `NaN` como equivalente,
  - solo difiere de **la igualdad del mismo valor** al tratar `-0` como equivalente a `0` .
- esto hace que tenga el comportamiento más sentsato durante la **búsqueda,** especialmente cuando se trabaja con `NaN` .
  - se utiliza para **comparar la igualdad de claves** por ejemplo:
    - `Array.prototype.includes()`
    - `TypedArray.prototype.includes()`
    - así como los métodos de `Map` y `Set` .

Comparativa de métodos de igualdad

Sucede que hay gente que a menudo compara los operadores === y == diciendo que uno es una versión "mejorada" del otro.

- por ejemplo, el doble igual podría entenderse como una versión extendida del triple igual, porque el primero hace todo lo que hace el segundo, pero con conversión de tipos en sus operandos: por ejemplo, `6 == "6"` .
- alternativamente, se puede afirmar que el doble igual es la línea de base y el triple igual es una versión mejorada, porque requiere que los dos operandos sean del mismo tipo, por lo que agrega una restricción adicional.
- sin embargo, esta forma de pensar implica que las comparaciones de igualdad forman un "espectro" unidimensional donde "totalmente estricto" se encuentra en un extremo y "totalmente flexible" en el otro.
- este modelo se queda corto con `Object.is`, porque no es "más flexible" que el doble igual o "más estricto" que el triple igual, ni encaja en algún punto intermedio (es decir, siendo ambos más estrictos que el doble igual, pero más flexible que el triple igual).
- podemos ver en la siguiente tabla de comparaciones de igualdad que esto se debe a la forma en que `Object.is` maneja `NaN` .
  - observe que si `Object.is(NaN, NaN)` se evaluó como `false`,
    - podríamos decir *que* encaja en el espectro flexible/estricto como una forma aún más estricta de triple igual, una que distingue entre `-0` y `+0` .
    - el manejo de `NaN` significa que esto no es cierto.
  - Desafortunadamente, `Object.is` tiene que ser pensado en términos de sus características específicas, en lugar de su laxitud o rigurosidad con respecto a los operadores de igualdad.

X	y	==	===	Objeto.is	SameValueZero
undefined	undefined	✓ true	✓ true	✓ true	✓ true
null	null	✓ true	✓ true	✓ true	✓ true
true	true	✓ true	✓ true	✓ true	✓ true
false	false	✓ true	✓ true	✓ true	✓ true
'foo'	'foo'	✓ true	✓ true	✓ true	✓ true
0	0	✓ true	✓ true	✓ true	✓ true
+0	-0	✓ true	✓ true	✗ false	✓ true
+0	0	✓ true	✓ true	✓ true	✓ true
-0	0	✓ true	✓ true	✗ false	✓ true
0n	-0n	✓ true	✓ true	✓ true	✓ true
0	false	✓ true	✗ false	✗ false	✗ false
""	false	✓ true	✗ false	✗ false	✗ false
""	0	✓ true	✗ false	✗ false	✗ false
'0'	0	✓ true	✗ false	✗ false	✗ false
'17'	17	✓ true	✗ false	✗ false	✗ false
[1, 2]	'1,2'	✓ true	✗ false	✗ false	✗ false
new String('foo')	'foo'	✓ true	✗ false	✗ false	✗ false
null	undefined	✓ true	✗ false	✗ false	✗ false
null	false	✗ false	✗ false	✗ false	✗ false
undefined	false	✗ false	✗ false	✗ false	✗ false
{ foo: 'barra' }	{ foo: 'barra' }	✗ false	✗ false	✗ false	✗ false
new String('foo')	new String('foo')	✗ false	✗ false	✗ false	✗ false

X	y	==	===	Objeto.is	SameValueZero
0	null	✗ false	✗ false	✗ false	✗ false
0	NaN	✗ false	✗ false	✗ false	✗ false
'foo'	NaN	✗ false	✗ false	✗ false	✗ false
NaN	NaN	✗ false	✗ false	✓ true	✓ true

Cuándo usar `Object.is()` frente a triples iguales

En general, el único caso en que el comportamiento especial de `Object.is` hacia los ceros puede ser de interés es en la búsqueda de ciertos **esquemas de metaprogramación,** especialmente con respecto a los **descriptores de propiedades,** cuando es deseable para lo que tienes que hacer que refleje algunas de las características de `Object.defineProperty` .

- si este no es el caso, se sugiere evitar `Object.is` y usar `===` en su lugar.
- incluso si tus requisitos implican que las comparaciones entre dos valores de `NaN` se evalúen como `true`, por lo general, es más fácil aplicar casos especiales a las comprobaciones de `NaN` (usando el método `isNaN` disponible en versiones anteriores de ECMAScript) que determinar cómo los cálculos circundantes podrían afectar el signo de cualquier cero que encuentre en su comparación.

Aquí hay una lista no exhaustiva de **métodos integrados** y **operadores integrados** que pueden hacer que se manifieste una distinción entre `-0` y `+0` en su código:

`-` (negación unaria)

- considera el siguiente ejemplo:

```
const fuerzadetencion = obj.masa * -obj.velocidad;
```

Si `obj.velocidad` es `0` (o se calcula en `0`), se introduce un `-0` en ese lugar y se propaga hacia `fuerzadetencion`

```
const obj = {velocidad: 0, masa: 20};
undefined
const fuerzadetencion = obj.masa * -obj.velocidad;
undefined
fuerzadetencion;
-0
```

`Math.atan2`, `Math.ceil`, `Math.pow`, `Math.round`

- en algunos casos, es posible que se introduzca un `-0` en una expresión como valor de retorno de estos métodos incluso cuando no existe un `-0` como uno de los parámetros.
  - por ejemplo, usar `Math.pow` para elevar `-Infinity` a la potencia de cualquier exponente impar negativo se evalúa como `-0` .

`Math.floor`, `Math.max`, `Math.min`, `Math.sin`, `Math.sqrt`, `Math.tan`

- es posible obtener un valor de retorno `-0` de estos métodos en algunos casos donde existe un `-0` como uno de los parámetros.
  - por ejemplo, `Math.min(-0, +0)` evalúa a `-0` .

`~`, `<<`, `>>`

- cada uno de estos operadores utiliza el **algoritmo ToInt32** internamente.
- dado que solo hay una representación para 0 en el tipo de entero interno de 32 bits, `-0` no sobrevivirá a un viaje de ida y vuelta después de una operación inversa.
  - por ejemplo, tanto `Object.is(~~(-0), -0)` como `Object.is(-0 << 2 >> 2, -0)` evaluar a `false` .

Confiar en `Object.is` cuando no se tiene en cuenta el **signo de los ceros puede ser peligroso** : por supuesto, cuando la intención es distinguir entre `-0` y `+0`, hace exactamente lo que se desea.

Advertencia: `Object.is()` y `NaN`

La especificación `Object.is` trata todas las instancias de `NaN` como el mismo objeto.

- sin embargo, dado que los Typed Arrays están disponibles, podemos tener distintas **representaciones de punto flotante** de `NaN` que no se comportan de manera idéntica en todos los contextos.
- esto sólo es relevante al trabajar con TypedArrays
- por ejemplo:

```
const f2b = (x) => new Uint8Array(new Float64Array([x]).buffer);
const b2f = (x) => new Float64Array(x.buffer)[0];

// Obtener una representación de bytes de NaN
const n = f2b(NaN);

// Cambia el primer bit, que es el bit de signo y no importa para NaN
n[7] = 255;
const nan2 = b2f(n);
console.log(nan2);                  // NaN
console.log(Object.is(nan2, NaN));  // true
console.log(f2b(NaN));              // Uint8Array(8) [0, 0, 0, 0, 0, 0, 248, 127]
console.log(f2b(nan2));             // Uint8Array(8) [1, 0, 0, 0, 0, 0, 248, 127]
```

NaN = 7F F8 00 00 00 00 00 00
127 248

-NaN = FF F8 00 00 00 00 00 00
255 248