Equality comparisons and sameness

JavaScript provides three different value-comparison operations:

- `===` — strict equality (triple equals)
- `==` — loose equality (double equals)
- `Object.is()`

Which operation you choose depends on what sort of comparison you are looking to perform. Briefly:

- Double equals (`==`) will perform a type conversion when comparing two things, and will handle `NaN`, `-0`, and `+0` specially to conform to IEEE 754 (so `NaN != NaN`, and `-0 == +0`);
- Triple equals (`===`) will do the same comparison as double equals (including the special handling for `NaN`, `-0`, and `+0`) but without type conversion; if the types differ, `false` is returned.
- `Object.is()` does no type conversion and no special handling for `NaN`, `-0`, and `+0` (giving it the same behavior as `===` except on those special numeric values).

They correspond to three of four equality algorithms in JavaScript:

- IsLooselyEqual: `==`
- IsStrictlyEqual: `===`
- SameValue: `Object.is()`
- SameValueZero: used by many built-in operations

Note that the distinction between these all have to do with their handling of primitives; none of them compares whether the parameters are conceptually similar in structure. For any non-primitive objects `x` and `y` which have the same structure but are distinct objects themselves, all of the above forms will evaluate to `false`.

Strict equality using ===

Strict equality compares two values for equality. Neither value is implicitly converted to some other value before being compared. If the values have different types, the values are considered unequal. If the values have the same type, are not numbers, and have the same value, they're

considered equal. Finally, if both values are numbers, they're considered equal if they're both not `NaN` and are the same value, or if one is `+0` and one is `-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(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
```

Strict equality is almost always the correct comparison operation to use. For all values except numbers, it uses the obvious semantics: a value is only equal to itself. For numbers it uses slightly different semantics to gloss over two different edge cases. The first is that floating point zero is either positively or negatively signed. This is useful in representing certain mathematical solutions, but as most situations don't care about the difference between `+0` and `-0`, strict equality treats them as the same value. The second is that floating point includes the concept of a not-a-number value, `NaN`, to represent the solution to certain ill-defined mathematical problems: negative infinity added to positive infinity, for example. Strict equality treats `NaN` as unequal to every other value — including itself. (The only case in which (`x !== x`) is true is when `x` is `NaN`.)

Besides `===`, strict equality is also used by array index-finding methods including `Array.prototype.indexOf()`,`Array.prototype.lastIndexOf()`,`TypedArray.prototype.indexOf()` or `TypedArray.prototype.lastIndexOf()`, and case-matching. This means you cannot use `indexOf(NaN)` to find the index of a `NaN` value in an array, or use `NaN` as a case value in a switch statement and make it match anything.

```
console.log([NaN].indexOf(NaN)); // -1
switch (NaN) {
  case NaN:
    console.log("Surprise"); // Nothing is logged
}
```

Loose equality using ==

Loose equality is *symmetric*: `A == B` always has identical semantics to `B == A` for any values of A and B (except for the order of applied conversions). The behavior for performing loose equality using `==` is as follows:

1. If the operands have the same type, they are compared as follows:
    - Object: return `true` only if both operands reference the same object.
    - String: return `true` only if both operands have the same characters in the same order.
    - Number: return `true` only if both operands have the same value. `+0` and `-0` are treated as the same value. If either operand is `NaN`, return `false`; so `NaN` is never equal to `NaN`.
    - Boolean: return true only if operands are both true or both false.
    - BigInt: return true only if both operands have the same value.
    - Symbol: return true only if both operands reference the same symbol.

2. If one of the operands is `null` or `undefined`, the other must also be `null` or `undefined` to return `true`. Otherwise return `false`.

3. If one of the operands is an object and the other is a primitive, convert the object to a primitive.

4. At this step, both operands are converted to primitives (one of String, Number, Boolean, Symbol, and BigInt). The rest of the conversion is done case-by-case.
    - If they are of the same type, compare them using step 1.
    - If one of the operands is a Symbol but the other is not, return `false`.
    - If one of the operands is a Boolean but the other is not, convert the boolean to a number: `true` is converted to 1, and `false` is converted to 0. Then compare the two operands loosely again.
    - Number to String: convert the string to a number. Conversion failure results in `NaN`, which will guarantee the equality to be `false`.
    - Number to BigInt: compare by their numeric value. If the number is ±Infinity or `NaN`, return `false`.
    - String to BigInt: convert the string to a BigInt using the same algorithm as the BigInt() constructor. If conversion fails, return `false`.

Traditionally, and according to ECMAScript, all primitives and objects are loosely unequal to `undefined` and `null`. But most browsers permit a very narrow class of objects (specifically, the document.all object for any page), in some contexts, to act as if they emulate the value

`undefined`. Loose equality is one such context: `null == A` and `undefined == A` evaluate to `true` if, and only if, `A` is an object that emulates `undefined`. In all other cases an object is never loosely equal to `undefined` or `null`.

In most cases, using loose equality is discouraged. The result of a comparison using strict equality is easier to predict, and may evaluate more quickly due to the lack of type coercion.

The following example demonstrates loose equality comparisons involving the number primitive `0`, the bigint primitive `0n`, the string primitive '`0`', and an object whose `toString()` value is '`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
```

Loose equality is only used by the `==` operator.

Same-value equality using Object.is()

Same-value equality determines whether two values are functionally identical in all contexts. (This use case demonstrates an instance of the Liskov substitution principle.) One instance occurs when an attempt is made to mutate an immutable property:

```
// Add an immutable NEGATIVE_ZERO property to the Number constructor.
Object.defineProperty(Number, "NEGATIVE_ZERO", {
  value: -0,
  writable: false,
  configurable: false,
  enumerable: false,
});
function attemptMutation(v) {
  Object.defineProperty(Number, "NEGATIVE_ZERO", { value: v });
}
```

`Object.defineProperty` will throw an exception when attempting to change an immutable property, but it does nothing if no actual change is requested. If `v` is `-0`, no change has been requested, and no error will be thrown. Internally, when an immutable property is redefined, the newly-specified value is compared against the current value using same-value equality.

Same-value equality is provided by the `Object.is` method. It's used almost everywhere in the language where a value of equivalent identity is expected.

Same-value equality using Object.is()

Similar to same-value equality, but +0 and -0 are considered equal.

Same-value-zero equality is not exposed as a JavaScript API, but can be implemented with custom code:

```
function sameValueZero(x, y) {
  if (typeof x === "number" && typeof y === "number") {
    // x and y are equal (may be -0 and 0) or they are both NaN
    return x === y || (x !== x && y !== y);
  }
  return x === y;
}
```

Same-value-zero only differs from strict equality by treating `NaN` as equivalent, and only differs from same-value equality by treating `-0` as equivalent to `0`. This makes it usually have the most sensible behavior during searching, especially when working with `NaN`. It's used by `Array.prototype.includes()`, `TypedArray.prototype.includes()`, as well as `Map` and `Set` methods for comparing key equality.

Comparing equality methods

People often compare double equals and triple equals by saying one is an "enhanced" version of the other. For example, double equals could be said as an extended version of triple equals, because the former does everything that the latter does, but with type conversion on its operands — for example, `6 == "6"`. Alternatively, it can be claimed that double equals is the baseline, and triple equals is an enhanced version, because it requires the two operands to be the same type, so it adds an extra constraint.

However, this way of thinking implies that the equality comparisons form a one-dimensional "spectrum" where "totally strict" lies on one end and "totally loose" lies on the other. This model falls short with `Object.is`, because it isn't "looser" than double equals or "stricter" than triple equals, nor does it fit somewhere in between (i.e., being both stricter than double equals, but looser than triple equals). We can see from the sameness comparisons table below that this is due to the way that `Object.is` handles `NaN`. Notice that if `Object.is(NaN, NaN)` evaluated to `false`, we could say that it fits on the loose/strict spectrum as an even stricter form of triple equals, one that distinguishes between `-0` and `+0`. The `NaN` handling means this is untrue, however. Unfortunately, `Object.is` has to be thought of in terms of its specific characteristics, rather than its looseness or strictness with regard to the equality operators.

x	y	==	===	Object.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: 'bar' }	{ foo: 'bar' }	✗ false	✗ false	✗ false	✗ false
new String('foo')	new String('foo')	✗ false	✗ false	✗ false	✗ false
0	null	✗ false	✗ false	✗ false	✗ false
0	NaN	✗ false	✗ false	✗ false	✗ false
'foo'	NaN	✗ false	✗ false	✗ false	✗ false
NaN	NaN	✗ false	✗ false	☑ true	☑ true

When to use Object.is() versus triple equals

In general, the only time `Object.is`'s special behavior towards zeros is likely to be of interest is in the pursuit of certain meta-programming schemes, especially regarding property descriptors, when it is desirable for your work to mirror some of the characteristics of `Object.defineProperty`. If your use case does not require this, it is suggested to avoid `Object.is` and use `===` instead. Even if your requirements involve having comparisons between two `NaN` values evaluate to true, generally it is easier to special-case the `NaN` checks (using the `isNaN` method available from previous versions of ECMAScript) than it is to work out how surrounding computations might affect the sign of any zeros you encounter in your comparison.

Here's a non-exhaustive list of built-in methods and operators that might cause a distinction between `-0` and `+0` to manifest itself in your code:

- (unary negation)

    Consider the following example:

    ```
    const stoppingForce = obj.mass * -obj.velocity;
    ```

    If `obj.velocity` is `0` (or computes to `0`), a `-0` is introduced at that place and propagates out into `stoppingForce`.

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

: In some cases, it's possible for a `-0` to be introduced into an expression as a return value of these methods even when no `-0` exists as one of the parameters. For example, using `Math.pow` to raise `-Infinity` to the power of any negative, odd exponent evaluates to `-0`. Refer to the documentation for the individual methods.

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

: It's possible to get a `-0` return value out of these methods in some cases where a `-0` exists as one of the parameters. E.g., `Math.min(-0, +0)` evaluates to `-0`. Refer to the documentation for the individual methods.

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

: Each of these operators uses the ToInt32 algorithm internally. Since there is only one representation for 0 in the internal 32-bit integer type, -0 will not survive a round trip after an inverse operation. E.g., both Object.is(~~(-0), -0) and Object.is(-0 << 2 >> 2, -0) evaluate to false.

Relying on `Object.is` when the signedness of zeros is not taken into account can be hazardous. Of course, when the intent is to distinguish between `-0` and `+0`, it does exactly what's desired.

Caveat: Object.is() and NaN

The `Object.is` specification treats all instances of `NaN` as the same object. However, since typed arrays are available, we can have distinct floating point representations of `NaN` which don't behave identically in all contexts. For example:

```
const f2b = (x) => new Uint8Array(new Float64Array([x]).buffer);
const b2f = (x) => new Float64Array(x.buffer)[0];
// Get a byte representation of NaN
const n = f2b(NaN);
// Change the first bit, which is the sign bit and doesn't matter for NaN
n[0] = 1;
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]
```