Equality comparisons and sameness

There are four equality algorithms in ES2015:

- **Abstract Equality Comparison (==)**
- **Strict Equality Comparison (===): used by `Array.prototype.indexOf`, `Array.prototype.lastIndexOf`, and `case`-matching**
- **SameValueZero: used by `%TypedArray%` and `ArrayBuffer` constructors, as well as `Map` and `Set` operations, and up-coming `String.prototype.includes` in ES2016**
- **SameValue: used in all other places**

JavaScript provides three different value-comparison operations:

- **strict equality (or "triple equals" or "identity") using `===`,**
- **loose equality ("double equals") using `==`,**
- **and `Object.is` (new in ECMAScript 2015).**

The choice of which operation to use depends on what sort of comparison you are looking to perform.

Briefly, double equals will perform a type conversion when comparing two things; triple equals will do the same comparison without type conversion (by simply always returning false if the types differ); and `Object.is` will behave the same way as triple equals, but with special handling for `NaN` and `-0` and `+0` so that the last two are not said to be the same, while `Object.is(NaN, NaN)` will be `true`. (Comparing `NaN` with `NaN` ordinarily—i.e., using either double equals or triple equals—evaluates to `false`, because IEEE 754 says so.) Do 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. Otherwise, if the values have the same type and are not numbers, they're considered equal if they have the same value. 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`.

```
var num = 0;
var obj = new String('0');
var 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`.)

Loose equality using ==

Loose equality compares two values for equality, *after* converting both values to a common type. After conversions (one or both sides may undergo conversions), the final equality comparison is performed exactly as === performs it. 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 equality comparison is performed as follows for operands of the various types:

	Operand B

		Operand B				
		Undefined	Null	Number	String	Boo
Operand A	Undefined	`true`	`true`	`false`	`false`	fa
	Null	`true`	`true`	`false`	`false`	fa
	Number	`false`	`false`	`A === B`	`A === ToNumber(B)`	A = ToNuml
	String	`false`	`false`	`ToNumber(A) === B`	`A === B`	ToNuml = ToNuml
	Boolean	`false`	`false`	`ToNumber(A) === B`	`ToNumber(A) === ToNumber(B)`	A =
	Object	`false`	`false`	`ToPrimitive(A) == B`	`ToPrimitive(A) == B`	ToPrimi == ToNu

In the above table, `ToNumber(A)` attempts to convert its argument to a number before comparison. Its behavior is equivalent to +A (the unary + operator). `ToPrimitive(A)` attempts to convert its object argument to a primitive value, by attempting to invoke varying sequences of `A.toString` and `A.valueOf` methods on A.

Traditionally, and according to ECMAScript, all 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`.

```
var num = 0;
var obj = new String('0');
var str = '0';

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

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

```

```
console.log(null == undefined); // true

// both false, except in rare cases
console.log(obj == null);
console.log(obj == undefined);
```

Some developers consider that it is pretty much never a good idea to use loose equality. The result of a comparison using strict equality is easier to predict, and as no type coercion takes place the evaluation may be faster.

Same-value equality

Same-value equality addresses a final use case: determining 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: fal

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

`Object.defineProperty` will throw an exception when attempting to change an immutable property would actually change it, 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. But if v is `+0`, `Number.NEGATIVE_ZERO` would no longer have its immutable value. 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.

Same-value-zero equality

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

Abstract equality, strict equality, and same value in the specification

In ES5, the comparison performed by == is described in Section 11.9.3, The Abstract Equality Algorithm. The === comparison is 11.9.6, The Strict Equality Algorithm. (Go look at these. They're brief and readable. Hint: read the strict equality algorithm first.) ES5 also describes, in

Section 9.12, The SameValue Algorithm for use internally by the JS engine. It's largely the same as the Strict Equality Algorithm, except that 11.9.6.4 and 9.12.4 differ in handling `Number`s. ES2015 simply proposes to expose this algorithm through `Object.is`.

We can see that with double and triple equals, with the exception of doing a type check upfront in 11.9.6.1, the Strict Equality Algorithm is a subset of the Abstract Equality Algorithm, because 11.9.6.2–7 correspond to 11.9.3.1.a–f.

A model for understanding equality comparisons?

Prior to ES2015, you might have said of double equals and triple equals that one is an "enhanced" version of the other. For example, someone might say that double equals is an extended version of triple equals, because the former does everything that the latter does, but with type conversion on its operands. E.g., `6 == "6"`. (Alternatively, someone might say 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. Which one is the better model for understanding depends on how you choose to view things.)

However, this way of thinking about the built-in sameness operators is not a model that can be stretched to allow a place for ES2015's `Object.is` on this "spectrum". `Object.is` isn't simply "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` simply has to be thought of in terms of its specific characteristics, rather than its looseness or strictness with regard to the equality operators.

Sameness Comparisons

x	y	==	===	Object.is
undefined	undefined	true	true	true
null	null	true	true	true
true	true	true	true	true
false	false	true	true	true
'foo'	'foo'	true	true	true
0	0	true	true	true
+0	-0	true	true	false

x	y	==	===	Object.is
0	false	true	false	false
""	false	true	false	false
""	0	true	false	false
'0'	0	true	false	false
'17'	17	true	false	false
[1, 2]	'1, 2'	true	false	false
new String('foo')	'foo'	true	false	false
null	undefined	true	false	false
null	false	false	false	false
undefined	false	false	false	false
{ foo: 'bar' }	{ foo: 'bar' }	false	false	false
new String('foo')	new String('foo')	false	false	false
0	null	false	false	false
0	NaN	false	false	false
'foo'	NaN	false	false	false
NaN	NaN	false	false	true

When to use `Object.is` versus triple equals

Aside from the way it treats `NaN`, generally, 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 an in-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)`

It's obvious that negating `0` produces `-0`. But the abstraction of an expression can cause `-0` to creep in when you don't realize it. For example, consider:

```
1 | let stoppingForce = obj.mass * -obj.velocity;
```

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

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

It's possible for a `-0` to be introduced into an expression as a return value of these methods in some cases, even when no `-0` exists as one of the parameters. E.g., 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)` evalutes 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.

See also

- JS Comparison Table