2nd December 2019

Comparisons

We know many comparison operators from maths:

- Greater/less than: `a > b`, `a < b`.
- Greater/less than or equals: `a >= b`, `a <= b`.
- Equals: `a == b` (please note the double equals sign `=`. A single symbol `a = b` would mean an assignment).
- Not equals. In maths the notation is ≠, but in JavaScript it's written as an assignment with an exclamation sign before it: `a != b`.

Boolean is the result

Like all other operators, a comparison returns a value. In this case, the value is a boolean.

- `true` – means "yes", "correct" or "the truth".
- `false` – means "no", "wrong" or "not the truth".

For example:

```
1  alert( 2 > 1 );  // true (correct)
2  alert( 2 == 1 ); // false (wrong)
3  alert( 2 != 1 ); // true (correct)
```

A comparison result can be assigned to a variable, just like any value:

```
1  let result = 5 > 4; // assign the result of the comparison
2  alert( result ); // true
```

String comparison

To see whether a string is greater than another, JavaScript uses the so-called "dictionary" or "lexicographical" order.

In other words, strings are compared letter-by-letter.

For example:

```
1  alert( 'Z' > 'A' ); // true
2  alert( 'Glow' > 'Glee' ); // true
```

```
3  alert( 'Bee' > 'Be' ); // true
```

The algorithm to compare two strings is simple:

1. Compare the first character of both strings.
2. If the first character from the first string is greater (or less) than the other string's, then the first string is greater (or less) than the second. We're done.
3. Otherwise, if both strings' first characters are the same, compare the second characters the same way.
4. Repeat until the end of either string.
5. If both strings end at the same length, then they are equal. Otherwise, the longer string is greater.

In the examples above, the comparison `'Z' > 'A'` gets to a result at the first step while the strings `"Glow"` and `"Glee"` are compared character-by-character:

1. `G` is the same as `G`.
2. `l` is the same as `l`.
3. `o` is greater than `e`. Stop here. The first string is greater.

Not a real dictionary, but Unicode order

The comparison algorithm given above is roughly equivalent to the one used in dictionaries or phone books, but it's not exactly the same.

For instance, case matters. A capital letter `"A"` is not equal to the lowercase `"a"`. Which one is greater? The lowercase `"a"`. Why? Because the lowercase character has a greater index in the internal encoding table JavaScript uses (Unicode). We'll get back to specific details and consequences of this in the chapter Strings.

Comparison of different types

When comparing values of different types, JavaScript converts the values to numbers.

For example:

```
1  alert( '2' > 1 ); // true, string '2' becomes a number 2
2  alert( '01' == 1 ); // true, string '01' becomes a number 1
```

For boolean values, `true` becomes `1` and `false` becomes `0`.

For example:

```
1  alert( true == 1 ); // true
2  alert( false == 0 ); // true
```

A funny consequence

It is possible that at the same time:

- Two values are equal.
- One of them is `true` as a boolean and the other one is `false` as a boolean.

For example:

```
let a = 0;
alert( Boolean(a) ); // false

let b = "0";
alert( Boolean(b) ); // true

alert(a == b); // true!
```

From JavaScript's standpoint, this result is quite normal. An equality check converts values using the numeric conversion (hence `"0"` becomes `0`), while the explicit `Boolean` conversion uses another set of rules.

Strict equality

A regular equality check `==` has a problem. It cannot differentiate `0` from `false`:

```
alert( 0 == false ); // true
```

The same thing happens with an empty string:

```
alert( '' == false ); // true
```

This happens because operands of different types are converted to numbers by the equality operator `==`. An empty string, just like `false`, becomes a zero.

What to do if we'd like to differentiate `0` from `false`?

A strict equality operator `===` checks the equality without type conversion.

In other words, if `a` and `b` are of different types, then `a === b` immediately returns `false` without an attempt to convert them.

Let's try it:

```
alert( 0 === false ); // false, because the types are different
```

There is also a "strict non-equality" operator `!==` analogous to `!=`.

The strict equality operator is a bit longer to write, but makes it obvious what's going on and leaves less room for errors.

Comparison with null and undefined

There's a non-intuitive behavior when `null` or `undefined` are compared to other values.

For a strict equality check `===`

These values are different, because each of them is a different type.

```
1  alert( null === undefined ); // false
```

For a non-strict check `==`

There's a special rule. These two are a "sweet couple": they equal each other (in the sense of `==`), but not any other value.

```
1  alert( null == undefined ); // true
```

For maths and other comparisons `< > <= >=`

`null/undefined` are converted to numbers: `null` becomes `0`, while `undefined` becomes `NaN`.

Now let's see some funny things that happen when we apply these rules. And, what's more important, how to not fall into a trap with them.

Strange result: null vs 0

Let's compare `null` with a zero:

```
1  alert( null > 0 );  // (1) false
2  alert( null == 0 ); // (2) false
3  alert( null >= 0 ); // (3) true
```

Mathematically, that's strange. The last result states that "`null` is greater than or equal to zero", so in one of the comparisons above it must be `true`, but they are both false.

The reason is that an equality check `==` and comparisons `> < >= <=` work differently. Comparisons convert `null` to a number, treating it as `0`. That's why (3) `null >= 0` is true and (1) `null > 0` is false.

On the other hand, the equality check `==` for `undefined` and `null` is defined such that, without any conversions, they equal each other and don't equal anything else. That's why (2) `null == 0` is false.

An incomparable undefined

The value `undefined` shouldn't be compared to other values:

```
1  alert( undefined > 0 ); // false (1)
2  alert( undefined < 0 ); // false (2)
3  alert( undefined == 0 ); // false (3)
```

Why does it dislike zero so much? Always false!

We get these results because:

- Comparisons `(1)` and `(2)` return `false` because `undefined` gets converted to `NaN` and `NaN` is a special numeric value which returns `false` for all comparisons.
- The equality check `(3)` returns `false` because `undefined` only equals `null`, `undefined`, and no other value.

Evade problems

Why did we go over these examples? Should we remember these peculiarities all the time? Well, not really. Actually, these tricky things will gradually become familiar over time, but there's a solid way to evade problems with them:

Just treat any comparison with `undefined/null` except the strict equality `===` with exceptional care.

Don't use comparisons `>= > < <=` with a variable which may be `null/undefined`, unless you're really sure of what you're doing. If a variable can have these values, check for them separately.

Summary

- Comparison operators return a boolean value.
- Strings are compared letter-by-letter in the "dictionary" order.
- When values of different types are compared, they get converted to numbers (with the exclusion of a strict equality check).
- The values `null` and `undefined` equal `==` each other and do not equal any other value.
- Be careful when using comparisons like `>` or `<` with variables that can occasionally be `null/undefined`. Checking for `null/undefined` separately is a good idea.

Tasks

Comparisons

importance: 5

What will be the result for these expressions?

```
1  5 > 4
2  "apple" > "pineapple"
3  "2" > "12"
4  undefined == null
5  undefined === null
6  null == "\n0\n"
7  null === +"\n0\n"
```

solution

Previous lesson Next lesson

Share

Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)