**Comparisons**

Many comparison operators we know from maths:

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

**[Boolean is the result](https://javascript.info/comparison" \l "boolean-is-the-result)**

Just as all other operators, a comparison returns a value. The value is of the boolean type.

* true – means “yes”, “correct” or “the truth”.
* false – means “no”, “wrong” or “a lie”.

For example:

alert( 2 > 1 ); // true (correct)

alert( 2 == 1 ); // false (wrong)

alert( 2 != 1 ); // true (correct)

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

let result = 5 > 4; // assign the result of the comparison

alert( result ); // true

**[String comparison](https://javascript.info/comparison" \l "string-comparison)**

To see which string is greater than the other, the so-called “dictionary” or “lexicographical” order is used.

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

For example:

alert( 'Z' > 'A' ); // true

alert( 'Glow' > 'Glee' ); // true

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

The algorithm to compare two strings is simple:

1. Compare first characters of both strings.
2. If the first one is greater(or less), then the first string is greater(or less) than the second. We’re done.
3. Otherwise if first characters are equal, compare the second characters the same way.
4. Repeat until the end of any string.
5. If both strings ended simultaneously, then they are equal. Otherwise the longer string is greater.

In the example above, the comparison 'Z' > 'A' gets the result at the first step.

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 book 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? Actually, the lowercase "a" is. Why? Because the lowercase character has a greater index in the internal encoding table (Unicode).

**[Comparison of different types](https://javascript.info/comparison" \l "comparison-of-different-types)**

When compared values belong to different types, they are converted to numbers.

For example:

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

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

For boolean values, true becomes 1 and false becomes 0, that’s why:

alert( true == 1 ); // true

alert( false == 0 ); // true

[**Strict equality**](https://javascript.info/comparison#strict-equality)

A regular equality check "==" has a problem. It cannot differ 0 from false:

alert( 0 == false ); // true

The same thing with an empty string:

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

That’s because operands of different types are converted to a number by the assignment 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 also exists a “strict non-equality” operator !==, as an analogy for !=.

The string equality check operator is a bit longer to write, but makes it obvious what’s going on and leaves less space for errors.

**[Comparison with null and undefined](https://javascript.info/comparison" \l "comparison-with-null-and-undefined)**

Let’s see more edge cases.

There’s a non-intuitive behavior when null or undefined is compared with other values.

**For a strict equality check ===**

These values are different, because each of them belong to a separate type of it’s own.

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 no any other value.

alert( null == undefined ); // true

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

Values null/undefined are converted to a number: **null** becomes **0**, while **undefined** becomes **NaN**.

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

**[Strange result: null vs 0](https://javascript.info/comparison" \l "strange-result-null-vs-0)**

Let’s compare null with a zero:

alert( null > 0 ); // (1) false

alert( null == 0 ); // (2) false

alert( null >= 0 ); // (3) true

Yeah, mathematically that’s strange. The last result states that "null is equal or greater than zero". Then one of the comparisons above must be correct, but they are both falsy.

The reason is that an equality check == and comparisons > < >= <= work differently. Comparisons convert null to a number, hence treat 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 works by the rule, without any conversions. They equal each other and don’t equal anything else. That’s why (2) null == 0 is false.

**[An incomparable undefined](https://javascript.info/comparison" \l "an-incomparable-undefined)**

The value undefined shouldn’t participate in comparisons at all:

alert( undefined > 0 ); // false (1)

alert( undefined < 0 ); // false (2)

alert( undefined == 0 ); // false (3)

Why does it dislike a zero so much? Always false!

We’ve got 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 and no other value.

**[Evade problems](https://javascript.info/comparison" \l "evade-problems)**

Why did we observe these examples? Should we remember these pecularities all the time? Well, not really. Actually, these tricky things will gradually become familiar over the time, but there’s a solid way to evade any 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 are really sure what you’re doing. If a variable can have such values, then check for them separately.

**[Summary](https://javascript.info/comparison" \l "summary)**

* Comparison operators return a logical 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).
* 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. Making a separate check for null/undefined is a good idea.

**Questions and Exercises**

* 1. What will be the result for expressions?

5 > 4; // true

"apple" > "pineapple"; // false (ASCII Value)

"2" > "12"; // true

undefined == null; // true

undefined === null; // false (not same type)

null == "\n0\n"; // false

null === +"\n0\n"; // false

* 1. Create a program to alert the results of the above code block and use this to check your answers.