

# Data Types and Conditionals

<https://javascript.info/types>

## Data types

There are 8 basic types in Javascript.

We can put any type in a variable. For example, a variable can at one moment be a string and then store a number:

```
let message = "hello";  
message = 123456;
```

Programming languages that allow such things, such as JavaScript, are called "dynamically typed", meaning that there exist data types, but variables are not bound to any of them.

## Number

```
let n = 123;  
n = 12.345;
```

The *number* type represents both integer and floating point numbers.

If there's a `NaN` somewhere in a mathematical expression, it propagates to the whole result (there's only one exception to that: `NaN ** 0` is `1`).

## Mathematical operations are safe

Doing maths is "safe" in JavaScript. We can do anything: divide by zero, treat non-numeric strings as numbers, etc.

The script will never stop with a fatal error ("die"). At worst, we'll get `NaN` as the result.

## BigInt

In JavaScript, the "number" type cannot safely represent integer values larger than  $(2^{53} - 1)$  (a little over 9 quadrillion), or less than  $-(2^{53} - 1)$  for negatives.

To be really precise, the "number" type can store larger integers (up to  $1.7976931348623157 \times 10^{308}$ ), but outside of the safe integer range  $\pm(2^{53} - 1)$  there'll be a precision error, because not all digits fit into the fixed 64-bit storage. So an "approximate" value may be stored.

For example, these two numbers (right above the safe range) are the same:

```
console.log(9007199254740991 + 1); // 9007199254740992
console.log(9007199254740991 + 2); // 9007199254740992
console.log(9007199254740991 + 1 == 9007199254740991 + 2); // true
```

Usually  $\pm(2^{53} - 1)$  range is quite enough, but sometimes we need the entire range of really big integers, e.g. for cryptography or microsecond-precision timestamps.

The `BigInt` type was recently added to the language to represent integers of arbitrary length.

A `BigInt` value is created by appending `n` to the end of an integer:

```
const bigInt = 1234567890123456789012345678901234567890n;
typeof(bigInt); // bigint
```

`BigInt`s are rarely needed.

## String

A string in JavaScript must be surrounded by quotes:

```
let str = "Hello";
let str2 = 'Single quotes are ok too';
let phrase = `can embed another ${str}`;
```

There are 3 types of quotes in JavaScript:

1. Double quotes: `"Hello"`.
2. Single quotes: `'Hello'`.
3. Backticks: ``Hello``. (The spaces at the beginning and end are just because of formatting in a markdown file)

Double and single quotes are "simple" quotes. There's practically no difference between them in JavaScript.

Backticks are "extended functionality" quotes. They allow us to embed variables and expressions into a string by wrapping them in `${...}` (similar to an f-string in Python), for example:

```
let name = "John";

// embed a variable
alert(`Hello, ${name}!`); // Hello, John!

// embed an expression
alert(`the result is ${1 + 2}`); // the result is 3
```

The expression inside `${...}` is evaluated and the result becomes part of the string. We can put anything in there: a variable like `name` or an arithmetical expression like `1 + 2` or something more complex.

Note that this can only be done in backticks. Other quotes don't have this embedding functionality!

```
alert( "the result is ${1 + 2}" ); // the result is ${1 + 2} (double quotes do nothing)
```

There is no character type.

## Boolean (logical type)

The boolean type has only two values: `true` and `false`.

An example of uses of booleans:

```
let nameFieldChecked = true; // yes, name field is checked
let ageFieldChecked = false; // no, age field is not checked
```

## The "null" value

The special `null` value does not belong to any of the types described above.

It forms a separate type of its own which contains only the `null` value:

```
let age = null;
```

In JavaScript, `null` is not a "reference to a non-existing object" or a "null pointer" like in some other languages.

It's just a special value which represents "nothing", "empty", or "value unknown" (like `None` in Python).

The code above states that `age` is unknown.

## The "undefined value"

The special value `undefined` also stands apart. It makes a type of its own, just like `null`.

The meaning of `undefined` is "value is not assigned".

If a variable is declared, but not assigned, then its value is `undefined`:

```
let age;  
alert(age); // shows "undefined"
```

Technically, you can explicitly assign `undefined` to a variable:

```
let age = 100;  
// Change the value to undefined  
age = undefined;  
alert(age); // undefined
```

But we **don't** recommend doing that. Normally, use `null` to assign an "empty" value to a variable. `undefined` is reserved as the default initial value for unassigned things.

## Objects and Symbols

The `object` type is special.

All other types are "primitive" meaning their values can only contain a single thing (be it a string or a number or whatever). In contrast, objects are used to store collections of data and more complex entities.

The `symbol` type is used to create unique identifiers for objects.

## The typeof operator

The `typeof` operator returns the type of the operand. It's useful when we want to process values of different types differently or just want to do a quick check.

A call to `typeof x` returns a string with the type name:

```
1  typeof undefined // "undefined"
2
3  typeof 0 // "number"
4
5  typeof 10n // "bigint"
6
7  typeof true // "boolean"
8
9  typeof "foo" // "string"
10
11 typeof Symbol("id") // "symbol"
12
13 typeof Math // "object" (1)
14
15 typeof null // "object" (2)
16
17 typeof alert // "function" (3)
```

The last three lines may need additional explanation:

1. `Math` is a built-in object that provides mathematical operations.
2. The result of `typeof null` is "object". That's an **officially recognized error** in `typeof`, coming from very early days of JavaScript and kept for compatibility. Definitely, `null` is not an object. It is a special value with a separate type of its own. The behavior of `typeof` is **wrong** here.
3. The result of `typeof alert` is "function", because `alert` is a function. There's no special "function" type in JavaScript. Functions belong to the object type. But `typeof` treats them differently, returning "function". That also comes from the early days of JavaScript. Technically, such behavior isn't correct, but can be convenient in practice.

## The typeof(x) syntax

You may come across another syntax: `typeof(x)` . It's the same as `typeof x` .

To put it clear, `typeof` is an operator (just like `/` for instance), not a function. The parentheses here aren't part of `typeof` . It's the kind of parentheses used for mathematical grouping.

## Summary

The 8 types are:

1. number
  2. bigint
  3. string
  4. boolean
  5. null
  6. undefined
  7. symbol
  8. object
- 

[https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First\\_steps/Strings](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/Strings)

## Handling text - strings in JavaScript

### The power of words

Since the Web is a largely text-based medium designed to allow humans to communicate and share information, it is useful for us to have control over the words that appear on it. HTML provides structure and meaning to our text, CSS allows us to precisely style it, and JavaScript contains a number of features for manipulating strings, creating custom welcome messages and prompts, showing the right text labels when needed, sorting terms into the desired order, and much more.

### Declaring strings

Strings declared using backticks are a special kind of string called a template literal. In most ways, template literals are like normal strings, but they have some special properties:

- you can embed JavaScript in them (with `${}` )

- you can declare template literals over multiple lines

## Embedding JavaScript

### Concatenation in context

Here is an example of concatenation:

HTML

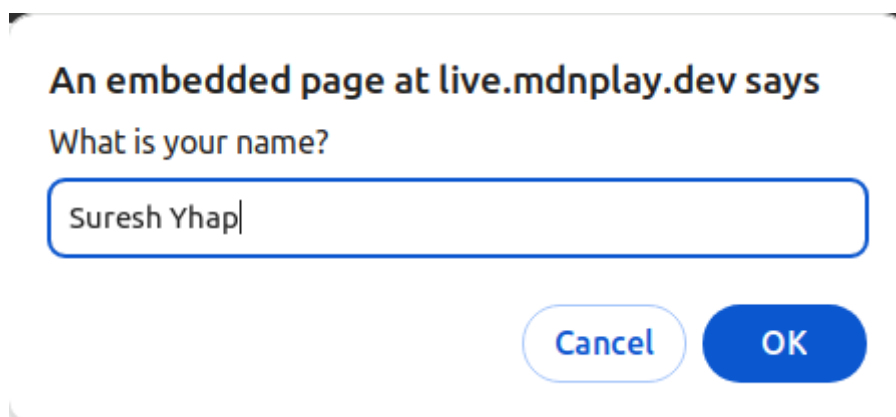
```
<button>Press me</button>
<div id="greeting"></div>
```

JS

```
const button = document.querySelector("button");

function greet() {
    const name = prompt("What is your name?");
    const greeting = document.querySelector("#greeting");
    greeting.textContent = `Hello ${name}, nice to see you!`;
}

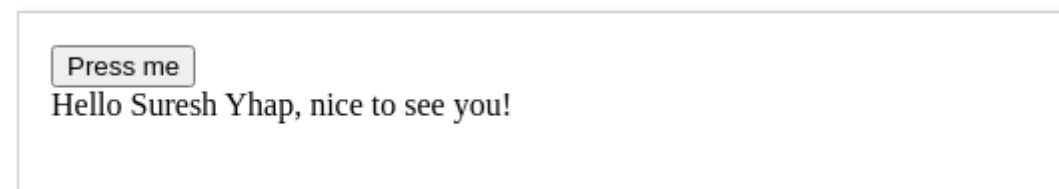
button.addEventListener("click", greet);
```



An embedded page at live.mdnplay.dev says

What is your name?

Cancel OK



Press me

Hello Suresh Yhap, nice to see you!

Here we use the `window.prompt()` function, which asks the user to answer a question via a popup dialog box. The inputted value is stored in `name` and displayed using backtick string template literal embedding.

## Concatenation using "+"

You can concatenate strings using the `+` operator or by embedding strings:

```
const greeting = "Hello";
const name = "Chris";
console.log(greeting + ", " + name); // "Hello, Chris"
```

```
const greeting = "Hello";
const name = "Chris";
console.log(`${greeting}, ${name}`); // "Hello, Chris"
```

## Including expressions in strings

Example:

```
const song = "Fight the Youth";
const score = 9;
const highestScore = 10;
const output = `I like the song ${song}, I gave it a score of ${
    (score / highestScore) * 100
}%`;
console.log(output); // "I like the song Fight the Youth. I gave it a score of 90%."
```

## Multiline strings

Template literals respect line breaks in the source code, so you can write strings that span multiple lines:

```
const newline = `One day you finally knew
what you had to do, and began`;
console.log(newline);
```



```
/*  
One day you finally knew  
what you had to do, and began,  
*/
```

To have the equivalent with a normal string you'd have to include line break characters like `\n`.

## Including quotes in strings

Since we use quotes to indicate the start and end of strings, how can we include actual quotes in strings?

One common option is to use one of the other characters to declare the string:

```
const goodQuotes1 = 'She said "I think so!"';  
const goodQuotes2 = `She said "I'm not going in there!"`;
```

Another option is to *escape* the problem quotation mark. Escaping characters means that we do something to them to make sure they are recognized as text, not part of the code. In JavaScript, we do this by putting a backslash just before the character:

```
const bigmouth = 'I\'ve got no right to take my place';  
console.log(bigmouth);
```

## Numbers vs. strings

`Number()` or unary `+` can be used to convert a value to a `number` (in particular a `string` to a `number`).

`String()` can convert a value to a `string` (in particular a `number` to a `string`).

---

[https://www.w3schools.com/jsref/jsref\\_obj\\_string.asp](https://www.w3schools.com/jsref/jsref_obj_string.asp)

## JavaScript String Methods

# JavaScript String Length

```
let text = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";  
let length = text.length; // 26
```

## Extracting String Characters

There are 4 methods for extracting string characters:

- The `at(position)` Method
- The `charAt(position)` Method
- The `charCodeAt(position)` Method
- Using property access `[]` like in arrays

## JavaScript String `charAt()`

`charAt()` method returns the character at a specified index (position) in a string:

```
let text = "HELLO WORLD";  
let char = text.charAt(0); // H
```

## JavaScript String `charCodeAt()`

The `charCodeAt()` method returns the code of the character at a specified index in a string:

The method returns a UTF-16 code (an integer between 0 and 65535).

```
let text = "HELLO WORLD";  
let char = text.charCodeAt(0); // 72
```

## JavaScript String `at()`

`at()` is relatively new. It is the same as `charAt()` except `at()` allows for negative indices. Also `at()` returns `undefined` if out of range index but `charAt()` returns empty string.

```
const name = "W3Schools";/  
let letter = name.at(2); // S  
let letter = name.at(-2); // l
```

## Property Access

Finally we can use indexing with `[]`:

```
const name = "W3Schools";  
let letter = name[2]; // S
```

Beware, property access makes strings look like arrays (but they are not in JavaScript). Also just like `at()`, if no character is found it returns undefined. Finally, it is read-only:

```
let text = "HELLO WORLD";  
text[0] = "A"; // Gives no error but it DOES NOT WORK
```

## Extracting String Parts

There are 3 methods for extracting a part of a string:

- `slice(start, end)`
- `substring(start, end)`
- `substr(start, length)`

## JavaScript String slice()

`slice()` extracts a part of a string and returns the extracted part in a new string.

The method takes 2 parameters: start position, and end position (end not included).

```
let text = "Apple, Banana, Kiwi";  
let part = text.slice(7, 13); // "Banana"
```

If you omit the second parameter, the method will slice out the rest of the string:

```
let text = "Apple, Banana, Kiwi";  
let part = text.slice(7); // "Banana, Kiwi"
```

If a parameter is negative, the position is counted from the end of the string:

```
let text = "Apple, Banana, Kiwi";  
let part = text.slice(-12); // "Banana, Kiwi"
```

This example slices out a portion of a string from position -12 to position -6 (end not included):

```
let text = "Apple, Banana, Kiwi";  
let part = text.slice(-12, -6); // "Banana"
```

## JavaScript String substring()

`substring()` is similar to `slice()`, the difference being that start and end values less than 0 are treated as 0 in `substring()`.

```
let str = "Apple, Banana, Kiwi";  
let part = str.substring(7, 13); // "Banana"
```

Similar to `slice()`, if you omit the second parameter, `substring()` will slice out the rest of the string.

## JavaScript String substr()

`substr()` is similar to `slice()`, the difference being that the second parameter specifies the **length** of the extracted part.

```
let str = "Apple, Banana, Kiwi";  
let part = str.substr(7, 6); // Banana
```

If you omit the second parameter, `substr()` will slice out the rest of the string.

If the first parameter is negative, the position counts from the end of the string.

```
let str = "Apple, Banana, Kiwi";  
let part = str.substr(-4); // "Kiwi"
```

## Converting to Upper and Lower Case

A string is converted to upper case with `toUpperCase()` or converted to lower case with `toLowerCase()`:

```
let text1 = "Hello World!";  
let text2 = text1.toUpperCase(); // "HELLO WORLD!"  
let text3 = text1.toLowerCase(); // "hello world!"
```

## JavaScript String concat()

`concat()` joins two or more strings:

```
let text1 = "Hello";  
let text2 = 'World';  
let text3 = `${5 + 6}`;  
let text4 = text1.concat(text2, text3); // "HelloWorld11"
```

The `concat()` method does the same thing as a the plus operator:

```
let text1 = "Hello" + " " + "World!"; // "Hello World!"  
let text2 = "Hello".concat(" ", "World!"); // "Hello World!"
```

All string methods return a new string. They don't modify the original string.

Strings are immutable: they cannot be changed, only replaced.

## JavaScript String trim()

`trim()` removes whitespace from both sides of a string:

```
let text1 = "      Hello World!      ";  
let text2 = text1.trim(); // "Hello World!"
```

`trimStart()` removes whitespace only from the start of string.

```
let text1 = "    Hello World!    ";  
let text2 = text1.trimStart(); // "Hello World!    "
```

`trimEnd()` removes whitespace only from the end of a string.

```
let text1 = "    Hello World!    ";  
let text2 = text1.trimEnd(); // "    Hello World!"
```

## JavaScript String `padStart()`

The `padStart()` method pads a string from the start with another string a certain amount of times until it reaches a given length:

```
let text = "5";  
let padded = text.padStart(4, "x"); // "xxx5"
```

To pad a number, convert the number to a string first:

```
let num = 5;  
let text = num.toString();  
let padded = text.padStart(4, "0"); // "0005"
```

`padEnd()` is the same but it pads onto the end.

## JavaScript String `repeat()`

The `repeat()` method returns a new string with a number of copies of a string:

```
let text = "Hello world!";  
let result = text.repeat(4); // "Hello world!Hello world!Hello world!Hello world!"
```

## Replacing String content

The `replace()` method replaces a specified value with another value in a string:

```
let text = "Please visit Microsoft!";
let newText = text.replace("Microsoft", "W3Schools"); // "Please visit
W3Schools!"
```

It replaces **only the first** match.

```
let text = "Please visit Microsoft and Microsoft!";
let newText = text.replace("Microsoft", "W3Schools"); // "Please visit
W3Schools and Microsoft!"
```

If you want to replace all matches, use a regular expression with the `/g` flag set (global match).

```
let text = "Please visit Microsoft and Microsoft!";
let newText = text.replace(/Microsoft/g, "W3Schools"); // "Please visit
W3Schools and W3Schools!"
```

Regular expressions are written without quotes.

By default, the `replace()` method is case sensitive so it will not replace if the case does not match exactly.

To replace case insensitive, use a **regular expression** with an `/i` flag (insensitive):

```
let text = "Please visit Microsoft!";
let newText = text.replace(/MICROSOFT/i, "W3Schools"); // "Please visit
W3Schools!"
```

The `replaceAll()` method is another way to replace all:

```
let text = "I love cats. Cats are very easy to love. Cats are very
popular.";
text.replaceAll("Cats", "Dogs"); // "I love cats. Dogs are very easy to
love. Dogs are very popular."
```

It is seen to be case-sensitive.

You can pass it a regular expression to be replaced; if so, the global flag (g) must be set:

```
let text = "I love cats. Cats are very easy to love. Cats are very popular.";
text = text.replaceAll(/Cats/g,"Dogs"); // "I love cats. Dogs are very easy to love. Dogs are very popular."
```

## Converting a String to an Array

A string can be converted to an array with the `split()` method:

```
text.split(","); // Split on commas
text.split(" "); // Split on spaces
text.split("|"); // Split on pipes
```

If the separator is omitted, the returned array will contain the whole string in index [0].

If the separator is `""`, the returned array will be an array of single characters:

```
text.split("");
```

---

[https://www.w3schools.com/jsref/jsref\\_obj\\_string.asp](https://www.w3schools.com/jsref/jsref_obj_string.asp)

## JavaScript String Reference



Name	Description
<code>charAt()</code>	Returns the character at a specified index (position)
<code>charCodeAt()</code>	Returns the Unicode of the character at a specified index
<code>concat()</code>	Returns two or more joined strings
<code>constructor</code>	Returns the string's constructor function
<code>endsWith()</code>	Returns if a string ends with a specified value
<code>fromCharCode()</code>	Returns Unicode values as characters
<code>includes()</code>	Returns if a string contains a specified value
<code>indexOf()</code>	Returns the index (position) of the first occurrence of a value in a string
<code>lastIndexOf()</code>	Returns the index (position) of the last occurrence of a value in a string
<code>length</code>	Returns the length of a string
<code>localeCompare()</code>	Compares two strings in the current locale
<code>match()</code>	Searches a string for a value, or a regular expression, and returns the matches
<code>prototype</code>	Allows you to add properties and methods to an object
<code>repeat()</code>	Returns a new string with a number of copies of a string
<code>replace()</code>	Searches a string for a pattern, and returns a string where the first match is replaced
<code>replaceAll()</code>	Searches a string for a pattern and returns a new string where all matches are replaced
<code>search()</code>	Searches a string for a value, or regular expression, and returns the index (position) of the match
<code>slice()</code>	Extracts a part of a string and returns a new string
<code>split()</code>	Splits a string into an array of substrings
<code>startsWith()</code>	Checks whether a string begins with specified characters

<code>startsWith()</code>	Checks whether a string begins with specified characters
<code>substr()</code>	Extracts a number of characters from a string, from a start index (position)
<code>substring()</code>	Extracts characters from a string, between two specified indices (positions)
<code>toLocaleLowerCase()</code>	Returns a string converted to lowercase letters, using the host's locale
<code>toLocaleUpperCase()</code>	Returns a string converted to uppercase letters, using the host's locale
<code>toLowerCase()</code>	Returns a string converted to lowercase letters
<code>toString()</code>	Returns a string or a string object as a string
<code>toUpperCase()</code>	Returns a string converted to uppercase letters
<code>trim()</code>	Returns a string with removed whitespaces
<code>trimEnd()</code>	Returns a string with removed whitespaces from the end
<code>trimStart()</code>	Returns a string with removed whitespaces from the start
<code>valueOf()</code>	Returns the primitive value of a string or a string object

---

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/String](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String)

## Exhaustive list of String methods

---

# Comparisons

## String comparison

JavaScript uses the "lexicographical" order:

```
alert("Z" > "A"); // true
alert("Glow" > "Glee"); // true
alert("Bee" > "Be"); // true
alert("a" > "A"); // true because the Unicode value of "a" is greater than
that of "A"
```

## Comparison of different types

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

For example:

```
alert("2" > 1); // true
alert("01" == 1); // true
```

Boolean values become their numeric equivalent ( `true` becomes `1` and `false` becomes `0` ):

```
alert(true == 1); // true
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 is `false`

For example:

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

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

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.

How can we differentiate `0` from `false` for instance?

**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.

For instance:

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

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

You should default to `===` and `!==` unless you have a reason to use `==` or `!=`.

## Comparison with null and undefined

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

Using `===`, these values are different because each of them is a different type:

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

Using `==` is true; they equal each other but not any other value other than themselves:

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

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

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

## Strange result: null vs. 0

Let's compare `null` with a zero:

```
alert(null > 0); // false
alert(null == 0); // false
alert(null >= 0); // true!
```

Mathematically, that's strange. The last result states that "`null` is greater than **or** equal to zero", so 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 `null >= 0` is true and `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 (other than themselves). That's why `null == 0` is false.

## An incomparable undefined

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

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

We get these results because:

- The first two comparisons return `false` because `undefined` gets converted to `NaN` and `NaN` is a special numeric value which returns `false` for all comparisons.

- The third comparison returns `false` because `undefined` only equals `null`, `undefined`, and no other value.

To avoid problems:

- 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 (but they also equal themselves).
- Be careful when using comparisons like `>` or `<` with variables that can occasionally be `null/undefined`. Checking for `null/undefined` separately is a good idea.

## Examples

```
alert(5 > 4); // true
```

```
alert("apple" > "pineapple"); // false because "a" < "p"
```

```
alert("2" > "12"); // true because it is another dictionary comparison, "2" > "1". Only when the types DIFFER do you convert them to numbers for comparison
```

```
alert(undefined == null); // true because they equal each other only (other than themselves)
```

```
alert(undefined === null); // false because different types on both sides (undefined is of type `undefined` and null has type `object` even though that's not correct either)
```

```
alert(null == "\n0\n"); // false because null only equals undefined (well other than null too)
```

```
alert(null === +"\n0\n"); // false because of type mismatch
```

---

## JavaScript if, else, and else if

```
let time = new Date().getHours();
if (time < 10) {
    alert("Good morning!");
}
else if (time < 20) {
    alert("Good day!");
}
else {
    alert("Good evening!");
}
```

To show a certain random link based on a fair coin flip (50-50 odds) do:

```
...
<p id="demo"></p>

<script>
    let text;
    if (Math.random() < .5) {
        text = "<a href='https://w3schools.com'>Visit W3Schools</a>";
    }
    else {
        text = "<a href='https://wwf.org'>Visit WWF</a>";
    }
    document.getElementById("demo").innerHTML = text;
</script>
```

---

## Logical operators

There are four logical operators in JavaScript: `||`, `&&`, `!`, and `??` (Nullish Coalescing).

Although they are called "logical", they can be applied to values of any type, not just boolean. Their result can also be of any type.

## || (OR)

If an operand is not boolean, it is converted to boolean for the evaluation.

For instance, `1` is treated as `true` and `0` is treated as `false`:

```
if (1 || 0) { // works just like true || false
  alert("truthy!"); // "truthy!"
}
```

Most of the time, OR is used in an `if` statement to test if *any* of the given conditions is `true`.

## OR "||" finds the first truthy value

So far OR has behaved as it classically does in other languages. Now let's bring in the "extra" features of JavaScript.

Given multiple OR'ed values:

```
result = value1 || value2 || value3;
```

The OR `||` operator does the following:

- Evaluates operands from left to right
- For each operand, converts it to boolean. If the result is `true`, stops and returns the *original* value of that operand
- If all operands have been evaluated (i.e. all were `false`), returns the last operand

A value is returned in its original form, without the conversion.

In other words, a chain of OR `||` returns the first truthy value or the last one if no truthy value is found:

```
alert(1 || 0); // 1 (1 is truthy)

alert(null || 1); // 1 (1 is the first truthy value)
```

```
alert(null || 0 || 1); // 1 (1 is the first truthy value)

alert(undefined || null || 0); // 0 (all falsy so returns the last value)
```

This leads to some interesting usages compared to a "pure, classical, boolean-only OR":

### 1. Getting the first truthy value from a list of variables or expressions

For instance, we have `firstName`, `lastName`, and `nickName` variables from a form, all optional (i.e. can be undefined or have falsy values).

Let's use OR `||` to choose the one that has the data and show it (or `Anonymous` if nothing set):

```
let firstName = "";
let lastName = "";
let nickName = "SuperCoder";

alert(firstName || lastName || nickName || "Anonymous"); // "SuperCoder"
```

If all variables were falsy, `"Anonymous"` would show up.

### 2. Short-circuit evaluation

"Short-circuit evaluation" means that `||` processes its arguments until the first truthy value is reached, and then value is returned immediately, without even touching the other arguments.

The importance of this feature becomes clear if an operand isn't just a value, but an expression with a side effect, such as a variable assignment or a function call.

Here only the second message is printed:

```
true || alert("not printed");
false || alert("printed");
```

Sometimes, people use this feature to execute commands only if the condition on the left part is falsy.

## && (AND)



In classical programming, AND returns `true` if both operands are `truthy` and `false` otherwise:

```
alert(true && true); // true
alert(false && true); // false
alert(true && false); // false
alert(false && false); // false
```

## AND "&&" finds the first falsy value

Given multiple AND'ed values:

```
result = value1 && value2 && value3;
```

The AND `&&` operator does the following:

- Evaluates operands from left to right
- For each operand, converts it to a boolean. If the result is `false`, stops and returns the *original* value of that operand
- If all operands have been evaluated (i.e. all were `truthy`), returns the last operand

In other words, AND returns the first falsy value or the last value if none were found.

The rules above are similar to OR; the difference is that AND returns the first *falsy* value while OR returns the first *truthy* one.

Examples:

```
alert(1 && 0); // 0
alert(1 && 5); // 5
alert(null && 5); // null
alert(0 && "no matter what"); // 0
alert(1 && 2 && null && 3); // null
alert(1 && 2 && 3); // 3
```

Note: The precedence of AND `&&` is higher than OR `||`. So the code `a && b || c && d` is essentially the same as if the `&&` expressions were in parentheses: `(a && b) || (c && d)`.

Since `&&` can be used to execute a second part if the first is `true` (since it looks for the first falsy value), sometimes people use AND `&&` as a shorter way of writing `if`. For instance:

```
let x = 1;

(x > 0) && alert("Greater than zero!");
```

The action in the right part of `&&` will execute only if the evaluation reaches it; that is, only if `(x > 0)` is true. This is basically the functionality of an `if` statement:

```
let x = 1;

if (x > 0) alert("Greater than zero!");
```

The second of these alternatives is more readable because `if` was designed for this purpose while `&&` was designed primarily to do the AND operation.

## ! (NOT)

The boolean NOT operator, `!`, converts the operand to boolean type: `true/false` and then returns the inverse value.

For instance:

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

A double NOT `!!` is sometimes used for converting a value to boolean type:

```
alert(!!"non-empty string"); // true
alert(!!null); // false
```

There's a little more verbose way to do the same thing- a built-in `Boolean` function:

```
alert(Boolean("non-empty string")); // true
alert(Boolean(null)); // false
```

The precedence of NOT `!` is the highest of all logical operators, so it always executes before `&&` or `||`.

Examples:

```
alert(alert(1) || 2 || alert(3)); // Alerts 1 then alerts 2
```

The call to `alert` does not return a value; in other words, it returns `undefined`. `undefined` is `false` so the OR goes onto the second operand searching for a truthy value after it executes the `alert(1)`. `2` is truthy so execution is halted and the outer `alert` alerts 2. There is no 3 because execution does not reach `alert(3)`.

---

```
alert(alert(1) && alert(2)); // Alerts 1 then undefined
```

Remember that the call to `alert` returns `undefined`. So the `&&` evaluates `alert(1)` which alerts 1 then returns `undefined`. This is the first falsy value seen so it is returned to the outer `alert` call which alerts `undefined`.

---

```
alert(null || 2 && 3 || 4); // Alerts 3
```

`&&` executes before `||`. `2 && 3` evaluates to `3` (the last value since none were falsy) so the expression is essentially:

```
null || 3 || 4
```

 which searches for the first truthy value, i.e. `3`.

The following is an example conditional flow for a login attempt:

```
let userName = prompt("Who's there?", "");
if (userName === "Admin") {
    let password = prompt("Password?", "");
    if (password === "TheMaster") {
        alert("Welcome!");
    }
    else if (password === "" || password === null) {
        alert("Canceled");
    }
    else {
        alert("Wrong password");
    }
}
else if (userName == "" || userName == null) {
```

```
    alert("Canceled");  
  }  
  else {  
    alert("I don't know you");  
  }  
}
```

---

## Making decisions in our code - conditionals

In a weather app, if it is being looked at in the morning, show a sunrise graphic; show stars and a moon if it is nighttime.

This is an example of a simple weather forecast application:

### HTML

```
<label for="weather">Select the weather type today: </label>  
<select id="weather">  
  <option value="">--Make a choice--</option>  
  <option value="sunny">Sunny</option>  
  <option value="rainy">Rainy</option>  
  <option value="snowing">Snowing</option>  
  <option value="overcast">Overcast</option>  
</select>  
  
<p></p>
```

### JS

```
const select = document.querySelector("select");  
const para = document.querySelector("p");  
  
select.addEventListener("change", setWeather);  
  
function setWeather() {  
  const choice = select.value;
```

```

if (choice === "sunny") {
    para.textContent =
        "It is nice and sunny outside today. Wear shorts! Go to the beach, or
the park, and get an ice cream.";
} else if (choice === "rainy") {
    para.textContent =
        "Rain is falling outside; take a rain coat and an umbrella, and don't
stay out for too long.";
} else if (choice === "snowing") {
    para.textContent =
        "The snow is coming down – it is freezing! Best to stay in with a cup
of hot chocolate, or go build a snowman.";
} else if (choice === "overcast") {
    para.textContent =
        "It isn't raining, but the sky is grey and gloomy; it could turn any
minute, so take a rain coat just in case.";
} else {
    para.textContent = "";
}
}

```

Select the weather type today:

The snow is coming down — it is freezing! Best to stay in with a cup of hot chocolate, or go build a snowman.

1. We've got an HTML `<select>` element allowing us to make different weather choices, and a simple paragraph.
2. In the JavaScript, we store a reference to both the `<select>` and `<p>` elements, and adding an event listener to the `<select>` element so that when its value is changed, the `setWeather()` function is run.
3. When this function is run, we first set a variable called `choice` to the current value selected in the `<select>` element. We then print a message based on the current selected value.
4. The last choice is a catch all "last resort" option. It empties the text out of the paragraph if nothing is selected, for example, if a user decides to re-select the "--Make a choice--" placeholder option shown at the beginning.

## A note on comparison operators

Any value that is not `false`, `undefined`, `null`, `0`, `NaN`, or an empty string (`""`) returns `true` when tested as a conditional statement so you can use a variable name on its own to test whether it is `true`, or even that it exists (that is, it is not undefined). For instance,

```
let cheese = "Cheddar";

if (cheese) {
  console.log("Yay! Cheese available for making cheese on toast.");
} else {
  console.log("No cheese on toast for you today.");
}
```

## Nested if...else

An example of nested if else:

```
if (choice === "sunny") {
  if (temperature < 86) {
    para.textContent = `It is ${temperature} degrees outside – nice and sunny. Let's go out to the beach, or the park, and get an ice cream.`;
  } else if (temperature >= 86) {
    para.textContent = `It is ${temperature} degrees outside – REALLY HOT! If you want to go outside, make sure to put some sunscreen on.`;
  }
}
```

## switch statements

If you just want to set a variable to a certain choice of value or print out a particular statement depending on a condition, a `switch` statement might be simpler to use.

The choice in the `switch` statement can even be things like strings or some general expression.

## Ternary operator

The ternary conditional operator works like it does in C++:

```
const greeting = isBirthday
  ? "Happy birthday Mrs. Smith – we hope you have a great day!"
  : "Good morning Mrs. Smith.";
```

The ternary operator is not just for setting variable values; you can also run functions:

Here is an example of a theme picker that can be black or white:

## HTML

```
<label for="theme">Select theme: </label>
<select id="theme">
  <option value="white">White</option>
  <option value="black">Black</option>
</select>

<h1>This is my website</h1>
```

## JS

```
const select = document.querySelector("select");
const html = document.querySelector("html");
document.body.style.padding = "10px";

function update(bgColor, textColor) {
  html.style.backgroundColor = bgColor;
  html.style.color = textColor;
}

select.addEventListener("change", () =>
  select.value === "black"
    ? update("black", "white")
    : update("white", "black"),
  );
```



Here we've got a `<select>` element to choose a theme (black or white), plus a simple `<h1>` to display a website title. We also have a function called `update()`, which takes two colors as parameters. The website's background color is set to the first provided color, and its text color is set to the second provided color.

Finally, we've also got an `onchange` event listener that serves to run a function containing a ternary operator. It starts with a test condition: `select.value === "black"`. If this returns `true`, we run the `update()` function with parameters of black and white, meaning that we end up with a background color of black and a text color of white. If it returns `false`, we run the `update()` function with parameters of white and black, meaning that the site colors are inverted.

## A simple calendar example

In this example, we have

- a `<select>` element to allow the user to choose between different months.
- an `onchange` event handler to detect when the value selected in the `<select>` menu is changed.
- a function called `createCalendar()` that draws the calendar and displays the correct month in an `h1` element

## HTML

```
<!DOCTYPE html>
<html lang="en">
```



```

<head>
  <meta charset="utf-8">
  <title>Simple Calendar</title>
  <link href="style.css" rel="stylesheet">
</head>
<body>
  <label for="month-selector">Select month:</label>
  <select id="month-selector">
    <option value="0">Month</option>
    <option value="1">January</option>
    <option value="2">February</option>
    <option value="3">March</option>
    <option value="4">April</option>
    <option value="5">May</option>
    <option value="6">June</option>
    <option value="7">July</option>
    <option value="8">August</option>
    <option value="9">September</option>
    <option value="10">October</option>
    <option value="11">November</option>
    <option value="12">December</option>
  </select>
  <h1></h1>
  <ul class="days"></ul>
  <script src="script.js"></script>
</body>
</html>

```

## CSS

```

html {
  box-sizing: border-box;
}

*,
*::before,
*::after {
  box-sizing: inherit;
}

```

```

ul {
  list-style: none;
  display: flex;
  color: white;
  flex-wrap: wrap;
  width: 500px;
  padding: 0px;
  gap: 2px;
}

li {
  background-color: #4A2DB6;
  flex-basis: 20%;
  padding: 2px;
}

```

## JS

```

const select = document.querySelector("select");
const h1 = document.querySelector("h1");
const list = document.querySelector("ul");

function createCalendar(days, monthNameChoice) {
  list.innerHTML = "";
  h1.textContent = monthNameChoice;
  for (let i = 1; i <= days; ++i) {
    const listItem = document.createElement("li");
    listItem.textContent = i;
    list.appendChild(listItem);
  }
}

select.addEventListener("change", () => {
  const choice = +select.value;
  const MONTHS = [
    "", "January", "February", "March", "April", "May", "June",
    "July", "August", "September", "October", "November", "December"
  ];

```

```
let month = MONTHS[choice];
let days;
switch (choice) {
  case 4:
  case 6:
  case 9:
  case 11:
    days = 30;
    break;
  case 2:
    days = 28;
    break;
  case 0:
    days = 0;
    break;
  default:
    days = 31;
}
createCalendar(days, month);
});
```

Select month:

## February

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16
17	18	19	20
21	22	23	24
25	26	27	28

Select month:

## June

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16
17	18	19	20
21	22	23	24
25	26	27	28
29	30		

---

## More color choices

Instead of toggling between black and white colors, we can add more:


```
const select = document.querySelector('select');
const html = document.querySelector('.output');

select.addEventListener('change', () => {
  const choice = select.value;

  switch(choice) {
    case 'black':
      update('black', 'white');
      break;
    case 'white':
      update('white', 'black');
      break;
    case 'purple':
      update('purple', 'white');
      break;
    case 'yellow':
      update('yellow', 'purple');
      break;
    case 'psychedelic':
```


```
        update('lime','purple');
        break;
    }
});

function update(bgColor, textColor) {
    html.style.backgroundColor = bgColor;
    html.style.color = textColor;
}
```



Select theme: Yellow ▼

**This is my website**



Select theme: Purple ▼

**This is my website**

---

`includes()` is a string method that tests if a different string is a substring of the string.

---

The following example prints a message depending on what score range your test score lies and whether the scoring machine is on or not:

## HTML

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <p class="display-score"></p>
    <div class="scoring"></div>
    <script src="script.js"></script>
  </body>
</html>
```

## JS

```
function getRandomInt(start, end) {
  return Math.random() * (end - start) + start;
}

let machineActive = true;
let score = getRandomInt(0, 100);
let response;

if (machineActive) {
  if (score < 0 || score > 100) {
    response = "This is not possible, an error has occurred.";
  }
  else if (score < 20) {
    response = "That was a terrible score – total fail!";
  }
  else if (score < 40) {
```

```
        response = "You know some things, but it\'s a pretty bad score.
Needs improvement.";
    }
    else if (score < 70) {
        response = "You did a passable job, not bad!";
    }
    else if (score < 90) {
        response = "That\'s a great score, you really know your stuff."
    }
    else {
        response = "What an amazing score! Did you cheat? Are you for
real?";
    }
}
else {
    response = "Machine is inactive. Please turn the machine on!";
}

const scoring_div = document.querySelector("div.scoring");
const print_score = document.querySelector("p.display-score");
print_score.textContent = `You got a ${score}.`;
print_score.style.fontWeight = "bold";
let statement = document.createElement("p");
statement.textContent = response;
scoring_div.appendChild(statement);
```

**You got a 91.46644182684692.**

What an amazing score! Did you cheat? Are you for real?

**You got a 57.804815802442235.**

You did a passable job, not bad!

---

"0" is considered true!

In a switch case statement block, both the switch and the cases can be any arbitrary expressions. The equality check is strict (like `===`)