<https://javascript.info/currying-partials>

<https://javascript.info/closure#:~:text=The%20Lexical%20Environment%20object%20consists,associated%20with%20the%20outer%20code>.

# Currying

## [Summary](https://javascript.info/currying-partials" \l "summary)

Currying is a transform that makes f(a,b,c) callable as f(a)(b)(c). JavaScript implementations usually both keep the function callable normally and return the partial if the arguments count is not enough.

Currying allows us to easily get partials. As we’ve seen in the logging example, after currying the three argument universal function log(date, importance, message) gives us partials when called with one argument (like log(date)) or two arguments (like log(date, importance))

[Currying](https://en.wikipedia.org/wiki/Currying) is an advanced technique of working with functions.

Currying is a transformation of functions that translates a function from callable as f(a, b, c) into callable as f(a)(b)(c).

Currying doesn’t call a function. It just transforms it.

Let’s see an example first, to better understand what we’re talking about, and then practical applications.

We’ll create a helper function curry(f) that performs currying for a two-argument f. In other words, curry(f) for two-argument f(a, b) translates it into a function that runs as f(a)(b):

function curry(f) { // curry(f) does the currying transform

return function(a) {

return function(b) {

return f(a, b);

};

};

}

// usage

function sum(a, b) {

return a + b;

}

let curriedSum = curry(sum);

alert( curriedSum(1)(2) ); // 3

As you can see, the implementation is straightforward: it’s just two wrappers.

* The result of curry(func) is a wrapper function(a).
* When it is called like curriedSum(1), the argument is saved in the Lexical Environment, and a new wrapper is returned function(b).
* Then this wrapper is called with 2 as an argument, and it passes the call to the original sum.

More advanced implementations of currying, such as [\_.curry](https://lodash.com/docs#curry) from lodash library, return a wrapper that allows a function to be called both normally and partially:

function sum(a, b) {

return a + b;

}

let curriedSum = \_.curry(sum); // using \_.curry from lodash library

alert( curriedSum(1, 2) ); // 3, still callable normally

alert( curriedSum(1)(2) ); // 3, called partially

## [Currying? What for?](https://javascript.info/currying-partials" \l "currying-what-for)

To understand the benefits we need a worthy real-life example.

For instance, we have the logging function log(date, importance, message) that formats and outputs the information. In real projects such functions have many useful features like sending logs over the network, here we’ll just use alert:

function log(date, importance, message) {

alert(`[${date.getHours()}:${date.getMinutes()}] [${importance}] ${message}`);

}

## Currying using closures

The closure also makes the currying possible in JavaScript. A closure can be defined as a JavaScript feature in which the inner function has access to the outer function variable. In JavaScript, every time a closure is created with the creation of a function.

function mul(val1){

    return function(val2){

        document.write("**<p>**" + val1 \* val2 + "**</p>**");

    }

}

let a = mul(2);

a(5);

a(6);

1. function multiply(a) {
2. return function(b) {
3. return function(c) {
4. return a \* b \* c;
5. }
6. }
7. }
8. document.write(multiply(2)(5)(8));

In the above code, we have used currying as the parameters passed one by one until the last function calls the last parameter. So, before and after currying on the same function with the same parameters, the output will remain the same.

So, that's all about the article. Hope you understand the concept of currying and also the process of achieving it.