**Currying in JavaScript**

### What is Currying?

Currying is a function that *takes one argument at a time and returns a new function expecting the next argument*. It is a conversion of functions from callable as f(a,b,c)into callable as f(a)(b)(c).

### Basically Currying doesn’t call a function. It just transforms a function. They are constructed by chaining closures by immediately returning their inner functions simultaneously. ****Convert f(a, b) into f(a)(b).****

/\*f(a,b) implementation \*/

function f(a,b) {

return "Works"

}  
  
/\*f(a)(b) implementation \*/

function f(a) {

return (b) => {

"Works"

}

}

console.log(f(1)(2)) // works

console.log(f(1)); /\* (b) => {return "Works" } \*/

### ****Why should currying be used?****

✅ It makes a function pure which makes it expose to less errors and side effects.

✅ It helps in avoiding the same variable again and again.

✅ It is a checking method that checks if you have all the things before you proceed.

✅ It divides one function into multiple functions so that one handles one set of responsibility.

### ****How does currying work?****

Currying is a function that takes multiple arguments as input. It transform the function into a number of functions where every function will accept one argument.

/\*Simple function\*/

const add = (a, b, c)=>{

return a+ b + c

}

console.log(add(1,2 ,3)); // 6

/\* Curried Function \*/

const addCurry = (a) => { // takes one argument

return (b)=>{ //takes second argument

return (c)=>{ //takes third argument

return a+b+c;

}

}

}

console.log(addCurry(1)(2)(3)); //6

function sum(a) {

return (b) => {

return (c) => {

return a + b + c

}

}

}

/\* you can call it in two ways\*/

1️⃣ console.log(sum(1)(2)(3)); //6

2️⃣ const sum1 = sum(1);

const sum2 = sum1(2);

const result = sum2(3);

console.log(result); // 6

### Evaluate(”sum”)(2)(4) ⇒ 2+4 = 6 on basis of input given to first param

function evaluate(operation){

return (a) => {

return (b) => {

if(operation === 'sum') return a + b;

else if(operation === 'multiply') return a \* b;

else if(operation === 'subtract') return a - b;

else if(operation === 'divide') return a / b;

else return 'invalid operation'

}

}

}

console.log(evaluate('sum')(2)(3));

console.log(evaluate('multiply')(2)(3));

console.log(evaluate('subtract')(2)(3));

console.log(evaluate('divide')(2)(3));

### Write a currying function that takes infinite arguments.

/\*Straightforward and time-taking solution\*/

const sum = function(a) {

return function(b) {

return function(c) {

return function(d) {

...

...

...

return a + b + c + d + ... n;

}

}

}

}

//recursive solution

const infiniteSum = (a) => {

return (b) => {

if (b) {

return infiniteSum(a + b);

} else {

return a;

}

}

}

// Usage:

console.log(infiniteSum(1)(2)(3)(4)()); // Outputs: 10

console.log(infiniteSum(5)(10)(15)(20)(25)()); // Outputs: 75

**How can we manipulate DOM using currying?**

<div>

<h1 id="header">Hello Piyush</h1>

</div>

const updateElemText = id => content => document.querySelector(`#${id}`).textContent= content;

const updateHeaderText = updateElemText('header');

updateHeaderText('Hello RoadsideCoder!');

### Write a function curry() that converts f(a,b,c) into a curried function f(a)(b)(c) .

function curry(func) {

// args takes arguments in the form of array eg - [a, b, c]

return function curriedFunc(...args) {

//check if current args passed equals the number of args function expects

if(args.length >= func.length) {

// if yes, we spread args elements to pass into func (spread). This is our base case.

return func(...args)

} else {

/\* if not, we return a function that collects the next arguments passed in next and

we recursively call curriedFunc, accumulating and spreading the values of args first and then

the values of next. next will take into consideration a variable amount of next arguments

e.g (1, 2) (1) (1,2,3) \*/

return function(...next) {

return curriedFunc(...args,...next);

}

}

}

}

const join = (a, b, c) => {

return `${a}\_${b}\_${c}`

}

const curriedJoin = curry(join)

// curriedJoin(1, 2, 3) // '1\_2\_3'

// curriedJoin(1)(2, 3) // '1\_2\_3'

curriedJoin(1, 2)(3) // '1\_2\_3'

### Write a function curry() that converts f(a,b,c) into a curried function f(a)(b)(c) with placeholder ( \_ ) support.

function curry(func) {

return function curried(...args) {

// Check if enough arguments are provided or if one of them is a placeholder

if (args.length >= func.length && !args.includes('\_')) {

return func(...args);

} else {

return function (...nextArgs) {

const combinedArgs = args.map(arg => arg === '\_' && nextArgs.length ? nextArgs.shift() : arg)

.concat(nextArgs);

return curried(...combinedArgs);

};

}

};

}

// Example usage:

function add(a, b, c) {

return a + b + c;

}

const curriedAdd = curry(add);

console.log(curriedAdd(1)(2)(3)); // Output: 6

console.log(curriedAdd(1, 2)(3)); // Output: 6

console.log(curriedAdd(1)(2, 3)); // Output: 6

console.log(curriedAdd(1, 2, 3)); // Output: 6

console.log(curriedAdd(1, '\_', 3)(2)); // Output: 6