**What is Currying?**

Currying is a process in functional programming in which we can transform a function with multiple arguments into a sequence of nesting functions. It returns a new function that expects the next argument inline.

A curried function is a function that takes multiple arguments one at a time.

Given a function with 3 parameters, the curried version will take one argument and return a function that takes the next argument, which returns a function that takes the third argument. The last function returns the result of applying the function to all of its arguments.

It keeps returning a new function (that expects the current argument, like we said earlier) until all the arguments are exhausted. The arguments are kept "alive"(via closure) and all are used in execution when the final function in the currying chain is returned and executed.

*Note:* The term arity, refers to the number of arguments a function takes. For example,

function fn(a, b) {  
 //...  
}function \_fn(a, b, c) {  
 //...  
}

So, currying transforms a function with multiple arguments into a sequence/series of functions each taking a **single argument**.

Let’s look at a simple example:

function multiply(a, b, c) {  
 return a \* b \* c;  
}

This function takes three numbers, multiplies the numbers and returns the result.

multiply(1,2,3); // 6

See, how we called the multiply function with the arguments in full. Let’s create a curried version of the function and see how we would call the same function (and get the same result) in a series of calls:

function multiply(a) {  
 return (b) => {  
 return (c) => {  
 return a \* b \* c  
 }  
 }  
}

log(multiply(1)(2)(3)) // 6

We have turned the multiply(1,2,3) function call to multiply(1)(2)(3) multiple function calls.

One single function has been turned to a series of functions. To get the result of multiplication of the three numbers 1, 2 and 3, the numbers are passed one after the other, each number prefilling the next function inline for invocation.

We could separate this multiply(1)(2)(3) to understand it better:

const mul1 = multiply(1);  
const mul2 = mul1(2);  
const result = mul2(3);  
log(result); // 6

Let’s take it one after the other. We passed 1 to the multiply function:

let mul1 = multiply(1);

It returns the function:

return (b) => {  
 return (c) => {  
 return a \* b \* c  
 }  
 }

Now, mul1 holds the above function definition which takes an argument b.

We called the mul1 function, passing in 2:

let mul2 = mul1(2);

The mul1 will return the third function:

return (c) => {  
 return a \* b \* c  
 }

The returned function is now stored in mul2 variable.

In essence, mul2 will be:

mul2 = (c) => {  
 return a \* b \* c  
 }

When mul2 is called with 3 as the parameter,

const result = mul2(3);

it does the calculation with the previously passed in parameters: a = 1, b = 2 and returns 6.

log(result); // 6

Being a nested function, mul2 has access to the variable scope of the outer functions, multiply and mul1.

This is how mul2 could perform the multiplication operation with variables defined in the already exit-ed functions. Though the functions have long since returned and garbage collected from memory, yet its variables are somehow still kept "alive".

You see that the three numbers were applied one at a time to the function, and at each time, a new function is returned until all the numbers are exhausted.

Another example to show reusability with curries function

function curriedMultiplyTwo(a) {

return function (b) {

return a \* b;

};

};

const double = curriedMultiplyTwo(2);

const triple = curriedMultiplyTwo(3);

console.log(double(3); // prints 6

console.log(triple(4); // prints 12

**What is Function composition ?**

Function composition is a mechanism of combining multiple simple functions to build a more complicated one. The result of each function is passed to the next one.

Example:

const **add** = (a, **b)** => a + **b;**

const **mult** = (a, **b)** => a \* **b;**

**add(2, mult(3,** 5))