Currying in JavaScript

Currying is a functional programming technique.

This article will discuss *currying,* a functional programming technique we can use to write code that is modular, easy to test, and highly reusable. Functional programming is a declarative paradigm that emphasizes immutability and *pure functions* — meaning the function is side-effect free and for any given input it will always return the same output.

This helps make code more readable and easier to reason about in general, and currying is just one part of it. Let's get started with currying!

How currying works

In order to understand the point of currying and how it works, we can first look at a non-curried function.

Below, we've created a non-curried function `add()`. `add()` will still execute if called with a missing argument, which would result in the missing variable being evaluated as `undefined`. For the `add()` function below, if `add()` was called with an argument for `a` but not for `b`, the function call would evaluate as `1 + undefined` and return `NaN`. That's not great and could be tricky to debug in practice.

```
function add(a,b) {
    return a + b;
}
```

Why does this happen? If `add()` is missing an argument when it's called, it will evaluate the function call as `add(a, undefined)`. For example, if we use 1 as an argument for the variable `a` and no argument for `b`, calling `add(1)` would run as `add(1, undefined)`.

Function call:	→	Evaluated as:
`add(1)`	→	`add(1, undefined)`

Let's curry the `add()` function to see how we can handle the function call better if only one argument is available. What we can do is transform the `add()` function, which normally expects multiple arguments, into a series of nested functions that will each take a single argument.

This makes the function calls more modular. With curried functions, calling the outer function returns the next function in the chain and so on until the innermost function is called, which then returns a value.

In code that would look like this:

```
// traditional function
function add(a,b) {
    return a + b;
}
```

```
function curried_add(a) {
    // The outer function returns a nested single-argument function
    return function nested_func(b) {
        return a + b;
    }
}
```

What we did in the code is declare `curried_add()` as a function that takes a single argument and returns another function named `nested_func()`. (The returned function can also be anonymous, but has a name in this example for clarity.)

- Calling the outer function `curried_add()` returns `nested_func()`.
- Calling `nested_func` uses the arguments supplied from calling `curried_add` and `nested_func` to return the evaluated result of `a + b`.

When you call `curried_add(a)`, the returned output is the function `nested_func(b)`. Then you can call `nested_func(b)` and the output will be the evaluated result of `a + b`.

Function call:	→	Returns:
`curried_add(1)`	→	`nested_func(b)`
`nested_func(b)`	→	`a + b`

You can call that in your code as `curried_add(a)(b)`, which invokes `curried_add()`, then invokes `nested_func()` immediately following the first function call.

Let's break that down further. In the code block below, we are assigning the result of calling `curried_add()` to a variable named `add_one`.

```
let add_one = curried_add(1);
// returns nested_func()
```

Now, when we call `add_one()`, it will return the value of `a + b`, because it is the function `nested_func()`.

```
add_one(10);
// returns 11
```

The argument from calling `curried_add()` is available to the nested functions due to *closure*. A closure means that the nested function retains the scope of parent functions based on where the function is defined, even if the nested function is executed outside of that lexical scope.

You may recall that a function can access variables both in its inner and outer scope. That behavior is an example of *lexical* scoping rules, which means the scoping is based on the structure of the code.

Taking that one step further, when a function is invoked, lexical scoping is retained. So nested functions will continue to have access to any variables that are declared in the outer scope of parent functions. This is true even if that parent function is done executing.

Overall, that means that when you run the line `let add_one = curried_add(1);`, `add_one()` will retain the scope from `curried_add()` and therefore have access to the variable created for the argument `a` as 1, which you can see explained in the code snippet line by line:

```
function curried_add(a) {
    // has access to the argument for a
    return function nested_add(b) {
        // has access to the arguments for a and b
        return a + b;
    }
}

// creates a local variable a and assigns it the value 1
let add_one = curried_add(1);

// add_one() still has access to the argument from curried_add()
add_one(10);
```

Currying with Arrow Functions

The same `curried_add()` function from earlier can be rewritten much more concisely using ES6 arrow function syntax:

```
let curried_add = a => b => a + b;
```

Let's break that down:

- `let curried_add` is a variable assignment to the outer arrow function, `a => ...`.
- Calling `curried_add` takes an argument `a` and returns `b => a + b`.
- Invoking the second arrow function returns the sum, or `a + b`.

Currying in Context

Now that we understand the basics of how currying works, let's look at a use case that's more interesting than addition.

For this example, a social sports league company needs a way to filter through an array of players from various cities, sports leagues, and age brackets to generate some graphs about their player data in a dashboard. Each time they generate a list of filtered values, they have to repeat filters that iterate over the entire `players` array, so they want to modify their code.

```
const players = [
    { age: 5, sport: "soccer", city: "Chicago", dateJoined: new
Date('2021-01-20') },
    { age: 6, sport: "baseball", city: "Boulder", dateJoined: new
Date('2019-12-30') },
    { age: 10, sport: "soccer", city: "Chicago", dateJoined: new
Date('2020-11-12') },
    { age: 11, sport: "handball", city: "San Francisco", dateJoined: new
Date('2020-08-21') },
    { age: 6, sport: "soccer", city: "Chicago", dateJoined: new
Date('2021-07-06') },
    { age: 8, sport: "softball", city: "Boulder", dateJoined: new
Date('2019-02-27') },
    { age: 7, sport: "tennis", city: "San Francisco", dateJoined: new
Date('2019-05-31') },
    { age: 4, sport: "handball", city: "San Francisco", dateJoined: new
Date('2018-03-10') }
]
```

Let's say they're interested in seeing all players from the city of San Francisco sorted by age as well as seeing all players in San Francisco who play handball. We can filter over the array of objects to grab the relevant objects, then sort the results or do additional filtering.

Below, `sortPlayersByValueFromCity()` takes three arguments:

- an array of players
- the city to filter by
- a sort key to sort by

The function filters the array of objects by the city key, then sorts by the sort criteria.

```
const sortPlayersByValueFromCity = (playersArr, city, sortKey) => {
    return playersArr.filter(player => {
        return player.city === city;
    }).sort((a,b) => {
        return a[sortKey] - b[sortKey]
    });
}

console.log(sortPlayersByValueFromCity(players, "San Francisco",
"age"));
```

Meanwhile, in the following code block, `filterPlayersByValueFromCity()` takes four arguments:

- an array to filter
- a city to filter the array by
- an additional filter key,
- the matching filter value to search for

The function filters by the city then filters the results by the key-value pair.

```
const filterPlayersByValueFromCity = (playersArr, city, filterKey,
filterValue) => {
 return playersArr.filter(player => {
   return player.city === city;
 }).filter(playersFromCity => playersFromCity[filterKey] ===
filterValue)
}

console.log(filterPlayersByValueFromCity(players, "San Francisco",
"sport", "handball"));
```

There are a couple things you should notice about those functions:

- Both functions are filtering by city and performing the same filter each time the function is executed for San Francisco.
- Both functions take several arguments.

Let's make the code more modular with currying to eliminate potential problems in the code and to avoid repeating the repetitive filter operations.

Identify errors faster

Say that we call `filterPlayersByValueFromCity()` as `filterPlayersByValueFromCity(players, "San Fran", "sport")`. The returned result is going to be an empty array. There are several things wrong with that function call, but it will still run and it will be unclear why an empty array was returned. The errors include:

- San Francisco is input as San Fran.
- The fourth argument is missing.

As we covered earlier in the article, with currying, we can guarantee the final filter will not execute until all arguments have been entered and each nested function has been called. Since you're handling each argument one off in single argument functions, you can handle validating each argument as you go through the chain and hone in on any argument errors before code proceeds or has a chance to return an empty array.

Make Code Easier to Read and Reuse

The code for `sortPlayersByValueFromCity()` and `filterPlayersByValueFromCity()` is hard to read. With currying, we can write functions that handle one task, and are therefore not only easier to read and understand, but more reusable. For example, we can create a curried function that filters an array of objects by a provided key and value.

```
const setFilter = array => key => value => array.filter(x => x[key] ===
value);
const filterPlayers = setFilter(players);
const filterPlayersByCity = filterPlayers('city');
const filteredPlayersBySanFrancisco = filterPlayersByCity('San
Francisco');
const filterPlayersBySport = filterPlayers('sport');
const filteredPlayersBySoccer = filterPlayersBySport('soccer');

console.log(filteredPlayersBySanFrancisco); // Returns an array of
players from San Francisco
console.log(filteredPlayersBySoccer); // Returns an array of players
that play soccer
```

Make Code More Modular

Now we can reuse the filtered San Francisco object for additional specialized functions. Let's use it to sort the players by the date they joined the sports league.

```
const sortArrayByValue = sortArray => sortKey => {
    return sortArray.sort(function(a, b){
        if(a[sortKey] < b[sortKey]) { return -1; }
        if(a[sortKey] > b[sortKey]) { return 1; }
        return 0;
    });
}

const sortSanFrancisco
= sortArrayByValue(filteredPlayersBySanFrancisco);
const sortSFByDateJoined = sortSanFrancisco("dateJoined");
console.log(sortSFByDateJoined);
```

Summary

In this article, we took a look at how currying works under the hood thanks to closures in JavaScript, and how you can use different syntax techniques to curry your functions. Overall, thanks to the modularity of curried functions, you can now use currying in your code to make your functions have a single purpose and therefore be easier to test, debug, maintain, and read.