Tip 35: Combine Currying & Array Methods for Partial Application

In this tip, you'll learn to lock in variables with partial application of functions.

We'll cover the following

In the previous tip, you saw how you can give parameters a single responsibility with higher-order functions and partial application. It solved the problem of having unrelated parameters, but it didn't solve the problem of using the same parameters over and over. You still passed in the same parameters multiple times.

Avoiding parameter repetition

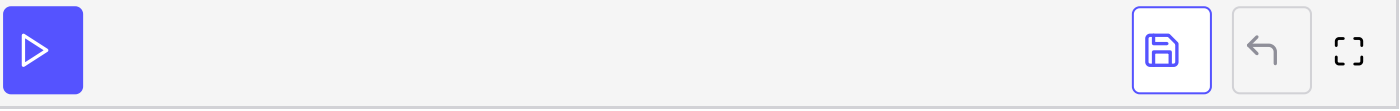
With higher-order functions, you can avoid repetition by creating a new function with values you lock-in once and use later. When you return a higher-order function, you don't have to invoke it right away. After you invoke it once, you have another pre-made function that you can use over and over. It's like you wrote it with the argument hard-coded.

To reuse the `building` and `manager` from the previous tip, you can assign the return value from the first function call to a variable. You now have a prebuilt function with some information locked in place.

Invoking it once and reusing the captured parameters is no different from declaring a function knowing the inside variables ahead of time. These are equivalent.

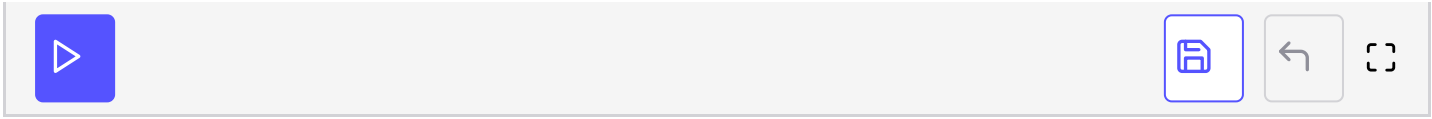
equivalent.

```
const building = {
    hours: '8 a.m. - 8 p.m.',
    address: 'Jayhawk Blvd',
};

const manager = {
    name: 'Augusto',
    phone: '555-555-5555',
};

const program = {
    name: 'Presenting Research',
    room: '415',
    hours: '3 - 6',
};

const exhibit = {
    name: 'Emerging Scholarship',
    contact: 'Dyan',
};


const setStrongHallProgram = program => {
    const defaults = {
        hours: '8 a.m. - 8 p.m.',
        address: 'Jayhawk Blvd',
        name: 'Augusto',
        phone: '555-555-5555'
    }
    return { ...defaults, ...program }
}

const setStrongHallProgramInfo = mergeProgramInformation(building, manager);
const programInfo = setStrongHallProgramInfo(program);
console.log(programInfo);
const exhibitInfo = setStrongHallProgramInfo(exhibit);
console.log(exhibit);
```



```
const program = {
    name: 'Presenting Research',
    room: '415',
    hours: '3 - 6',
};

const exhibit = {
    name: 'Emerging Scholarship',
    contact: 'Dyan',
};

const setStrongHallProgram = program => {
    const defaults = {
        hours: '8 a.m. - 8 p.m.',
        address: 'Jayhawk Blvd',
```

```
        name: 'Augusto',
        phone: '555-555-5555'
    }

    return { ...defaults, ...program }
}
const programs = setStrongHallProgram(program);
console.log(programs);
const exhibits = setStrongHallProgram(exhibit);
console.log(exhibits);
```



You built the first function with partial application of a higher-order function. You built the second function with hard-coded information. A higher-order function that takes two rounds of arguments will be more flexible than the one with hard-coded information, but it helps to think about how they're similar.

Currying

You know higher-order functions can keep parameters separate, but they have an even more important use: *separating arguments* so that you can reduce the number of arguments that a function needs before it's fully resolved. Building functions that take only one argument at a time is called **"currying"**, and it's an invaluable technique when you're working with methods that pass only one argument. And although currying in its pure form isn't fully supported in JavaScript,6 partially applying a function to reduce parameters to a series of single parameters is common.

Currying and Partial Application

Think back to when you were filtering an array of `dogs` in Tip 22, Create Arrays of a Similar Size with map(). At that point, you only added the filters—you never applied them. Here's a slightly modified version of your array of `dogs`.

```
const dogs = [
    {
        name: 'max',
        weight: 10,
        breed: 'boston terrier',
        state: 'wisconsin',
        color: 'black',
    },
    {
        name: 'don',
```

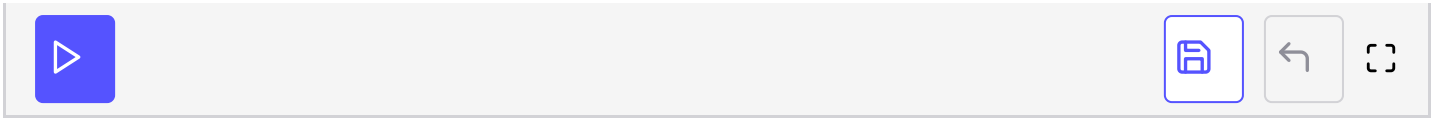
```
        weight: 90,
        breed: 'labrador',
        state: 'kansas',
        color: 'black',
    },
    {
        name: 'shadow',
        weight: 40,
        breed: 'labrador',
        state: 'wisconsin',
        color: 'chocolate',
    },
];
```

Example

> Try to write a function that takes the `dogs` and a filter and returns just the names of the dogs that match the filter.

Pass the dogs as the first parameter and use a combination of array methods—`filter()` and `map()`—to get the final result set.

```
const dogs = [
    {
        name: 'max',
        weight: 10,
        breed: 'boston terrier',
        state: 'wisconsin',
        color: 'black',
    },
    {
        name: 'don',
        weight: 90,
        breed: 'labrador',
        state: 'kansas',
        color: 'black',
    },
    {
        name: 'shadow',
        weight: 40,
        breed: 'labrador',
        state: 'wisconsin',
        color: 'chocolate',
    },
];


function getDogNames(dogs, filter) {
    const [key, value] = filter;
    return dogs
        .filter(dog => dog[key] === value)
        .map(dog => dog.name);
}
console.log(getDogNames(dogs, ['color', 'black']));
```



This function looks pretty good, but it's actually severely limited. There are two issues.

Issue 1

First, your `filter` function is constrained. It will work only when you're doing an exact comparison between a filter and each individual dog. In other words, it works only when using `===`. What if you need to do a different comparison, such as finding all the dogs below a certain weight?

Issue 2

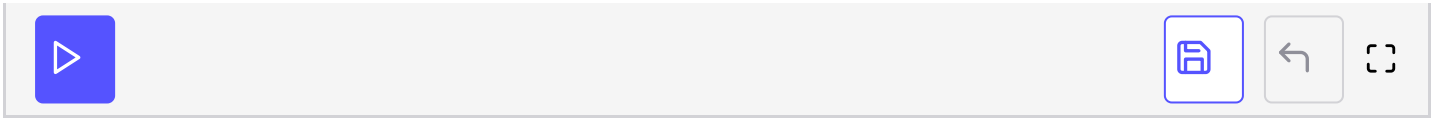
Second, the `map`, like all array methods, can take only one argument—the item being checked—so you have to somehow get your other variables in scope. Because `map` is a function inside another function, it has access to the variables in the wrapper function. That means you'll need to figure out how to pass them in as parameters to the outside function.

Solution: Locking values using a partially applied function

Start by trying to solve the first problem. Rewrite the function so that you can find all `dogs` below a certain weight. As you saw in Tip 32, Write Functions for Testability, you can inject functions into other functions. Start there. Instead of hard coding a comparison function, pass in the `filter` function as a *callback*.

```
const dogs = [
    {
        name: 'max',
        weight: 10,
        breed: 'boston terrier',
        state: 'wisconsin',
        color: 'black',
    },
    {
        name: 'don',
        weight: 90,
        breed: 'labrador',
        state: 'kansas',
        color: 'black',
    },
    {
        name: 'shadow',
        weight: 40,
        breed: 'labrador',
        state: 'wisconsin',
```

```
        color: 'chocolate',
    },
];


function getDogNames(dogs, filterFunc) {
    return dogs
        .filter(filterFunc)
        .map(dog => dog.name)
}

console.log(getDogNames(dogs, dog => dog.weight < 20));
```



You're partway there, but you're still forced to hard code a value, the number `20` in this case. This means you'll still have to either code the value by hand or make sure there are no scope conflicts if you're using a variable. This may not seem like a big deal, but scope conflicts creep in when you least expect them. It's much better to inject values in a function rather than trust them to be able to access variables in an upper scope at runtime.

The goal is to have a partially applied function with some values locked in. You can assign a partially applied function to a variable and pass it as data to another function, which can then provide the remaining arguments.

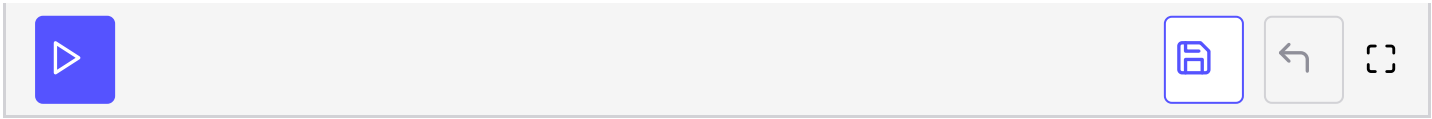
At this point, you don't even need to rewrite your `getDogNames()` function. It takes any comparison function, so you're all set. What you do need to do is rewrite your comparison function so that you don't need to hard code the comparison value.

Use the technique from the previous tip to create two sets of arguments—*the first argument will be a `weight`, the second set will be the individual `dog`.*

Now you can apply the function first with one `weight` and another time with a different `weight`. The actual number will be locked in the function. This means you can reuse the function over and over with different weights. Scope conflicts will be much less likely.

```
const dogs = [
    {
        name: 'max',
        weight: 10,
        breed: 'boston terrier',
        state: 'wisconsin',
        color: 'black',
    },
    {
```

```
        name: 'don',
        weight: 90,
        breed: 'labrador',
        state: 'kansas',
        color: 'black',
    },
    {
        name: 'shadow',
        weight: 40,
        breed: 'labrador',
        state: 'wisconsin',
        color: 'chocolate',
    },
];

function getDogNames(dogs, filterFunc) {
    return dogs
        .filter(filterFunc)
        .map(dog => dog.name)
}

const weightCheck = weight => dog => dog.weight < weight;
console.log(getDogNames(dogs, weightCheck(20)));
console.log(getDogNames(dogs, weightCheck(50)));
```



By currying the function, you've made it so you can pass multiple parameters at different points. You're also able to pass a function around as data.

And the best part is you don't need to limit yourself to just two functions and two sets of arguments. What if you wanted to rewrite your original comparison function using currying?

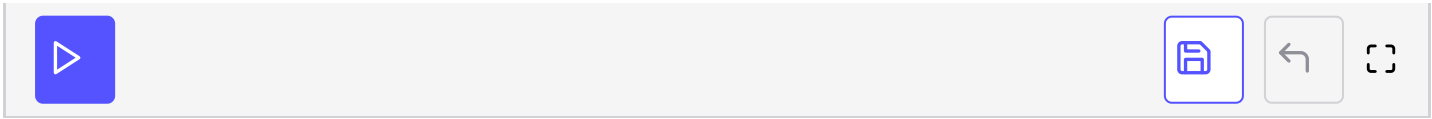
Comparison function using currying

First, you'd pass in the field you want to compare, such as *color*. In the next function, you'd pass the value you want to compare against, such as *black*. The final function takes the individual dog.

The result is a set of comparisons you build up using the same logic but different parameters.

```
const dogs = [
    {
        name: 'max',
        weight: 10,
        breed: 'boston terrier',
        state: 'wisconsin',
        color: 'black',
    },
    {
```

```
    {
        name: 'don',
        weight: 90,
        breed: 'labrador',
        state: 'kansas',
        color: 'black',
    },
    {
        name: 'shadow',
        weight: 40,
        breed: 'labrador',
        state: 'wisconsin',
        color: 'chocolate',
    },
];


const identity = field => value => dog => dog[field] === value;
const colorCheck = identity('color');
const stateCheck = identity('state');
console.log(getDogNames(dogs, colorCheck('chocolate')));
console.log(getDogNames(dogs, stateCheck('kansas')));
```



Now think about what you've created. You took a function that had specific requirements and made something abstract that can take many different comparisons. Because you can assign partially applied functions to variables, they're now just another piece of data you can pass around. This means you can build very sophisticated comparisons using a small set of simple tools.

For example, if you only wanted dogs that meet every criteria, you can pass an array of checks and use the `every()` array method, which returns *true* if all values return *true*.

If you only wanted the dogs that meet at least one criteria, you can write a function that uses the `some()` array method, which returns *true* if any value returns *true*.

```
const dogs = [
    {
        name: 'max',
        weight: 10,
        breed: 'boston terrier',
        state: 'wisconsin',
        color: 'black',
    },
    {
        name: 'don',
        weight: 90,
        breed: 'labrador',
        state: 'kansas',
        color: 'black',
    },
```

```
    {
        name: 'shadow',
        weight: 40,

        breed: 'labrador',
        state: 'wisconsin',
        color: 'chocolate',
    },
];


function allFilters(dogs, ...checks) {
    return dogs
        .filter(dog => checks.every(check => check(dog)))
        .map(dog => dog.name);
}
console.log(allFilters(dogs, colorCheck('black'), stateCheck('kansas')));

function anyFilters(dogs, ...checks) {
    return dogs
        .filter(dog => checks.some(check => check(dog)))
        .map(dog => dog.name);
}
console.log(anyFilters(dogs, weightCheck(20), colorCheck('chocolate')));
```

Does that make your head spin? Hopefully not. Try playing with the code above. That's the best way to learn. Just remember this: If you have functions that can't take more than one argument, currying is a great tool. It makes otherwise complicated problems very straightforward.

1

`multiply` is a currying function. Is this statement true or false?

```
const multiply = (a , b) => a * b
```

2

What will be the output of the following program?

```
let arr = [1,2,3,4,5,6,7,8]

const func = f1 => f2 => arr => arr.filter(f1).map(f2)

const isEven = val => val%2 === 0;
const double = val => val*2;

console.log(func(isEven)(double)(arr));
```

Retake Quiz

In the next tip, you'll see a problem related to variable scope—context. You'll learn how to use arrow functions to solve nagging problems related to the `this` keyword.