Tip 34: Focused Parameters with Partially Applied Functions

In this tip, you'll learn to keep parameters focused with partially applied functions.

We'll cover the following

In the last tip, you saw how you can easily create *higher-order* functions with arrow functions. If you come from an object-oriented background or just haven't seen much code that uses higher-order functions, you may have problems understanding when you should use higher-order functions.

Partially applied functions provide unique value by locking in parameters so you can complete the function later while still maintaining access to the original arguments. They also isolate parameters so you can keep intentions clear. In the next tip, you'll see more about locking in parameter data. In this tip, you'll see how you leverage higher-order functions to give parameters single responsibility.

Partially applied functions & focused parameters

A higher-order function is a function that returns another function. What this means is that you have at least two rounds of parameters before the function is fully resolved. With a *partially applied* function, you pass some parameters and you get back a function that locks those parameters in place while taking more parameters. In other words, a **partially applied** function *reduces* the total number of arguments for a function—*also called the "arity"*— while giving you another function that needs a few more arguments.

The takeaway is that you can have multiple sets of parameters that are independent of one another. Perhaps it seems like parameters already have single responsibility. They are, after all, the input data to the function so they must relate

to one another. That's true, but even inputs have different relationships. Some inputs are related to one another while others are more independent.

Think about an events page on a website. An event is going to occur in a specific space. Each event is unique, but the space isn't going to change radically between events. The *address, name, building hours*, and so on will be the same. In addition, spaces are managed by people who are points of contact, and they will seldom change between events.

With that in mind, consider a function that needs to combine information about the space, the space manager, and an event on a page. You'll likely get each piece of information from a different source, and you'll need to combine them together to return the complete information.

Example

Here's a sample of the data you'll receive.

- The `building` has an `address` and `hours`.
- The `manager` has a `name` and `phone` number.

Then you have two different event types.

- The first, a `program`, will have a specific hour range that's shorter than the building `hours`.
- The second, an `exhibit`, will be open as long as the `building` is open but will need the curator as a `contact`.

```
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',
};
```

At this point, you just need to write a simple function that takes three arguments—`building`, `manager`, `program`/event—and combines them into one set of information.

```
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',
};

function mergeProgramInformation(building, manager, program) {
    const { hours, address } = building;
    const { name, phone } = manager;
    const defaults = {
        hours,
        address,
        contact: name,
        phone,
    };
    return { ...defaults, ...program };
}

console.log(mergeProgramInformation(building,manager,program));
```

Notice every time you call the function for a `building`, you have to pass the same first parameters. The function call is repetitive.

```
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',
};

function mergeProgramInformation(building, manager, program) {
    const { hours, address } = building;
    const { name, phone } = manager;
    const defaults = {
        hours,
        address,
        contact: name,
        phone,
    };
    return { ...defaults, ...program };
}

console.log(mergeProgramInformation(building,manager,program));
console.log(mergeProgramInformation(building, manager, exhibit));
```

This repetition is a clue that your function has a natural division. The first two parameters are establishing a base for a `building`, which is then applied to a series of programs and exhibits.

A *higher-order* function can create single responsibility parameters, allowing you to reuse the first two arguments. The responsibility of the first set of parameters is to gather baseline data. The second set will be the custom information that overrides the baseline.

To accomplish this, you need to make the top function take only two parameters—*the `building` and the `manager`*—and have it return a function that takes only one parameter—*a program (which could be a `program`, an event, an exhibit, and so on).*

```
function mergeProgramInformation(building, manager) {
    const { hours, address } = building;
    const { name, phone } = manager;
    const defaults = {
        hours,
        address,
        contact: name,
        phone,

    };
    return program => {
        return { ...defaults, ...program };
    };
}
```

This can look intimidating, but it's actually simple. Again, a higher-order function is just a function that needs to be called multiple time before it's fully resolved. That's all. To invoke both parts of the functions in a single call, all you have to do is put parentheses right after one another. This invokes the outer function, then immediately invokes the inner function. The result is the same as before.

```
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',
};

function mergeProgramInformation(building, manager) {
    const { hours, address } = building;
    const { name, phone } = manager;
    const defaults = {
        hours,
        address,
        contact: name,
        phone,
    };
    return program => {
        return { ...defaults, ...program };
    };
}

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

You may have given the parameters a single responsibility, but it doesn't eliminate the repetition. Fortunately, with partial application, you can get around that problem also. You'll see how you can reuse a returned function in the next tip.

Reusing the rest operator

To finish up, there's another reason to use partial application and higher-order functions to give your parameters single responsibility: *you can reuse the rest operator*.

As you probably remember from Tip 31, Pass a Variable Number of Arguments with the Rest Operator, nothing can come after the rest parameter. In other words, you can only have a single rest parameter in a set of arguments. That's fine most of the time, but occasionally you'll have a situation where you want to have multiple rest parameters.

This comes up often when you have an array of data and more data that has a one-to-one correspondence with your original data.

For example, if you have a function that takes an array of states and returns the state bird, the resulting array is nice, but you'll eventually need to connect the original and the result together into a nice array of pairs.

```
function getBirds(...states) {
  return ['meadowlark', 'robin', 'roadrunner'];
}

const birds = getBirds('kansas', 'wisconsin', 'new mexico');
console.log(birds);
```

Combining two arrays into pairs is so common that it has a name: **"zip."**

`zip` function

To write a `zip` function that can take multiple parameters, you need to write a *higher-order* function that takes the original array (call it `left`), returns a function that takes the results array (`right`), and combines them. Guess what? Because the parameters are independent, you can use your *rest* parameters both times.

```
function getBirds(...states) {
  return ['meadowlark', 'robin', 'roadrunner'];
}

const zip = (...left) => (...right) => {
    return left.map((item, i) => [item, right[i]]);
};

const birds = getBirds('kansas', 'wisconsin', 'new mexico');
console.log(zip('kansas', 'wisconsin', 'new mexico')(...birds));
```

This isn’t a technique you’ll use often, but it’s very valuable when you want to keep an interface clear. Sometimes parameters just don’t belong together, yet you still need all the information. Partially applied functions are a great way to combine parameters without a lot of effort.

In the next tip, you’ll go even further and learn how you can invoke a function once to capture information and then reuse it over and over again.