Tip 33: Reduce Complexity with Arrow Functions

In this tip, you'll learn how to use arrow functions to destructure arguments, return objects, and construct higher-order functions.

You explored arrow functions once in Tip 20, Simplify Looping with Arrow Functions. It's time to take a deeper dive.

Arrows functions: Deep dive

As a reminder, *arrow functions* allow you to remove extraneous information, such as the *function declaration, parentheses, return statements, even curly braces*. Now you're going to see how to handle a few more concepts that you've just learned, such as *destructuring*. You'll also get an introduction to new ideas that you'll explore further in future tips.

Arrow functions & destructuring

Let's begin with destructuring. You're going to take an object that has a `first` and `last` name and combine them in a string. You can't get more simple than that.

```
const name = {
    first: 'Lemmy',
    last: 'Kilmister',
};

function getName({ first, last }) {
    return `${first} ${last}`;
}
console.log(getName(name));
```

That should be very easy to convert to an arrow function. Remove everything except the parameter and the template literal. Add a fat arrow, `=>`, and you should be done.

Not quite. Everything is the same except the parameters. When you're using any kind of special parameter action—*destructuring, rest parameters, default parameters*—you still need to include the parentheses.

This sounds trivial, but it will trip you up if you aren't aware. It's hard for the JavaScript engine to know if you're performing a function declaration and not an object declaration. You'll get an *error* like this:

```
const getName = {first, last} => `${first} ${last}`;
// Error: Unexpected token '=>'. Expected ';' after variable declaration
```

And that's if you're lucky. If you try this in a *Node.js REPL*, it will just hang like you forgot to add a *closing curly brace*. It can be very confusing.

The solution is simple: If you're using any special parameters, just wrap the parameter in parentheses as you normally would.

```
const comic = {
    first: 'Peter',
    last: 'Bagge',
    city: 'Seattle',
    state: 'Washington',
};
const getName = ({ first, last }) => `${first} ${last}`;
console.log(getName(comic));
```

If you're returning an object, you have to be careful when omitting the `return` statement. Because an arrow function can't tell whether the curly braces are for an object or to wrap a function body, you'll need to indicate the return object by wrapping the whole thing in parentheses.

```
const comic = {
    first: 'Peter',
    last: 'Bagge',
    city: 'Seattle',
    state: 'Washington',
};
const getFullName = ({ first, last }) => ({ fullName: `${first} ${last}` });
console.log(getFullName(comic));
```

It gets even better. When you *return* a value using *parentheses*, you aren't limited to a single line. You can return *multi-line* items while still omitting the `return` statement.

```
const comic = {
    first: 'Peter',
    last: 'Bagge',
    city: 'Seattle',
    state: 'Washington',
};

const getNameAndLocation = ({ first, last, city, state }) => ({
    fullName: `${first} ${last}`,
    location: `${city}, ${state}`,
});

console.log(getNameAndLocation(comic));
```

Arrow functions & higher-order functions

Finally, arrow functions are great ways to make **higher-order functions**—*functions that return other functions*. You'll explore higher-order functions in upcoming tips, so for now, let's just see how to structure them.

Because a *higher-order* function is merely a function that returns another function, the *initial* parameter is the same. And you can return a *function* from the body like you always would.

```
const discounter = discount => {
    return price => {
        return price * (1 - discount);
    };
};
const tenPercentOff = discounter(0.1);
console.log(tenPercentOff(100));
```

Of course, because the return value is another function, you can leverage the implicit return to return the *function* without even needing extra curly braces. Try

it out.

```
const discounter = discount => price => price * (1 - discount);
const tenPercentOff = discounter(0.1);
console.log(tenPercentOff(100));
```

If you're anything like me, you're probably already forgetting all about higher-order functions. *When are you going to use them?* Turns out, they can be very helpful. Not only are they great ways to lock in parameters, but they'll also help you take some of the ideas you've already seen—array methods, rest parameters—even further.

In all the examples, you invoked the higher-order functions by first assigning the returned function to a variable before calling that with another parameter. That's not necessary. You can call one function after the other by just adding the second set of parameters in parentheses right after the first. This essentially turns a higher-order function into a single function with two different parameter sets.

```
const discounter = discount => price => price * (1 - discount);
const result = discounter(0.1)(100);
console.log(result);
```

1

What will be the output of the following code?

```
const person = {
    name: 'Alex',
    age: '20',
    major : 'CS',
    gpa: '3.3',
    job: 'Developer',
    sport: 'tennis',
};
const getCredentials = ({major, gpa , job }) => ({ credentials: `
${major} ${gpa} ${job}` });
```

```
console.log(getCredentials(person));
```

2

The two functions in the code below are equivalent. Is this statement true or false?

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

const multiplyArrow = val1 => val2 => val3 => val1*val2*val3;
```

Retake Quiz

In the next tip, you’ll see why using higher-order functions to keep parameters separate is such a game-changer by learning how to create single responsibility parameters.