Tip 20: Simplify Looping with Arrow Functions

In this tip, you'll learn to cut out extraneous information with arrow functions.

We'll cover the following

- Arrow functions
- Example 1: Function with one parameter
  - Converting to an arrow function
- Example 2: Function with no parameters
  - Before
  - After
- Example 3: Function with more than one parameter
  - Before
  - After
- Example 4: One line functions
  - Before
  - After
- Functions as arguments
  - Anonymous function to arrow function

Arrow functions

In JavaScript, you see a lot of callback functions. Those are functions that are passed as a parameter to other functions. Most of the loops you're about to learn depend on callbacks.

Like most pre-ES6 code, functions are wordy. You have to explicitly declare that you're creating a function with the function keyword, add curly braces to signify the body, use return statements to end the function, and so on. It's not unusual for the callback to be longer than the function you inject it into.

Arrow functions changed that and made writing functions *simple and short*. And learning about them now will make all the loops you see in future tips much more

interesting. It will come as no surprise that arrow functions combined with array methods have led some to abandon for loops altogether.
Well then, what are arrow functions? **Arrow functions** strip away as much extraneous information as possible.

How much extraneous information is there in a function? Quite a bit. It turns out, you can communicate a function without:

- The word function
- Parentheses around arguments
- The word return
- Curly braces

All you need to do is use the fat arrow `=>` to indicate that you'll be building a function. You might be thinking that you've just lost everything that makes a function a function. It's true you can get functions to a minimal state, but there are still a few rules you must follow.

Before you begin, you should know that arrow functions look simple, but they actually have a lot of subtle quirks. Fortunately, you don't need to understand subtleties now. You'll get to those when you explore them more in Tip 36, Prevent Context Confusion with Arrow Functions. To start, look at a function that still has most of the extra stuff (parentheses, curly braces, return statements).

Example 1: Function with one parameter

Here's a simple function that takes an argument and returns a slightly modified value. In this case, it takes a `name` and returns the `name` with a capitalized first letter.

```
function capitalize(name) {
    return name[0].toUpperCase() + name.slice(1);
}
console.log(capitalize('joe'));
```

Easy, right? But before you can translate this into an arrow function, you should notice that this is a **named function**. All that means is the name is declared as part of a function, like this:

```
function namedFunction() {
}
```

That's not the only way to create a function. You can also create an **anonymous function**—*a function that doesn't have a name—and assign it to a variable*:

```
const assignedFunction = function() {
}
```

Here's the same function as an anonymous function. Everything is the same except you're assigning it to a variable.

```
const capitalize = function (name) {
    return name[0].toUpperCase() + name.slice(1);
};

console.log(capitalize('joe'));
```

Converting to an arrow function

An arrow function version uses the same idea: *an anonymous function you assign to a variable*. You can remove the `function` keyword and replace it with a **fat arrow**. As a bonus, if you have exactly **one** parameter (which will be the case for many array methods), you can dispense with the parentheses.

```
const capitalize = name => {
    return name[0].toUpperCase() + name.slice(1);
};

console.log(capitalize('joe'));
```

That's all there is to it. You'll dive deep into where and how to use arrow functions later. For now, look at how normal functions translate to regular functions. To keep it quick, you'll get a rule, then a named function version, then an arrow version. But please try and write it out before you look at the translation. This feature is supported right now in many browsers, so open up a Chrome console and try it out.

Example 2: Function with no parameters

That last example took one parameter, which means that you don’t need the parentheses. But if you have no parameters, you still need parentheses.

Before

```
function key() {
    return 'abc123';
}

console.log(key())
```

After

```
const key = () => {
    return 'abc123';
};

console.log(key())
```

Example 3: Function with more than one parameter

If you have more than one parameter, you’ll also need to use parentheses.

Before

```
const capitalize = name => {
    return name[0].toUpperCase() + name.slice(1);
};

function greet(first, last) {
    return `Oh, hi ${capitalize(first)} ${capitalize(last)}`;
}

console.log(greet('joe', 'morgan'));
```

After

```
const capitalize = name => {
    return name[0].toUpperCase() + name.slice(1);
};

const greet = (first, last) => {
    return `Oh, hi ${capitalize(first)} ${capitalize(last)}`;
};

console.log(greet('joe', 'morgan'));
```

Example 4: One line functions

When the body of your function (the part normally inside the curly braces) is only **one** line, you can move everything—*the fat arrow, the parameters, the return statement*—to a single line.

And if the function itself is only one line, you don't even need to use the `return` keyword. In other words, you `return` the *result of the function body* line automatically.

Before

```
const capitalize = name => {
    return name[0].toUpperCase() + name.slice(1);
};

function formatUser(name) {
    return `${capitalize(name)} is logged in.`;
}

console.log(formatUser('joe'));
```

After

```
const capitalize = name => {
    return name[0].toUpperCase() + name.slice(1);
};

const formatUser = name => `${capitalize(name)} is logged in.`;

console.log(formatUser('joe'));
```

Finally, you can use arrow functions as anonymous functions without needing to assign them to variables. This is how you'll be using it the most in the upcoming tips, so it's worth a look on its own.

Functions as arguments

In JavaScript, you can pass a function as an argument to another function. Functions are just another form of data. Passing a function as an argument is very common for functions that take a callback, a function that will be executed at the end of the logic of the original function.

Here you have a simple function that will apply a custom greeting. You're passing in a function to return a custom greeting as a callback. This is sometimes called **injecting a function**.

```
const capitalize = name => {
    return name[0].toUpperCase() + name.slice(1);
};

const greet = (name) => {
    return `Oh, hi ${name}`;
};

function applyCustomGreeting(name, callback) {
    return callback(capitalize(name));
}

console.log(applyCustomGreeting('joe',greet));
```

It's perfectly okay to create a *named function* and pass that in. But often, it's just more convenient to create an *anonymous function* when you call the original function. In other words, you can call the function `applyCustomGreeting()` and pass in an anonymous function that you write on the spot. You never assign it to a variable first.

```
const capitalize = name => {
    return name[0].toUpperCase() + name.slice(1);
};

function applyCustomGreeting(name, callback) {
  return callback(capitalize(name));
}
```

```
const greeting = applyCustomGreeting('mark', function (name) {
  return `Oh, hi ${name}!`;
});

console.log(greeting);
```

What do you have here? You have a simple anonymous function that takes a single parameter with a body that's a single line long, and that single line is just a `return` statement. This is exactly the situation where arrow functions excel. That anonymous function has so much of that extra fluff you don't need.

Anonymous function to arrow function

Now that you have the tools, try rewriting the anonymous function as an arrow function. You'll get something like this:

```
const capitalize = name => {
    return name[0].toUpperCase() + name.slice(1);
};

function applyCustomGreeting(name, callback) {
  return callback(capitalize(name));
}

const greeting = applyCustomGreeting('mark', name => `Oh, hi ${name}!`);
console.log(greeting);
```

I hope you like the look of that because you're about to see a lot more of that kind of function as you work through array methods. Array methods and arrow functions are convenient ways to update a collection of data.

Q

Study the code below:

```
function multiplybyTwo(val) {
    return val * 2;
}
```

For the function above, which of the options below shows the correct conversion to an arrow function?

conversion to an arrow function?

Retake Quiz

In the next tip, you’ll learn why array methods prevent a lot of mutations and extraneous variables. You’ll start to see why they’ve become so popular in JavaScript.