Arrow Functions

Arrow functions, a.k.a. Fat Arrows (`=>`), are a more concise way of declaring functions. Arrow functions were introduced in ES2015 as a way of solving many of the inconveniences of the normal callback function syntax.

Two major factors influenced the reason behind the desire for arrow functions: the need for shorter functions and behavior of `this` and context.

When you finish this reading you should be able to:

- Define an arrow function
- Given an arrow function, deduce the value of `this` without executing the code

Arrow functions solving problems

Let's start by looking at the arrow function in action!

```
// function declaration
let average = function(num1, num2) {
  let avg = (num1 + num2) / 2;
  return avg;
};

// fat arrow function style!
let averageArrow = (num1, num2) => {
  let avg = (num1 + num2) / 2;
  return avg;
};
```

Both functions in the example above accomplish the same thing. However, the arrow syntax is a little shorter and easier to follow.

Anatomy of an arrow function

The syntax for a multiple statement arrow function is as follows:

```
(parameters, go, here) => {
  statement1;
  statement2;
  return <a value>;
}
```

So let's look at a quick translation between a function declared with a function expression syntax and a fat arrow function. Take notice of the removal of the `function` keyword, and the addition of the fat arrow (`=>`).

```
function fullName(fname, lname) {
  let str = "Hello " + fname + " " + lname;
  return str;
}

// vs.

let fullNameArrow = (fname, lname) => {
  let str = "Hello " + fname + " " + lname;
  return str;
};
```

If there is only a single parameter you may omit the `( )` around the parameter declaration:

```
param1 => {
  statement1;
  return value;
};
```

If you have no parameters with an arrow function you must still use the `( )`:

```
// no parameters will use parenthesis
() => {
  statements;
  return value;
};
```

Let's see an example of an arrow function with a single parameter with no parenthesis:

```
const sayName = name => {
  return "Hello " + name;
};

sayName("Jared"); // => "Hello Jared"
```

Single expression arrow functions

Reminder: In JavaScript, an *expression*is a line of code that returns a value. *Statements*are, more generally, any line of code.

One of the most fun things about single expression arrow functions is they allow for something previously unavailable in JavaScript: **implicit returns**. Meaning, in an arrow function with a single-expression block, the curly braces (`{ }`) and the `return`are keyword are **implied**.

```
argument => expression; // equal to (argument) => { return expression };
```

Look at the below example you can see how we use this snazzy *implicit returns*syntax:

```
const multiply = function(num1, num2) {
  return num1 * num2;
};

// do not need to explicitly state return!
const arrowMultiply = (num1, num2) => num1 * num2;
```

However this doesn't work if the fat arrow uses multiple statements:

```
const halfMyAge = myAge => {
  const age = myAge;
  age / 2;
};

console.log(halfMyAge(30)); // "undefined"
```

To return a value from a fat arrow with multiple statements, you *must*explicitly return:

```
const halfMyAge = myAge => {
  const age = myAge;
  return age / 2;
};

console.log(halfMyAge(30)); // 15
```

Syntactic ambiguity with arrow functions

In Javascript, `{}` can signify either an empty object or an empty block.

```
const ambiguousFunction = () => {};
```

Is `ambiguousFunction`supposed to return an empty object or an empty code block? Confusing right? JavaScript standards state that the curly braces after a

fat arrow evaluate to an empty block (which has the default value of `undefined`):

```
ambiguousFunction(); // undefined
```

To make a single-expression fat arrow return an empty object, wrap that object within parentheses:

```
// this will implicitly return an empty object
const clearFunction = () => ({});
clearFunction(); // returns an object: {}
```

Arrow functions are anonymous

Fat arrows are *anonymous*, like their lambda counterparts in other languages.

```
sayHello(name) => console.log("Hi, " + name); // SyntaxError
(name) => console.log("Hi, " + name); // this works!
```

If you want to name your function you must assign it to a variable:

```
const sayHello = name => console.log("Hi, " + name);

sayHello("Curtis"); // => Hi, Curtis
```

That's about all you need to know for arrow functions syntax-wise. Arrow functions aren't just a different way of writing functions, though. They *behave* differently too - especially when it comes to context!

Arrow functions with context

Arrow functions, unlike normal functions, **carry over context, binding `this` lexically**. In other words, `this` means the same thing inside an arrow function that it does outside of it. Unlike all other functions, the value of `this` inside an arrow function is not dependent on how it is invoked.

Let's do a little compare and contrast to illustrate this point:

```
const testObj = {
  name: "The original object!",
  createFunc: function() {
    return function() {
      return this.name;
    };
  },

  createArrowFunc: function() {
    // the context within this function is the testObj
    return () => {
      return this.name;
    };
  }
};

const noName = testObj.createFunc();
const arrowName = testObj.createArrowFunc();

noName(); // undefined
arrowName(); // The original object!
```

Let's walk through what just happened - we created a `testObj` with two methods that each returned an anonymous function. The difference between these two methods is that the `createArrowFunc` function contained an arrow function inside it. When we invoked both methods we created two function - the `noName` function creating it's own scope and context while

the `arrowName` **kept** the context of the function that created it (`createArrowFunc`'s context of `testObj`).

An arrow function will always have the same context as the function that created it - giving it access to variables available in that context (like `this.name` in this case!)

No binding in arrow functions

One thing to know about arrow functions is since they already have a *bound context*, unlike normal functions, you can't reassign `this`. The `this` in arrow functions is always what it was at the time that the arrow function was declared.

```
const returnName = () => this.name;

returnName(); // undefined

// arrow functions can't be bound
let tryToBind = returnName.bind({ name: "Party Wolf" }); // undefined
tryToBind(); //  will still be undefined
```

What you learned

- How to define an arrow function
- how to deduce the value of `this` in an arrow function