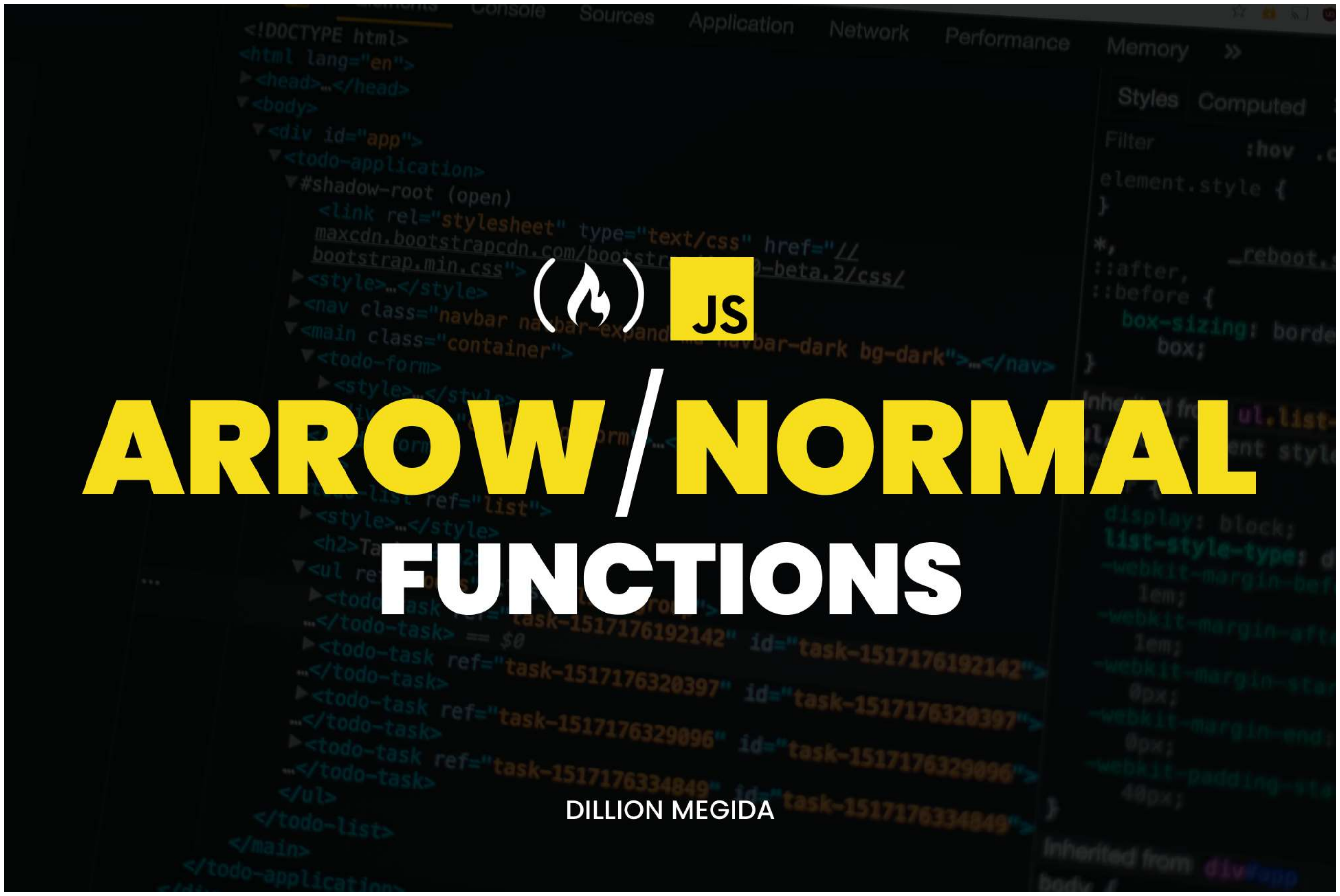
APRIL 13, 2023 / #JAVASCRIPT

Arrow Functions vs Regular Functions in JavaScript – What's the Difference?

Dillion Megida



normal functions and arrow functions. Let's explore the difference between them in this article.

Arrow functions was introduced in ES6. And it introduced a simple and shorter way to create functions.

Here's how to create a normal function, with arguments, which returns something:

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

If you want to transform this into an arrow function, here's what you'll have:

```
const multiply = (num1, num2) => {
  const result = num1 * num2
  return result
}
```

If the `return` statement is the only statement in the function, you can even have a shorter function expression. For example:

This function only contains the `return` statement. With arrow functions, we can have something shorter like this:

```
const multiply = (num1, num2) => num1 * num2
```

We skip the curly braces and the `return` keyword. Shorter; one-liner.

But the syntax of writing both types of functions is not the only difference. There's more, so let's look at them.

I have a video version of this topic which you can also check out :)

1. No `arguments` object in arrow functions

A normal function has an `arguments` object which you can access in the function:

```
function print() {
  console.log(arguments)
}
```

```
print("hello", 400, false)

// {
//   '0': 'hello',
//   '1': 400,
//   '2': false
// }
```

As you can see here, the three arguments passed when calling `print()` are contained in the `arguments` object which we log to the console. We can access the first argument with `arguments[0]`, the second with `arguments[1]` and the third with `arguments[2]`

But this object does not exist in arrow functions. Let's try it out. Say we have `print` using an arrow function:

```
const print = () => {
  console.log(arguments)
}

print("hello", 400, false)
// Uncaught ReferenceError: arguments is not defined
```

Now we have a reference error: **arguments is not defined**. That's because the `arguments` variable does not exist in arrow functions.

In normal functions, a `this` variable is created which references the objects that call them. For example:

```
const obj = {
  name: 'deeecode',
  age: 200,
  print: function() {
    console.log(this)
  }
}

obj.print()
// {
//   name: 'deeecode',
//   age: 200,
//   print: [Function: print]
// }
```

As you can see here, the `this` in the `print` method points to `obj` , which is the object that calls the method.

Here's another example:

```
const obj = {
  name: 'deeecode',
  age: 200,
  print: function() {
    function print2() {
      console.log(this)
    }

    print2()
  }
```

Here, we have two functions. The first one is `print` which is a method of the `obj` object. The second is `print2` which is a function declared inside `print`. `print2()` is also called directly.

In this case, `print` is called by `obj` (`obj.print()`) but no object calls `print2` (`print2()`). So the `this` in `print2` would reference the window object by default.

Now let's see what happens with an arrow function.

```
const obj = {
  name: 'deeecode',
  age: 200,
  print: () => {
    console.log(this)
  }
}

obj.print()
// Window
```

By using an arrow function for `print`, this function does not automatically create a `this` variable. As a result, any reference to `this` would point to what `this` was before the function was created.

As you see in the result, `this` was pointing to the `Window` object before `print` was created.

```
const obj = {
  name: 'deeecode',
  age: 200,
  print: function() {
    const print2 = () => {
      console.log(this)
    }

    print2()
  }
}

obj.print()
// {
//   name: 'deeecode',
//   age: 200,
//   print: [Function: print]
// }
```

Here, we have `print` as a normal function which means a `this` variable is automatically created in it. Then we have `print2` which is an arrow function.

Because `obj` is calling `print` (as in `obj.print()`), the `this` in `print` would point to `obj` .

Since `print2` is an arrow function, it doesn't create its own `this` variable. Therefore, any reference to `this` would point to what the value of `this` was before the function was created. In this case where `obj` calls `print` , `this` was pointing to `obj` before `print2` was created. As you can see in the results, by logging `this` from `print2` , `obj` is the result.

3. Arrow functions cannot be used as constructors

With normal functions, you can create constructors which serve as a special function for instantiating an object from a class.

Here is an example of an `Animal` class which we instantiate two objects from:

```
class Animal {
  constructor(name, numOfLegs) {
    this.name = name
    this.numOfLegs = numOfLegs
  }

  sayName() {
    console.log(`My name is ${this.name}`)
  }
}

const Dog = new Animal("Bingo", 4)
const Bird = new Animal("Steer", 2)

Dog.sayName()
// My name is Bingo

Bird.sayName()
// My name is Steer
```

Here, we have the `Animal` constructor which can be instantiated with different parameters. In the constructor, two arguments are required: `name` and `noOfLegs`.

In the case of `Bird`, we create a new instance of `Animal` using "Steer" as the `name` and 2 as the `noOfLegs`.

By calling `sayName` on `Dog` and `Bird`, you can see how the method works currently with each object. The `this` variable points to the objects and the `name` field is gotten from each of them.

The `this` variable is very important for classes and constructors. In point 2, we saw that arrow functions cannot create their own `this`. For this reason, arrow functions also cannot be used as constructors.

Let's attempt it and see what happens:

```
class Animal {
  constructor = (name, numOfLegs) => {
    this.name = name
    this.numOfLegs = numOfLegs
  }

  sayName() {
    console.log(`My name is ${this.name}`)
  }
}

// Uncaught SyntaxError: Classes may not have a field named 'constructor'
```

Here, we have an arrow function used for the `constructor`. But, we get a SyntaxError: **Classes may not have a field named 'constructor'**.

name.

But, we can use arrow functions for the methods in the class without getting errors. For example:

```
class Animal {
  constructor (name, numOfLegs){
    this.name = name
    this.numOfLegs = numOfLegs
  }

  sayName = () => {
    console.log(`My name is ${this.name}`)
  }
}

const Dog = new Animal("Bingo", 4)

Dog.sayName()
// My name is Bingo
```

Here, we have a normal function for the `constructor`, and an arrow function for the `sayName` method. `sayName` is a field. And we do not get errors.

By calling `sayName()` on `Dog`, we still get "My name is Bingo". Though `sayName` as an arrow function does not create its own `this`, remember that any reference to `this` would point to the value of it before the arrow function was created. In this case, the value of `this` pointed to `Dog` before `sayName` was created.

When it comes to functions, you need to understand **function declaration** and **function expression**.

Function declarations involve the `function` keyword and a name for the function. For example:

```
function printHello() {
  console.log("hello")
}
```

`printHello` is a **declared function.** But, check out this example:

```
const printHello = function() {
  console.log("hello")
}
```

In this case, `printHello` is not a declared function. We have an anonymous function (not named) on the right side of the assignment operator. This function is a **function expression**, which is assigned to the `printHello` variable.

Though the `function` keyword is used, there is no name assigned, which makes it an expression and not a declaration. To prove that it is not a declaration, try the following:

Because this expression is not assigned to a variable, you get an error: **SyntaxError: Function statements require a function name**

Back to arrow functions. Normal functions can be declared when you use the function keyword and a name, but arrow functions cannot be declared. They can only be expressed because they are anonymous:

```
const printHello = () => {
  console.log("hello")
}
```

As you see here, we have an anonymous function (starting from `() => ...`) which is assigned to the `printHello` variable. `printHello` is not a declared function here. It is a variable that holds the evaluated value from the function expression.

5 Arrow functions cannot be accessed before initialization

Hoisting is a concept where a variable or function is lifted to the top of its global or local scope before the whole code is executed. This makes it possible for such a variable/function to be accessed before initialization. Here's a function example:

```
function printName() {
  console.log("i am dillion")
}

// i am dillion
// hello
```

As you can see here, we called `printName` before it was actually declared in the code. But we don't get any errors. `printName()` is executed (logging "i am dillion" to the console) before `console.log("hello")`.

What happens here is hoisting.

The `printName` function is raised to the top of the global scope (the scope it is declared in) before the whole code is executed, thereby making it possible to execute the function earlier.

But not all kinds of functions can be accessed before initialization. All functions and variables in JavaScript are hoisted, but **only declared functions can be accessed before initialization.**

Here's an example with an arrow function:

```
printName()

console.log("hello")

const printName = () => {
  console.log("i am dillion")
```

Running this code, you get an error: **ReferenceError: Cannot access 'printName' before initialization**.

As we saw in point 4, `printName` is not a declared function. It is a variable, declared with `const` which is assigned a function expression. Variables declared with `let` and `const` are hoisted, but they cannot be accessed before the line they are initialized.

Let's say we use `var` for our arrow function:

```
printName()

console.log("hello")

var printName = () => {
  console.log("i am dillion")
}

// TypeError: printName is not a function
```

Here, we have declared the `printName` variable with `var`. The error we get now is **TypeError: printName is not a function**. The reason for this is that variables declared with `var` are hoisted and accessible, but they have a default value of `undefined`. So attempting to access `printName` before the line it was initialized with the function expression is interpreted as `undefined()`, and as you know, "undefined is not a function".

Wrap up

Although arrow functions allow you to write functions more concisely, they also have limitations. As we have seen in this article, some of the behaviors that occur in normal functions do not happen in arrow functions.

If you want to create constructors, retain the normal behavior of `this` or have your functions hoisted, then arrow functions are not the right approach.

If you enjoyed this article, please share it with others :)

Dillion Megida

Developer Advocate and Content Creator passionate about sharing my knowledge on Tech. I simplify JavaScript / ReactJS / NodeJS / Frameworks / TypeScript / et al My YT channel: youtube.com/c/deeecode

Trending Guides

JS isEmpty Equivalent	Coalesce SQL
Submit a Form with JS	Python join()
Add to List in Python	JS POST Request
Grep Command in Linux	JS Type Checking
String to Int in Java	Read Python File
Add to Dict in Python	SOLID Principles
Java For Loop Example	Sort a List in Java

Forum Donate

Learn to code — free 3,000-hour curriculum

Nested Lists in Python	SQL CONVERT Function
Rename Column in Pandas	Create a File in Terminal
Delete a File in Python	Clear Formatting in Excel
K-Nearest Neighbors Algo	Accounting Num Format Excel
iferror Function in Excel	Check if File Exists Python
Remove From String Python	Iterate Over Dict in Python

Our Charity

About Alumni Network Open Source Shop Support Sponsors Academic Honesty

Code of Conduct Privacy Policy Terms of Service Copyright Policy