## **FACTORY FUNCTIONS AND THE MODULE PATTERN**

### **What’s wrong with constructors?**

Object constructors are one of about a million ways to start organizing your code. They are fairly common in the wild and are a fundamental building block of the JavaScript language.

*However…*

There are many people who argue *against* using constructors at all. Their arguments boil down to the fact that if you aren’t careful, it can be easy to introduce bugs into your code when using constructors. [This](https://tsherif.wordpress.com/2013/08/04/constructors-are-bad-for-javascript/) article does a pretty decent job of outlining the issues (spoiler alert: the author ends up recommending factory functions).

One of the biggest issues with constructors is that while they *look* just like regular functions, they do not behave like regular functions at all. If you try to use a constructor function without the new keyword, your program will not work as expected, but it won’t produce error messages that are easy to trace.

The main takeaway is that while constructors aren’t necessarily *evil*, they aren’t the only way, and they may not be the best way either. Of course, this doesn’t mean that time learning about them was wasted! They are a common pattern in real-world code and many tutorials that you’ll come across on the net.

### **Learning Outcomes**

By the end of this lesson, you should be able to do the following:

* Describe common bugs you might run into using constructors.
* Write a factory method that returns an object.
* Explain how scope works in JavaScript (bonus points if you can point out what ES6 changed!).
* Explain what Closure is and how it impacts private functions & variables.
* Describe how private functions & variables are useful.
* Use inheritance in objects using the factory pattern.
* Explain the module pattern.
* Describe IIFE. What does it stand for?
* Briefly explain namespacing and how it’s useful.

### **Factory function introduction**

The factory function pattern is similar to constructors, but instead of using new to create an object, factory functions simply set up and return the new object when you call the function. Check out this example:

const personFactory = (name, age) => { const sayHello = () => console.log('hello!'); return { name, age, sayHello }; }; const jeff = personFactory('jeff', 27); console.log(jeff.name); // 'jeff' jeff.sayHello(); // calls the function and logs 'hello!'

For reference, here is the same thing created using the constructor pattern:

const Person = function(name, age) { this.sayHello = () => console.log('hello!'); this.name = name; this.age = age; }; const jeff = new Person('jeff', 27);

#### **Object Shorthand**

A quick note about line 3 from the factory function example. In 2015, a handy new shorthand for creating objects was added into JavaScript. Without the shorthand, line 3 would have looked something like this:

return {name: name, age: age, sayHello: sayHello}

Put simply, if you are creating an object where you are referring to a variable that has the exact same name as the object property you’re creating, you can condense it like so:

return {name, age, sayHello}

With that knowledge in your pocket, check out this little hack:

const name = "Maynard" const color = "red" const number = 34 const food = "rice" // logging all of these variables might be a useful thing to do, // but doing it like this can be somewhat confusing. console.log(name, color, number, food) // Maynard red 34 rice // if you simply turn them into an object with brackets, // the output is much easier to decipher: console.log({name, color, number, food}) // { name: 'Maynard', color: 'red', number: 34, food: 'rice' }

### **Scope and Closure**

From simply reading the above example, you are probably already in pretty good shape to start using factory functions in your code. Before we get there though, it’s time to do a somewhat deep dive into an incredibly important concept: closure.

However, before we’re able to make sense of closure, we need to make sure you have a *really* good grasp on scope in JavaScript. Scope is the term that refers to where things like variables and functions can be used in your code.

In the following example, do you know what will be logged on the last line?

let a = 17; const func = x => { let a = x; }; func(99); console.log(a); // ???????

Is it 17 or 99? Do you know why? Can you edit the code so that it prints the other value?

The answer is 17, and the reason it’s not 99 is that on line 4, the outer variable a is not redefined, rather a *new* a is created inside the scope of that function. In the end, figuring out scope in most contexts is not all that complicated, but it is *crucial* to understanding some of the more advanced concepts that are coming up soon, so take your time to understand what’s going on in the following resources.

1. [This video](https://www.youtube.com/watch?v=SBwoFkRjZvE) is simple and clear! Start here.
2. [This article](https://toddmotto.com/everything-you-wanted-to-know-about-javascript-scope/) starts simple and reiterates what the video covered, but goes deeper and is more specific about the appropriate terminology. At the end, he defines closure *and* describes the module pattern, both of which we’ll talk about more soon.

* The previous article is great, but there is one inaccurate statement:  
  All scopes in JavaScript are created with Function Scope *only*, they aren’t created by for or while loops or expression statements like if or switch. New functions = new scope - that’s the rule  
  That statement *was* true in 2013 when the article was written, but ES6 has rendered it incorrect. Read [this](http://wesbos.com/javascript-scoping/) article to get the scoop!

### **Private Variables and Functions**

Now that we’ve cemented your knowledge of scope in JavaScript, take a look at this example:

const FactoryFunction = string => { const capitalizeString = () => string.toUpperCase(); const printString = () => console.log(`----${capitalizeString()}----`); return { printString }; }; const taco = FactoryFunction('taco'); printString(); // ERROR!! capitalizeString(); // ERROR!! taco.capitalizeString(); // ERROR!! taco.printString(); // this prints "----TACO----"

Because of the concept of scope, neither of the functions created inside of FactoryFunction can be accessed outside of the function itself, which is why lines 9, 10, and 11 fail. The only way to use either of those functions is to return them in the object (see line 4), which is why we can call taco.printString() but *not* taco.capitalizeString(). The big deal here is that even though *we* can’t access the capitalizeString() function, printString() can. That is closure.

The concept of closure is the idea that functions retain their scope even if they are passed around and called outside of that scope. In this case, printString has access to everything inside of FactoryFunction, even if it gets called outside of that function.

Here’s another example:

const counterCreator = () => { let count = 0; return () => { console.log(count); count++; }; }; const counter = counterCreator(); counter(); // 0 counter(); // 1 counter(); // 2 counter(); // 3

In this example, counterCreator initializes a local variable (count) and then returns a function. To use that function, we have to assign it to a variable (line 9). Then, every time we run the function it console.logs count and increments it. Keep in mind, counter() is calling the return value of counterCreator. As above, the function counter is a closure. It has access to the variable count and can both print and increment it, but there is no other way for our program to access that variable.

In the context of factory functions, closures allow us to create private variables and functions. Private functions are functions that are used in the workings of our objects that are not intended to be used elsewhere in our program. In other words, even though our objects might only do one or two things, we are free to split our functions up as much as we want (allowing for cleaner, easier to read code) and only export the functions that the rest of the program is going to use. Using this terminology with our printString example from earlier, capitalizeString is a private function and printString is public.

The concept of private functions is very useful and should be used as often as possible! For every bit of functionality that you need for your program, there are likely to be several supporting functions that do NOT need to be used in your program as a whole. Tucking these away and making them inaccessible makes your code easier to refactor, easier to test, and easier to reason about for you and anyone else that wants to use your objects.

### **Back to Factory Functions**

Now that we’ve got the theory out of the way, let’s return to factory functions. Factories are simply plain old JavaScript functions that return objects for us to use in our code. Using factories is a powerful way to organize and contain the code you’re writing. For example, if we’re writing any sort of game, we’re probably going to want objects to describe our players and encapsulate all of the things our players can do (functions!).

const Player = (name, level) => { let health = level \* 2; const getLevel = () => level; const getName = () => name; const die = () => { // uh oh }; const damage = x => { health -= x; if (health <= 0) { die(); } }; const attack = enemy => { if (level < enemy.getLevel()) { damage(1); console.log(`${enemy.getName()} has damaged ${name}`); } if (level >= enemy.getLevel()) { enemy.damage(1); console.log(`${name} has damaged ${enemy.getName()}`); } }; return {attack, damage, getLevel, getName} }; const jimmie = Player('jim', 10); const badGuy = Player('jeff', 5); jimmie.attack(badGuy);

Take a minute to look through this example and see if you can figure out what’s going on.

What would happen here if we tried to call jimmie.die()? What if we tried to manipulate the health: jimmie.health -= 1000? Well, those are things that we have NOT exposed publicly so we would get an error. This is a very good thing! Setting up objects like this makes it easier for us to use them because we’ve actually put some thought into how and when we are going to want to use the information. In this case, we have jimmie’s health hiding as a private variable inside of the object which means we need to export a function if we want to manipulate it. In the long run, this will make our code *much* easier to reason about because all of the logic is encapsulated in an appropriate place.

#### **Inheritance with factories**

In the constructors lesson, we looked fairly deeply into the concept of prototypes and inheritance, or giving our objects access to the methods and properties of another object. There are a few easy ways to accomplish this while using factories. Check this one out:

const Person = (name) => { const sayName = () => console.log(`my name is ${name}`) return {sayName} } const Nerd = (name) => { // simply create a person and pull out the sayName function with destructuring assignment syntax! const {sayName} = Person(name) const doSomethingNerdy = () => console.log('nerd stuff') return {sayName, doSomethingNerdy} } const jeff = Nerd('jeff') jeff.sayName() //my name is jeff jeff.doSomethingNerdy() // nerd stuff

This pattern is *great* because it allows you to pick and choose which functions you want to include in your new object. If you want to go ahead and lump ALL of another object in, you can certainly do that as well with Object.assign (read the docs for that one [here](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign)).

const Nerd = (name) => { const prototype = Person(name) const doSomethingNerdy = () => console.log('nerd stuff') return Object.assign({}, prototype, {doSomethingNerdy}) }

* Before moving on, have a look at [this](https://medium.com/javascript-scene/3-different-kinds-of-prototypal-inheritance-es6-edition-32d777fa16c9) article. In the second half of the article, the author goes into some things that we aren’t really talking too much about here, but you’ll be rewarded if you spend some time figuring out what he’s talking about. Good stuff!

### **The Module Pattern**

Quick sidenote: ES6 introduced a new feature in JavaScript called ‘modules’. These are essentially a syntax for importing and exporting code between different JavaScript files. They’re very powerful and we WILL be covering them later. They are *not*, however, what we’re talking about here.

Modules are actually very similar to factory functions. The main difference is how they’re created.

Meet a module:

const calculator = (() => { const add = (a, b) => a + b; const sub = (a, b) => a - b; const mul = (a, b) => a \* b; const div = (a, b) => a / b; return { add, sub, mul, div, }; })(); calculator.add(3,5) // 8 calculator.sub(6,2) // 4 calculator.mul(14,5534) // 77476

The concepts are exactly the same as the factory function. However, instead of creating a factory that we can use over and over again to create multiple objects, the module pattern wraps the factory in an IIFE (Immediately Invoked Function Expression).

* Read up about IIFE’s in [this article](http://adripofjavascript.com/blog/drips/an-introduction-to-iffes-immediately-invoked-function-expressions.html). The concept is simple: write a function, wrap it in parentheses, and then immediately call the function by adding () to the end of it.
* An example of creating and using a module pattern: [JavaScript Module Pattern Basics](https://coryrylan.com/blog/javascript-module-pattern-basics).
* Additional example of creating and using a module pattern: [Module pattern in JavaScript](https://dev.to/tomekbuszewski/module-pattern-in-javascript-56jm).
* For those who prefer video lessons, here is an excellent YouTube series on modular JS that covers most of the content in this guide: [Modular Javascript](https://www.youtube.com/playlist?list=PLoYCgNOIyGABs-wDaaxChu82q_xQgUb4f).

In our calculator example above, the function inside the IIFE is a simple factory function, but we can just go ahead and assign the object to the variable calculator since we aren’t going to need to be making lots of calculators, we only need one. Just like the factory example, we can have as many private functions and variables as we want, and they stay neatly organized, tucked away inside of our module, only exposing the functions we actually want to use in our program.

A useful side-effect of encapsulating the inner workings of our programs into objects is namespacing. Namespacing is a technique that is used to avoid naming collisions in our programs. For example, it’s easy to imagine scenarios where you could write multiple functions with the same name. In our calculator example, what if we had a function that added things to our HTML display, and a function that added numbers and operators to our stack as the users input them? It is conceivable that we would want to call all three of these functions, which, of course, would cause trouble in our program. If all of them were nicely encapsulated inside of an object, then we would have no trouble: calculator.add(), displayController.add(), operatorStack.add().

### **Additional Resources**

This section contains helpful links to other content. It isn’t required, so consider it supplemental if you need to dive deeper into something.

* [Learning JavaScript Design Patterns by Addy Osmani](https://addyosmani.com/resources/essentialjsdesignpatterns/book/)