

# The Flavors of Object-Oriented Programming (in JavaScript)



Zell Liew on Sep 28, 2020 (Updated on Oct 29, 2020)

In my research, I've found there are four approaches to Object-Oriented Programming in JavaScript:

1. [Using Constructor functions \(#constructor-functions\)](#)
2. [Using Classes \(#class-syntax\)](#)
3. [Using Objects linking to other objects \(OLOO\) \(#oloo\)](#)
4. [Using Factory functions \(#factory-functions\)](#)

Which methods should I use? Which one is “the best” way? Here I'll present my findings along with information that may help you decide which is right for you.

To make that decision, we're not just going to look at the different flavors but compare conceptual aspects between them:

1. [Classes vs. Factory functions – Inheritance \(#inheritance\)](#)
2. [Classes vs. Factory functions – Encapsulation \(#encapsulation\)](#)
3. [Classes vs. Factory functions – this \(#this-variable\)](#)
4. [Classes vs. Factory functions – Event listeners \(#event-listeners\)](#)

Let's start with a foundation of OOP in JavaScript.

↳ **[\(#what-is-object-oriented-programming\)](#) What is Object-Oriented Programming?**

Object-Oriented Programming is a way of writing code that allows you to create different objects from a common object. The common object is usually called a *blueprint* while the created objects are called *instances*.

Each instance has properties that are not shared with other instances. For example, if you have a Human blueprint, you can create human instances with different names.

The second aspect of Object-Oriented Programming is about *structuring* code when you have multiple levels of blueprints. This is commonly called Inheritance or subclassing.

The third aspect of Object Oriented Programming is about *encapsulation* where you hide certain pieces of information within the object so they're not accessible.

**Hey!** If you need more than this brief intro, here's an article (<https://scotch.io/tutorials/object-oriented-programming-in-javascript>) that introduces this aspect of Object-Oriented Programming if you need help with it.

Let's begin with the basics — an introduction to the four flavors of Object-Oriented Programming.

## ↳ **(#the-four-flavors-of-object-oriented-programming)** **The four flavors of Object-Oriented Programming**

There are four ways to write Object-Oriented Programming in JavaScript. They are:

Part 1: [Using Constructor functions \(#constructor-functions\)](#)

Part 2: [Using Classes \(#class-syntax\)](#)

Part 3: [Using Objects Linking to Other Objects \(OLOO\) \(#oloo\)](#)

Part 4: [Using Factory functions \(#factory-functions\)](#)

## ↳ **(#using-constructor-functions)** **Using Constructor functions**

Constructors are functions that contain a `this` keyword.

```
function Human (firstName, lastName) {  
  this.firstName = firstName  
  this.lastName = lastName  
}
```

this lets you store (and access) unique values created for each instance. You can create an instance with the `new` keyword.

```
const chris = new Human('Chris', 'Coyier')  
console.log(chris.firstName) // Chris  
console.log(chris.lastName) // Coyier  
  
const zell = new Human('Zell', 'Liew')  
console.log(zell.firstName) // Zell  
console.log(zell.lastName) // Liew
```

## ↳ (#class-syntax) Class syntax

Classes are said to be the “syntactic sugar” of Constructor functions. As in, Classes are an easier way of writing Constructor functions.

**Hey!** There's serious contention about whether Classes are bad (like this (<https://everyday.codes/javascript/please-stop-using-classes-in-javascript/>) and this (<https://www.toptal.com/javascript/es6-class-chaos-keeps-js-developer-up>)). We're not going to dive into those arguments here. Instead, we're just going to look at how to write code with Classes and decide whether Classes are better than constructors based on the code we write.

Classes can be written with the following syntax:

```
class Human {  
  constructor(firstName, lastName) {  
    this.firstName = firstName  
    this.lastName = lastName  
  }  
}
```

Notice the `constructor` function contains the same code as the Constructor syntax above? We need to do this since we want to initialize values into `this`. (We can skip `constructor` if we don't need to initialize values. More on this later under [Inheritance \(#inheritance\)](#)).

At first glance, classes seem to be inferior to constructors — there's more code to write! Hold your horses and don't form a conclusion at this point. We have a lot more to cover. Classes begin to shine later.

As before, you can create an instance with the `new` keyword.

```
const chris = new Human('Chris', 'Coyier')

console.log(chris.firstName) // Chris
console.log(chris.lastName) // Coyier
```

## ↳ (#objects-linking-to-other-objects-oloo) Objects Linking to Other Objects (OLOO)

---

OLOO was coined and popularized by Kyle Simpson (<https://twitter.com/getify>). In OLOO, you define the blueprint as a normal object. You then use a method (often named `init`, but that isn't required in the way `constructor` is to a `Class`) to initialize the instance.

```
const Human = {
  init (firstName, lastName) {
    this.firstName = firstName
    this.lastName = lastName
  }
}
```

You use `Object.create` to create an instance. After creating the instance, you need to run your `init` function.

```
const chris = Object.create(Human)
chris.init('Chris', 'Coyier')

console.log(chris.firstName) // Chris
console.log(chris.lastName) // Coyier
```

You can chain `init` after `Object.create` if you returned `this` inside `init`.

```
const Human = {
  init () {
    // ...
    return this
  }
}
```

```

    }
  }

  const chris = Object.create(Human).init('Chris', 'Coyier')
  console.log(chris.firstName) // Chris
  console.log(chris.lastName) // Coyier

```

## ▷ (#factory-functions) Factory functions

Factory functions are functions that return an object. You can return any object. You can even return a Class instance or OLOO instance — and it'll still be a valid Factory function.

Here's the simplest way to create Factory functions:

```

JavaScript
function Human (firstName, lastName) {
  return {
    firstName,
    lastName
  }
}

```

You don't need `new` to create instances with Factory functions. You simply call the function.

```

JavaScript
const chris = Human('Chris', 'Coyier')

console.log(chris.firstName) // Chris
console.log(chris.lastName) // Coyier

```

Now that we've seen these four OOP setup possibilities, let's look at how you declare properties and methods on each of them so we can get a little better understanding of working with them before getting to the bigger comparisons we're trying to make.

## ▷ (#declaring-properties-and-methods) Declaring properties and methods

Methods are functions declared as an object's property.

```
JavaScript
const someObject = {
  someMethod () { /* ... */ }
}
```

In Object-Oriented Programming, there are two ways to declare properties and methods:

1. Directly on the instance
2. In the Prototype

Let's learn to do both.

## ↳ (#declaring-properties-and-methods-with-constructors) Declaring properties and methods with Constructors

If you want to declare a property directly on an instance, you can write the property inside the constructor function. Make sure to set it as the property for `this`.

```
JavaScript
function Human (firstName, lastName) {
  // Declares properties
  this.firstName = firstName
  this.lastname = lastName

  // Declares methods
  this.sayHello = function () {
    console.log(`Hello, I'm ${firstName}`)
  }
}

const chris = new Human('Chris', 'Coyier')
console.log(chris)
```

Methods are commonly declared on the Prototype because Prototype allows instances to use the same method. It's a smaller "code footprint."

To declare properties on the Prototype, you need to use the `prototype` property.

```
JavaScript
function Human (firstName, lastName) {
  this.firstName = firstName
```

```

    this.lastname = lastName
  }

  // Declaring method on a prototype
  Human.prototype.sayHello = function () {
    console.log(`Hello, I'm ${this.firstName}`)
  }

```

It can be clunky if you want to declare multiple methods in a Prototype.

```

// Declaring methods on a prototype
Human.prototype.method1 = function () { /*...*/ }
Human.prototype.method2 = function () { /*...*/ }
Human.prototype.method3 = function () { /*...*/ }

```

You can make things easier by using merging functions like `Object.assign`.

```

Object.assign(Human.prototype, {
  method1 () { /*...*/ },
  method2 () { /*...*/ },
  method3 () { /*...*/ }
})

```

**Hey!** `Object.assign` does not support the merging of Getter and Setter functions. You need another tool. [Here's why](https://zellwk.com/blog/copy-properties-of-one-object-to-another-object/) (<https://zellwk.com/blog/copy-properties-of-one-object-to-another-object/>). And here's a tool (<https://zellwk.com/blog/creating-a-deep-assign-library/>) I created to merge objects with Getters and Setters.

## ↳ (#declaring-properties-and-methods-with-classes) Declaring properties and methods with Classes

You can declare properties for each instance inside the `constructor` function.

```

class Human {
  constructor (firstName, lastName) {
    this.firstName = firstName
    this.lastname = lastName

    this.sayHello = function () {
      console.log(`Hello, I'm ${firstName}`)
    }
  }
}

```

It's easier to declare methods on the prototype. You write the method after constructor like a normal function.

```
class Human (firstName, lastName) {  
  constructor (firstName, lastName) { /* ... */ }  
  
  sayHello () {  
    console.log(`Hello, I'm ${this.firstName}`)  
  }  
}
```

It's easier to declare multiple methods on Classes compared to Constructors. You don't need the `Object.assign` syntax. You just write more functions.

**Hey!** **Note:** there's no `,` between method declarations in a Class.

```
class Human (firstName, lastName) {  
  constructor (firstName, lastName) { /* ... */ }  
  
  method1 () { /*...*/ }  
  method2 () { /*...*/ }  
  method3 () { /*...*/ }  
}
```

## ↳ (#declaring-properties-and-methods-with-oloo) Declaring properties and methods with OLOO

You use the same process for declaring properties and methods on an instance. You assign them as a property of `this`.

```
const Human = {  
  init (firstName, lastName) {  
    this.firstName = firstName  
    this.lastName = lastName  
    this.sayHello = function () {  
      console.log(`Hello, I'm ${firstName}`)  
    }  
  
    return this  
  }  
}
```



```
const chris = Object.create(Human).init('Chris', 'Coyier')
console.log(chris)
```

To declare methods in the prototype, you write the method like a normal object.

```
const Human = {
  init () { /*...*/ },
  sayHello () {
    console.log(`Hello, I'm ${this.firstName}`)
  }
}
```

JavaScript

## ↳ (#declaring-properties-and-methods-with-factory-functions) Declaring properties and methods with Factory functions

You can declare properties and methods directly by including them in the returned object.

```
function Human (firstName, lastName) {
  return {
    firstName,
    lastName,
    sayHello () {
      console.log(`Hello, I'm ${firstName}`)
    }
  }
}
```

JavaScript

You cannot declare methods on the Prototype when you use Factory functions. If you really want methods on the prototype, you need to return a Constructor, Class, or OLOO instance. (Don't do this since it doesn't make any sense.)

```
// Do not do this
function createHuman (...args) {
  return new Human(...args)
}
```

JavaScript

## ▷ (#where-to-declare-properties-and-methods) Where to declare properties and methods

Should you declare properties and methods directly on the instance? Or should you use prototype as much as you can?

Many people take pride that JavaScript is a “Prototypal Language” (which means it uses prototypes). From this statement, you may make the assumption that using “Prototypes” is better.

The real answer is: **It doesn’t matter.**

If you declare properties and methods on instances, each instance will take up slightly more memory. If you declare methods on Prototypes, the memory used by each instance will decrease, but not much. This difference is insignificant with computer processing power what it is today. Instead, you want to look at how easy it is to write code — and whether it is possible to use Prototypes in the first place.

For example, if you use Classes or OLOO, you’ll be better off using Prototypes since the code is easier to write. If you use Factory functions, you cannot use Prototypes. You can only create properties and methods directly on the instance.

I wrote a separate article on [understanding JavaScript Prototypes](https://zellwk.com/blog/javascript-prototype) (<https://zellwk.com/blog/javascript-prototype>) if you’re interested in finding out more.

## ▷ (#preliminary-verdict) Preliminary verdict

We can make a few notes from the code we wrote above. These opinions are my own!

- 1. Classes are better than Constructors** because its easier to write multiple methods on Classes.
- 2. OLOO is weird because of the `Object.create` part.** I gave OLOO a run for a while, but I always forget to write `Object.create`. It’s weird enough for me not to use it.
- 3. Classes and Fackry Functions are easiest to use.** The problem is that Factory functions don’t support Prototypes. But like I said, this doesn’t really matter in production.

We're down to two. Should we choose Classes or Factory functions then? Let's compare them!

## ▷ [\(#classes-vs-factory-functions-inheritance\)](#) **Classes vs. Factory functions — Inheritance**

To continue the discussion on Classes and Factory functions, we need to understand three more concepts that are tied closely to Object-Oriented Programming.

1. Inheritance
2. Encapsulation
3. `this`

Let's start with Inheritance.

### ▷ [\(#what-is-inheritance\)](#) **What is Inheritance?**

Inheritance is a loaded word. Many people in the industry use Inheritance incorrectly, in my opinion. The word “inheritance” is used when you receive things from somewhere. For example:

- ⦿ If you get an inheritance from your parents, it means you get money and assets from them.
- ⦿ If you inherit genes from your parents, it means you get your genes from them.
- ⦿ If you inherit a process from your teacher, it means you get that process from them.

Fairly straightforward.

In JavaScript, Inheritance can mean the same thing: where you get properties and methods from the parent blueprint.

**This means *all* instances actually inherit from their blueprints.** They inherit properties and methods in two ways:

1. by creating a property or method directly upon creating the instance
2. via the Prototype chain

We discussed how to do both methods in the [previous article](http://#) (<http://#>) so refer back to it if you need help seeing these processes in code.

There's a *second* meaning for Inheritance in JavaScript — where you create a derivative blueprint from the parent blueprint. This process is more accurately called *Subclassing*, but people sometimes will call this Inheritance as well.

## ▷ [\(#understanding-subclassing\)](#) Understanding Subclassing

Subclassing is about creating a derivative blueprint from a common blueprint. You can use any Object-Oriented Programming flavor to create the Subclass.

We'll talk about this with the Class syntax first because it's easier to understand.

## ▷ [\(#subclassing-with-class\)](#) Subclassing with Class

When you create a Subclass, you use the `extends` keyword.

```
class Child extends Parent {  
  // ... Stuff goes here  
}
```

For example, let's say we want to create a `Developer` class from a `Human` class.

```
// Human Class  
class Human {  
  constructor (firstName, lastName) {  
    this.firstName = firstName  
    this.lastName = lastName  
  }  
  
  sayHello () {  
    console.log(`Hello, I'm ${this.firstName}`)  
  }  
}
```

The `Developer` class will extend `Human` like this:

```

JavaScript
class Developer extends Human {
  constructor(firstName, lastName) {
    super(firstName, lastName)
  }

  // Add other methods
}

```

**Hey! Note:** `super` calls the `Human` (also called the “parent”) Class. It initiates the `constructor` from `Human`. If you don’t need extra initiation code, you can omit `constructor` entirely.

```

JavaScript
class Developer extends Human {
  // Add other methods
}

```

Let’s say a `Developer` can code. We can add the `code` method directly to `Developer`.

```

JavaScript
class Developer extends Human {
  code (thing) {
    console.log(`${this.firstName} coded ${thing}`)
  }
}

```

Here’s an example of an instance of `Developer`:

```

JavaScript
const chris = new Developer('Chris', 'Coyier')
console.log(chris)

```



Instance of a `Developer` class.

## ↳ (#subclassing-with-factory-functions) Subclassing with Factory functions

There are four steps to creating Subclasses with Factory functions:

1. Create a new Factory function
2. Create an instance of the Parent blueprint
3. Create a new copy of this instance
4. Add properties and methods to this new copy

The process looks like this:

```
function Subclass (...args) {  
  const instance = ParentClass(...args)  
  return Object.assign({}, instance, {  
    // Properties and methods go here  
  })  
}
```

We'll use the same example — creating a Developer Subclass — to illustrate this process. Here's the Human factory function:

```
function Human (firstName, lastName) {  
  return {  
    firstName,  
    lastName,  
    sayHello () {  
      console.log(`Hello, I'm ${firstName}`)  
    }  
  }  
}
```

We can create Developer like this:

```
function Developer (firstName, lastName) {  
  const human = Human(firstName, lastName)  
  return Object.assign({}, human, {  
    // Properties and methods go here  
  })  
}
```

Then we add the code method like this:

```
function Developer (firstName, lastName) {  
  const human = Human(firstName, lastName)  
  return Object.assign({}, human, {  
    code (thing) {  
      console.log(`${this.firstName} coded ${thing}`)  
    }  
  })  
}
```

Here's an example of a Developer instance :

```
JavaScript
const chris = Developer('Chris', 'Coyier')
console.log(chris)
```

 Example of a Developer instance with Factory functions.

**Hey!** **Note:** You cannot use `Object.assign` if you use Getters and Setters. You'll need another tool, like [mix](https://zellwk.com/blog/creating-a-deep-assign-library/) (<https://zellwk.com/blog/creating-a-deep-assign-library/>) . I explain why in [this article](https://zellwk.com/blog/copy-properties-of-one-object-to-another-object/) (<https://zellwk.com/blog/copy-properties-of-one-object-to-another-object/>).

## ↳ (#overwriting-the-parents-method) Overwriting the Parent's method

Sometimes you need to overwrite the Parent's method inside the Subclass. You can do this by:


1. Creating a method with the same name
2. Calling the Parent's method (optional)
3. Changing whatever you need in the Subclass's method

The process looks like this with Classes:

```
JavaScript
class Developer extends Human {
  sayHello () {
    // Calls the parent method
    super.sayHello()

    // Additional stuff to run
    console.log(`I'm a developer.`)
  }
}

const chris = new Developer('Chris', 'Coyier')
chris.sayHello()
```

 Overwriting a parent's method.

The process looks like this with Factory functions:

```
JavaScript
function Developer (firstName, lastName) {
  const human = Human(firstName, lastName)
```


```

return Object.assign({}, human, {
  sayHello () {
    // Calls the parent method
    human.sayHello()

    // Additional stuff to run
    console.log(`I'm a developer.`)
  }
})
}

const chris = new Developer('Chris', 'Coyier')
chris.sayHello()

```

 Overwriting a parent's method.

## ↳ (#inheritance-vs-composition) Inheritance vs. Composition

No talk about Inheritance ever concludes without the mention of Composition. Experts like [Eric Elliot](https://twitter.com/_ericelliott) ([https://twitter.com/\\_ericelliott](https://twitter.com/_ericelliott)) often suggests we should favor [Composition over Inheritance](https://medium.com/javascript-scene/composing-software-an-introduction-27b72500d6ea) (<https://medium.com/javascript-scene/composing-software-an-introduction-27b72500d6ea>).

*“Favor object composition over class inheritance” the Gang of Four, “Design Patterns: Elements of Reusable Object Oriented Software”*

*“In computer science, a composite data type or compound data type is any data type which can be constructed in a program using the programming language’s primitive data types and other composite types. [...] The act of constructing a composite type is known as composition.” ~ Wikipedia*

So let's give Composition a deeper look and understand what it is.

## ↳ (#understanding-composition) Understanding Composition

Composition is the act of combining two things into one. It's about merging things together. The most common (and simplest) way of merging objects is with `Object.assign`.

```

const one = { one: 'one' }
const two = { two: 'two' }
const combined = Object.assign({}, one, two)

```

JavaScript



The use of Composition can be better explained with an example. Let's say we already have two Subclasses, a `Designer` and `Developer`. Designers can design, while developers can code. Both designers and developers inherit from the `Human` class.


Here's the code so far:

```
class Human {  
  constructor(firstName, lastName) {  
    this.firstName = firstName  
    this.lastName = lastName  
  }  
  
  sayHello () {  
    console.log(`Hello, I'm ${this.firstName}`)  
  }  
}  
  
class Designer extends Human {  
  design (thing) {  
    console.log(`${this.firstName} designed ${thing}`)  
  }  
}  
  
class Developer extends Designer {  
  code (thing) {  
    console.log(`${this.firstName} coded ${thing}`)  
  }  
}
```

Now let's say you want to create a third Subclass. This Subclass is a mix of a `Designer` and a `Developer` — they can design and code. Let's call it `DesignerDeveloper` (or `DeveloperDesigner`, whichever you fancy).

How would you create the third Subclass?

We cannot extend `Designer` and `Developer` classes at the same time. This is impossible because we cannot decide which properties come first. This is often called The Diamond Problem.

 Diamond problem.

The Diamond Problem can be easily solved if we do something like `Object.assign` — where we prioritize one object over the other. If we use the `Object.assign` approach, we may be able to extend classes like this. But this is not supported in JavaScript.

JavaScript

```
// Doesn't work
class DesignerDeveloper extends Developer, Designer {
  // ...
}
```

So we need to rely on Composition.

Composition says: Instead of trying to create `DesignerDeveloper` via Subclassing, let's create a new object that stores common features. We can then include these features whenever necessary.

In practice, it can look like this:

JavaScript

```
const skills = {
  code (thing) { /* ... */ },
  design (thing) { /* ... */ },
  sayHello () { /* ... */ }
}
```

We can then skip `Human` altogether and create three different classes based on their skills.


Here's the code for `DesignerDeveloper`:

JavaScript

```
class DesignerDeveloper {
  constructor (firstName, lastName) {
    this.firstName = firstName
    this.lastName = lastName

    Object.assign(this, {
      code: skills.code,
      design: skills.design,
      sayHello: skills.sayHello
    })
  }
}

const chris = new DesignerDeveloper('Chris', 'Coyier')
console.log(chris)
```

 Composing methods into a class

You can do the same with Developer and Designer.

JavaScript

```
class Designer {
  constructor (firstName, lastName) {
    this.firstName = firstName
    this.lastName = lastName

    Object.assign(this, {
      design: skills.design,
      sayHello: skills.sayHello
    })
  }
}

class Developer {
  constructor (firstName, lastName) {
    this.firstName = firstName
    this.lastName = lastName


    Object.assign(this, {
      code: skills.code,
      sayHello: skills.sayHello
    })
  }
}
```

Did you notice we're creating methods directly on the instance? This is just one option. We can still put methods into the Prototype, but I think the code looks clunky. (It's as if we're writing Constructor functions all over again.)

JavaScript

```
class DesignerDeveloper {
  constructor (firstName, lastName) {
    this.firstName = firstName
    this.lastName = lastName
  }
}

Object.assign(DesignerDeveloper.prototype, {
  code: skills.code,
  design: skills.design,
  sayHello: skills.sayHello
})
```


 Composition via Classes by putting methods into the Prototype.

Feel free to use whatever code structure you're attracted to. The results are kinda the same anyway.

## ▷ **(#composition-with-factory-functions) Composition with Factory Functions**

Composition with Factory functions is essentially adding the shared methods into the returned object.

```
function DesignerDeveloper (firstName, lastName) {  
  return {  
    firstName,  
    lastName,  
    code: skills.code,  
    design: skills.design,  
    sayHello: skills.sayHello  
  }  
}
```

 Composing methods into a factory function

## ▷ **(#inheritance-and-composition-at-the-same-time) Inheritance and Composition at the same time**


Nobody says we can't use Inheritance and Composition at the same time. We can!

Using the example we've ironed out so far, `Designer`, `Developer`, and `DesignerDeveloper` Humans are still humans. They can extend the `Human` object.

Here's an example where we use both inheritance and composition with the class syntax.


```
class Human {  
  constructor (firstName, lastName) {  
    this.firstName = firstName  
    this.lastName = lastName  
  }  
  
  sayHello () {  
    console.log(`Hello, I'm ${this.firstName}`)  
  }  
}  
  
class DesignerDeveloper extends Human {}  
Object.assign(DesignerDeveloper.prototype, {  
  code: skills.code,
```

```
    design: skills.design
  })
```

 Subclassing and Composition at the same time.

And here's the same thing with Factory functions:

```
function Human (firstName, lastName) {  
  return {  
    firstName,  
    lastName,  
    sayHello () {  
      console.log(`Hello, I'm ${this.firstName}`)  
    }  
  }  
}  
  
function DesignerDeveloper (firstName, lastName) {  
  const human = Human(firstName, lastName)  
  return Object.assign({}, human, {  
    code: skills.code,  
    design: skills.design  
  })  
}
```

 Subclassing and Composition in Factory functions


## ▷ **(#subclassing-in-the-real-world) Subclassing in the real world**

One final point about Subclassing vs. Composition. Even though experts have pointed out that Composition is more flexible (and hence more useful), Subclassing still has its merits. Many things we use today are built with the Subclassing strategy.

For example: The `click` event we know and love is a `MouseEvent`. `MouseEvent` is a Subclass of a `UIEvent`, which in turn is a Subclass of `Event`.

 `MouseEvent` is a subclass of `UIEvent`.

Another example: HTML Elements are Subclasses of Nodes. That's why they can use all properties and methods of Nodes.

 `HTMLElement` is a subclass of `Node`.

## ▷ **(#preliminary-verdict) Preliminary verdict**

Classes and Factory functions can both use Inheritance and Composition. Composition seems to be cleaner in Factory functions though, but that's not a big win over Classes.

We'll examine Classes and Factory Functions more in detail next.

## ▷ (#classes-vs-factory-functions-encapsulation) **Classes vs. Factory functions — Encapsulation**

We've looked at the four different Object-Oriented Programming flavors so far. Two of them — Classes and Factory functions — are easier to use compared to the rest.

But the questions remain: Which should you use? And why?

To continue the discussion on Classes and Factory functions, we need to understand three concepts that are tied closely to Object-Oriented Programming:

1. Inheritance
2. Encapsulation
3. `this`

We just talked about Inheritance. Now let's talk about Encapsulation.

## ▷ (#encapsulation) **Encapsulation**

Encapsulation is a **big** word, but it has a simple meaning. Encapsulation is the act of enclosing one thing inside another thing so the thing inside doesn't leak out. Think about storing water inside a bottle. The bottle prevents water from leaking out.

In JavaScript, we're interested in enclosing variables (which can include functions) so these variables don't leak out into the external scope. This means you need to understand scope to understand encapsulation. We'll go through an explanation, but you can also use [this article \(https://css-tricks.com/javascript-scope-closures/\)](https://css-tricks.com/javascript-scope-closures/) to beef up your knowledge regarding scopes.

## ↳ (#simple-encapsulation) Simple Encapsulation


The simplest form of Encapsulation is a block scope.

```
JavaScript
{
  // Variables declared here won't leak out
}
```

When you're in the block, you can access variables that are declared outside the block.

```
JavaScript
const food = 'Hamburger'


{
  console.log(food)
}
```

 Logs food from inside the block. Result: Hamburger.

But when you're outside the block, you cannot access variables that are declared inside the block.

```
JavaScript
{
  const food = 'Hamburger'
}

console.log(food)
```

 Logs food from outside the block. Results: Error.


**Hey!** **Note:** Variables declared with `var` don't respect block scope. This is why I recommend you use `let` or `const` (<https://zellwk.com/blog/es6/#let-and-const>) to declare variables.

## ↳ (#encapsulating-with-functions) Encapsulating with functions

Functions behave like block scopes. When you declare a variable inside a function, they cannot leak out of that function. This works for all variables, even those declared with `var`.


```
JavaScript
function sayFood () {
  const food = 'Hamburger'
}
```

```
sayFood()  
console.log(food)
```

 Logs food from outside the function. Results: Error.

Likewise, when you're inside the function, you can access variables that are declared outside of that function.

```
JavaScript  
const food = 'Hamburger'  
  
function sayFood () {  
  console.log(food)  
}  
  
sayFood()
```

 Logs food from inside the function. Result: Hamburger.

Functions can return a value. This returned value can be used later, outside the function.

```
JavaScript  
function sayFood () {  
  return 'Hamburger'  
}  
  
console.log(sayFood())
```

 Logs return value from function. Result: Hamburger.

## ↳ (#closures) Closures

Closures are an advanced form of Encapsulation. They're simply functions wrapped in functions.


```
JavaScript  
// Here's a closure  
function outsideFunction () {  
  function insideFunction () { /* ... */ }  
}
```

Variables declared in `outsideFunction` can be used in `insideFunction`.



JavaScript

```
function outsideFunction () {  
  const food = 'Hamburger'  
  console.log('Called outside')  
  
  return function insideFunction () {  
    console.log('Called inside')  
    console.log(food)  
  }  
}  
  
// Calls `outsideFunction`, which returns `insideFunction`  
// Stores `insideFunction` as variable `fn`  
const fn = outsideFunction()  
  
// Calls `insideFunction`  
fn()
```

 Closure logs.

## ▷ (#encapsulation-and-object-oriented-programming) Encapsulation and Object-Oriented Programming

When you build objects, you want to make some properties publicly available (so people can use them). But you also want to keep some properties private (so others can't break your implementation).

Let's work through this with an example to make things clearer. Let's say we have a Car blueprint. When we produce new cars, we fill each car up with 50 liters of fuel.

JavaScript

```
class Car {  
  constructor () {  
    this.fuel = 50  
  }  
}
```

Here we exposed the `fuel` property. Users can use `fuel` to get the amount of fuel left in their cars.

JavaScript

```
const car = new Car()  
console.log(car.fuel) // 50
```

Users can also use the `fuel` property to set any amount of fuel.

```
const car = new Car()
car.fuel = 3000
console.log(car.fuel) // 3000
```

Let's add a condition and say that each car has a maximum capacity of 100 liters. With this condition, we don't want to let users set the `fuel` property freely because they may break the car.

There are two ways to do prevent users from setting `fuel` :

1. Private by convention
2. Real Private Members

### ▷ (#private-by-convention) Private by convention

In JavaScript, there's a practice of prepending underscores to a variable name. This denotes the variable is private and should not be used.

```
class Car {
  constructor () {
    // Denotes that `_fuel` is private. Don't use it!
    this._fuel = 50
  }
}
```

We often create methods to get and set this “private” `_fuel` variable.

```
class Car {
  constructor () {
    // Denotes that `_fuel` is private. Don't use it!
    this._fuel = 50
  }

  getFuel () {
    return this._fuel
  }

  setFuel (value) {
    this._fuel = value
    // Caps fuel at 100 liters
    if (value > 100) this._fuel = 100
  }
}
```

Users should use the `getFuel` and `setFuel` methods to get and set fuel.

```
JavaScript
const car = new Car()
console.log(car.getFuel()) // 50

car.setFuel(3000)
console.log(car.getFuel()) // 100
```

But `_fuel` is not actually private. It is still a public variable. You can still access it, you can still use it, and you can still abuse it (even if the abusing part is an accident).

```
JavaScript
const car = new Car()
console.log(car.getFuel()) // 50

car._fuel = 3000
console.log(car.getFuel()) // 3000
```

We need to use real private variables if we want to completely prevent users from accessing them.

## ▷ (#real-private-members) Real Private Members


Members here refer to variables, functions, and methods. It's a collective term.

## ▷ (#private-members-with-classes) Private Members with Classes

Classes let you create private members by prepending `#` to the variable.

```
JavaScript
class Car {
  constructor () {
    this.#fuel = 50
  }
}
```

Unfortunately, you can't use `#` directly inside a `constructor` function.

 Error when declaring `#` directly in constructor function.

You need to declare the private variable outside the constructor first.

```
JavaScript
class Car {
  // Declares private variable
```


```
#fuel
constructor () {
  // Use private variable
  this.#fuel = 50
}
}
```

In this case, we can use a shorthand and declare `#fuel` upfront since we set fuel to 50.

```
JavaScript
class Car {
  #fuel = 50
}
```

You cannot access `#fuel` outside `Car`. You'll get an error.

```
JavaScript
const car = new Car()
console.log(car.#fuel)
```

 Cannot access `#fuel`.

You need methods (like `getFuel` or `setFuel`) to use the `#fuel` variable.

```
JavaScript
class Car {
  #fuel = 50

  getFuel () {
    return this.#fuel
  }

  setFuel (value) {
    this.#fuel = value
    if (value > 100) this.#fuel = 100
  }
}

const car = new Car()
console.log(car.getFuel()) // 50

car.setFuel(3000)
console.log(car.getFuel()) // 100
```

**Hey!** Note: I prefer Getters and Setters instead of `getFuel` and `setFuel`. [The syntax is easier to read.](https://zellwk.com/blog/why-use-getter-and-setter-functions/)  
(<https://zellwk.com/blog/why-use-getter-and-setter-functions/>)

JavaScript

```
class Car {  
  #fuel = 50  
  
  get fuel () {  
    return this.#fuel  
  }  
  
  set fuel (value) {  
    this.#fuel = value  
    if (value > 100) this.#fuel = 100  
  }  
}  
  
const car = new Car()  
console.log(car.fuel) // 50  
  
car.fuel = 3000  
console.log(car.fuel) // 100
```

## ▷ (#private-members-with-factory-functions) Private Members with Factory functions

Factory functions create Private Members automatically. You just need to declare a variable like normal. Users will not be able to get that variable anywhere else. This is because variables are function-scoped and hence encapsulated by default.

JavaScript

```
function Car () {  
  const fuel = 50  
}  
  
const car = new Car()  
console.log(car.fuel) // undefined  
console.log(fuel) // Error: `fuel` is not defined
```

We can create getter and setter functions to use this private fuel variable.

JavaScript

```
function Car () {  
  const fuel = 50  
  
  return {  
    get fuel () {  
      return fuel  
    },  
  
    set fuel (value) {  
      fuel = value  
    }  
  }  
}
```

```
        if (value > 100) fuel = 100
      }
    }
  }

  const car = new Car()
  console.log(car.fuel) // 50

  car.fuel = 3000
  console.log(car.fuel) // 100
```

That's it! Simple and easy!

## ↳ (#verdict-for-encapsulation) Verdict for Encapsulation

Encapsulation with Factory functions are simpler and easier to understand. They rely on the scopes which are a big part of the JavaScript language.

Encapsulation with Classes, on the other hand, requires prepending # to the private variable. This can make things clunky.

We'll look at the final concept — `this` to complete the comparison between Classes and Factory functions — in the next section.

## ↳ (#classes-vs-factory-functions-the-this-variable) Classes vs. Factory Functions — The `this` variable

`this` (ha!) is one of the main arguments against using Classes for Object-Oriented Programming. Why? Because `this` value changes depending on how it is used. It can be confusing for many developers (both new and experienced).

But the concept of `this` is relatively simple in reality. There are only six contexts in which you can use `this`. If you master these six contexts, you'll have no problems using `this`.

The six contexts are:

1. In a global context
2. In an object construction
3. In an object property / method
4. In a simple function
5. In an arrow function
6. In an event listener

I covered these six contexts in detail. (<https://zellwk.com/blog/this/>) Give it a read if you need help understanding `this`.

**Hey!** **Note:** Don't shy away from learning to use (<https://zellwk.com/blog/should-you-use-this/>) `this`. It's an important concept you need to understand if you intend on mastering JavaScript.


Come back to this article after you've solidified your knowledge on `this`. We'll have a deeper discussion about using `this` in Classes and Factory functions.

Back yet? Good. Let's go!

## ↳ (#using-this-in-classes) Using `this` in Classes

`this` refers to the instance when used in a Class. (It uses the "In an object property / method" context.) This is why you can set properties and methods on the instance inside the constructor function.

```
class Human {  
  constructor (firstName, lastName) {  
    this.firstName = firstName  
    this.lastName = lastName  
    console.log(this)  
  }  
}  
  
const chris = new Human('Chris', 'Coyier')
```

 `<code>this</code>` points to the instance

## ↳ (#using-this-in-constructor-functions) Using `this` in Constructor functions


If you use `this` inside a function and `new` to create an instance, `this` will refer to the instance. This is how a Constructor function is created.

```

JavaScript
function Human (firstName, lastName) {
  this.firstName = firstName
  this.lastName = lastName
  console.log(this)
}

const chris = new Human('Chris', 'Coyier')

```

 `<code>this</code>` points to the instance.


I mentioned Constructor functions because you can use `this` inside Factory functions. But `this` points to Window (or undefined if you use ES6 Modules, or a bundler like webpack).

```

JavaScript
// NOT a Constructor function because we did not create instances with the `new` keyword
function Human (firstName, lastName) {
  this.firstName = firstName
  this.lastName = lastName
  console.log(this)
}

const chris = Human('Chris', 'Coyier')

```

 `<code>this</code>` points to Window.

Essentially, when you create a Factory function, you should not use `this` as if it's a Constructor function. This is one small hiccup people experience with `this`. I wanted to highlight the problem and make it clear.

## ▷ (#using-this-in-a-factory-function) Using `this` in a Factory function

The correct way to use `this` in a Factory function is to use it “in an object property / method” context.


```

JavaScript
function Human (firstName, lastName) {
  return {
    firstName,
    lastName,
    sayThis () {
      console.log(this)
    }
  }
}

```



```
const chris = Human('Chris', 'Coyier')
chris.sayThis()
```

 `<code>this</code>` points to the instance.

Even though you can use `this` in Factory functions, you don't need to use them. You can create a variable that points to the instance. Once you do this, you can use the variable instead of `this`. Here's an example at work.

```
function Human (firstName, lastName) {  
  const human = {  
    firstName,  
    lastName,  
    sayHello() {  
      console.log(`Hi, I'm ${human.firstName}`)  
    }  
  }  
}  
  
return human  
}  
  
const chris = Human('Chris', 'Coyier')  
chris.sayHello()
```

JavaScript

`human.firstName` is clearer than `this.firstName` because `human` definitely points back to the instance. You know when you see the code.

If you're used to JavaScript, you may also notice there's no need to even write `human.firstName` in the first place! Just `firstName` is enough because `firstName` is in the lexical scope. (Read this article (<https://css-tricks.com/javascript-scope-closures/>) if you need help with scopes.)

```
function Human (firstName, lastName) {  
  const human = {  
    firstName,  
    lastName,  
    sayHello() {  
      console.log(`Hi, I'm ${firstName}`)  
    }  
  }  
}  
  
return human  
}
```

JavaScript

```
const chris = Human('Chris', 'Coyier')
chris.sayHello()
```

Runs `chris.sayHello()`

What we covered so far is simple. It's not easy to decide whether `this` is actually needed until we create a sufficiently complicated example. So let's do that.

## ▷ (#detailed-example) Detailed example

Here's the setup. Let's say we have a `Human` blueprint. This `Human` has `firstName` and `lastName` properties, and a `sayHello` method.

We have a `Developer` blueprint that's derived from `Human`. `Developers` can code, so they'll have a `code` method. `Developers` also want to proclaim they're developers, so we need to overwrite `sayHello` and add `I'm a Developer` to the console.

We'll create this example with `Classes` and `Factory functions`. (We'll make an example with `this` and an example without `this` for `Factory functions`).

## ▷ (#the-example-with-classes) The example with `Classes`

First, we have a `Human` blueprint. This `Human` has a `firstName` and `lastName` properties, as well as a `sayHello` method.

```
class Human {
  constructor (firstName, lastName) {
    this.firstName = firstName
    this.lastName = lastName
  }

  sayHello () {
    console.log(`Hello, I'm ${this.firstName}`)
  }
}
```

JavaScript

We have a `Developer` blueprint that's derived from `Human`. `Developers` can code, so they'll have a `code` method.

JavaScript

```
class Developer extends Human {
  code (thing) {
    console.log(`${this.firstName} coded ${thing}`)
  }
}
```

Developers also want to proclaim that they're developers. We need to overwrite `sayHello` and add `I'm a Developer` to the console. We do this by calling `Human`'s `sayHello` method. We can do this using `super`.

JavaScript

```
class Developer extends Human {
  code (thing) {
    console.log(`${this.firstName} coded ${thing}`)
  }

  sayHello () {
    super.sayHello()
    console.log('I'm a developer')
  }
}
```

## ↳ (#the-example-with-factory-functions-with-this) The example with Factory functions (with `this`)

Again, first, we have a `Human` blueprint. This `Human` has `firstName` and `lastName` properties, as well as a `sayHello` method.

JavaScript

```
function Human () {
  return {
    firstName,
    lastName,
    sayHello () {
      console.log('Hello, I'm ${this.firstName}')
    }
  }
}
```

Next, we have a `Developer` blueprint that's derived from `Human`. Developers can code, so they'll have a `code` method.

JavaScript

```
function Developer (firstName, lastName) {
  const human = Human(firstName, lastName)
  return Object.assign({}, human, {
```

```

    code (thing) {
      console.log(`${this.firstName} coded ${thing}`)
    }
  })
}

```

Developers also want to proclaim they're developers. We need to overwrite `sayHello` and add `I'm a Developer` to the console.

We do this by calling `Human`'s `sayHello` method. We can do this using the `human` instance.

```

function Developer (firstName, lastName) {
  const human = Human(firstName, lastName)
  return Object.assign({}, human, {
    code (thing) {
      console.log(`${this.firstName} coded ${thing}`)
    },

    sayHello () {
      human.sayHello()
      console.log('I\'m a developer')
    }
  })
}

```

JavaScript

## [2. \(#the-example-with-factory-functions-without-this\) The example with Factory functions \(\*without this\*\)](#)

Here's the full code using Factory functions (with `this`):

```

function Human (firstName, lastName) {
  return {
    firstName,
    lastName,
    sayHello () {
      console.log(`Hello, I'm ${this.firstName}`)
    }
  }
}

function Developer (firstName, lastName) {
  const human = Human(firstName, lastName)
  return Object.assign({}, human, {
    code (thing) {
      console.log(`${this.firstName} coded ${thing}`)
    },

```

JavaScript

```

    sayHello () {
      human.sayHello()
      console.log('I\'m a developer')
    }
  })
}

```

Did you notice `firstName` is available within the lexical scope in both `Human` and `Developer`? This means we can omit `this` and use `firstName` directly in both blueprints.

```

function Human (firstName, lastName) {
  return {
    // ...
    sayHello () {
      console.log(`Hello, I'm ${firstName}`)
    }
  }
}

function Developer (firstName, lastName) {
  // ...
  return Object.assign({}, human, {
    code (thing) {
      console.log(`${firstName} coded ${thing}`)
    },

    sayHello () { /* ... */ }
  })
}

```

See that? This means you can safely omit `this` from your code when you use Factory functions.

## ↳ **(#verdict-for-this) Verdict for this**

In simple terms, Classes require `this` while Factory functions don't. I prefer Factory functions here because:

1. The context of `this` can change (which can be confusing)
2. The code written with factory functions is shorter and cleaner (since we can use encapsulated variables without writing `this.#variable`).

Next up is the last section where we build a simple component together with both Classes and Factory functions. You get to see how they differ and how to use event listeners with each flavor.

## ↳ **(#classes-vs-factory-functions-event-listeners) Classes vs Factory functions — Event listeners**

Most Object-Oriented Programming articles show you examples without event listeners. Those examples can be easier to understand, but they don't reflect the work we do as frontend developers. The work we do requires event listeners — for a simple reason — because we need to build things that rely on user input.

Since [event listeners change the context of this \(https://zellwk.com/blog/this/\)](https://zellwk.com/blog/this/), they can make Classes troublesome to deal with. At the same time, they make Factory functions more appealing.

But that's not really the case.

The change in `this` doesn't matter if you know how to handle `this` in both Classes and Factory functions. Few articles cover this topic so I thought it would be good to complete this article with a simple component using Object-Oriented Programming flavors.

## ↳ **(#building-a-counter) Building a counter**

We're going to build a simple counter in this article. We'll use everything you learned in this article — including private variables.

Let's say the counter contains two things:

1. The count itself
2. A button to increase the count

Here's the simplest possible HTML for the counter:

```

<div class="counter">
  <p>Count: <span>0</span>
  <button>Increase Count</button>
</div>

```

## ↳ (#building-the-counter-with-classes) Building the Counter with Classes

To make things simple, we'll ask users to find and pass the counter's HTML into a Counter class.

```

class Counter () {
  constructor (counter) {
    // Do stuff
  }
}

// Usage
const counter = new Counter(document.querySelector('.counter'))

```

We need to get two elements in the Counter class:

1. The <span> that contains the count – we need to update this element when the count increases
2. The <button> – we need to add an event listener to this element class

```

Counter () {
  constructor (counter) {
    this.countElement = counter.querySelector('span')
    this.buttonElement = counter.querySelector('button')
  }
}

```

We'll initialize a count variable and set it to what the countElement shows. We'll use a private #count variable since the count shouldn't be exposed elsewhere.

```

class Counter () {
  #count
  constructor (counter) {
    // ...

    this.#count = parseInt(countElement.textContent)
  }
}

```

```
}  
}
```

When a user clicks the `<button>`, we want to increase `#count`. We can do this with another method. We'll name this method `increaseCount`.

```
class Counter () {  
  #count  
  constructor (counter) { /* ... */ }  
  
  increaseCount () {  
    this.#count = this.#count + 1  
  }  
}
```

Next, we need to update the DOM with the new `#count`. Let's create a method called `updateCount` to do this. We will call `updateCount` from `increaseCount`:

```
class Counter () {  
  #count  
  constructor (counter) { /* ... */ }  
  
  increaseCount () {  
    this.#count = this.#count + 1  
    this.updateCount()  
  }  
  
  updateCount () {  
    this.countElement.textContent = this.#count  
  }  
}
```

We're ready to add the event listener now.

## ▷ (#adding-the-event-listener) Adding the event listener

We will add the event listener to the `this.buttonElement`. Unfortunately, we cannot use `increaseCount` as the callback straightaway. You'll get an error if you try it.

```
class Counter () {  
  // ...  
  
  constructor (counter) {  
    // ...
```




```


    this.buttonElement.addEventListener('click', this.increaseCount)
  }

  // Methods
}

```

 Error accessing #count because this doesn't point to the instance

You get an error because `this` points to `buttonElement`. (This is the event listener context.) You'll see the `buttonElement` if you logged `this` into the console.

 `this` points to the button element

We need to change the value of `this` back to the instance for `increaseCount` in order for things to work. There are two ways to do it:

1. Use `bind`
2. Use arrow functions

Most people use the first method (but the second one is easier).

## ↳ (#adding-the-event-listener-with-bind) Adding the event listener with `bind`

`bind` returns a new function. It lets you change `this` to the first argument that's passed. People normally create event listeners by calling `bind(this)`.

```

class Counter () {
  // ...

  constructor (counter) {
    // ...
    this.buttonElement.addEventListener('click', this.increaseCount.bind(this))
  }

  // ...
}

```

JavaScript

This works, but it's not very nice to read. It's also not beginner-friendly because `bind` is seen as an advanced JavaScript function.

## ↳ (#arrow-functions) Arrow functions

The second way is to use arrow functions. Arrow functions work because it preserves the `this` value to the lexical context.

Most people write methods inside the arrow function callback, like this:

```
class Counter () {  
  // ...  
  
  constructor (counter) {  
    // ...  
    this.buttonElement.addEventListener('click', _ => {  
      this.increaseCount()  
    })  
  }  
  
  // Methods  
}
```

This works, but it is a long way around. There's actually a shortcut.

You can create `increaseCount` with arrow functions. If you do this, the `this` value for `increaseCount` will be bound to the instance's value straightaway.

So here's the code you need:

```
class Counter () {  
  // ...  
  
  constructor (counter) {  
    // ...  
    this.buttonElement.addEventListener('click', this.increaseCount)  
  }  
  
  increaseCount = () => {  
    this.#count = this.#count + 1  
    this.updateCounter()  
  }  
  
  // ...  
}
```

## ↳ [\(#the-code\)](#) The code

Here's a complete version of the Class-based code (using arrow functions).

Embedded Pen Here

## ↳ **(#creating-the-counter-with-factory-functions) Creating the Counter with Factory functions**

We'll do the same thing here. We'll get users to pass the Counter's HTML into the Counter factory.

```
function Counter (counter) {  
  // ...  
}  
  
const counter = Counter(document.querySelector('.counter'))
```

JavaScript

We need to get two elements from `counter` — the `<span>` and the `<button>`. We can use normal variables (without `this`) here because they are private variables already. We won't expose them.

```
function Counter (counter) {  
  const countElement = counter.querySelector('span')  
  const buttonElement = counter.querySelector('button')  
}
```

JavaScript

We will initialize a count variable to the value that's present in the HTML.

```
function Counter (counter) {  
  const countElement = counter.querySelector('span')  
  const buttonElement = counter.querySelector('button')  
  
  let count = parseInt(countElement.textContent)  
}
```

JavaScript

We will increase this `count` variable with an `increaseCount` method. You can choose to use a normal function here, but I like to create a method to keep things neat and tidy.

```
function Counter (counter) {  
  // ...  
  const counter = {
```

JavaScript

```

    increaseCount () {
      count = count + 1
    }
  }
}

```

Finally, we will update the count with an `updateCount` method. We will also call `updateCount` from `increaseCount`.

```

function Counter (counter) {
  // ...
  const counter = {
    increaseCount () {
      count = count + 1
      counter.updateCount()
    }

    updateCount () {
      increaseCount()
    }
  }
}

```

Notice I used `counter.updateCount` instead of `this.updateCount`? I like this because `counter` is clearer compared to `this`. I also do this because beginners can also make a mistake with `this` inside Factory functions (which I'll cover later).

## ↳ (#adding-event-listeners) Adding event listeners

We can add event listeners to the `buttonElement`. When we do this, we can use `counter.increaseCount` as the callback straight away.

We can do this because we didn't use `this`, so it doesn't matter even if event listeners change the `this` value.

```

function Counter (counterElement) {
  // Variables

  // Methods
  const counter = { /* ... */ }

  // Event Listeners
  buttonElement.addEventListener('click', counter.increaseCount)
}

```

## ▷ (#the-this-gotcha) The this gotcha

You can use `this` in Factory functions. But you need to use `this` in a method context.

In the following example, if you call `counter.increaseCount`, JavaScript will also call `counter.updateCount`. This works because `this` points to the `counter` variable.

```
function Counter (counterElement) {  
  // Variables  
  
  // Methods  
  const counter = {  
    increaseCount() {  
      count = count + 1  
      this.updateCount()  
    }  
  }  
  
  // Event Listeners  
  buttonElement.addEventListener('click', counter.increaseCount)  
}
```

Unfortunately, the event listener wouldn't work because the `this` value was changed. You'll need the same treatment as Classes — with `bind` or arrow functions to — get the event listener working again.

And this leads me to the second gotcha.

## ▷ (#second-this-gotcha) Second this gotcha

If you use the Factory function syntax, you cannot create methods with arrow functions. This is because the methods are created in a simple function context (<https://zellwk.com/blog/this/#this-in-a-simple-function>).

```
function Counter (counterElement) {  
  // ...  
  const counter = {  
    // Do not do this.  
    // Doesn't work because `this` is `Window`  
    increaseCount: () => {  
      count = count + 1  
      this.updateCount()  
    }  
  }  
}
```

```
// ...  
}
```

So, I highly suggest skipping `this` entirely if you use Factory functions. It's much easier that way.

## ▷ [\(#the-code\)](#) **The code**

Embedded Pen Here

## ▷ [\(#verdict-for-event-listeners\)](#) **Verdict for event listeners**

Event listeners change the value of `this`, so we must be very careful about using the `this` value. If you use Classes, I recommend creating event listeners callbacks with arrow functions so you don't have to use `bind`.

If you use Factory functions, I recommend skipping `this` entirely because it may confuse you. That's it!

## ▷ [\(#conclusion\)](#) **Conclusion**

We talked about the four flavors of Object-Oriented Programming. They are:

1. Constructor functions
2. Classes
3. OLOO
4. Factory functions

First, we concluded that Classes and Factory functions are easier to use from a code-related point of view.

Second, we compared how to use Subclasses with Classes and Factory functions. Here, we see creating Subclasses is easier with Classes, but Composition is easier with Factory functions.

Third, we compared Encapsulation with Classes and Factory functions. Here, we see Encapsulation with Factory functions is natural — like JavaScript — while encapsulation with Classes requires you to add a `#` before variables.

Fourth, we compared the usage of `this` in Classes and Factory functions. I feel Factory functions win here because `this` can be ambiguous. Writing `this.#privateVariable` also creates longer code compared to using `privateVariable` itself.

Finally, in this article, we built a simple Counter with both Classes and Factory functions. You learned how to add event listeners to both Object-Oriented Programming programming flavors. Here, both flavors work. You just need to be careful whether you use `this` or not.

That's it!

I hope this shines some light on Object-Oriented Programming in JavaScript for you. If you liked this article, you may like my JavaScript course, [Learn JavaScript \(https://learnjavascript.today\)](https://learnjavascript.today), where I explain (almost) everything you need to know about JavaScript in a format as clear and succinct as this.

If you have any questions on JavaScript or front-end development in general, feel free to reach out to me (<https://zellwk.com/contact>). I'll see how I can help!